



One of the important transactions is that their effect on shared data is serially equivalent. This means that any data that is touched by a set of transactions must be in such a state that the results could have been obtained if all the transactions executed serially (one after another) in some order (it does not matter which). What is invalid, is for the data to be in some form that cannot be the result of serial execution (e.g. two transactions modifying data concurrently).

One easy way of achieving this guarantee is to ensure that only one transaction executes at a time. We can accomplish this by using mutual exclusion and having a "transaction" resource that each transaction must have access to.

However, this is usually overkill and does not allow us to take advantage of the concurrency that we may get in distributed systems (for instance, it is obviously overkill if two transactions don't even access the same data).

What we would like to do is allow multiple transactions to execute simultaneously *but* keep them out of each other's way and ensure serializability.

This is called **concurrency control**.



We can use exclusive locks on a resource to serialize execution of transactions that share resources. A transaction locks an object that it is about to use. If another transaction requests the same object and it is locked, the transaction must wait until the object is unlocked.

To implement this in a distributed system, we rely on a **lock manager** - a server that issues locks on resources. This is exactly the same as a centralized mutual exclusion server: a client can request a lock and then send a message releasing a lock on a resource (by resource in this context, we mean some specific block of data that may be read or written).

One thing to watch out for, is that we still need to preserve serial execution: if two transactions are accessing the same set of objects, the results must be the same as if the transactions executed in some order (transaction A cannot modify some data while transaction B modifies some other data and then transaction A accesses that modified data -- this is concurrent modification).

To ensure serial ordering on resource access, we impose a restriction that states that <u>a transaction is not allowed to get any new locks after it has released a lock</u>. This is known as **two-phase locking**. The first phase of the transaction is a *growing phase* in which it acquires the locks it needs. The second phase is the *shrinking phase* where locks are released.



A problem with two-phase locking is that if a transaction aborts, some other transaction may have already used data from an object that the aborted transaction modified and then unlocked. If this happens, any such transactions will also have to be aborted. This situation is known as **cascading aborts**.

To avoid this, we can strengthen our locking by requiring that a transaction will hold all its locks to the very end: until it commits or aborts rather than releasing the lock when the object is no longer needed.

This is known as strict two-phase locking.



A typical system will have many objects and typically a transaction will access only a small amount of data at any given time (and it will frequently be the case that a transaction will not clash with other transactions).

The granularity of locking affects the amount of concurrency we can achieve. If we can have a smaller granularity (lock smaller objects or pieces of objects) then we can generally achieve higher concurrency.

For example, suppose that all of a bank's customers are locked for any transaction that needs to modify a single customer datum: concurrency is severely limited because any other transactions that need to access *any* customer data will be blocked. If, however, we use a customer record as the granularity of locking, transactions that access different customer records will be capable of running concurrently.



Paul Krzyzanowski

CS 417

6

There is no harm having multiple transactions read from the same object as long as it has not been modified by any of the transactions. This way we can increase concurrency by having multiple transactions run concurrently if they are only reading from an object.

However, only one transaction should be allowed to write to an object.

Once a transaction has modified an object, no other transactions should be allowed to read or write the modified object.

To support this, we now use two locks: *read locks* and *write locks*. Read locks are also known as *shared locks* (since they can be shared by multiple transactions)

If a transaction needs to read an object, it will request a read lock from the lock manager.

If a transaction needs to modify an object, it will request a write lock from the lock manager.

If the lock manager cannot grant a lock, then the transaction will wait until it can get the lock (after the transaction with the lock committed or aborted).

To summarize lock granting:

If a transaction has:	another transaction may obtain:
no locks	read lock or write lock
read lock	read lock (wait for write lock)
write lock	wait for read or write locks



Two-version locking is an optimistic concurrency control scheme that allows one transaction to write tentative versions of objects while other transactions read from committed versions of the same objects.

Read operations only wait if another transaction is currently committing the same object.

This scheme allows more concurrency than read-write locks, *but* writing transactions risk waiting (or rejection) when they attempt to commit.

Transactions cannot commit their *write* operations immediately if other uncommitted transactions have read the same objects. Transactions that request to commit in this situation have to wait until the reading transactions have completed.



The two-version locking scheme requires three types of locks: read, write, and commit locks. Before an object is read, a transaction must obtain a *read lock*. Before an object is written, the transaction must obtain a *write lock* (same as with two-phase locking). Neither of these locks will be granted if there is a *commit lock* on the object.

When the transaction is ready to commit:

- all of the transaction's write locks are changed to commit locks

- if any objects used by the transaction have outstanding read locks, the transaction must wait until the transactions that set these locks have completed and the locks are released.

If we compare the performance difference between two-version locking and strict two-phase locking (read/write locks):

- read operations in two-version locking are delayed only while transactions are being committed rather than during the entire execution of transactions (usually the commit protocol takes far less time than the time to perform the transaction)

- but... read operations of one transaction can cause a delay in the committing of other transactions.



Locks are not without drawbacks

Locks have an overhead associated with them: a lock manager is needed to keep track of locks - there is overhead in requesting them. Even read-only operations must still request locks.

The use of locks can result in deadlock. We need to have software in place to detect or avoid deadlock.

Locks can decrease the potential concurrency in a system by having a transaction hold locks for the duration of the transaction (until a commit or abort).



King and Robinson (1981) proposed an alternative technique for achieving concurrency control, called **optimistic concurrency control**.

This is based on the observation that, in most applications, the chance of two transactions accessing the same object is low.

We will allow transactions to proceed as if there were no possibility of conflict with other transactions: a transaction does not have to obtain or check for locks.

This is the **working phase**. Each transaction has a tentative version (private workspace) of the objects it updates - copy of the most recently committed version. Write operations record new values as tentative values.

Before a transaction can commit, a validation is performed on all the data items to see whether the data conflicts with operations of other transactions. This is the **validation phase**.

If the validation fails, then the transaction will have to be aborted and restarted later.

If the transaction succeeds, then the changes in the tentative version are made permanent. This is the **update phase**.

Optimistic control is deadlock free and allows for maximum parallelism (at the expense of possibly restarting transactions)



Another approach to concurrency control was presented by Reed in 1983. This is called **timestamp ordering**.

Each transaction is assigned a unique timestamp when it begins (can be from a physical or logical clock).

Each object in the system has a *read* and *write* timestamp associated with it (two timestamps per object). The *read* timestamp is the timestamp of the last committed transaction that read the object. The *write* timestamp is the timestamp of the last committed transaction that modified the object (note - the timestamps are obtained from the transaction timestamp - the start of that transaction)

The rule of timestamp ordering is:

- if a transaction wants to write an object, it compares its own timestamp with the object's read and write timestamps. If the object's timestamps are older, then the ordering is good.

- if a transaction wants to read an object, it compares its own timestamp with the object's write timestamp. If the object's write timestamp is older than the current transaction, then the ordering is good.

If a transaction attempts to access an object and does not detect proper ordering, the transaction is aborted and restarted (improper ordering means that a newer transaction came in and modified data before the older one could access the data or read data that the older one wants to modify).