MODULE VI

OVERVIEW OF CONCURRENCY CONTROL AND RECOVERY

- A DBMS is **single-user** if at most one user at a time can use the system, and it is multiuser if many users can use the system-and hence access the database-concurrently.
- Single-user DBMSs are mostly restricted to personal computer systems;
- Most other DBMSs are multiuser. For example, an airline reservations system is used by hundreds of travel agents and reservation clerks concurrently. Systems in banks, insurance agencies, stock exchanges, supermarkets, and the like are also operated on by many users who submit transactions **concurrently** to the system.
- Multiple users can access databases-and use computer systems-simultaneously because of the concept of **multiprogramming**, which allows the computer to execute multiple programs-or processes-at the same time.

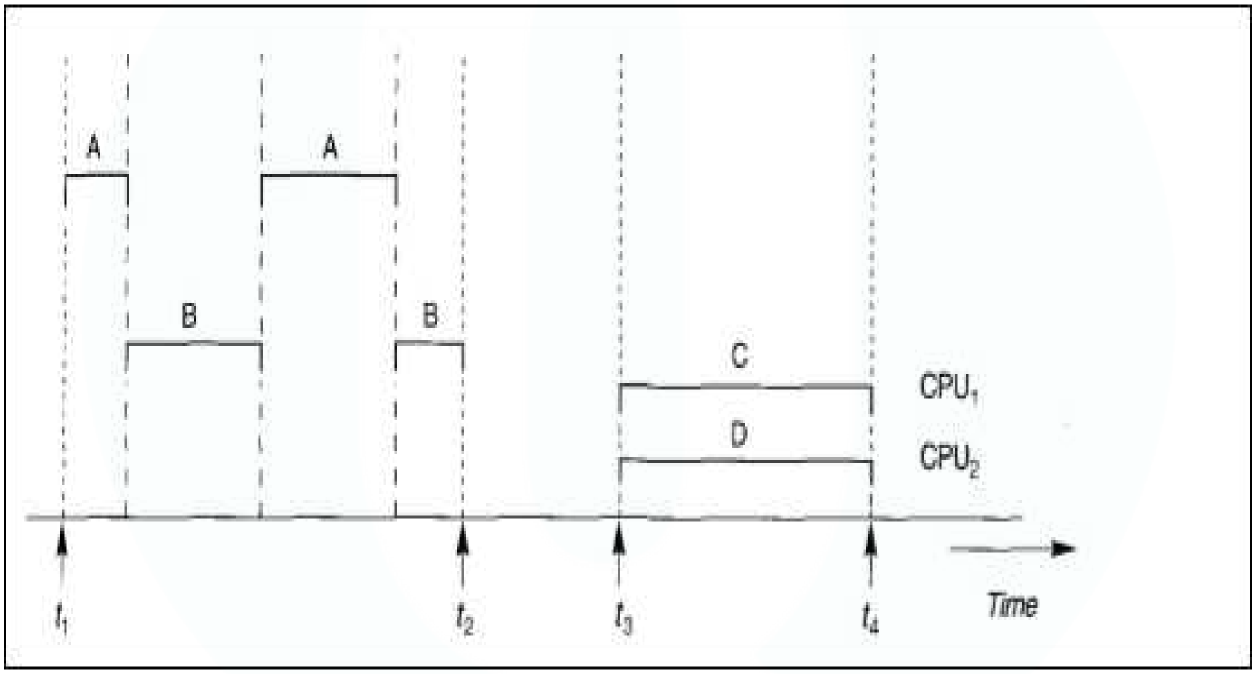


FIGURE 17.1 Interleaved processing versus parallel processing of concurrent transactions.

- However, multiprogramming operating systems execute some commands from one process, then suspend that process and execute some commands from the next process, and so on.

- A process is resumed at the point where it was suspended whenever it gets its turn to use the CPU again. Hence, concurrent execution of processes is actually **interleaved.**
- If the computer system has multiple hardware processors (crus), **parallel processing** of multiple processes is possible, as illustrated by processes C and D in Figure 17.1.

Transaction

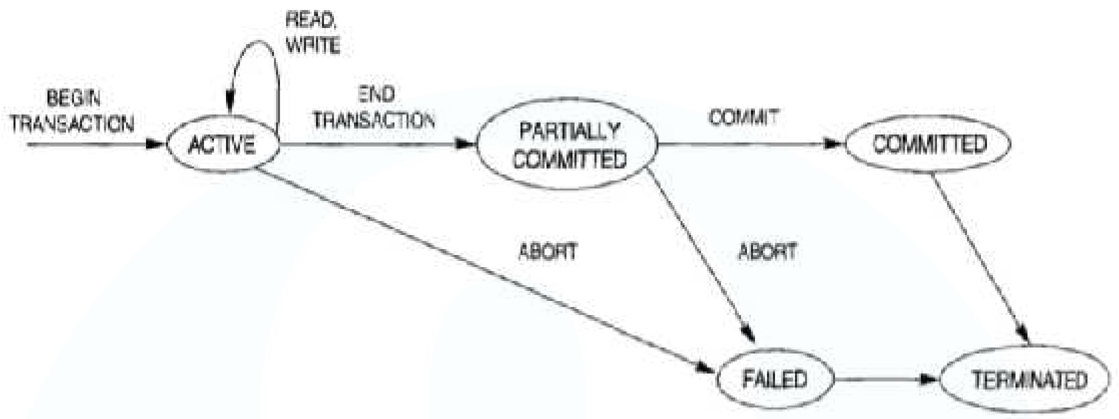


FIGURE 17.4 State transition diagram illustrating the states for transaction execution.

- A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all.
- For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts .
- Hence, the 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 of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability .

• **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.

State transition diagram shown above describes how a transaction moves through its execution states.

(a)A transaction goes into an **active state** immediately after it starts execution, where it can issue READ and WRITE operations.

(b)When the transaction ends, it moves to the **partially committed state**.

(c)At this point, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently .

(d)Once this check is successful, the transaction is said to have reached its **commit point** and enters the **committed state**.

(e)Once a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database.

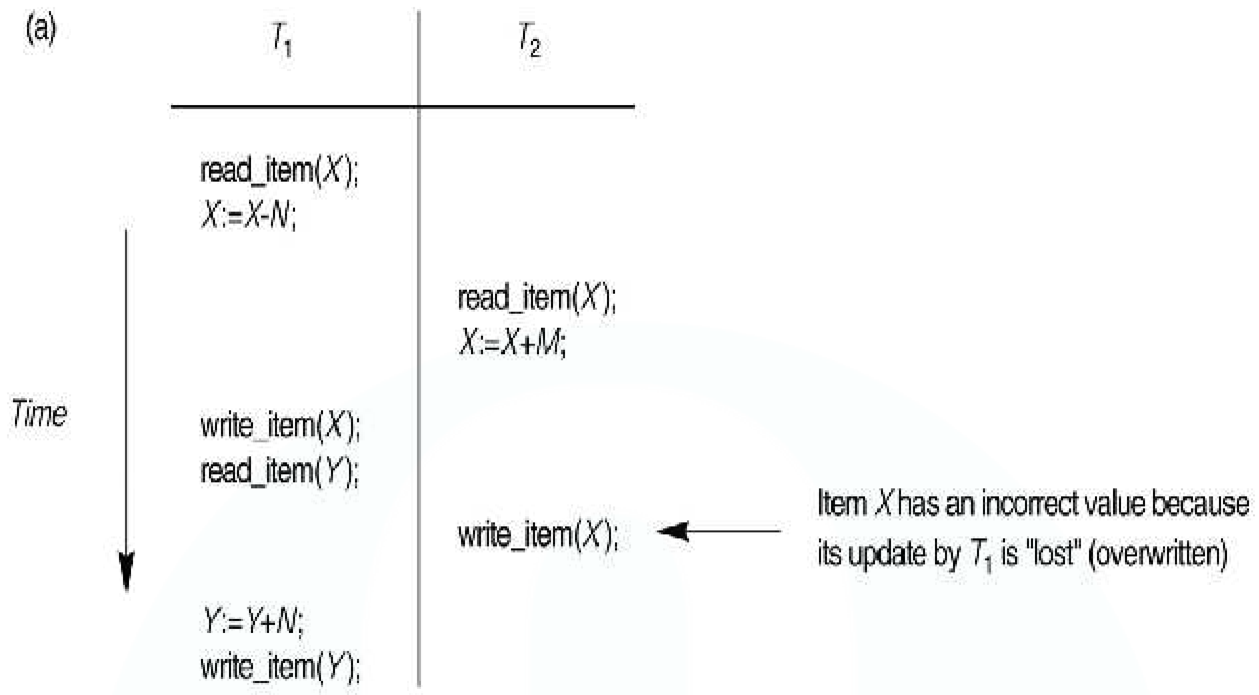
(f)However, a transaction can go to the **failed state** if one of the **checks fails** or if the transaction is **aborted during its active state**.

(g)The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database.

(h)The **terminated state** corresponds to the transaction leaving the system.

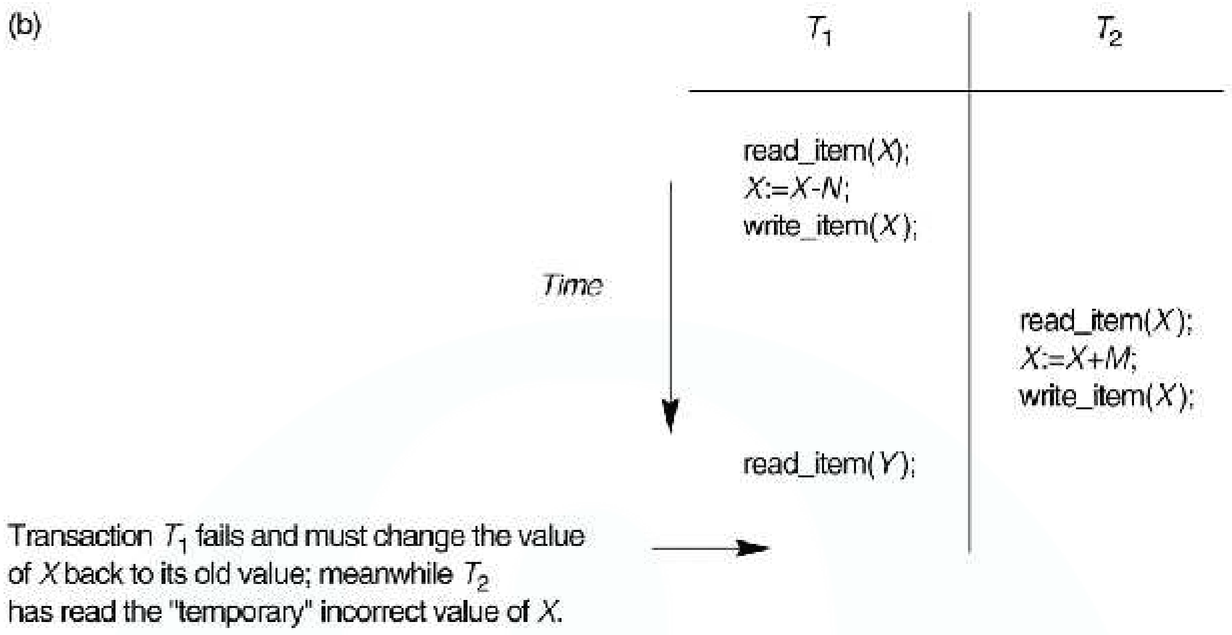
Why Concurrency Control Is Needed

(a)The Lost Update Problem.



- This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect.
- Suppose that transactions T1 and T2 are submitted at approximately the same time, and suppose that their operations are interleaved as shown in Figure; then the final value of item X is incorrect, because T2 reads the value of X before T1 changes it in the database, and hence the updated value resulting from T1 is lost.
- For example, if X = 80 at the start (originally there were 80 reservations on the flight), N = 5 (T1 transfers 5 seat reservations from the flight corresponding to X to the flight corresponding to Y), and M = 4 (T2 reserves 4 seats on X), the final result should be X = 79; but in the interleaving of operations shown in Figure , it is X = 84 because the update in T 1 that removed the five seats from X was lost.

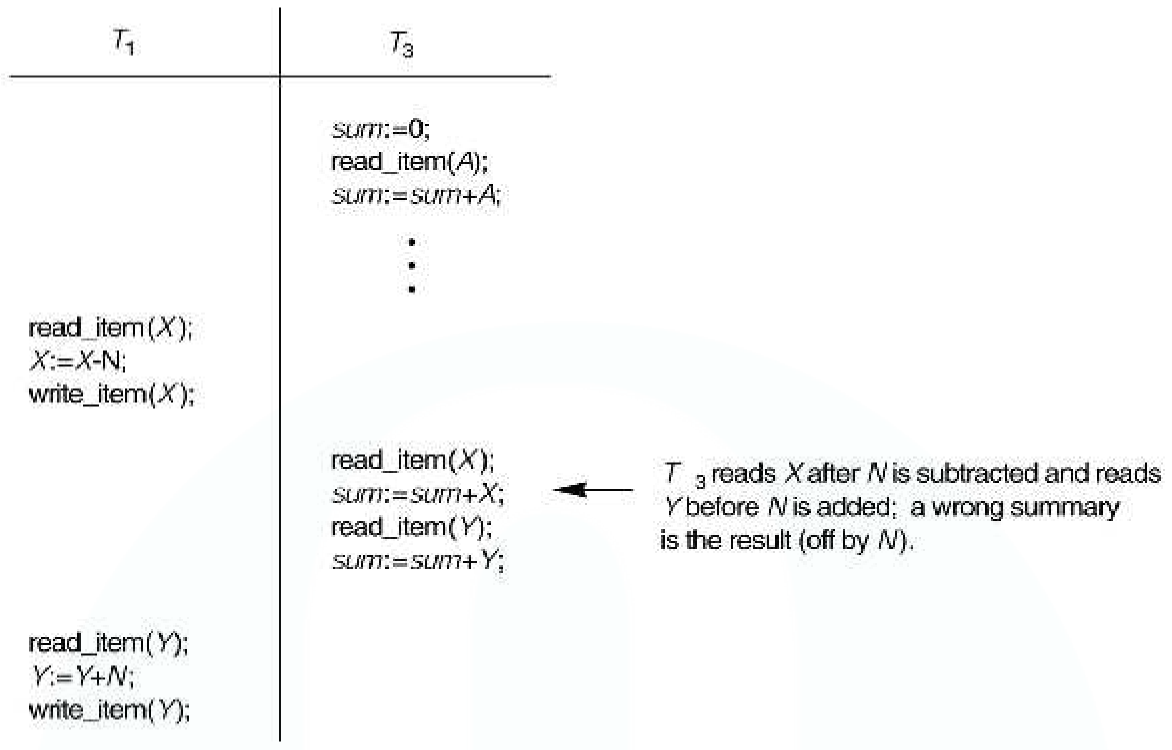
(b)The Temporary Update (or Dirty Read) Problem.


(b)
T_1
T_2
read_item(X);
X:=X-N;
write_item(X);
Time
read_item(X);
X:=X+M;
write_item(X);
read_item(Y);
Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the "temporary" incorrect value of X.

- This problem 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.
- Figure shows an example where T1 updates item X and then fails before completion, so the system **must change X back to its original value**. Before it can do so, however, transaction T2 reads the "temporary" value of X, which will not be recorded permanently in the database because of the failure of T1 .
- The value of item X that is read by T2 is called **dirty data**, because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the **dirty read problem.**

(c)The Incorrect Summary Problem.

(c)



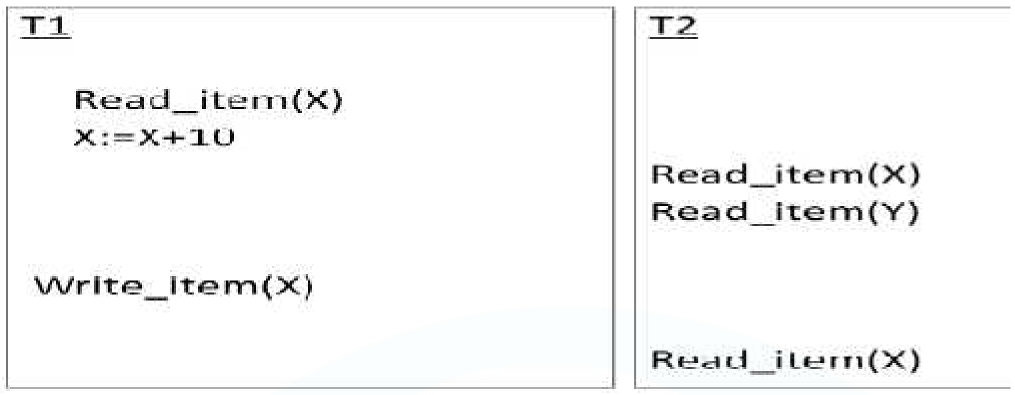
```
T1                          | T3
----------------------------+----------------------------
                            | sum:=0;
                            | read_item(A);
                            | sum:=sum+A;
                            |     .
                            |     .
                            |     .
read_item(X);               |
X:=X-N;                     |
write_item(X);              |
                            | read_item(X);
                            | sum:=sum+X;
                            | read_item(Y);
                            | sum:=sum+Y;
read_item(Y);               |
Y:=Y+N;                     |
write_item(Y);              |
```

T 3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

- 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.
- For example, suppose that a transaction T3 is calculating the total number of reservations on all the flights; meanwhile, transaction T1 is executing. If the interleaving of operations shown in Figure occurs, the result of T3 will be off by an amount N because T3 reads the value of X after N seats have been subtracted from it but reads the value of Y before those N seats have been added to it.

(d) **Unrepeatable read**

- Another problem that may occur is called unrepeatable read, where a transaction T reads an item twice and the item is changed by another transaction T' between the two reads.
- Hence, T receives **different values for its two reads of the same item**.
- This may occur, for example, if during an airline reservation transaction, a customer is inquiring about seat availability on several flights.
- When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation.



T1	T2
Read_item(X)	
X:=X+10	
	Read_item(X)
	Read_item(Y)
Write_item(X)	
	Read_item(X)

Why Recovery Is Needed

- Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either

(1) all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or

(2) the transaction has no effect whatsoever on the database or on any other transactions.

- The DBMS must not permit some operations of a transaction T to be applied to the database.
- This may happen if a transaction fails after executing some ofits operations but before executing all of them.

ACID PROPERTIES

Transactions should possess several properties. These are often called the ACID properties:

1. **A**tomicity: *A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all*. If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database.

2. **C**onsistency preservation: *A transaction is consistency preserving if its complete execution rakejs) the database from one consistent state to another*. A **consistent state** of the database satisfies the constraints specified in the schema as well as any other constraints that should hold on the database. A database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the **complete** execution of the transaction, assuming that **no interference with other transactions** occurs.

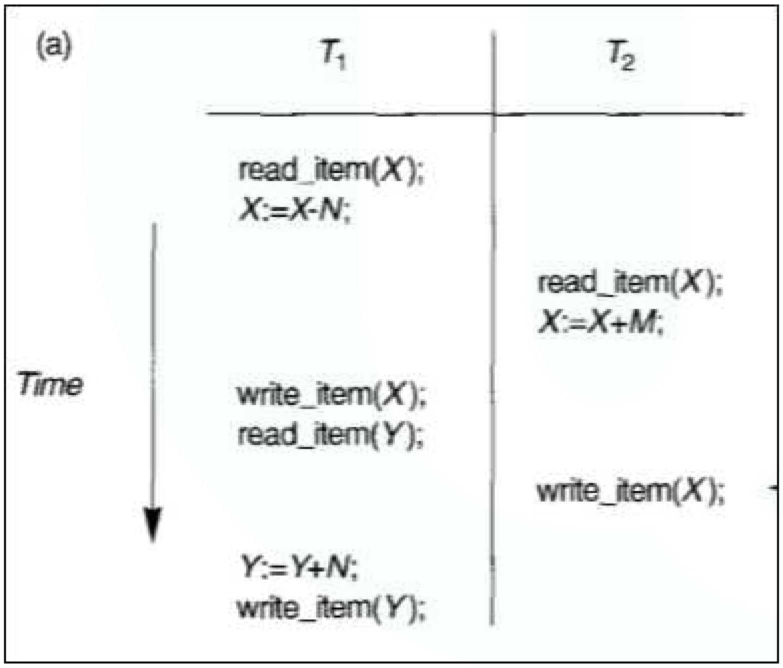
3. **I**solation: *A transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.* If every transaction does not make its updates visible to other transactions until it is committed, oneform of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks

4. **D**urability or permanency: *The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.*

SERIAL AND CONCURRENT SCHEDULES

Schedules (Histories) of Transactions

- A **schedule (or history)** S of n transactions T1 , T2, ... , Tn is an ordering of the operations of the transactions subject to the constraint that, for each transaction T, that participates in S, the operations of T, in S must appear in the same order in which they occur in T.



The **schedule** of corresponding transaction shown above:

$$S_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$$

Here , r1(x) denotes reading an item X from Transaction 1(T1)

- Once a transaction T is committed, it should never be necessary to roll back T.
- The schedules that theoretically meet this criterion are called recoverable schedules and those that do not are called non recoverable, and hence should not be permitted.
- 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.** (i.e: if T1,T2,T3…. are the transactions in order then T3 can be commited only after transactions T1,T2 commits their operations)
- A transaction T reads from transaction T' in a schedule S if **some item X is first written by T'** and **later read by T.**
- In addition, T' should not have been aborted before T reads item X, and there should be no transactions that write X after T' writes it and before T reads it.

Consider the schedule Sa' given below, two commit operations have been added

$$S_a': r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); c_2; w_1(Y); c_1;$$

Sa': r1(x); r2(x); w1(x); r1(y); w2(x); c2; w1(y); c1;

$$S_c: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_2; a_1;$$

$$S_d: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2;$$

$$S_e: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); a_1; a_2;$$

Sa', Sc is an example for non recoverable schedule

Sd is an example for recoverable schedule

Se is an example for cascading rollback

- Sc is **not recoverable**, because T2 reads item X from T1, and then **T2 commits before T1 commits**. If T1 aborts after the c2 operation in Sc then the value of X that T2 read is no longer valid and T2 must be aborted after it had been committed, leading to a schedule that is not recoverable.
- For the schedule to be recoverable, the C2 operation in Sc must be postponed until after T1 commits, as shown in Sd; if T1 aborts instead of committing, then T2 should also abort as shown in Se because the value of X it read is no longer valid.
- In a recoverable schedule, no committed transaction ever needs to be rolled back. However, it is possible for a phenomenon known as **cascading rollback** (or cascading

abort) to occur, where an uncommitted transaction has to be rolled back because it read an item from a transaction that failed. This is illustrated in schedule Se where transaction T2 has to be rolled back because it read item X from T1 , and T1 then aborted.

- A schedule is said to be **cascadeless, or to avoid cascading rollback**, if every transaction in the schedule reads only items that were written by committed transactions.
- In this case, all items read will not be discarded, so no cascading rollback will occur.
- Finally, there is a third, more restrictive type of schedule, called a **strict schedule**, in which transactions can neither read nor write an item X until the last transaction that wrote X has committed (or aborted).

Serial , Non-serial

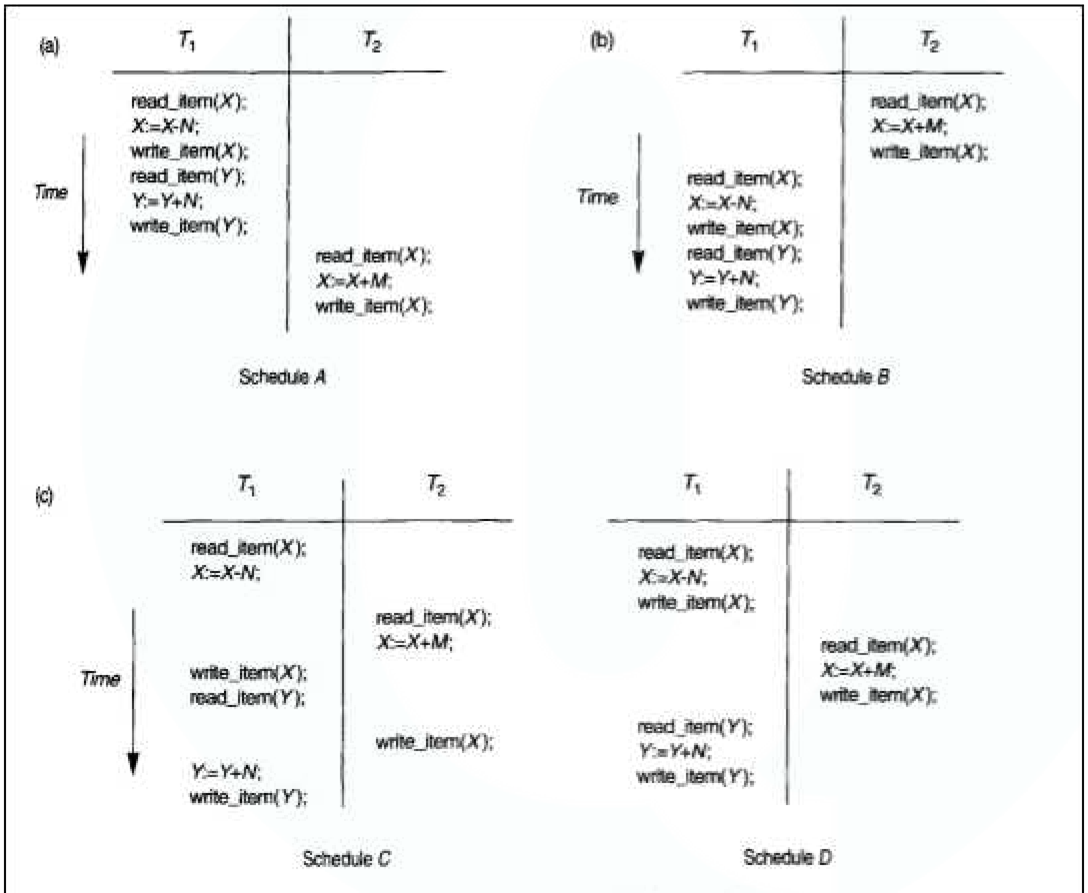
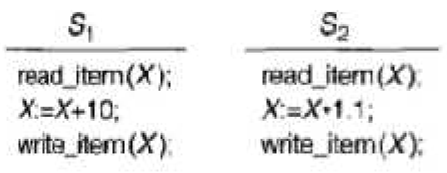


FIGURE 17.5 Examples of serial and nonserial schedules involving transactions T_1 and T_2. (a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1. (c) Two nonserial schedules C and D with interleaving of operations.

- Schedules A and B in Figure 17.5a and b are called **serial** because the operations of each transaction are executed consecutively, without any interleaved operations from the other transaction.
- In a serial schedule, entire transactions are performed in serial order: T1 and then T2 in Figure 17.5a, and T2 and then T1 in Figure 17.5b.
- Schedules C and D in Figure 17.5c are called **non-serial** because each sequence interleaves operations from the two transactions.
- Formally, a schedule S is **serial** if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule; otherwise, the schedule is called **non serial.**
- Hence, in a serial schedule, only one transaction at a time is active-the commit (or abort) of the active transaction initiates execution of the next transaction.
- The **problem** with serial schedules is that they **limit concurrency** or interleaving of operations. In a serial schedule, if a transaction waits for an I/O operation to complete, we cannot switch the CPU processor to another transaction, **thus wasting valuable CPU processing time.**
- In addition**, if some transaction T is quite long, the other transactions must wait for T to complete all its operations** before commencing.
- A schedule S of n transactions is **serializable** if it is *equivalent to some serial schedule* of the same n transactions.
- We can form two disjoint groups of the non serial schedules: those that are equivalent to one (or more) of the serial schedules, and hence are **serializable;** and those that are not equivalent to any serial schedule and hence are **not serializable.**
- Two schedules are called **result equivalent** if they produce the same final state of the database.
- For two schedules to be equivalent, the operations applied to each data item affected by the schedules should be applied to that item in both schedules in the ***same order***.
- However, two different schedules may accidentally produce the same final state. For example, in Figure 17.6, schedules S1 and S2 will produce the same final database state if they execute on a database with an initial value of X = 100; but for other initial values of X, the schedules are **not result equivalent.(since operations are different in both transactions)**



S_1	S_2
read_item(X);	read_item(X);
X:=X+10;	X:=X*1.1;
write_item(X);	write_item(X);

FIGURE 17.6 Two schedules that are result equivalent for the initial value of X = 100 but are not result equivalent in general.

<u>CONFLICT SERIALIZABILITY</u>

- Two definitions of equivalence of schedules are generally used:

(a) conflict equivalence
(b) view equivalence.
But only conflict equivalence is important.

- Two schedules are said to be **conflict equivalent** if the order of **any two conflicting operations is the same in both schedules.**
- If two conflicting operations are applied in different orders in two schedules, the effect can be different on the database or on other transactions in the schedule, and hence the schedules are not conflict equivalent.

<u>wo operations in a schedule are said to **conflict if they satisfy all three of the following conditions::**</u>

- **If they belong to different transactions,**
- **They access the same database item say,X,**
- **At least one of the two operations is a write_item operation.**

- For example, if a read and write operation occur in the order r l (X), w2(X) in schedule S1 and in the reverse order w2(X), r1(X) in schedule S2, the value read by r1(X) can be different in the two schedules.
- Similarly, if two write operations occur in the order w1(X), w2(X) in S1 and in the reverse order w2(X), w1(X) in S2 the next r(X) operation in the two schedules will read potentially different values; or if these are the last operations writing item X in the schedules, the final value of item X in the database will be different.

Testing of conflict serializability

- Consider the schedule:
  S : R1(x) R2(x) W1(x) R1(y) W2(x) W1(y)

  (a) Conflicting operations in order from schedule S:

  R1(x) W2(x)

  R2(x) W1(x)

  W1(x) W2(x)

  (b)We can consider a serial schedule (T1 then T2) from schedule S

  S1 : T1 T2

  R1(x) W1(x) R1(y) W1(y) R2(x) W2(x)

  (c) Conflicting operations in order from serial schedule S1:

  R1(x) W2(x)

  W1(x) R2(x)

  W1(x) W2(x)

(d)Comparing (a) & (c) we observe that the conflicting operations W1(x) R2(x) is not in order .

Hence not conflict serializable.

If all the conflicting operations in both (a) & (b) are in order, then it is conflict serializable.

Testing for Conflict Serializability of a Schedule using precedence graph

- Algorithm below can be used to test a schedule for conflict serializability.
- The algorithm looks at only the read_item and write_item operations in a schedule to construct a **precedence graph (or serialization graph),** which is a directed graph G = (N, E) that consists of a set of nodes N ={T1, T2, ... , Tn} and a set of directed edges E ={e1,e2 ... , em}.

Algorithm

1. For each transaction Ti participating in schedule S, create a node labeled Ti in the precedence graph.

i.e : No.of nodes =No.of transactions

2.For each case in S where Tj executes a read_item(X) after Ti, executes a write_ item(X), create an edge (Ti →Tj) in the precedence graph.

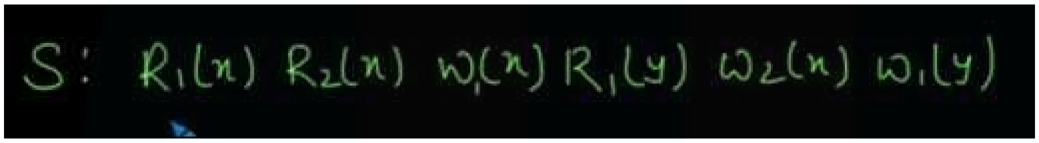
For each case in S where Tj executes a write_item(X) after Ti, executes a read_ item(X) , create an edge (Ti →Tj) in the precedence graph.

For each case in S where Tj executes a write_item(X) after Ti executes a write_ item(X), create an edge (Ti →Tj) in the precedence graph.

i.e Directed edge i❼j for each **conflicting operatings** Oi❼Oj

3. The schedule S is serializable if and only if the precedence graph has no cycles.

i.e If the graph contain a cycle then the schedule is not conflict serializable.



S: $R_1(x)\ R_2(x)\ W_1(x)\ R_1(y)\ W_2(x)\ W_1(y)$

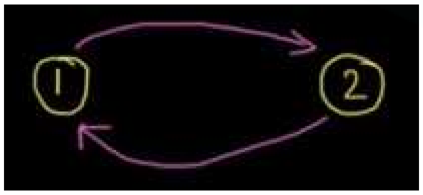
No.of transactions = 2

Conflicting operations include:

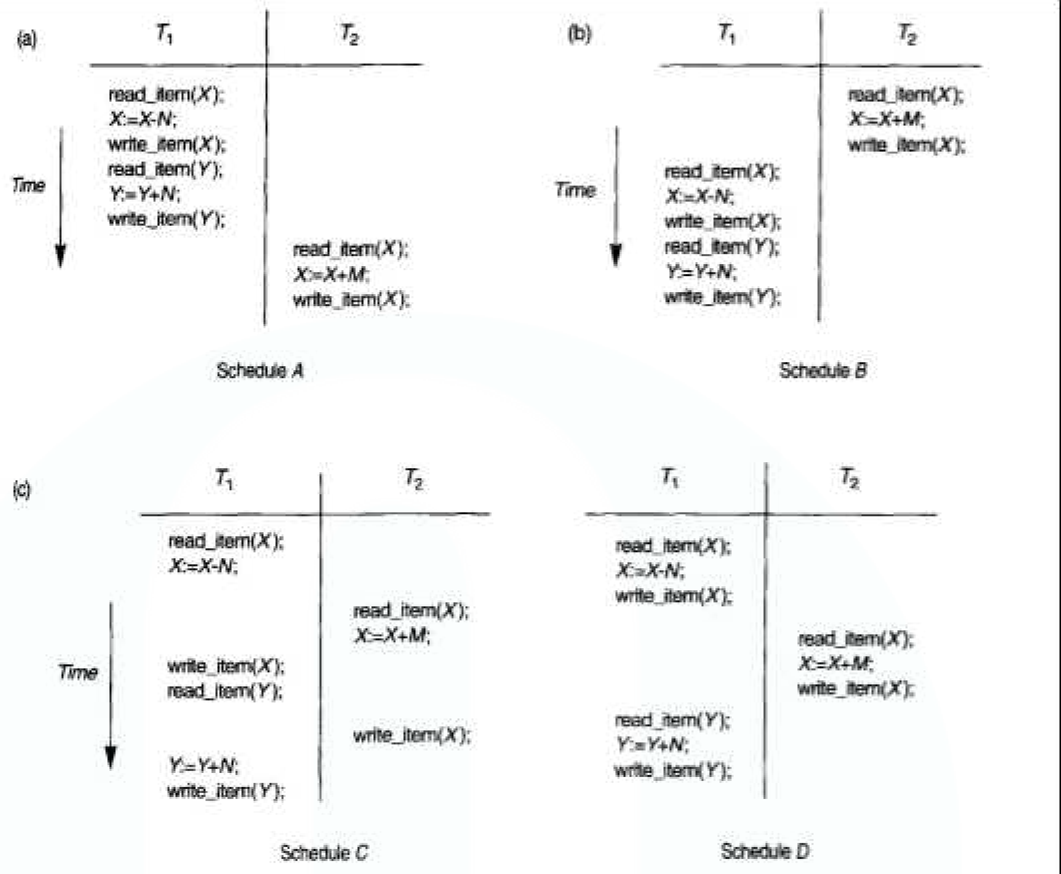
R1(x) W2(x)

R2(x) W1(x)

W2(x) W1(x)



$$\begin{tikzcd} \boxed{1} \arrow[r, bend left] & \boxed{2} \arrow[l, bend left] \end{tikzcd}$$



(a)

T_1	T_2
read_item(X);	
X:=X-N;	
write_item(X);	
read_item(Y);	
Y:=Y+N;	
write_item(Y);	
	read_item(X);
	X:=X+M;
	write_item(X);

Schedule *A*

(b)

T_1	T_2
	read_item(X);
	X:=X+M;
	write_item(X);
read_item(X);	
X:=X-N;	
write_item(X);	
read_item(Y);	
Y:=Y+N;	
write_item(Y);	

Schedule *B*

(c)

T_1	T_2
read_item(X);	
X:=X-N;	
	read_item(X);
	X:=X+M;
write_item(X);	
read_item(Y);	
	write_item(X);
Y:=Y+N;	
write_item(Y);	

Schedule *C*

T_1	T_2
read_item(X);	
X:=X-N;	
write_item(X);	
	read_item(X);
	X:=X+M;
	write_item(X);
read_item(Y);	
Y:=Y+N;	
write_item(Y);	

Schedule *D*

FIGURE 17.5 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.

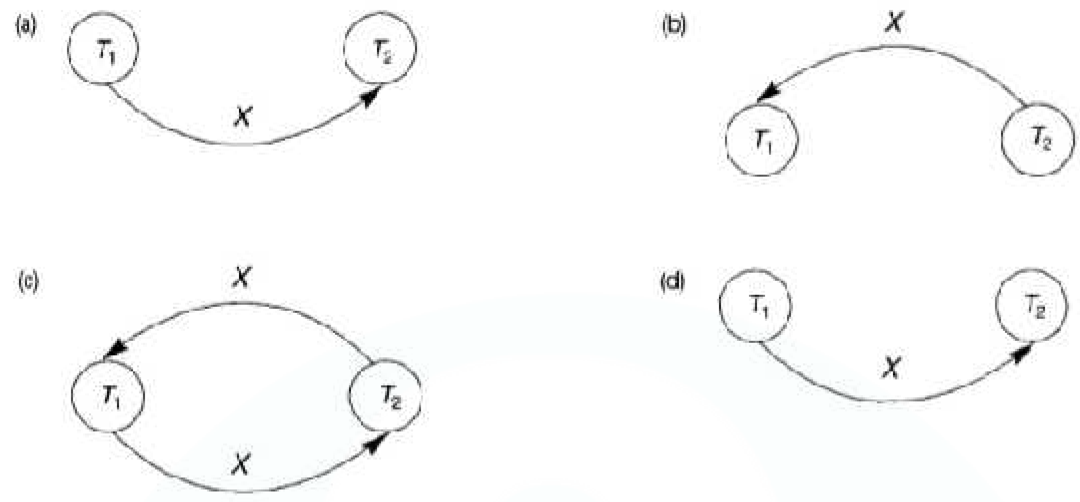


FIGURE 17.7 Constructing the precedence graphs for schedules *A* to *D* from Figure 17.5 to test for conflict serializability. (a) Precedence graph for serial schedule *A*. (b) Precedence graph for serial schedule *B*. (c) Precedence graph for schedule *C* (not serializable). (d) Precedence graph for schedule *D* (serializable, equivalent to schedule *A*).

TWO-PHASE LOCKING

While one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a lock on that item

Locks

There are various modes in which a data item may be locked. In this section, we restrict our attention to **two modes:**

1. **Shared.** 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.** If a transaction Ti has obtained an exclusive-mode lock(denoted by X) on item Q, then Ti can both read and write Q.

The use of these two lock modes allows multiple transactions to read a data item but limits write access to just one transaction at a time.

- 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.

One protocol that ensures serializability is the **two-phase locking protocol.**

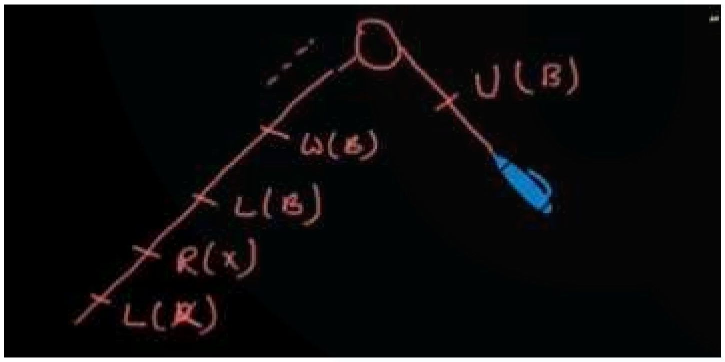
This protocol requires that each transaction issue lock and unlock requests **in two phases:**

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.



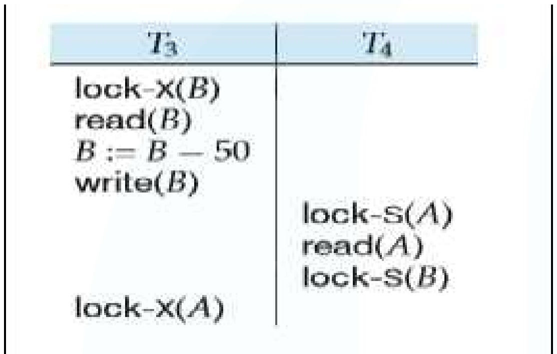
Left edge denotes the growing phase.(locks are granted)

Right edge denotes the shrinking phase.(locks are releasing)

- 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.
- The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the **lock point of the transaction**.
- Now, transactions can be ordered according to their lockpoints— this ordering is, in fact, a serializability ordering for the transactions.

Difficulties faced by 2-phase locking—deadlock, cascading rollback

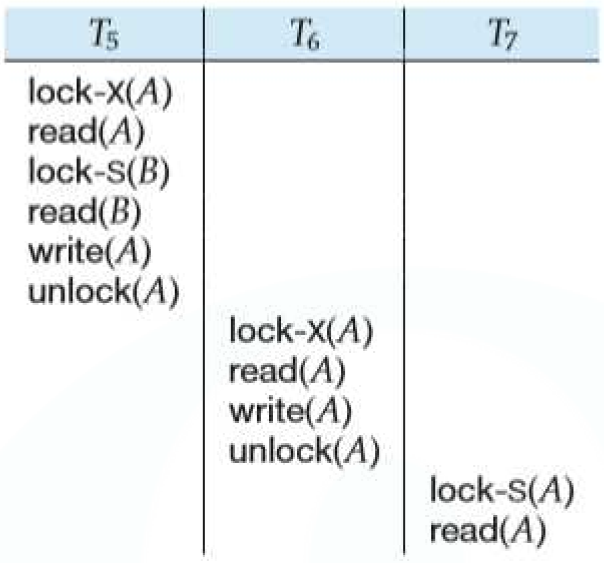
Two-phase locking does not ensure freedom from ***deadlock.*** Observe that transactions T3 and T4 are two phase, but, in schedule 2 (Figure 15.7), they are deadlocked.



T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Figure 15.7 Schedule 2.

- Since T3 is holding **an exclusive mode** lock on B and T4 is requesting a **shared-mode** lock on B(which can't be granted),thus T4 is waiting for T3 to unlock B.
- Similarly, since T4 is holding a **shared-mode lock** on A and T3 is requesting an **exclusive-mode lock** on A(which can't be granted), T3 is waiting for T4 to unlock A.
- Thus ,we have arrived at a state where neither of these transactions can ever proceed with its normal execution .This situation is called **deadlock**.
- When deadlock occurs, the system must roll back one of the two transactions.
- ***Cascading rollback*** may occur under two-phase locking.
- Each transaction observes the two-phase locking protocol, but the failure of T5 after the read(A) step of T7 leads to **cascading rollback(series of rollback) of T6 and T7.**



T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Figure 15.8 Partial schedule under two-phase locking.

- Cascading rollbacks can be **avoided** by a modification of two-phase locking called the ***strict two-phase locking protocol***. **This protocol requires not only that locking be two phase, but also that all exclusive-mode locks taken by a transaction be held until that transaction commits.** This requirement ensures that any data **written by an uncommitted transaction are locked** in exclusive mode until the transaction commits, preventing any other transaction from reading the data.
- Another variant of two-phase locking is the ***rigorous two-phase locking protocol***, which requires that all locks be held until the transaction commits.
- **With rigorous two-phase locking, transactions can be serialized in the order in which they commit.**
- Strict two-phase locking and rigorous two-phase locking (with lock conversions) are used extensively in commercial database systems.
- In basic two-phase locking protocol, **lock conversions** are allowed
- A mechanism for upgrading a shared lock to an exclusive lock, and downgrading an exclusive lock to a shared lock.
- We denote conversion from shared to exclusive modes by upgrade, and from exclusive to shared by downgrade.
- Upgrading can take place in only the **growing phase,** whereas downgrading can take place in only the **shrinking phase.**

FAILURE CLASSIFICATION,

Types of Failures. Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:

1. *A computer failure (system crash):* A **hardware, software, or network error** occurs in the computer system during transaction execution. Hardware crashes are usually media failures-for example, main memory failure.

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*: During transaction execution, **certain conditions** may occur that necessitate cancellation of the transaction. For example, data for the transaction may not be found. Notice that an exception condition**," such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled.** This exception should be programmed in the transaction itself, and hence would not be considered a failure.

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. Failures are generally classified as transaction, system, and media failures.

STORAGE STRUCTURE,

Three categories of storage:

• Volatile storage

• Nonvolatile storage

• Stable storage

STABLE STORAGE

- Stable storage plays a critical role in recovery algorithms.
- 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.
- RAID systems (is the mirrored disk, which keeps two copies of each block, on separate disks),**cannot** guard against data loss due to disasters such as **fires or flooding.**
- Many systems store archival **backups of tapes offsite** to guard against such disasters.
- However, since tapes cannot be carried offsite continually, updates since the most recent time that tapes were carried offsite could be lost in such a disaster.
- More secure systems keep a copy of each block of stable storage at a remote site, writing it out over a computer network, in addition to storing the block on a local disk system.
- Since **the blocks are output to a remote system** as and when they are output to local storage, once an output operation is complete, **the output is not lost**, **even in the event of a disaster such as a fire or flood.**
- If the **system fails** while blocks are being written, it is possible that the two copies of a block are **inconsistent** with each other. During recovery, for each block, the system would need to examine two copies of the 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(that is, updates all copies)or results in no change.

- The requirement of comparing every corresponding pair of blocks during recovery is expensive to meet.
- We can reduce the cost greatly by keeping track of block writes that are in progress, using a small amount of nonvolatile RAM. On recovery, only blocks for which writes were in progress need to be compared.

LOG BASED RECOVERY

System Log

- To be able to recover from failures that affect transactions, the system maintains a **log** to keep track of all transaction operations that affect the values of database items.
- This information may be needed to permit recovery from failures. The log is **kept on disk**, so it is not affected by any type of failure except for disk or catastrophic failure.
- The types of entries-called **log records**-that are written to the log and the action each performs. In these entries, T refers to a unique transaction-id that is generated automatically by the system and is used to identify each transaction:

1.[start-transaction , T]: Indicates that transaction T has started execution.

2.[write_item, T , X ,old_value,new_value]: Indicates that transaction T has changed the value of database item X from old_value to new_value.

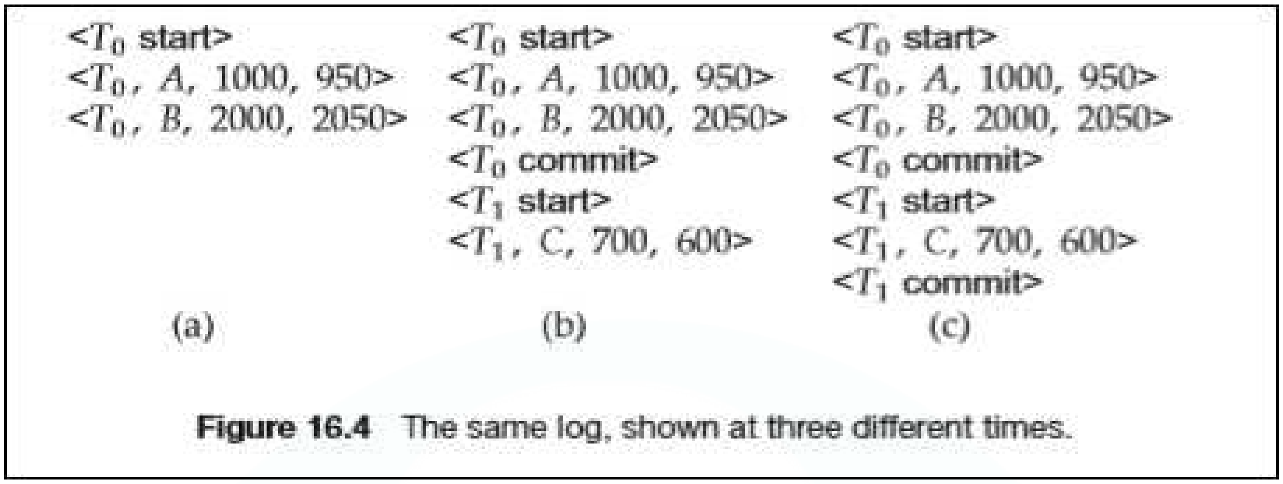
3.[read_item, T, X]: Indicates that transaction T has read the value of database item X.

4.[commit, T]: Indicates that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.

5.[abort, T]: Indicates that transaction T has been aborted.

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.**

Redoing the operations of a transaction may also be needed if all its updates are recorded in the log but a failure occurs before we can be sure that all these **new_values** have been written permanently in the actual database on disk." Redoing the operations of transaction T is applied by **tracing forward through the log** and setting all items changed by a WRITE operation of T to their new_values.



(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>

Figure 16.4 The same log, shown at three different times.

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

Transaction T i needs to be redone if the log contains the record<Ti, start> and either the record<Ti, commit>or the record <Ti, abort>.

- **undo(T i)** restores the value of all data items updated by transaction T i to the old values.

Transaction Ti needs to be **undone** if the log contains the record **<Ti,start>,** but **does not** contain either the record<Ti,commit>or the record<Ti,abort>.

The undo operation not only **restores the data items to their old value**, but also writes log records to **record the updates performed** as part of the undo process. These log records are **special redo-only log records.**

The state of the logs for each of these cases appears in Figure 16.4.

(a)First, let us assume that the crash occurs just after the log record for the step:

write(B)

of transactionT0 has been written to stable storage(**Figure16.4a).**When the system comes back up, it finds the record <T0 start> in the log, but no corresponding <T0 commit> or <T0 abort> record.

- Thus, transaction T0 must be undone, so an **undo**(T0) is performed.
- As a result, the values in accounts A and B (on the disk) are restored to \$1000 and \$2000, respectively.

(b)Next,let us assume that the crash comes just after the log record for the step:

write(C)

of transaction T1 has been written to stable storage (**Figure 16.4b**). When the system comes back up, two recovery actions need to be taken. The operation **undo**(T1) must be performed, since the record <T1 start> appears in the log, but there is no record <T1 commit> or <T1 abort>. The operation **redo**(T0) must be performed, since the log contains both the record

<T0 start> and the record <T0 commit>.

- At the end of the entire recovery procedure, the values of accounts A, B, and C are $950, $2050, and $700, respectively.

(c) Finally, let us assume that the crash occurs just after the log record:

<T1 commit>

has been written to stable storage(**Figure16.4c)**.When the system comes back up, both T0 and T1 need to be **redone** ,since the records<T0,start>and<T0,commit> appear in the log, as do the records<T1,start>and<T1,commit>.After the system performs the recovery procedures redo(T0) and redo(T1), the values in accounts A, B, and C are $950, $2050, and $600, respectively.

DEFERRED DATABASE MODIFICATION

- 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**.
- The database is partitioned into fixed-length storage units called **blocks.**
- A transaction creates a log record prior to modifying the database.
- The log records allow the system to undo changes made by a transaction in the event that the transaction must be aborted; they allow the system also to redo changes made by a transaction if the transaction has committed but the system crashed before those changes could be stored in the database on disk.
- The steps a transaction takes in modifying a data item:

1.The transaction performs some computations in its own private part of main memory.
2. The transaction modifies the data block in the disk buffer in main memory holding the data item.
3.The database system executes the output operation that writes the data block to disk.

We say a *transaction modifies* the database if it performs an **update on a disk buffer**, or on the disk itself; updates to the private part of main memory do not count as database modifications.

If a transaction does not modify the database **until it has committed**, it is said to use the **deferred-modification technique.**

If database **modifications occur while the transaction is still active**, the transaction is said to use the **immediate-modification technique.**

Deferred modification has the **overhead that transactions need to make local copies of all updated data items;** further, if a transaction reads a data item that it has updated, it must read the value from its local copy.

Because all database modifications must be preceded by the creation of a log record, the system has available both the old value prior to the modification of the data item and the new value that is to be written for the data item.

This allows the system to perform undo and redo operations as appropriate.

• **Undo** using a log record sets the data item specified in the log record to the old value.

• **Redo** using a log record sets the data item specified in the log record to the new value.

<u>Recovery After a System Crash</u>

Recovery actions, when the database system is restarted after a crash, take place in two phases:

(a) Redo phase

(b) Undo phase

In the ***redo phase***, the system replays updates of all transactions by scanning the log forward from the last checkpoint. The log records that are replayed include log records for transactions that were rolled back before system crash, and those that had not committed when the system crash occurred.

This phase also determines all transactions that were incomplete at the time of the crash, and must therefore be rolled back.

The specific steps taken while scanning the log are as follows:

a. The list of transactions to be rolled back, ***undo-list***, is initially set to the **list L** in the <checkpoint L> log record.
b. Whenever a log record of the form **<T i start>** is found, **T i is added to undo-list.**

c. Whenever a log record of the form **<Ti abort> or <Ti commit>** is found, **Ti is removed from undo-list.**

In ***the undo phase***, the system rolls back all transactions in the undo-list. It performs rollback by scanning the log backward from the end.

a. When the system **finds a <T i start> log record for a transaction Ti in undo-list**, it writes a **<T i abort> log** record to the log, and **removes Ti** from undo-list.

b. The undo phase terminates once undo-list becomes empty, that is, the system has found <Ti start> log records for all transactions that were initially in undo-list.

After the undo phase of recovery terminates, normal transaction processing can resume.

CHECK-POINTING

When a system crash 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.

To reduce these types of overhead, we introduce checkpoints. We describe below a simple checkpoint scheme that

(a) does not permit any updates to be performed while the checkpoint operation is in progress, and

(b) outputs all modified buffer blocks to disk when the checkpoint is performed.

A checkpoint is performed as follows:

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 of the form <checkpoint L>,where L is a list of transactions active at the time of the checkpoint.

Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress.

Consider a transaction Ti that completed prior to the checkpoint. For such a transaction ,the

<Ti commit>record(or <Ti abort> record) appears in the log before the <checkpoint> record. Any database modifications made by Ti must have been written to the database either prior to the checkpoint or as part of the checkpoint itself.

Thus, at recovery time, there is no need to perform a redo operation on Ti.

After a system crash has occurred, the system examines the log to find the last <checkpoint L> record (this can be done by searching the log backward, from the end of the log, until the first <checkpoint L> record is found).

The redo or undo operations need to be applied only to transactions in L ,and to all transactions that started execution after the <checkpoint L> record was written to the log. Let us denote this set of transactions as T.

• For all transactions Tk in T that have no <Tk commit> record or <Tk abort> record in the log, execute undo(Tk).

• For all transactions Tk in T such that either the record <Tk commit> or the record <Tk abort> appears in the log, execute redo(Tk).

Consider the set of transactions {T0, T1,...,T100}.

- Suppose that the most **recent** checkpoint took place **during the execution** of transaction T67 and T69, **while T68 and all transactions with subscripts lower than 67 *completed* before the checkpoint.**
- Thus, only transactions **T67, T69,……..,T100** need to be considered during the recovery scheme. Each of them needs to be redone if it has completed (that is, either committed or aborted); otherwise, it was incomplete, and needs to be undone.
- The requirement that transactions must not perform any updates to buffer blocks or to the log during check pointing can be bothersome, since transaction processing has to halt while a checkpoint is in progress. **A fuzzy checkpoint** is a checkpoint where transactions are allowed to perform updates **even while** buffer blocks are being written out.

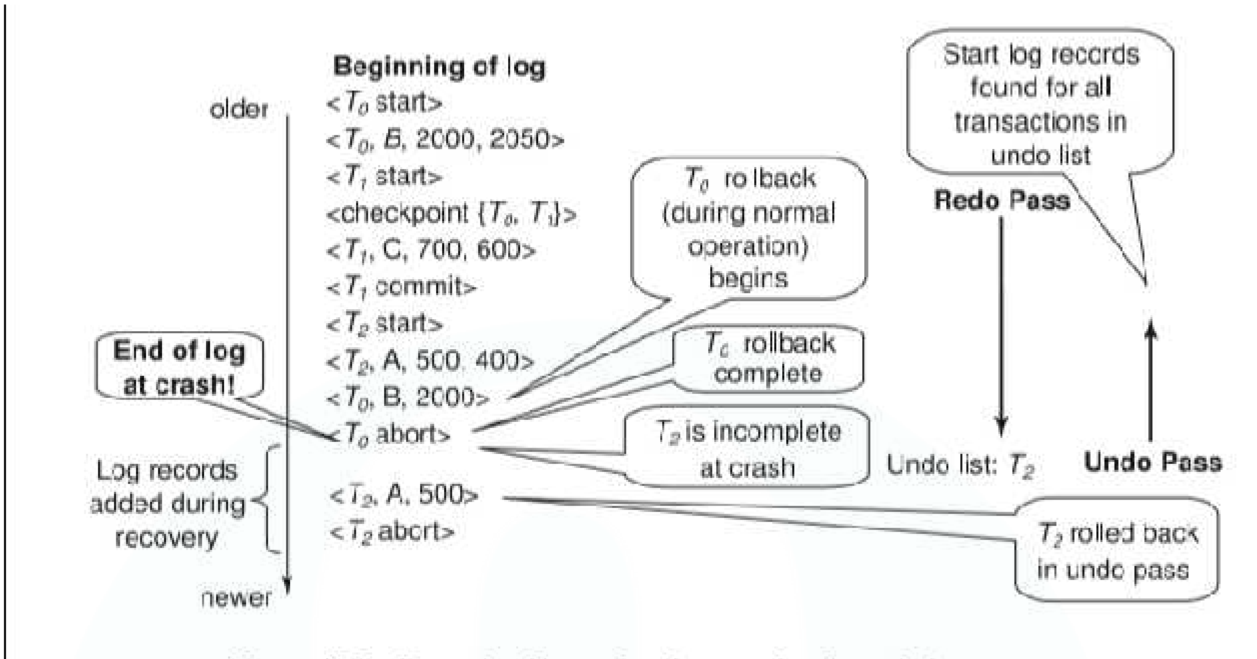


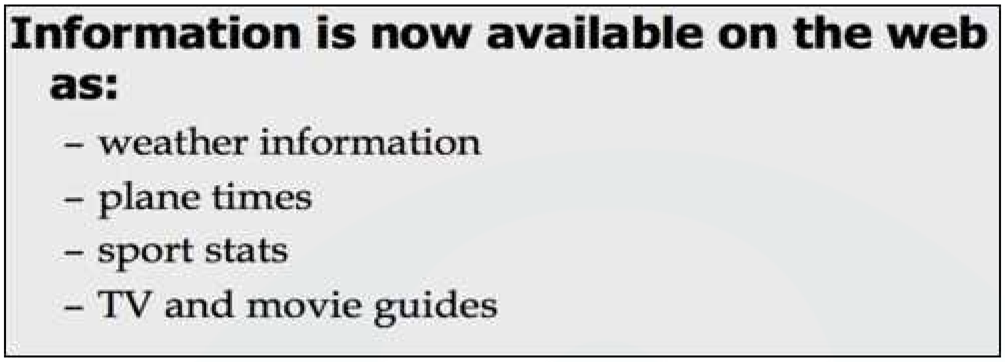
Figure 16.5 Example of logged actions, and actions during recovery.

Figure 16.5 shows an example of actions logged during normal operation, and actions performed during failure recovery.

- In the log shown in the figure, transaction T1 had committed, and transaction T0 had been completely rolled back, before the system crashed.
- Observe how the value of data item B is restored during the rollback of T0.
- Observe also the **checkpoint record**, with the **list of active transactions containing T0 and T1**. When recovering from a crash, ***in the redo phase***, the system performs a redo of all operations after the last checkpoint record.
- In this phase, **the list undo-list** initially contains T0 and T1; T1 is removed first when its commit log record is found, while T2 is added when its start log record is found.
- Transaction T0 is removed from undo-list when its abort log record is found, leaving only T2 in undo-list.
- ***The undo phase*** scans the log backwards from the end, and when it finds a log record of T2 updating A, the old value of A is restored ,and are do-only log record written to the log.
- When the start record for T2 is found, an abort record is added for T2.
- Since undo-list contains no more transactions, the undo phase terminates, completing recovery.

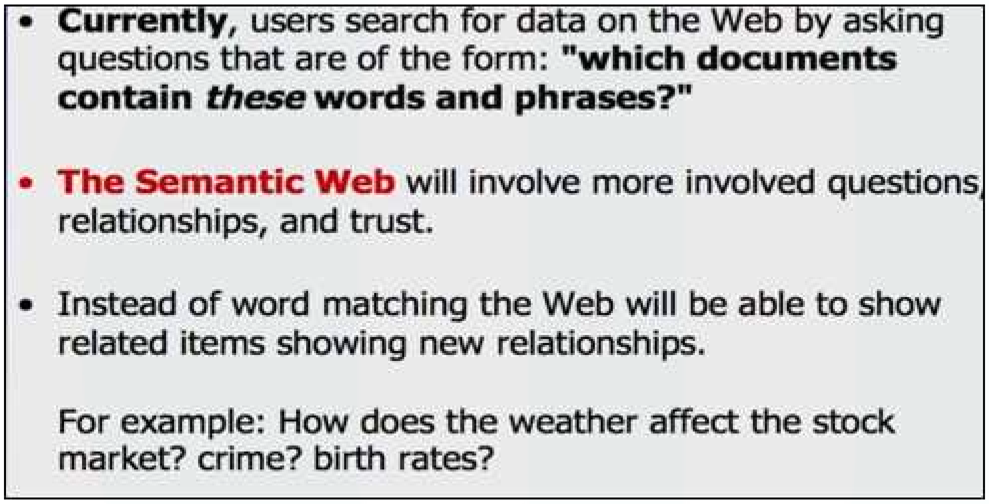
<u>**SEMANTIC WEB AND RDF**</u>

- The Semantic Web is an extension of the World Wide Web through standards by the World Wide Web Consortium (W3C).
- According to the W3C, "The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries".



Information is now available on the web as:

- weather information
- plane times
- sport stats
- TV and movie guides



- **Currently,** users search for data on the Web by asking questions that are of the form: **"which documents contain *these* words and phrases?"**
- **The Semantic Web** will involve more involved questions, relationships, and trust.
- Instead of word matching the Web will be able to show related items showing new relationships.

  For example: How does the weather affect the stock market? crime? birth rates?

- When we search for information, we get information about information in a wider range in semantic web.
- If we search for a medicine, we can even get information regarding the doctor's practicing in this area.
- At the base of Semantic Web is RDF, Resource Description Framework .
- An RDF expression consists of a set of statements, where a statement is a fact about a resource, for example title, description, creation date, or a relation to another resource.
- A statement can be seen as a very simple sentence in natural language following the pattern "subject predicate object" where subject and predicate are resources while the object is either a resource or a literal .
- In figure 1 we see two statements in graph notation, and in table 1 the same statements are listed in a plain three-column layout.

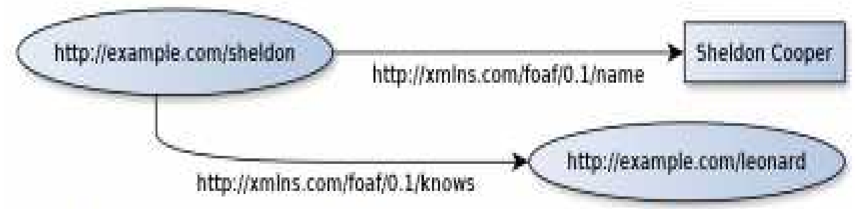

http://example.com/sheldon
http://xmlns.com/foaf/0.1/name
Sheldon Cooper
http://xmlns.com/foaf/0.1/knows
http://example.com/leonard

Figure 1: RDF graph with two statements, resources are depicted as ellipses and literals as squares.

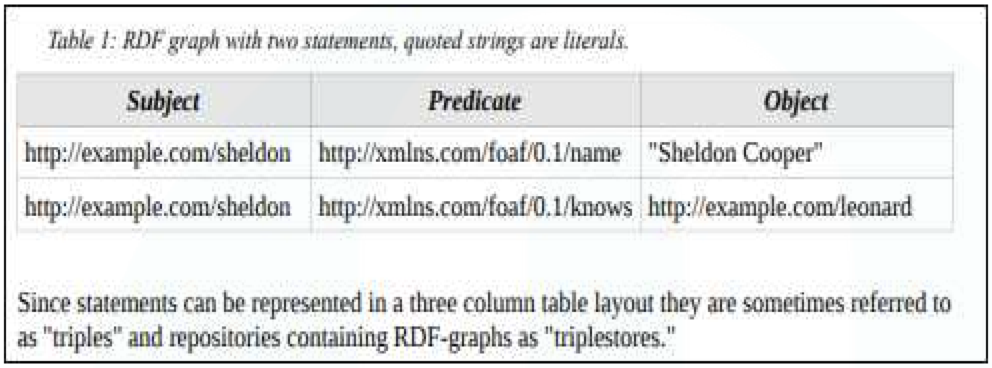


Table 1: RDF graph with two statements, quoted strings are literals.

Subject	Predicate	Object
http://example.com/sheldon	http://xmlns.com/foaf/0.1/name	"Sheldon Cooper"
http://example.com/sheldon	http://xmlns.com/foaf/0.1/knows	http://example.com/leonard

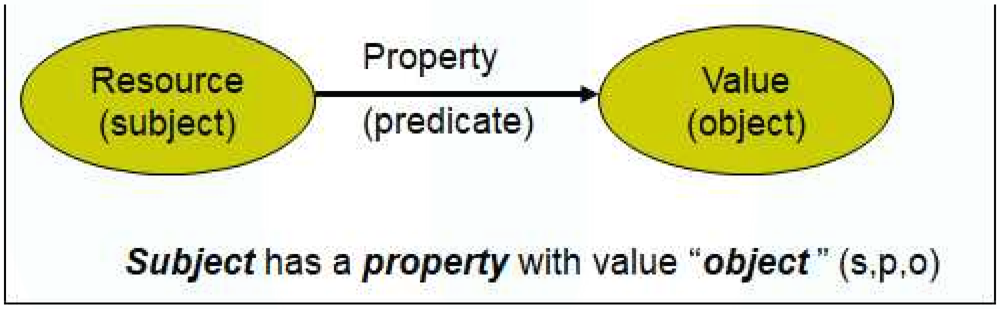
Since statements can be represented in a three column table layout they are sometimes referred to as "triples" and repositories containing RDF-graphs as "triplestores."

- RDF(**Resource Description Framework**) is the key part of the Semantic Web activities. It helps to realize the vision of the Semantic Web that:
  - Web information should have exact meaning
  - Web information can be understood and processed by computers
  - Computers can integrate information from the Web
- It provides a model for data, and a syntax so that independent parties can exchange and use it.
- RDF is designed mainly to be read and understood by computers
- RDF is not designed for being displayed to people
- RDF is written in XML
  - Any XML processor, parsers can parse and process RDF
  - The XML language used for RDF is called RDF/XML

- RDF will provide machine the knowledge tree (semantic graph), where order is not important. – RDF data model is labeled (unordered) graph with two kinds of nodes (resources and literals).
- While XML will give a person a document tree where the order or serilaization is important. – XML data model is node-labeled tree (with left-to-right ordered)
- **Semantic Web** technologies enable people to create data stores on the Web, build vocabularies, and write rules for handling data.
- The Semantic Web is a mesh of information linked up in such a way as to be easily processable by machines, on a global scale.

<u>Basic Ideas behind RDF</u>

- RDF uses Web identifiers (URIs) to identify resources
- RDF describes resources with properties and property values
    - Everything can be represented as triples
- The essence of RDF is the (s,p,o) triple

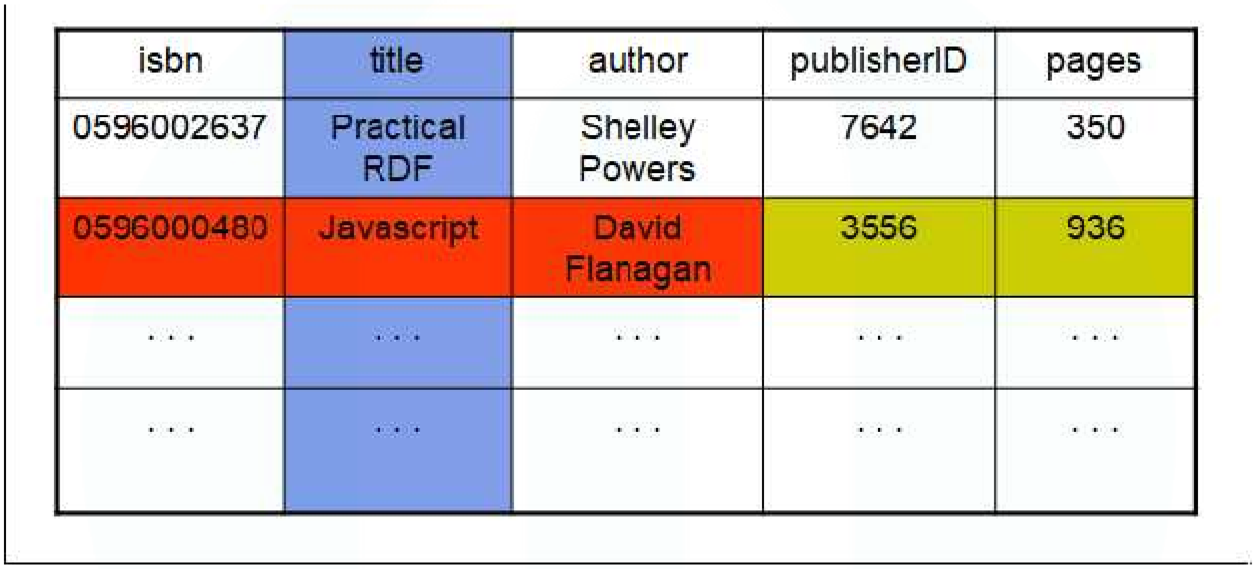


- **Triple**
    - ✓ A **Resource (Subject)** is anything that can have a URI: URIs or blank nodes .
    - ✓ A **Property (Predicate)** is one of the features of the Resource: URIs .
    - ✓ A **Property value (Object)** is the value of a Property, which can be literal or another resource: URIs, literal, blank nodes.
- Any expression in RDF is a collection of triples (subject, predicate, object)
- A set of such triples is called an ***RDF graph***
    - o The **nodes** of an RDF graph are its subjects and objects

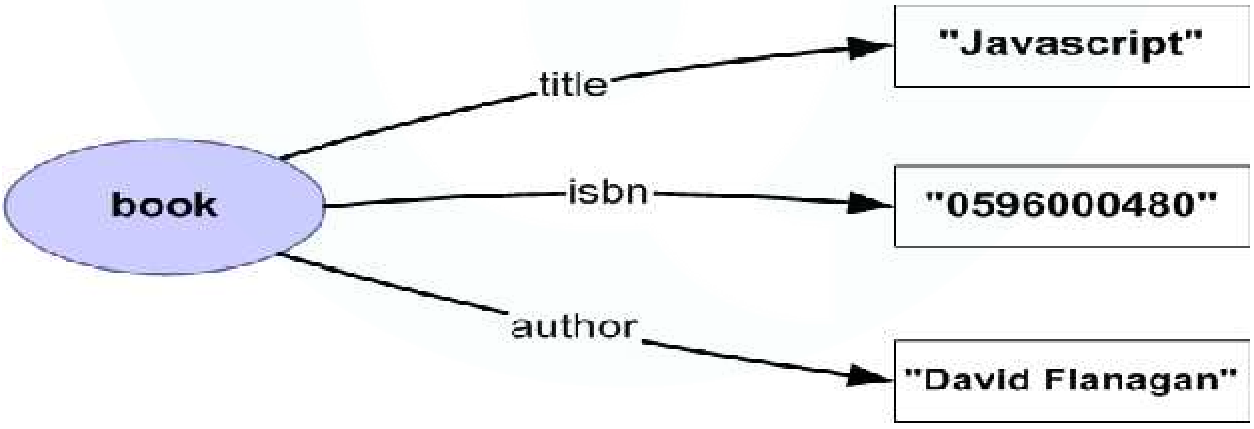
- The **direction** of the arc is significant: it always points toward the object.

- A assertion of an RDF triple says the relationship (indicated by teh predicate) holds between subject and object.
- The meaning of an RDF graph is conjunction (AND) of the statements corresponding to all the triples it contains
- RDF does not provide means to express negation (NOT) or disjunction (OR)

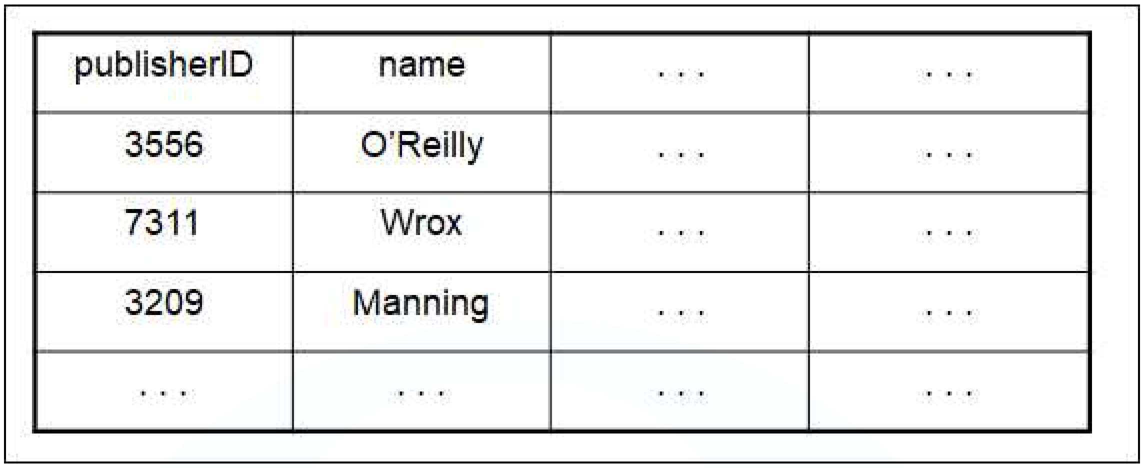
<u>BOOK TABLE</u>



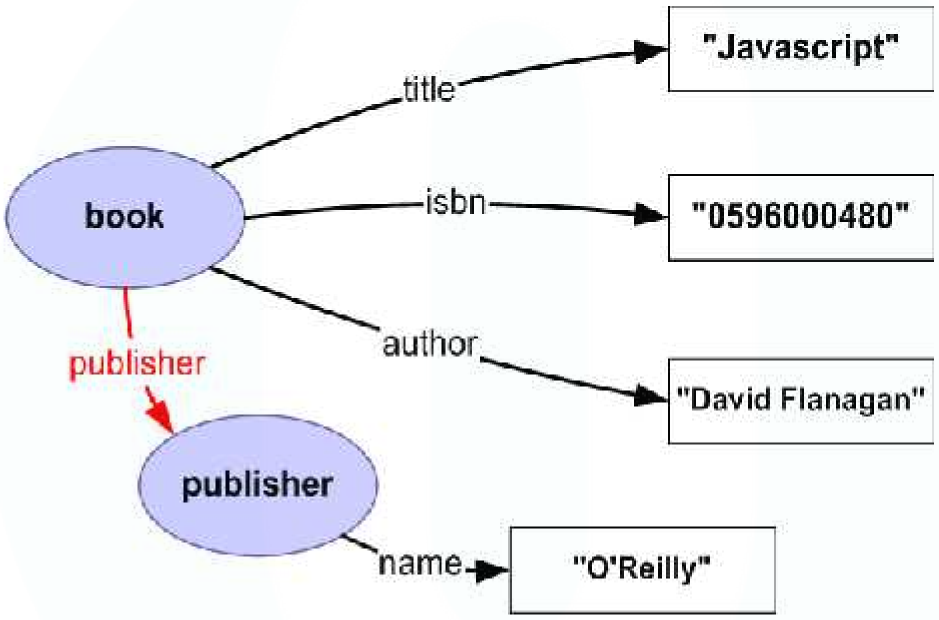
isbn	title	author	publisherID	pages
0596002637	Practical RDF	Shelley Powers	7642	350
0596000480	Javascript	David Flanagan	3556	936
. . .	. . .	. . .	. . .	. . .
. . .	. . .	. . .	. . .	. . .



- Publishers table: publisherID is primary key and exists as foreign key in book table



publisherID	name	. . .	. . .
3556	O'Reilly	. . .	. . .
7311	Wrox	. . .	. . .
3209	Manning	. . .	. . .
. . .	. . .	. . .	. . .



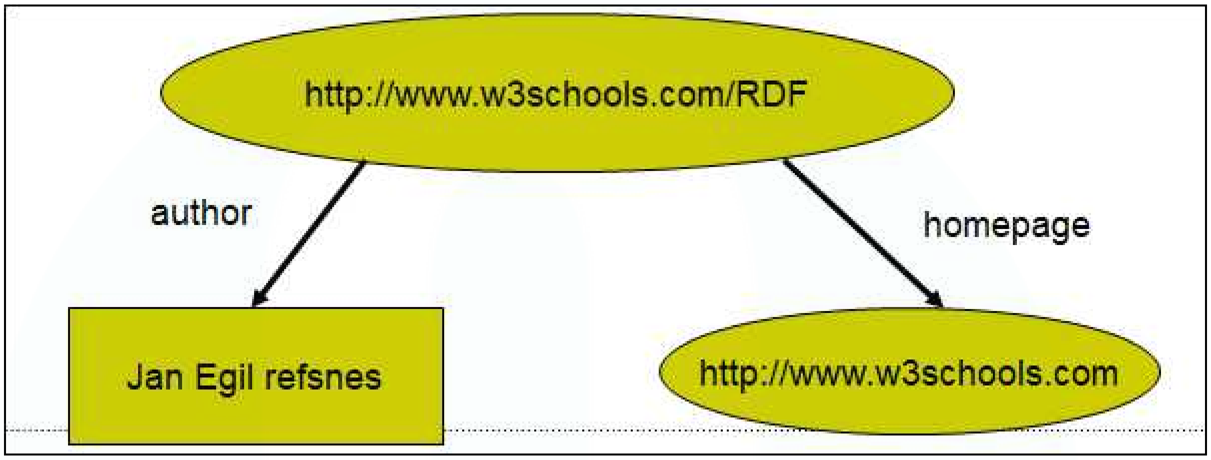
<u>RDF is a graph</u>

- An (s, p, o) triple can be viewed as a labeled graph
- The formal semantics of RDF is also described using graphs
- Think in terms of graphs, not XML or documents
- Nodes in graph are things (resources), arcs are relationship between things (resources)

<u>Simple example of RDF</u>

<RDF>

```
  <Description about="http://www.w3schools.com/RDF">
    <author>Jan Egil refsnes</author>
    <homepage>http://www.w3schools.com</homepage>
  </Description>
</RDF>
```



GIS

BIOLOGICAL DATABASES

BIG DATA

BIG DATA

- **Big data** is the term for collection of data sets so large and complex that it becomes difficult to process using on-hand database system tools or traditional data processing applications.
- Creators of web search engines were among the first to confront this problem.
- To meet the challenge of processing such large data sets, Google created MapReduce. Google's work and Yahoo's creation of the Hadoop MapReduce implementation has spawned an ecosystem of big data processing tools.

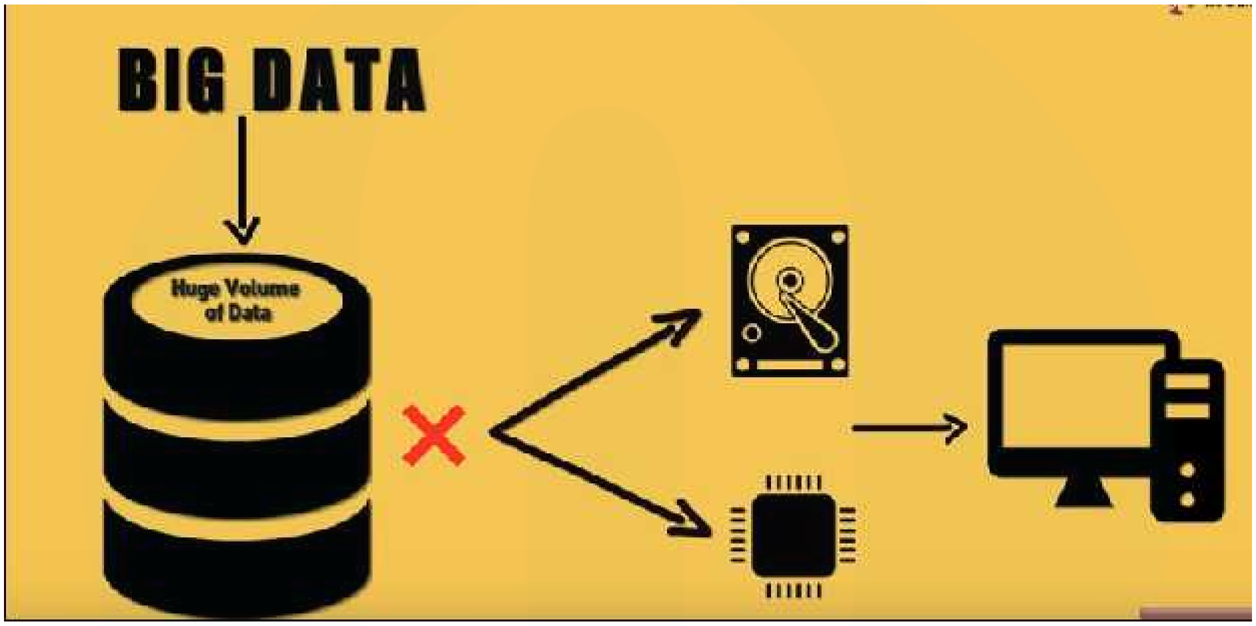
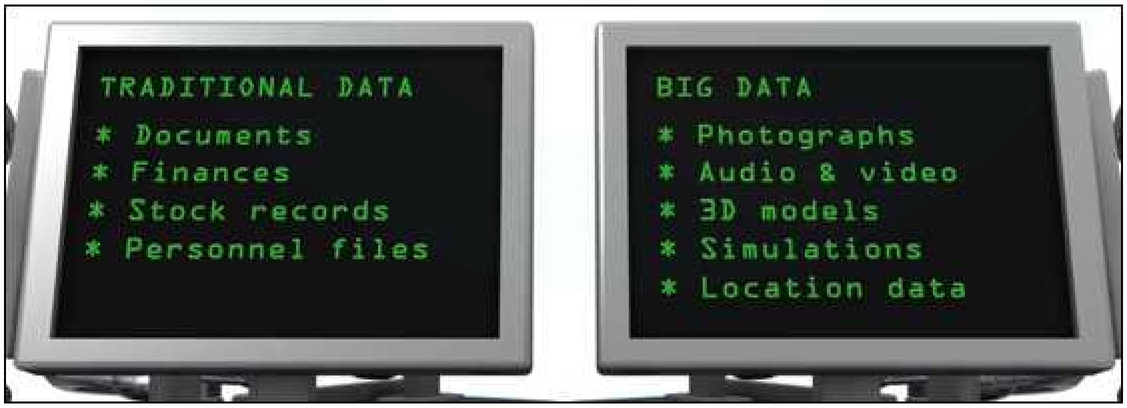
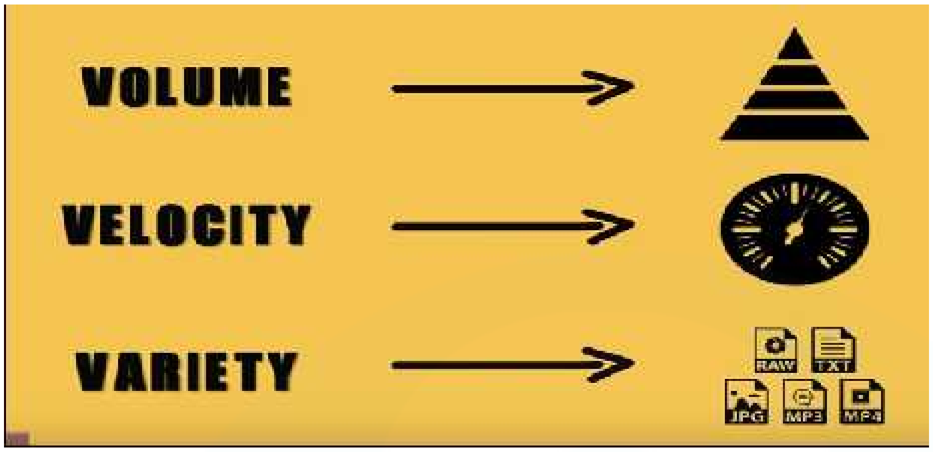


Fig: Big data cannot be processed using conventional methods.



Characteristics of big data



(i)Volume – The quantity of generated and stored data. The name 'Big Data' itself is related to a size which is enormous. Size of data plays very crucial role in determining value out of data. Also, whether a particular data can actually be considered as a Big Data or not, is dependent upon volume of data. Hence, 'Volume' is one characteristic which needs to be considered while dealing with 'Big Data'.

(ii)Variety – Big data draws from text, images, audio, video. Variety refers to heterogeneous sources and the nature of data, both structured and unstructured. During earlier days, spreadsheets and databases were the only sources of data considered by most of the applications. Now days, data in the form of emails, photos, videos, monitoring devices, PDFs, audio, etc. is also being considered in the analysis applications. This variety of unstructured data poses certain issues for storage, mining and analysing data.

(iii)Velocity –Big data is often available in real-time. The term 'velocity' refers to the speed of generation of data. How fast the data is generated and processed to meet the demands, determines real potential in the data.

Big Data Velocity deals with the speed at which data flows in from sources like business processes, application logs, networks and social media sites, sensors, Mobile devices, etc. The flow of data is massive and continuous.

https://examupdates.in/big-data-analytics/

Challenges associated with big data

- How do we store the data

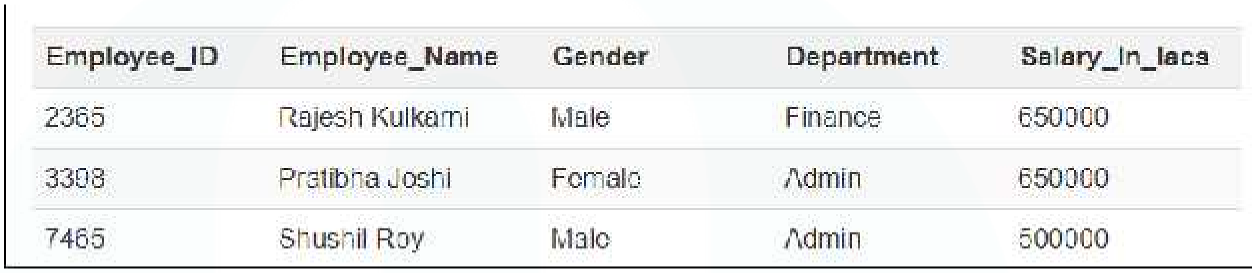
- How do we process the data

<u>Categories Of 'Big Data'</u>

Big data' could be found in three forms:

1. Structured

   An ’Employee’ table in a database is an example of Structured Data

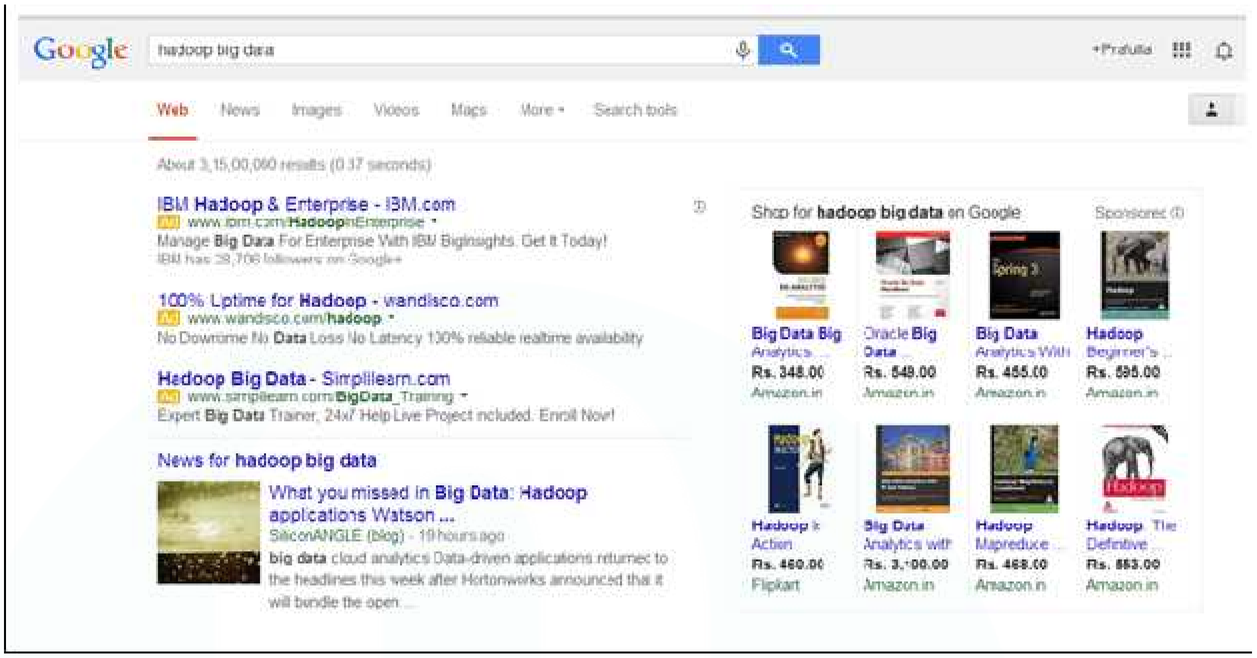


Employee_ID	Employee_Name	Gender	Department	Salary_In_lacs
2365	Rajesh Kulkarni	Male	Finance	650000
3398	Pratibha Joshi	Female	Admin	650000
7465	Shushil Roy	Male	Admin	500000

2. Unstructured

   Examples Of Un-structured Data

   Output returned by 'Google Search'



hadoop big data

Web News Images Videos Maps More ▾ Search tools

+Prafulla

About 3,15,00,060 results (0.37 seconds)

IBM Hadoop & Enterprise - IBM.com
Ad www.ibm.com/HadoopinEnterprise ▾
Manage Big Data For Enterprise With IBM BigInsights. Get it Today!
IBM has 28,706 followers on Google+

100% Uptime for Hadoop - wandisco.com
Ad www.wandisco.com/hadoop ▾
No Downtime No Data Loss No Latency 100% reliable realtime availability

Hadoop Big Data - Simplilearn.com
Ad www.simplilearn.com/BigData_Training ▾
Expert Big Data Trainer, 24x7 Help Live Project included. Enroll Now!

News for hadoop big data

What you missed in Big Data: Hadoop applications Watson ...
SiliconANGLE (blog) - 19 hours ago
big data cloud analytics Data-driven applications returned to the headlines this week after Hortonworks announced that it will bundle the open ...

Shop for hadoop big data on Google Sponsored

Big Data Big Analytics ...	Oracle Big Data ...	Big Data Analytics With	Hadoop Beginner's ...
Rs. 348.00	Rs. 549.00	Rs. 455.00	Rs. 595.00
Amazon.in	Amazon.in	Amazon.in	Amazon.in

Hadoop in Action	Big Data Analytics with	Hadoop Mapreduce ...	Hadoop The Definitive ...
Rs. 460.00	Rs. 3,100.00	Rs. 468.00	Rs. 863.00
Flipkart	Amazon.in	Amazon.in	Amazon.in

3. Semi-structured

Personal data stored in a XML file-

```
<rec><name>Prashant Rao</name><sex>Male</sex><age>35</age></rec>
<rec><name>Seema R.</name><sex>Female</sex><age>41</age></rec>
<rec><name>Satish Mane</name><sex>Male</sex><age>29</age></rec>
<rec><name>Subrato Roy</name><sex>Male</sex><age>26</age></rec>
<rec><name>Jeremiah J.</name><sex>Male</sex><age>35</age></rec>
```

Apache Hadoop is a collection of open-source software utilities that facilitate using a network of many computers to solve problems involving massive amounts of data and computation.

It provides a software framework for distributed storage and processing of big data using the MapReduce programming model.

Core components of hadoop

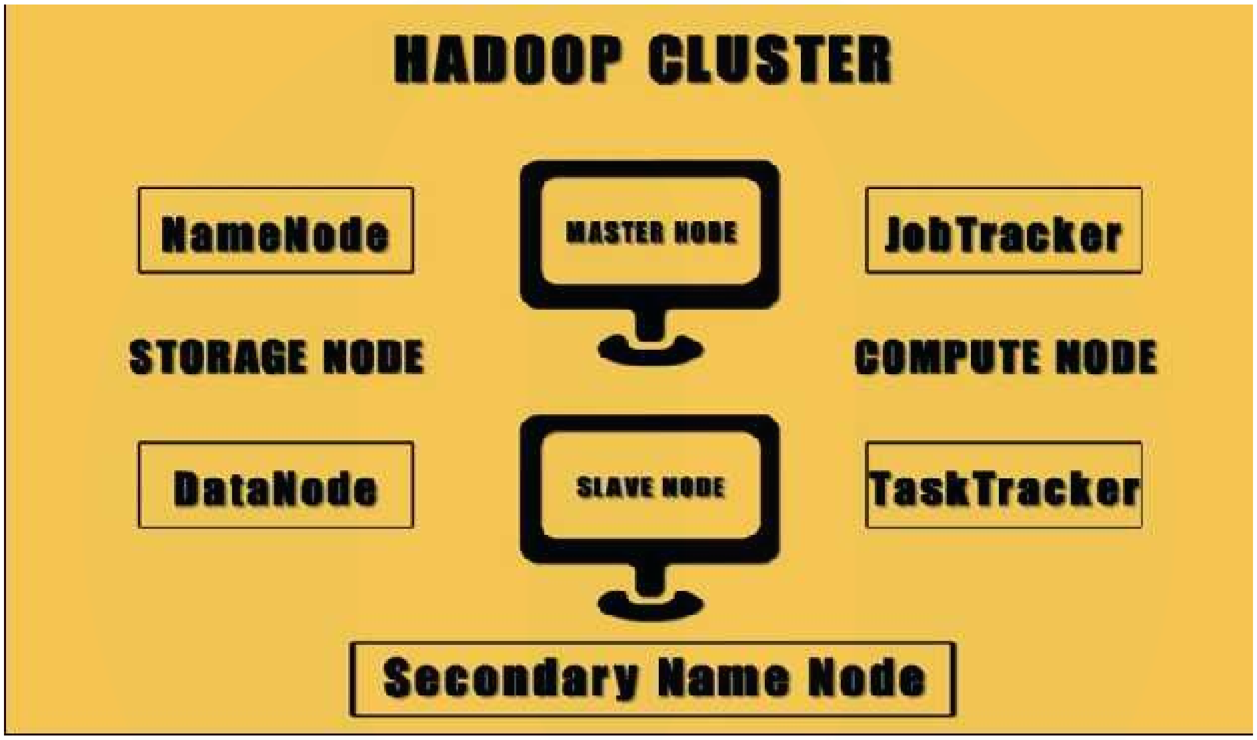
- Hadoop distributed file system

Hadoop Distributed File System (HDFS) – a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster;

- Map reduce

  An implementation of the MapReduce programming model for large-scale data processing.

Hadoop cluster



Master node responsible for running namenode

Slave node is responsible for running datanode

Atop the file systems comes the MapReduce Engine, which consists of one ***JobTracker,*** to which client applications submit MapReduce jobs.

The JobTracker pushes work to available ***TaskTracker*** nodes in the cluster, striving to keep the work as close to the data as possible.

The JobTracker knows which node contains the data, and which other machines are nearby.

If the work cannot be hosted on the actual node where the data resides, priority is given to nodes in the same rack.

If a TaskTracker fails or times out, that part of the job is rescheduled.

Features of hadoop

- No special hardware is required
- More computing and storage facility due to the presence of large cluster of nodes
- Within the cluster parallel processing
- Data distributed and replicated within the cluster
- Framework replace-admins need not worry
- All configurations and data are transferred
- Codes are transferred not data

1. Hadoop Brings Flexibility In Data Processing:

Hadoop manages data whether structured or unstructured, encoded or formatted, or any other type of data. Hadoop brings the value to the table where unstructured data can be useful in decision making process.

2. Hadoop Is Easily Scalable

This is a huge feature of Hadoop. It is an open source platform and runs on industry-standard hardware. That makes Hadoop extremely scalable platform where new nodes can be easily added in the system as and data volume of processing needs grow without altering anything in the existing systems or programs.

3. Hadoop Is Fault Tolerant

In Hadoop, the data is stored in HDFS where data automatically gets replicated at two other locations. So, even if one or two of the systems collapse, the file is still available on the third system at least. This brings a high level of fault tolerance.

The level of replication is configurable and this makes Hadoop incredibly reliable data storage system. This means, even if a node gets lost or goes out of service, the system automatically reallocates work to another location of the data and continues processing as if nothing had happened!

4. Hadoop Is Great At Faster Data Processing

While traditional ETL and batch processes can take hours, days, or even weeks to load large amounts of data, the need to analyze that data in real-time is becoming critical day after day.

Hadoop is extremely good at high-volume batch processing because of its ability to do parallel processing. Hadoop can perform batch processes 10 times faster than on a single thread server or on the mainframe.

5. Hadoop Ecosystem Is Robust:

Hadoop has a very robust ecosystem that is well suited to meet the analytical needs of developers and small to large organizations. Hadoop Ecosystem comes with a suite of tools and technologies making i a very much suitable to deliver to a variety of data processing needs.

6. Hadoop Is Very Cost Effective

Hadoop generates cost benefits by bringing massively parallel computing to commodity servers, resulting in a substantial reduction in the cost per terabyte of storage, which in turn makes it reasonable to model all your data.