11 Concurrency Control and Transactions

- Problem restatement
- Locking
- Optimistic control
- Timestamping

11.1 Why concurrency control?

- To increase performance, multiple transactions must be able to carry on work simultaneously...
- ...but if data is shared, then can lead to problems such as lost updates and inconsistent retrievals.
- So we must ensure schedules of access to data for concurrent transactions are computationally equivalent to a serial schedule of the transactions.

11.2 Locking

- As in operating systems, locks control access for different clients
- Granularity of data locked should be small so as to maximise concurrency, with trade-off against complexity.
- To prevent intermediate leakage, once lock is obtained, it must be held till transaction commits or aborts

11.2.1 Conflict rules

- Conflict rules determine rules of lock usage
- If operations are not in conflict, then locks can be shared $\Rightarrow$ read locks are shared
- Operations in conflict imply operations should wait on lock $\Rightarrow$ write waits on read or write lock, read

waits on write lock

- Since can't predict other item usage till end of transactions, locks must be held till transaction commits or aborts.
- If operation needs to do another operation on same data then *promotes* lock if necessary and possible - operation may conflict with existing shared lock

11.2.2 Rules for strict two phase locking

1. When operation accesses data item within transaction
   (a) If item isn't locked, then server locks and proceeds
   (b) If item is held in a conflicting lock by another transaction, transaction must wait till lock released
   (c) If item is held by non-conflicting

lock, lock is shared and operation proceeds

(d) If item is already locked by same transaction, lock is promoted if possible (refer to rule b)

2. When transaction commits or aborts, locks are released

11.2.3 Locking Implementation

- Locks generally implemented by a lock manager

  lock(transId,DataItem,LockType) Lock the specified item if possible, else wait according to rules above

  unLock(transId) Release all locks held by the transaction

- Lock manager generally multi-threaded, requiring internal synchronisation
- Heavyweight implementation

11.2.4 Example

Transactions T and U.

- T: i.read(), j.write(44)
- U: i.write(55),j. read(), j.write(66)

Question What are the possible schedules allowed under strict locking?

Question Are there any schedules computationally equivalent to a serial schedule which are disallowed?

11.2.5 Deadlocks

- Locks imply deadlock, under following conditions
  1. Limited access (eg mutex or finite buffer)
  2. No preemption (if someone has resource can't take it away)
  3. Hold and wait. Independent threads must possess some of its needed

resources and waiting for the remainder to become free.

4. Circular chain of requests and ownership.

- Most common way of protecting against deadlock is through timeouts. After timeout, lock becomes *vulnerable* and can be broken if another transaction attempts to gain lock, leading to aborted transactions

11.2.6 Drawbacks of Locking

- Locking is overly restrictive on the degree of concurrency
- Deadlocks produce unnecessary aborts
- Lock maintenance is an overhead, that may not be required

11.3 Optimistic Concurrency Control

- Most transactions do not conflict with each other
- So proceed without locks, and check on close of transaction that there were no conflicts
  - Analyse conflicts in *validation* process
  - If conflicts could result in non-serialisable schedule, abort one or more transactions
  - else commit

11.3.1 Implementation of Optimistic Concurrency Control

Transaction has following phases

1. Read phase in which clients read values and acquire tentative versions of items they wish to update
2. Validation phase in which operations are checked to see if they are

in conflict with other transactions - complex part. If invalid, then abort.

3. If validated, tentative versions are written to permanence, and transaction can commit (or abort).

11.3.2 Validation approaches

- Validation based upon conflict rules for serialisability
- Validation can be either against completed transactions or active transactions - backward and forward validation.
- Simplify by ensuring only one transaction in validation and write phase at one time
- Trade-off between number of comparisons, and transactions that must be stored.

Forward Validation

1. A transaction in validation is compared against all transactions that haven't yet committed
2. Writes may affect ongoing reads
3. The write set of the validating transaction is compared against the read sets of all other active transactions.
4. If the sets conflict, then either abort validating transaction, delay validation till conflicting transaction completes, or abort conflicting transaction.

Backward validation

1. Writes of current transaction can't affect previous transaction reads, so we only worry about reads with overlapping transactions that have committed.

2. If current read sets conflict with already validated overlapping transactions write sets, then abort validating transaction

11.4 Timestamping

Operates on tentative versions of data

- Each Transaction receives global unique timestamp on initiation
- Every object, x, in the system or database carries the maximum (ie youngest) timestamp of last transaction to read RTM(x)[1] and maximum of last transaction to write WTM(x)[2]
- If transaction requests operation that conflicts with younger transaction, older transaction restarted with new timestamp.

[1] Read Timestamp Maximum
[2] Write Timestamp Maximum

- Transactions committed in order of timestamps, so a transaction may have to wait for earlier transaction to commit or abort before committing.
- Since tentative version is only written when transaction is committed, read operations may have to wait until the last transaction to write has committed.

An operation in transaction T_i with start time TS_i is valid if:

- The operation is a *read* operation and the object was last written by an older transaction ie $TS_i > WTM(x)$. If read permissible, $RTM(x) = MAX(TS_i, RTM(x))$
- The operation is a *write* operation and the object was last read and written by older transactions ie $TS_i > RTM(x)$ and $TS_i >$

$WTM(x)$. If permissible, $WTM(x) = TS_i$

11.5 Summary

- Locks are commonest ways of providing consistent concurrency
- Optimistic concurrency control and timestamping used in some systems
- But, consistency in concurrency is application dependent - for shared editors, people may prefer to trade speed of execution against possibilities of conflict resolution. Problems can occur with long term network partition. Approaches based on notification and people resolution becoming popular.