Chapter 5: Concurrency Control Techniques

Adama Science and Technology University
School of Electrical Engineering and Computing
Department of CSE
CSEg 2208: Database Systems
(2022)

Outline

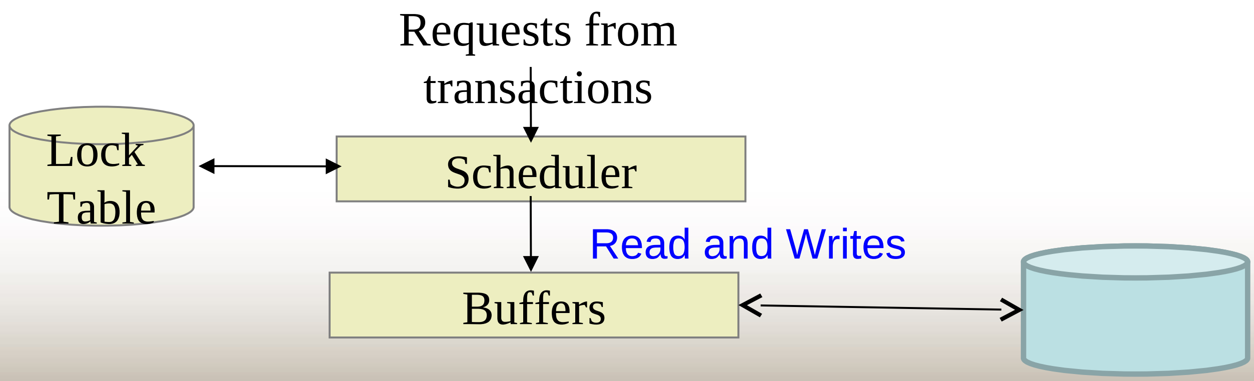
- Databases Concurrency Control
- Purpose of Concurrency Control
- Concurrency Control Techniques:
  - Locking
  - Timestamp
  - Optimistic
  - Multiversion
  - Lock Granularity

Database Concurrency Control

Transaction Processor is divided into:

- A concurrency-control manager/scheduler, responsible for assuring *isolation of transactions*.
- A logging and recovery manager, responsible for the *durability of transactions*.

- The scheduler (***concurrency-control manager***) must assure that the individual actions of *multiple transactions* are executed in such an order that the net effect is the same as if the transactions had in fact *executed one-at-a-time*.



Purpose of Concurrency Control

Example:

Bank database: 3 Accounts

Account Balances → A = 500, B = 500, C = 500

A = 500

B = 500

C = 500

Property: A + B + C = 1500

Money does not *leave the system*

Purpose of Concurrency Control

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

• A *typical scheduler* does its (concurrency control) work by maintaining *locks on certain pieces* of the database.

• These locks *prevent two transactions* from accessing the same piece of data at the same time. Example:

- In concurrent execution environment if T_1 conflicts with T_2 over a data item A, then the *existing concurrency control* decides if T_1 or T_2 should get the A and if the other transaction is rolled-back or waits.

Purpose of Concurrency Control

Example

[*Transaction T1*: Transfer 100 from *A to B*]

Read (A, t)

t = t - 100

Write (A, t)

Read (B, t)

t = t + 100

Write (B, t)

[*Transaction T2*: Transfer 100 from *A to C*]

Read (A, s)

s = s - 100

Write (A, s)

Read (C, s)

s = s + 100

Write (C, s)

Purpose of Concurrency Control

Transaction T_1	Transaction T_2	A	B	C
Read (A, t)		500	500	500
t = t - 100				
	Read (A, s)			
	s = s - 100			
	Write (A, s)	400	500	500
Write (A, t)		400	500	500
Read (B, t)				
t = t + 100				
Write (B, t)		400	600	500
	Read (C, s)			
	s = s + 100			
	Write (C, s)	400	600	600

400 + 600 + 600 = 1600

Schedule

Purpose of Concurrency Control

Transaction T1	Transaction T2	A	B	C
Read (A, t)		500	500	500
t = t - 100				
Write (A, t)		400	500	500
	Read (A, s)			
	s = s - 100			
	Write (A, s)	300	500	500
Read (B, t)				
t = t + 100				
Write (B, t)		300	600	500
	Read (C, s)			
	s = s + 100			
	Write (C, s)	300	600	600

300 + 600 + 600
= 1500

So What ?

Alternative Schedule

Concurrency Control Techniques

- **Basic concurrency control techniques:**
  - **Locking,**
  - **Timestamping**
  - **Optimistic methods**
- The First two are conservative approaches: delay transactions in case they *conflict with other transactions*.
- Optimistic methods assume conflict is *rare and only check for conflicts at commit*.
- **Locking:**
  - Lock is a *variable* associated with a *data item* that *describes the status of the data item* with respect to the possible operations that can be applied to it.

Concurrency Control Techniques

- Generally, a transaction must claim a *shared (read)* or *exclusive (write)* lock on a data item before read or write.
- *Lock prevents* another transaction from *modifying item or even reading it*, in the case of a write lock.
- **Locking is an operation which secures :**
  - (a) permission to *Read*,
  - (b) permission to *Write a data item* for a transaction.
  - Example:
    - Lock (X). Data item X is locked in behalf of the *requesting transaction*.
- **Unlocking** is an operation *which removes* these permissions from the data item.
  - Example:
    - Unlock (X): Data item X is made available to *all other transactions*.
- Lock and Unlock are ***Atomic operations***.

Concurrency Control Techniques

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

	Read	Write
Read	Y	N
Write	N	N

Concurrency Control Techniques

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

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

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

Concurrency Control Techniques

Locking - Basic Rules:

- It has two oprerations : Lock_item(X) and unLock_item(X).
- A transaction request access to an item X by first issuing a lock_Item(x) opreation .
- If lock (x)=1, the transaction is forced to wait.
- If lock (X)= 0; it is set to 1 and the transaction is allowed to access x.
- When a transaction finished operation on X it issues an Unlock _item operation which set lock(x) to 0 so that X may be accessed by another transaction
- If transaction has shared lock on item, can read but not update item.
- If transaction has exclusive lock on item, can both read and update item.
- Reads cannot conflict, so *more than one transaction* can hold shared locks simultaneously on same item.
- Exclusive lock gives transaction *exclusive access* to that item.

Concurrency Control Techniques

- The following code performs the read operation: ***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;
```

Concurrency Control Techniques

- **The following code performs the write lock operation: *write_lock(X)*:**

```
B: if LOCK (X) = "unlocked" then
                LOCK (X) ← "write-locked";
   else
        wait (until LOCK(X) = "unlocked"
              and the lock manager wakes up the transaction);
   goto B
        end;
```

Concurrency Control Techniques

- **Lock conversion**
  - *Lock upgrade*: existing read lock to *write lock*:

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

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

    else

    force Ti to wait until Tj unlocks X
  - *Lock downgrade*: existing write lock to *read lock*:

    Ti has a write-lock (X) (*no transaction can have any lock on X*) convert write-lock (X) to read-lock (X)
- Using such locks in the transaction do not guarantee serializability of schedule on its own: Example

Concurrency Control Techniques

Schedule: **T1** **T2**

Write Lock(X)
Read (X)
X=X+100
Write(X)
Unlock(X)
Write Lock(X)
Read (X)
X= X*1.1
Write(X)
Unlock(X)
Write Lock(Y)
Read (Y)
Y= Y*1.1
Write(Y)
Unlock(Y)
Commit
write_lock(Y)
read(Y)
Y= Y-100
write(Y)
unlock(Y)
commit

Example : Incorrect Locking Schedule

Concurrency Control Techniques

- If at start, X = 100, Y = 400, result should be:
  - X = 220, Y = 330, if T_1 executes before T_2, or
  - X = 210, Y = 340, if T_2 executes before T_1
- However, result gives X= 220 and Y = 340.
- S is not a serializable schedule. Why?
- Problem is that transactions release locks too early (soon), resulting in loss of *total isolation* and *atomicity*.
- To guarantee serializability, we need an *additional protocol* concerning the *positioning of lock* and *unlock operations* in every transaction.

Concurrency Control Techniques

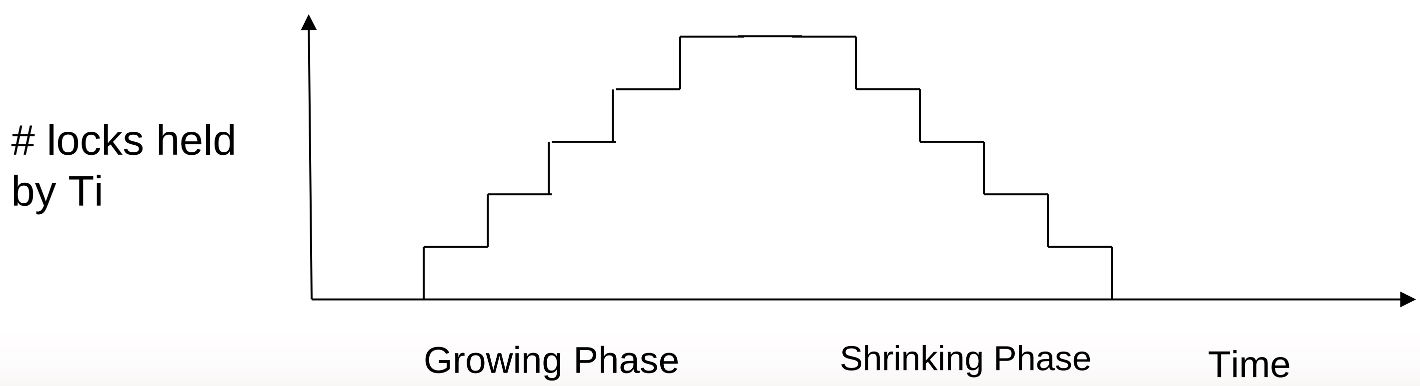
2.2 Two-Phase Locking Techniques: The algorithm

- Transaction follows 2PL protocol if all *locking* operations *precede first unlock operation* in the transaction.
- Every transaction can be divided into *Two Phases*: **Locking (Growing) & Unlocking (Shrinking)**
  - ***Locking (Growing) Phase*:**
    - A transaction applies locks (read or write) on *desired data items* one at a time.
    - Acquires *all locks but cannot release any locks*.
  - ***Unlocking (Shrinking) Phase*:**
    - A transaction unlocks its *locked data items one at a time*.
    - Releases locks but *cannot acquire any new locks*.

Concurrency Control Techniques

Requirement:

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



Concurrency Control Techniques

Example

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);

Result

Initial values: X=20; Y=30

Result of serial execution

T1 followed by T2

X=50, Y=80.

Result of serial execution

T2 followed by T1

X=70, Y=50

Concurrency Control Techniques

T’1	T’2
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);

.

- T’1 and T’2 follow two-phase policy but they are subject to *deadlock*, which must be dealt with

- If every transaction in the schedule follows the 2PL protocol , the schedule is guaranteed to be *serializabe*.
- **Limitation -It may limit the number of concurrency that can occur in the schedule . How?**
- Remark: The use of this locking can cause two additional problems
  - **Deadlock and starvation**

Concurrency Control Techniques

Dealing with Deadlock and Starvation

- **Deadlock**
  - It is a state that may result when *two or more transaction* are each waiting for locks held by the other to be released.
  - Example :

T1	**T2**
read_lock (Y); read_item (Y);	
	read_lock (X); read_item (X);
write_lock (X);	
	write_lock (Y);

Concurrency Control Techniques

- T1 is in the *waiting queue* for X which is locked by T2.
- T2 is on the *waiting queue* for Y which is locked by T1.
- No transaction can *continue until the other transaction completes*.
- T1 and T2 did follow two-phase policy but they are deadlock.

- So the DBMS must either *prevent* or *detect* and *resolve* such deadlock situations

There are possible solutions : Deadlock prevention, deadlock detection and avoidance ,and lock timeouts

i. Deadlock prevention protocol: two possibilities

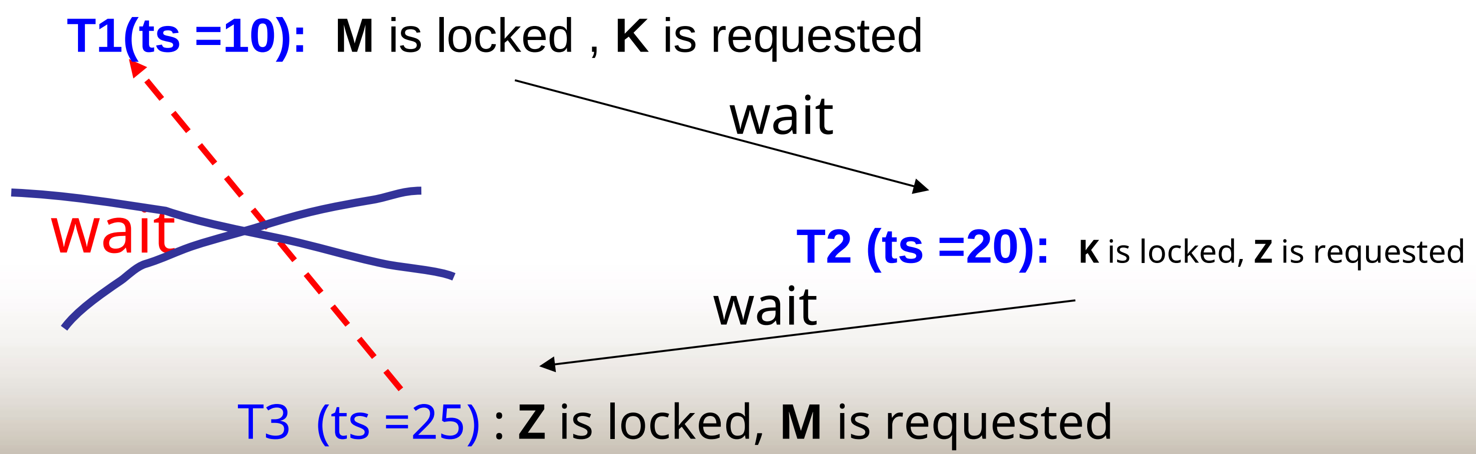
- **The conservative two-phase locking**
  - A transaction locks *all data items it refers* to before it begins execution.
  - This way of locking *prevents deadlock* since a transaction never waits for a data item.
  - Limitation : It restricts concurrency

Concurrency Control Techniques

- Transaction Timestamp(TS(T))
  - We can prevent deadlocks by *giving each transaction a priority* and ensuring that *lower priority transactions* are not allowed to wait for higher priority transactions (or vice versa).
  - One way to *assign priorities* is to give each transaction a *timestamp* when it starts up.
  - it is a *unique identifier given to each transaction* based on time in which it is started. i.e if T1 starts before T2 , TS(T1)<TS(T2)
  - The *lower the timestamp*, the *higher* the *transaction's priority*, that is, the oldest transaction has the highest priority.
  - **If a transaction Ti requests a lock and transaction Tj holds a conflicting lock, the lock manager can use one of the following two policies:** Wait-die & Would-wait

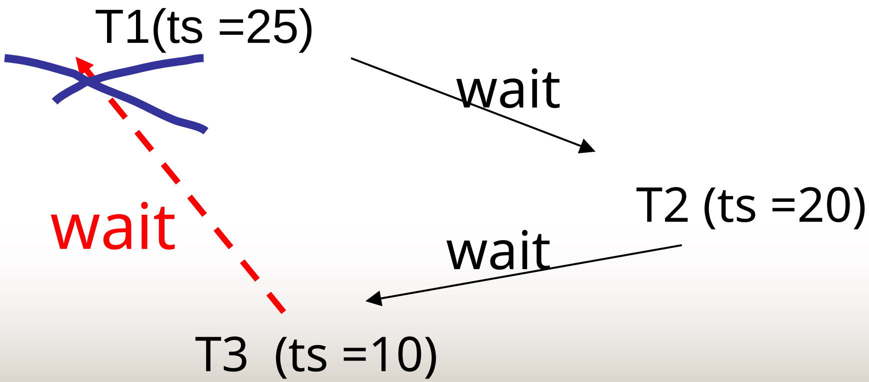
Concurrency Control Techniques

- **Wait-die**
  - If Ti has *higher priority*, it is allowed to wait; otherwise it is aborted.
  - An older transaction is allowed to wait on a younger transaction.
  - A *younger transaction requesting* an item held by an older transaction is *aborted*.
  - If TS(Ti) < TS(Tj), then (Ti older than Tj) Ti is allowed to *wait*.
  - Otherwise (Ti younger than Tj) *Abort* Ti (Ti dies) and restart it later with the same timestamp.



Concurrency Control Techniques

- **Wound-wait**
  - The opposite of wait-die.
  - If Ti has *higher priority*, abort Tj; otherwise Ti waits.
  - A younger transaction is allowed to wait on al older one
  - An older transaction requesting an item held by a younger transaction preempts the younger transaction by aborting it.
  - If TS(Ti) < TS(Tj), then (Ti older than Tj), *Abort* Tj (Ti wounds Tj) and restart Tj later with the same timestamp.
    - Otherwise (Ti younger than Tj) Ti is allowed to *wait*.



Concurrency Control Techniques

- **Remark:**
  - Both methods ended by aborting *the younger of the two transaction* that may be involved in the deadlock.
  - Limitation:
    - Both techniques may cause *some transaction* to be *aborted* and *restarted* needlessly.

ii. Deadlock Detection and resolution

- In this approach, deadlocks are *allowed to happen*.
- The scheduler maintains a *wait-for-graph for detecting cycle*.
- When a chain like: Ti waits for Tj waits for Tk waits for Ti or Tj occurs, then this creates a cycle.
- When the system is in the *state of deadlock* , some of the transaction should be aborted by selection(victim) and rolled-back.
- This can be done by aborting those transaction: that have made the least work, the one with the lowest locks, and that have the least # of abortion and so on
- Example:

Concurrency Control Techniques

iii. Timeouts

- It uses the *period of time* that several transaction have been waiting to lock items.
- It has *lower overhead cost* and it is *simple*.
- If the *transaction wait for a longer time* than the predefined time out period, the system assume that may be *deadlocked* and *aborted* it.

- **Starvation**
  - Starvation occurs when a particular transaction *consistently waits* or *restarted* and never gets a chance to proceed further while other transaction continue normally.
  - This may occur , if the *waiting method* for item locking:
    - Gave priority for some transaction over others.
    - Problem in Victim selection algorithm- it is possible that the same transaction may consistently be selected as victim and rolled-back .example In **Wound-Wait**
  - **Solution**
    - FIFO,
    - Allow for transaction that *wait for a longer time*,
    - Give *higher priority* for transaction that have been *aborted for many times*.

Concurrency Control Techniques

Timestamp based concurrency control algorithm

- **Timestamp**
  - In lock based concurrency control , conflicting actions of different transactions are ordered by the order in which locks are obtained.
  - But here, Timestamp values are assigned based on time in which the transaction are submitted to the system using the current date & time of the system
  - A monotonically increasing variable (integer) indicating the age of an operation or a transaction.
  - A larger timestamp value indicates a more recent event or operation.
  - Timestamp based algorithm uses timestamp to serialize the execution of concurrent transactions.
  - It doesn't use lock, thus deadlock cannot be occurred
  - In the timestamp ordering, conflicting operation in the schedule shouldn't violate serilazable ordering
  - This can be achieved by associating timestamp value (TS) to each database item which is denoted as follow:

Concurrency Control Techniques

a) Read_Ts(x): the read timestamp of x – this is the largest time among all the time stamps of transactions that have successfully read item X.

b) Write_TS(X): the largest of all the timestamps of transactions that have successfully written item X.

- The concurrency control algorithm check whether conflict operation violate the timestamp ordering in the following manner: three options

i. Basic Timestamp Ordering

- Transaction T issues 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/write the values of the data item x before T had a chance to write X . so abort and roll-back T and restarted with a new, larger timestamp. Why is with new timestamp?, is there a difference b/n this timestamp protocol and the 2PL for dead lock prevention?
  - If the condition above does not exist, then execute write_item(X) of T and set write_TS(X) to TS(T).

Concurrency Control Techniques

- **Transaction T issues a read_item(X) operation:**
  - If write_TS(X) > TS(T), then an younger transaction has already written to the data item,so abort and roll-back T and reject the operation.
  - If write_TS(X) ≤ TS(T), then execute read_item(X) of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X)
- **Limitation:** cyclic restart/starvation may occur when a transaction is continuously aborted and restarted

Concurrency Control Techniques

4. Multiversion Concurrency Control Techniques

- This approach maintains a *number of versions of a data item* and *allocates the right version* to a read operation of a transaction.
- Thus unlike other mechanisms a *read operation* in this mechanism is *never rejected*.
- This algorithm uses the concept of *view* serilazabilty than *conflict* serialiazabilty.
- **Side effect:**
  - Significantly more storage (RAM and disk) is required to *maintain multiple versions*. To check unlimited growth of versions, a garbage collection is run when some criteria is satisfied.
- Two schemes : **based on time stamped ordering & 2PL**

Concurrency Control Techniques

i. Multiversion technique based on timestamp ordering

– Assume X1, X2, …, Xn are the version of a data item X created by a *write operation of transactions*.

– With each Xi a read_TS (read timestamp) and a write_TS (write timestamp) are associated.)

  – **read_TS(Xi)**: The read timestamp of Xi is the largest of all the timestamps of transactions that have successfully read version Xi.

  – **write_TS(Xi)**: The write timestamp of Xi that wrote the value of version Xi.

  – A new version of Xi is created only by a *write operation*.

Concurrency Control Techniques

- **To ensure serializability, the following two rules are used:**

  i. **If transaction T issues write_item (X)** and version i of X has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), and read _TS(Xi) > TS(T), then abort and roll-back T; otherwise create a new version Xi and *set read_TS(X) = write_TS(Xi) = TS(T).*

  ii. **If transaction T issues read_item (X),** find the version i of X that has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), then return the value of Xi to T, and *set the value of read _TS(Xi) to the largest of TS(T) and the current read_TS(Xi).*

– Note that: ***Rule two indicates that read request will never be rejected***

Concurrency Control Techniques

- ii. **Multiversion Two-Phase Locking Using Certify Lock**
  - Allow a transaction T’ to read a data item X while it is write locked by a conflicting transaction T.
  - This is accomplished by maintaining two versions of each data item X where one version must always have been written by some committed transaction. This means a write operation always creates a new version of X.

Steps

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

Concurrency Control Techniques

Compatibility tables for

read/write locking scheme

	Read	Write
Read	yes	no
Write	no	no

read/write/certify locking scheme

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

Note:

- In multiversion 2PL read and write operations from conflicting transactions can be processed concurrently.
- This improves concurrency but it may delay transaction commit because of obtaining certify locks on all its writes.
- It avoids cascading abort but like strict two phase locking scheme conflicting transactions may get deadlocked.

Concurrency Control Techniques

Validation (Optimistic) Concurrency Control Schemes

- This technique allow transaction to *proceed asynchronously* and only at the time of commit, serializability is checked & transactions are aborted in case of non-serializable schedules.
- Good if there is *little interference* among transaction.
- It has three phases: ***Read, Validation , and Write phase.***

i.. **Read phase**:

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

Concurrency Control Techniques

ii. **Validation phase**:.

- If the transaction Ti decides that it wants to commit, the DBMS checks whether the transaction could possibly have conflicted with any other concurrently executing transaction.
- While one transaction ,Ti, is being validated , no other transaction can be allowed to commit.
- This phase for Ti checks that, for each transaction Tj that is either committed or is in its validation phase, one of the following conditions holds:

Concurrency Control Techniques

- Tj completes its write phase before Ti starts its read phase.
- Ti starts its write phase after Tj completes its write phase and the read set of Ti has no item in common with the write set of Tj
- Both the read_set and write_set of Ti have no items in common with the write_set of Tj.

– When validating Ti, the first condition is checked first for each transaction Tj, since (1) is the simplest condition to check. If (1) is false then (2) is checked and if (2) is false then (3) is checked.

– If none of these conditions holds, the validation fails and Ti is aborted.

iii. **Write phase**:

- On a successful validation, transactions' updates are applied to the database; otherwise, transactions are restarted.

Concurrency Control Techniques

6. Multiple Granularity Locking

- A *lockable unit* of data defines its *granularity*.
- Granularity can be *coarse* (*entire database*) or it can be *fine* (*an attribute of a relation*).
- Example of *data item granularity*:
  - A field of a database record
  - A database record
  - A disk block/ page
  - An entire file
  - The entire database
- Data item granularity significantly affects *concurrency control performance*.
- Thus, the degree of concurrency is *low for coarse granularity* and *high for fine granularity*.

Concurrency Control Techniques

- Example:
  - A transaction that expects to access most of the pages in a file should probably set a lock on the entire file , rather than locking individual pages or records.
  - If a transaction that requires to access relatively few pages of the file , it is better to lock those pages.
  - Similarly , if a transaction access several records on a page , it should lock the entire page and if it access just a few records , it should lock some those records.
  - This example will hold true , if a lock on the node locks that node and implicitly all its descendants.

Concurrency Control Techniques

- The following diagram illustrates a hierarchy of granularity from *coarse* (*database*) to *fine* (*record*).

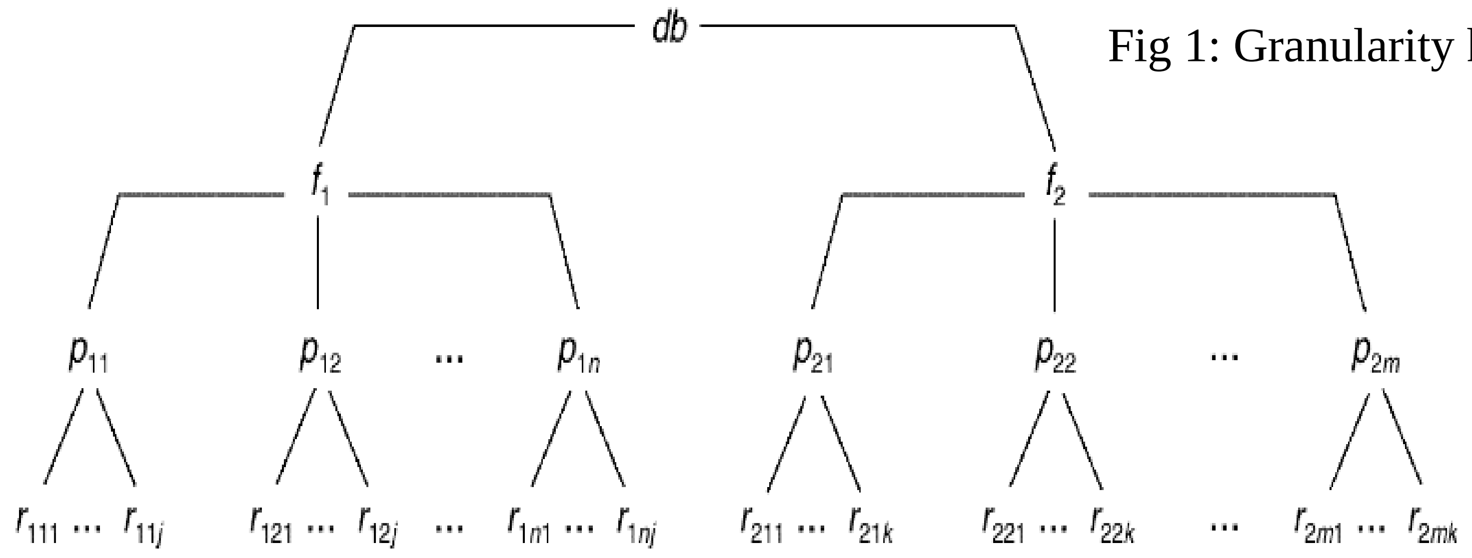


Fig 1: Granularity hierarchy

Concurrency Control Techniques

- To manage such hierarchy, in addition to read(S) and write(X), 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 node(s).
  - **Intention-exclusive (IX):** indicates that an exclusive lock(s) will be requested on some descendent node(s).
  - **Shared-intention-exclusive (SIX):** indicates that the current node is locked in shared mode but an exclusive lock(s) will be requested on some descendent nodes(s).
    - If a transaction needs to read an entire file and modify a few of the records in it. i.e it needs S lock on a file & IX lock on some of the contained object

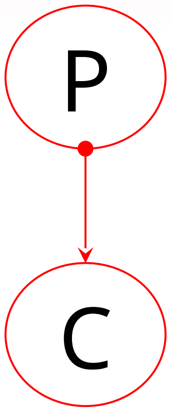
Concurrency Control Techniques

- The set of rules which must be followed for producing serializable schedule are:
  1. The lock compatibility must adhered to.
  2. The root of the tree must be locked first, in any mode.
  3. A node N can be locked by a transaction T in S or IX mode only if the parent node is already locked by T in either IS or IX mode.
  4. A node N can be locked by T in X, IX, or SIX mode only if the parent of N is already locked by T in either IX or SIX mode.
  5. T can lock a node only if it has not unlocked any node (to enforce 2PL policy).
  6. T can unlock a node, N, only if none of the children of N are currently locked by T.

Concurrency Control Techniques

Summery :

Parent locked in	Child can be locked in
IS	IS, S
IX	IS, S, IX, X, SIX
SIX	X, IX
X	None



Concurrency Control Techniques

- To lock a node in *S* mode, a transaction must first lock all its ancestors in *IS* mode. Thus, if a transaction locks a node in *S* mode, no other transaction can have locked any ancestor in *X* mode;
- Similarly, if a transaction locks a node in *X* mode, no other transaction can have locked any ancestor in *S* or *X* mode.
- These two cases ensures that no other transaction holds a lock on an ancestor that conflicts with the requested *S* or *X* lock on the node.

Concurrency Control Techniques

- These locks are applied using the following compatibility matrix:

Requestor : T1

Holder: T2

	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

- **Consider the following three example based Fig 1 diagram given in slide 40**
  - T1 wants to update record r111 and r211
  - T2 wants to update all records on page p12
  - T3 wants to read records r11j and the entire f2 file

Concurrency Control Techniques

Multiple Granularity Locking: Lock operation to show a serializable execution

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

Multiple Granularity Locking: Lock operation to show a serializable execution(continued)

T1	T2	T3
		S(f2)
	unlock(p12)	
	unlock(f1)	
	unlock(db)	
unlock(r111)		
unlock(p11)		
unlock(f1)		
unlock(db)		
		unlock (r111j)
		unlock (p11)
		unlock (f1)
		unlock(f2)

Question & Answer

Thanks !!!