```
Transactions (local transactions)


Transactions groups a set of read/write operations together
begin_tx
v <-- read(checking)
v' <-- read(saving)
write(checking, v-10)
write(saving,v'+10)
commit_tx

Transactions Properties: ACID
Atomicity, Consistency, Isolation, Durability

Two goals:
1. handle failure (A, D)
2. handle concurrency (I)

First goal first.
----- Failure handling ---------------
begin_tx
end_tx
commit_tx starts the commit protocol:
write(checking, 90) to disk
write(saving, 110)  to disk

In example,
what happens when crash happens after first write, before second write.
---> after recovery, partial writes, money is lost

Ideal property:
all-or-nothing atomicity

How to implement all-or-nothing atomicity: Write-ahead Logging

One type of logging: redo logging
append_log_entry(checking=90,saving=110)
write(checking, 90)  <------ does not have to flush to disk
write(saving, 110)   <-------does not have to flush to disk

how to ensure each log entry can be written atomically?
(each log entry is checksummed)

how to recover from logs?
replay from the beginning of time
recovery takes too log

periodic checkpoint
how to checkpoint?
naive way: stop all transactions, flush all storage state to disk, write a check
point entry to log
2nd approach: flush all storage state to disk, write a checkpoint entry to log
Question: is 2nd approach correct?
At recovery time, persistent state might contain partial writes from committed t
ransactions.
---> some writes may be applied more than once --> okay?

it's correct if log entries contain the after-image.
not correct if log entries contain the operations. e.g. append_log_write(transfe
r 10 from checking to saving)

Note that writes *during* a transaction cannot be flushed to disk
otherwise, redo-logging + checkpoint would not be incorrect


Second goal
-------- Concurrency control --------------
T1:
transfer $10 from checking to saving
```



```
T2:
withdaw $20 from checking

T3:
read the sum of checking, saving

Bad interleaving
T1: Read(C)=100,              Read(S)              Write(C, 90), Write(S, 110)
T2:              Read(C)=100          Write(C, 80)

What's the expected outcome?
Isolation: T1 happens either before or after T2 (sometimes called before-after a
tomicity)

Ideal isolation semantics: serializability

Definition: execution of a set of transactions is equivalent to some serial orde
r
- Two executions are *equivalent* if they have the same effect on database and p
roduce the same output
- typically, people mean strict serializability, i.e. the equivalent serial orde
r cannot re-order commit-to-begin_tx ordering.

Conflict serializability
- An execution schedule is an ordering of read/write/commit/abort operations (ex
ecuted by the local system)

e.g.
R_1(C), R_2(C), R_1(S), W_2(C), Commit2, W_1(C), W_1(S), Commit1

Two schedules are equivalent if they:
* contain the same operations
* order conflicting operations the same way:
A pair of operations conflict if they access the same data item and one is a wri
te.

Examples:
Serializable? R_1(C), R_1(S), R_2(C), W_2(C), Commit2, W_1(C), W_1(S), Commit1
No
Serialiazable? R_1(C), R_1(S), R_3(C), R_3(S), Commit3, W_1(C), W_1(S), Commit1
equivalent to R_3(C), R_3(S), Commit3, R_1(C), R_1(S), W_1(C), W_1(S), Commit1
Serialiazable? R_1(C), R_1(S), W_1(C), R_3(C), R_3(S), Commit3, W_1(S), Commit1
no.

How to ensure a serializable schedule?
Locking-based approach
Strawman solution 1:
* Grab globa lock when transaction starts
* release global lock when transaction finishes committing


Strawman solution 2:
- grab lock on item X before read/write X
- release lock on item X after read/write X


Problem with strawman 2?
Permits this non-serializable interleaving
R_1(C), R_1(S), W_1(C), R_3(C), R_3(S), Commit3, W_1(S), Commit1

Look at W_1(C), if lock on C is released, then another transaction T' can read n
ew value of C,
but T' must be able to read new value of S that T has not yet written yet.
If write lock is not held, ---> read of uncommitted value
So, write lock must be held till the transaction commits

How about read locks, must it also be held till transaction commits?
R_3(C), R_1(C), R_1(S), W_1(S), W_1(S), Commit1, R_3(S), Commit3
```

```
This is a non-serializable schedule
R_3(C) reads old value
R_3(S) reads new value (non-repeatable reads)

So, read locks must also be held till commit time


2-phase-locking (2PL)
- a growing phase in which transaction is acquiring locks
- a shrinking phase in which locks are released

in practice
growing phase is the entire transaction
shrinking phase is during commit

Optimization: use read/write locks instead of exclusive locks

2PL example:
T1:
Begin_tx
R_lock(C)
v<-R(C)
R_Lock(S)
v'<-R(S)
write_lock(C)
write v-10 to  C (in T1's write-buffer)
write_lock(S)
write v'+10 to S (in T1's write-buffer)
Commit_tx
write (C=90)
write (S=110)
unlock(C)
unlock(S)

T3:
begin_tx
read_lock(C)
v = read(C)
read_lock(S)
v = read(S)
commit_tx
unlock(C)
unlock(S)

Can R_1(C), R_1(S), W_1(C), R_3(C), R_3(S), Commit3, W_1(S), Commit1 ocurr?

Proof sketch of 2PL:
suppose exists some non-serializable schedule
in this schedule, there must exist a pair of transactions such as
T -> T' (according to conflicting access on item a)
T' -> ... -> T (according to conflicting access on item b)

1) T releases lock of a
2) T' grabs lock of a
3) T' releases lock of b
4) T grabs lock of b

according to non-serializable schedule
1 -> 2
3 -> 4

according to rules of 2PL
4 -> 1
2 -> 3

so,
4->1->2
2->3->4
a contradiction
```



```
More on 2PL:
* what is a lock is unavailable?
* deadlock possible?
* how to cope with deadlocks?
  - grab locks in order? not always possible
  - (central) system detects deadlock cycles and aborts involved transactions


The phantom problem:
database has fancier ops than k/v store
T1:
begin_tx
update employee (set salary=1.1*salary)
where dept = "CS"
commit_tx


T2: begin_tx
insert into employee ("carol", "CS")
insert into employee ("mike", "CS")


If only locking individual items
T1: update Bob
T1: update Herbert
T2: inserted carol
T2: inserted mike
T1: update larry
T1: update mike
T1: ...

Problem: cannot just lock items touched, but must also guarantee the non-existen
ce of new ones!
Solution: * predicate lock (lock the predicate statement "depart=cs")
          * range locking (lock the range of B-tree index on the department)
          * often ignored in practice


Degree of isolation paper
* degree 0 (read uncommitted)
short locks
* degree 1 (read committed)
short read locks, long write locks
* degree 2 (repeatable read)
long read/write locks. (serializable unless you squint, default mysql isolation
level)
* degree 3 (serializable)
long read/write locks + solving the phantom problem



Why offer less than serializability?
Performance (allow more concurrent operations. e.g. running a big analytics jobs
 on the database)

Disadvantage of locking-based approach?
* need to detect deadlocks
* big read-only transactions are performance killers

Other concurrency control methods:
Optimistic concurrency control
T1:
Begin_tx
v<-R(C)  # remember read version c#=1
v'<-R(S) # remember read version s#=1
write v-10 to C (in T1's write-buffer)
write v'+10 to S (in T1's write-buffer)
Commit_tx
```

```
Lock(C)
Lock(S)
validate C# still 1
validate S# still 1
write (C=90), version 2
write (S=110), version 2
unlock(C)
unlock(S)

*OCC has no deadlock problem
*OCC does not read a consistent snapshot (no good for read-only transactions)

Multi-version concurrency control
* Each data item is associated with multiple versions (not just version #)
* multi-version transactions:
  - reads choose the appropriate version
  - at commit time, system validates if okay to make writes visible (by generati
ng new versions)

Snapshot isolation
A transaction
* reads a "snapshot" of database image
* can commit only if there are no write-write conflict

------------------------------------------------------------>
     T1: begin_tx, Read(C),    Write(C)
 begin_tx : T is assigned a start timestamp T.sts
 Read: T reads the biggest version i such that i < = T.sts
 Write: buffers writes, T.wset += {C}
 commit_tx: T is assigned a commit timestamp T.cts
            checks for all T' such that T'.cts \in [T.sts, T.cts] whether T'.wse
t and T.wset overlaps
            write item with version T.cts


Recall our earlier bad interleaving:
T1: Read(C) Read(S) Write(C, 90)                Write(S, 110)
T3:                              Read(C) Read(S)

T3 reads old value of C and S (due to T3.sts < T1.cts)


Does snapshot isolation implement serializability? No
The write-skew problem

R_1(A), R_2(B), W_1(B), W_2(A), C_1(A), C_2(B)
non-serializable interleaving
but, possible under SI
```