Introduction to Transactions
(Atomicity, in particular)

Hari Balakrishnan

6.033 Spring 2015
April 6, 2015
xfer(fromacct, toacct, amt)

xfer(from, to, amt):
  # debit “from”
  f = read_disk(from)
  f ← f – amt
  write_disk(from, f)
  # credit “to”
  t = read_disk(to)
  t = t + amt
  write_disk(to, t)

xfer(from, to, amt):
  # debit “from”
  x = read_disk(from)
  x = x – amt
  write_disk(from, x)
  CRASH!
  # credit “to”
  t = read_disk(to)
  t = t + amt
  write_disk(to, t)
  CRASH!
  CRASH!
  CRASH!
All-or-nothing atomicity

A sequence of steps is an *all-or-nothing action* if, from the point of view of its invoker, the sequence always either

- **completes**, or
- **aborts** in such a way that it appears that the sequence had never been undertaken (i.e., it *backs out*).

All-or-nothing: “*Do it all or not at all*”
Now consider concurrent xfer()s

xfer(from, to, amt):
    # debit "from"
    f = read_disk(from)
    f = f – amt
    write_disk(from, f)

    # credit "to"
    t = read_disk(to)
    t = t + amt
    write_disk(to, t)

audit(from, to, TOTAL) {
    sum = read_disk(TOTAL)
    f = read_disk(from)
    t = read_disk(to)
    if f + t != sum:
        raise_alarm()
Before-or-after atomicity

Concurrent actions have the *before-or-after* property if their effect from the point of view of their invokers is as if the actions occurred *either completely before or completely after* one another.
Isn’t this just locking?

- Well, yes...

- But developers need to do it

- And what if you want to *atomically* do
  xfer(A, B)
  xfer(B, C)
  xfer(C, D)
Atomicity

• Atomic = All-or-nothing + Before-or-after

• An invoker (a higher layer) cannot discover the internal structure of an atomic action’s implementation
Implementing all-or-nothing atomicity

• Special case: all_or_nothing disk sector put and get – today

• General approaches
  • Version histories (in book; not covered)
  • Logging → Wednesday (write-ahead logging) and Thursday recitation (log-structured file system)
Golden Rule of Atomicity

Never modify the only copy!
All-or-nothing disk sectors

• Failure model: crash in the middle of a disk sector write, corrupting data

• careful_get(sector, data): returns OK if and only if data is good (correct, via checksum)

• careful_put(sector, data): may fail if crash occurs during operation (e.g., power failure or other crash)

• How to achieve all_or_nothing_put(sector, data) so that all_or_nothing_get(sector, data) returns last successful put()?
All-or-nothing disk sector write ("put")

all_or_nothing_put(s, data):
    # s is a disk sector address
    status = careful_get(s.D0, buffer)
    if status == OK:
        careful_put(s.D1, data)
        careful_put(s.D0, data)
    else:
        careful_put(s.D0, data)
        careful_put(s.D1, data)
All-or-nothing disk sector read ("get")

```python
call_or_nothing_get(s, data):
    status = careful_get(s.D0, data)
    if status == OK:
        return OK
    return careful_get(virtual_sector.D1, data)
```
Transactions: A Programming Model

• All-or-nothing ("Atomic" in the database literature, but "All-or-nothing" in 6.033)
• Before-or-after ("Isolation")
• Effects persist ("Durable")
• "Consistent": satisfies higher-level constraints (e.g., all salaries > 0)

• Aka "ACID"
Transactions

BEGIN TRANSACTION

...  

Pre-commit phase  

...  

Could ABORT anywhere before COMMIT

COMMIT

→ At this point effects are visible to other actions (transactions)
→ Post-commit operations here

END TRANSACTION
Simple programming model

```plaintext
xfer(from, to, amt) {
    /* debit “from” */
    f ← read_disk(from);
    f ← f – amt;
    write_disk(fromacct, f);

    /* credit “to” */
    t ← read_disk(to);
    t ← t + amt;
    write_disk(to, t);
}

BEGIN TRANSACTION
    xfer(savings, checking, 1000)
    COMMIT
    issue_receipt
END TRANSACTION

BEGIN TRANSACTION
    audit(savings, checking, TOTAL) {
        sum = read_disk(TOTAL);
        f ← read_disk(from);
        t ← read_disk(to);
        if (f + t != sum)
            raise_alarm();
    }
    COMMIT
    print_audit_report
END TRANSACTION
```
Benefits of the transaction model

• User doesn’t have to explicitly invoke locks
• All-or-nothing
• Before-or-after (= isolation = “serial equivalence” = “conflict serializability”)
• No need to pre-declare operations: outcomes become visible at COMMIT point
• Extremely powerful abstraction for users (hard to implement for system designer)