Lec 16 Concurrency Control Theory

Want: data protection, transaction semantics, concurrency recovery

Want:
- High throughput
- Low latency
- Correct
- Fair

The Problem (transaction manager)

Query : {Read | Write | Action} × TransactionID
Actions : BEGIN, COMMIT, ABORT
Input : string of queries
Decide : interleave correctly

Criteria: ACID

Atomicity All or nothing, no partial completion

Consistency Start consistent and end consistent (think schema)

Isolation Different txns isolated

Durability Persist after txn commit

Atomicity

- Logging
  - log all the changes in log file
- Shadow Pages

Consistency

→ Use indices
→ Consistency checking algorithms

* Or, one could require eventual consistency (like CRDT) in DB usually we want immediate consistency

Isolation

Each txn run as if without interleaving

T_1	T_2
begin	begin
A -= 100	A *= 1.06
B += 100	B *= 1.06
commit	commit

Allow result by either
- T_1 then T_2
- T_2 then T_1

"Serial execution"

Schedule 1

begin
A -= 100
B += 100
commit
begin
A *= 1.06
B *= 1.06
commit

≡

Schedule 2

begin
A *= 1.06
B *= 1.06
commit
begin
A -= 100
B += 100
commit

Interleaving 1 (correct)

begin
A -= 100
begin
A *= 1.06
B += 100
commit
B *= 1.06
commit

Interleaving 2 (incorrect)

begin
A -= 100
begin
A *= 1.06
B *= 1.06
commit
B += 100
commit

Protocols:

- Pessimistic — stop / delay if conflict could happen
- Optimistic — do both, if conflict detected, come back and fix

Dependency Alg

Build R/W conflict graph

- R-W - aka unrepeatable read
- W-R - aka dirty read
- W-W - aka lost update

Thm. If no cycle in this graph then it's conflict serialisable

Graph: V = { txns } E = { dependencies }

T_1	T_2
⋮	⋮
W(A) →	R(A)
⋮	⋮
R(B) ←	W(B)
⋮	⋮

T_1 ⇄ T_2
└ completes cycle — bad

Thm direction in dep. graph tells equivalent serial ordering

* One can also implement view serialisability that allows more valid schedule depending on application semantics

All schedule ⊃ View serialisable ⊃ Conflict serialisable ⊃ Serial