Transactions and Concurrency Control
Transactions and Concurrency Control

- A Transaction defines a sequence of server operations that is guaranteed to be atomic in the presence of multiple clients and server crash.

- All concurrency control protocols are based on serial equivalence and are derived from rules of conflicting operations.
  - Locks are used to order transactions that access the same object according to request order.
  - Optimistic concurrency control allows transactions to proceed until they are ready to commit, whereupon a check is made to see any conflicting operation on objects.
  - Timestamp ordering uses timestamps to order transactions that access the same object according to their starting time.

For more study materials: WWW.KTUSTUDENTS.IN
Banking Example

- Each account is represented by a remote object whose interface `Account` provides operations for making deposits and withdrawals and for enquiring about and setting the balance.

- Each branch of the bank is represented by a remote object whose interface `Branch` provides operations for creating a new account, for looking up an account by name and for enquiring about the total funds at that branch.

- **Main issue**: unless a server is carefully designed, its operations performed on behalf of different clients may sometimes interfere with one another. Such interference may result in incorrect values in the object.
### Operations of the Account interface

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deposit(amount)</td>
<td>deposit amount in the account</td>
</tr>
<tr>
<td>withdraw(amount)</td>
<td>withdraw amount from the account</td>
</tr>
<tr>
<td>getBalance()</td>
<td>return the balance of the account</td>
</tr>
<tr>
<td>setBalance(amount)</td>
<td>set the balance of the account to amount</td>
</tr>
</tbody>
</table>

### Operations of the Branch interface

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create(name) -&gt; account</td>
<td>create a new account with a given name</td>
</tr>
<tr>
<td>lookUp(name) -&gt; account</td>
<td>return a reference to the account with the given name</td>
</tr>
<tr>
<td>branchTotal() -&gt; amount</td>
<td>return the total of all the balances at the branch</td>
</tr>
</tbody>
</table>
Simple Synchronization without Transactions

- The use of multiple threads is beneficial to the performance.
- Multiple threads may access the same objects.
  - For example, deposit and withdraw methods
- Synchronized keyword can be applied to method in Java, so only one thread at a time can access an object.

  (If one thread invokes a synchronized method on an object, then that object is locked, another thread that invokes one of the synchronized method will be blocked.)
Enhancing Client Cooperation by \textit{synchronization of server operations}

- We have seen how clients may use a server as a means of sharing some resources.
  - E.g. some clients update the server's objects and other clients access them.
- In some applications, threads need to communicate and coordinate their actions.
- Producer and Consumer problem.
  - Wait and Notify actions.
What is a Transaction?

- Transaction - originally from database management systems.

- Clients require a sequence of separate requests to a server to be atomic in the sense that:
  - Other concurrent clients should not interfere; and
  - Either all of the operations must be completed successfully or they must have no effect at all in the presence of server crashes.
Atomicity

- All or nothing: a transaction either completes successfully, and effects of all of its operations are recorded in the object, or it has no effect at all.
  - Failure atomicity: effects are atomic even when server crashes
  - Durability: after a transaction has completed successfully, all its effects are saved in permanent storage for recover later.

- Isolation: each transaction must be performed without interference from other transactions. The intermediate effects of a transaction must not be visible to other transactions.
A client’s banking transaction

Transaction T:
   a. withdraw(100);
   b. deposit(100);
Operations in *Coordinator* interface

`openTransaction() -> trans;`

starts a new transaction and delivers a unique TID `trans`. This identifier will be used in the other operations in the transaction.

`closeTransaction(trans) -> (commit, abort);`

ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

`abortTransaction(trans);`

aborts the transaction.
If a transaction aborts for any reason (self abort or server abort), it must be guaranteed that future transaction will not see its effect either in the object or in their copies in permanent storage.

For more study materials: WWW.KTUSTUDENTS.IN
Concurrency Control

- Two well-known problems of concurrent transactions in the context of the banking example –
  - The ‘lost update’ problem and
  - The ‘inconsistent retrievals’ problem.
Concurrency Control: the lost update problem

Transaction $T$: 

\[
\begin{align*}
\text{balance} &= b.\text{getBalance}(); \\
b.\text{setBalance}(\text{balance} \times 1.1); \\
a.\text{withdraw}(\text{balance}/10)
\end{align*}
\]

Transaction $U$: 

\[
\begin{align*}
\text{balance} &= b.\text{getBalance}(); \\
b.\text{setBalance}(\text{balance} \times 1.1); \\
c.\text{withdraw}(\text{balance}/10)
\end{align*}
\]

\[
\begin{align*}
\text{balance} &= \ 200 \\
b.\text{setBalance}(\text{balance} \times 1.1); \\
a.\text{withdraw}(\text{balance}/10)
\end{align*}
\]

\[
\begin{align*}
\text{balance} &= \ 220 \\
c.\text{withdraw}(\text{balance}/10)
\end{align*}
\]

a, b and c initially have bank account balance are: 100, 200, and 300. T transfers an amount from a to b. U transfers an amount from c to b.
Concurrence Control: The \textit{inconsistent retrievals} problem

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

\[a.\text{withdraw}(100); \quad $100\]

\[a.\text{withdraw}(100); \quad $100\]

\[total = a.\text{getBalance}() \quad $100\]

\[total = total + b.\text{getBalance}() \quad $300\]

\[total = total + c.\text{getBalance}()\]

\[b.\text{deposit}(100) \quad $300\]

\[\therefore\]

a, b accounts start with 200 both.
Serial equivalence

- If these transactions are done one at a time in some order, then the final result will be correct.
- If we do not want to sacrifice the concurrency, an interleaving of the operations of transactions may lead to the same effect as if the transactions had been performed one at a time in some order.
- We say it is a **serially equivalent interleaving**.
- The use of serial equivalence is a criterion for correct concurrent execution to prevent lost updates and inconsistent retrievals.
A serially equivalent interleaving of $T$ and $U$

<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>

<table>
<thead>
<tr>
<th>Action</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$balance = b.getBalance()$</td>
<td>$$200$</td>
</tr>
<tr>
<td>$b.setBalance(balance*1.1)$</td>
<td>$$220$</td>
</tr>
<tr>
<td>$a.withdraw(balance/10)$</td>
<td>$$80$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$balance = b.getBalance()$</td>
<td>$$220$</td>
</tr>
<tr>
<td>$b.setBalance(balance*1.1)$</td>
<td>$$242$</td>
</tr>
<tr>
<td>$c.withdraw(balance/10)$</td>
<td>$$278$</td>
</tr>
</tbody>
</table>
Conflicting Operations

- When we say a pair of operations conflicts we mean that their combined effect depends on the order in which they are executed. eg: read and write

- Three ways to ensure serializability:
  - Locking
  - Timestamp ordering
  - Optimistic concurrency control

<table>
<thead>
<tr>
<th></th>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td></td>
<td>write($A$)</td>
<td>write($A$)</td>
</tr>
<tr>
<td></td>
<td>read($B$)</td>
<td>read($B$)</td>
</tr>
<tr>
<td></td>
<td>write($B$)</td>
<td>write($B$)</td>
</tr>
</tbody>
</table>

For more study materials: WWW.KTUSTUDENTS.IN
**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 <em>read</em> 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 <em>read</em> and a <em>write</em> 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 <em>write</em> operations depends on the order of their execution</td>
</tr>
</tbody>
</table>

For more study materials: WWW.KTUSTUDENTS.IN
Recoverability from aborts

- Servers must record the effect of all committed transactions and none of the effects of the aborted transactions.

- Two problems associated with aborting transactions that may occur in the presence of serially equivalent execution of transactions:
  - Dirty reads
  - Premature writes
A dirty read when transaction $T$ aborts

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$.getBalance()</td>
<td>$a$.getBalance()</td>
</tr>
<tr>
<td>$a$.setBalance(balance + 10)</td>
<td>$a$.setBalance(balance + 20)</td>
</tr>
</tbody>
</table>

$balance = a$.getBalance() $100$

$balance = a$.getBalance() $110$

$commit$ transaction

$abort$ transaction

Dirty reads caused by a read in one transaction $U$ and an earlier unsuccessful write in another transaction $T$ on the same object. $T$ will be rolled back and restore the original $a$ value, thus $U$ will have seen a value that never existed. $U$ is committed, so cannot be undone. $U$ performs a dirty read.

For more study materials: WWW.KTUSTUDENTS.IN
Premature Write: 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>$100</td>
<td>$110</td>
</tr>
<tr>
<td>a.setBalance(105)</td>
<td>$105</td>
</tr>
<tr>
<td>a.setBalance(110)</td>
<td>$110</td>
</tr>
</tbody>
</table>

Premature write: related to the interaction between write operations on the same object belonging to different transactions.

a. If U aborts and then T commit, we got a to be correct 105.
Some systems restore value to “Before images” value for abort action, namely the value before all the writes of a transaction. a is 100, which is the before image of T’s write. 105 is the before image of U’s write.

b. Consider if U commits and then T aborts, we got wrong value of 100.

c. Similarly if T aborts then U aborts, we got 105, which is wrong and should be 100.
So to ensure correctness, write operations must be delayed until earlier transactions that updated the same object have either committed or aborted.
Strict executions of transactions

- Generally, it is required that transactions delay both their read and write operations so as to avoid both ‘dirty reads’ and ‘premature writes’.
- The executions of transactions are called strict if the service delays both read and write operations on an object until all transactions that previously wrote that object have either committed or aborted.
Tentative versions

- For a server of recoverable objects to participate in transactions, it must be designed so that any updates of objects can be removed if and when a transaction aborts.
- All of the update operations performed during a transaction are done in tentative versions of objects in volatile memory.
- The tentative versions are transferred to the objects only when a transaction commits, by which time they will also have been recorded in permanent storage.
- This is performed in a single step, during which other transactions are excluded from access to the objects that are being altered.
Nested transactions

- Nested transaction extend the transaction model by allowing transactions to be composed of other transactions.
- The outermost transaction in a set of nested transactions is called the top-level transaction.
- Transactions other than the top-level transaction are called subtransactions.
Nested transactions

For more study materials: WWW.KTUSTUDENTS.IN
Nested transactions

The rules for committing of nested transactions are:

- A transaction may commit or abort only after its child transactions have completed.
- When a subtransaction completes, it makes an independent decision either 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, the 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.

For more study materials: WWW.KTUSTUDENTS.IN
A simple example of a serializing mechanism is the use of exclusive locks.

Server can lock any object that is about to be used by a client.

If another client wants to access the same object, it has to wait until the object is unlocked in the end.
Transactions $T$ and $U$ with exclusive locks

<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><code>openTransaction</code></td>
<td><code>lock B</code></td>
<td><code>openTransaction</code></td>
<td><code>waits for $T$'s lock on $B$</code></td>
</tr>
<tr>
<td>$bal = b.getBalance()$</td>
<td><code>Lock A</code></td>
<td>$bal = b.getBalance()$</td>
<td><code>lock $B$</code></td>
</tr>
<tr>
<td>$b.setBalance(bal*1.1)$</td>
<td><code>unlock A, B</code></td>
<td>$b.setBalance(bal*1.1)$</td>
<td><code>lock $C$</code></td>
</tr>
<tr>
<td>$a.withdraw(bal/10)$</td>
<td></td>
<td>$c.withdraw(bal/10)$</td>
<td></td>
</tr>
<tr>
<td><code>closeTransaction</code></td>
<td><code>unlock A, B</code></td>
<td><code>closeTransaction</code></td>
<td><code>unlock B, C</code></td>
</tr>
</tbody>
</table>

For more study materials: [WWW.KTUSTUDENTS.IN](http://WWW.KTUSTUDENTS.IN)
Two-phase locking

- Serial equivalence requires that all of a transaction’s accesses to a particular object be serialized with respect to accesses by other transactions. All pair of conflicting operations of two transactions should be executed in the same order.
- To ensure this, a transaction is not allowed any new locks after it has released a lock.

**Two-phase locking**
- The first phase of each transaction is a ‘growing phase’, during which new locks are acquired.
- In the second phase, the locks are released (a ‘shrinking phase’).

**Strict two-phase locking**
- Any locks applied during the progress of a transaction are held until the transaction commits or aborts.

For more study materials: WWW.KTUSTUDENTS.IN
It is preferable to adopt a locking scheme that controls the access to each object so that there can be several concurrent transactions reading an object, or a single transaction writing an object, but not both – commonly referred to as a ‘many readers/single writer’ scheme.

Two types of locks are used: **read locks and write locks**

- Before a transaction’s read operation is performed, a read lock should be set on the object.
- Before a transaction’s write operation is performed, a write lock should be set on the object.
- Whenever it is **impossible** to set a lock immediately, the transaction must wait until it is possible to do so.
- As pair of read operations from different transactions do not conflict, an attempt to set a read lock on an object with a read lock is always successful. Therefore, **read locks are sometimes called shared lock**.

For more study materials: WWW.KTUSTUDENTS.IN
Lock compatibility

An object can be read and write. From the compatibility table, we know pairs of read operations from different transactions do not conflict. So a simple exclusive lock used for both read and write reduces concurrency more than necessary.

Rules:
1. If T has already performed a read operation, then a concurrent transaction U must not write until T commits or aborts.
2. If T already performed a write operation, then concurrent U must not read or write until T commits or aborts.
Strict two-phase Locking Protocol

- Because transaction may abort, strict execution are needed to prevent dirty reads and premature writes, which are caused by read or write to same object accessed by another earlier unsuccessful transaction that already performed an write operation.
- So to prevent this problem, a transaction that needs to read or write an object must be delayed until other transactions that wrote the same object have committed or aborted.
- Rule:
  - Any **locks** applied during the progress of a transaction are **held until** the transaction **commits** or **aborts**.
Strict two-phase Locking Protocol

$$D = \begin{bmatrix}
T1 & T2 \\
S(A) & S(A) \\
R(A) & R(A) \\
   & X(B) \\
   & R(B) \\
   & W(B) \\
   & Commit \\
   & X(C) \\
   & R(C) \\
   & W(C) \\
   & Commit
\end{bmatrix}$$
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.

A transaction with a read lock that is shared by other transactions cannot promote its read lock to a write lock, because write lock will conflict with other read locks.

For more study materials: WWW.KTUSTUDENTS.IN
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*/)
            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();
}
Locking rules for nested transactions

- every lock that is acquired by a successful subtransaction is inherited by its parent when it completes
- Parent transactions are not allowed to run concurrently with their child transactions
### Deadlock with write locks

<table>
<thead>
<tr>
<th>Transaction</th>
<th>$T$</th>
<th>Transaction</th>
<th>$U$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operations</strong></td>
<td><strong>Locks</strong></td>
<td><strong>Operations</strong></td>
<td><strong>Locks</strong></td>
</tr>
<tr>
<td>$a.deposit(100)$;</td>
<td>write lock $A$</td>
<td>$b.deposit(200)$</td>
<td>write lock $B$</td>
</tr>
<tr>
<td>$b.withdraw(100)$</td>
<td>waits for $U$'s lock on $B$</td>
<td>$a.withdraw(200)$;</td>
<td>waits for $T$'s lock on $A$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>For more study materials: <a href="http://WWW.KTUSTUDENTS.IN">WWW.KTUSTUDENTS.IN</a></td>
<td>For more study materials: <a href="http://WWW.KTUSTUDENTS.IN">WWW.KTUSTUDENTS.IN</a></td>
<td>For more study materials: <a href="http://WWW.KTUSTUDENTS.IN">WWW.KTUSTUDENTS.IN</a></td>
<td>For more study materials: <a href="http://WWW.KTUSTUDENTS.IN">WWW.KTUSTUDENTS.IN</a></td>
</tr>
</tbody>
</table>
The wait-for graph for Figure 13.19

For more study materials: WWW.KTUSTUDENTS.IN
Figure 13.21
A cycle in a wait-for graph
T and W then request write locks on object C and a deadlock arises. V is involved in two cycles.
Deadlock Prevention

Deadlock prevention:

- Simple way is to lock all of the objects used by a transaction when it starts. It should be done as an atomic action to prevent deadlock. a. inefficient, say lock an object you only need for short period of time. b. Hard to predict what objects a transaction will require.

- Judge if system can remain in a Safe state by satisfying a certain resource request. Banker’s algorithm.

- Order the objects in certain order. Acquiring the locks need to follow this certain order.
Safe State

System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes such that for each $P_i$, the resources that $P_i$ can still request can be satisfied by currently available resources + resources held by all the $P_j$, with $j < i$.

If a system is in safe state $\Rightarrow$ no deadlocks.
If a system is in unsafe state $\Rightarrow$ possibility of deadlock.
Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state

➢ Banker’s Algorithm
Deadlock Detection

- Deadlock may be detected by finding cycles in the wait-for-graph. Having detected a deadlock, a transaction must be selected for abortion to break the cycle.
  - If lock manager blocks a request, an edge can be added. Cycle should be checked each time a new edge is added.
  - One transaction will be selected to abort in case of cycle. Age of transaction and number of cycles involved when selecting a victim

- **Timeouts** is commonly used to resolve deadlock. Each lock is given a limited period in which it is invulnerable. After this time, a lock becomes vulnerable.
  - If no other transaction is competing for the object, vulnerable object remained locked. However, if another transaction is waiting, the lock is broken.

Disadvantages:

- Transaction aborted simply due to timeout and waiting transaction even if there is no deadlock. (may add deadlock detection)
- Hard to set the timeout time
Resolution of the deadlock – Time-outs

<table>
<thead>
<tr>
<th>Transaction T</th>
<th>Transaction U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations</td>
<td>Operations</td>
</tr>
<tr>
<td>a.deposit(100);</td>
<td>b.deposit(200)</td>
</tr>
<tr>
<td>b.withdraw(100)</td>
<td></td>
</tr>
<tr>
<td>• • •</td>
<td></td>
</tr>
<tr>
<td>T’s lock on A becomes vulnerable, unlock A, abort T</td>
<td>a.withdraw(200);</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a.withdraw(200);</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Kung and Robinson [1981] identified a number of inherent disadvantages of locking and proposed an alternative optimistic approach to the serialization of transactions that avoids these drawbacks. Disadvantages of lock-based:

- Lock maintenance represents an overhead that is not present in systems that do not support concurrent access to shared data. Locking sometimes are only needed for some cases with low probabilities.
- The use of lock can result in deadlock. Deadlock prevention reduces concurrency severely. The use of timeout and deadlock detection is not ideal for interactive programs.
- To avoid cascading aborts, locks cannot be released until the end of the transaction. This may reduce the potential for concurrency.
Optimistic Concurrency Control

- It is based on observation that, in most applications, the likelihood of two clients’ transactions accessing the same object is low. Transactions are allowed to proceed as though there were no possibility of conflict with other transactions until the client completes its task and issues a closeTransaction request.

- When conflict arises, some transaction is generally aborted and will need to be restarted by the client.
Optimistic Concurrency Control

Each transaction has the following phases:

- **Working phase**: Each transaction has a tentative version of each of the objects that it updates. This is a copy of the most recently committed version of the object. The tentative version allows the transaction to abort with no effect on the object, either during the working phase or if it fails validation due to other conflicting transaction. Several different tentative values of the same object may coexist. In addition, two records are kept of the objects accessed within a transaction, a read set and a write set containing all objects either read or written by this transaction. Read are performed on committed version (no dirty read can occur) and write record the new values of the object as tentative values which are invisible to other transactions.
Optimistic Concurrency Control

- **Validation phase:** When `closeTransaction` request is received, the transaction is validated to establish whether or not its operations on objects conflict with operations of other transaction on the same objects. If successful, then the transaction can commit. If fails, then either the current transaction or those with which it conflicts will need to be aborted.

- **Update phase:** If a transaction is validated, all of the changes recorded in its tentative versions are made permanent. Read-only transaction can commit immediately after passing validation. Write transactions are ready to commit once the tentative versions have been recorded in permanent storage.
Validation of Transactions

- Validation uses the read-write conflict rules to ensure that the scheduling of a particular transaction is serially equivalent with respect to all other overlapping transactions—that is, any transactions that had not yet committed at the time this transaction started. Each transaction is assigned a number when it enters the validation phase (when the client issues a `closeTransaction`). Such number defines its position in time. A transaction always finishes its working phase after all transactions with lower numbers. That is, a transaction with the number Ti always precedes a transaction with number Tj if i < j.

For more study materials: WWW.KTUSTUDENTS.IN
Serializability of transaction $T$ with respect to transaction $T_i$

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

The validation test on transaction $Tv$ is based on conflicts between operations in pairs of transaction $Ti$ and $Tv$, for a transaction $Tv$ to be serializable with respect to an overlapping transaction $Ti$, their operations must conform to the above rules.
Figure 13.28
Validation of transactions

Earlier committed transactions

Working Validation Update

Transaction being validated

Later active transactions

active_1

active_2

For more study materials: WWW.KTUSTUDENTS.IN
Validation

- Backward Validation: checks the transaction undergoing validation with other preceding overlapping transactions - those that entered the validation phase before it.

- Forward Validation: checks the transaction undergoing validation with other later transactions, which are still active.
Validation of Transactions

Backward validation of transaction $T_v$

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;
}

Forward validation of transaction $T_v$

boolean valid = true;
for (int $T_{id} = active1; T_{id} <= activeN; T_{id}++){
    if (write set of $T_v$ intersects read set of $T_{id}$) valid = false;
}
Comparison of methods for Concurrency Control

- The timestamp ordering method is similar to two-phase locking in that both use pessimistic approaches in which conflicts between transactions are detected as each object is accessed. On the one hand, timestamp ordering decides the serialization order statically – when a transaction starts. On the other hand, two-phase locking decides the serialization order dynamically – according to the order in which objects are accessed. Timestamp ordering, and in particular multiversion timestamp ordering, is better than strict two-phase locking for read-only transactions. Two-phase locking is better when the operations in transactions are predominantly updates.

- The pessimistic methods differ in the strategy used when a conflicting access to an object is detected. Timestamp ordering aborts the transaction immediately, whereas locking makes the transaction wait – but with a possible later penalty of aborting to avoid deadlock.

For more study materials: WWW.KTUSTUDENTS.IN
When optimistic concurrency control is used, all transactions are allowed to proceed, but some are aborted when they attempt to commit, or in forward validation transactions are aborted earlier. This results in relatively efficient operation when there are few conflicts, but a substantial amount of work may have to be repeated when a transaction is aborted.

Historically, the predominant method of concurrency control of access to data in distributed systems is by locking – for example, as mentioned earlier, the CORBA Concurrency Control Service is based entirely on the use of locks.
The above concurrency control mechanisms are not always adequate for twenty-first-century applications that enable users to share documents over the Internet. Many of the latter use optimistic forms of concurrency control followed by conflict resolution instead of aborting one of any pair of conflicting operations. The following are some examples.

- **Dropbox**: Dropbox [www.dropbox.com] is a cloud service that provides file backup and enables users to share files and folders, accessing them from anywhere. Dropbox uses an optimistic form of concurrency control, keeping track of consistency and preventing clashes between users’ updates – which are at the granularity of whole files. Thus if two users make concurrent updates to the same file, the first write will be accepted and the second rejected. However, Dropbox provides a version history to enable users to merge their updates manually or restore previous versions.

For more study materials: WWW.KTUSTUDENTS.IN
Google apps: Google Apps include Google Docs, a cloud service that provides web-based applications (word processor, spreadsheet, and presentation) that allow users to collaborate with one another by means of shared documents. If several people edit the same document simultaneously, they will see each other’s changes. In the case of a word processor document, users can see one another’s cursors and updates are shown at the level of individual characters as they are typed by any participant. Users are left to resolve any conflicts that occur, but conflicts are generally avoided because users are continuously aware of each other’s activities. In the case of a spreadsheet document, users’ cursors and changes are displayed and updated at the granularity of single cells. If two users access the same cell simultaneously, the last update wins.

Wikipedia: Concurrency control for editing is optimistic, allowing editors concurrent access to web pages in which the first write is accepted and a user making a subsequent write is shown an ‘edit conflict’ screen and asked to resolve the conflicts.