Transactions and Commits

- Transactions. The ACID properties
- Concurrency between transactions
- Recovery
- Control of concurrency by
  - Two phase locking
  - Timestamping
  - Optimistic Concurrency Control
- Deadlock prevention, detection and recovery
- Distributed Transactions
  - Concurrency Control
  - Distributed Commitment

Transactions

A **Transaction** is a sequence of actions intended to be executed atomically.

We bracket the transaction with start and either commit or abort.

For example:

```
start ;
Amount bal = balance( accountA )
A.debit( bal ) ;
B.credit( bal )
if( B.checkBal( 1000 ) ) {
        B.debit( 1000 ) ;
        C.credit( 1000 ) ;
        commit ; }
else abort ;
```

ACID properties.

- Atomicity: The transaction is either completely done, or none of it is done. If the transaction aborts itself or is aborted because of a failure, then it is as if the transaction had never started.
  - In the example, if abort is reached, the money transferred from A to B must be transferred back.
- Consistency: The transaction takes the system from one consistent state to another.
  - Example. In a banking system, the books should balance after each transaction.
- Isolation: The intermediate results of the transaction are not revealed to other transaction.
  - In example above, no other transaction sees the state in which money has been withdrawn from A but is not yet deposited in B.
- Durability. The once the transaction is committed, subsequent failures can be recovered.

Concurrency

Consider two transactions: T is a transfer

```
start
co
        A.debit(1000)
//
        B.credit( 1000 )
oc
commit
```

S calculates the total in two accounts.

```
start
co
        x := A.read()
//
        y := B.read()
oc
Statement.print( x+y )
commit
```

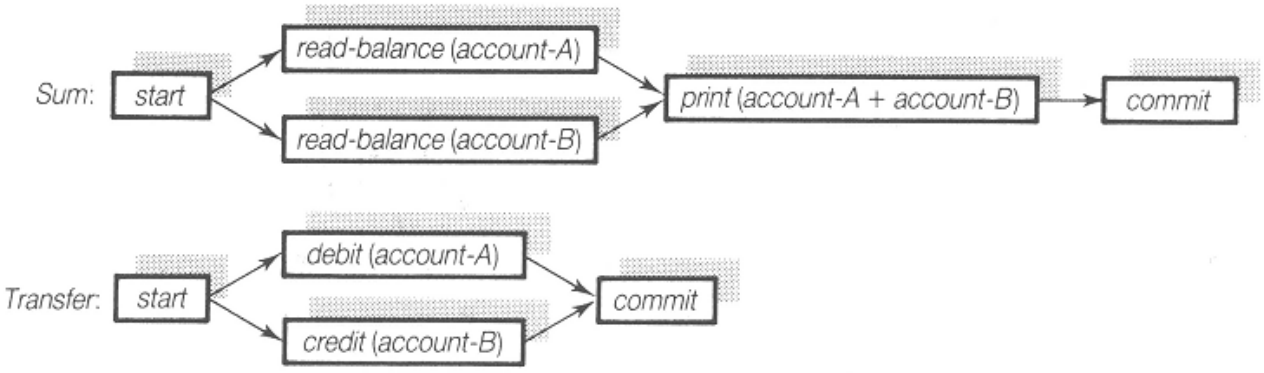
For efficiency we might want to run the transactions concurrently. There is concurrency available both within and between the transactions.

The effect of running a set of transactions should be the same as if they were executed sequentially in some order.

In the example, the effect should be S;T or T;S

Each transaction is a directed acyclic graph where

- Nodes are atomic actions on objects
- Edges express ordering constraints



Clearly these two transactions can interfere with each other, so we must be careful executing them concurrently:

Interleaving 0	Interleaving 1
S : x := A.read()	T: A.debit(1000)
T: A.debit(1000)	S: x := A.read()
T: B.credit(1000)	S: y := B.read()
S: y := B.read()	T: B.credit(1000)
S : Statement.print(x+y)	S : Statement.print(x+y)

Neither of these execution sequences agrees with either serial order.

Two operations a and b on the same object are conflicting if the don't commute

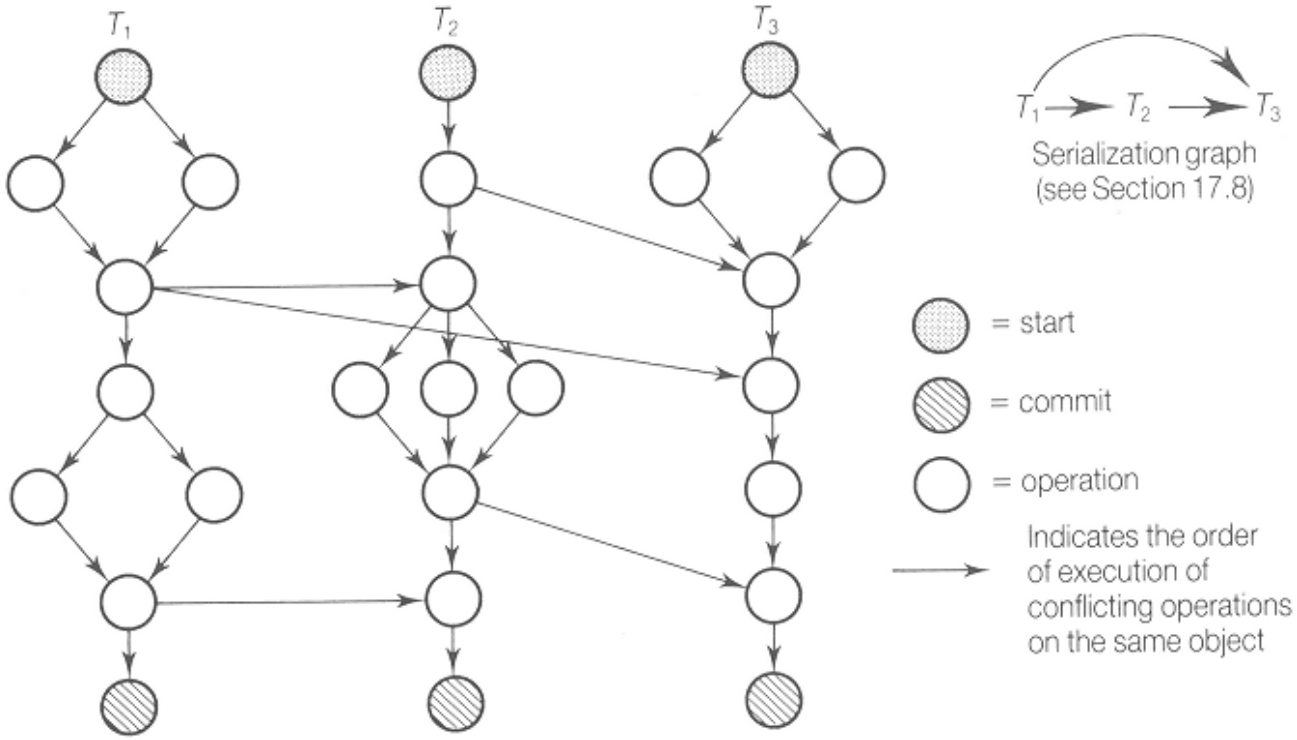
$$o.a(); o.b() \neq o.b(); o.a()$$

Given a set of transactions we add edges between conflicting operations from different transactions such that there is a total order on transactions

$$T_0, T_1, ..., T_{n-1}$$

and edges between transactions respect this order

$$T_i : o.a() \rightarrow T_j : o.b() \Rightarrow i < j$$



The operations in the graph can be executed concurrently in any order so long as the arrows are respected.

Dealing with aborts by write-ahead

A transaction may "abort" because of

- self-abort: transaction decides it has failed.
- failure of server
- aborted to resolve deadlock
- aborted because of cascading aborts

In the last three cases, the transaction should be restarted later.

Undo: To implement aborts, each operation must be capable of being undone

$$o.a(); o.undo(a) = skip$$

($skip$ is the identity operations.)

Redo: In addition operations can be redone.

$$o.a() = o.a(); o.undo(a); o.redo(a)$$

Undo and Redo must be "idempotent" so

$$o.undo(a); o.undo(a) \;=\; o.undo(a)$$
$$o.a(); o.redo(a) \;=\; o.a()$$

Write-ahead logs

To facilitate undo and redo we keep a “write-ahead log” in persistent store.

Records in “write-ahead logs” contain

- Transaction identifier
- Object identifier
- Operation within transaction
- Enough information to undo the operation (e.g. part of original state)
- Enough information to redo the operation (e.g. part of final state)

Also logged are: start, commit, and abort operations.

To execute an operation:

- Read the object state
- Calculate the new state
- Write to the write-ahead log.
- Write to the object.

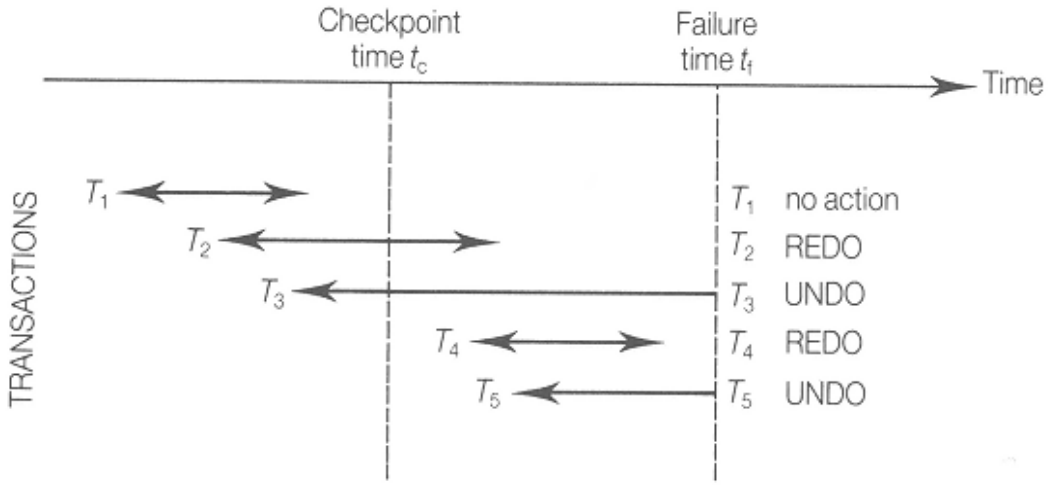
It is important that the write to the log goes to nonvolitile memory (e.g. disk).

Checkpointing

Periodically

- Write a checkpoint record to the log including a list of all active transactions.
- Force all object data to disk.

Recovery



After crash. Object state is recovered from disk

- Transactions committed before the last checkpoint.
  - No action required.
- Transaction committed since last checkpoint
  - Transaction is redone (from last checkpoint)
- Transactions not committed
  - Transaction is undone (to last checkpoint)
  - Transaction is restarted

Cascading Aborts

If a transaction t is aborted, but transaction s has used an object that was modified by t.

- Then s must be aborted too and restarted later.

Concurrency Control

How do we keep concurrent transactions from conflicting.

Concurrency control via locking

Locking is not sufficient.

```
S : Lock(A) ;
x := A.read() ;
Unlock(A) ;
                        T: Lock( A) ;
                        A.debit(1000) ;
                        Unlock(A) ;
                        Lock( B) ;
                        B.credit(1000) ;
                        Unlock( B)
Lock( B) ;
y := B.read() ;
Unlock(B) ;
S: Lock( Stmt ) ;
Stmt.print( x+y ) ;
Unlock( Stmt ) ;
```

Two phase-locking (2PL):

- Locking phase: Transactions acquire locks on objects as they need them.
- Unlocking phase: No lock is released until all locks needed have been acquired

Result: Transactions can not interfere since conflicting pairs are scheduled in the same order

Example:

```
S : Lock(A) ; x := A.read() ;
Lock( B) ; y := B.read() ;        T: Lock( A) ...
Lock( Stmt ) ;                    T delays
Unlock(A) ;
                                  ... ; A.debit(1000) ;
                                  Lock( B) ...
Unlock(B) ;                       T delays
Stmt.print( x+y ) ;               ... ; Unlock( A) ;
Unlock( Stmt ) ;
                                  B.credit(1000) ; Unlock( B) ;
```

Strict two-phase locking Locks are only released as part of commit or abort.

Concurrency control by time stamp ordering (TSO)

Each transaction is given a time stamp $ts(t)$

Each object is stamped with the $ts(t)$ of the last transaction to operate on it.

When t executes an operation on object o

---

lock(o);
if($ts(o) > ts(t)$) { unlock(o); abort(t); }
else { *do operation* ; $ts(o) := ts(t)$; unlock(o); }

Example

```
S: start ;
S : x := A.read() ;      T: start ;
                         T: A.debit(1000) ... aborts
S: y := B.read() ;
S: Stmt.print( x+y );
```

Strict TSO

Locks are held until commit or abort.

Optimistic concurrency control

At commit time, the history of actions on each object is inspected.

If all histories are consistent with some serialization order

- The commit succeeds
- Otherwise: The transaction aborts and is rescheduled.

Each transaction has three phases

1. Execution: Make shadow copies of objects and execute on those, recording the history of actions.
2. Validation: Following commit, check the history for consistency with some serialization order.
3. Update: Objects are written to persistent store.

But the last phase is inefficient as it must be done atomically.

Instead objects can be written back nonatomically.

Another transaction might start with an inconsistent set of objects

- A transaction is only allowed to commit if
  - At its start time all objects were consistent
  - Its history is consistent with a serialization order

Deadlock

Locking can lead to deadlock

S : Lock(A) ;	T: Lock(B)
S: Lock(B)...	T: Lock(A) ...

Deadlock prevention

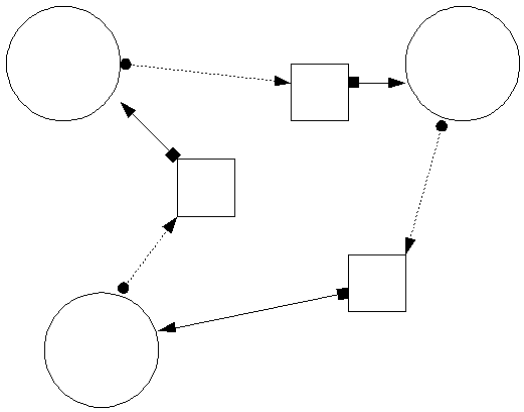
Deadlocks can be prevented if every transaction obtains locks in a fixed order.

Deadlock detection

We make a “resource allocation graph” with edges

- From locked resource r to transaction t that holds the lock. $\longrightarrow$
- From transaction t to resource r when t is waiting for a lock on r. $\dashrightarrow$

Deadlock exists when there is a cycle in the graph.



Alternatively, an object can assume deadlock if obtaining a lock times out.

Deadlock resolution.

Deadlock can be resolved by aborting transactions.

Distributed Transactions

Transaction processing is distributed over multiple servers

Each object has a “home server”

Operations implemented by RPC

Concurrency Control

2 phase locking

Deadlock detection is difficult since knowledge of the “resource allocation graph” is distributed.

But time-outs can be used.

Time stamp ordering

We can use Lamport’s logical clocks for time stamps.

Optimistic Concurrency Control

Logical clocks are used for timestamps.

Information from multiple nodes must be used to determine if serialization is possible.

Distributed Commit

Regardless of the method of concurrency control,

- multiple servers must agree on whether a transaction commits or aborts.

Partial failure: Some servers may fail while others remain up.

Two phase commit (2PC)

One server serves as “commit manager”.

1. First phase
   a. The commit manager requests each server to vote
   b. Each server votes for commit or abort.
   c. Servers that vote for abort stop
2. Second phase
   a. The commit manager tallies votes. Commit requires unanimity.
   b. The commit manager sends result to servers that voted for commit.
   c. Remaining servers commit or abort

Coping with failure in 2PC Nodes may fail. Links may fail.

Assumption: Sender is informed of any communications failure.

Assuming manager runs.

- Request (1.a.) lost. Server doesn't vote
  - Manager assumes "abort"
- Server fails to vote (1.b.) or vote is lost.
  - Manager assumes vote is to "abort"
- Server crashes before result arrives or result is lost
  - Server requests a resend.

If manager crashes

- Before all votes are in
  - Can default to abort on recovery.
- Before sending all results
  - Servers will time out and request a resend
  - Manager must be ready to resend after recovery