PARVATHA REDDY BABUL REDDY
VISVODAYA INSTITUTE OF TECHNOLOGY AND SCIENCE's

VITSPACE

For more Materials
☺ Click the link given below ☺

vitspace.netlify.com

Scroll to read your material

UNIT- V

***Transaction Management**: Transactions Concept, A Simple Transactional Model, Storage Structures, Transaction Atomicity and Durability, Transaction Isolation, Serializability, Isolation and Atomicity, Transaction Isolation Levels, Implementation of Isolation Levels, Transactions as SQL Statements.*

***Concurrency Control**: Lock based Protocols, Deadlock Handling, Multiple granularities, Timestamp based Protocols, Validation based Protocols.*

***Recovery System**: Failure Classification, Storage, Recovery and Atomicity, Recovery Algorithm, Buffer Management, Failure with Loss of Nonvolatile Storage, Early Lock Release and Logical Undo Operations.*

Transaction Management

A transaction is a logical unit of work that contains one or more SQL statements. A transaction is an atomic unit. The effects of all the SQL statements in a transaction can be either all committed (applied to the database) or all rolled back (undone from the database). . i.e Collections of operations that form a single logical unit of work are called **transactions**.

1. Transaction Concept

A transaction is a unit of program execution that accesses and possibly updates various data items. Usually, a transaction is initiated by a user program written in a high-level data-manipulation language or programming language (for example, SQL, COBOL, C, C++, or Java), where it is delimited by statements (or function calls) of the form **begin transaction** and **end transaction**. The transaction consists of all operations executed between the **begin transaction** and **end transaction**.

To ensure integrity of the data, we require that the database system maintain the following properties of the transactions known as **ACID** properties:

- **Atomicity**. Either all operations of the transaction are reflected properly in the database, or none are.
- **Consistency.** Execution of a transaction in isolation (that is, with no other transaction

executing concurrently) preserves the consistency of the database.

- **Isolation**. Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions Ti and Tj, it appears to Ti that either Tj finished execution before Ti started, or Tj started execution after Ti finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.
- **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

2. A Simple Transaction Model

Transactions access data using **two** operations:

- **read(X),** which transfers the data item X from the database to a local buffer belonging to the transaction that executed the read operation.
- **write(X),** which transfers the data item X from the local buffer of the transaction that executed the write back to the database.

Ex : Let Ti be a transaction that transfers $50 from account A to account B. This transaction can be defined as

```
Ti:   read(A);
      A := A − 50;
      write(A);
      read(B);
      B := B + 50;
      write(B).
```

Atomicity

Users should be able to regard the execution of each transaction as atomic in which either all actions of a transaction are carried out or none actions are carried out. That is we should not have to worry about the effect of incomplete transactions when a system crash occurs.

Transactions can be incomplete for <u>three</u> reasons as follows

1. A transaction can be aborted or terminated unsuccessfully by the DBMS because some changes arise during execution. If a transaction is aborted by the DBMS due to some internal reason, it is automatically restarted executed as a new transaction.
2. The system may crash because the power supply is interrupted, while one more transactions are in progress.

3. A transaction may encounter an unexpected situation (Ex, read an unexpected data value or be unable to access some disk) and decide to abort (i.e terminate itself).

Consistency

Execution of a transaction in isolation (i.e with no other transaction ececuting concurrently at the same time) preserves the consistency of the database.

Isolation

This property ensures that each transaction is unaware of other transaction ececuting concurrently in the system. Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions Ti and Tj, it appears that Ti is unaware of Tj i.e where Tj has started or finished or not and Tj is un aware of Ti i.e whether Tj has started or finished or not. Moreover, the DBMS provides no guarantee about which of these orders: whether; first Tj then Ti or whether first Ti then Tj is effectively chosen.

Durability

After a transaction completes successfully, the changes made to the database will be successful, even if the system failure occurs after that, the database remains safe.

The durability property guarantees that, once a transaction completes successfully,all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.

We can guarantee durability by ensuring the following TWO reasons

1.The updates carried out by the transaction have been written to disk before the transaction completes.

2.Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.

Ensuring durability is the responsibility of a component of the database system called the **recovery-management component**. The transaction-management component and the recovery-management component are closely related

3. Storage Structure

Storage media can be distinguished by their relative speed, capacity, and resilience to failure, and classified as **volatile storage** or **nonvolatile storage**. and another class of storage, called **stable storage.**

Volatile Memory

These are the primary memory devices in the system, and are placed along with the CPU. These memories can store only small amount of data, but they are very fast. E.g.:- main memory, cache memory etc. these memories cannot endure system crashes- data in these memories will be lost on failure.

Non-Volatile memory

These are secondary memories and are huge in size, but slow in processing. E.g.:- Flash memory, hard disk, magnetic tapes etc. these memories are designed to withstand system crashes.

Stable Memory

Information residing in stable storage is never lost. Copies of same non volatile memories are stored at different places. This is because, in case of any crash and data loss, data can be recovered from other copies. This is even helpful if there one of non-volatile memory is lost due to fire or flood. It can be recovered from other network location. But there can be failure while taking the backup of DB into different stable storage devices.

4. Transaction Atomicity and Durability or Transaction States

In the absence of failures, all transactions complete successfully. However, a transaction may not always complete its execution successfully. Such a transaction is termed **aborted**. If we are to ensure the atomicity property, an aborted transaction must have no effect on the state of the database. Thus, any changes that the aborted transaction made to the database must be undone. Once the changes caused by an aborted transaction have been undone, we say that the transaction has been **rolled back**. A transaction that completes its execution successfully is said to be **committed.**

A transaction must be in one of the following states:

- **Active**, the initial state; the transaction stays in this state while it is executing

• **Partially committed**, after the final statement has been executed

• **Failed**, after the discovery that normal execution can no longer proceed

• **Aborted**, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction

• **Committed**, after successful completion

The state diagram corresponding to a transaction appears in following Figure

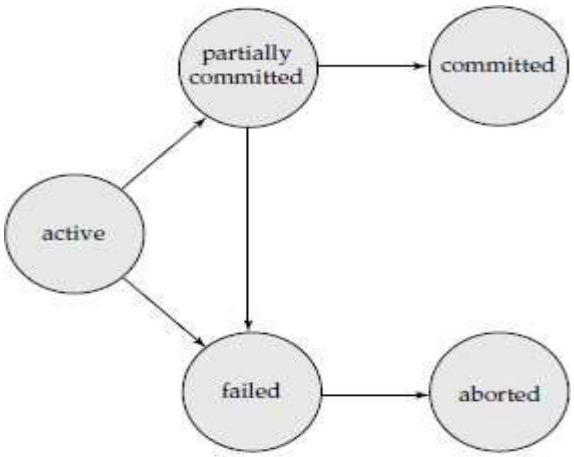


Figure State diagram of a transaction.

A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may stop its successful completion.

When the database system then writes out successfully the output information to disk, then the transaction enters the committed state.

A transaction may also enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (for example, because of hardware or logical errors). Such a transaction must be rolled back. Then, it enters the aborted state. At this point, the system has two options:

•It can **restart** the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.

• It can **kill** the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was

bad, or because the desired data were not found in the database.

5. Transaction Isolation

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data. Consistency in spite of concurrent execution of transactions requires extra work. it is far easier to insist that transactions run serially—that is, one at a time, each starting only after the previous one has completed. There are two good reasons for allowing concurrency:

- **Improved throughput and resource utilization**
- **Reduced waiting time**.

Improved throughput and resource utilization

- A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU.
- The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel.
- While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction.
- All of this increases the throughput of the system—that is, the number of transactions executed in a given amount of time.
- Correspondingly, the processor and disk utilization also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.

Reduced waiting time:

- There may be a mix of transactions running on a system, some short and some long.
- If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction.
- If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them.

- Concurrent execution reduces the unpredictable delays in running transactions.
- Moreover, it also reduces the average response time: the average time for a transaction to be completed after it has been submitted.

The idea behind using concurrent execution in a database is essentially the same as the idea behind using multi programming in an operating system. The database system must control the interaction among the concurrent transactions to prevent them from destroying the consistency of the database. It is achieved using **concurrency-control schemes**.

Concurrent Executions :

In **concurrent** schedule, CPU time is shared among two or more transactions in order to run them concurrently. However, this creates the possibility that more than one transaction may need to access a single data item for read/write purpose and the database could contain inconsistent value if such accesses are not handled properly. Let us explain with the help of an example.

Let us consider there are two transactions T1 and T2, that transfer funds one account to another. Transaction T1 transfers 50 from account A to account B. it is defined as whose instruction sets are given as following.

```
T1:   read(A);
      A := A − 50;
      write(A);
      read(B);
      B := B + 50;
      write(B).
```

Transaction T2 transfers 10 percent of the balance from account A to account B. It is defined as

```
T2:   read(A);
      temp := A * 0.1;
      A := A − temp;
      write(A);
      read(B);
      B := B + temp;

      write(B).
```

Suppose the current values of accounts A and B are $1000 and $2000, respectively. Suppose also that the two transactions are executed one at a time in the order T1 followed by T2. This execution sequence appears in following schedule. In the schedule the sequence of instruction

steps is in chronological order from top to bottom, with instructions of T1 appearing in the left column and instructions of T2 appearing in the right column. The final values of accounts A and B, after the execution in following schedule takes place, are $855 and $2145, respectively. Thus, the total amount of money in accounts *A* and *B*—that is, the sum *A* + *B*—is preserved after the execution of both transactions.

Similarly, if the transactions are executed one at a time in the order *T*2 followed by *T*1, then the corresponding execution sequence is that of Figure 15.4. Again, as expected, the sum *A* + *B* is preserved, and the final values of accounts *A* and *B* are $850 and $2150, respectively.

The execution sequences just described are called **schedules**

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

Schedule 2—a serial schedule in which T_2 is followed by T_1.

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

Schedule 3—a concurrent schedule equivalent to schedule 1.

5.6 Serializability

Serializability is the classical concurrency scheme. If the action of different transactions are not interleaved then it is called serializable schedule. In serial schedule transactions are executed from start to finish one by one without any interference of other transactions.

It ensures that a schedule for executing concurrent transactions is equivalent to one that executes the transactions serially in some order. It assumes that all accesses to the database are done using read and write operations.

A serrializable schedule over a set S of committed transactions is schedule whose effect on any consistent database is guaranteed to be identical to that of some complete serial over S. i.e even though the actions of transactions are interleaved, the result of executing transactions serially in different orders may produce different results.

Serial Schedule	
T1	T2
R1(A)	
W1(A)	
R1(B)	
R1(B)	
C1	
	R2(A)
	W2(A)
	R2(B)
	W2(B)
	C2

schedule-1

Interleaved schedule	
T1	T2
R1(A)	
W1(A)	
	R2(A)
	W2(A)
R1(B)	
R1(B)	
C1	
	R2(B)
	W2(B)
	C2

schedule-2

Anomalies due to interleaved execution

There are three main situations when the actions of two transactions T_1 and T_2 conflict with each other in the interleaved execution on the same data object.

The three anomalies associated with interleaved execution are as follows

1. Write Read (WR) conflict : Reading uncommitted data
2. Read-Write (RW) conflict : Unrepeatable Read
3. Write-Write (WW) conflict : Over writing Uncommitted Data

Two major types of serializability exist:

1. ***conflict-serializability*, and**
2. ***view-serializability***

1. Conflict Serializability

Two instructions of two different transactions may want to access the same data item in order to perform a read/write operation. Conflict Serializability deals with detecting whether the instructions are conflicting in any way, and specifying the order in which these two instructions will be executed in case there is any conflict. A **conflict** arises if at least one (or both) of the instructions is a write operation. The following rules are important in Conflict Serializability:

1. If two instructions of the two concurrent transactions are both for **read operation**, then they are **not in conflict**, and can be allowed to take place in any order.
2. If one of the instructions wants to perform a **read operation** and the other instruction wants to perform a **write operation**, then they are in **conflict**, hence their ordering is important. If the read instruction is performed first, then it reads the old value of the data item and after the reading is over, the new value of the data item is written. If the write instruction is performed first, then updates the data item with the new value and the read instruction reads the newly updated value.
3. If both the transactions are for **write operation**, then they are in **conflict** but can be allowed to take place in any order, because the transaction do not read the value updated by each other. However, the value that persists in the data item after the schedule is over is the one written by the instruction that performed the last write.

It may happen that we may want to execute the same set of transaction in a different schedule on another day. Keeping in mind these rules, we may sometimes alter parts of one schedule (S1) to create another schedule (S2) by swapping only the non-conflicting parts of the first schedule. The conflicting parts cannot be swapped in this way because the ordering of the conflicting instructions is important and cannot be changed in any other schedule that is derived from the first. If these two schedules are made of the same set of transactions, then both S1 and S2 would yield the same result if the conflict resolution rules are maintained while creating the new schedule. In that case the schedule S1 and S2 would be called **Conflict Equivalent**.

Conflict Equivalent : Two schedules would be conflicting if they have the following properties

- Both accesses the same data item.
- At least one of them is "write" operation.

Two schedules having multiple transactions with conflicting operations are said to be conflict equivalent if and only if

- Both the schedules contain the same set of Transactions.
- The order of conflicting pairs of operation is maintained in both the schedules.

View Equivalence

Two schedules would be view equivalence if the transactions in both the schedules perform similar actions in a similar manner.

For example –

- If T reads the initial data in S1, then it also reads the initial data in S2.
- If T reads the value written by J in S1, then it also reads the value written by J in S2.
- If T performs the final write on the data value in S1, then it also performs the final write on the data value in S2.

View Serializability:

This is another type of serializability that can be derived by creating another schedule out of an existing schedule, involving the same set of transactions. These two schedules would be

called View Serializable, if the following rules are followed while creating the second schedule out of the first.

A Schedule is view serializable if it is view equivalent to any serial schedule

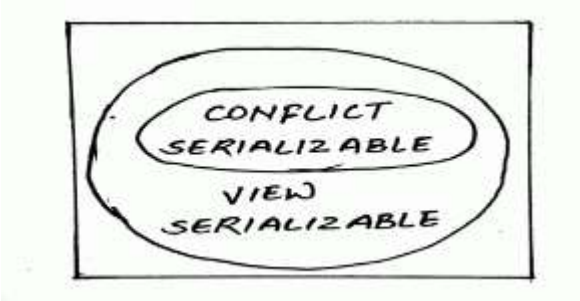
Let us consider that the transactions T1 and T2 are being serialized to create two different schedules S1 and S2 which we want to be **View Equivalent** and both T1 and T2 wants to access the same data item.

1. If in S1, T1 reads the initial value of the data item, then in S2 also, T1 should read the initial value of that same data item.
2. If in S1, T1 writes a value in the data item which is read by T2, then in S2 also, T1 should write the value in the data item before T2 reads it.
3. If in S1, T1 performs the final write operation on that data item, then in S2 also, T1 should perform the final write operation on that data item.

Except in these three cases, any alteration can be possible while creating S2 by modifying S1.

Note − View equivalent schedules are view serializable and conflict equivalent schedules are conflict serializable. All conflict serializable schedules are view serializable too.

1. Conflict serializability is easy to achieve but view serializability is difficult to achieve
2. Every conflict serializable is view serializable but the reverse is not true.
3. It is easy to test conflict serializability but expensive to test view serializability.
4. Most of the concurrency control schemes used in practice are based on conflict serializability.



5.7 Transaction Isolation and Atomicity (Recoverability)

Data recoverability is the process of restoring data that has been lost, accidentally deleted, corrupted or made inaccessible for any reason.

If a transaction Ti fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction. In a system that allows concurrent execution, it is necessary also to ensure that any transaction Tj that is dependent on Ti (that is, Tj has read data written by Ti) is also aborted. To achieve this surety, we need to place restrictions on the type of schedules permitted in the system. The following two subsections, address the issue of what schedules are acceptable from the viewpoint of recovery from transaction failure.

Recoverability Schedules

A recoverable schedule is one where, for each pair of Transaction T_i and T_j such that T_j reads data item previously written by T_i the commit operation of T_i appears before the commit operation T_j .

T8	T9		
read(A)			T9 is dependent on T8
write(A)			Non recoverable schedule if T9 commits before T8
	read(A)		
read(B)			

Suppose that the system allows T9 to commit immediately after execution of read(A) instruction.Thus T9 commit before T8 does.

Now suppose that T8 fails before it commits. Since T9 has read the value of data item A written by T8 we must abort T9 to ensure transaction Atomicity.

However, T9 has already committed and cannot be aborted.Thus we have a situation where it is impossible to recover correctly from the failure of T8.

Cascadeless schedules

T10	T11	T12
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

Transaction T10 writes a value of A that is read by Transaction T11. Transaction T11 writes a value of A that is read by Transaction T12. Suppose at this point T10 fails. T10 must be rolled back, since T11 is dependent on T10, T11 must be rolled back, T12 is dependent on T11, T12 must be rolled back.

This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks is called **Cascading rollback**.

- Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work.
- It is desirable to restrict the schedules to those where cascading rollbacks cannot occur, Such schedules are called Cascadeless Schedules.
- Formally, a cascadeless schedule is one where for each pair of transaction T_i and T_j such that T_j reads data item, previously written by T_i the commit operation of T_i appears before the read operation of T_j.

Every Cascadeless schedule is also recoverable schedule.

8. Transaction Isolation Levels

The SQL standard defines **four** isolation levels

1. **Read Uncommitted –** Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transactions, thereby allowing dirty reads. At this level, transactions are not isolated from each other.
2. **Read Committed –** This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allow dirty read. The transaction holds a

read or write lock on the current row, and thus prevents other transactions from reading, updating, or deleting it.

3. **Repeatable Read** – This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on referenced rows for update and delete actions. Since other transactions cannot read, update or delete these rows, consequently it avoids non-repeatable read.
4. **Serializable** – This is the highest isolation level. A *serializable* execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

5.9 Concurrency Control

Lock - Based Protocols

Locks : Lock is a small variable, book keeping object which is association with each data item. Manipulation of the value of lock is called **locking**. The value of a lock variable is used in the locking scheme to control the concurrent access and manipulation of the associated data item. Locking the items being used by transaction can prevent other concurrently running transaction from using these locked items. The locking is done by a subsystem of the database management system, usually called the lock manager.

Locks are used as a means of synchronizing the access by concurrent transactions to the database item.

There are mainly **two** modes in which a data item may be locked.

1.Shared. It is also called as read lock If a transaction Ti has obtained a shared-mode lock (denoted by S) on item Q, then Ti can read, but cannot write, Q.

2.Exclusive. This lock is also called an update or write lock. If a transaction Ti has obtained an exclusive-mode lock (denoted by X) on item Q, then Ti can both read and write Q.

	S	X
S	true	false
X	false	false

Lock-compatibility matrix comp.

In Locks shared mode is compatible with shared mode, but not with exclusive mode. At any time, several shared-mode locks can be held simultaneously (by different transactions) on a particular data item. A subsequent exclusive-mode lock request has to wait until the currently held shared-mode locks are released. A transaction requests a **shared lock** on data item Q by executing the **lock-S(Q)** instruction. Similarly, a transaction requests an **exclusive lock** through the **lock-X(Q)** instruction. A transaction can **unlock** a data item Q by the **unlock(Q)** instruction. To access a data item, transaction Ti must first lock that item. If the data item is already locked by another transaction in an incompatible mode, the concurrency control manager will not grant the lock until all incompatible locks held by other transactions have been released. Thus, Ti is made to **wait** until all incompatible locks held by other transactions have been released.

```
T1: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    unlock(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(A).
```

Transaction T_1.

```
T2: lock-S(A);
    read(A);
    unlock(A);
    lock-S(B);
    read(B);
    unlock(B);
    display(A + B).
```

Transaction T_2.

Granting of Locks

When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted. Suppose a transaction *T*2 has a shared-mode lock on a data item, and another transaction *T*1 requests an exclusive-mode lock on the data item. Clearly, *T*1 has to wait for *T*2 to release the

shared-mode lock. Meanwhile, a transaction *T*3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to *T*2, so *T*3 may be granted the shared-mode lock. At this point *T*2 may release the lock, but still *T*1 has to wait for *T*3 to finish. But again, there may be a new transaction *T*4 that requests a shared-mode lock on the same data item, and is granted the lock before *T*3 releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but *T*1 never gets the exclusive-mode lock on the data item. The transaction *T*1 may never make progress, and is said to be **starved**. We can avoid starvation of transactions by granting locks in the following manner:

When a transaction *Ti* requests a lock on a data item *Q* in a particular mode *M*, the concurrency-control manager grants the lock provided that

1. There is no other transaction holding a lock on *Q* in a mode that conflicts with *M*.
2. There is no other transaction that is waiting for a lock on *Q*, and that made its lock request before *Ti*.

The Two-Phase Locking Protocol

The two-phase locking protocol ensures serializability. This protocol requires that each transaction issue lock and unlock requests in **two** phases:

1. **Growing phase**. A transaction may obtain locks, but may not release any lock.
2. **Shrinking phase.** A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests. Thus, the transaction acquires lock and releases lock as needed. A modification of two-phase locking is called the Strict Two-Phase Locking Protocols.

Strict Two-Phase Locking Protocol

The Two-Phase Locking Protocol, requires that in addition to locking being two-phase, all exclusive-mode locks(Exclusive lock is the mode of locking to provide exclusive use of the data item to only and only transaction) taken by a transaction, must be held until that transaction commits successfully, preventing any other transaction from reading the data.

Thus, the most widely used locking protocol is the Strict Two-Phase Locking Protocol, also called as Strict 2PL, has following two rules.

Rule 1 : If a transaction T wants to read an object, it first requests a Shared lock. But, if a transaction T wants to modify an object, it first requests an exclusive lock on the object.

Rule 2 : All locks held by a transaction are released when the transaction is completed.

The locking protocol can have two cases as follows :

1. If two transactions access completely different parts of the database, then they proceed without interruption on their ways.
2. If two transactions access same object of the database, then their actions are ordered serially i.e., all actions of locked transactions are completed first, then this lock is releases and the other transaction can now proceed.

Thus, the locking protocol only helps the second case. i.e., interleaving of transactions.

Moreover, the shared lock on database A is denoted as S(A) and the exclusive lock on database object A is denoted as X(A).

Ex : Suppose that both transactions T_1 and T_2 read the same data objects A and B perform the following operations.

i) T_1 deducts $100 from account A

ii) T_2 reads accounts of A and B and adds 6% interest to each

iii) T_1 adds $100 to account B

This example, may use interleaving of transactions and produce incorrect results. But, if the Strict 2PL protocol is used, such interleaving is disallowed and performs both transactions as follows.

First T_1 would obtain an exclusive lock on A and then read and write A. Then T_2 would request a lock A, but this request cannot be granted until T_1 releases its exclusive lock on A, and the DBMS therefore suspends T_2 as follows

T_1	T_2
X(A)	
R(A)	
A:=A-100	
W(A)	

Now T_1 proceeds to obtain an exclusive lock on B reads and writes B, then finally commits and the Locks on A and B are released. Now, T_2's lock request is granted and it proceeds. As a result the Strict 2PL protocol results in a serial execution of the two transactions as follows.

The following schedule illustrates Strict 2PL with serial execution

T_1	T_2
X(A)	
R(A)	
A:=A-100	
W(A)	
X(B)	
B:=B+100	
W(B)	
Commit;	
	X(A)
	R(A)
	A:=A+0.06A
	W(A)
	X(B)
	B:=B+0.06B
	W(B)
	Commit;

T_1	T_2
S(A)	
R(A)	
	S(A)
	R(A)
	X(A)

Moreover, the following schedule illustrates Strict 2PL with interleaved action, where shared lock on A can only read A , But cannot Write A and both T_1 and T_2 can hold shared lock at the same time where as an exclusive lock on B and C can both react and write the respective B and C but an exclusive lock cannot be held at the same time, instead an exclusive lock must holds each data item serially one after the other.

	R(B)
	W(B)
	Commit
X(C)	
R(C)	
W(C)	
commit	

Implementation of Locking

A **lock manager** can be implemented as a process that receives messages from transactions and sends messages in reply. The lock-manager process replies to lock-request messages with lock-grant messages, or with messages requesting rollback of the transaction Unlock messages require only an acknowledgment in response, but may result in a grant message to another waiting transaction.

The lock manager uses this data structure: For each data item that is currently locked, it maintains a linked list of records, one for each request, in the order in which the requests arrived. It uses a hash table, indexed on the name of a data item, to find the linked list (if any) for a data item; this table is called the **lock table**. Each record of the linked list for a data item notes which transaction made the request, and what lock mode it requested. The record also notes if the request has currently been granted.

The lock manager processes requests this way:

- When a lock request message arrives, it adds a record to the end of the linked list for the data item, if the linked list is present. Otherwise it creates a new linked list, containing only the record for the request.

It always grants the first lock request on a data item. But if the transaction requests a lock on an item on which a lock has already been granted, the lock manager grants the request only if it is compatible with all earlier requests, and all earlier requests have been granted already.

Otherwise the request has to wait.

•When the lock manager receives an unlock message from a transaction, it deletes the record for that data item in the linked list corresponding to that transaction. It tests the record that

follows, if any, as described in the previous paragraph, to see if that request can now be granted. If it can, the lock manager grants that request, and processes the record following it, if any, similarly, and so on.

• If a transaction aborts, the lock manager deletes any waiting request made by the transaction. Once the database system has taken appropriate actions to undo the transaction), it releases all locks held by the aborted transaction.

Locking Protocol Properties

i) Conflict Equivalent

ii) Conflict Serializable

iii) Serializabilty / Precedence Graph

Timestamp-Based Protocols

The most commonly used concurrency protocol is the timestamp based protocol. This protocol uses either system time or logical counter as a timestamp.

Lock-based protocols manage the order between the conflicting pairs among transactions at the time of execution, whereas timestamp-based protocols start working as soon as a transaction is created.

Timestamps

Each transaction Ti in the system associated with a unique fixed timestamp, denoted by TS(Ti). This timestamp is assigned by the database system before the transaction Ti starts execution. If a transaction Ti has been assigned timestamp TS(Ti), and a new transaction Tj enters the system, then TS(Ti) < TS(Tj). There are two simple methods for implementing this scheme:

1. Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.
2. Use a logical counter that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

The timestamps of the transactions determine the serializability order. Thus, if TS(Ti) < TS(Tj), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction Ti appears before transaction Tj .

To implement this scheme, we associate with each data item Q two timestamp values:

- **W-timestamp(Q**) denotes the largest timestamp of any transaction that executed write(Q) successfully.
- **R-timestamp(Q)** denotes the largest timestamp of any transaction that executed read(Q) successfully.

These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed.

The Timestamp-Ordering Protocol

The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

Timestamp ordering protocol works as follows −

- **If a transaction Ti issues a read(X) operation −**
  - If TS(Ti) < W-timestamp(X)
    - Operation rejected.
  - If TS(Ti) >= W-timestamp(X)
    - Operation executed.
  - All data-item timestamps updated.
- **If a transaction Ti issues a write(X) operation −**
  - If TS(Ti) < R-timestamp(X)
    - Operation rejected.
  - If TS(Ti) < W-timestamp(X)
    - Operation rejected and Ti rolled back.
  - Otherwise, operation executed.

Thomas' Write Rule : It does not enforce conflict serializability; but it rejects fewer write operations, by modifying the checks for the write_item(X) operation as follows:

This rule states if TS(Ti) < W-timestamp(X), then the operation is rejected and T_i is rolled back.

Time-stamp ordering rules can be modified to make the schedule view serializable. Instead of making T_i rolled back, the 'write' operation itself is ignored.

Validation-Based Protocols

Each transaction T_i executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order,

1.Read phase. During this phase, the system executes transaction T_i. It reads the values of the various data items and stores them in variables local to T_i. It performs all write operations on temporary local variables, without updates of the actual database.

2.Validation phase. Transaction T_i performs a validation test to determine whether it can copy to the database the temporary local variables that hold the results of write operations without causing a violation of serializability.

3.Write phase. If transaction T_i succeeds in validation (step 2), then the system applies the actual updates to the database. Otherwise, the system rolls back T_i.

The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.

Each transaction Ti has 3 timestamps

- **Start(Ti)**: the time when Ti started its execution
- **Validation(Ti):** the time when Ti entered its validation phase
- **Finish(Ti):** the time when Ti finished its write phase

Serializability order is determined by timestamp given at validation time, to increase concurrency. Thus TS(Ti) is given the value of **Validation(Ti).**

This protocol is useful and gives greater degree of concurrency if probability of conflicts is low. That is because the serializability order is not pre-decided and relatively less transactions will have to be rolled back.

The **validation test** for transaction Tj requires that, for all transactions Ti with TS(Ti) < TS(Tj), one of the following two conditions must hold:

1.Finish(Ti) < Start(Tj). Since Ti completes its execution before Tj started, the serializability order is indeed maintained.

2.start(Tj) < **finish**(Ti) < **validation**(Tj) **and** the set of data items written by Ti does not intersect with the set of data items read by Tj . then validation succeeds and Tj can be committed. Otherwise, validation fails and Tj is aborted.

This validation scheme is called the **optimistic concurrency control** scheme since transactions execute optimistically, assuming they will be able to finish execution and validate at the end.

5.10 Multiple Granularity

Multiple Granularity is the hierarchically breaking up the database into portions which are lockable and maintaining the track of what to be lock and how much to be lock so that it can be decided very quickly either to lock a data item or to unlock a data item.

Suppose a database is divided into files; files are divided into pages; pages are divided into records.

If there is a need to lock a record, then a transaction can easily lock it. But if there is a need to lock a file, the transaction have to lock firstly all the records one after another, then pages in that file and finally the file. So, there is a need to provide a mechanism for locking the files also which is provided by **multiple granularity**.

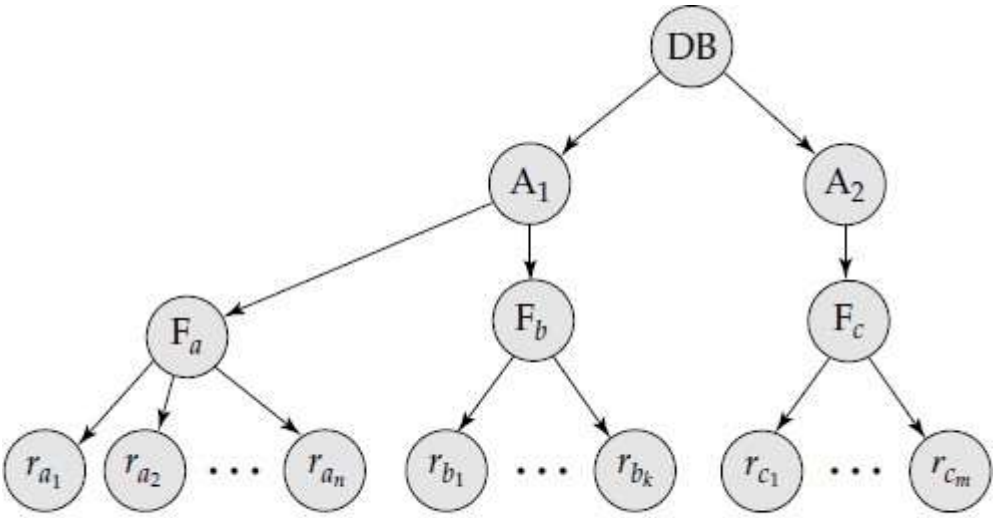


Figure Granularity hierarchy.

Each node in the tree can be locked individually. As we did in the two-phase locking protocol, we shall use **shared** and **exclusive** lock modes. When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in the same lock mode. For example, if transaction *Ti* gets an **explicit lock** on file *Fc* of above Figure, in exclusive mode, then it has an **implicit lock** in exclusive mode all the records belonging to that file. It does not need to lock the individual records of *Fc* explicitly.

A more efficient way to gain this knowledge is to introduce a new class of lock modes, called **inten- tion lock modes**. If a node is locked in an intention mode, explicit locking is being done at a lower level of the tree (that is, at a finer granularity). Intention locks are put on all the ancestors of a node before that node is locked explicitly. Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully.

There is an intention mode associated with shared mode, and there is one with exclusive mode. If a node is locked in **intention-shared (IS) mode**, explicit locking is being done at a lower level of the tree, but with only shared-mode locks. Similarly, if a node is locked in **intention-exclusive (IX) mode**, then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks. Finally, if a node is locked in **shared and intention-exclusive (SIX) mode**, the sub tree rooted by that node is locked explicitly in shared mode, and

that explicit locking is being done at a lower level with exclusive-mode locks. The compatibility function for these lock modes is in compatibility matrix.

The **multiple-granularity locking protocol**, which ensures serializability, in this:

Each transaction *Ti* can lock a node *Q* by following these rules:

1. It must observe the lock-compatibility function.

2. It must lock the root of the tree first, and can lock it in any mode.

3.It can lock a node *Q* in S or IS mode only if it currently has the parent of *Q* locked in either IX or IS mode.

4.It can lock a node *Q* in X, SIX, or IX mode only if it currently has the parent of *Q* locked in either IX or SIX mode.

Observe that the multiple-granularity protocol requires that locks be acquired in ***top- down*** **(root-to-leaf)** order, whereas locks must be released in ***bottom-up*** **(leaf-to-root)** order.

This protocol enhances concurrency and reduces lock overhead. It is particularly useful in applications that include a mix of

• Short transactions that access only a few data items

• Long transactions that produce reports from an entire file or set of files

Recovery System

A computer system, like any other device, is subject to failure from a variety of causes: disk crash, power outage, software error, a fire in the machine room, even sabotage. In any failure, information may be lost. Therefore, the database system must take actions in advance to ensure that the atomicity and durability properties of transactions, If it fails or crashes amid transactions, it is expected that the system would follow some sort of algorithm or techniques to recover lost data. An integral part of a database system is a **recovery scheme** that can restore the database to the consistent state that existed before the failure. The recovery scheme must also provide **high availability**; that is, it must minimize the time for which the database is not usable after a crash.

5.11 Failure Classification

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner. The simplest type of failure is one that does not result in the loss of information in the system. The failures that are more difficult to deal with are those that result in loss of information. Failures are classified into the following types :

- **Transaction failure**. There are two types of errors that may cause a transaction to fail:
  - **Logical error**. The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.
  - **System error**. The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be re executed at a later time.
- **System crash**. There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage, and brings transaction processing to a halt. The content of nonvolatile storage remains intact, and is not corrupted.

  The assumption that hardware errors and bugs in the software bring the system to a halt, but do not corrupt the nonvolatile storage contents, is known as the **fail-stop assumption**.

Well-designed systems have numerous internal checks, at the hardware and the software level, that bring the system to a halt when there is an error. Hence, the fail-stop assumption is a reasonable one.

- **Disk failure :** A disk block loses its content as a result of either a head crash or failure during a data transfer operation. Copies of the data on other disks, or archival backups on tertiary media, such as tapes, are used to recover from the failure.

To determine how the system should recover from failures, we need to identify the failure modes of those devices used for storing data. The following **recovery algorithms** to ensure database consistency and transaction atomicity despite failures. It have two parts:

1. Actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures.
2. Actions taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity, and durability.

5.12 Storage Structure

The storage structure can be divided into following categories :

- **Volatile storage**. Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main memory and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself, and because it is possible to access any data item in volatile storage directly.
- **Nonvolatile storage**. Information residing in nonvolatile storage survives system crashes. Examples of such storage are disk and magnetic tapes. Disks are used for online storage, whereas tapes are used for archival storage. Nonvolatile storage is slower than volatile storage by several orders of magnitude. In database systems, disks are used for most nonvolatile storage. Other nonvolatile media are normally used only for backup data.
- **Stable storage**. Information residing in stable storage is *never* lost . Stable storage is theoretically impossible to obtain, it can be closely approximated by techniques that make data loss extremely unlikely.

Stable-Storage Implementation

To implement stable storage, we need to replicate the needed information in several nonvolatile storage media (usually disk) with independent failure modes, and to update the information in a controlled manner to ensure that failure during data transfer does not damage the needed information.

RAID systems guarantee that the failure of a single disk will not result in loss of data. The simplest and fastest form of RAID is the mirrored disk, which keeps two copies of each block, on separate disks. Other forms of RAID offer lower costs, but at the expense of lower performance. RAID systems, however, cannot guard against data loss due to disasters such as fires or flooding. Many systems store archival backups of tapes off-site to guard against such disasters.

Storage media can be protected from failure during data transfer. Block transfer between memory and disk storage can result in

• **Successful completion**. The transferred information arrived safely at its destination.

•**Partial failure**. A failure occurred in the midst of transfer, and the destination block has incorrect information.

•**Total failure**. The failure occurred sufficiently early during the transfer that the destination block remains intact.

If a **data-transfer failure** occurs, the system detects it and invokes a recovery procedure to restore the block to a consistent state. To do so, the system must maintain two physical blocks for each logical database block; in the case of mirrored disks, both blocks are at the same location; in the case of remote backup, one of the blocks is local, whereas the other is at a remote site. An output operation is executed as follows:

1. Write the information onto the first physical block.

2.When the first write completes successfully, write the same information onto the second physical block.

3. The output is completed only after the second write completes successfully.

During recovery, the system examines each pair of physical blocks. If both are the same and no detectable error exists, then no further actions are necessary. If the system detects an error in one block, then it replaces its content with the content of the other block. If both blocks contain no detectable error, but they differ in content, then the system replaces the content of the first block

with the value of the second. This recovery procedure ensures that a write to stable storage either succeeds completely or results in no change.

Data Access

The database system resides permanently on nonvolatile storage, and is partitioned into fixed-length storage units called **blocks**. Blocks are the units of data transfer **to and from** disk, and may contain several data items.

Transactions **input** information from **the disk to main memory**, and then **output** the **information back onto the disk**. The input and output operations are done in block units. The blocks residing on the disk are referred to as **physical blocks**; the blocks residing temporarily in main memory are referred to as **buffer blocks**. The area of memory where blocks reside temporarily is called the **disk buffer**.

Block movements between disk and main memory are initiated through the following **two** operations:

1. **input(*B*)** transfers the physical block *B* to main memory.
2. **output(*B*)** transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.

The following figure illustrates this scheme.

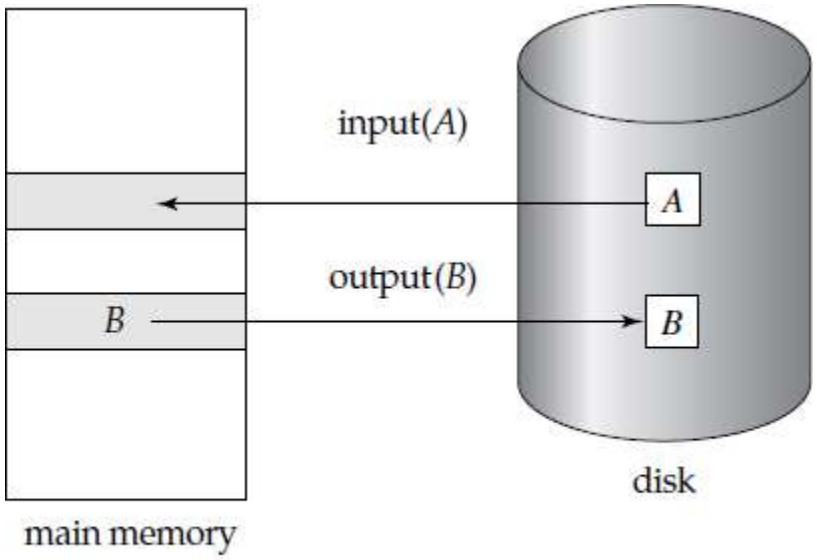


Figure Block storage operations.

Data can be transferred by these **two** operations:

1. **read(*X*)** assigns the value of data item *X* to the local variable *xi*. It executes this

operation as follows:

a. If block B_X on which X resides is not in main memory, it issues input(B_X).

b. It assigns to xi the value of X from the buffer block.

1. **write(X)** assigns the value of local variable xi to data item X in the buffer block. It executes this operation as follows:

a. If block B_X on which X resides is not in main memory, it issues input(B_X).

b. It assigns the value of xi to X in buffer BX.

5.13 Recovery and Atomicity

When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.

When a DBMS recovers from a crash, it should maintain the following −

- It should check the states of all the transactions, which were being executed.
- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
- It should check whether the transaction can be completed now or it needs to be rolled back.
- No transactions would be allowed to leave the DBMS in an inconsistent state.

There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction −

- Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
- Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

Log-Based Recovery

The log is a sequence of **log records**, which maintains the records of actions performed by a transaction. There are several types of log records. An **update log record** describes a single database write. It has these fields:

- **Transaction identifier** is the unique identifier of the transaction that performed the write operation.
- **Data-item identifier** is the unique identifier of the data item written. Typically, it is the location on disk of the data item.
- **Old value** is the value of the data item prior to the write.
- **New value** is the value that the data item will have after the write.

Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction.

We denote the various types of **log records** as:

- **<*Ti* start>.** Transaction *Ti* has started.
- **<*Ti, Xj, V*1, *V*2>.** Transaction *Ti* has performed a write on data item *Xj* . *Xj* had value *V*1 before the write, and will have value *V*2 after the write.
- **<*Ti* commit>.** Transaction *Ti* has committed.
- **<*Ti* abort>.** Transaction *Ti* has aborted.

The database can be modified using following approaches −

Deferred Database Modification

The **deferred-modification technique** ensures transaction atomicity by recording all database modifications in the log, but deferring the execution of all **write** operations of a transaction after partially commits. .ie All logs are written on to the stable storage and the database is updated when a transaction commits.

The version of the deferred-modification technique that we describe in this section assumes that transactions are executed serially.

When a transaction partially commits, the information on the log associated with the transaction is used in executing the deferred writes. If the system crashes before the transaction

completes its execution, or if the transaction aborts, then the information on the log is simply ignored.

The execution of transaction *Ti* proceeds as follows. Before *Ti* starts its execution, a record <*Ti* start> is written to the log. A write(*X*) operation by *Ti* results in the writing of a new record to the log. Finally, when *Ti* partially commits, a record <*Ti* commit> is written to the log.

Example : Let T_0 be a transaction that transfers $50 from account A to account B:

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

Let T1 be a transaction that withdraws $100 from account C:

T_1: read(C);
$C := C - 100$;
write(C).

Suppose that these transactions are executed serially, in the order T0 followed by T1, and that the values of accounts A, B, and C before the execution took place were $1000, $2000, and $700, respectively. The portion of the log containing the relevant information on these two transactions appears in following schedule

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

Portion of the database log corresponding to T_0 and T_1.

The recovery scheme uses the following recovery procedure:

- **redo(Ti)** sets the value of all data items updated by transaction Ti to the new values.

The redo operation must be **idempotent**; that is, executing it several times must be equivalent to executing it once. This characteristic is required if we are to guarantee correct behavior even if a failure occurs during the recovery process.

After a failure, the recovery subsystem consults the log to determine which transactions need to be redone. **Transaction *Ti* needs to be redone if and only if the log contains both the record <*Ti* start> and the record <*Ti* commit>.**

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

State of the log and database corresponding to T_0 and T_1.

(a)	(b)	(c)
$<T_0$ start>	$<T_0$ start>	$<T_0$ start>
$<T_0, A, 950>$	$<T_0, A, 950>$	$<T_0, A, 950>$
$<T_0, B, 2050>$	$<T_0, B, 2050>$	$<T_0, B, 2050>$
	$<T_0$ commit>	$<T_0$ commit>
	$<T_1$ start>	$<T_1$ start>
	$<T_1, C, 600>$	$<T_1, C, 600>$
		$<T_1$ commit>

Schedules : the same same log as above schedule shown at three different times

If log on stable storage at time of crash is as in case:

(a) No redo actions need to be taken

` (b) redo(T_0) must be performed since <T_0 commit> is present

(c)redo(T_0) must be performed followed by redo(T_1) since <T_0commit> and <Ticommit> are present

Immediate Database Modification

The **immediate-modification technique** allows database modifications to be output to the database while the transaction is still in the active state. Data modifications written by active transactions are called **uncommitted modifications.** That is, the database is modified immediately after every operation. In the event of a crash or a transaction failure, the system must use the old-value field of the log records.

Before a transaction Ti starts its execution, the system writes the record <Ti start> to the log. During its execution, any write(X) operation by Ti is preceded by the writing of the appropriate new update record to the log. When Ti partially commits, the system writes the record <Ti commit> to the log.

Using the log, the system can handle any failure that does not result in the loss of information in nonvolatile storage. The recovery scheme uses two recovery procedures:

• **undo(*Ti*)** restores the value of all data items updated by transaction *Ti* to the old values.

• **redo(*Ti*)** sets the value of all data items updated by transaction *Ti* to the new values.

The set of data items updated by *Ti* and their respective old and new values can be found in the log. The **undo and redo** operations must be idempotent to guarantee correct behavior even if a failure occurs during the recovery process. After a failure has occurred, the recovery scheme consults the log to determine which transactions need to be redone, and which need to be undone:

- Transaction *Ti* needs to be undone if the log contains the record <*Ti* start>, but does not contain the record <*Ti* commit>.
- Transaction *Ti* needs to be redone if the log contains both the record <*Ti* start> and the record <*Ti* commit>.

Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000.

(b) undo (T_1) and redo (T_0): C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to determine this information. There are two major difficulties with this approach:

1. The search process is time consuming.
2. Most of the transactions that, according to our algorithm, need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will nevertheless cause recovery to take longer.

To reduce these types of overhead, **checkpoints** are introduced.

Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

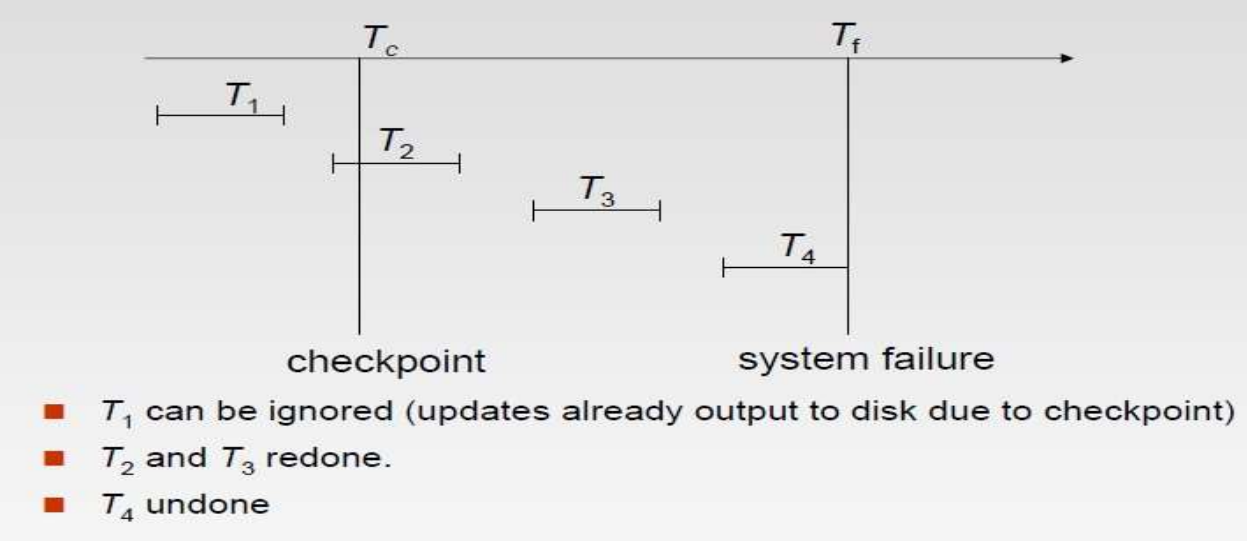
The system periodically performs **checkpoints**, which require the following sequence of actions to take place:

1. Output onto stable storage all log records currently residing in main memory.
2. Output to the disk all modified buffer blocks.
3. Output onto stable storage a log record <checkpoint>.

When a system with concurrent transactions crashes and recovers, it behaves in the following manner −

- The recovery system reads the logs backwards from the end to the last checkpoint.
- It maintains two lists, an undo-list and a redo-list.
- If the recovery system sees a log with <T_n, Start> and <T_n, Commit> or just <T_n, Commit>, it puts the transaction in the redo-list.
- If the recovery system sees a log with <T_n, Start> but no commit or abort log found, it puts the transaction in undo-list.

All the transactions in the undo-list are then undone and their logs are removed. All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.



Shadow Paging

An alternative to log-based crash-recovery techniques is **shadow paging**. shadow paging may require fewer disk accesses than do the log-based methods the database is partitioned into some number of fixed-length blocks, which are referred to as **pages**. The term *page* is borrowed from operating systems, since we are using a paging scheme for memory management.

Assume that there are *n* pages, numbered 1 through *n*. These pages do not need to be stored in any particular order on disk. However, there must be a way to find the *i*th page of the database for any given *i*. We use a **page table**, as illustrated in following figure.

The page table has *n* entries—one for each database page. Each entry contains a pointer to a page on disk. The first entry contains a pointer to the first page of the database, the second entry points to the second page, and so on.

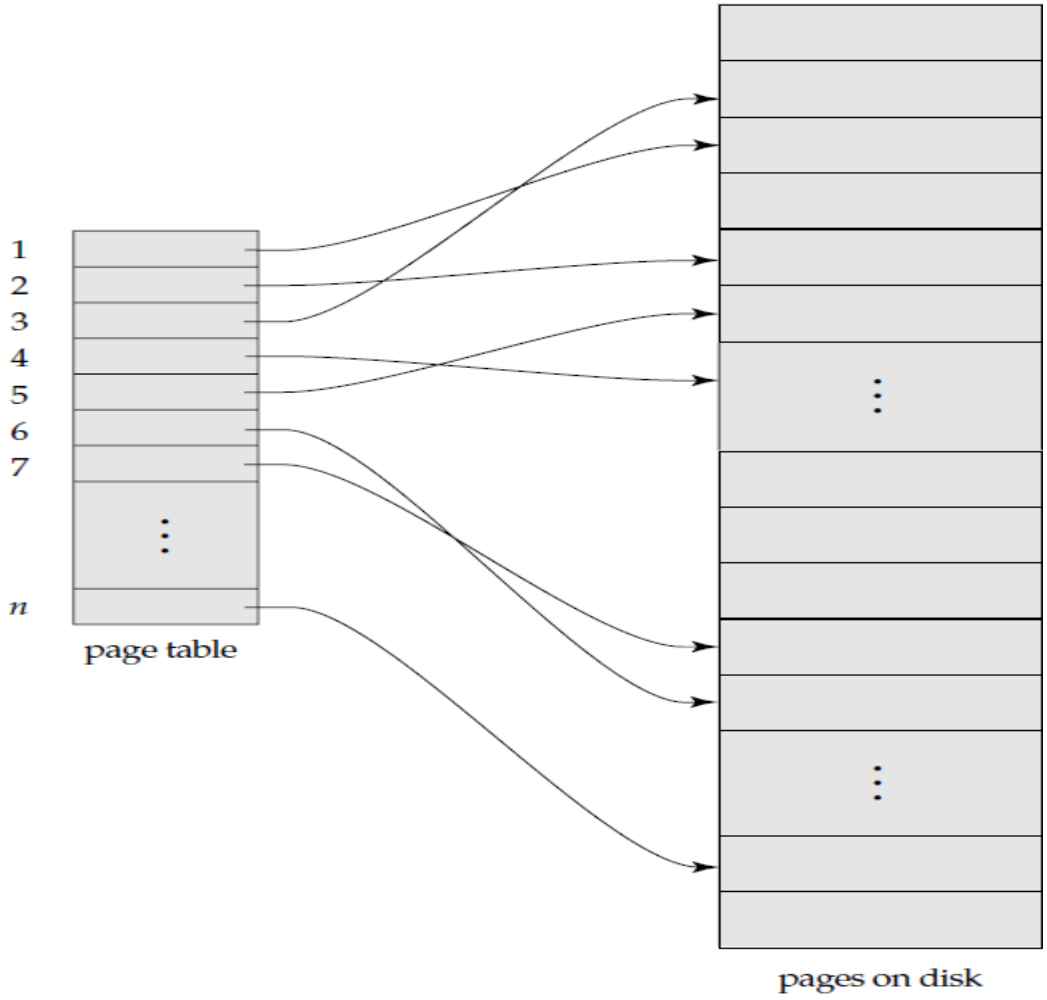


Figure Sample page table.

The key idea behind the shadow-paging technique is to maintain ***two*** page tables during the life of a transaction: the **current page table** and the **shadow page table**. When the transaction starts, both page tables are identical. The shadow page table is never changed over the duration of the transaction. The current page table may be changed when a transaction performs a write operation. All input and output operations use the current page table to locate database pages on disk.

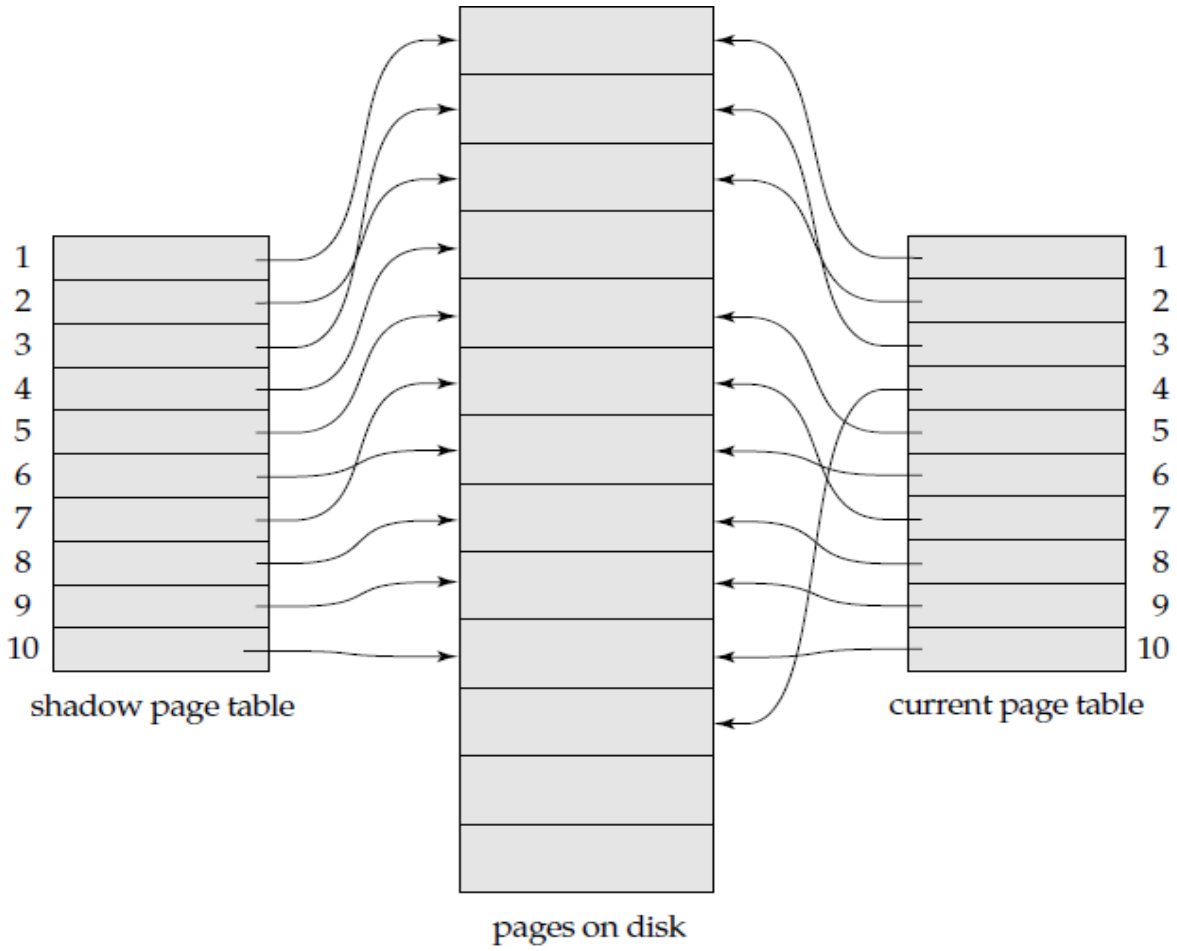


Figure Shadow and current page tables.

Drawbacks of the shadow-page technique:

•**Commit overhead**. The commit of a single transaction using shadow paging requires multiple blocks to be output—the actual data blocks, the current page table, and the disk address of the current page table.

•**Data fragmentation**. we consider strategies to ensure locality —that is, to keep related database pages close physically on the disk. Locality allows for faster data transfer. Shadow paging causes database pages to change location when they are updated.

•**Garbage collection**. Each time that a transaction commits, the database pages containing the old version of data changed by the transaction become inaccessible. Such pages are considered **garbage**, since they are not part of free space and do not contain usable information. Garbage

may be created also as a side effect of crashes. Periodically, it is necessary to find all the garbage pages, and to add them to the list of free pages. This process, called **garbage collection**,

In addition to the drawbacks shadow paging is more difficult than logging to adapt to systems that allow several transactions to execute concurrently.

5.14 Recovery with Concurrent Transactions

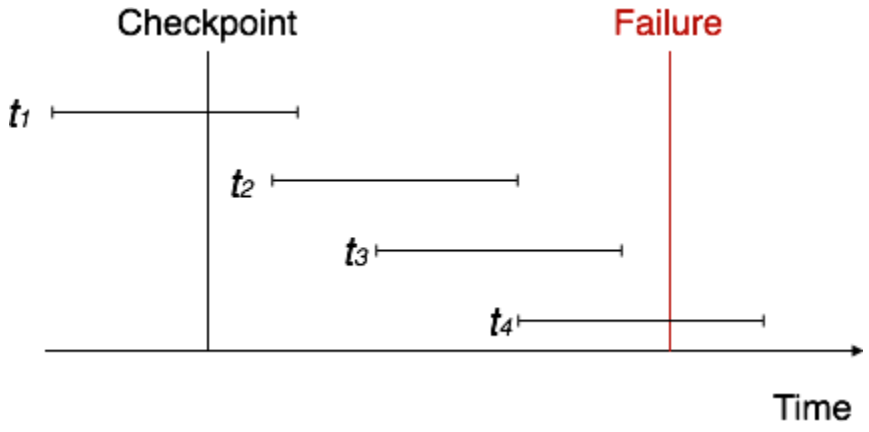
When more than one transaction are being executed in parallel, the logs are interleaved. At the time of recovery, it would become hard for the recovery system to backtrack all logs, and then start recovering. To ease this situation, most modern DBMS use the concept of 'checkpoints'.

Checkpoint

Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. As time passes, the log file may grow too big to be handled at all. Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

Recovery

When a system with concurrent transactions crashes and recovers, it behaves in the following manner −



- The recovery system reads the logs backwards from the end to the last checkpoint.
- It maintains two lists, an undo-list and a redo-list.

- If the recovery system sees a log with <T_n, Start> and <T_n, Commit> or just <T_n, Commit>, it puts the transaction in the redo-list.
- If the recovery system sees a log with <T_n, Start> but no commit or abort log found, it puts the transaction in undo-list.

All the transactions in the undo-list are then undone and their logs are removed. All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.

5.14 Buffer Management

Log-Record Buffering

The cost of performing the output of a block to stable storage is sufficiently high that it is desirable to output multiple log records at once, using a buffer. When the buffer is full, It is output with as few output operations as possible. However, a log record may reside in only main memory for a considerable time before it is actually written to stable storage. Such log records are lost if the system crashes. It is necessary, therefore, to write all buffers related to a transaction when it is committed. There is no problem written the other uncommitted transactions at this time

Database Buffering

Database buffering is the standard operating system concept of virtual memory. Whenever blocks of the database in memory must be replaced, all modified data blocks and log records associated with those blocks must be written to the disk.

Operating System Role in Buffer Management

We can manage the database buffer sing one of two approaches:

1. The database system reserves part of main memory to serve as a buffer that the DBMS manages instead of the operating system. This means that the buffer must be kept as small as possible (because of its impact on other processes active on the CPU) and it adds to the complexity of the DBMS
2. The DBMS implements its buffer within the virtual memory of the operating system. The operating system would then have to coordinate the swapping of pages to insure that the appropriate buffers were also written to disk. Unfortunately, almost all current-generation operating systems retain complete control of virtual memory. The operating system reserves space on disk for storing virtual memory pages that are not currently in main memory, called swap space. This approach may result in extra output to the disk.

15. Failure with Loss of Nonvolatile Storage

To recover from the loss of nonvolatile storage, the system restores the database to disk by using the most recent dump. Then, it consults the log and redoes all the transactions that have committed since the most recent dump occurred. Notice that no undo operations need to be executed.

A dump of the database contents is also referred to as an **archival dump**, since we can archive the dumps and use them later to examine old states of the database. Dumps of a database and check pointing of buffers are similar. The simple dump procedure is costly for the following two reasons.

First, the entire database must be be copied to stable storage, resulting in considerable data transfer.

Second, since transaction processing is halted during the dump procedure, CPU cycles are wasted. **Fuzzy dump** schemes have been developed, which allow transactions to be active while the dump is in progress.

16. Advance Recovery systems

Logical Undo Log Records : To allow logical undo of operations, before an operation is performed to modify an index, the transaction creates a log record <Ti , Oj , operation-begin>, where Oj is a unique identifier for the operation instance. While the system is executing the operation, it creates update log records in the normal fashion for all updates performed by the operation. When the operation finishes, it writes an operation-end log record of the form <*Ti , Oj* , operation-end, *U*>, where the *U* denotes undo information.

Logging of information about operations is called **logical logging**. In contrast, logging of old-value and new-value information is called **physical logging**, and the corresponding log records are called **physical log records**.

Transaction Rollback :

When rolling back a transaction Ti , the log is scanned backwards.

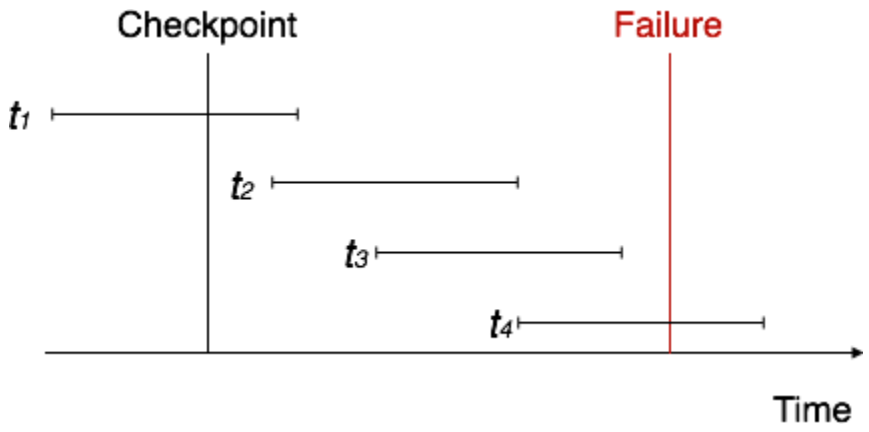
Checkpoint

Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. As time passes, the log file may grow too big to be handled at all. Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

Restart Recovery

Redo and undo explanation.

When a system with concurrent transactions crashes and recovers, it behaves in the following manner −



- The recovery system reads the logs backwards from the end to the last checkpoint.
- It maintains two lists, an undo-list and a redo-list.
- If the recovery system sees a log with <T_n, Start> and <T_n, Commit> or just <T_n, Commit>, it puts the transaction in the redo-list.
- If the recovery system sees a log with <T_n, Start> but no commit or abort log found, it puts the transaction in undo-list.

All the transactions in the undo-list are then undone and their logs are removed. All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.

fuzzy checkpoints

In fuzzy check pointing, At the time of checkpoint, all the active transactions are written in the log. In case of power failure, the recovery manager processes only those transactions that were active during checkpoint and later. The transactions that have been committed before checkpoint are written to the disk and hence need not be redone.

5.17 ARIES

Algorithms for **Recovery** and **Isolation Exploiting Semantics**. In computer science, **Algorithms** for **Recovery** and **Isolation Exploiting Semantics**, or **ARIES** is a **recovery algorithm** designed to work with a no-force, steal database approach; it is used by IBM DB2, Microsoft SQL Server and many other database systems.

The major differences between ARIES and the recovery algorithm presented earlier are that ARIES:

1. Uses a **log sequence number (LSN)** to identify log records, and stores LSNs in database pages to identifywhich operations have been applied to a database page.

2.Supports **physiological redo operations**, which are physical in that the affected page is physically identified, but can be logical within the page.

3. Uses a **dirty page table** to minimize unnecessary redos during recovery. Dirty pages are those that have been updated in memory, and the disk version is not up-to-date.

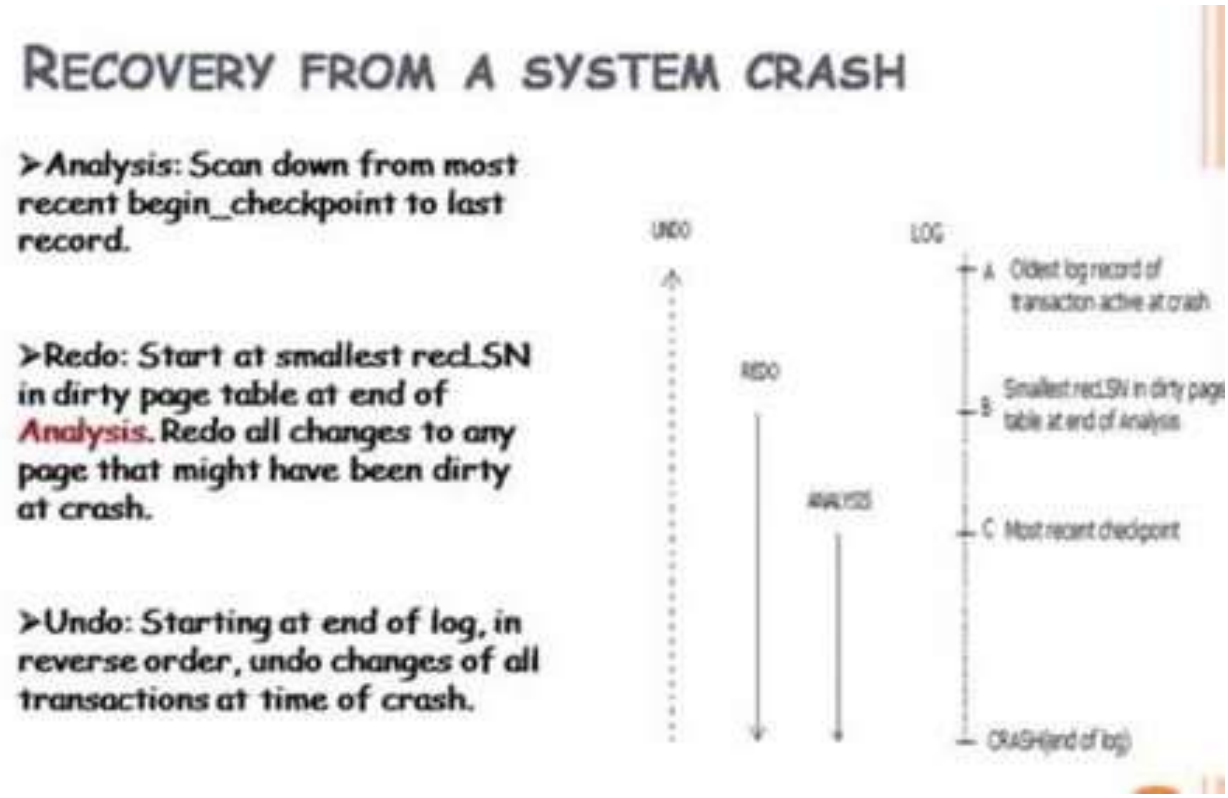
4. Uses a **fuzzy-checkpointing** scheme that records only information about of dirty pages to disk. It flushes dirty pages in the background, continuously, instead of writing them during checkpoints.

ARIES recovers from a system crash in **three** passes.

•**Analysis pass**: This pass determines which transactions to undo, which pages were dirty at the time of the crash, and the LSN from which the redo pass should start.

• **Redo pass**: This pass starts from a position determined during analysis, and performs a redo, repeating history, to bring the database to a state it was in before the crash.

• **Undo pass**: This pass rolls back all transactions that were incomplete at the time of crash.



Analysis Pass

<<<<<class Notes >>>>>

Redo Pass

<<<<<class Notes >>>>>

Undo Pass

<<<<<class Notes >>>>>

Among other key features that ARIES provides are:

• **Recovery independence**: Some pages can be recovered independently from others, so that they can be used even while other pages are being recovered. If some pages of a disk fail, they can be recovered without stopping transaction processing on other pages.

• **Save points**: Transactions can record save points, and can be rolled back partially, up to a save point. This can be quite useful for deadlock handling, since transactions can be rolled back up to a point that permits release of required locks, and then restarted from that point. Programmers

can also use save points to undo a transaction partially, and then continue execution; this approach can be useful to handle certain kinds of errors detected during the transaction execution.

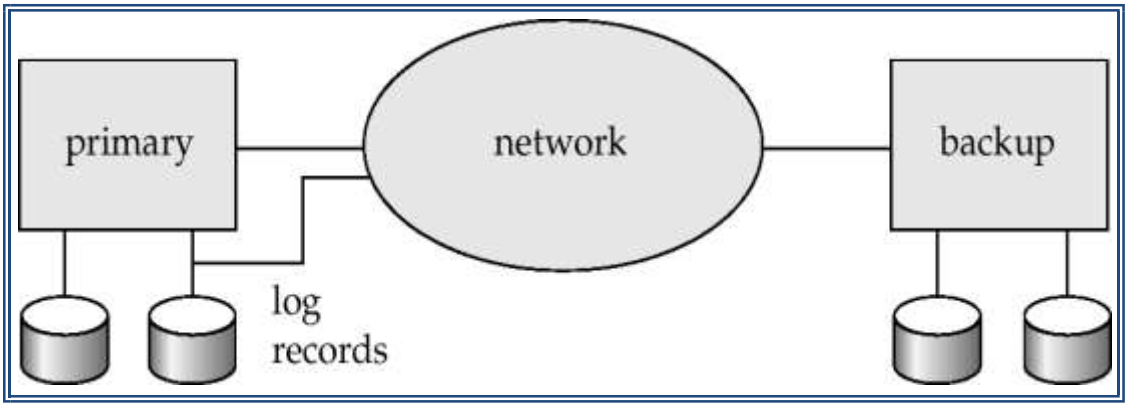
•**Fine-grained locking**: The ARIES recovery algorithm can be used with index concurrency-control algorithms that permit tuple-level locking on indices, instead of page-level locking, which improves concurrency significantly.

• **Recovery optimizations**: The Dirty Page Table can be used to prefetch pages during redo, instead of fetching a page only when the system finds a log record to be applied to the page. Out-of-order redo is also possible: Redo can be postponed on a page being fetched from disk, and performed when the page is fetched. Meanwhile, other log records can continue to be processed.

5.18 Remote Backup Systems

Traditional transaction-processing systems are centralized or client–server systems. Such systems are vulnerable to environmental disasters such as fire, flooding, or earthquakes. Increasingly, there is a need for transaction-processing systems that can function in spite of system failures or environmental disasters. Such systems must provide **high availability**; that is, the time for which the system is unusable must be extremely small.

We can achieve high availability by performing transaction processing at one site, called **the primary site**, and having a **remote backup** site where all the data from the primary site are replicated. The remote backup site is sometimes also called the **secondary site**. The remote site must be kept synchronized with the primary site, as updates are performed at the primary. We achieve synchronization by sending all log records from primary site to the remote backup site. The remote backup site must be physically separated from the primary.



When the primary site fails, the remote backup site takes over processing. First, however, it performs recovery, using its copy of the data from the primary, and the log records received from the primary. In effect, the remote backup site is performing recovery actions that would have been performed at the primary site when the latter recovered. Standard recovery algorithms, with minor modifications, can be used for recovery at the remote backup site. Once recovery has been performed, the remote backup site starts processing transactions. Availability is greatly increased over a single-site system, since the system can recover even if all data at the primary site are lost.

Several issues must be addressed in designing a remote backup system:

Detection of failure. It is important for the remote backup system to detect when the primary has failed. Failure of communication lines can fool the remote backup into believing that the primary has failed. To avoid this problem, we maintain several communication links with independent modes of failure between the primary and the remote backup.

•**Transfer of control**. When the primary fails, the backup site takes over processing and becomes the new primary. When the original primary site recovers, it can either play the role of remote backup, or take over the role of primary site again. In either case, the old primary must receive a log of updates carried out by the backup site while the old primary was down. The simplest way of transferring control is for the old primary to receive redo logs from the old backup site, and to catch up with the updates by applying them locally. The old primary can then act as a remote backup site. If control must be transferred back, the old backup site can pretend to have failed, resulting in the old primary taking over.

•**Time to recover**. If the log at the remote backup grows large, recovery will take a long time. The remote backup site can periodically process the redo log records that it has received and can perform a checkpoint, so that earlier parts of the log can be deleted. The delay before the remote backup takes over can be significantly reduced as a result.

A **hot-spare** configuration can make takeover by the backup site almost instantaneous. In this configuration, the remote backup site continually processes redo log records as they arrive, applying the updates locally. As soon as the failure of the primary is detected, the backup site completes recovery by rolling back incomplete transactions; it is then ready to process new transactions.

•**Time to commit**. To ensure that the updates of a committed transaction are durable, a transaction must not be declared committed until its log records have reached the backup site. This delay can result in a longer wait to commit a transaction, and some systems therefore permit lower degrees of durability. The degrees of durability can be classified as follows:

- **One-safe**. A transaction commits as soon as its commit log record is written to stable storage at the primary site.
- **Two-very-safe**. A transaction commits as soon as its commit log record is written to stable storage at the primary and the backup site.
- **Two-safe.** This scheme is the same as two-very-safe if both primary and backup sites are active. If only the primary is active, the transaction is allowed to commit as soon as its commit log record is written to stable storage at the primary site.

This scheme provides better availability than does two-very-safe,while avoiding the problem of lost transactions faced by the one-safe scheme. It results in a slower commit than the one-safe scheme, but the benefits

generally outweigh the cost.