MODULE 5

TRANSACTION MANAGEMENT

Transaction A transaction is a unit of a program execution that accesses and possibly modifies various data objects (tuples, relations).
A transaction is a Logical unit of database processing that includes one or more access operations (read -retrieval, write - insert or update, delete).
A transaction (set of operations) may be stand-alone specified in a high level language like SQL submitted interactively, or may be embedded within a program.
A transaction (collection of actions) makes transformations of system states while preserving the database consistency.
A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.
A transaction is the DBMS's abstract view of a user program: a sequence of reads and writes.

Transaction boundaries: Begin and End transaction.
[Note: An **application program** may contain several transactions separated by Begin and End transaction boundaries]

Transaction Processing Systems: Large multi-user database systems supporting thousands of *concurrent transactions* (user processes) per minute

Two Modes of Concurrency

- **Interleaved processing**: concurrent execution of processes is interleaved in a single CPU
- **Parallel processing**: processes are concurrently executed in multiple CPUs [below figure]
- Basic transaction processing theory assumes interleaved concurrency

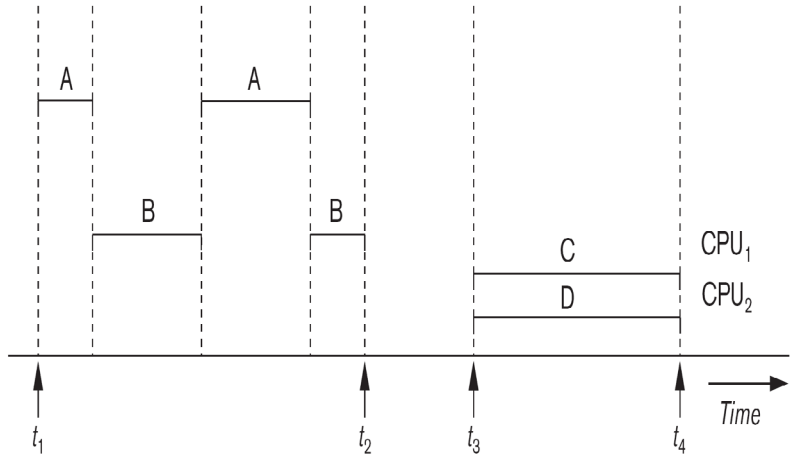


Figure 21.1
Interleaved processing versus parallel processing of concurrent transactions.

SIMPLE MODEL OF A DATABASE

A **database** is a collection of named data items.

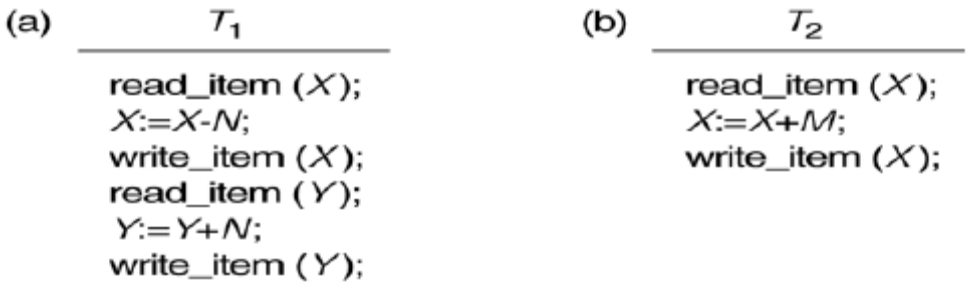
Granularity of data - a field, a record, or a whole disk block (Concepts are independent of granularity).
Basic operations are **read** and **write**:

read_item(X): Reads a database item named X into a program variable. To simplify our notation, we assume that the program variable is also named X.
write_item(X): Writes the value of program variable X into the database item named X.

READ AND WRITE OPERATIONS:
Basic unit of data transfer from the disk to the computer main memory is one block. In general, a data item (what is read or written) will be the field of some record in the database, although it may be a larger unit such as a record or even a whole block.

read_item(X) command includes the following steps:
1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the buffer to the program variable named X.

write_item(X) command includes the following steps:
1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the program variable named X into its correct location in the buffer.
4. Store the updated block from the buffer back to disk (either immediately or at some later point in time).
Two sample transactions



Transaction Example in MySQL
START TRANSACTION;
SELECT@A:=SUM(salary) FROMtable1 WHEREtype=1;
UPDATEtable2 SETsummary=@A WHEREtype=1;
COMMIT;

Transaction Example in Oracle(same with SQL Server)
•When you connect to the database with sqlplus(Oracle command-line utility that runs SQL and PL/SQL commands interactively or from a script) a transaction begins.
•Once the transaction begins, every SQL DML (Data Manipulation Language) statement you issue subsequently becomes a part of this transaction

TRANSACTION STATES
1. Active state
2. Partially committed state
3. Committed state
4. Failed state
5. Terminated State

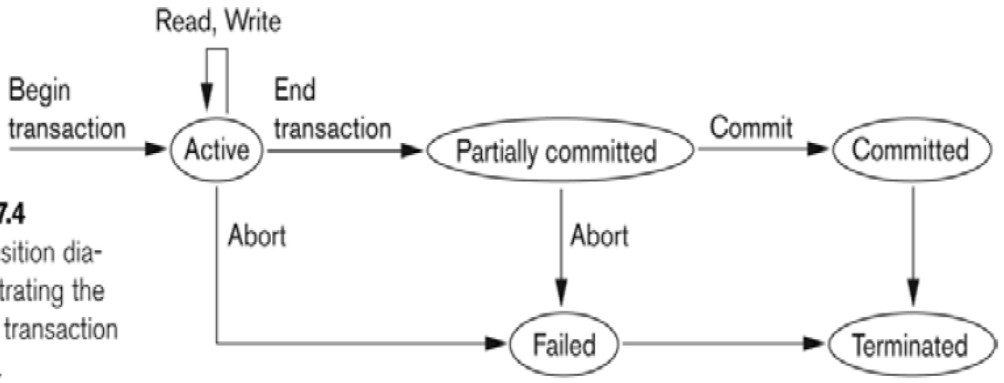


Figure 17.4
State transition diagram illustrating the states for transaction execution.

Why Concurrency Control is needed?
Problems that can occur for certain transaction schedules without appropriate concurrency
Control mechanisms:

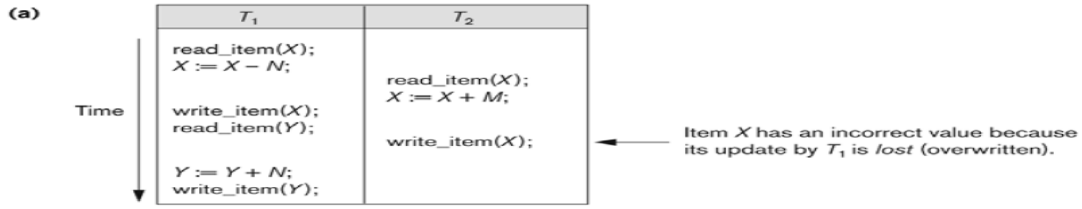
The Lost Update Problem
This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.

The Temporary Update (or Dirty Read) Problem
This occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed by another transaction before it is changed back to its original value.

The Incorrect Summary Problem
If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

a) The Lost Update Problem

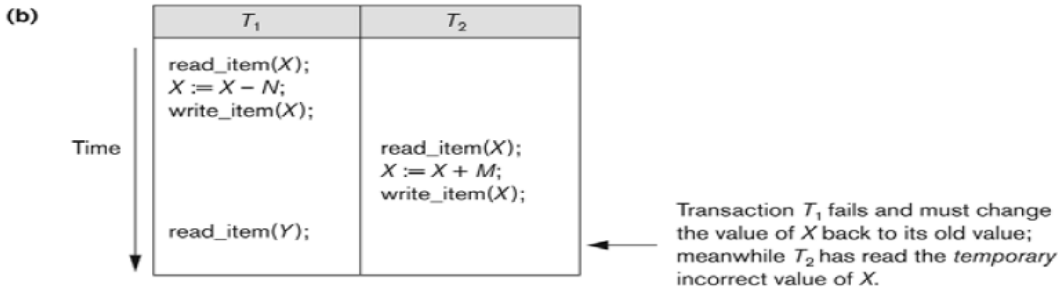
Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



The update performed by T1 gets lost; possible solution: T1 locks/unlocks database object X =) T2 cannot read X while X is modified by T1

b) The temporary update problem

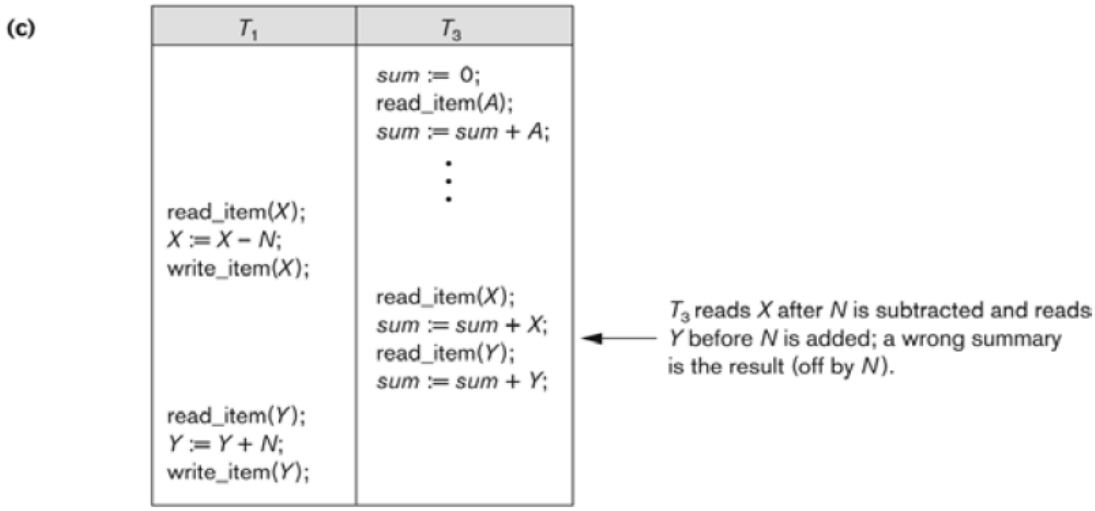
Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



T1 modifies db object, and then the transactionT1 fails for some reason. Meanwhile the modified db object, however, has been accessed by another transaction T2. Thus T2 has read data that never existed.

c) The incorrect summary problem

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



In this schedule, the total computed by T1 is wrong.=) T1 must lock/unlock several db objects

The following are the Problems that arise when interleaving Transactions (and they are already discussed above but the terminology is different):

Problem 1: Reading Uncommitted Data (WR Conflicts)
Reading the value of an uncommitted object might yield an inconsistency
–Dirty Reads or Write-then-Read (WR) Conflicts.

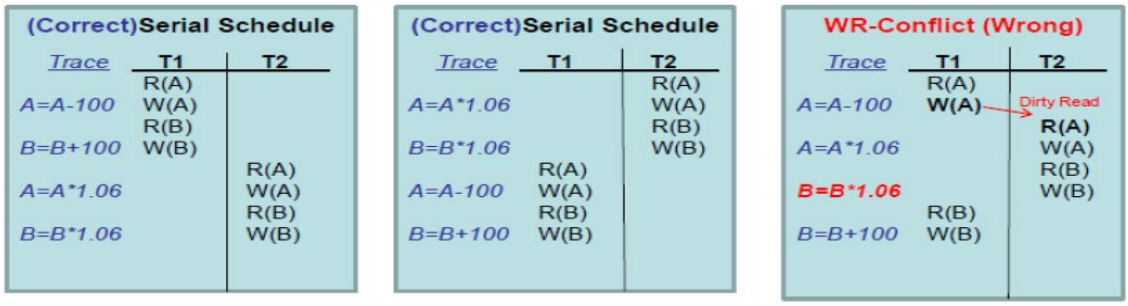
Problem 2: Unrepeatable Reads (RW Conflicts)
Reading the same object twice might yield an inconsistency
–Read-then-Write (RW) Conflicts (Write-After-Read)

Problem 3: Overwriting Uncommitted Data (WW Conflicts)
Overwriting an uncommitted object might yield an inconsistency
–Lost Update or Write-After-Write (WW) Conflicts.

Remark: There is no notion of RR-Conflicts no object is changed

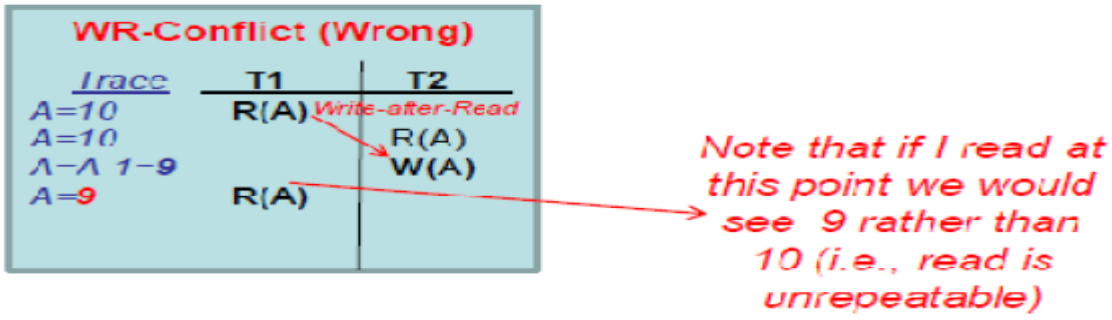
The Unrepeatable Read Problem .
A transaction T1 may read an item (say, available seats on a flight); later, T1 may read the same item again and get a different value because another transaction T2 has updated the item (reserved seats on the flight) between the two reads by T1

1. Reading Uncommitted Data (WR Conflicts)
To illustrate the WR-conflict consider the following problem:
T1: Transfer $100 from Account A to Account B
T2: Add the annual interest of 6% to both A and B.



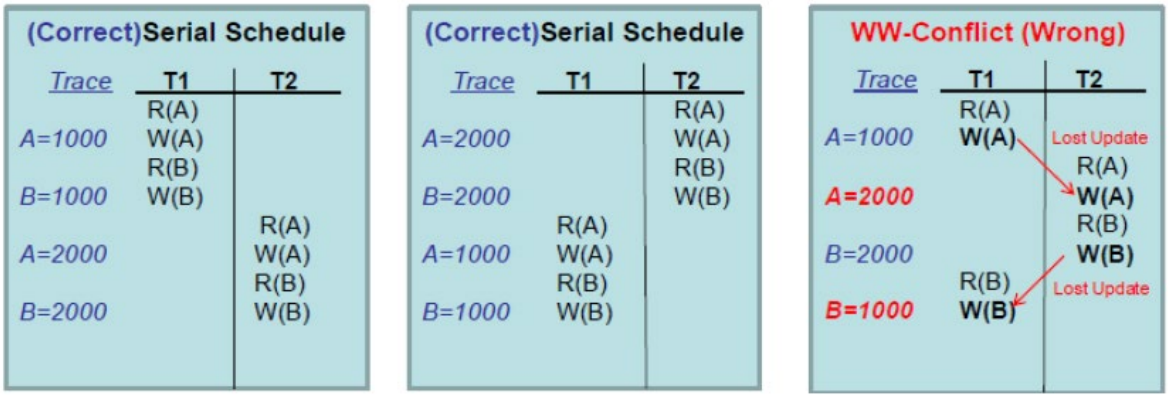
Problem caused by the WR-Conflict? Account B was credited with the interest on a smaller amount (i.e., 100$ less), thus the result is not equivalent to the serial schedule.

2. Unrepeatable Reads (RW Conflicts)
To illustrate the RW-conflict, consider the following problem:
T1: Print Value of A
T2: Decrease Global counter A by 1.



Problem caused by the RW-Conflict? Although the "A" counter is read twice within T1 (without any intermediate change) it has two different values (unrepeatable read)! What happens if T2 aborts? _1 has shown an incorrect result.

3. Overwriting Uncommitted Data (WW Conflicts)
To illustrate the WW-conflict consider the following problem:
Salary of employees A and B must be kept equal
T1: Set Salary to 1000; T2: Set Salary equal to 2000



Problem caused by the WW-Conflict?
Employee "A" gets a salary of 2000 while employee "B" gets a salary of 1000, thus result is not equivalent to the serial schedule!

Why **recovery is needed:** (What causes a Transaction to fail)
1. A computer failure (system crash):
A hardware or software error occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.

2. A transaction or system error:
Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.

3. Local errors or exception conditions detected by the transaction:
Certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled. A programmed abort in the transaction causes it to fail.

4. Concurrency control enforcement:
The concurrency control method may decide to abort the transaction, to be restarted later,because it violates serializability or because several transactions are in a state of deadlock.

5. Disk failure:
Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

6. Physical problems and catastrophes:
This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Recovery manager keeps track of the following operations:

begin_transaction: This marks the beginning of transaction execution.

read or write: These specify read or write operations on the database items that are executed as part of a transaction.

end_transaction: This specifies that read and write transaction operations have ended and marks the end limit of transaction execution.At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database or whether the transaction has to be aborted because it violates concurrency control or for some other reason.

commit_transaction: This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.

rollback (or abort): This signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.

Recovery techniques use the following operators:
undo: Similar to rollback except that it applies to a single operation rather than to a whole transaction.

redo: This specifies that certain *transaction* *operations must be redone to ensure that all the* operations of a committed transaction have been applied successfully to the database.

The System Log
Log or Journal: The log keeps track of all transaction operations that affect the values of database items.

This information may be needed to permit recovery from transaction failures. The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.

In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.

T in the following discussion refers to a unique **transaction-id** that is generated automatically by the system and is used to identify each transaction:

The following actions are recorded in the log:
-Ti writes an object: the old value and the new value.
Log record must go to disk before the changed page!
-Ti commits/aborts: a log record indicating this action.

Log records are chained together by Xact id, so it's easy to undo a specific Xact. Log is often duplexed and archived on stable storage.

All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.

Types of log record:
[start_transaction,T]: Records that transaction T has started execution.
[write_item,T,X,old_value,new_value]: Records that transaction T has changed the value of
database item X from old_value to new_value.
[read_item,T,X]: Records that transaction T has read the value of database item X.
[commit,T]: Records that transaction T has completed successfully, and affirms that its effect
can be committed (recorded permanently) to the database.
[abort,T]: Records that transaction T has been aborted.
Protocols for recovery that *avoid cascading rollbacks do not require that read operations be*
*written to the system log, whereas other protocols*require these entries for recovery.
Strict protocols require simpler write entries that do not include new_value

Recovery using log records:
If the system crashes, we can recover to a consistent database state by examining the log.

1. Because the log contains a record of every write operation that changes the value of some database item, it is possible to **undo the effect of these write operations of a transaction T** by tracing backward through the log and resetting all items changed by a write operation of T to their old_values.
2. We can also **redo the effect of the write operations of a** transaction T by tracing forward through the log and setting all items changed by a write operation of T (that did not get done permanently) to their new_values.

Commit Point of a Transaction:
Definition a Commit Point: A transaction T reaches its commit point when all its operations that access the database have been executed successfully *and the effect of all the transaction operations on* the database has been recorded in the log.

Beyond the commit point, the transaction is said to be committed, and its effect is assumed to be permanently recorded in the database.

The transaction then writes an entry [commit,T] into the log.

Roll Back of transactions:
Needed for transactions that have a [start_transaction,T] entry into the log but no commit entry [commit,T] into the log.

Redoing transactions:
Transactions that have written their commit entry in the log must also have recorded all their write operations in the log; otherwise they would not be committed, so their effect on the database can be redone from the log entries. (Notice that the log file must be kept on disk. At the time of a system crash, only the log entries that have been written back to disk are considered in the recovery process because the contents of main memory may be lost.)

Force writing a log:
Before a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called force-writing the log file before committing a transaction.

ACID Properties:
The DBMS need to ensure the following properties of transactions:

1. Atomicity

– Transactions are either done or not done
– They are never left partially executed

An executing transaction completes in its entirety or it is aborted altogether.
–e.g., Transfer_Money (Amount, X, Y) means
i) DEBIT (Amount, X);
ii) CREDIT (Amount, Y). Either both take place or none

2. Consistency

-Transactions should leave the database in a consistent state If each Transaction is consistent, and the DB starts consistent, then the Database ends up consistent.
-If a transaction violates the database's consistency rules, the entire transaction will be rolled back and the database will be restored to a state consistent with those rules.

3. Isolation

– Transactions must behave as if they were executed in isolation.
An executing transaction cannot reveal its (incomplete) results before it commits.
–Consequently, the net effect is identical to executing all transactions, the one after the other in some serial order.

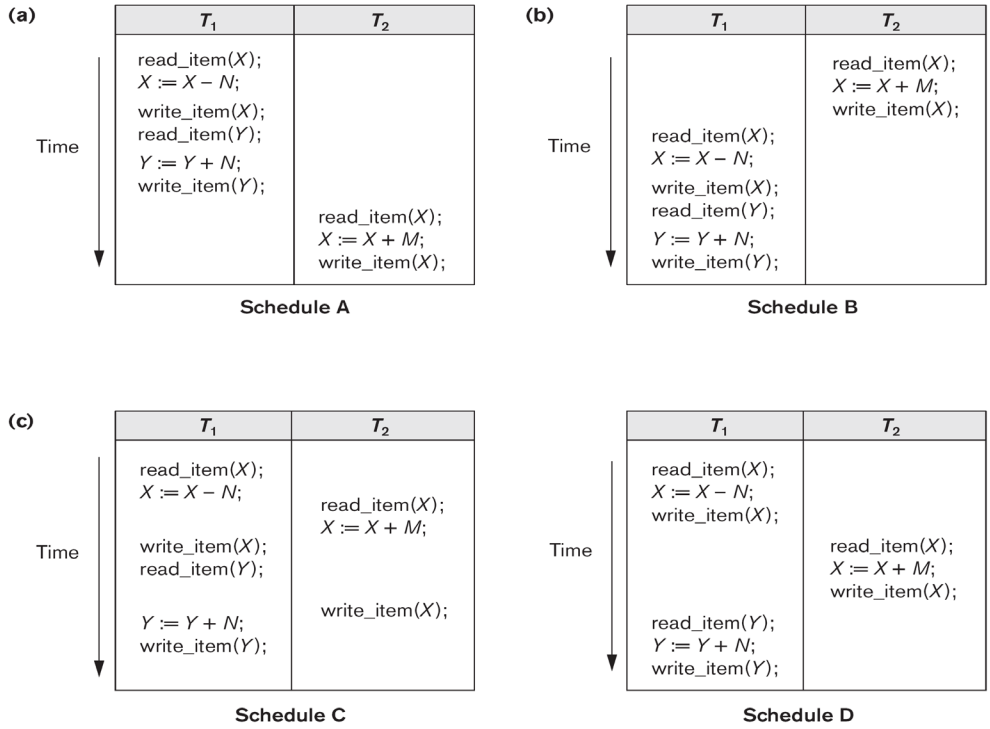
4. Durability

– Effects of completed transactions are resilient against failures
Once a transaction commits, the system must guarantee that the results of its operations will never be lost, in spite of subsequent failures.

Schedules of Transactions

- **Transaction schedule (or history):** When transactions are executing concurrently in an interleaved fashion, the *order of execution* of operations from the various transactions forms what is known as a **transaction schedule** (or history).
- Below Figure shows 4 possible schedules (A, B, C, D) of two transactions T1 and T2:
  - Order of operations from top to bottom
  - Each schedule includes same operations
  - Different order of operations in each schedule

Figure 21.5
Examples of serial and nonserial schedules involving transactions T_1 and T_2. (a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1. (c) Two nonserial schedules C and D with interleaving of operations.



–

- Schedules can also be displayed in more compact notation
- Order of operations from left to right
- Include only read (r) and write (w) operations, with transaction id (1, 2, …) and item name (X, Y, …)
- Can also include other operations such as b (begin), e (end), c (commit), a (abort)
- Schedules in above Figure would be displayed as follows:
  - Schedule A: r1(X); w1(X); r1(Y); w1(Y); r2(X); w2(x);
  - Schedule B: r2(X); w2(X); r1(X); w1(X); r1(Y); w1(Y);
  - Schedule C: r1(X); r2(X); w1(X); r1(Y); w2(X); w1(Y);
  - Schedule D: r1(X); w1(X); r2(X); w2(X); r1(Y); w1(Y);

Formal definition of a **schedule** (or **history**) S of n transactions T1, T2, ..., Tn : An ordering of all the operations of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of Ti in S must appear *in the same order* in which they occur in Ti.

[Note: Operations from other transactions Tj can be interleaved with the operations of Ti in S.]

For n transactions T1, T2, ..., Tn, where each Ti has mi read and write operations, the number of possible schedules is (! is *factorial* function):

(m1 + m2 + … + mn)! / ((m1)! * (m2)! * … * (mn)!) are Generally very large number of possible schedules. Some schedules are easy to recover from after a failure, while others are not and Some schedules produce correct results, while others produce incorrect results

Characterizing Schedules based on Recoverability:
Schedules classified into two main classes:

- **Recoverable schedule:** One where no *committed* transaction needs to be rolled back (aborted).

A schedule S is **recoverable** if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.

- **Non-recoverable schedule:** A schedule where a committed transaction may have to be rolled back during recovery.

This violates **Durability** from ACID properties (a committed transaction cannot be rolled back) and so non-recoverable schedules *should not be allowed*.

- **Example:** Schedule A below is **non-recoverable** because T2 reads the value of X that was written by T1, but then T2 commits before T1 commits or aborts
- To make it **recoverable**, the commit of T2 (c2) must be delayed until T1 either commits, or aborts (Schedule B)
- If T1 commits, T2 can commit
- If T1 aborts, T2 must also abort because it read a value that was written by T1; this value must be undone (reset to its old value) when T1 is aborted
  - known as *cascading rollback*
- Schedule A: r1(X); w1(X); r2(X); w2(X); c2; r1(Y); w1(Y); c1 (or a1)
- Schedule B: r1(X); w1(X); r2(X); w2(X); r1(Y); w1(Y); c1 (or a1); …

Recoverable schedules can be further refined:
Cascadeless schedule: A schedule in which a transaction T2 cannot read an item X until the transaction T1 that last wrote X has committed and The set of cascadeless schedules is a *subset of* the set of recoverable schedules.

Schedules requiring cascaded rollback: A schedule in which an uncommitted transaction T2 that read an item that was written by a failed transaction T1 must be rolled back

Example: Schedule B below is **not cascadeless** because T2 reads the value of X that was written by T1 before T1 commits

- If T1 aborts (fails), T2 must also be aborted (rolled back) resulting in *cascading rollback*
- To make it **cascadeless**, the r2(X) of T2 must be delayed until T1 commits (or aborts and rolls back the value of X to its previous value) – see Schedule C
- Schedule B: r1(X); w1(X); r2(X); w2(X); r1(Y); w1(Y); c1 (or a1);
- Schedule C: r1(X); w1(X); r1(Y); w1(Y); c1; r2(X); w2(X); …

Cascadeless schedules can be further refined**:**
Strict schedule: A schedule in which a transaction T2 can neither read *nor write* an item X until the transaction T1 that last wrote X has committed.

The set of strict schedules is a *subset of* the set of cascadeless schedules.

If *blind writes* are not allowed, all cascadeless schedules are also strict

Blind write: A write operation w2(X) that is not preceded by a read r2(X).
Example: Schedule C below is cascadeless and also strict (because it has no blind writes)

- Schedule D is cascadeless, but not strict (because of the blind write w3(X), which writes the value of X before T1 commits)
- To make it strict, w3(X) must be delayed until after T1 commits – see Schedule E
- Schedule C: r1(X); w1(X); r1(Y); w1(Y); c1; r2(X); w2(X); …
- Schedule D: r1(X); w1(X); w3(X); r1(Y); w1(Y); c1; r2(X); w2(X); …
- Schedule E: r1(X); w1(X); r1(Y); w1(Y); c1; w3(X); r2(X); w2(X); …

Serial schedules are *not feasible* for performance reasons:

- No interleaving of operations
- Long transactions force other transactions to wait
- System cannot switch to other transaction when a transaction is waiting for disk I/O or any other event
- Need to allow concurrency with interleaving without sacrificing correctness

Serializable schedule: A schedule S is **serializable** if it is **equivalent** to some serial schedule of the same n transactions. There are (n)! serial schedules for n transactions – a serializable schedule can be equivalent to *any of the serial schedules*

Equivalence of Schedules
Two schedules are called result equivalent if they produce the same final state of the database.

It Difficult to determine without *analyzing the internal operations of the transactions*, which is not feasible in general and it May also get result equivalence *by chance* for a particular input parameter even though schedules *are not equivalent in general*

Figure 21.6
Two schedules that are result equivalent for the initial value of $X = 100$ but are not result equivalent in general.

S_1

read_item(X);
$X := X + 10$;
write_item(X);

S_2
read_item(X); $X := X * 1.1$; write_item (X);

Conflict equivalent:
Two schedules are conflict equivalent if the relative order of *any two conflicting operations* is the same in both schedules.

Two operations are **conflicting** if:
- They access the same data item X
- They are from two different transactions
- At least one is a write operation

Read-Write conflict example: r1(X) and w2(X)
Write-write conflict example: w1(Y) and w2(Y)

A serializable schedule is considered to be correct because it is equivalent to a serial schedule, and any serial schedule is considered to be correct
- It will leave the database in a consistent state.
- The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution and interleaving of operations from different transactions.

Serializability is generally hard to check at run-time:
- Interleaving of operations is generally handled by the operating system through the process scheduler
- Difficult to determine beforehand how the operations in a schedule will be interleaved
- Transactions are continuously started and terminated

Testing for conflict serializability (Algorithm)
- Looks at only r(X) and w(X) operations in a schedule
- Constructs a precedence graph (serialization graph) – **one node for each transaction**, plus directed edges
- An **edge is created** from T_i to T_j if one of the operations in T_i appears before a conflicting operation in T_j
- The schedule is serializable if and only if the precedence graph **has no cycles**.

Algorithm 21.1. Testing Conflict Serializability of a Schedule S

1. For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph.
2. For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
3. For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
4. For each case in S where T_j executes a write_item(X) after T_i executes a write_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

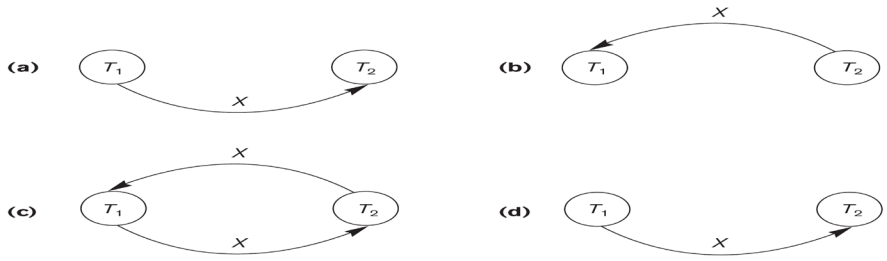


Figure 21.7
Constructing the precedence graphs for schedules A to D from Figure 21.5 to test for conflict serializability. (a) Precedence graph for serial schedule A. (b) Precedence graph for serial schedule B. (c) Precedence graph for schedule C (not serializable). (d) Precedence graph for schedule D (serializable, equivalent to schedule A).

Figure 21.8
Another example of serializability testing. (a) The read and write operations of three transactions T_1, T_2, and T_3. (b) Schedule E. (c) Schedule F.

(a)

Transaction T_1	Transaction T_2	Transaction T_3

read_item(X);
write_item(X);
read_item(Y);
write_item(Y);
	read_item(Z);
	read_item(Y);
	write_item(Y);
	read_item(X);
	write_item(X);
		read_item(Y);
		read_item(Z);
		write_item(Y);
		write_item(Z);

(b)

Time

Transaction T_1	Transaction T_2	Transaction T_3

	read_item(Z);
	read_item(Y);
	write_item(Y);	

read_item(X); write_item(X);		read_item(Y); read_item(Z);
		write_item(Y); write_item(Z);
	read_item(X);	
read_item(Y); write_item(Y);		
	write_item(X);	

Schedule E

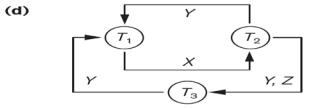
(c)

Time

Transaction T_1	Transaction T_2	Transaction T_3
		read_item(Y); read_item(Z);
read_item(X); write_item(X);		
		write_item(Y); write_item(Z);
	read_item(Z);	
read_item(Y); write_item(Y);		

	read_item(Y);
	write_item(Y);
	read_item(X);
	write_item(X);	

Schedule F



Equivalent serial schedules

None

Reason

Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$
Cycle $X(T_1 \rightarrow T_2), YZ\ (T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$
$T_3 \rightarrow T_2 \rightarrow T_1$

Figure 21.8 (continued)
Another example of serializability testing. (d) Precedence graph for schedule E. (e) Precedence graph for schedule F. (f) Precedence graph with two equivalent serial schedules.

View equivalence: A less restrictive definition of equivalence of schedules than conflict serializability *when blind writes are allowed*

View serializability: definition of serializability based on view equivalence. A schedule is *view serializable* if it is *view equivalent* to a serial schedule

Introduction to Transaction Support in SQL

- A single SQL statement is always considered to be atomic. Either the statement completes execution without error or it fails and leaves the database unchanged.

- With SQL, there is no explicit Begin Transaction statement. Transaction initiation is done implicitly when particular SQL statements are encountered.

- Every transaction must have an explicit end statement, which is either a COMMIT or ROLLBACK.

Characteristics specified by a SET TRANSACTION statement in SQL:

- **Access mode:** READ ONLY or READ WRITE. The default is READ WRITE unless the isolation level of READ UNCOMITTED is specified, in which case READ ONLY is assumed.
- **Diagnostic size** n, specifies an integer value n, indicating the number of conditions that can be held simultaneously in the diagnostic area. (To supply run-time feedback information to calling program for SQL statements executed in program)
-

Characteristics specified by a SET TRANSACTION statement in SQL (cont.):

- **Isolation level** <isolation>, where <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ or SERIALIZABLE. The default is SERIALIZABLE.

If all transactions is a schedule specify isolation level SERIALIZABLE, the interleaved execution of transactions will adhere to serializability. However, if any transaction in the schedule executes at a lower level, serializability may be violated

Potential problem with lower isolation levels:

- Dirty Read: Reading a value that was written by a transaction that failed.
- Nonrepeatable Read: Allowing another transaction to write a new value between multiple reads of one transaction.

A transaction T1 may read a given value from a table. If another transaction T2 later updates that value and then T1 reads that value again, T1 will see a different value. Example: T1 reads the No. of seats on a flight. Next, T2 updates that number (by reserving some seats). If T1 reads the No. of seats again, it will see a different value.

- **Phantoms**: New row inserted after another transaction accessing that row was started.

A transaction T1 may read a set of rows from a table (say EMP), based on some condition specified in the SQL WHERE clause (say DNO=5). Suppose a transaction T2 inserts a new EMP row whose DNO value is 5. T1 should see the new row (if equivalent serial order is T2; T1) or not see it (if T1; T2). The record that did not exist when T1 started is called a **phantom record**.

- **Phantoms**: New row inserted after another transaction accessing that row was started.

A transaction T1 may read a set of rows from a table (say EMP), based on some condition specified in the SQL WHERE clause (say DNO=5). Suppose a transaction T2 inserts a new EMP row whose DNO value is 5. T1 should see the new row (if equivalent serial order is T2; T1) or not see it (if

T1; T2). The record that did not exist when T1 started is called a **phantom record**.

Sample SQL transaction:

```
EXEC SQL whenever sqlerror go to UNDO;
EXEC SQL SET TRANSACTION
          READ WRITE
          DIAGNOSTICS SIZE 5
          ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT
          INTO EMPLOYEE (FNAME, LNAME, SSN, DNO, SALARY)
          VALUES ('Robert','Smith','991004321',2,35000);
EXEC SQL UPDATE EMPLOYEE
          SET SALARY = SALARY * 1.1
          WHERE DNO = 2;
EXEC SQL COMMIT;
GO TO  THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END:  …
```

Table 21.1 Possible Violations Based on Isolation Levels as Defined in SQL

	Type of Violation		
Isolation Level	**Dirty Read**	**Nonrepeatable Read**	**Phantom**
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

Database Concurrency Control

Purpose of Concurrency Control

- To enforce Isolation (through mutual exclusion) among conflicting transactions.
- To preserve database consistency through consistency preserving execution of transactions.
- To resolve read-write and write-write conflicts

Two-Phase Locking Techniques

- Locking is an operation which secures
    - (a) permission to Read
    - (b) permission to Write a data item for a transaction.

- Example:
  - Lock (X). Data item X is locked in behalf of the requesting transaction.
- Unlocking is an operation which removes these permissions from the data item.
- Example:
  - Unlock (X): Data item X is made available to all other transactions.
- Lock and Unlock are Atomic operations.

Two-Phase Locking Techniques: Essential components

- Two locks modes:
  - (a) shared (read) (b) exclusive (write).
- Shared mode: shared lock (X)
  - More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.
- Exclusive mode: Write lock (X)
  - Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.
- Conflict matrix

	Read	Write
Read	Y	N
Write	N	N

Two-Phase Locking Techniques: Essential components

- Lock Manager:
  - Managing locks on data items.
- Lock table:
  - Lock manager uses it to store the identify of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list.

Transaction ID	Data item id	lock mode	Ptr to next data item
T1	X1	Read	Next

  - Database requires that all transactions should be well-formed. A transaction is well-formed if:
    - It must lock the data item before it reads or writes to it.
    - It must not lock an already locked data items and it must not try to unlock a free data item.

The following code performs the lock operation:

```
B:      if LOCK (X) = 0 (*item is unlocked*)
        then LOCK (X) ← 1 (*lock the item*)
        else begin
            wait (until lock (X) = 0) and
            the lock manager wakes up the transaction);
        goto B
        end;
```

The following code performs the unlock operation:

```
LOCK (X) ← 0 (*unlock the item*)
if any transactions are waiting then wake up one of the waiting the
transactions;
```

The following code performs the read operation:

```
            B: if LOCK (X) = "unlocked" then
begin LOCK (X) ← "read-locked";
            no_of_reads (X) ← 1;
end
```

The following code performs the write lock operation:

```
            B: if LOCK (X) = "unlocked" then
begin LOCK (X) ← "read-locked";
            no_of_reads (X) ← 1;
end
else if LOCK (X) ← "read-locked" then
                no_of_reads (X) ← no_of_reads (X) +1
              else begin wait (until LOCK (X) = "unlocked" and
                  the lock manager wakes up the transaction);
                  go to B
                end;

else if LOCK (X) ← "read-locked" then
                no_of_reads (X) ← no_of_reads (X) +1
              else begin wait (until LOCK (X) = "unlocked" and
                  the lock manager wakes up the transaction);
                  go to B
                end;
```

The following code performs the unlock operation:

```
            if LOCK (X) = "write-locked" then
begin LOCK (X) ← "unlocked";
             wakes up one of the transactions, if any
end
else if LOCK (X) ← "read-locked" then
            begin
                no_of_reads (X) ← no_of_reads (X) -1
                if  no_of_reads (X) = 0 then
                begin
               LOCK (X) = "unlocked";
              wake up one of the transactions, if any
                end
            end;
```

Lock conversion

- Lock upgrade: existing read lock to write lock

  if Ti has a read-lock (X) and Tj has no read-lock (X) ($i \neq j$) then

  convert read-lock (X) to write-lock (X)

  else

  force Ti to wait until Tj unlocks X

- Lock downgrade: existing write lock to read lock

  Ti has a write-lock (X) (*no transaction can have any lock on X*)

  convert write-lock (X) to read-lock (X)

Two-Phase Locking Techniques: The algorithm

(a) Locking (Growing)

(b) Unlocking (Shrinking).

Locking (Growing) Phase:A transaction applies locks (read or write) on desired data items one at a time.

Unlocking (Shrinking) Phase: A transaction unlocks its locked data items one at a time.

Requirement: For a transaction these two phases must be mutually exclusively, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.

Two-Phase Locking Techniques: The algorithm

T1	T2	Result
read_lock (Y);	read_lock (X);	Initial values: X=20; Y=30
read_item (Y);	read_item (X);	Result of serial execution
unlock (Y);	unlock (X);	T1 followed by T2
write_lock (X);	Write_lock (Y);	X=50, Y=80.
read_item (X);	read_item (Y);	Result of serial execution
X:=X+Y;	Y:=X+Y;	T2 followed by T1
write_item (X);	write_item (Y);	X=70, Y=50
unlock (X);	unlock (Y);	

Two-Phase Locking Techniques: The algorithm

T1	T2	Result
read_lock (Y); read_item (Y); **unlock (Y);**		X=50; Y=50 Nonserializable because it. violated two-phase policy.
	read_lock (X); read_item (X); **unlock (X);** **write_lock (Y);** read_item (Y); Y:=X+Y; write_item (Y); unlock (Y);	
write_lock (X); read_item (X); X:=X+Y; write_item (X); unlock (X);		

Two-Phase Locking Techniques: The algorithm

T'1	T'2	
read_lock (Y); read_item (Y); write_lock (X); unlock (Y); read_item (X); X:=X+Y; write_item (X); unlock (X);	read_lock (X); read_item (X); Write_lock (Y); unlock (X); read_item (Y); Y:=X+Y; write_item (Y); unlock (Y);	T1 and T2 follow two-phase policy but they are subject to deadlock, which must be dealt with.

Two-phase policy generates two locking algorithms

Conservative: Prevents deadlock by locking all desired data items before transaction begins execution.

Basic: Transaction locks data items incrementally. This may cause deadlock which is dealt with

Strict:A more stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm.

Dealing with Deadlock and Starvation

- **Deadlock**

T'1	T'2	
read_lock (Y); read_item (Y);		T1 and T2 did follow two-phase policy but they are deadlock
	read_lock (X); read_item (Y);	
write_lock (X); (waits for X)		
	write_lock (Y); (waits for Y)	

- Deadlock (T'1 and T'2)

Dealing with Deadlock and Starvation

Deadlock prevention: A transaction locks all data items it refers to before it begins execution.

This way of locking prevents deadlock since a transaction never waits for a data item.

The conservative two-phase locking uses this approach.

Deadlock detection and resolution

- In this approach, deadlocks are allowed to happen. The scheduler maintains a wait-for-graph for detecting cycle. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back.
- A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like: Ti waits for Tj waits for Tk waits for Ti or Tj occurs, then this creates a cycle. One of the transaction o

Deadlock avoidance

- There are many variations of two-phase locking algorithm.
- Some avoid deadlock by not letting the cycle to complete.
- That is as soon as the algorithm discovers that blocking a transaction is likely to create a cycle, it rolls back the transaction.
- Wound-Wait and Wait-Die algorithms use timestamps to avoid deadlocks by rolling-back victim.

Starvation

- Starvation occurs when a particular transaction consistently waits or restarted and never gets a chance to proceed further.
- In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled-back.
- This limitation is inherent in all priority based scheduling mechanisms.
- In Wound-Wait scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.

Timestamp based concurrency control algorithm

- **Timestamp**
  - A monotonically increasing variable (integer) indicating the age of an operation or a transaction. A larger timestamp value indicates a more recent event or operation.
  - Timestamp based algorithm uses timestamp to serialize the execution of concurrent transactions.
  -

Basic Timestamp Ordering

1. Transaction T issues a write_item(X) operation:
   - If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then an younger transaction has already read the data item so abort and roll-back T and reject the operation.

- If the condition in part (a) does not exist, then execute write_item(X) of T and set write_TS(X) to TS(T).

2. Transaction T issues a read_item(X) operation:
   - If write_TS(X) > TS(T), then an younger transaction has already written to the data item so abort and roll-back T and reject the operation.
   - If write_TS(X) ≤ TS(T), then execute read_item(X) of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

Strict Timestamp Ordering

1. Transaction T issues a write_item(X) operation:
   - If TS(T) > read_TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).
2. Transaction T issues a read_item(X) operation:
   - If TS(T) > write_TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

Thomas's Write Rule

- If read_TS(X) > TS(T) then abort and roll-back T and reject the operation.
- If write_TS(X) > TS(T), then just ignore the write operation and continue execution. This is because the most recent writes counts in case of two consecutive writes.
- If the conditions given in 1 and 2 above do not occur, then execute write_item(X) of T and set write_TS(X) to TS(T).

Multiversion Two-Phase Locking Using Certify Locks

- Steps
  1. X is the committed version of a data item.
  2. T creates a second version X' after obtaining a write lock on X.
  3. Other transactions continue to read X.
  4. T is ready to commit so it obtains a certify lock on X'.
  5. The committed version X becomes X'.
  6. T releases its certify lock on X', which is X now.

Compatibility tables for

	Read	Write
Read	yes	no
Write	no	no

	Read	Write	Certify
Read	yes	no	no
Write	no	no	no
Certify	no	no	no

Validation (Optimistic) Concurrency Control Schemes

- In this technique only at the time of commit serializability is checked and transactions are aborted in case of non-serializable schedules.

- Three phases:
  1. Read phase 2. Validation phase 3. Write phase

1. Read phase:
A transaction can read values of committed data items. However, updates are applied only to local copies (versions) of the data items (in database cache).

2. Validation phase: Serializability is checked before transactions write their updates to the database.

- This phase for Ti checks that, for each transaction Tj that is either committed or is in its validation phase, one of the following conditions holds:
  - Tj completes its write phase before Ti starts its read phase.
  - Ti starts its write phase after Tj completes its write phase, and the read_set of Ti has no items in common with the write_set of Tj
  - Both the read_set and write_set of Ti have no items in common with the write_set of Tj, and Tj completes its read phase.
  - When validating Ti, the first condition is checked first for each transaction Tj, since (1) is the simplest condition to check. If (1) is false then (2) is checked and if (2) is false then (3) is checked. If none of these conditions holds, the validation fails and Ti is aborted.

3.Write phase: On a successful validation transactions' updates are applied to the database; otherwise, transactions are restarted

Granularity of data items and Multiple Granularity Locking

- These locks are applied using the following compatibility matrix:

	IS	IX	S	SIX	X
IS	yes	yes	yes	yes	no
IX	yes	yes	no	no	no
S	yes	no	yes	no	no
SIX	yes	no	no	no	no
X	no	no	no	no	no

Intention-shared (IS Intention-exclusive (IX) Shared-intention-exclusive (SIX)

Granularity of data items and Multiple Granularity Locking
The set of rules which must be followed for producing serializable schedule are

1. The lock compatibility must adhered to.
2. The root of the tree must be locked first, in any mode..
3. A node N can be locked by a transaction T in S or IX mode only if the parent node is already locked by T in either IS or IX mode.

4. A node N can be locked by T in X, IX, or SIX mode only if the parent of N is already locked by T in either IX or SIX mode.
5. T can lock a node only if it has not unlocked any node (to enforce 2PL policy).
6. T can unlock a node, N, only if none of the children of N are currently locked by T.

Granularity of data items and Multiple Granularity Locking: An example of a serializable execution:

T1	T2	T3
IX(db)		
IX(f1)		
	IX(db)	
		IS(db)
		IS(f1)
		IS(p11)
IX(p11)		
X(r111)		
	IX(f1)	
	X(p12)	
		S(r11j)
IX(f2)		
IX(p21)		
IX(r211)		
Unlock (r211)		
Unlock (p21)		
Unlock (f2)		
		S(f2)

Database Recovery Techniques

- To bring the database into a consistent state after a failure occurs.
- To ensure the transaction properties of Atomicity (a transaction must be done in its entirety; otherwise, it has to be rolled back) and Durability (a committed transaction cannot be canceled and all its updates must be applied permanently to the database).
- After a failure, the DBMS recovery manager is responsible for bringing the system into a consistent state before transactions can resume

Types of Failure

- **Transaction failure**: Transactions may fail because of errors, incorrect input, deadlock, incorrect synchronization, etc.

- **System failure:** System may fail because of application error, operating system fault, RAM failure, etc.

- **Media failure:** Disk head crash, power disruption, etc.

The Log File

- Holds the information that is necessary for the recovery process
- Records all relevant operations in the order in which they occur (looks like a *schedule of transactions*, see Chapter 21)
- Is an append-only file.
- Holds various types of log records (or log entries).
-

Types of records (entries) in log file:

- [start_transaction,T]: Records that transaction T has started execution.
- [write_item,T,X,old_value,new_value]: T has changed the value of item X from old_value to new_value.
- [read_item,T,X]: T has read the value of item X (not needed in many cases).
- [end_transaction,T]: T has ended execution
- [commit,T]: T has completed successfully, and committed.
- [abort,T]: T has been aborted.

For **write_item** log entry, *old value* of item before modification (**BFIM** - BeFore Image) and the *new value* after modification (**AFIM** – AFter Image) are stored. BFIM needed for UNDO, AFIM needed for REDO. A sample log is given below. **Back P** and **Next P** point to the previous and next log records of the same transaction.

T ID	Back P	Next P	Operation	Data item	BFIM	AFIM
T1	0	1	Begin			
T1	1	4	Write	X	X = 100	X = 200
T2	0	8	Begin			
T1	2	5	W	Y	Y = 50	Y = 100
T1	4	7	R	M	M = 200	M = 200
T3	0	9	R	N	N = 400	N = 400
T1	5	nil	End			

Database Cache:
Database Cache: A set of *main memory* buffers; each buffer typically holds contents of one disk block. Stores the disk blocks that contain the data items being read and written by the database transactions.

Data Item Address: (disk block address, offset, size in bytes).

Cache Table: Table of entries of the form (buffer addr, disk block addr, modified bit, pin/unpin bit, ...) to indicate which disk blocks are currently in the cache buffers.

Data items to be modified are first copied into database cache by the Cache Manager (CM) and after modification they are flushed (written) back to the disk. The flushing is controlled by **Modified** and **Pin-Unpin** bits.

Pin-Unpin: If a buffer is pinned, it cannot be written back to disk until it is unpinned.

Modified: Indicates that one or more data items in the buffer have been changed.

Data Update:

- **Immediate Update**: A data item modified in cache can be written back to disk *before the transaction commits*.
- **Deferred Update**: A modified data item in the cache cannot be written back to disk till *after the transaction commits* (buffer is **pinned**).
- **Shadow update**: The modified version of a data item does not overwrite its disk copy but is written at a separate disk location (new version).
- **In-place update**: The disk version of the data item is overwritten by the cache version
-

UNDO and REDO Recovery Actions
To maintain atomicity and durability, some transaction's may have their operations **redone** or **undone** during recovery. UNDO (roll-back) is needed for transactions that are not committed yet. REDO (roll-forward) is needed for committed transactions whose writes may have not yet been flushed from cache to disk.

Undo: Restore all BFIMs from log to database on disk. UNDO proceeds backward in log (from most recent to oldest UNDO).

Redo: Restore all AFIMs from log to database on disk. REDO proceeds forward in log (from oldest to most recent REDO).

Write-ahead Logging Protocol
The information needed for recovery must be written to the log file on disk before changes are made to the database on disk. **Write-Ahead Logging** (WAL) protocol consists of two rules:

For Undo: Before a data item's AFIM is flushed to the database on disk (overwriting the BFIM) its BFIM must be written to the log and the log must be saved to disk.

For Redo: Before a transaction executes its commit operation, all its AFIMs must be written to the log and the log must be saved on a stable store.

Checkpointing
To minimize the **REDO** operations during recovery.
The following steps define a checkpoint operation:

- Suspend execution of transactions temporarily.
- Force write modified buffers from cache to disk.
- Write a [checkpoint] record to the log, save the log to disk. This record also includes other info., such as the *list of active transactions* at the time of checkpoint.
- Resume normal transaction execution.

During recovery **redo** is required only for transactions that have committed *after the last [checkpoint] record* in the log.

Other Database Recovery Concepts
Steal/No-Steal and Force/No-Force
Specify how to flush database cache buffers to database on disk:

Steal: Cache buffers updated by a transaction may be flushed to disk before the transaction commits (recovery may require UNDO).

No-Steal: Cache buffers cannot be flushed until after transaction commit (NO-UNDO). (Buffers are *pinned* till transactions commit).

Force: Cache is flushed (forced) to disk before transaction commits (NO-REDO).

No-Force: Some cache flushing may be deferred till after transaction commits (recovery may require REDO).
These give rise to four different ways for handling recovery:
Steal/No-Force (Undo/Redo),
Steal/Force (Undo/No-redo),
No-Steal/No-Force (Redo/No-undo),
No-Steal/Force (No-undo/No-redo).

Deferred Update (NO-UNDO/REDO) Recovery Protocol
System must impose **NO-STEAL** rule. Recovery subsystem analyzes the log, and creates two lists:

Active Transaction list: All active (uncommitted) transaction ids are entered in this list.

Committed Transaction list: Transactions committed after the last checkpoint are entered in this table.

During recovery, transactions in **commit** list are **redone**; transactions in **active** list are *ignored* (because of NO-STEAL rule, none of their writes have been applied to the database on disk). Some transactions may be **redone** twice; this does not create inconsistency because **REDO** is "**idempotent**", that is, one REDO for an AFIM is equivalent to multiple REDO for the same AFIM.

Advantage: Only **REDO** is needed during recovery.

Disadvantage: Many buffers may be pinned while waiting for transactions that updated them to commit, so system may run out of cache buffers when requests are made by new transactions.

UNDO/NO-REDO Recovery Protocol
In this method, **FORCE** rule is imposed by system (AFIMs of a transaction are flushed to the database on disk under Write Ahead Logging *before the transaction commits*).

Transactions in active list are **undone**; transactions in committed list are ignored (because based on FORCE rule, all their changes are already written to the database on disk).

Advantage: During recovery, only **UNDO** is needed.
Disadvantages:

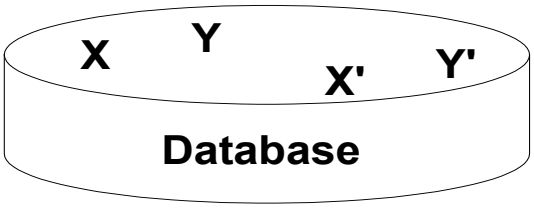
1. Commit of a transaction is delayed until all its changes are force-written to disk.
2. Some buffers may be written to disk multiple times if they are updated by several transactions.

Recovery can require both UNDO of some transactions and REDO of other transactions (Corresponds to STEAL/NO-FORCE). Used most often in practice because of disadvantages of the other two methods. To minimize REDO, checkpointing is used.

The recovery performs:

1. **Undo** of a transaction if it is in the active transaction list.
2. **Redo** of a transaction if it is in the list of transactions that committed since the last checkpoint.

Shadow Paging (NO-UNDO/NO-REDO)
The AFIM does not overwrite its BFIM but is recorded at another place (new version) on the disk. Thus, a data item can have AFIM and BFIM (Shadow copy of the data item) at two different places on the disk



X and Y: Shadow (old) copies of data items
X` and Y`: Current (new) copies of data items

ARIES Database Recovery
Used in practice, it is based on:

1. WAL (Write Ahead Logging)
2. Repeating history during redo: ARIES will retrace all actions of the database system prior to the crash to reconstruct the correct database state.
3. Logging changes during undo: It will prevent ARIES from repeating the completed undo operations if a failure occurs during recovery, which causes a restart of the recovery process.

The ARIES recovery algorithm consists of three steps:

1. **Analysis**: step identifies the dirty (updated) page buffers in the cache and the set of transactions active at the time of crash. The set of transactions that committed after the last checkpoint is determined, and the appropriate point in the log where redo is to start is also determined.

2. **Redo**: necessary redo operations are applied.

3. **Undo**: log is scanned backwards and the operations of transactions active at the time of crash are undone in reverse order.

The Log and Log Sequence Number (LSN)
A log record is written for (a) data update (Write), (b) transaction commit, (c) transaction abort, (d) undo, and (e) transaction end. In the case of undo a *compensating* log record is written.

A **unique LSN** is associated with every log record. LSN increases monotonically and indicates the disk address of the log record it is associated with. In addition, each data page stores the LSN of the latest log record corresponding to a change for that page.

A log record stores:

1. LSN of previous log record for same transaction: It links the log records of each transaction.
2. Transaction ID.
3. Type of log record.

For a write operation, additional information is logged:

1. Page ID for the page that includes the item
2. Length of the updated item
3. Its offset from the beginning of the page
4. BFIM of the item
5. AFIM of the item
6.

The Transaction table and the Dirty Page table

For efficient recovery, the following tables are also stored in the log during checkpointing:

Transaction table: Contains an entry for each active transaction, with information such as transaction ID, transaction status, and the LSN of the most recent log record for the transaction.

Dirty Page table: Contains an entry for each dirty page (buffer) in the cache, which includes the page ID and the LSN corresponding to the earliest update to that page.

"Fuzzy" Checkpointing

A checkpointing process does the following:

1. Writes a *begin_checkpoint* record in the log, then forces updated (dirty) buffers to disk.
2. Writes an *end_checkpoint* record in the log, along with the contents of transaction table and dirty page table.
3. Writes the LSN of the *begin_checkpoint* record to a special file. This special file is accessed during recovery to locate the last checkpoint information.

To allow the system to continue to execute transactions, ARIES uses "fuzzy checkpointing".

The following steps are performed for recovery

1. **Analysis phase**: Start at the begin_checkpoint record and proceed to the end_checkpoint record. Access transaction table and dirty page table that were appended to the log. During this phase some other records may be written to the log and the transaction table may be modified. The analysis phase compiles the set of redo and undo operations to be performed and ends.

2. **Redo phase**: Starts from the point in the log where all dirty pages have been flushed, and move forward. Operations of committed transactions are redone.

3. **Undo phase**: Starts from the end of the log and proceeds backward while performing appropriate undo. For each undo it writes a compensating record in the log.

The recovery completes at the end of undo phase.

(a)

Lsn	Last_lsn	Tran_id	Type	Page_id	Other_information
1	0	T_1	update	C	. . .
2	0	T_2	update	B	. . .
3	1	T_1	commit		. . .
4	begin checkpoint				
5	end checkpoint				
6	0	T_3	update	A	. . .
7	2	T_2	update	C	. . .
8	7	T_2	commit		. . .

(b)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	2	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2

(c)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	8	commit
T_3	6	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2
A	6

Figure 23.5
An example of recovery in ARIES. (a) The log at point of crash. (b) The Transaction and Dirty Page Tables at time of checkpoint. (c) The Transaction and Dirty Page Tables after the analysis phase.

Recovery in Multi-database Transactions (Two-phase commit)

A multidatabase transaction can access several databases: e.g. airline database, car rental database, credit card database. The transaction commits only when all these multiple databases agree to commit individually the part of the transaction they were executing. This commit scheme is referred to as "*two-phase commit*" (2PC). If any one of these nodes fails or cannot commit its part of the transaction, then the whole transaction is aborted. Each node recovers the transaction under its own recovery protocol.

Phase 1: Coordinator (usually application program running in middle-tier of 3-tier architecture) sends "Ready-to-commit?" query to each participating database, then waits for replies. A participating database replies Ready-to-commit only after saving all actions in its local log on disk.

Phase 2: If coordinator receives Ready-to-commit signals from all participating databases, it sends Commit to all; otherwise, it send Abort to all.
This protocol can survive most types of crashes.