Concurrency Control

Mirunalini.P

SSNCE

April 1, 2020

Table of Contents

Session Objective

- Locking Mechanisms
- Types of Locks
- Two phase protocol

Session Outcome

At the end of this session, participants will be able to

- Understand Locking Mechanisms
- Understand Types of Locks such as Binary Lock and Shared/Exclusive Lock

What is Concurrency Control?

- Concurrency control is the procedure in dbms for managing multiple transactions without conflicting with each other.
- Concurrency control is used
  - To address conflicts which mostly occur with a multiple transactions.
  - To ensure database transactions are performed concurrently without violating data integrity

Potetional Problems of Concurrency

- Lost Update Problem
- Dirty Read
- Incorrect Summary Problem
- Non-Repeatable Read

Purpose of Concurrency Control

- To enforce Isolation (through mutual exclusion) between conflicting transactions.
- To preserve database consistency through consistency preserving execution of transactions.
- To resolve read-write and write-write conflict issuses.
- Concurrency control helps to ensure serializability.

Concurrency control protocols ensures serializability of schedules.

Concurrency Control Protocols

There are different Concurrency Control Protocols that ensure serializability of schedules

- Locking - Based Protocols
- Timestamp- Based protocols
- Validation- Based Protocols

Locking Technique

Main techniques used to control concurrent execution of transactions are based on the concept of locking data items.

- Lock is an operation which secures permission to **Read or Write** a data item for a transaction.
- A lock a variable associated with a data item that describes the status of the item
- There is **one Lock** for each data item in the database.
- **Lock (X):** Data item X is locked in behalf of the requesting transaction
- **Unlock (X):** Data item X is made available to all other transactions.
- Lock and Unlock are atomic operations

Types Of Locks

- **Binary locks:** Only two states of a lock; too simple
- **Shared/exclusive locks:** Provide more general locking capabilities and are used in practical database locking schemes. (Read Lock as a shared lock, Write Lock as an exclusive lock).

Binary Locks

- A binary lock can have two states or values: **locked and unlocked** (or 1 and 0, for simplicity).
- A binary lock enforces mutual exclusion on the data item;
- If the value of the **lock on X is 1, item X cannot be accessed** by a database operation that requests the item.
- If the value of the **lock on X is 0, the item can be accessed** when requested.
- **LOCK(X)** : Indicates the current value (or **state**) of the lock associated with item X

Binary locks

Every transaction must obey the following rules.
Rules are enforced by the **LOCK MANAGER**

- A transaction T must issue the operation lock_item(X) before any read_item(X) or write_item(X) operations are performed in T.
- A transaction T must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed in T.
- A transaction T will not issue a lock_item(X) operation if it already holds the lock on item X .
- A transaction T will not issue an unlock_item(X) operation on X unless it already holds the lock on item X.

Binary Lock - Algorithm

lock_item (*X*):

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

unlock_item (*X*):

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

Disadvantages of Binary lock

- Binary lock is too restrictive for database items because atmost one transaction can hold a lock on a given item.
- Binary locking system cannot be used for practical purpose.

Shared / Exclusive Lock

Two locks modes Shared (read) and Excusice (Write).

- **Shared Lock(S)**: More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.
- **Exclusive mode or Write lock (X)** Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.

Shared/Exclusive or Read/Write locks

A lock associated with an item X, **LOCK (X)**, now has three possible states: read-locked, write-locked, or unlocked.

- A **read-locked** item is also called **share-locked** because other transactions are allowed to read the item
- A **write-locked** item is called **exclusive-locked** because a single transaction exclusively holds the lock on the item.
- An **unlock** item is issued after all **read_item(X) and write_item(X)** operations are completed.

	S	X
S	true	false
X	false	false

Compatibility Graph

Shared/Exclusive or Read/Write locks Rules

- A transaction T must issue the operation **read_lock(X) or write_lock(X)** before any read_item(X) operation is performed in T.
- A transaction T must issue the operation **write_lock(X)** before any write_item(X) operation is performed in T.
- A transaction T must issue the operation unlock(X) after all **read_item(X) and write_item(X)** operations are completed in T.
- A transaction T will not issue a read_lock(X) operation if it already holds a read lock or a write lock on item X. **(exceptions: downgrading of lock from write to read)**
- A transaction T will not issue a write_lock(X) operation if it already holds a read lock or write lock on item X. **(exceptions: upgrading of lock from read to write)**

Lock Manager

- Lock Manager: Managing locks on data items.
- **Lock table**: Lock manager uses it to store the identity of transaction locking a data item, the data item, lock mode and pointer to the next data item locked.
- One simple way to implement a lock table is through linked list.
- Ex1 (Binary Lock): <Data_item_name, Lock, Locking_transaction>
- Ex2 (S/X Lock): Transaction-ID Data-item Lock-mode Ptr-to-next-data-item

T1 **X1** **Read** **Next**

read_Lock(X):

Pseudocode: read_Lock(x)

```
B: if LOCK(x)= ''unlocked''
then begin Lock(X) <- ``read-locked'';
no_of_reads(X) <- 1
end

elseif LOCK(X)=''read-locked''
then no_of_reads(X) <- no_of_reads(X)+1
elsebegin

wait(untill Lock(X)=''unlocked''
and the lock manager wakes up the transaction)
go to B
end;
```

write_Lock(X):

Pseudocode: write_Lock(x)

```
B: if LOCK(x)= ''unlocked''
then Lock(X) <- ``write-locked'';

else begin

wait(untill Lock(X)=''unlocked''
and the lock manager wakes up the transaction)
go to B
end;
```

UnLock(X):

Pseudocode:UnLock(X)

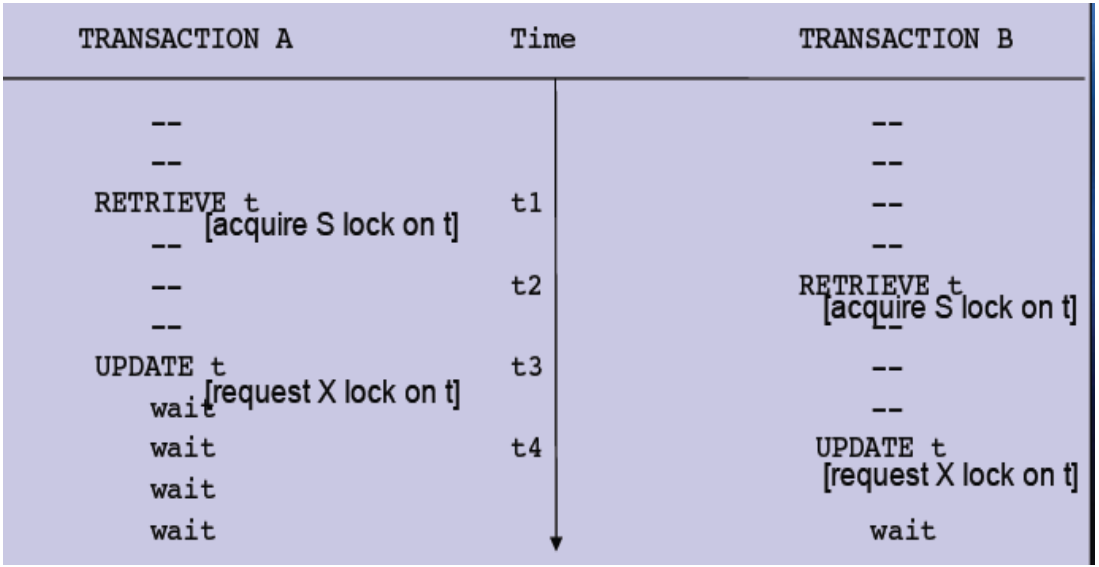
```
if LOCK(x)= ''write-locked''
then begin Lock(X) <- ``unlocked `` ;
wakeup one of the waiting transactions, if any
End

else if LOCK(X)=''read-locked''
then begin
no_of_reads(X) <-- no_of_reads(X)-1
if no_of_reads(X)= 0
then begin LOCK(X)=''unlocked'';
wakes up the transaction if any;
end
end;
```

Conversion of Locks

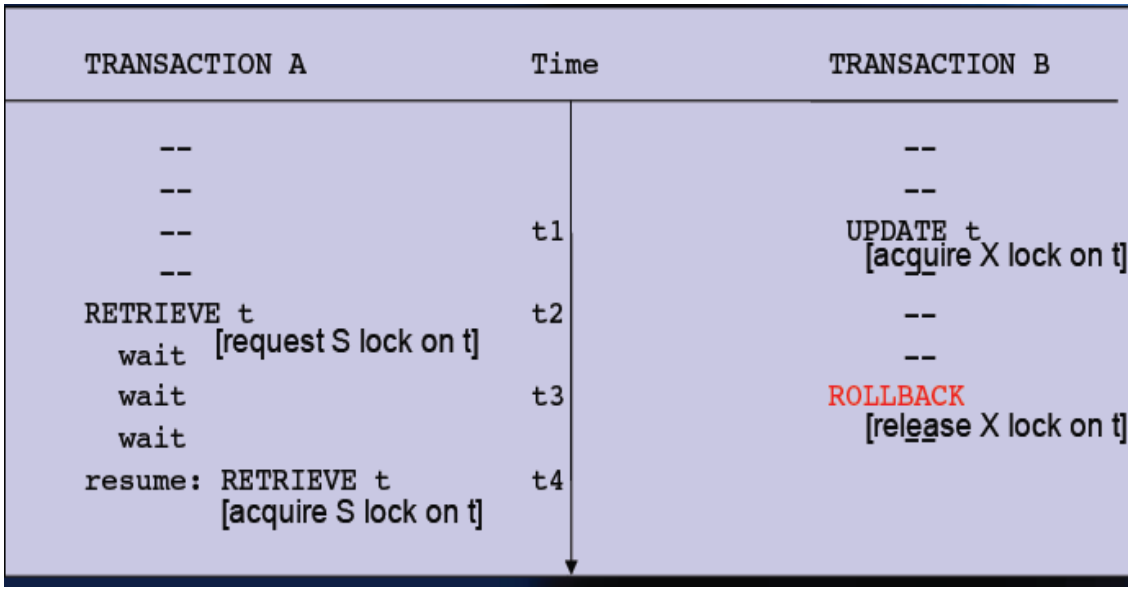
- **Lock conversion**: A transaction that already holds a lock on item X is allowed under certain conditions to convert the lock from one locked state to another.
- **Upgrading Lock**: It is possible for a transaction T to issue a read_lock (X) and then later to upgrade the lock by issuing a write_lock (X) operation.
- If T is the only transaction holding a read lock on X at the time it issues the write_lock (X) operation, the lock can be upgraded; otherwise, the transaction must wait.
- It is also possible for a transaction T to issue a write_lock (X) and then later to downgrade the lock by issuing a read_lock (X) operation.

Lost-Update

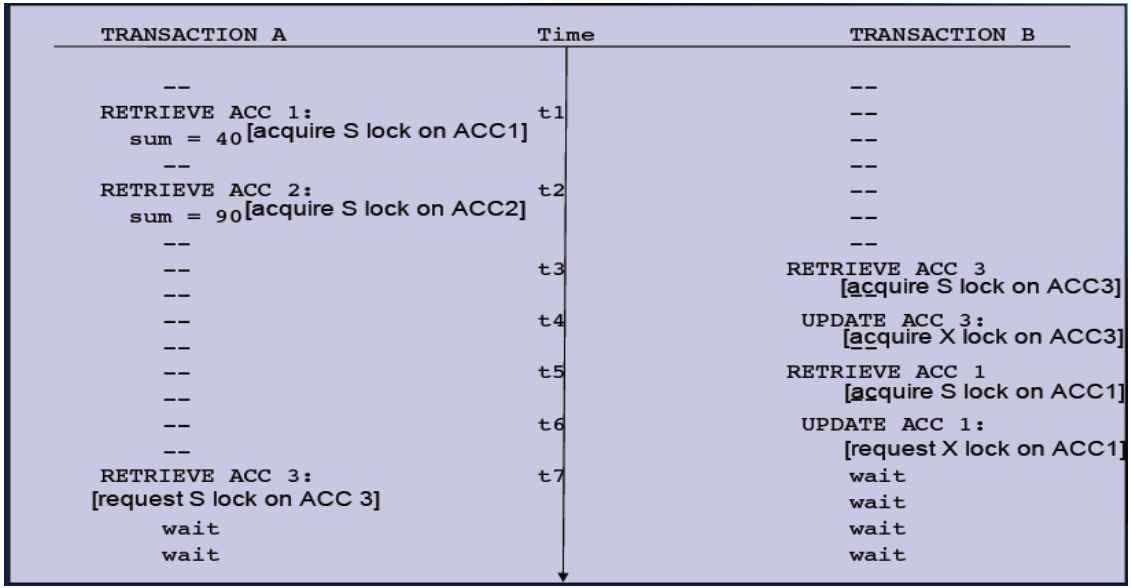


Under strict two-phase locking protocol, the lost update problem is resolved.

Dirty Read Problem



Incorrect Summary Problem



Guaranteeing Serializability by Two-Phase Locking

A transaction is said to follow the two-phase locking protocol if all **locking operations (read_lock, write_lock) precede** the first **unlock operation** in the transaction.

Two Phases:

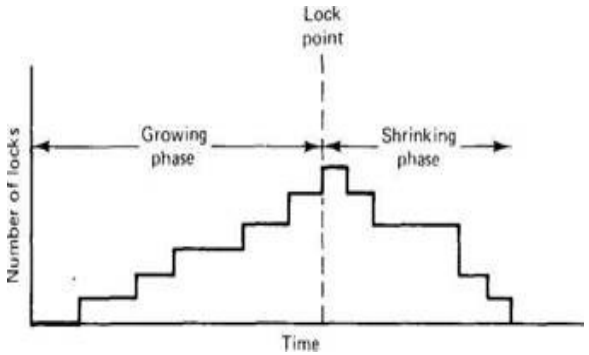
- **Expanding or Growing Phase**: During which new locks on items can be acquired but none can be released;
- **Shrinking (second) phase**: During which existing locks can be released but no new locks can be acquired.

With Lock Conversion:

- **Upgrading of locks:** (from read-locked to write-locked) must be done during the expanding phase,
- **Downgrading of locks:** (from write-locked to read-locked) must be done in the shrinking phase.

Two Phase Locking Protocol

- When transaction starts executing, it is in the locking phase, and it can request locks on new items or upgrade locks.
- A transaction may be blocked (forced to wait) if a lock request is not granted.
- Once the transaction unlocks an item (or downgrades a lock), it starts its shrinking phase and can no longer upgrade locks or request new locks.
- The combination of locking rules and 2-phase rule ensures **serializable schedules**



Two phase Locking protocol - serializability

X must remain locked by T until all items that the transaction needs to read or write have been locked; only then can X be released by T.

Another transaction seeking to access X may be forced to wait, even though T is done with X;

Conversely, if Y is locked earlier than it is needed, another transaction seeking to access Y is forced to wait even though T is not using Y yet.

Theorem: If every transaction in a schedule follows the 2PL rules, the schedule must be serializable.

The 2PL suffers from the problem of **deadlock and cascading rollback**

Table 1: Transactions that do not obey two-phase locking.

T1	T2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
X := X + Y;	Y := X + Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Initial values: X=20, Y=30
Result serial schedule $T1$ followed by $T2$: X=50, Y=80
Result of serial schedule $T2$ followed by $T1$: X=70, Y=50

Table 2: Transactions that obey two-phase locking.

$T1'$	$T2'$
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
write_lock(X);	write_lock(Y);
unlock(Y);	unlock(X);
read_item(X);	read_item(Y);
X := X + Y;	Y := X + Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

They produce Deadlock

Deadlock

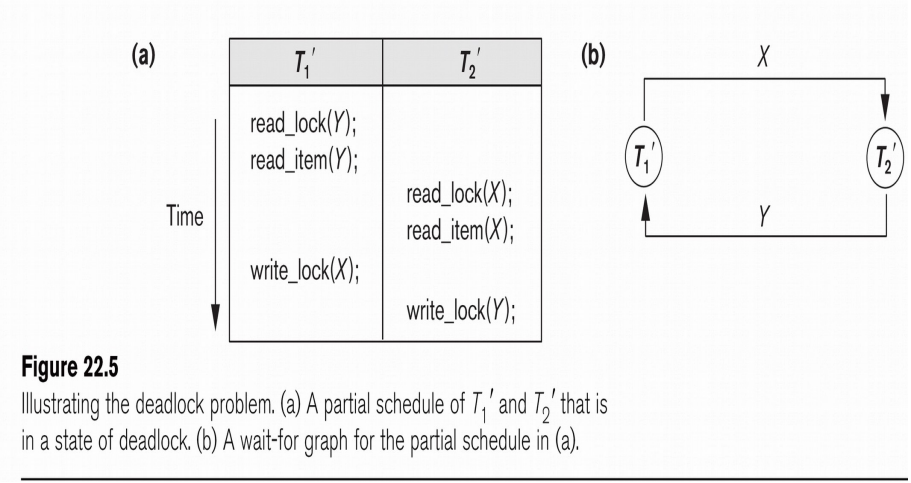


Figure 22.5
Illustrating the deadlock problem. (a) A partial schedule of T_1' and T_2' that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

A state of deadlock would occur instead because $T2'$ would try to lock Y (which is locked by $T1'$ in conflicting mode) and forced to wait, then $T1'$ would try to lock X (which is locked by T2 in conflicting mode) and forced to wait neither transaction can continue to unlock the item they hold as they are both blocked (waiting)

Types of Two-phase Locking

Two-phase policy generates different locking algorithms

- **Basic 2PL:** Transaction **locks data items incrementally**, results in deadlock.
- **Strict 2PL**: A more stricter version of Basic algorithm where **unlocking of exclusive write** is performed after a transaction terminates (commits or aborts and rolledback).
- **Conservative 2PL:** Prevents deadlock by **locking all desired data items before transaction begins execution**.
- **RIGOROUS 2PL** : A more restrictive variation of strict 2PL is rigorous 2PL which also guarantees strict schedules. where **unlocking of exclusive write or read** is performed after a transaction terminates

Strict 2PL

The most popular variation of 2PL is strict 2PL, which guarantees strict schedules
In this variation, a transaction T does not release any of its **exclusive (write) locks** until after it commits or aborts.
Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability.

- **Policy** - Release write locks only after terminating. Transaction is in expanding phase until it ends (may release some read locks before commit).
- **Property**: NOT a deadlock-free protocol but no cascading rollback
- **Practical** : Possible to enforce recoverability.

Rigorous 2PL

A transaction T does not release any of its locks **(exclusive or shared)** until after it commits or aborts.

Does not unlock any of its items until after it terminates (by committing or aborting), so the transaction is in its **expanding phase** until it ends.

Behaves similar to Strict 2PL except it is more restrictive, but easier to implement since all locks are held till commit.

No cascading rollback and deadlock may happen

Conservative 2PL

- Requires a transaction to lock all the items(atomic manner) it accesses before the transaction begins execution, by predeclaring its read-set and write- set.
- If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking.
- **Policy**: Lock all that you need before reading or writing. Transaction is in **shrinking phase after it starts**.
- **Property**: Conservative 2PL is a deadlock-free protocol
- **Practical** : Difficult to use because of difficulty in predeclaring the read-set and write-set.

Reference

 Fundamentals of Database systems 7^{th} Edition by Ramez Elmasri.