Concurrency Control and Recovery [Franklin]
Recovery Manager of System R [Gray et al.]

Background context

We've seen three hard facts of life for computer systems:

- concurrency is needed for performance: to get both high throughput and low latency in the face of a mixture of I/O devices and CPUs, must exploit concurrency
  - parallelism – use multiple CPUs simultaneously
  - overlapping I/O and computation of the same job, or from multiple jobs, to overcome latency of device/network

- concurrency in the face of data sharing creates consistency problems
  - need to use some form of synchronization or conflict detection to avoid races and resulting inconsistency *(the **concurrency control** problem)*

- failures happen!
  - to software, hardware, and storage media (corruption or crash)!
  - and, as a result, we have to worry about partial computations and the correctness of computations after restart *(the **recovery** problem)*

Nowhere is this more evident than when dealing with persistent data. A database is a system that, amongst other things, solves concurrency control and recovery on behalf of programmers, shielding them from needing to worry about this.

Transactions

A turing-award-winning idea; a transaction is an abstraction provided to programmers that encapsulates a unit of work against a database. Specifically, by wrapping a set of accesses and updates in a transaction, the database guarantees:

- **A**tomicity
  - all or nothing: either all operations in the transaction will complete successfully, or none of them will
  - said differently, after a transaction commits or aborts, the database will not reflect a partial result of that transaction. all transactions will commit or abort.
  - Q: if one were to guarantee failure-freeness, does atomicity come "for free"? A: yes, though it is a wide definition of "failure" for this to be true, e.g., no rollback of conflicting or deadlocking transactions.

- **C**onsistency
  - transactions preserve some higher-level consistency constraints on the data: a transaction on an internally consistent database leaves it in an internally consistent state
  - internal consistency == set of integrity constraints (e.g., salary of an employee cannot be negative)
- **I**solation
  - A transaction's behavior is not impacted by the presence of other, concurrently executing transactions
  - said differently, a transaction will "see" only the state of the DB that would occur if the transaction were the only one running against the database, and it will produce results that it could produce if it were running alone
  - Q: if one executes only a single transaction at a time, does "isolation" come for free? A: yes! this is tied to the very definition of isolation.
- **D**urability
  - The effects of committed transactions survive failures
    - If there is non-volatile storage in the system: the effects of a committed transaction must be reflected in non-volatile storage at all times.
    - After a failure, the effects of committed transactions must be recoverable or already reflected in the DB.

We'll ignore the "C", and focus on AID.

Motivating examples

```
TRANSFER()
    01 A_bal := Read(A)
    02 A_bal := A_bal - $50
    03 Write(A,A_bal)
    04 B_bal := Read(B)
    05 B_bal := B_bal + $50
    06 Write(B,B_bal)
```

```
REPORTSUM()
    01 A_bal := Read(A)
    02 B_bal := Read(B)
    03 Print(A_bal + B_bal)
```

Transfer:

- assume statements operateson DB immediately, and DB is a single data structure
  - what happens if there is a crash after 03 but before 06?
  - database loses consistency – money is lost
- Why? because the transfer is **non-atomic**
  - need some way of making sure entire transfer happens, or none

ReportSum:

- fine if ReportSum() executes before or after Transfer()
- what happens if interleaved with Transfer()?

- it depends on the interleaving…some are OK, some are not. The following is not:

TRANSFER	REPORTSUM
01 A_bal := **Read**(A)	
02 A_bal := A_bal - $50	
03 **Write**(A,A_bal)	
	01 A_bal := **Read**(A) /* value is $250 */
	02 B_bal := **Read**(B) /* value is $200 */
	03 Print(A_bal + B_bal) /* result = $450 */
04 B_bal := **Read**(B)	
05 B_bal := B_bal + $50	
06 **Write**(B,B_bal)	

Figure 1: An Incorrect Interleaving of TRANSFER and REPORTSUM

Why is this not OK?

- because ReportSum and Transfer both depend on the same data
- and, Transfer is modifying that data
- and, ReportSum sees both data that predates Transfer() (B) and post-dates Transfer() (A)
    - violates this illusion of sequential execution! we've lost **isolation**.

Concurrency control, theory and practice

We need to solve two different problems to really get a handle on isolation

1. We know we need to interleave the operations of multiple transactions to get the efficiencies of concurrency. Let's call a specific interleaving a schedule (actually, it's a partial ordering…more soon). We have to decide which schedules are correct, and which are incorrect.

   The most widely accepted notion of correctness is serializability: a schedule for a set of transactions produces the same output as some serial execution execution of those transactions (doesn't matter which – why?).

   There are weaker notions of correctness that are possible too, that permit greater degrees of concurrency. Notion of tradeoff between degree of consistency (weak vs strong) and the amount of concurrency possible.

2. Once we have our notion of serializability dialed in, we need some mechanism for enforcing it. Two main ways of doing it:

   a. using locks (or other synchronization primitives) to prevent conflicting accesses. Two major issues: deadlock avoidance/detection, and granularity-related tradeoff between lock overhead amount of concurrency allowed.

b. optimistically permitting transactions to execute, detecting conflicts after the fact, and recovering from them with some sort of rollback.

Serializability:

- a schedule is just a partial ordering of the operations performed by a set of transactions
- a schedule (an ordering) is partial, since it only needs to specify two kinds of dependencies in the schedule:
  - all operations of a given transaction, for which an order is specified by the transaction, must appear in that order in the schedule. (Q: is it possible for operations to not need an order in a transaction? A: yes, of course! e.g., compute an aggregate over a set of tuples, doesn't matter which order.)
  - the ordering of **conflicting operations** from different transactions must be specified. two operations conflict if they both operate on the same data and at least one is a write()
- two schedules are equivalent if (a) they contain the same transactions and operations, and (b) they order all conflicting operations of non-aborting transactions in the same way
- example serial schedule:

$$r_0[A] \rightarrow w_0[A] \rightarrow r_0[B] \rightarrow w_0[B] \rightarrow c_0 \rightarrow r_1[A] \rightarrow r_1[B] \rightarrow c_1$$

A schedule is serializable if and only if it is equivalent to some serial schedule!

The following is serializable, because it is equivalent to the above:

$$r_0[A] \rightarrow w_0[A] \rightarrow r_1[A] \rightarrow r_0[B] \rightarrow w_0[B] \rightarrow c_0 \rightarrow r_1[B] \rightarrow c_1$$

The following is not serializable, since any serial execution of Transfer and ReportSum will have both writes of Transfer preceding both reads of ReportSum (or vice versa):

$$r_0[A] \rightarrow w_0[A] \rightarrow r_1[A] \rightarrow r_1[B] \rightarrow c_1 \rightarrow r_0[B] \rightarrow w_0[B] \rightarrow c_0$$

How does one implement serializability with locks?

- pretty deep and complex topic
- high-order bit: 2-phase locking
  - two kinds of locks
    - shared lock (S) – read-only permission
    - exclusive lock (X) – read or write
  - compatibility matrix between them

        - if a transaction holds a lock, no other transaction can hold a conflicting lock

	S	X
S	y	n
X	n	n

    - assume a transaction is well-formed: always grabs an S while reading, and an X while writing.
        - well-formedness is not sufficient to enforce serializability! why?
        - because you also have to think about when it is safe to **release** a lock, otherwise the bad schedule from above can still happen.
    - two-phase locking says:
        - once a transaction has released a lock, it is not allowed to obtain any additional locks (provides a kind of barrier – is a promise not to read/modify new items not already expressed through locks)
            - growing phase: transaction acquires locks
            - shrinking phase: transaction releases locks
    - two-phase locking is sufficient, but not necessary
        - are some serializable schedules that 2PL excludes! (see above)

<u>Recovery – coping with failures</u>

Useful to have a model of the database. Two-level storage:

- fast, page-allocated, volatile, lower-capacity RAM
    - want to use RAM wherever possible to speed things up, but suffer from (a) lower capacity than the DB, and (b) non-volatile after crash, so need to page stuff out for durability
- slow, page-allocated, non-volatile, high-capacity secondary storage (flash or disk)
    - data is durable in here in case of crash
    - think of there being a data structure that contains consistent data
        - cannot simply overwrite, since not atomic
        - need to either shadow (Gray paper) or use a write-ahead log (Franklin)

How to get durability and atomicity in spite of crashes? Need to be able to do two kinds of things, logically:

- need to be able to remove the effects of an incomplete / aborted transaction, in order to preserve atomicity
    - UNDO
    - Q: can we avoid having to keep undo information explicitly?

        - need to undo if have modified the main data structure; can decide to only modify data structure after a commit, in which case don't need an explicit undo record.
- need to be able to re-instate effects of committed transactions that have not yet reached the main data structure
    - REDO
    - Q: can we avoid having to keep redo information explicitly?
        - can avoid REDO if there is some way to do pointer-swap style atomic commit.

Buffer management strategies:

- STEAL: if the buffer manager allows an update made by an uncommitted transaction to overwrite the most recently committed value on non-volatile storage, it supports STEAL
    - otherwise, NO-STEAL
    - obviously, NO-STEAL obliges buffer manager to keep stuff in memory until commit, or to write to a temporary location
    - STEAL is more efficient, but need logging (undo)
- FORCE: if the buffer manager ensures that all updates made by a transaction are reflected on non-volatile storage before the transaction is allowed to commit
    - otherwise NO-FORCE
    - if NO-FORCE, need some logging for durability (redo)

NO-STEAL / FORCE place the fewest demands on the UNDO/REDO part of recovery management, but are the least efficient.

Aries: super-hairy but awesome STEAL / NO-FORCE recovery manager.

Logging:

- sequential file that stores info about transactions and state of the system
- each entry is a log record with a log sequence number (position in the log)
    - entry can be an update to a record
    - entry can be a system checkpoint – information about what's in the buffer pool, active transactions, etc.
- physical logging
    - location of modified data in DB image, plus the data itself
    - if support for UNDO (i.e., STEAL is used), then the before image (old value) must be in a log record
    - if support for REDO (i.e., NO-FORCE is used), then the after image (new value) must be in a log record
    - physical records are idempotent, which is great, but large, which isn't
- logical logging
    - high-level information; less chatty, but more complex – logical actions might not be atomic, which complicates recovery.

Write-ahead logging (WAL)

- a protocol for ensuring that in the event of a crash, the recovery log contains sufficient information to perform the necessary REDO and UNDO work.
- protocol has two requirements:
    a. all log records about an updated page must be written to non-volatile storage before the page itself is allowed to be overwritten in non-volatile storage (to permit UNDO in case of STEAL)
    b. a transaction is not committed until all of its log records, including the commit record, have been written to stable storage (to permit REDO in case of NO-FORCE)

Woohoo!