MODULE 5

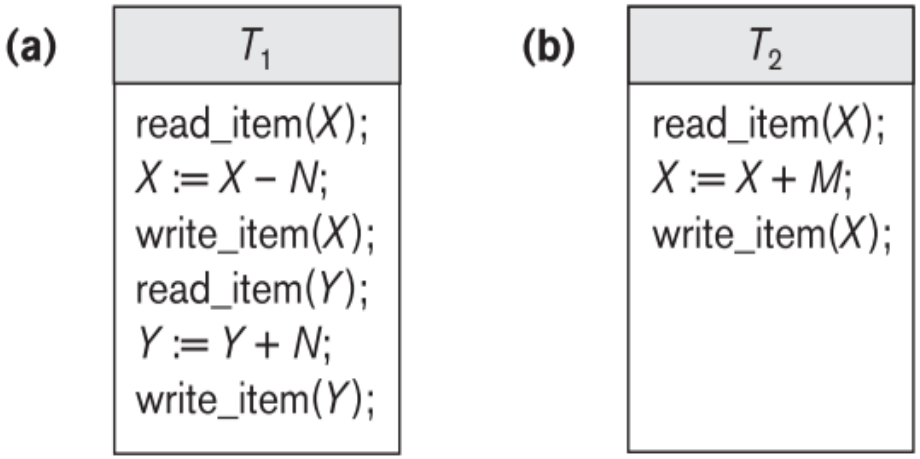
Transaction Processing Concepts

Transaction Processing

- An action or series of action performed by a user or an application program which reads or updates the content of database
- A **transaction** is an executing program that forms a logical unit of database processing
- includes one or more database access operations - insertion, deletion, modification, or retrieval operations.
- Example for transaction processing : Airline reservation, Banking transactions etc..

- One way of specifying the transaction boundaries is by specifying **begin transaction** and **end transaction** statements in an application program.
- the basic database access operations that a transaction can include are as follows:
  - **read_item(*X*).** Reads a database item named *X* into a program variable X
  - **write_item(*X*).** Writes the value of program variable *X* into the database item named *X*.

- Figure shows examples of two very simple transactions. The **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that the transaction writes

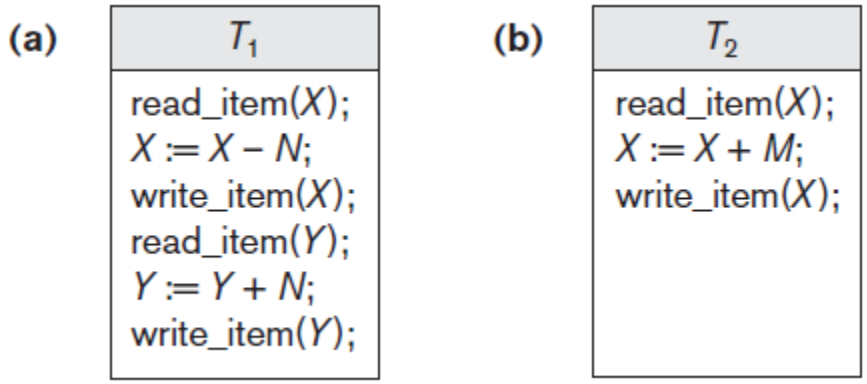


Concurrent Transactions

- Several Transaction will be executed in **concurrent manner**
- Concurrency control and recovery mechanisms are mainly concerned with the database commands in a transaction
- If this concurrent execution is *uncontrolled*, it may lead to problems

WHY CONCURRENCY CONTROL IS NEEDED

- When multiple transactions execute concurrently in an uncontrolled or unrestricted manner, then it might lead to several problems.
- These problems are commonly referred to as concurrency problems in database environment



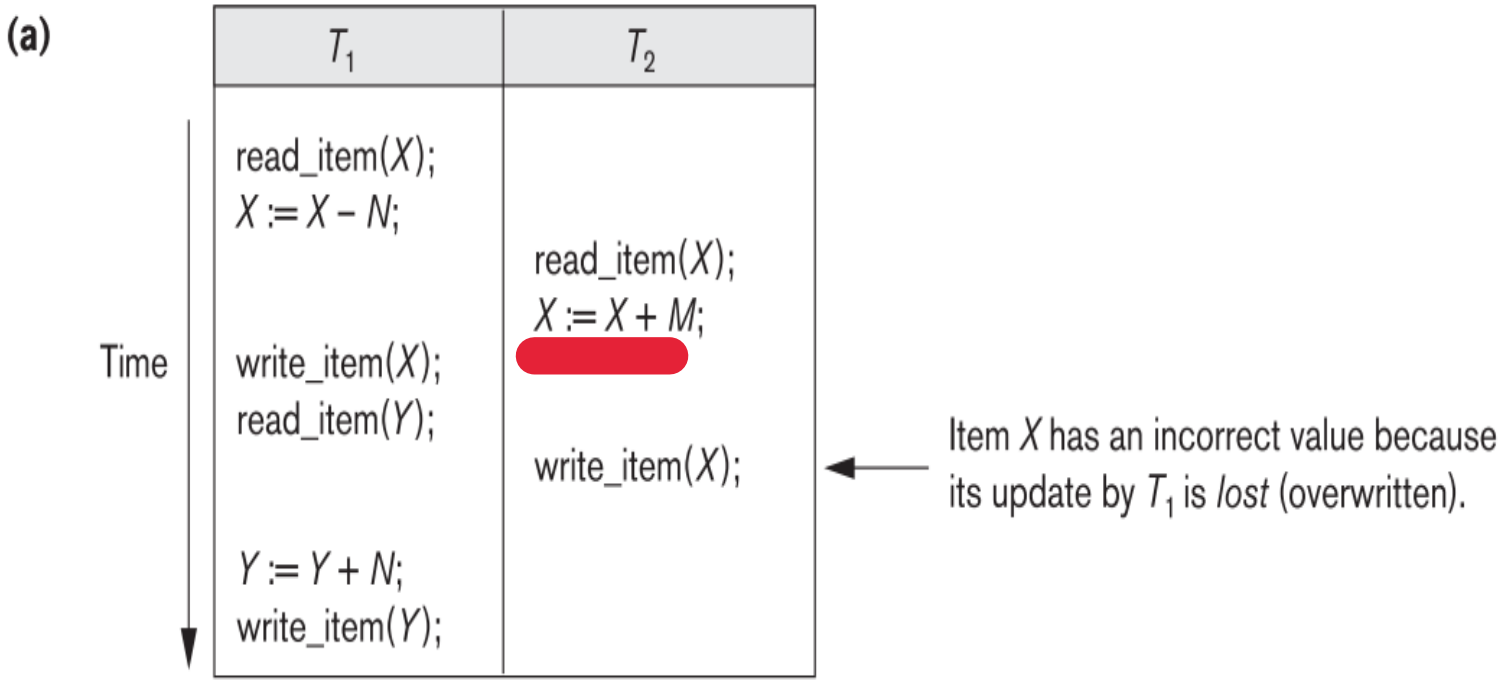
- Figure shows two sample transactions
- If we try to execute the transaction in concurrent (interleaved manner) following problem exists

The five concurrency problems that can occur in database are:

- **The Lost Update Problem**
- **The Temporary Update (or Dirty Read) Problem**
- **The Incorrect Summary Problem**
- **The Unrepeatable Read Problem**

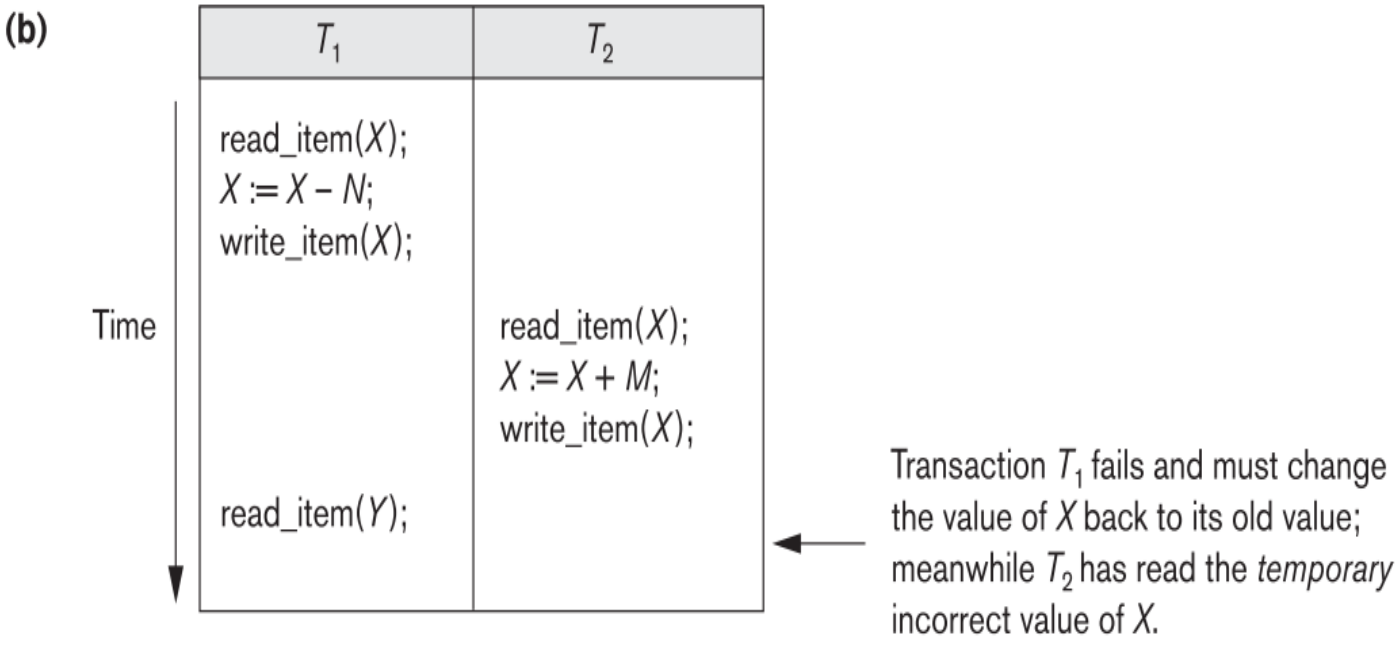
The Lost Update Problem:

- update done to a data item by a transaction is lost as it is overwritten by the update done by another transaction



The Temporary Update (or Dirty Read) Problem

- when one transaction updates a database item and then the transaction fails, then the updated item is read by another transaction before it is changed back to its original value



The Incorrect Summary Problem

- Consider a situation, where one transaction is applying the aggregate function on some records while another transaction is updating these records.
- The aggregate function may calculate some values before the values have been updated and others after they are updated.

(c)

T_1	T_3
	$sum := 0$; 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).

The Unrepeatable Read Problem

- When a transaction *T1* reads the same item twice and the item is changed by another transaction *T 2* between the two reads.
- Hence, *T1* receives different values for its two reads of the same item.

T1	T2
Read(X)	
	Read(X)
	X=X+10
	Write(X)
Read(X)	

WHY RECOVERY IS NEEDED

- Whenever a transaction is submitted to a DBMS
  - **Committed** - transaction are completed successfully and their effect is recorded permanently in the database.
  - **Aborted -** transaction does not have any effect on the database or any other transactions.

Types of Failures

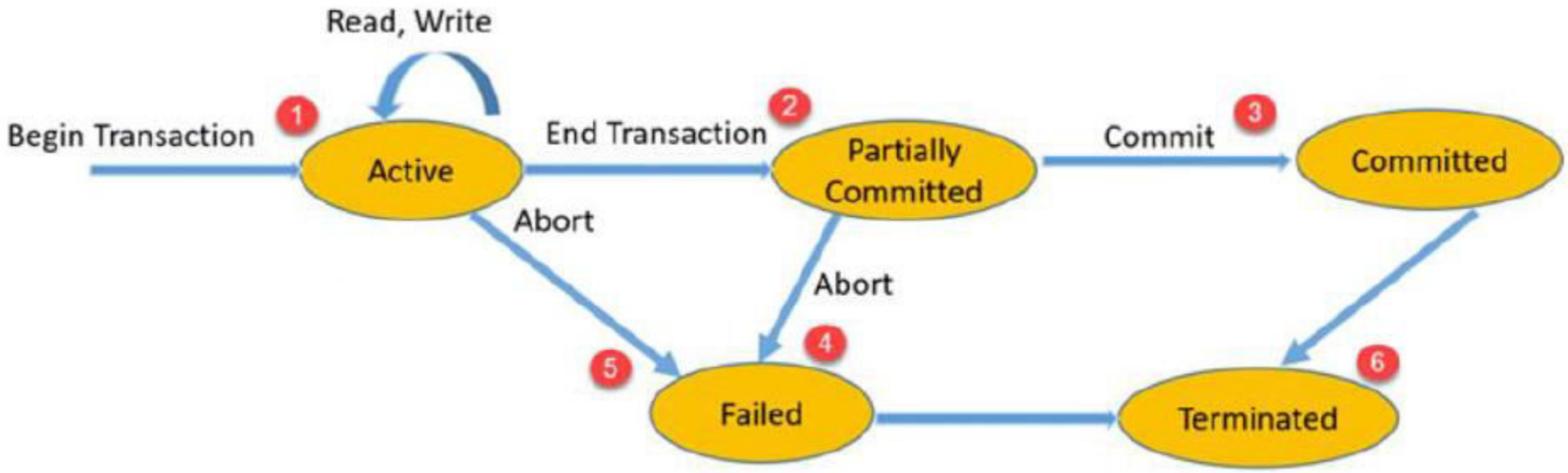
- Computer failure (system crash).
- Transaction or system error.
- Local errors or exception conditions detected by the transaction
- Disk failure
- Physical problems and catastrophes.

What to be done if a transaction fails

- If a transaction **fails** after executing some of its operations but before executing all of them, the operations already executed must be **undone**
- whenever a failure occurs, the system must quickly recover from the failure.

TRANSACTION STATES

State	Transaction types
Active State	A transaction enters into an active state when the execution process begins. During this state read or write operations can be performed.
Partially Committed	A transaction goes into the partially committed state after the end of a transaction.
Committed State	When the transaction is committed to state, it has already completed its execution successfully. Moreover, all of its changes are recorded to the database permanently.
Failed State	A transaction considers failed when any one of the checks fails or if the transaction is aborted while it is in the active state.
Terminated State	State of transaction reaches terminated state when certain transactions which are leaving the system can't be restarted.



State Transition Diagram for a Database Transaction

1. Once a transaction states execution, it becomes active. It can issue READ or WRITE operation.
2. Once the READ and WRITE operations complete, the transactions becomes partially committed state.
3. Next, some recovery protocols need to ensure that a system failure will not result in an inability to record changes in the transaction permanently. If this check is a success, the transaction commits and enters into the committed state.
4. If the check is a fail, the transaction goes to the Failed state.
5. If the transaction is aborted while it's in the active state, it goes to the failed state. The transaction should be rolled back to undo the effect of its write operations on the database.
6. The terminated state refers to the transaction leaving the system.

THE SYSTEM LOG

- To recover from failures the system maintains a log to keep track of all transaction operations
- The **log** is a file that is kept on disk
- The following are the types of entries—called **log records**—that are written to the log file

- **[start_transaction, T].** Indicates that transaction T has started execution.
- **[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.
- **[read_item, T, X].** Indicates that transaction T has read the value of database item X.
- **[commit, T].** Indicates that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
- **[abort, T].** Indicates that transaction T has been aborted.

PROPERTIES OF TRANSACTIONS

- Transactions should possess several properties, often called the **ACID**

- **Atomicity.**
  - A transaction is an atomic unit of processing
  - either the entire transaction takes place at once or doesn't happen at all

- **Consistency preservation.**
  - **i**f transaction is completely executed from beginning to end without interference from other transactions,
  - it should take the database from one consistent state to another.

- **Isolation.**
  - It states that all the transactions will be carried out and executed as if it is the only transaction in the system.
  - Existence of a transaction must not interfere other transaction

- **Durability 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.

Schedule of Transactions

- A Schedule S of n transactions T1, T2…Tn is an ordering of the operations of the transactions
- Operations from different transactions can be interleaved in the schedule *S*.
- For each transaction *Ti* that participates in the schedule S, the order in which *Ti occurs* in *S* must be in the same order in which they occur in *Ti*

A Schedule 'S' →

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*);	

- A shorthand notation for describing a schedule uses the symbols *b*, *r*, *w*, e, *c*, and *a*
  - *b*-> begin_transaction,
  - r-> read_item,
  - w->write_item,
  - e-> end_transaction,
  - c-> commit,
  - a-> abort

- *Sa*, can be written as follows in this notation:

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);	

- ***Sa*: *r*1(*X*); *r*2(*X*); *w*1(*X*); *r*1(*Y*); *w*2(*X*); *w*1(*Y*);**

Conflict Operations in Schedule

- Two operations in a schedule are said to **conflict** if they satisfy all three of the following conditions:
  - They belong to different transactions;
  - They access the same item X;
  - At least one of the operations is a write_item(X).

- *In the schedule Sa:*
- ***Sa: r1(X); r2(X); w1(X); r1(Y); w2(X); w1(Y);***
- **Conflicting operations**
  - *r*1(*X*) and *w*2(*X*)
  - *r*2(*X*) and*w*1(*X*)
  - *w*1(*X*) and *w*2(*X*)

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);	

- **Non Conflicting operations**
  - *r*1(*X*) and *r*2(*X*)) [both are read opns]
  - *w*2(*X*) and *w*1(*Y*) [operate on diff data items *X* and *Y*]
  - *r*1(*X*) and *w*1(*X*) [they belong to the same transaction]

Changing order of conflicting operations

- Two operations are conflicting if changing their order can result in a different outcome
- Eg in *r*1(*X*); *w*2(*X*) -> value is read by Transaction T1 before it is written by Transaction T2 (**read-write conflict**)
- This can be changed as *w*2(*X*); *r*1(*X*) -> the value of *X* is changed by *w*2(*X*) before it is read by *r*1(*X*)

- W1(x);w2(x) -> w2(x); w1(x)
  - This type is called **write-write conflict**
  - Last value of x will differ because in one case it is written by T2 and in other case by T1.
- Two read read operations are not conflicting because changing order makes no difference

CHARACTERIZING SCHEDULES BASED ON RECOVERABILITY

Recoverable Schedule

- Schedules in which transactions commit only after all transactions whose changes they read commit

T	T'
Read_Item(X) Commit	Write_item(X) Commit

- 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.

Irrecoverable Schedule

T1	T1's buffer space	T2	T2's Buffer Space	Database
				A=5000
R(A);	A=5000			A=5000
A=A-100;	A=4000			A=5000
W(A);	A=4000			A=4000
		R(A);	A=4000	A=4000
		A=A+500;	A=4500	A=4000
		W(A);	A=4500	A=4500
		Commit;		
Failure Point				
Commit;				

- The table shows a schedule with two transactions, T_1 reads and writes A and that value is read and written by T_2.
- T_2 commits. But later on, T_1 fails.
- So we have to rollback T_1. Since T_2 has read the value written by T_1, it should also be rollbacked.
- But we have already committed that.
- So this schedule is irrecoverable schedule.
- When T_j is reading the value updated by T_i and T_j is committed before committing of T_i, the schedule will be irrecoverable.

Example

- Consider a schedule Sc

Sc: r1(X); w1(X); r2(X); r1(Y); w2(X); c2; a1;

Is this Recoverable?

Soln : No **because T2 reads item X from T1, but T2 commits before T1 commits**

For the schedule to be recoverable, the c2 operation in Sc must be postponed until after T1 commits, as shown in Sd

Sd: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); c1; c2;

Cascading rollback

- Cascading Rollback (or cascading abort) to occur in some recoverable schedules, where an *uncommitted transaction has to be rolled back because it read an item from a transaction that failed.*

T1	T1's buffer space	T2	T2's Buffer Space	Database
				A=5000
R(A);	A=5000			A=5000
A=A-100;	A=4000			A=5000
W(A);	A=4000			A=4000
		R(A);	A=4000	A=4000
		A=A+500;	A=4500	A=4000
		W(A);	A=4500	A=4500
Failure Point				
Commit;				
		Commit;		

- The table shows a schedule with two transactions, T_1 reads and writes A and that value is read and written by T_2.
- But later on, T_1 fails. So we have to rollback T_1.
- Since T_2 has read the value written by T_1, it should also be rollbacked.
- As it has not committed, we can rollback T_2 as well. So it is recoverable with cascading rollback.
- *If T_j is reading value updated by T_i and commit of T_j is delayed till commit of T_i, the schedule is called recoverable with cascading rollback.*

Example

Se: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); a1; a2;

Cascadeless Recoverable rollback/ Cascadeless Schedule

- 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.*

T1	T1's buffer space	T2	T2's Buffer Space	Database
				A=5000
R(A);	A=5000			A=5000
A=A-100;	A=4000			A=5000
W(A);	A=4000			A=4000
Commit;				
		R(A);	A=4000	A=4000
		A=A+500;	A=4500	A=4000
		W(A);	A=4500	A=4500
		Commit;		

- The table shows a schedule with two transactions, T1 reads and writes A and commits and that value is read by T2.
- But if T1 fails before commit, no other transaction has read its value, so there is no need to rollback other transaction.
- So this is a Cascadeless recoverable schedule.
- *If Tj reads value updated by Ti only after Ti is committed, the schedule will be cascadeless recoverable.*

Example

r1(X); w1(X); r1(Y); w1(Y); c1; r2(X); w2(X); c2;

- Determine if the following schedule is recoverable. Is the schedule cascadeless?
- Justify your answer.
- r1(X), r2(Z), r1(Z), r3(X), r3(Y), w1(X), c1, w3(Y), c3, r2(Y), w2(Z), w2(Y), c2

Strict Schedule

- More restrictive type of schedule, called a **strict schedule**,
- Transactions can neither read nor write an item X until the last transaction that wrote X has committed (or aborted)

Serial Schedule

- Schedules in which the transactions are executed non-interleaved
  - a serial schedule is one in which no transaction starts until a running transaction has ended are called serial schedules.

T_1	T_2
R(A)	
W(A)	
R(B)	
	W(B)
	R(A)
	R(B)

Example: Consider the schedule involving two transactions T1 and T2. This is a serial schedule since the transactions perform serially in the order T1 —> T2

Non Serial Schedule

- This is a type of Scheduling where the operations of multiple transactions are interleaved.
  - Unlike the serial schedule where one transaction must wait for another to complete all its operation, in the non-serial schedule, the other transaction proceeds without waiting for the previous transaction to complete.
- This might lead to a rise in the concurrency problem.
- It can be of two types namely, Serializable and Non-Serializable Schedule.

Serial, Non-serial Schedules

X=90 & Y=90 ,N=3 and M=2

(a)

Time

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)

Time

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 B

(c)

Time

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 C

(d)

Time

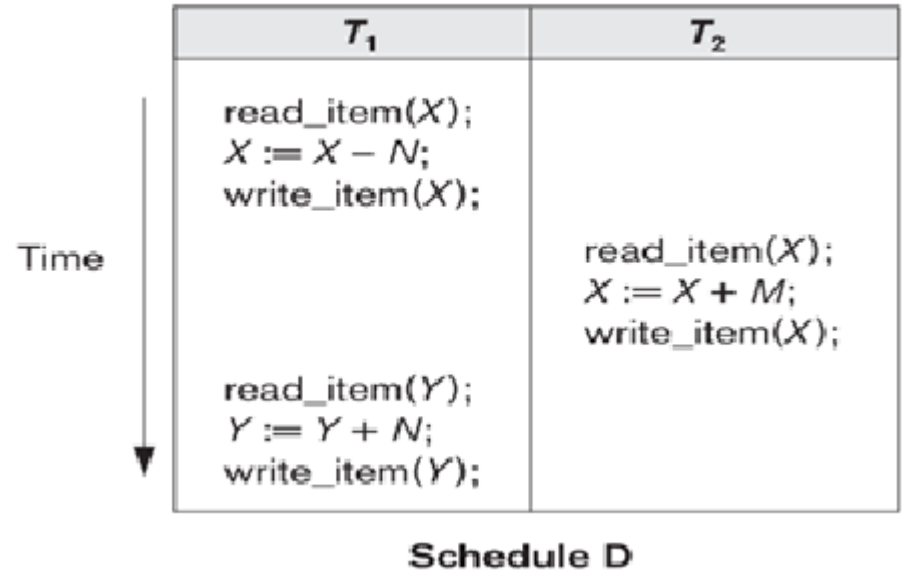
T_1	T_2

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);

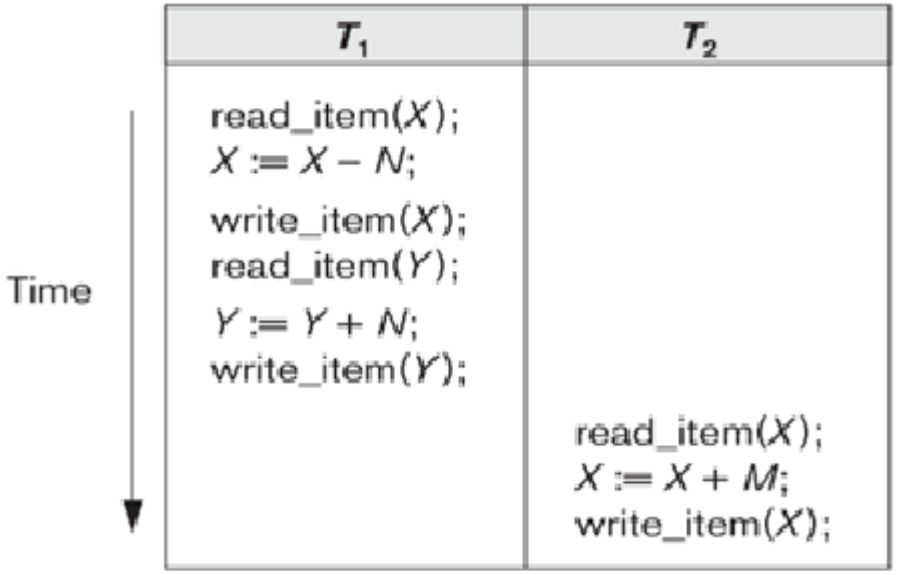
Schedule D

Serializable schedule

- A schedule *S* of *n* transactions is **serializable** <u>if it is *equivalent to some serial schedule* of the same *n* transactions</u>



Schedule D



Schedule A

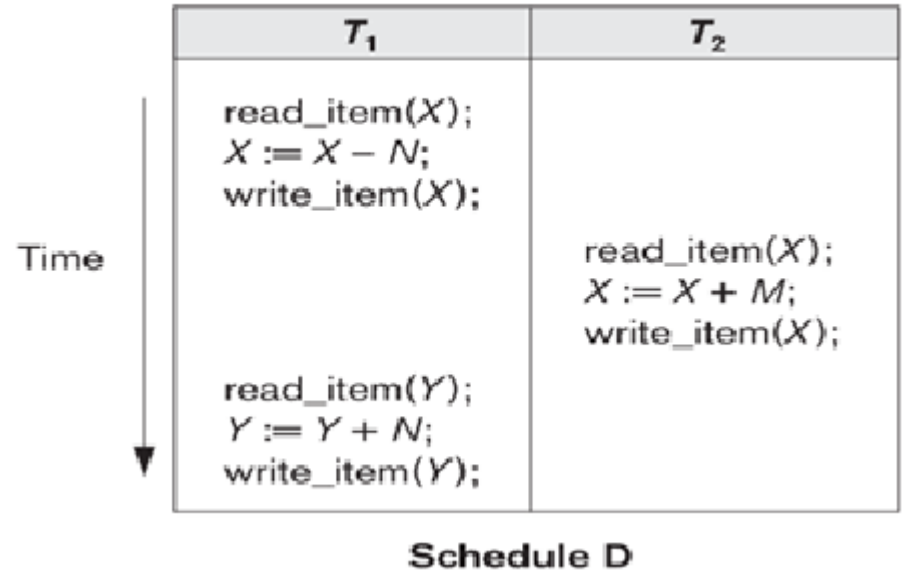
- Schedules A and B in Figure (a) and (b) are called *serial*
- Schedules C and D in Figure (c) are called *non-serial*

- The problem with serial schedules - they limit concurrency by prohibiting interleaving of operations.

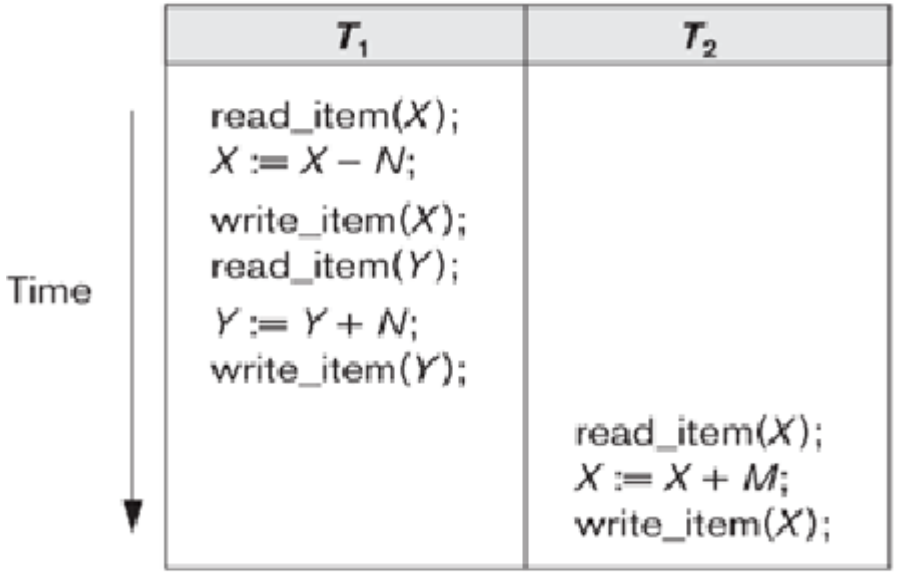
- In a serial schedule, if a transaction **waits for an I/O operation, we cannot switch the CPU processor to another transaction**, thus wasting valuable CPU processing time

Serializable schedule

- A schedule *S* of *n* transactions is **serializable** <u>if it is *equivalent to some serial schedule* of the same *n* transactions</u>



Schedule D



Schedule A

Equivalent Schedule

- For two schedules to be **equivalent**, the operations applied to each data item in both schedules must be *in the same order*.
- Two definitions of equivalence of schedules are generally used:
  - ***conflict equivalence* and**
  - ***view equivalence***

Conflict equivalence

- Two schedules are said to be **conflict equivalent** if the order of any two *conflicting operations* is the same in both schedules

T1	T2
Read_item(X)	
Write_item(X)	
	Read_item(X)
	Write_item(X)
Read_item(Y)	
	Read_item(Y)

T1	T2
Read_item(X)	
Write_item(X)	
	Read_item(X)
	Read_item(Y)
Read_item(Y)	
	Write_item(X)

conflict serializable

- A schedule S to be **conflict serializable** if it is (conflict) equivalent to some serial schedule S'
- If swapping of non conflicting instructions can make it into a serial schedule then it is known as **conflict serializable**

T1	T2
Read_item(X)	
Write_item(X)	
	Read_item(X)
	Write_item(X)
Read_item(Y)	
Write_item(X)	
	Read_item(Y)
	Write_item(Y)

T1	T2
Read_item(X)	
Write_item(X)	
Read_item(Y)	
Write_item(X)	
	Read_item(X)
	Write_item(X)
	Read_item(Y)
	Write_item(Y)

- According to this definition, schedule D in Figure (c) is equivalent to the serial schedule A in Figure (a). Schedule C in Figure (c) is not equivalent to either of the two possible serial schedules A and B, and hence is *not serializable*.

View Equivalent

- Two schedules T1 & T2 are view equivalent, if they satisfy the foll condition
  - Initial Read : Initial read of each data item in transactions must match in both schedules
  - Final Write : Final write operation on each data item must match in both the schedules
  - Update Read : If in a schedule S1 , the transaction T1 is reading a data item updated by T2 then in schedule S2 T1 should read the value after updated by T2

View Serializability

1. Initial Read

T1	T2
Read (A)	
	Write(A)

Schedule 1

T1	T2
	Write(A)
Read (A)	

Schedule 2

2. Updated Read

T1	T2	T3
Write(A)		
	Write(A)	
		Read(A)

Schedule 1

T1	T2	T3
	Write(A)	
Write(A)		
		Read(A)

Schedule 2

3. Final Write

T1	T2	T3
Write(A)		
	Read(A)	
		Write(A)

Schedule 1

T1	T2	T3
	Read(A)	
Write(A)		
		Write(A)

Schedule 2

Testing for Conflict Serializability of a Schedule

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.

(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);

Time

Schedule A

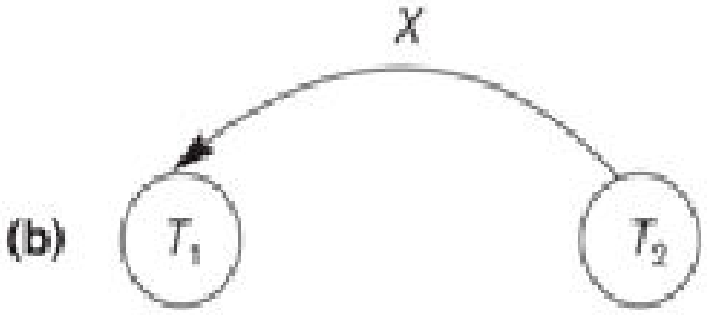
(b)

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);

Time

Schedule B

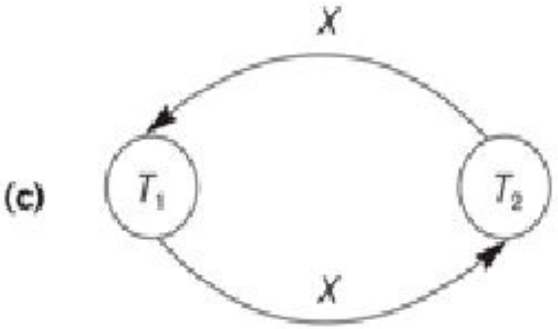


(c)

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);	

Time

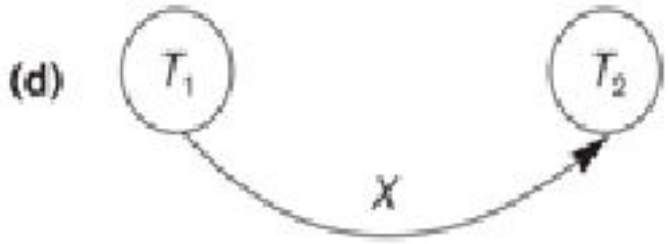
Schedule C



Time

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



LOCK-BASED PROTOCOLS

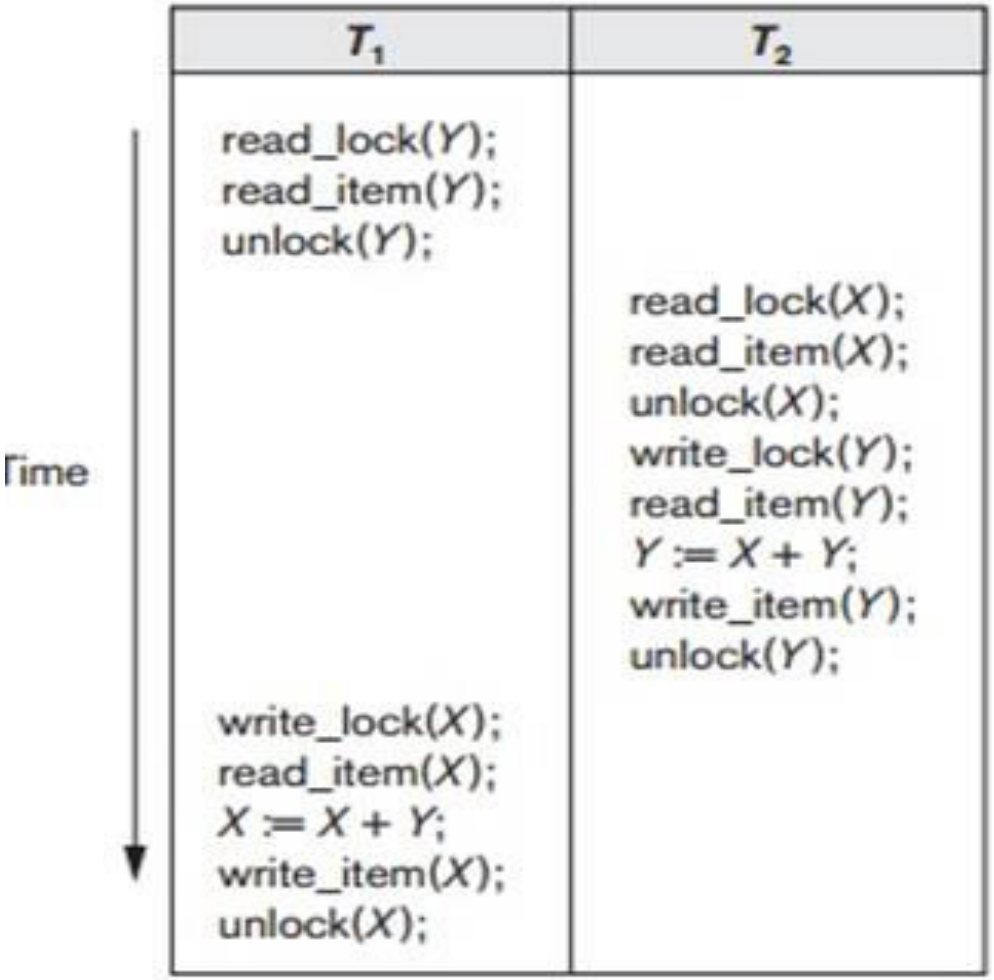
LOCK-BASED PROTOCOLS

- To ensure isolation data items be accessed in a mutually exclusive manner; i.e , while one transaction is accessing a data item, no other transaction can modify that data item
- The 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.
- A **lock** is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it.
- Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

Two Modes of Lock

- **Shared**: If a transaction T_i has obtained a shared-mode lock (denoted by S) on item Q, then T_i can read, but cannot write, Q.
- **Exclusive**: If a transaction T_i has obtained an exclusive-mode lock (denoted by X) on item Q, then T_i can both read and write Q.

- The transaction can proceed with the operation only after granting the lock to the transaction.
- 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.


T_1
T_2
read_lock(Y);
read_item(Y);
unlock(Y);
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);
Time

Compatibility function on Lock modes

- Let A and B represent arbitrary lock modes. Suppose that a transaction Ti requests a lock of mode A on item Q on which transaction Tj currently holds a lock of mode B.
- If transaction Ti can be granted a mode A lock on Q immediately, in spite of the presence of the mode B lock, then we say **mode A is compatible with mode B**

	S	X
S	true	false
X	false	false

Granting of Locks

- When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted.

- Suppose a transaction T2 has a shared lock on a data item, and another transaction T1 requests an exclusive-mode lock on the data item. Clearly, T1 has to wait for T2 to release the shared-mode lock.

- Meanwhile, a transaction T3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to T2, so T3 may be granted the shared-mode lock.

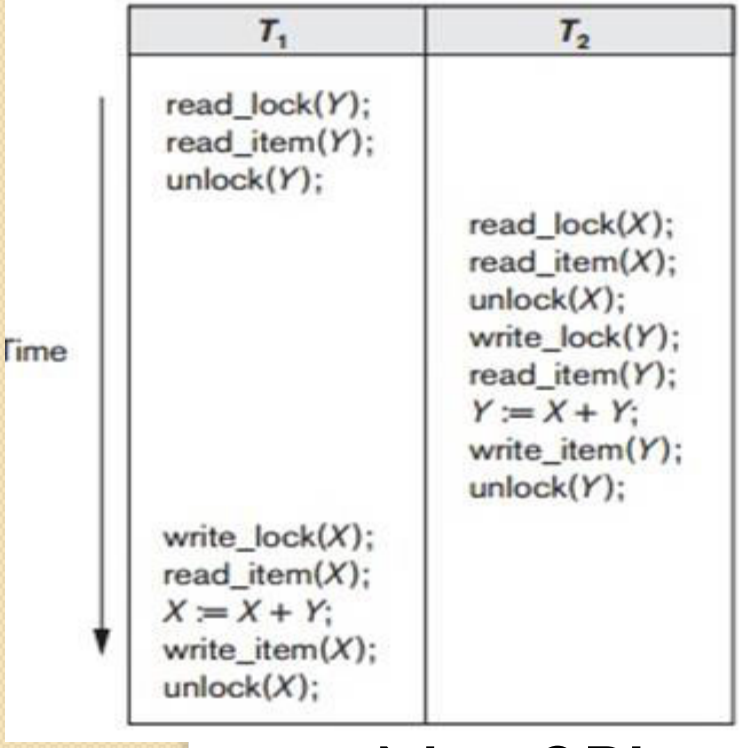
- T1 never gets the exclusive-mode lock on the data item. The transaction T1 may never make progress, and is said to be **starved**.

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

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

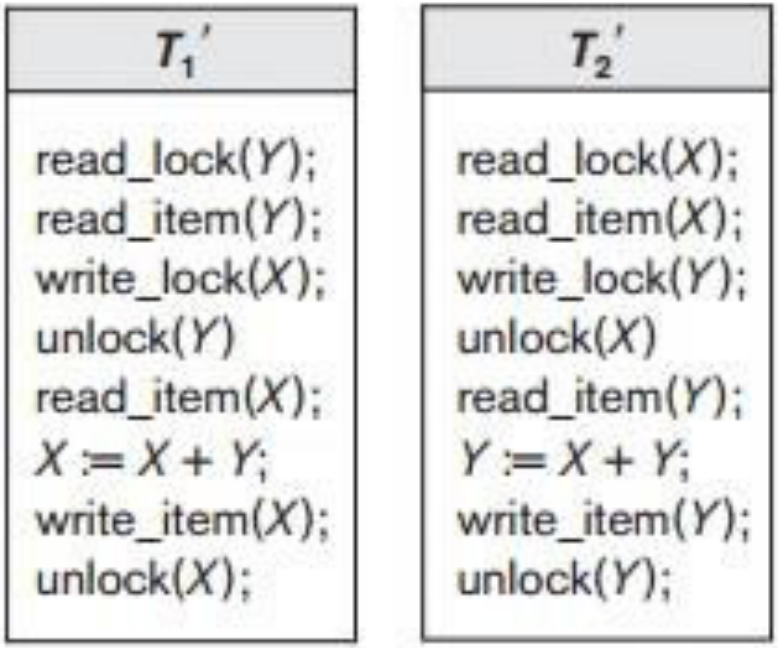
TWO-PHASE LOCKING PROTOCOL

- A transaction can be divided into two phases:
- An **expanding** or **growing (first) phase**, during which new locks on items can be acquired but none can be released;
- **shrinking (second) phase,** - existing locks can be released but no new locks can be acquired
- The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the **lock point**


T_1
T_2
read_lock(Y);
read_item(Y);
unlock(Y);
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);
Time

- Not 2PL

.


T_1'
read_lock(Y);
read_item(Y);
write_lock(X);
unlock(Y)
read_item(X);
X := X + Y;
write_item(X);
unlock(X);
T_2'
read_lock(X);
read_item(X);
write_lock(Y);
unlock(X)
read_item(Y);
Y := X + Y;
write_item(Y);
unlock(Y);

2PL

Example of 2PL

- Two-phase locking does not ensure freedom from **deadlock**

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)	

Variation to 2PL

- **Conservative 2PL**
- **Strict 2PL**
- **Rigorous 2PL**

Conservative 2PL

- **Conservative 2PL** (or **static 2PL**) requires a transaction to lock all the items it accesses *before the transaction begins execution*

Eg

- LOCK-X (A)
  LOCK-S (B)
  Read(A)
  Read(B)
  Write(A)
  UNLOCK (A)
  COMMIT
  UNLOCK (B)

Strict 2PL

- **Strict 2PL**, in which a transaction T does not release any of its exclusive (write) locks until after it commits or aborts
- Strict 2PL is not deadlock-free.

LOCK-X (A)
LOCK-S (B)
Read(A)
Read(B)
Write(A)
UNLOCK (B)
COMMIT
UNLOCK (A)

Rigorous 2PL

- A more restrictive variation of strict 2PL
- a transaction *T* does not release any of its locks (exclusive or shared) until after it commits or aborts, and so it is easier to implement than strict 2PL

LOCK-X (A)
LOCK-S (B)
Read(A)
Read(B)
Write(A)
COMMIT
UNLOCK (A)
UNLOCK (B)

DATABASE MODIFICATION

- A transaction creates a log record prior to modifying the database.
- The log records allow the system to **undo** changes made by a transaction if it gets aborted.
- They allow the system to **redo** changes made by a transaction if the transaction has committed but the system crashed before those changes are made permanently on disk.

Steps a transaction takes in modifying a data item are:

- The transaction performs some computations in main memory.
- The transaction modifies the data block in the buffer in main memory holding the data item.
- The database system executes the output operation that writes the data block to disk.

Deferred modification and Immediate modification

- A transaction modifies the database if it performs an update on the disk itself.
- Updates to the 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**

- Database modifications must be preceded by the creation of a log record, the system has available both the old value prior to the modification and the new value after modification.
- 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.

CHECKPOINT

- Another type of entry in the log called **checkpoint**.
- A[checkpoint] record is written into the log periodically , at that point the system writes out all DBMS buffers that have been modified to the disk.
- The list of transaction ids for active transactions is included in the checkpoint record, so that these transactions can be easily identified during recovery.

Sequence of actions in Checkpointing

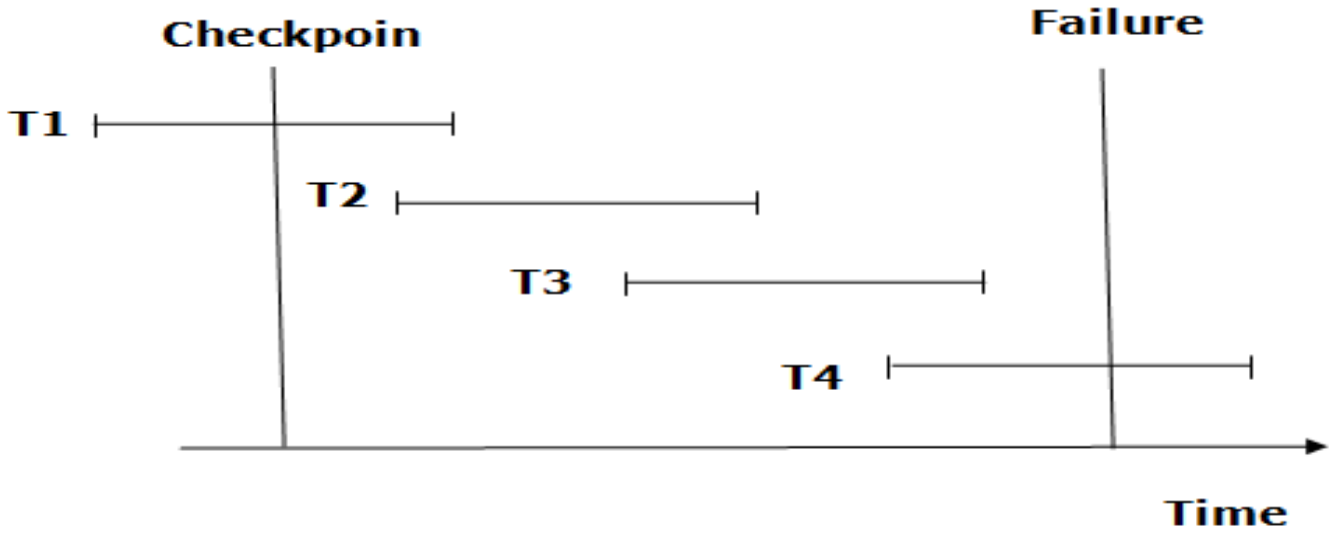
- Output onto stable storage all log records currently residing in main memory.
- Output to the disk all modified buffer blocks.
- Output onto stable storage a log record of the form <**checkpoint L**>, where L is a list of transactions active at the time of the checkpoint.

- **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**.

- The presence of a <checkpoint L> record in the log allows the system to simplify its recovery procedure.

- **After a system crash** has occurred, the system examines the log to find the last <checkpoint L> record.

- During recovery we need to consider only the most recent transactions Ti that started before the checkpoint , and transaction that started after Ti
  - Scan backwards from end of log to find the most recent <check point, L> record
  - Continue scanning backward till a record <Ti Start> is found.
  - Need only consider the those transaction that started before checkpoint
  - For all transactions with no <Ti Commit>, execute undo(Ti)
  - For all transactions starting from Ti or later with a <Ti Commit> , execute redo(Ti)

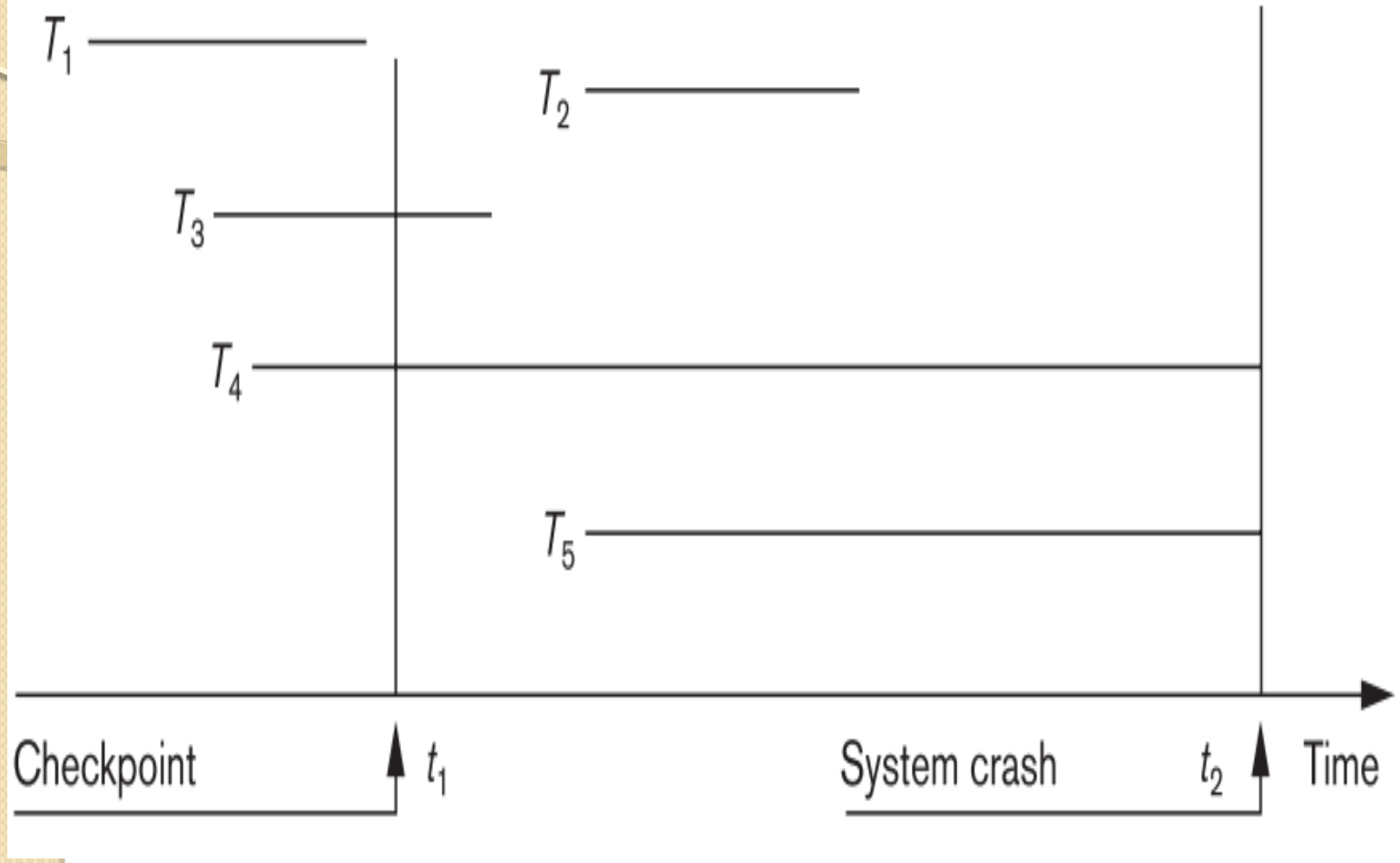


- T1 can be ignored (updates already output to disk due to checkpoint
- T2 & T3 to redone
- T4 to be undone.

NO-UNDO/REDO recovery based on DEFERRED DATABASE MODIFICATION

- In deferred update it postpone any actual updates to the database on disk until the transaction commits
- After the transaction reaches its commit point and the log is force written to disk, the updates are recorded in the database.
- If a transaction fails before reaching its commit point, there is no need to undo any operations because the transaction has not affected the database on disk.

- When a checkpoint is reached, all the transactions which are committed after the last checkpoint are written to the disk.
- If the system crashes in between the checkpoints, REDO all the WRITE operations of the committed transactions from the log,
- The transactions that are active and did not commit are effectively cancelled and must be resubmitted

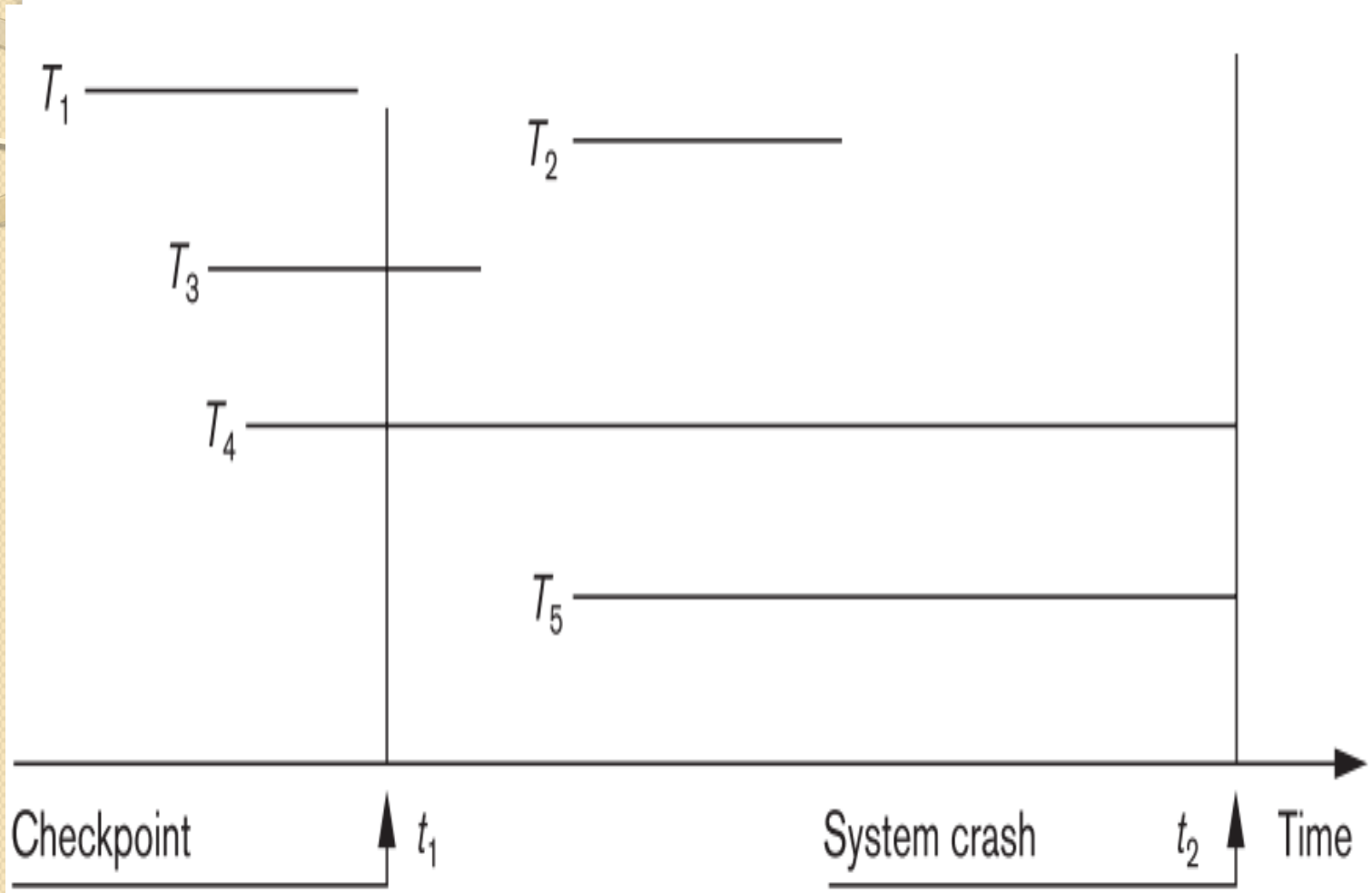

T_1
T_2
T_3
T_4
T_5
Checkpoint
t_1
System crash
t_2
Time

- When the checkpoint was taken at time *t*1, transaction *T*1 had committed, whereas transactions *T*3 and *T*4 had not.
- Before the system crash at time *t*2, *T*3 and *T*2 were committed but not *T*4 and *T*5.
- There is no need to redo the write operations of transaction *T*1—or any transactions committed before the last checkpoint time *t*1. The *T*2 and *T*3 must be redone,
- however, because both transactions reached their commit points after the last checkpoint. Transactions *T*4 and *T*5 are ignored:

UNDO/REDO recovery based on IMMEDIATE DATABASE MODIFICATION

- The immediate database modification technique allows database modifications to be updated to the database while the transaction is still in active state.
- Data modifications written by active transactions are called uncommitted modifications.
- In case of failure, system uses old value in log records to restore modified data items to the value they had prior to the start of the transaction.

- When a checkpoint is reached, all the transactions which are committed as well as the active transactions are written to the disk. If the system crashes in between the checkpoints do the following:
  - Undo all the write operations of the *active* (uncommitted) transactions.
  - Redo all the write operations of the *committed* transactions from the log, in the order in which they were written into the log.


T_1
T_2
T_3
T_4
T_5
Checkpoint
t_1
System crash
t_2
Time

- Figure illustrates a timeline for a possible schedule of executing transactions.
- When the checkpoint was taken at time *t*1, all transaction are updated on the disk. Before the system crash at time *t*2, *T*3 and *T*2 were committed but not *T*4 and *T*5.
- There is no need to redo the write operations of transaction *T*1—or any transactions committed before the last checkpoint time *t*1.
- The write operations of *T*2 and *T*3 must be redone, because both transactions reached their commit points after the last checkpoint.
- Transactions *T*4 and *T*5 must be roll backed (undo operation) since their write operations were recorded in the database on disk under the immediate update protocol.

Introduction to NoSQL Databases

What is No SQL

- It is a non-relational Data Management System,
- Never provide tables with flat fixed-column records
- It does not require a fixed schema.
- Horizontal scaling is possible.
- The major purpose of using a NoSQL database is for distributed data stores with very large data storage needs.
- Much faster performance than relational DB

- The concept of NoSQL databases became popular with Internet giants like Google, Facebook, Amazon, etc. who deal with huge volumes of data.

Types of No SQL databases

- Key-value Pair Based
- Column database
- Graphs based
- Document-oriented

Column database

- It is a DBMS that store data in columns instead of rows
- In a column DB, all the column1 values are physically together followed by all the column 2 values
- Some examples of column-store databases include Casandra, CosmoDB, Bigtable, and HBase.

ID	Name	Grade	GPA
001	John	Senior	4.00
002	Karen	Freshman	3.67
003	Bill	Junior	3.33

- In a row oriented DBMS the data will be stored as

(001,John,Senior,4.00 ; 002,Karen, Freshman, 3.67 ; 003, Bill, junior, 3.33)

- In a Column oriented DBMS data will be stored as

(001,002,003; John,Karen,Bill ; Senior,Freshman,Junior; 4.00,3.67,3.33)

- Benefits
  - It is self indexing
  - It uses less disk space compared that a RDMS containing the same data

Document- database

- Document DB make it easier for developer to store and Query data in a DB by using the some document model format .
- Document DB are efficient for strong catalogue
- Store semi – structure data as document typically in JSON or XML

- Examples - MongoDB, Cosmos DB, PostgreSQL, RavenDB.

```
{
    "_id": "sammyshark",
    "firstName": "Sammy",
    "lastName": "Shark",
    "email": "sammy.shark@digitalocean.com",
    "department": "Finance"
}
```

- Notice that the document is written as a JSON object.
- JSON is a human-readable data format that has become quite popular in recent years.
- While many different formats can be used to represent data within a document database, such as XML . JSON is one of the most common choices.

Relational vs Document DB

Relational

ID	first_name	last_name	cell	city	year_of_birth	location_x	location_y
1	'Mary'	'Jones'	'516-555-2048'	'Long Island'	1986	'-73.9876'	'40.7574'

ID	user_id	profession
10	1	'Developer'
11	1	'Engineer'

ID	user_id	name	version
20	1	'MyApp'	1.0.4
21	1	'DocFinder'	2.5.7

ID	user_id	make	year
30	1	'Bentley'	1973
31	1	'Rolls Royce'	1965

MongoDB

```
first_name: "Mary",
last_name: "Jones",
cell: "516-555-2048",
city: "Long Island",
year_of_birth: 1986,
location: {
        type: "Point",
        coordinates: [-73.9876, 40.7574]
},
profession: ["Developer", "Engineer"],
apps: [
 { name: "MyApp",
   version: 1.0.4 },
 { name: "DocFinder",
   version: 2.5.7 }
],
cars: [
   { make: "Bentley",
     year: 1973 },
   { make: "Rolls Royce",
     year: 1965 }
]
```

Benefits

- With a high level of control over the data structure, document databases can adapt to new emerging requirements.
- New fields can be added right away and existing ones can be changed any time.
- Ability to manage structured and unstructured data
- Scalability by design: document databases are designed as distributed systems , it allow to scale horizontally (meaning that you split a single database up across multiple servers).

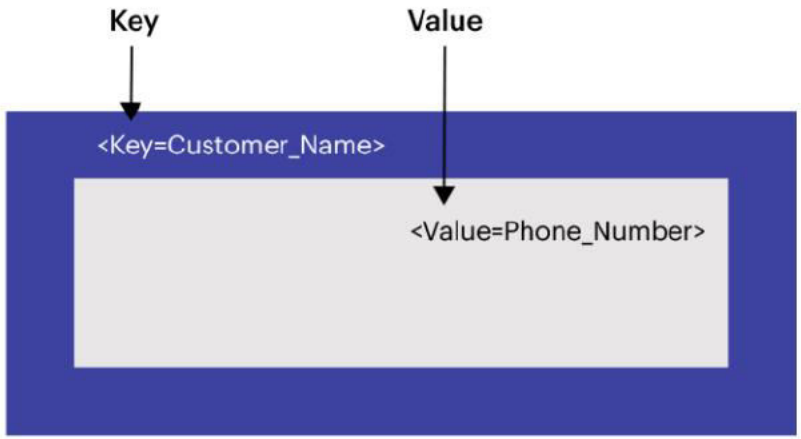
Key-value Pair Based

- A key-value database (sometimes called a key-value store) uses a simple key-value method to store data.
- This type of NoSQL database implements a hash table to store unique keys along with the pointers to the corresponding data values.
- A value can be stored as an integer, a string, JSON, or an array—with a key used to reference that value.

- Values cannot be queried or searched upon. Only the key can be queried.
- They are easy to design and implement.

Phone directory

Key	Value
Paul	(091) 9786453778
Greg	(091) 9686154559
Marco	(091) 9868564334



A simple example of key-value data store.

- When your application needs to handle lots of small continuous reads and writes, that may be volatile. Key-value databases offer fast in-memory access.

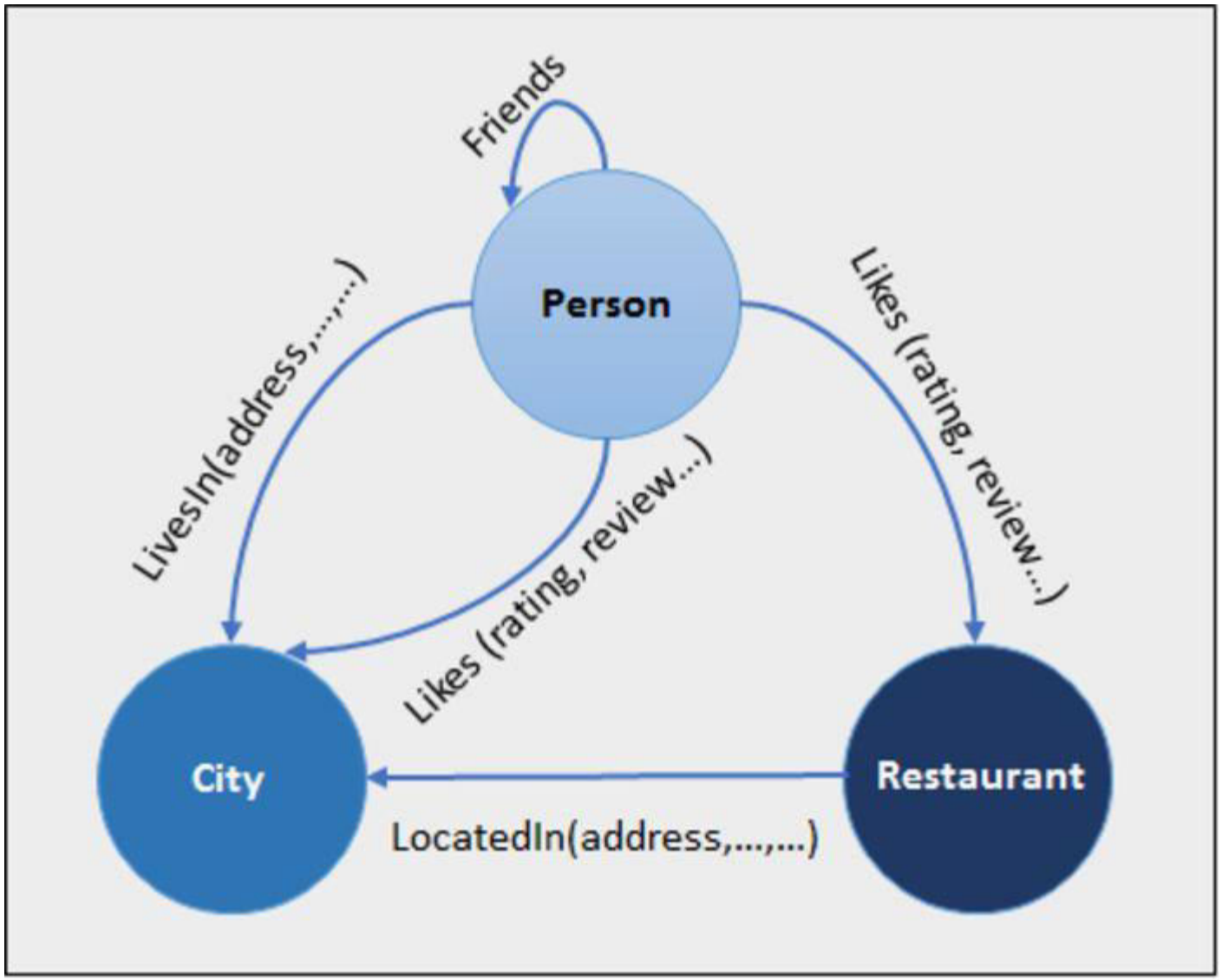
- When storing basic information, such as customer details; storing shopping-cart contents, product categories, e-commerce product details

Example of Key-Value databases

- Redis, Dynamo, Riakare some NoSQL examples of key-value store DataBases.

Graph Based No SQL

- They primarily are composed of two components
- The Node
  - This is the actual piece of data itself.
  - It can be the number of viewers of a youtubevideo, the number of people who have read a tweet, or it could even be basic information such as people's names, addresses, and so forth.
- The Edge
  - This explains the actual relationship between two nodes.
  - edges might also have directions describing the flow of said data.
- Examples of Graph Databases
  - Neo4j, ArangoDB


Friends
Person
LivesIn(address,...,...)
Likes (rating, review...)
Likes (rating, review...)
City
Restaurant
LocatedIn(address,...,...)