Transaction Processing

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

```
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 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 never started.

Consistency The transaction takes the system from one consistent state to another.

Isolation The intermediate results of the transaction are not revealed to any other transaction.

Durability Once the transaction is committed, subsequent failures will not undo it.

Transaction Primitives

begin_transaction mark the start of a transaction

end_transaction mark the end of a transaction

abort_transaction kill the transaction and restore the old values

Classification of Transactions

Flat — series of operations, satisfying ACID. Don't allow any partial commits.

Nested — transaction made up of sub-transactions, each of which may be further sub-divided (tree). Commit is relative to parent.

Distributed — flat transaction operating on distributed data.

Aborts

- Self-abort: transaction decides it has failed.
- Failure of server.
- Aborted to resolve deadlock.
- Aborted by parent in nested transaction.

Durability: Write-ahead Log

- Before any change is made, info about the change is written to a log (on disk).
- Information in the log must be sufficient to either undo or redo the change.
- Undo and redo must be *idempotent* — they can safely be done too many times (i.e., `o.undo(a); o.undo(a) == o.undo(a)` and `o.a(); o.redo(a) = o.a()`.

Also log start, commit and abort operations.

To execute an operation:

1) Read object state

2) Calculate the new state

3) Write the write-ahead log

4) Write to the object

It is important that the write goes first and is written to disk.

Checkpointing

Periodically

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

Crash Recovery

After a system crash:

- Recover object state from disk.
- Transactions committed before the last checkpoint — no action.
- Transactions committed since the last checkoint — redo from last checkpoint.
- Transactions not committed — undo to last checkpoint, restart.

Concurrency Control

Consider two transactions:

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

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

- For efficiency we'd like to run them concurrently.
- Concurency is available both within and between the transactions.
- Anomictiy means that the effect of running the transactions should be as if they were run sequentially in some order.

- Model transaction as directed acyclic graph:
  - Nodes are atomic actions on objects
  - Edges express ordering constraints

Two operations on the same object are *conflicting* if they don't commute (i.e., `o.a(); o.b() != o.b(); o.a()`)

Given a set of transactions we add edges between conflicting operations from different transactions to impose a total order on the operations.

Two-phase Locking

Locking is not sufficient:

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

Locking phase Transactions acquire locks 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.

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

Lock(A)	
x := A.read()	
	Lock(A) *delays* . . .
Lock(B)	
y := B.read()	
Lock(Stmt)	
Stmt.print(x+y)	
Unlock(A)	
	acquire lock . . . A.debit(1000)
	Lock(B) *delays* . . .
Unlock(B)	
	acquire lock B.credit(1000)
Unlock(Stmt)	
	Unlock(A)
	Unlock(B)

Deadlocks

Prevention — Every transaction obtains locks in the same order.

Detection — Look for "wait-for cycle" using a resource-allocation graph. Nodes for resources and transactions, edges

- from locked resource r to transaction t that holds the lock.
- from transaction t to resource r when t is waiting for a lock on r.

Also assume deadlock if obtaining a lock times out.

Resolution Abort transactions.

Time Stamp Ordering (TSO)

Each transaction, T is assigned a unique[1] timestamp, $ts(T)$, corresponding to the start time of that transaction.

Each object, o, is assigned timestamps, $ts_{\mathrm{RD}}(o)$ and $ts_{\mathrm{WR}}(o)$ corresponding to the timestamp of the transaction that most recently read or wrote it, respectivly.

If T wants to read o:

lock(o)
if $ts(T) < ts_{\mathrm{WR}}(o)$ **then #** *o has been written since T started*
  unlock(o); abort(T)
else
  read; $ts_{\mathrm{RD}}(o) := \max(ts(T), ts_{\mathrm{RD}}(o))$; unlock($o$)
end if

[1] See "Lamport's logical clocks algorithm" to see how this can be done.

If T wants to write o:

lock(o)
if $ts(T) < ts_{\mathrm{RD}}(o)$ **then #** *o has been read since T started*
unlock(o); abort(T)
else
write; $ts_{\mathrm{WR}}(o) := \max(ts(T), ts_{\mathrm{WR}}(o))$; unlock($o$)
end if

Optimistic Concurrency Control

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

- If histories of all objects are consistent with some serializatioon order then commit succeeds.
- Otherwise commit fails, transaction is aborted.

Transaction phases:

1) Execute: make shadow copies of objects and execute operations on those, recording history of actions.

2) Validate: following commit, check the history for consistency with some serialization order.

3) Update: write objects to persistent store.

Last phase is inefficient since it must be done atomically.

Alternatively, write objects non-atomically but then another transaction might start with an inconsistent set of objects: only allow commit if

- at start time all objects were consistent, and
- its hisotry is consistent with a serialization order.

Distributed (Two-phase) Commit

We want an operation to be performed on each of a distributed set of processes (process group) or none at all.

Use a coordinator process to initiate the commit.

1) Coordinator sends *Vote_request* to all participants.

2) Participant, upon receiving *Vote_request*, replys with *Vote_commit* if it is prepared to commit the transaction, or *Vote_abort* otherwise.

3) If coordinator receives *Vote_commit* from all participants then it sends *Global_commit* to all. If one or more replied *Vote_abort* then sends *Global_abort*.

4) Particpants take action (commit/abort) upon receiving *Global_commit* or *Global_abort*.

Coping with failures

Nodes or links may fail — Use timeout mechanism to avoid indefinite waits.

Consider states where messages may not be received:

Coordinator

Wait missing *Vote_** — broadcast *Global_abort*

Participant

Init no *Vote_request* — send *Vote_abort*

Ready no *Global_** — need to figure out what the global vote was.

- Could simply wait for coordinator to recover/timeout and re-send.
- Could ask other participants what they got.