Today:

- Undo/redo logging.
- Concurrency: controlling bad interactions among transactions.
- Serial and serializable schedules.

Soon:

- Locking: a mechanism for assuring serializability.
- Other serializability mechanisms: timestamps, validation.

Undo/Redo Logging

Problem: both previous methods have some downside:

- Redo requires keeping all modified blocks buffered until after commit.
- Undo can lose effects of transaction that appear (to the user) to have completed.

Undo/Redo Log

Log entries $<T, X, v, w>$, means transaction T updated DB element X from old value v to new value w.

Undo/Redo Rules

1. Generate a new/old record on the log whenever a DB element is modified (in buffer).
2. Flush the log before updating X on disk.
3. Flush log after writing a `<COMMIT` T`>` record.

- But there is no constraint about whether DB elements are flushed to disk before or after commit point.

Example

One possibility:

`Action`	t	M-A	M-B	D-A	D-B	`Log`
`READ(A,t)`	8	8		8	8	`<START` T`>`
`t := t*2`	16	8		8	8	
`WRITE(A,t)`	16	16		8	8	<$T, A, 8, 16$>
`READ(B,t)`	8	16	8	8	8	
`t := t*2`	16	16	8	8	8	
`WRITE(B,t)`	16	16	16	8	8	<$T, B, 8, 16$>
`FLUSH LOG`						
`OUTPUT(A)`	16	16	16	16	8	`<COMMIT` T`>`
`OUTPUT(B)`	16	16	16	16	16	
`FLUSH LOG`						

Redo/Undo Recovery

1. Find set of problematic transactions:

   ❖ Go back to previous checkpoint; include all that either started after the checkpoint began or are on the "active" list at `START CKPT`.

2. If a transaction has no `COMMIT` record, undo it.

   ❖ Must proceed latest to earliest.

3. If the transaction has a `COMMIT` record, redo it.

   ❖ Must proceed earliest to latest.

Idempotence

An operation is *idempotent* if the result of repeating it several times is the same as doing it once.

- Example: $f(x)$ defined by "execute `x := x+1`" is not idempotent; $f; f$ does not have the same effect as f.
- Example: $g(x)$ defined by "execute `x := 10` is idempotent; $g; g$ has exactly the same effect as g.
  - Thus, the recovery steps recommended for undo, redo, and undo/redo logging are all idempotent.

Problem

What if the transaction involves an inherently nonidempotent operation, such as spitting out cash from an ATM?

- How would you log withdrawals, and how would you recover in a situation where "spit out cash" can be neither undone nor redone?

Concurrent Transactions

Even when there is no "failure," several transactions can interact to turn a consistent state into an inconsistent state.

- Outline:

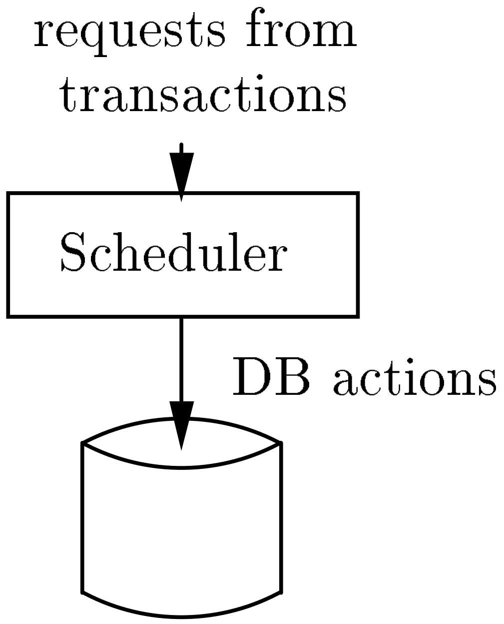
    1. How can a constraint be violated?

    2. *Conflict serializability*: a sufficient condition guaranteeing no violations.

    3. *View serializability*: a more general condition guaranteeing no violations, but harder to verify.

    4. Role of the scheduler in managing transactions so conflict serializability condition is met.

Overview



- Scheduler manages read/write requests from transactions; sends them to DB in an order that is "serializable," i.e., it looks like the transactions executed one-at-a-time.

Example

- Assume $A = B$ is required for consistency. Two transactions:

T_1	T_2
`READ(A,t)`	`READ(A,t)`
`t := t+100`	`t := t*2`
`WRITE(A,t)`	`WRITE(A,t)`
`READ(B,t)`	`READ(B,t)`
`t := t+100`	`t := t*2`
`WRITE(B,t)`	`WRITE(B,t)`

- Note that we omit `OUTPUT` steps for succinctness; they always come at the end.

An Acceptable Schedule

Assume initially $A = B = 25$. Here is one way to execute both transactions so they do not interfere.

T_1	T_2	A	B
		25	25
`READ(A,t)`			
`t := t+100`			
`WRITE(A,t)`		125	
`READ(B,t)`			
`t := t+100`			
`WRITE(B,t)`			125
	`READ(A,t)`		
	`t := t*2`		
	`WRITE(A,t)`	250	
	`READ(B,t)`		
	`t := t*2`		
	`WRITE(B,t)`		250

Another Acceptable Schedule

Here, we execute the transactions in the other order. The result is different, but consistency is maintained.

T_1	T_2	A	B
		25	25
	`READ(A,t)`		
	`t := t*2`		
	`WRITE(A,t)`	50	
	`READ(B,t)`		
	`t := t*2`		
	`WRITE(B,t)`		50
`READ(A,t)`			
`t := t+100`			
`WRITE(A,t)`		150	
`READ(B,t)`			
`t := t+100`			
`WRITE(B,t)`			150

Interleaving Doesn't Necessarily Hurt

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,t)		
	t := t*2		
	WRITE(A,t)	250	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
	READ(B,t)		
	t := t*2		
	WRITE(B,t)		250

But Then Again, It Might

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,t)		
	t := t*2		
	WRITE(A,t)	250	
	READ(B,t)		
	t := t*2		
	WRITE(B,t)		50
READ(B,t)			
t := t+100			
WRITE(B,t)			150

Semantics of Transactions Matters Too, If We Let It

Suppose T_3 is identical to T_2 but it multiplies by 1 instead of 2.

T_1	T_2	A	B
		25	25
`READ(A,t)`			
`t := t+100`			
`WRITE(A,t)`		125	
	`READ(A,t)`		
	`t := t*1`		
	`WRITE(A,t)`	125	
	`READ(B,t)`		
	`t := t*1`		
	`WRITE(B,t)`		25
`READ(B,t)`			
`t := t+100`			
`WRITE(B,t)`			125

We Need a Simpler Model

- Assume that whenever a transaction reads X, it changes X in some unique way.
  - Arithmetic coincidences never happen, so if there is *anything* T could have done with X to make the result of some sequence of events have the "wrong" outcome, then it did.
- Thus, we focus on the reads and writes only, assuming that whenever transaction T reads X and then writes it, X has changed in some unique way.
  - Notation: $r_T(X)$ = transaction T reads DB element X; $w_T(X) = T$ writes X.
  - If transactions are $T_1, \ldots, T_k$, then r_i and w_i are used instead of r_{T_i}, w_{T_i}.

Transaction

A *transaction* is a sequence of r and w actions on database elements.

Schedule

A *schedule* is the sequence of read/write events performed by a collection of transactions.

Serial Schedule

All actions for each transaction are consecutive.

- Example: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$; $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$;

Serializable Schedule

Abstractly, one whose effect is guaranteed to be equivalent to that of some serial schedule.

- We'll introduce sufficient conditions for serializability.

Conflicts

- Suppose $r_i(X); r_j(Y);$ is part of some schedule, and we flip the order of these operations.
  - ❖ Even if $X = Y$, the result of the transactions cannot change.
- We can flip $r_i(X);\ w_j(Y);$, as long as $X \neq Y$.
  - ❖ But $r_i(X);\ w_j(X);$ is not equivalent to $w_j(X);\ r_i(X);$. In the second case, T_i reads the value of X written by T_i; in the first case, not.
- We can flip $w_i(X);\ w_j(Y);$, provided $X \neq Y$.
  - ❖ But $w_i(X);\ w_j(X);$ is not equivalent to $w_j(X);\ w_i(X);$. The next to read X (or the final value of X) is different depending on which transaction writes last.
- Conclusion: a read and a write of the same X, or two writes of X *conflict* and may not be swapped in order. All other events may be swapped without the possibility that the effect of the schedule will change.

Example

Consider:

$$
\begin{array}{l}
r_1(A);\ w_1(A);\ r_2(A);\ w_2(A);\ r_1(B);\ w_1(B);\ r_2(B);\ w_2(B);\\
r_1(A);\ w_1(A);\ r_2(A);\ r_1(B);\ w_2(A);\ w_1(B);\ r_2(B);\ w_2(B);\\
r_1(A);\ w_1(A);\ r_1(B);\ r_2(A);\ w_2(A);\ w_1(B);\ r_2(B);\ w_2(B);\\
r_1(A);\ w_1(A);\ r_1(B);\ r_2(A);\ w_1(B);\ w_2(A);\ r_2(B);\ w_2(B);\\
r_1(A);\ w_1(A);\ r_1(B);\ w_1(B);\ r_2(A);\ w_2(A);\ r_2(B);\ w_2(B);
\end{array}
$$

- Result is the serial schedule "T_1, then T_2."