L10
Transactions, Concurrency, Recovery

Eugene Wu
Fall 2015

Overview

Why do we want transactions?

What guarantees do we want from transactions?

Why Transactions?

Concurrency (for performance)
- N clients, no concurrency
  - 1st client runs fast
  - 2nd client waits a bit
  - 3rd client waits a bit longer
  - Nth client walks away
- N clients, concurrency
  - client 1 runs x += y
  - client 2 runs x -= y
  - what happens?

Can we prevent stepping on toes? *Isolation*

```
x += y
a1 = read(x)
b1 = read(y)
store(a1 + b1)
x -= y
a2 = read(x)
b2 = read(y)
store(a2 – b2)
```

```
x += y
a1 = read(x)
a2 = read(x)
b2 = read(y)
store(a2 – b2)
b1 = read(y)
store(a1 + b1)
```

Why Transactions?

What about 1 client, no concurrency?
- Client runs big update query
  - update set x += y
- Power goes out
- What is the state of the database?

Why Transactions?

What about 1 client, no concurrency?
- Client runs big update query
  - update set x += y
- Aborts the query (e.g., ctrl-c)
- What is the state of the database?

If an abort happens, can the database recover to something sensible? *Atomicity, Durability*

Transactions

Transaction: a sequence of actions
action = read object, write object, commit, abort
API between app semantics and DBMS's view

User's view
T1: begin A=A+100 B=B-100 END
T2: begin A=1.5*A A=1.5*B END

DBMS's logical view
T1: begin r(A) w(A) r(B) w(B) END
T2: begin r(A) w(A) r(B) w(A) END

Transaction Guarantees

Atomicity
users never see in-between xact state.
only see a xact's effects once it's commited

Consistency
database always satisfies ICs.
xacts move from valid database to valid database

Isolation:
from xact's point of view, it's the only xact running

Durability:
if xact commits, its effects *must persist*

Concepts

Concurrency Control
techniques to ensure correct results when running transactions concurrently

what does this mean?

Recovery
On crash or abort, how to get back to a consistent (correct) state?

The two are intertwined! The CC mechanism dictates the complexity of recovery!

What is Correct?

Serializability
Regardless of the interleaving of operations, end result same as a serial ordering

Schedule
One specific interleaving of the operations

Serial Schedules

Logical xacts
T1: r(A) w(A) r(B) w(B)
T2: r(A) w(A) r(B) w(B)

No concurrency (serial 1)
T1: r(A) w(A) r(B) w(B)
T2: r(A) w(A) r(B) w(B)

No concurrency (serial 2)
T1: r(A) w(A) r(B) w(B)
T2: r(A) w(A) r(B) w(B)

Are serial 1 and serial 2 equivalent?

More Example Schedules

Logical xacts
T1: r(A) w(A) r(A) w(B)
T2: r(A) w(A) r(B) w(B)

Concurrency (bad)
T1: r(A) w(A) r(A) w(B)
T2: r(A) w(A) r(B) w(B)

Concurrency (same as serial 1!)
T1: r(A) w(A) r(A) w(B)
T2: r(A) w(A) r(B) w(B)

Concepts

Serial schedule
single threaded model. no concurrency.

Equivalent schedule
the database state same at end of both schedules

Serializable schedule (gold standard)
equivalent to a serial schedule

SQL → R/W Operations

```
UPDATE    accounts
SET       bal  =  bal  +  1000
WHERE     bal  >  1M
```

Read all balances for every tuple
Update those with balances > 1000
Does the access method mater?

Why Serializable Schedule? Anomalies

Reading in-between (uncommitted) data

T1: R(A) W(A) R(B) W(B) abort
T2: R(A) W(A) commit

WR conflict or dirty reads

Reading same data gets different values

T1: R(A) R(A) W(A) commit
T2: R(A) W(A) commit

RW conflict or unrepeatable reads

Why Serializable Schedule? Anomalies

Stepping on someone else's writes

T1: W(A) W(B) commit
T2: W(A) W(B) commit

WW conflict or lost writes

Notice: all anomalies involve writing to data that is read/written to.

If we track our writes, maybe can prevent anomalies

Conflict Serializability

What is a conflict?

For 2 operations, if run in different order, get different results

Conflict?	R	W
R	NO	YES
W	YES	YES

Conflict Serializability

def: possible to swap non-conflicting operations to derive a serial schedule.

$\forall$ conflicting operations O1 of T1, O2 of T2
O1 always before O2 in the schedule or
O2 always before O1 in the schedule

	1	2	3	4
T1:	R(A)	W(A)	R(B)	W(B)
	5	6	7	8
T2:	R(A)	W(A)	R(B)	W(B)

Logical

Conflicts
16, 25, 26, 38, 47, 48

Logical

	1	2	3	4
T1:	R(A)	W(A)	R(B)	W(B)
	5	6	7	8
T2:	R(A)	W(A)	R(B)	W(B)

Serializable

	1	2			3	4		
T1:	R(A)	W(A)			R(B)	W(B)		
			5	6			7	8
T2:			R(A)	W(A)			R(B)	W(B)

Logical

	1	2	3	4
T1:	R(A)	W(A)	R(B)	W(B)
	5	6	7	8
T2:	R(A)	W(A)	R(B)	W(B)

Not Serializable

	1		2		3	4		
T1:	R(A)		W(A)		R(B)	W(B)		
		5		6			7	8
T2:		R(A)		W(A)			R(B)	W(B)

Conflict Serializability

Transaction Precedence Graph

Edge Ti → Tj if:

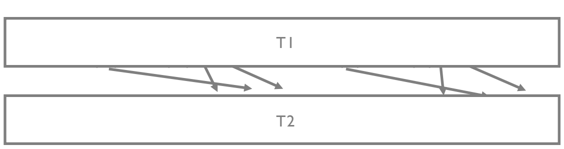
1. Ti read/write A before Tj writes A or
2. Ti writes some A before Tj reads A

If graph is acyclic (does not contain cycles) then conflict serializable!

Logical

	1	2	3	4
T1:	R(A)	W(A)	R(B)	W(B)
	5	6	7	8
T2:	R(A)	W(A)	R(B)	W(B)

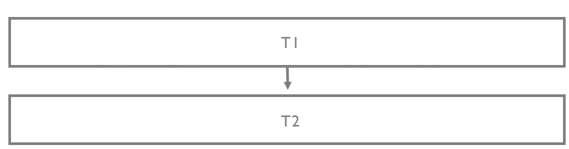
Serializable



Logical

	1	2	3	4
T1:	R(A)	W(A)	R(B)	W(B)
	5	6	7	8
T2:	R(A)	W(A)	R(B)	W(B)

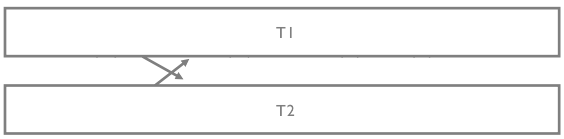
Serializable



Logical

	1	2	3	4
T1:	R(A)	W(A)	R(B)	W(B)
	5	6	7	8
T2:	R(A)	W(A)	R(B)	W(B)

Not Serializable



Fine, but what about COMMITing?

T1 R(A) W(A) R(B) ABORT
T2 R(A) COMMIT

Not recoverable

Promised T2 everything is OK. IT WAS A LIE.

T1 R(A) W(B) W(A) ABORT
T2 R(A) W(A)

Cascading Rollback.

T2 read uncommitted data → T1's abort undos T1's ops & T2's

Lock-based Concurrency Control

Must get a shared(read) or exclusive(write) lock BEFORE op
If other xact has lock, can get if lock table says so

YES

		T1	
	Allowed?	S	X
T2	S	Y	N
	X	N	N

Can this schedule happen?

T1 R(A) W(A) R(B) ABORT
T2 R(A) COMMIT

Lock-based Concurrency Control

Two-phase locking (2PL)
- Growing phase: acquire locks
- Shrinking phase: release locks

shrink here

T1 R(A) W(B)W(A) ABORT
T2 R(A) W(A)

Uh Oh, same problem

Lock-based Concurrency Control

Strict two-phase locking (Strict 2PL)
- Growing phase: acquire locks
- Shrinking phase: release locks
- Hold onto locks until commit/abort

Why? Which problem does it prevent?

T1 R(A) W(B) W(A) ABORT
T2 R(A) W(A)

Guarantees serializable schedules! Avoids cascading rollbacks!

Review

Issues
- TR: dirty reads
- RW: unrepeatable reads
- WW: lost writes

Schedules
- Equivalence
- Serial
- Serializable

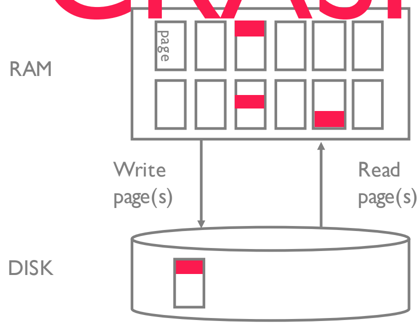
Serializability
- Conflict serializability
- how to detect

Conflict Serializable Issues
- Not recoverable
- Cascading Rollback

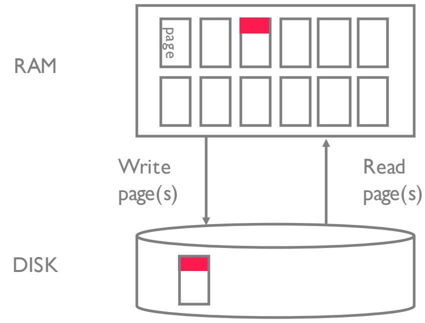
Strict 2 phase locking

Normal Execution

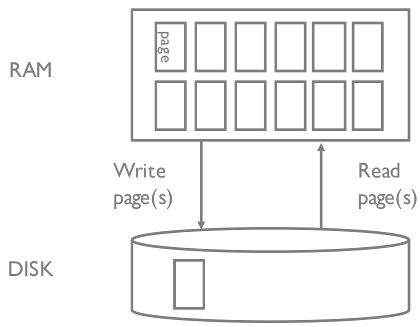
CRASH



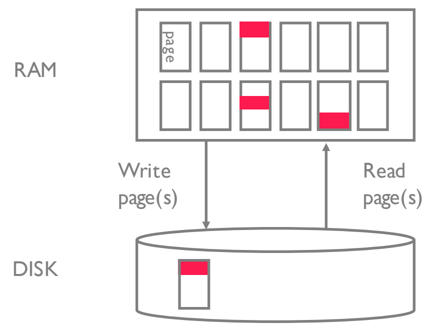
After a Crash



If DB did not say "OK, committed"



If T1 Committed and DB said "OK"



Recovery

Two properties: Atomicity, Durability

Assumption in class: disk is safe. memory is not.

Need to account for
- when pages are modified
- when pages are flushed to disk

Recovery

Deal with 2 cases

If T2 commits, what could make it not durable?
- didn't write all changed pages to disk

When could uncommitted ops appear after crash?
- wrote modified pages before commit

Aborts and Undos

If Tx aborts, all of its actions must be undone.

- Ty that read Tx's writes must be aborted (cascading abort)
- Strict 2PL avoids cascading aborts

Use a log to know what actions to undo

1. A = 1
2. B = 5
3. C = 10
4. BEGIN T5
5. A = 10
6. B = B + A
7. C = B – 2
8. ABORT
9. undo 7
10. undo 6

...

Aborts and Undos

If Tx aborts, all of its actions must be undone.

- Ty that read Tx's writes must be aborted (cascading abort)
- Strict 2PL avoids cascading aborts

Use a log to know what actions to undo

On crash, abort all non-committed xacts

1. A = 1
2. B = 5
3. C = 10
4. BEGIN T5
5. A = 10
6. B = B + A
7. CRASH

Logs

Log records

- writes: old & new value
- commit/abort actions
- xact id & xact's previous log record

Write ahead logging (WAL)

- log records stored on disk (persisted) *before* data pages can be persisted
- log is the *ground truth*

So far: use log to undo partial transactions

Durability

Bad scenario

- T1 writes to A
- T1 commits, log record written to disk
- start writing page with A to disk
- *crash*

Can undo help us?

Need to redo T1, otherwise no durability!

Aries Recovery Algorithm

3 phases

1. Analyze the log to find status of all xacts
2. Redo xacts that were committed
3. Undo partial xacts

Recovery is *extremely* tricky and *must be correct*