COS221 - L32 and L33 - Concurrency Control

Linda Marshall

22 and 24 May 2023

Purpose of Concurrency Control

Focus is on the I (Isolation) and C (Consistency) of the ACID properties of a transaction.

- To ensure that the **isolation** property is maintained while allowing transactions to execute concurrently (outcome of concurrent transactions should appear as though they were executed in isolation).
- To preserve database **consistency** by ensuring that the schedules of executing transactions are serialisable.
- To resolve read-write and write-write conflicts among transactions.

Concurrency Control Protocols

A Concurrency Control Protocol is a set of rules enforced by the DBMS to ensure serialisable schedules.

- Two important Concurrency Control Protocols are:
  - Two-phase locking (2PL)
  - Timestamps
- Other possible protocols include:
  - Multiversion Concurrency
  - Validation (Optimistic) Techniques
  - Multiple Granularity Locking

Two-phase locking protocol

2PL locks data items. Locks describe the status of the item and therefore what operations can be applied to the item.
There are several types of locks that can be used for concurrency control, these include:

- Binary locks
- Shared/Exclusive (Read/Write) Locks

Two-phase locking protocol - Binary Locks

Binary locks have two states, either the item is locked (1) or it is not (0) and each item in the database (X) has a distinct lock `LOCK(X)`.

- If the lock on item X is 1, then it cannot be accessed by the operation that requires that item.
- If the lock on item X is 0, then the lock can be obtained and the value of the lock is changed to 1

Two operations exist for binary locking:

- `lock_item(X)` - request to access item X
- `unlock_item(X)` - release item X

Two-phase locking protocol - Binary Locks

Algorithms for the `lock_item(X)` and `unlock_item(X)` operations:

```
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;
```

Figure 21.1
Lock and unlock operations for binary locks.

Two-phase locking protocol - Binary Locks

- A binary lock enforces **Mutual Exclusion** on the data item.
- The operations must be implemented as indivisible units (referred to as **critical sections**)
- The system maintains records for the items that are currently locked in a **lock table** which is organised as a hash file on the item name. Items not in the lock table are unlocked.
- The **lock manager subsystem** keeps track of and controls access to locks.

Two-phase locking protocol - Binary Locks

For Binary Locks, every transaction must obey the following rules:

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

Two-phase locking protocol - Binary Locks

Binary locks are too restrictive.

- At most one transaction can hold a lock on a given item.
- If all transactions are only reading the item, then one transaction holding a lock on the item will result in the other transactions waiting until the transaction releases the lock.

Need a more granular locking system.

Two-phase locking protocol - Shared/Exclusive Locks

Additional requirements for locking:

- Concurrent access to an item should be allowed for reading purposes. Reading an item by several transactions is not conflicting.
- A transaction writing to an item must have exclusive access to the item

Require a multimode lock, in this case the shared/exclusive (read/write) lock with three locking operations:

- `read_lock(X)`: transaction T requests a read (shared) lock on item X
- `write_lock(X)`: transaction T requests a write (exclusive) lock on item X
- `unlock(X)`: transaction T unlocks an item that it holds a lock on (shared or exclusive)

Two-phase locking protocol - Shared/Exclusive Locks

Algorithms for the `read_lock(X)` and `write_lock(X)` operations:

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

Two-phase locking protocol - Shared/Exclusive Locks

Algorithms for the `unlock(X)` operation:

```
unlock (X):
    if LOCK(X) = "write-locked"
        then begin LOCK(X) ← "unlocked";
                wakeup one of the waiting transactions, if any
                end
    else it 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";
                            wakeup one of the waiting transactions, if any
                            end
                end;
```

Figure 21.2
Locking and unlocking operations for two-mode (read/write, or shared/exclusive) locks.

Two-phase locking protocol - Shared/Exclusive Locks

For Shared/Exclusive Locks, the system must enforce the following rules:

1. A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.
2. A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T.
3. A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.
4. A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X.
5. A transaction T will not issue a write_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X.
6. A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Two-phase locking protocol - Shared/Exclusive Locks

Rules 4 and 5 may be relaxed allowing lock conversions.

- Lock upgrade: existing read lock to write lock
  if T_i holds a read-lock on X, and no other T_j holds a read lock on X ($i \neq j$) then it is possible to convert (upgrade) read_lock(X) to write_lock(X) else force T_i to wait until all other transactions T_j that hold read locks on X release their locks
- Lock downgrade: existing write lock to read lock
  if T_i holds a write lock on X (this implies that no other transaction can have any lock on X) then it is possible to convert (downgrade) write_lock(X) to read_lock(X)

Two-phase locking protocol - Guaranteeing Serialisability

A transaction is said to following the 2PL protocol, if all locking operations preceed the first unlock operation in the transaction. The transaction is divided into the expanding phase, where locks are acquired and the shrinking phase where locks are relinquished.

If lock conversions are allowed then upgrading of locks must be done in the expanding phase and downgrading of locks in the shrinking phase.

Two-phase locking protocol - Guaranteeing Serialisability

T_1	T_2

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

Does not follow the 2PL protocol

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);

Follows the 2PL protocol but may produce a Deadlock

Two-phase locking protocol - Deadlock and Starvation

A deadlock is a situation in which two or more transactions are waiting for one another to give up locks. But because the other transaction is also waiting (in a waiting queue), it will never release the lock.

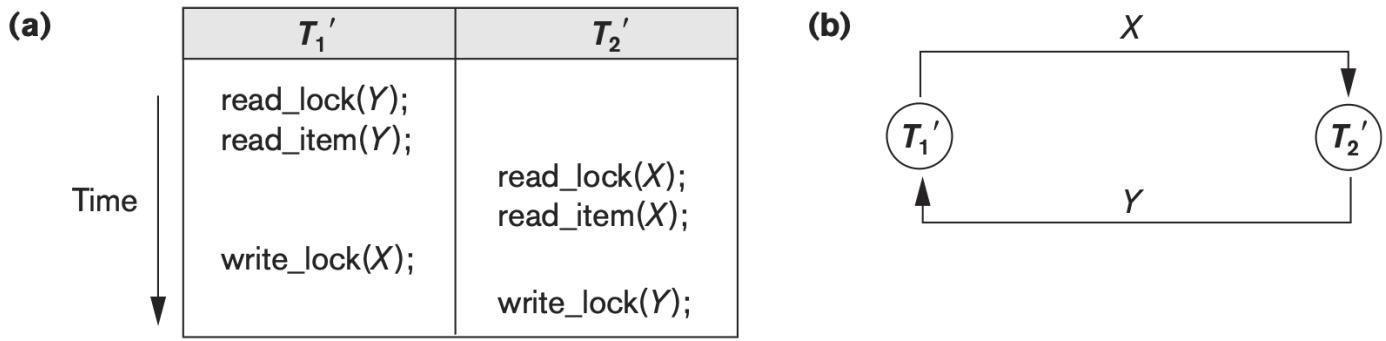

Figure 21.5
Illustrating the deadlock problem. (a) A partial schedule of T_1' and T_2' that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

T_1' is in the waiting queue for X, which is locked by T_2', while T_2' is in the waiting queue for Y, which is locked by T_1'.

Two-phase locking protocol - Variations

There are variations of the 2PL protocol:

- Conservative 2PL: A transaction must lock all its items before starting execution. If any item it needs is not available, it locks no items and tries again later. Conservative 2PL has no deadlocks since no lock requests are issued once transaction execution starts.
- Strict 2PL: All items that are writelocked by a transaction are not released (unlocked) until after the transaction commits. This is the most commonly used two-phase locking algorithm, and ensures strict schedules (for recoverability).
- Rigorous 2PL: All items that are writelocked or readlocked by a transaction are not released (unlocked) until after the transaction commits. Also guarantees strict schedules.

Two-phase locking protocol - Deadlock and Starvation

Deadlock prevention protocols

A. Lock all items needed in advance - aka Conservative 2PL protocol - if any of the items cannot be obtained for locking, then none of the items are locked and the transaction waits and tries again later.

B. Items in the database are ordered and transactions requiring the items lock them in that order.

C. Handle a transaction that is in a possible deadlock situation by either:
   - waiting
   - aborting
   - preempting and aborting another transaction

D. No Waiting (NW) and Cautious Waiting (CW) algorithms

E. Deadlock detection

Options A and B limit concurrency. Option C makes use of **transaction timestamps**, while option D does not.

Two-phase locking protocol - Deadlock and Starvation

C. Transaction Timestamps

A transaction timestamp TS(T'), which is a unique identifier assigned to each transaction and is a monotonically increasing unique id given to each transaction based on their starting time.

If $TS(T_1) < TS(T_2)$, it means that T_1 started before T_2 (T_1 older than T_2)

Two schemes that prevent deadlock are:

- Wait-die: An older transaction is allowed to wait for a younger transaction, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted.
- Wound-wait: A younger transaction is allowed to wait for an older one, whereas an older transaction requesting an item held by a younger transaction preempts the younger transaction by aborting it.

Two-phase locking protocol - Deadlock and Starvation

D. Waiting algorithms

- No-waiting protocol: A transaction never waits; if T_i requests an item that is held by T_j in conflicting mode, T_i is aborted. Can result in needless transaction aborts because deadlock might have never occurred
- Cautious waiting protocol: If T_i requests an item that is held by T_j in conflicting mode, the system checks the status of T_j; if T_j is not blocked, then T_i waits - if T_j is blocked, then T_i aborts. Reduces the number of needlessly aborted transactions

Two-phase locking protocol - Deadlock and Starvation

E. Deadlock Detection

- Deadlocks are allowed to happen. The system maintains a wait-for graph for detecting cycles. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back (aborted).
- **Timeouts** can be used for deadlocks. If a transaction waits for a period longer than a predefined time, the system assumes the transaction is in deadlock and aborts irrespective of whether the transaction is in deadlock or not.
- **Starvation** occurs when a particular transaction consistently waits or gets restarted and never gets a chance to proceed further. A solution is to use a fair priority-based scheme that increases the priority for transaction the longer they wait e.g first-come-first-served queue.

Two-phase locking protocol - Deadlock and Starvation

Examples of Starvation

- In deadlock detection/resolution it is possible that the same transaction may consistently be selected as victim and rolled-back.
- In conservative 2PL, a transaction may never get started because all the items needed are never available at the same time.
- In Wound-Wait scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.

Timestamps

Timestamps do not make use of locks and therefore deadlocks cannot occur.

Timestamps can be generated in several ways:

- Increment a counter each time its value is assigned to $TS(T_i)$. Counters have a finite value and therefore are periodically reset when there are no transactions running.
- Make use of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

Timestamps - Timestamp Ordering Algorithm

Instead of locks, the system keeps track of two values for each data item X:

- `Read_TS(X)`: The largest timestamp among all the timestamps of transactions that have successfully read item X
- `Write_TS(X)`: The largest timestamp among all the timestamps of transactions that have successfully written X

When a transaction T requests to read or write an item X, TS(T) is compared with `read_TS(X)` and `write_TS(X)` to determine if request is out-of-order.

Timestamps - Timestamp Ordering Algorithm

Basic Timestamp Ordering

1. Transaction T requests a `write_item(X)` operation:
   - If `read_TS(X)` $>$ TS(T) or if `write_TS(X)` $>$ TS(T), then a younger transaction has already read or written the data item so abort and roll-back T and reject the operation.
   - Otherwise, execute `write_item(X)` of T and set `write_TS(X)` to TS(T).
2. Transaction T requests a `read_item(X)` operation:
   - If `write_TS(X)` $>$ TS(T), then a younger transaction has already written the data item X so abort and roll-back T and reject the operation.
   - Otherwise, execute `read_item(X)` of T and set `read_TS(X)` to the larger of TS(T) and the current `read_TS(X)`.

Timestamps - Timestamp Ordering Algorithm

Strict Timestamp Ordering
Schedules are both strict and serialisable.

Transaction T issues a `read_item(X)` or `write_item(X)` such that TS(T) $>$ `write_TS(X)` has its read or write operation delayed until the transaction T' that wrote the value of X (hence TS(T') $=$ `write_TS(X)`) has committed or aborted.

Timestamps - Timestamp Ordering Algorithm

Thomas's Write Rule

This algorithm does not enforce conflict serialisability, but rejects fewer write operations as follows:

1. If `read_TS(X)` $>$ TS(T), then abort and roll back T and reject the operation.
2. If `write_TS(X)` $>$ TS(T), then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than TS(T) - and hence after T in the timestamp ordering - has already written the value of X. Thus, we must ignore the `write_item(X)` operation of T because it is already outdated and obsolete.
3. If neither the condition in 1. nor the condition in 2. occurs, then execute the `write_item(X)` operation of T and set `write_TS(X)` to TS(T).

Multiversion Concurrency

Multiversion concurrency control keeps multiple versions of the data item that have been written. When a read item is requested by a transaction the appropriate version is chosen to maintain serialisability.

Multiversion concurrency techniques in many cases make use of the concept of view serialisability rather than conflict serialisability.

A drawback of this technique is the added storage required to keep the multiple versions. However, some databases keep previous values of items making this technique a viable option.

Multiversion Concurrency

Multiversion concurrency can be implemented using either:.

- Timestamps - The following two timestamps are kept for each version of X_i,
  - `read_TS(`X_i`)`. The read timestamp of X_i is the largest of all the timestamps of transactions that have successfully read version X_ii.
  - `write_TS(`X_i`)`. The write timestamp of X_i is the timestamp of the transaction that wrote the value of version X_i.
- 2PL protocol - Where three locking modes are used
  - read and write as before
  - certify

(a)

	Read	Write
Read	Yes	No
Write	No	No

(b)

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

Figure 21.6
Lock compatibility tables.
(a) Lock compatibility table for read/write locking scheme.
(b) Lock compatibility table for read/write/certify locking scheme.

Validation (Optimistic) Techniques

The technique is optimistic because it assumes few conflicts will occur. The system therefore does not perform checks before reading and writing, but rather at the end of the transaction during the validation phase. The three phases occur in the following order:

1. Read phase - A transaction can read values of committed data items. However, writes are applied only to local copies (versions) of the data items.
2. Validation phase. Serialisability is checked by determining any conflicts with other concurrent transactions.
3. Write phase. On a successful validation, transaction updates are applied to the database on disk and become the committed versions of the data items; otherwise, transactions that fail the validation phase are restarted.

Validation (Optimistic) Techniques

The validation phase for T_i checks that, for each transaction T_j that is either committed or is in its validation phase, one of the following conditions holds:

1. T_j completes its write phase before T_i starts its read phase.
2. T_i starts its write phase after T_j completes its write phase, and the `read_set`(T_i) has no items in common with the `write_set`(T_j)
3. Both `read_set`(T_i) and `write_set`(T_i) have no items in common with the `write_set`(T_j), and T_j completes its read phase before T_i completes its write phase.

Validation (Optimistic) Techniques

When validating T_i, the first condition is checked first for each transaction T_j, since 1. is the simplest condition to check.

If 1. is false for a particular T_j, then
2. is checked and
only if 2. is false is 3. checked.

If none of these conditions holds for any T_j, the validation fails and T_i is aborted.

Multiple Granularity Locking

All database concurrency techniques assume that a database is a collection of named data items. A data item could be:

- A database record
- A field value of a database record
- A disk block
- A whole file
- The whole database

The choice of the granularity of a data item can influence the concurrency performance.

Multiple Granularity Locking

By breaking up a database into a hierarchical structure as follows:

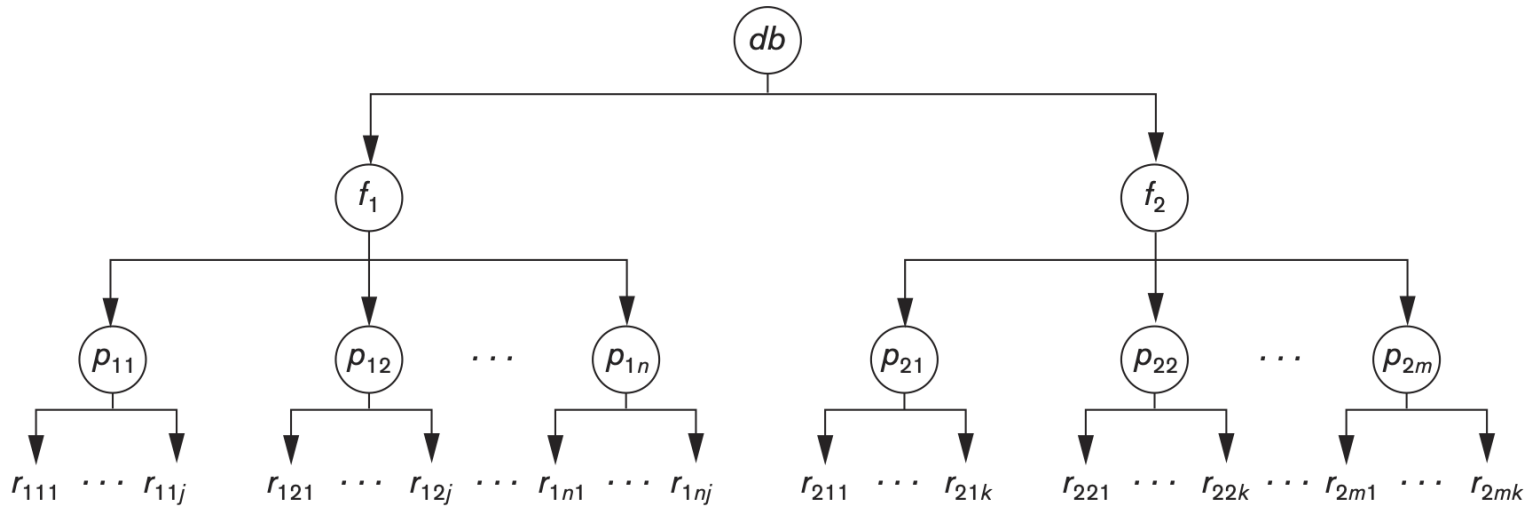

Figure 21.7
A granularity hierarchy for illustrating multiple granularity level locking.

a database can support multiple levels of granularity.

Multiple Granularity Locking

To manage such hierarchy, in addition to read or shared (S) and write or exclusive (X) locking modes, three additional locking modes, called intention lock modes are defined:

- Intention-shared (IS): indicates that a shared lock(s) will be requested on some descendent nodes(s).
- Intention-exclusive (IX): indicates that an exclusive lock(s) will be requested on some descendent nodes(s).
- Shared-intention-exclusive (SIX): indicates that the current node is requested to be locked in shared mode but an exclusive lock(s) will be requested on some descendent node(s).

Multiple Granularity Locking

	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

Figure 21.8
Lock compatibility matrix for multiple granularity locking.

Multiple Granularity Locking

The set of rules which must be followed for producing serialisable schedule are:

- The lock compatibility table must be adhered to.
- The root of the tree must be locked first, in any mode.
- A node N can be locked by a transaction T in S (or X) mode only if the parent node is already locked by T in either IS (or IX) mode.
- A node N can be locked by T in X, IX, or SIX mode only if the parent of N is already locked by T in either IX or SIX mode.
- T can lock a node only if it has not unlocked any node (to enforce 2PL policy).
- T can unlock a node, N, only if none of the children of N are currently locked by T.

Multiple Granularity Locking

T_1	T_2	T_3
IX(db)		
IX(f_1)		
	IX(db)	
		IS(db)
		IS(f_1)
		IS(p_{11})
IX(p_{11})		
X(r_{111})		
	IX(f_1)	
	X(p_{12})	
		S(r_{11j})
IX(f_2)		
IX(p_{21})		
X(p_{211})		
unlock(r_{211})		
unlock(p_{21})		
unlock(f_2)		
		S(f_2)
	unlock(p_{12})	
	unlock(f_1)	
	unlock(db)	
unlock(r_{111})		
unlock(p_{11})		
unlock(f_1)		
unlock(db)		
		unlock(r_{11j})
		unlock(p_{11})
		unlock(f_1)
		unlock(f_2)
		unlock(db)

Figure 21.9
Lock operations to illustrate a serializable schedule.

Other Concurrency techniques

Concurrency control can be applied to indexes. Two phase locking can be used in conjunction with B-Trees and B^+-Trees.

Insertion and Deletion of records may impact concurrency control resulting in a phantom record. A phantom record is a record that when being inserted satisfies the condition of another set of record accessed by another transaction. Phantom records can be dealt with by:

- index locking
- predicate locking

Interaction, e.g. by entering values on the screen and then submitting them may result in concurrency issues.