Database Management Systems (DBMS)

Lec 26: Transaction Processing, Concurrency Control, and Recovery (Contd.)

Ramesh K. Jallu
IIIT Raichur

Date: 15/06/21

Recap

- Determining a conflict serializability through precedence graph
- Need for concurrency control
    - The lost update problem
    - The temporary or dirty read problem
    - The incorrect summary problem
    - The unpredictable read problem
- Characterizing schedules based on recoverability

Recoverable schedules (Recap)

- A transaction T_j that is dependent on T_i (i.e., T_j has read data written by T_i) commits only after T_i is committed

- The atomicity property requires that any transaction T_j that is dependent on T_i is also must be aborted if T_i fails for whatever reason

T_6	T_7
read(A)	
write(A)	
	read(A)
	commit
read(B)	

T_8	T_9	T_{10}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
abort		

Cascadeless schedules

- A phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called **cascading rollback**
- It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called ***cascadeless*** schedules
- Every cascadeless schedule is a recoverable schedule as every transaction reads items that were written by committed transactions

T_8	T_9	T_{10}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
abort		

Concurrency control

- When several transactions execute concurrently in the database the isolation property may no longer be preserved

- The system must control the interaction among the concurrent transactions

- The mechanisims that controls the interaction among transactions is called the ***concurrency control schemes*** (ptotocols)

- The most commonly used protocols in practise
  1. *Lock based protocols*
  2. *Timestamp based protocols*

Lock based protocols

- To ensure isoloation is preserved, the data items be accessed in a *mutually exclusive* manner
  - That is, while one transaction is accessing a data item, no other transaction can modify that data item

- A ***lock*** is a *variable* associated with a data item that describes the status of the item

- The DBMS has a ***lock manager subsystem*** to keep track of and control access to locks. The system grants a transaction to access a data item only if it is currently holding a ***lock*** on that item

- Generally, there is one lock for each data item in the database

Types of locks

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

Binary locks

- A **binary lock** can have two *states* or *values*: locked (1) and unlocked (0)
- A distinct lock is associated with each database item *X*
- 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 *X "can be accessed"* when requested, and the lock value is changed to 1
- We refer to the current value (or state) of the lock associated with item *X* as ***lock(X)***

Operations on binary locks

- Two operations are used in binary locks : *lock_item* and *unlock_item*

- A transaction requests access to an item *X* by first issuing a ***lock_item(X)*** operation
  - If *lock(X)* = 1, the transaction is forced to wait
  - If *lock(X)* = 0, it is set to 1 (the transaction **locks** the item) and the transaction is allowed to access item *X*

- After execution/abort, the transaction issues an ***unlock_item(X)*** operation, which sets *lock(X)* back to 0 so that *X* may be accessed by other transactions

- Hence, a binary lock enforces **mutual exclusion** on the data item

Lock and unlock 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;
```

Rules associated with binary locks

- Every transaction must obey the following four 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* should not issue a *lock_item(X)* operation if it already holds the lock on item *X*
  4. A transaction *T* should not issue an *unlock_item(X)* operation unless it already holds the lock on item *X*

Advantages and disadvantages with binary locks

- It is simple to implement a binary lock; all that is needed is a binary-valued variable, *lock* , associated with each data item *X* in the database

- The system needs maintain a lock table such as a hash table to keep track the *only the items that are currently locked* . Items not in the lock table are considered to be unlocked

- The transactions that are waiting to access a locked item can be put in a queue

- **At most one transaction** can hold the lock on a particular item. Thus no two transactions can access the same item concurrently

Shared/Exclusive (or Read/Write) Locks

- If a transaction access X just for *reading purposes only*, then it doesn't need any lock. However, if the transaction *is to write* an item X, it must have exclusive access to X

- For this purpose a ***multiple-mode locks*** are used and are called ***shared/exclusive*** or **read/write**

- **Shared**: If a transaction T_i has obtained a **shared-mode lock** on item X, then T_i can read, but cannot write, X

- **Exclusive**: If a transaction T_i has obtained an **exclusive-mode lock** on item X, then T_i can both read and write X

Operations

- Shared/exclusive mode locking scheme has three locking operations on an item *X*
  - *read_lock(X)* (or) *shared_lock(X)*
  - *write_lock(X)* (or) *exclusive_lock(X)*
  - *unlock(X)*

- Every transaction must **request** a lock in an appropriate mode on data item *X*, depending on the types of operations that it will perform on

Locking and unlocking 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;
```

```
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;
```

Rules enforced in shared/exclusive locking scheme

1. A transaction *T* must issue the operation *read_lock(X)* or *write_lock(X)* before any *read* operation on *X* is performed in *T*

2. A transaction *T* must issue the operation *write_lock(X)* before any *write* operation is performed on *X* in *T*

3. A transaction *T* must issue the operation *unlock(X)* after all *read* and *write* operations on *X* are completed in *T*

4. A transaction *T* should not issue a *read_lock(X)*/ *write_lock(X)* operation if it already holds a *read_lock* or a *write_lock* on item *X*

5. A transaction *T* should not issue an *unlock(X)* operation unless it already holds a *read_lock* or a *write_lock* on item *X*

Conversion of locks

- Sometimes it is desirable to relax the rule 4 under ***certain cenarios***

- That is, a transaction is already holding a lock on an item *X* and the transaction is allowed to ***convert*** the lock from one locked state to another

- Two types of lock conversions
  1. Upgarde a lock
  2. Downgrade a lock

Upgrading and downgrading of locks

- **Upgrading a lock**: A transaction *T* first issued a *read_lock(X)* and later issues a *write_lock(X)*
  - This upgradation is possible ***only if*** *T* is the only transaction holding a read lock on *X* at the time it issues the *write_lock(X)* operation; otherwise, T must wait

- **Downgrading a lock**: A transaction *T* first issued a *write_lock(X)* and later issues a *read_lock(X)*

- When upgrading and downgrading of locks is used, the lock table must be updated accordingly

Do binary locks or read/write locks guarantee serializability?

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);

Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50

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);	

Result of schedule S:
X=50, Y=50
(nonserializable)

The two-phase locking protocol

- To guarantee serializability, we must follow *an additional protocol* concerning the positioning of locking and unlocking operations in every transaction

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

The two phases

- A transaction following the two-phase locking protocal has two phases
  1. **Growing phase**: During this phase, new locks on items can be acquired but none can be released
  2. **Shrinking phase**: During this phase, existing locks can be released but no new locks can be acquired
- If lock conversion is allowed, then upgrading of locks must be done during the expanding phase, and downgrading of locks must be done in the shrinking phase

Example

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);

$T_1{}'$

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);

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);	

The 2PL scheme guarntees serializability

- If *every* transaction in a schedule follows the two-phase locking protocol, the schedule is *guaranteed to be serializable*

- On contrary, assume 2PL doesn't ensure serializability
- Let T_1, T_2, ..., T_n be the transactions which obey 2PL scheme and produce a non-serializable schedule
- Implies that, there must be a cycle in the precedence graph and let this cycle be (WLG) $T_n \rightarrow T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_{n-1} \rightarrow T_n$
- Let t_i be the time at which T_i obtains its *last* lock, for all $1 \leq i \leq n$
- For all the transactions $T_i \rightarrow T_j$, it must be that $t_i < tj$
- As there is a cycle, we get $t_n < t_1 < t_2 < \ldots < t_{n-1} < t_n$

Thank you!