Chapter 16
Transactions and Concurrency Control
Chapter 16 Transactions and Concurrency Control

16.1 Introduction
16.2 Transactions
16.3 Nested transactions
16.4 Locks
16.5 Optimistic concurrency control
16.6 Timestamp ordering
16.7 Comparison of methods for concurrency control
16.8 Summary
16.1 Introduction

• Transaction
  – Definition: a sequence of sever operations that is guaranteed by the server to be \textit{atomic} in the presence of multiple clients and server crashes

• The goal of transactions
  – the objects managed by a server must remain in a consistent state
    • when they are accessed by multiple transactions and
    • in the presence of server crashes

• Recoverable objects
  – can be recovered after their server crashes (recovery in Chapter 14)
  – objects are stored in permanent storage

• Failure model
  – transactions deal with crash failures of processes and omission failures of communication

• Designed for an asynchronous system
  – It is assumed that messages may be delayed
Figure 16.1 Operations of the Account interface

\[\text{deposit}(\text{amount})\]
\[\text{withdraw}(\text{amount})\]
\[\text{getBalance}() \rightarrow \text{amount}\]
\[\text{setBalance}(\text{amount})\]

Used as an example. Each Account is represented by a remote object whose interface provides operations for making deposits and withdrawals and for setting and getting the balance.

Operations of the Branch interface

\[\text{create}(\text{name}) \rightarrow \text{account}\]
\[\text{lookUp}(\text{name}) \rightarrow \text{account}\]
\[\text{branchTotal}() \rightarrow \text{amount}\]

and each Branch of the bank is represented by a remote object whose interface provides operations for creating a new account, looking one up by name and enquiring about the total funds at the branch. It stores a correspondence between account names and their remote object references.
Atomic operations at server

• Simple synchronisation: without transactions
  – when a server uses multiple threads it can perform several client operations concurrently
  – if we allowed deposit and withdraw to run concurrently we could get inconsistent results

• Objects should be designed for safe concurrent access e.g. in Java use synchronized methods, e.g.
  – public synchronized void deposit(int amount) throws RemoteException

• Atomic operations are free from interference from concurrent operations in other threads
  – use any available mutual exclusion mechanism (e.g. mutex)
Client cooperation by means of synchronizing server operations

- In some applications clients share resources via a server and depend on one another to progress
  - e.g. some clients update server objects and others access them
  - e.g. one is a producer and another a consumer
  - e.g. one sets a lock and the other waits for it to be released

- Servers implementation with multiple threads
  - Not a good idea for a waiting client to poll the server to see whether a resource is yet available
    - Unfair (later clients might get earlier turns)
  - Java \textit{wait} and \textit{notify} methods allow threads to communicate with one another and to solve these problems
    - e.g. when a client requests a resource, the server thread waits until it is notified that the resource is available
Lampson’s failure model deals with failures of disks, servers, and communication
– algorithms work correctly when predictable faults occur
– but if a disaster occurs, we cannot say what will happen
• Writes to permanent storage may fail
  – e.g. by writing nothing or a wrong value (write to wrong block is a disaster)
  – reads can detect bad blocks by checksum
• Servers may crash occasionally
  – when a crashed server is replaced by a new process its memory is cleared and it carries out a recovery procedure to get its objects’ state
  – faulty servers are made to crash so they do not produce arbitrary failures
• There may be an arbitrary delay before a message arrives.
  A message may be lost, duplicated or corrupted
  – recipient can detect corrupt messages (by checksum)
  – forged messages and undetected corrupt messages are disasters
16.2 Transactions

- Some applications require a sequence of client requests to a server to be atomic in the sense that
  1. they are free from interference by operations being performed on behalf of other concurrent clients; and
  2. either all of the operations must be completed successfully or they must have no effect at all in the presence of server crashes.

- Retrospect:
  - Transactions originate from database management systems
  - Transactional file servers were built in the 1980s
  - Transactions on distributed objects late 80s and 90s
  - Middleware components e.g. CORBA Transaction service

- Transactions apply to recoverable objects and are intended to be atomic
  - Servers 'recover' - they are restated and get their objects from permanent storage
A client’s banking transaction

Transaction T:
\[ a.\text{withdraw}(100); \]
\[ b.\text{deposit}(100); \]
\[ c.\text{withdraw}(200); \]
\[ b.\text{deposit}(200); \]

• This transaction specifies a sequence of related operations involving bank accounts named A, B and C and referred to as a, b and c in the program
• The first two operations transfer $100 from A to B
• The second two operations transfer $200 from C to B
Atomicity of transactions

The atomicity has two aspects

1. **All or nothing**
   - It either completes successfully, and the effects of all of its operations are recorded in the objects or
   - It has no effect at all (if it fails or is aborted)

Two further aspects of its own
- failure atomicity: effects are atomic even when the server crashes;
- durability: after a transaction has completed successfully, all its effects are saved in permanent storage.

2. **Isolation**
   - Each transaction must be performed without interference from other transactions
     - There must be no observation by other transactions of a transaction's intermediate effects
     - Concurrency control ensures isolation
Operations in the *Coordinator* interface

- Transaction capabilities may be added to a server of recoverable objects
  - each transaction is created and managed by a *Coordinator* object whose interface follows:

  \[
  \text{openTransaction()} \rightarrow \text{trans}; \\
  \text{starts a new transaction and delivers a unique TID } \text{trans}. \text{ This identifier will be used in the other operations in the transaction.}
  \]
  \[
  \text{closeTransaction(trans)} \rightarrow (\text{commit, abort}); \\
  \text{ends a transaction: a } \text{commit} \text{ return value indicates that the transaction has committed; an } \text{abort} \text{ return value indicates that it has aborted.}
  \]
  \[
  \text{abortTransaction(trans);} \\
  \text{aborts the transaction.}
  \]
A transaction is either successful (it commits)
- the coordinator sees that all objects are saved in permanent storage

or it is aborted by the client or the server
- make all temporary effects invisible to other transactions
- how will the client know when the server has aborted its transaction?
- the client finds out next time it tries to access an object at the server
16.2.1 Concurrency control

• Two well-known concurrent transaction problems
  – Lost update
    • a lost update occurs when two transactions both read the old value of a variable and use it to calculate a new value
  – Inconsistent retrievals
    • inconsistent retrievals occur when a retrieval transaction observes values that are involved in an ongoing updating transaction

• Assumption
  – the operations deposit, withdraw, getBalance and setBalance are synchronized operations
  – that is, their effect on the account balance is atomic
## The lost update problem

<table>
<thead>
<tr>
<th><strong>Transaction</strong> $T$</th>
<th><strong>Transaction</strong> $U$</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>balance = b.getBalance();</code></td>
<td><code>balance = b.getBalance();</code></td>
</tr>
<tr>
<td><code>b.setBalance(balance*1.1);</code></td>
<td><code>b.setBalance(balance*1.1);</code></td>
</tr>
<tr>
<td><code>a.withdraw(balance/10)</code></td>
<td><code>c.withdraw(balance/10)</code></td>
</tr>
</tbody>
</table>

| `balance = b.getBalance();` | $200$ | `$200$` |
| `b.setBalance(balance*1.1);` | $220$ | `$220$` |
| `a.withdraw(balance/10)` | $80$ | `$80$` |
| `c.withdraw(balance/10)` | $280$ | `$280$` |

- The initial balances of accounts A, B, C are $100, $200, $300
- Both transfer transactions increase B's balance by 10%

The net effect should be to increase B by 10% twice - 200, 220, 242, but it only gets to 220. $T$'s update is lost.
The inconsistent retrievals problem

<table>
<thead>
<tr>
<th>Transaction $V$:</th>
<th>Transaction $W$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$.withdraw(100)</td>
<td>$a$.Branch.branchTotal()</td>
</tr>
<tr>
<td>$b$.deposit(100)</td>
<td></td>
</tr>
<tr>
<td>$a$.withdraw(100);</td>
<td>$total = a$.getBalance()</td>
</tr>
<tr>
<td>$\quad$ $100$</td>
<td>$\quad$ $100$</td>
</tr>
<tr>
<td>$b$.deposit(100)</td>
<td>$total = total + b$.getBalance()</td>
</tr>
<tr>
<td>$\quad$ $300$</td>
<td>$\quad$ $300$</td>
</tr>
</tbody>
</table>

- $V$ transfers $100$ from $A$ to $B$ while $W$ calculates branch total (which should be $600$)

we see an inconsistent retrieval because $V$ has only done the withdraw part when $W$ sums balances of $A$ and $B$
Serial equivalence

• The **same effect** means
  – the read operations return the same values
  – the instance variables of the objects have the same values at the end
• If each one of a set of transactions has the correct effect when *done on its own*
  then if they are *done one at a time in some order* the effect will be correct
• A **serially equivalent interleaving** is one in which the combined effect is the same as if the transactions had been done one at a time in some order
  – The transactions are scheduled to avoid overlapping access to the accounts accessed by both of them
A serially equivalent interleaving of $T$ and $U$ (lost updates cured)

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$balance = b.getBalance()$</td>
<td>$balance = b.getBalance()$</td>
</tr>
<tr>
<td>$b.setBalance(balance*1.1)$</td>
<td>$b.setBalance(balance*1.1)$</td>
</tr>
<tr>
<td>$a.withdraw(balance/10)$</td>
<td>$c.withdraw(balance/10)$</td>
</tr>
</tbody>
</table>

- $balance = b.getBalance()$ $\$200$
- $b.setBalance(balance*1.1)$ $\$220$
- $a.withdraw(balance/10)$ $\$80$
- $balance = b.getBalance()$ $\$220$
- $b.setBalance(balance*1.1)$ $\$242$
- $c.withdraw(balance/10)$ $\$278$

- if one of $T$ and $U$ runs before the other, they can’t get a lost update,
- the same is true if they are run in a serially equivalent ordering

their access to $B$ is serial, the other part can overlap
A serially equivalent interleaving of $V$ and $W$ (inconsistent retrievals cured)

<table>
<thead>
<tr>
<th>Transaction $V$:</th>
<th>Transaction $W$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$.withdraw(100); $a$.Branch.branchTotal()</td>
<td></td>
</tr>
<tr>
<td>$b$.deposit(100)</td>
<td></td>
</tr>
<tr>
<td>$a$.withdraw(100); $a$.getBalance()</td>
<td></td>
</tr>
<tr>
<td>$b$.deposit(100) $b$.getBalance()</td>
<td></td>
</tr>
<tr>
<td>$\text{total} = a\text{.getBalance()}$ $\text{total} = \text{total} + b\text{.getBalance()}$</td>
<td></td>
</tr>
<tr>
<td>$\text{total} = \text{total} + c\text{.getBalance()}$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

- if $W$ is run before or after $V$, the problem will not occur
- therefore it will not occur in a serially equivalent ordering of $V$ and $W$
- the illustration is serial, but it need not be

we could overlap the first line of $W$ with the second line of $V$
Read and write operation conflict rules

<table>
<thead>
<tr>
<th>Operations of different transactions</th>
<th>Conflict</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>read read</td>
<td>No</td>
<td>Because the effect of a pair of read operations does not depend on the order in which they are executed</td>
</tr>
<tr>
<td>read write</td>
<td>Yes</td>
<td>Because the effect of a read and a write operation depends on the order of their execution</td>
</tr>
<tr>
<td>write write</td>
<td>Yes</td>
<td>Because the effect of a pair of write operations depends on the order of their execution</td>
</tr>
</tbody>
</table>

- Conflicting operations: a pair of operations conflicts if their combined effect depends on the order in which they were performed
  - e.g. read and write (whose effects are the result returned by read and the value set by write)
Serial equivalence
defined in terms of conflicting operations

• For two transactions to be **serially equivalent**, it is necessary and sufficient that
  
  all pairs of conflicting operations of the two transactions be executed in the same order at all of the objects they both access

• Consider
  
  – T and U access i and j
    • \( T: x = \text{read}(i); \text{write}(i, 10); \text{write}(j, 20); \)
    • \( U: y = \text{read}(j); \text{write}(j, 30); z = \text{read}(i); \)
  
  – serial equivalence requires that either
    • \( T \) accesses \( i \) before \( U \) and \( T \) accesses \( j \) before \( U \). or
    • \( U \) accesses \( i \) before \( T \) and \( U \) accesses \( j \) before \( T \)

• Serial equivalence is used as a criterion for designing concurrency control schemes. Three alternative approaches
  
  – Locking: used by most practical systems
  – Optimistic concurrency control
  – Timestamp ordering
A non-serially equivalent interleaving of operations of transactions $T$ and $U$

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = \text{read}(i)$</td>
<td>$y = \text{read}(j)$</td>
</tr>
<tr>
<td>$\text{write}(i, 10)$</td>
<td>$\text{write}(j, 30)$</td>
</tr>
<tr>
<td>$\text{write}(j, 20)$</td>
<td></td>
</tr>
</tbody>
</table>

- Each transaction’s access to $i$ and $j$ is serialized w.r.t one another, but
  - $T$ makes all accesses to $i$ before $U$ does
  - $U$ makes all accesses to $j$ before $T$ does

therefore this interleaving is not serially equivalent
16.2.2 Recoverability from aborts

If a transaction aborts, the server must make sure that other concurrent transactions do not see any of its effects, we study two problems:

• ‘dirty reads’
  – an interaction between a read operation in one transaction and an earlier write operation on the same object (by a transaction that then aborts)
  – a transaction that committed with a ‘dirty read’ is not recoverable

• ‘premature writes’
  – interactions between write operations on the same object by different transactions, one of which aborts

For illustration, assume getBalance is a read operation and setBalance a write operation
A dirty read when transaction $T$ aborts

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a\text{.getBalance()}$</td>
<td>$a\text{.getBalance()}$</td>
</tr>
<tr>
<td>$a\text{.setBalance(balance + 10)}$</td>
<td>$a\text{.setBalance(balance + 20)}$</td>
</tr>
<tr>
<td>$balance = a\text{.getBalance()}$</td>
<td>$balance = a\text{.getBalance()}$</td>
</tr>
<tr>
<td>$100$</td>
<td>$110$</td>
</tr>
<tr>
<td>$a\text{.setBalance(balance + 10)}$</td>
<td>$a\text{.setBalance(balance + 20)}$</td>
</tr>
<tr>
<td>$110$</td>
<td>$160$</td>
</tr>
<tr>
<td>$commit\ transaction$</td>
<td>$commit\ transaction$</td>
</tr>
</tbody>
</table>

$U$ has committed, so it cannot be undone $\Rightarrow$ dirty read
**Premature writes – overwriting uncommitted values**

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a.setBalance(105)$</td>
<td>$a.setBalance(110)$</td>
</tr>
<tr>
<td>$a.setBalance(105)$</td>
<td>$100$</td>
</tr>
<tr>
<td>$a.setBalance(105)$</td>
<td>$105$</td>
</tr>
<tr>
<td>$a.setBalance(110)$</td>
<td>$110$</td>
</tr>
</tbody>
</table>

Some database systems keep ‘before images’ and restore them after aborts

- e.g. $100$ is before image of $T$’s write, $105$ is before image of $U$’s write
- If $U$ aborts we get the correct balance of $105$,
- But if $U$ commits and then $T$ aborts, we get $100$ instead of $110$
Strict executions of transactions

• Curing premature writes:
  – if a recovery scheme uses before images
    • write operations must be delayed until earlier transactions that updated the same objects have either committed or aborted

• Strict executions of transactions
  – to avoid both ‘dirty reads’ and ‘premature writes’.
    • delay both read and write operations
  – executions of transactions are called \textit{strict} if both \textit{read} and \textit{write} operations on an object are delayed until all transactions that previously wrote that object have either committed or aborted.
  – the strict execution of transactions enforces the desired property of isolation

• \textit{Tentative versions} are used during progress of a transaction
  – objects in tentative versions are stored in volatile memory
16.3 Nested transactions

- Transactions may be composed of other transactions
  - Several transactions may be started from within a transaction
  - We have a **top-level transaction** and **subtransactions** which may have their own subtransactions
Nested transactions

• To a parent, a subtransaction is atomic with respect to failures and concurrent access
  – Transactions at the same level (e.g. $T_1$ and $T_2$) can run concurrently but access to common objects is serialised
  – A subtransaction can fail independently of its parent and other subtransactions
    • When it aborts, its parent decides what to do, e.g. start another subtransaction or give up
• Commitment
  – A transaction may commit or abort only after its child transactions have completed
  – A subtransaction decides independently to commit provisionally or to abort. Its decision to abort is final
  – When a parent aborts, all of its subtransactions are aborted
  – When a subtransaction aborts, parent can decide whether to abort or not
  – If the top-level transaction commits, then all of the subtransactions that have provisionally committed can commit too, provided that none of their ancestors has aborted
Advantages of nested transactions (over flat ones)

- Subtransactions may run concurrently with other subtransactions at the same level
  - this allows additional concurrency in a transaction.
  - when subtransactions run in different servers, they can work in parallel.
    - e.g. consider the `branchTotal` operation
    - it can be implemented by invoking `getBalance` at every account in the branch.
      - these can be done in parallel when the branches have different servers
- Subtransactions can commit or abort independently
  - this is potentially more robust
  - a parent can decide on different actions according to whether a subtransaction has aborted or not
Summary on transactions

We consider only transactions at a single server, they are:

- **atomic in the presence of concurrent transactions**
  - which can be achieved by serially equivalent executions

- **atomic in the presence of server crashes**
  - they save committed state in permanent storage (recovery Ch.14)
  - they use strict executions to allow for aborts
  - they use tentative versions to allow for commit/abort

- **nested transactions are structured from sub-transactions**
  - they allow concurrent execution of sub-transactions
  - they allow independent recovery of sub-transactions
16.4 Locks

• Transactions must be scheduled so that their effect on shared objects is serially equivalent
  a) all access by a transaction to a particular object must be serialized with respect to another transaction’s access
  b) all pairs of conflicting operations of two transactions should be executed in the same order
     – A server can achieve serial equivalence by serializing access to objects, e.g. by the use of locks
     – to ensure (b), a transaction is not allowed any new locks after it has released a lock

• Two-phase locking - has a ‘growing’ and a ‘shrinking’ phase
  – growing phase: new locks are acquired
  – shrinking phase: the locks are released
A simple serializing mechanism: exclusive locks

• The server attempts to lock any object that is about to be used by any operation of a client’s transaction
  – If a client requests access to an object that is locked by another client’s transaction, the request is suspended and the client must wait until the object is unlocked

<table>
<thead>
<tr>
<th>Transaction T:</th>
<th>Transaction U:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( balance = b.getBalance() )</td>
<td>( balance = b.getBalance() )</td>
</tr>
<tr>
<td>( b.setBalance(bal*1.1) )</td>
<td>( b.setBalance(bal*1.1) )</td>
</tr>
<tr>
<td>( a.withdraw(bal/10) )</td>
<td>( c.withdraw(bal/10) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operations</th>
<th>Locks</th>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>openTransaction</td>
<td>lockA</td>
<td>openTransaction</td>
<td>lockB</td>
</tr>
<tr>
<td>( bal = b.getBalance() )</td>
<td></td>
<td>( bal = b.getBalance() )</td>
<td>waits for ( T )’s lock onB</td>
</tr>
<tr>
<td>( b.setBalance(bal*1.1) )</td>
<td></td>
<td>( b.setBalance(bal*1.1) )</td>
<td></td>
</tr>
<tr>
<td>( a.withdraw(bal/10) )</td>
<td></td>
<td>( c.withdraw(bal/10) )</td>
<td></td>
</tr>
<tr>
<td>closeTransaction</td>
<td>unlockA, B</td>
<td>closeTransaction</td>
<td>unlockB, C</td>
</tr>
</tbody>
</table>

Figure 16.14 (same as 16.7)
Strict two-phase locking

• Any locks applied during the progress of a transaction are held until the transaction commits or aborts
  – Strict executions prevent dirty reads and premature writes (if transactions abort)
  – A transaction that reads or writes an object must be delayed until other transactions that wrote the same object have committed or aborted
  – For recovery purposes, locks are held until updated objects have been written to permanent storage

• Granularity - apply locks to small things e.g. bank balances
  – There are no assumptions as to granularity in the schemes we present

• Read operations of different transactions do not conflict, so exclusive locks reduce concurrency more than necessary
  – The ‘many reader/single writer’ scheme allows several transactions to read an object or a single transaction to write it (but not both)
  – It uses read locks and write locks
    • read locks are sometimes called shared locks
Lock compatibility

The operation conflict rules tell us that:

1. If a transaction $T$ has already performed a *read* operation on a particular object, then a concurrent transaction $U$ must not *write* that object until $T$ commits or aborts.

2. If a transaction $T$ has already performed a *write* operation on a particular object, then a concurrent transaction $U$ must not *read* or *write* that object until $T$ commits or aborts.

<table>
<thead>
<tr>
<th>For one object</th>
<th>Lock already set</th>
<th>Lock requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>read</td>
<td>OK</td>
<td>wait</td>
</tr>
<tr>
<td>write</td>
<td>wait</td>
<td>wait</td>
</tr>
</tbody>
</table>

For one object Lock requested

To enforce 1, a request for a write lock is delayed by the presence of a read lock belonging to another transaction.

To enforce 2, a request for a read lock or write lock is delayed by the presence of a write lock belonging to another transaction.

Figure 16.15
Lock promotion

• Locking prevents the inconsistent retrievals problem
  – If the retrieval transaction comes first, its read locks delay the update transaction
  – If the retrieval transaction comes second, its request for read locks causes it to be delayed until the update transaction has completed
• *Lock promotion* is required to prevent the lost update problem
  – Lost updates occur when two transactions read an object and then use it to calculate a new value
  – Lost updates are prevented by making later transactions delay their reads until the earlier ones have completed
    • Each transaction sets a read lock when it reads and then promotes it to a write lock when it writes the same object
    • when another transaction requires a read lock it will be delayed
  – *Lock promotion*: the conversion of a lock to a stronger lock – that is, a lock that is more exclusive
Use of locks in strict two-phase locking

1. When an operation accesses an object within a transaction:
   (a) If the object is not already locked, it is locked and the operation proceeds.
   (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)

2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction

• The server applies locks when the read/write operations are about to be executed
• The server releases a transaction’s locks when it commits or aborts
Lock implementation

• The granting of locks will be implemented by a separate object in the server that we call the lock manager
• The lock manager holds a set of locks, for example in a hash table
• Each lock is an instance of the class Lock (Fig 16.17) and is associated with a particular object
  – its variables refer to the object, the holder(s) of the lock and its type
• The lock manager code uses wait (when an object is locked) and notify when the lock is released
• The lock manager provides setLock and unLock operations for use by the server
Figure 16.17 Lock class

```java
public class Lock {
    private Object object;  // the object being protected by the lock
    private Vector holders; // the TIDs of current holders
    private LockType lockType; // the current type
    public synchronized void acquire(TransID trans, LockType aLockType) {
        while(/*another transaction holds the lock in conflicting mode*/) {
            try {
                wait();
            } catch (InterruptedException e) {/*...*/ }
        }
        if(holders.isEmpty()) { // no TIDs hold lock
            holders.addElement(trans);
            lockType = aLockType;
        } else if(/*another transaction holds the lock, share it*/) {
            if(/* this transaction not a holder*/) holders.addElement(trans);
        } else if(/* this transaction is a holder but needs a more exclusive lock*/) {
            lockType.promote();
        }
    }
    public synchronized void release(TransID trans) {
        holders.removeElement(trans); // remove this holder
        // set locktype to none
        notifyAll();
    }
}
```
public class LockManager {
    private Hashtable theLocks;

    public void setLock(Object object, TransID trans, LockType lockType) {
        Lock foundLock;
        synchronized(this) {
            // find the lock associated with object
            // if there isn’t one, create it and add to the hashtable
            foundLock.acquire(trans, lockType);
        }
    }

    // synchronize this one because we want to remove all entries
    public synchronized void unLock(TransID trans) {
        Enumeration e = theLocks.elements();
        while (e.hasMoreElements()) {
            Lock aLock = (Lock) (e.nextElement());
            if (/* trans is a holder of this lock*/ ) aLock.release(trans);
        }
    }
}
Deadlock with write locks

<table>
<thead>
<tr>
<th>Transaction $T$</th>
<th>Operations</th>
<th>Locks</th>
<th>Transaction $U$</th>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a.\text{deposit}(100)$</td>
<td>write lock $A$</td>
<td></td>
<td>$b.\text{deposit}(200)$</td>
<td>write lock $B$</td>
<td></td>
</tr>
<tr>
<td>$b.\text{withdraw}(100)$</td>
<td></td>
<td>waits for $U$'s</td>
<td></td>
<td>$a.\text{withdraw}(200)$</td>
<td>waits for $T$'s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lock on $B$</td>
<td></td>
<td></td>
<td>lock on $A$</td>
</tr>
</tbody>
</table>

The \textit{deposit} and \textit{withdraw} methods are atomic. Although they read as well as write, they acquire write locks. $T$ accesses $A \rightarrow B$, $U$ accesses $B \rightarrow A$

When locks are used, each of $T$ and $U$ acquires a lock on one account and then gets blocked when it tries to access the account the other one has locked. We have a 'deadlock'.
• Definition of deadlock
  – deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock.
  – a *wait-for graph* can be used to represent the waiting relationships between current transactions

In a wait-for graph the nodes represent transactions and the edges represent wait-for relationships between transactions
A cycle in a wait-for graph

Suppose a wait-for graph contains a cycle $T \rightarrow U \rightarrow \ldots \rightarrow V \rightarrow T$

- Each transaction waits for the next transaction in the cycle
- All of these transactions are blocked waiting for locks
- None of the locks can ever be released (the transactions are deadlocked)
- If one transaction is aborted, then its locks are released and that cycle is broken
Another wait-for graph

- $T$, $U$ and $V$ share a read lock on $C$ and
- $W$ holds write lock on $B$ (which $V$ is waiting for)
- $T$ and $W$ then request write locks on $C$ and deadlock occurs
  - e.g. $V$ is in two cycles - look on the left
Deadlock prevention

• Deadlock prevention is unrealistic
  – e.g. lock all of the objects used by a transaction when it starts
    • unnecessarily restricts access to shared resources.
    • it is sometimes impossible to predict at the start of a transaction which objects will be used.
• Deadlock can also be prevented by requesting locks on objects in a predefined order
  – but this can result in premature locking and a reduction in concurrency
Deadlock detection

Deadlock detection: finding cycles in the wait-for graph

- After detecting a deadlock, a transaction must be selected to be aborted to break the cycle
- The software for deadlock detection can be part of the lock manager
- It holds a representation of the wait-for graph so that it can check it for cycles from time to time
- Edges are added to the graph and removed from the graph by the lock manager’s setLock and unLock operations
- When a cycle is detected, choose a transaction to be aborted and then remove from the graph all the edges belonging to it
- It is hard to choose a victim - e.g. choose the oldest or the one in the most cycles
Timeouts on locks

Lock timeouts can be used to resolve deadlocks

- Each lock is given a limited period in which it is *invulnerable*
  - after this time, a lock becomes *vulnerable*
- Provided that no other transaction is competing for the locked object, the vulnerable lock is allowed to remain
- But if any other transaction is waiting to access the object protected by a vulnerable lock, the lock is broken
  - (that is, the object is unlocked) and the waiting transaction resumes
- The transaction whose lock has been broken is normally aborted

Problems with lock timeouts

- Locks may be broken when there is no deadlock
- If the system is overloaded, lock timeouts will happen more often and long transactions will be penalised
- It is hard to select a suitable length for a timeout
16.5 Optimistic concurrency control

• The likelihood of two transactions conflicting is low
  – a transaction proceeds without restriction until the closeTransaction (no waiting, therefore no deadlock)
  – it is then checked to see whether it has come into conflict with other transactions
  – when a conflict arises, a transaction is aborted
• Each transaction has three phases
  – Working phase
    • the transaction uses a tentative version of the objects it accesses (dirty reads can’t occur as we read from a committed version or a copy of it)
    • the coordinator records the readset and writeset of each transaction
  – Validation phase
    • at closeTransaction the coordinator validates the transaction (looks for conflicts)
    • if the validation is successful the transaction can commit.
    • if it fails, either the current transaction, or one it conflicts with is aborted
  – Update phase
    • If validated, the changes in its tentative versions are made permanent.
    • read-only transactions can commit immediately after passing validation
Validation of transactions

We use the read-write conflict rules
- to ensure a particular transaction is serially equivalent with respect to all other overlapping transactions
- Each transaction is given a transaction number when it starts validation (the number is kept if it commits)
- The rules ensure serializability of transaction $T_v$ (transaction being validated) with respect to transaction $T_i$

<table>
<thead>
<tr>
<th>$T_v$</th>
<th>$T_i$</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>write</td>
<td>read</td>
<td>1. $T_i$ must not read objects written by $T_v$ forward</td>
</tr>
<tr>
<td>read</td>
<td>write</td>
<td>2. $T_v$ must not read objects written by $T_i$ backward</td>
</tr>
<tr>
<td>write</td>
<td>write</td>
<td>3. $T_i$ must not write objects written by $T_v$ and $T_v$ must not write objects written by $T_i$</td>
</tr>
</tbody>
</table>

Validation can be simplified by omitting rule 3 (if no overlapping of validate and update phases)
Validation of transactions

The earlier committed transactions are \( T_1, T_2 \) and \( T_3 \). \( T_1 \) committed before \( T_v \) started. (\emph{earlier} means they started validation earlier)

- Backward validation: check \( T_v \) with preceding overlapping transactions
  - Rule 1 (\( T_v \)'s \emph{write} vs \( T_i \)'s \emph{read}) is satisfied because reads of earlier transactions were done before \( T_v \) entered validation (and possible updates)
  - Rule 2 - check if \( T_v \)'s read set overlaps with write sets of earlier \( T_i \)
  - \( T_2 \) and \( T_3 \) committed before \( T_v \) finished its working phase.
  - Rule 3 - (\emph{write vs write}) assume no overlap of validate and commit.
Backward validation of transaction $T_v$

```java
boolean valid = true;
for (int $T_i = startTn+1; T_i <= finishTn; T_i++)
    if (read set of $T_v$ intersects write set of $T_i$) valid = false;
```

- $startTn$ is the biggest transaction number assigned to some other committed transaction when $T_v$ started its working phase
- $finishTn$ is biggest transaction number assigned to some other committed transaction when $T_v$ started its validation phase
- In figure, $StartTn + 1 = T_2$ and $finishTn = T_3$. In backward validation, the read set of $T_v$ must be compared with the write sets of $T_2$ and $T_3$.
- the only way to resolve a conflict is to abort $T_v$

To carry out this algorithm, we must keep write sets of recently committed transactions.
Forward validation

• Rule 1. the write set of $T_v$ is compared with the read sets of all overlapping active transactions
  – In Figure 16.28, the write set of $T_v$ must be compared with the read sets of $active1$ and $active2$.
• Rule 2. (read $T_v$ vs write $T_i$) is automatically fulfilled because the active transactions do not write until after $T_v$ has completed.

Forward validation of transaction $T_v$

```java
boolean valid = true;
for (int Tid = active1; Tid <= activeN; Tid++) {
    if (write set of $T_v$ intersects read set of Tid) valid = false;
}
```

• Read only transactions always pass validation
• The scheme must allow for the fact that read sets of active transactions may change during validation
• As the other transactions are still active, we may abort them or $T_v$
• if we abort $T_v$, it may be unnecessary as an active one may anyway abort
Comparison of forward and backward validation

• In conflict, choice of transaction to abort
  – forward validation allows flexibility, whereas backward validation allows only one choice (the one being validated)
• In general read sets > than write sets.
  – backward validation
    • compares a possibly large read set against the old write sets
    • overhead of storing old write sets
  – forward validation
    • checks a small write set against the read sets of active transactions
    • need to allow for new transactions starting during validation
• Starvation
  – after a transaction is aborted, the client must restart it, but there is no guarantee it will ever succeed
• In both cases, aborted transactions are not guaranteed future success
• Deadlock is less likely than starvation because locks make transactions wait
16.6 Timestamp ordering concurrency control

Each operation in a transaction is validated when it is carried out
  – if an operation cannot be validated, the transaction is aborted
  – each transaction is given a unique timestamp when it starts.
    • The timestamp defines its position in the time sequence of transactions.
  – requests from transactions can be totally ordered by their timestamps.

• Basic timestamp ordering rule (based on operation conflicts)
  – A request to write an object is valid only if that object was last read and
    written by earlier transactions.
  – A request to read an object is valid only if that object was last written by
    an earlier transaction

• This rule assumes only one version of each object

• Refine the rule to make use of the tentative versions
  – to allow concurrent access by transactions to objects
### Operation conflicts for timestamp ordering

- **Refined rule**
  - tentative versions are committed in the order of their timestamps (wait if necessary) but there is no need for the client to wait
  - but read operations wait for earlier transactions to finish
  - only wait for earlier ones (no deadlock)
  - each read or write operation is checked with the conflict rules

#### Figures

Figure 16.29

<table>
<thead>
<tr>
<th>Rule</th>
<th>$T_c$</th>
<th>$T_i$</th>
<th>Conflict Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><strong>write</strong></td>
<td><strong>read</strong></td>
<td>$T_c$ must not <strong>write</strong> an object that has been <strong>read</strong> by any $T_i$ where $T_i &gt; T_c$ this requires that $T_c \geq$ the maximum read timestamp of the object.</td>
</tr>
<tr>
<td>2.</td>
<td><strong>write</strong></td>
<td><strong>write</strong></td>
<td>$T_c$ must not <strong>write</strong> an object that has been <strong>written</strong> by any $T_i$ where $T_i &gt; T_c$ this requires that $T_c &gt;$ write timestamp of the committed object.</td>
</tr>
<tr>
<td>3.</td>
<td><strong>read</strong></td>
<td><strong>write</strong></td>
<td>$T_c$ must not <strong>read</strong> an object that has been <strong>written</strong> by any $T_i$ where $T_i &gt; T_c$ this requires that $T_c &gt;$ write timestamp of the committed object.</td>
</tr>
</tbody>
</table>
• As usual write operations are in tentative objects
• Each object has a write timestamp and a set of tentative versions
• Each with its own write timestamp and a set of read timestamps
• When a write operation is accepted it is put in a tentative version and given a write timestamp
• When a read operation is accepted it is directed to the tentative version with the maximum write timestamp less than the transaction timestamp $T_c$ is the current transaction, $T_i$ are other transactions
• $T_i > T_c$ means $T_i$ is later than $T_c$
• When a write operation is accepted it is put in a tentative version and given a write timestamp
Write operations and timestamps

- this illustrates the versions and timestamps, when we do $T_3$ write. For write to be allowed, $T_3 \geq$ maximum read timestamp (not shown)
- In cases (a), (b) and (c) $T_3 > \text{w.t.s}$ on committed version and a tentative version with w.t.s $T_3$ is inserted at an appropriate place in the list of versions
- In case (d), $T_3 < \text{w.t.s}$ on committed version and the transaction is aborted

(a) $T_3$ write

Before

After

Time

(b) $T_3$ write

Before

After

Time

(c) $T_3$ write

Before

After

Time

(d) $T_3$ write

Before

After

Time

Figure 16.30

Key:

- $T_i$: Committed
- $T_i$: Tentative

Object produced by transaction $T_i$

(with write timestamp $T_i$)

$T_1 < T_2 < T_3 < T_4$
by combining rules 1 (write/read) and 2 (write/write) we have the following rule for deciding whether to accept a write operation requested by transaction $T_c$ on object $D$
– rule 3 does not apply to writes

if ($T_c \geq$ maximum read timestamp on $D$ && $T_c >$ write timestamp on committed version of $D$)
  perform write operation on tentative version of $D$ with write timestamp $T_c$
else /* write is too late */
  Abort transaction $T_c$
Timestamp ordering read rule

• by using Rule 3 we get the following rule for deciding what to do about a read operation requested by transaction $T_c$ on object $D$. That is, whether to
  – accept it immediately,
  – wait or
  – reject it

if ($T_c >$ write timestamp on committed version of $D$) {
  let $D_{\text{selected}}$ be the version of $D$ with the maximum write timestamp $\leq T_c$
  if ($D_{\text{selected}}$ is committed)
    perform read operation on the version $D_{\text{selected}}$
  else
    Wait until the transaction that made version $D_{\text{selected}}$ commits or aborts
    then reapply the read rule
} else
  Abort transaction $T_c$
Read operations and timestamps

- Illustrates the timestamp, ordering read rule, in each case we have $T_3$ read. In each case, a version whose write timestamp is $\leq T_3$ is selected.
- In cases (a) and (b) the read operation is directed to a committed version,
  - in (a) this is the only version. In (b) there is a later tentative version.
- In case (c) the read operation is directed to a tentative version and the transaction must wait until the maker of the tentative version commits or aborts.
- In case (d) there is no suitable version and $T_3$ must abort.

![Diagram](image)

**Figure 16.31**

Key:
- Committed
- Tentative

Object produced by transaction $T_i$ (with write timestamp $T_i$)
$T_1 < T_2 < T_3 < T_4$
Transaction commits with timestamp ordering

• when a coordinator receives a commit request, it will always be able to carry it out because all operations have been checked for consistency with earlier transactions
  – committed versions of an object must be created in timestamp order
  – the server may sometimes need to wait, but the client need not wait
  – to ensure recoverability, the server will save the ‘waiting to be committed versions’ in permanent storage

• the timestamp ordering algorithm is strict because
  – the read rule delays each read operation until previous transactions that had written the object had committed or aborted
  – writing the committed versions in order ensures that the write operation is delayed until previous transactions that had written the object have committed or aborted
Remarks on timestamp ordering
concurrency control

The method avoids deadlocks, but is likely to suffer from restarts
• modification known as ‘ignore obsolete write’ rule is an improvement
  – If a write is too late it can be ignored instead of aborting the transaction, because if it had arrived in time its effects would have been overwritten anyway.
  – However, if another transaction has read the object, the transaction with the late write fails due to the read timestamp on the item
• multiversion timestamp ordering
  – allows more concurrency by keeping multiple committed versions
    • late read operations need not be aborted
  – there is not time to discuss the method now
### Figure 16.32
Timestamps in transactions $T$ and $U$

<table>
<thead>
<tr>
<th>$T$</th>
<th>$U$</th>
<th>Timestamps and versions of objects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$A$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RTS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{}</td>
</tr>
<tr>
<td>openTransaction</td>
<td>openTransaction</td>
<td>{T}</td>
</tr>
<tr>
<td>$bal = b.getBalance()$</td>
<td>$bal = b.getBalance()$</td>
<td>$S$, $T$</td>
</tr>
<tr>
<td>$b.setBalance(bal*1.1)$</td>
<td>$b.setBalance(bal*1.1)$</td>
<td>$S$, $T$</td>
</tr>
<tr>
<td>$a.withdraw(bal/10)$</td>
<td>$c.withdraw(bal/10)$</td>
<td></td>
</tr>
<tr>
<td>$commit$</td>
<td>$commit$</td>
<td></td>
</tr>
</tbody>
</table>

```java
openTransaction
bal = b.getBalance()
{
T
}
b.setBalance(bal*1.1)
bal = b.getBalance()
wait for T
S, T
a.withdraw(bal/10)

T
commit
bal = b.getBalance()
b.setBalance(bal*1.1)
c.withdraw(bal/10)

{U}
T, U
S, U
```
Late **write** operation would invalidate a **read**

**Figure 16.33**

- T1 < T2 < T3 < T4 < T5

Key:
- Object produced by transaction Ti (with write timestamp Ti and read timestamp Tk)

Diagram:
- T1 read; T2 write; T3 read; T4 write;
- T5 write; T3 read; T5 read; Time
- Committed
- Tentative
16.7 Comparison of methods for concurrency control

- **pessimistic approach** (detect conflicts as they arise)
  - timestamp ordering: serialisation order decided statically
  - locking: serialisation order decided dynamically
  - timestamp ordering is better for transactions where reads >> writes,
  - locking is better for transactions where writes >> reads
  - strategy for aborts
    - timestamp ordering – immediate
    - locking – waits but can get deadlock

- **optimistic methods**
  - all transactions proceed, but may need to abort at the end
  - efficient operations when there are few conflicts, but aborts lead to repeating work

- **the above methods are not always adequate e.g.**
  - in cooperative work there is a need for user notification
  - applications such as cooperative CAD need user involvement in conflict resolution
Summary

- Operation conflicts form a basis for the derivation of concurrency control protocols.
  - protocols ensure serializability and allow for recovery by using strict executions
  - e.g. to avoid cascading aborts
- Three alternative strategies are possible in scheduling an operation in a transaction:
  - (1) to execute it immediately, (2) to delay it, or (3) to abort it
  - strict two-phase locking uses (1) and (2), aborting in the case of deadlock
    - ordering according to when transactions access common objects
  - timestamp ordering uses all three - no deadlocks
    - ordering according to the time transactions start.
  - optimistic concurrency control allows transactions to proceed without any form of checking until they are completed.
    - Validation is carried out. Starvation can occur.