- scheduler has to ensure order of reads and writes is <u>conflict serializable</u>.
- any ordered pair of conflicting operations have to be ordered the same way ↓
- need to belong to diff. transactions, on the same objects, and at least one is a write
- need to be able to interleave operations is very important
  - ↳ response time and throughput would be killed otherwise
- <u>view equivalence</u>: allows more schedules, but NP-hard to compute
- serializability guarantees correctness, but other unpleasant situations can arise

T_i $w[x]$ abort

T_j $r[x]$ commit

- to abort T_i we need to undo a committed transaction T_j
  - ⇒ commits only in order of read-from dependency
  - ⇒ <u>recoverable schedule</u> (RC)

Cascadeless schedules (ACA) avoiding cascading aborts

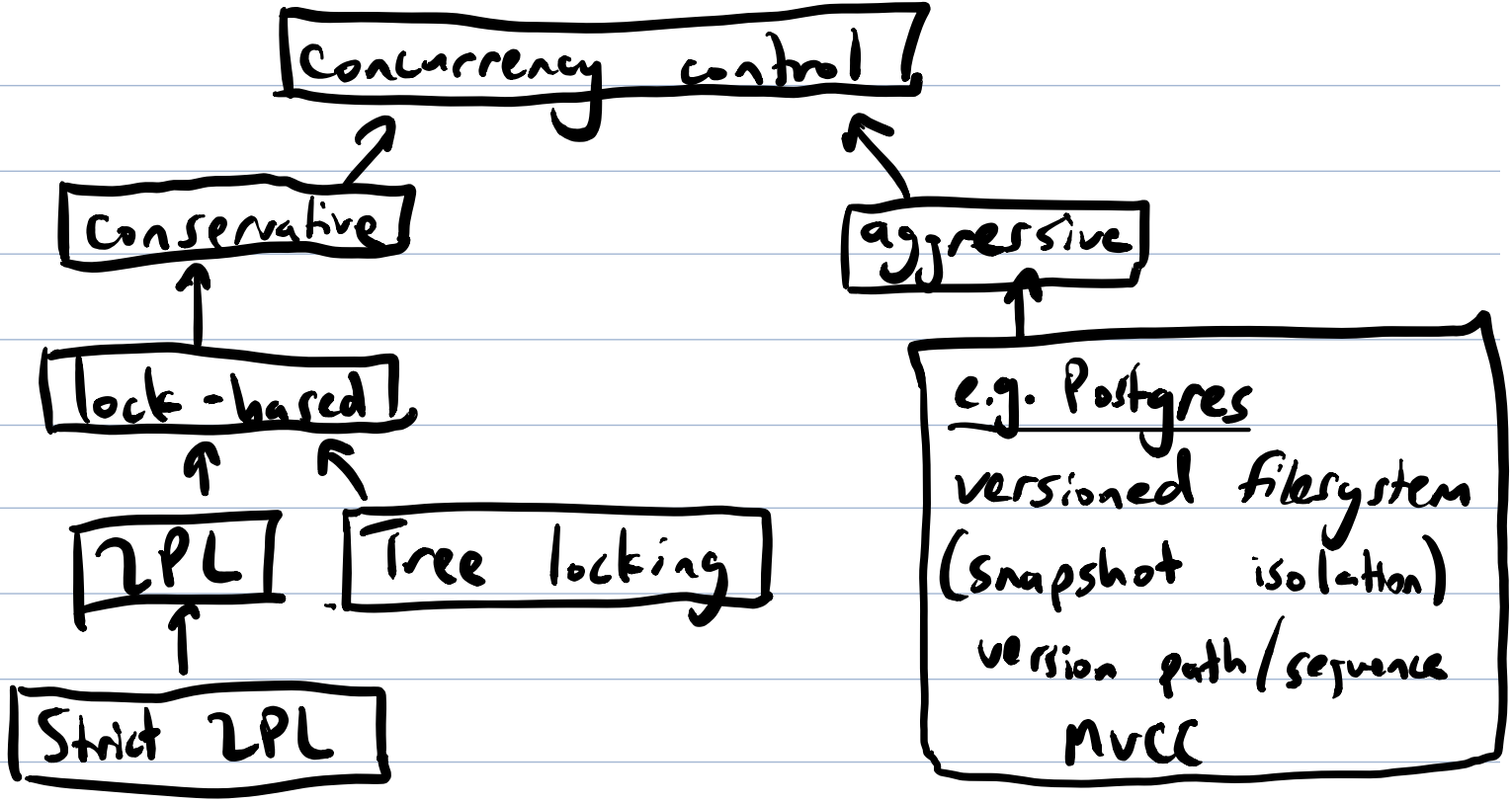
- if T_j above didn't commit we can abort it
  - → may lead to cascading aborts of many transactions
  - ⇒ no reading of uncommitted data

how to get a serializable schedule?

- scheduler receives ops from query processors
- for each op, decide whether to:
  - execute (send to lower module)
  - delay (insert into queue)
  - reject (abort tx)
  - ignore (no effect)
- two main kinds of schedulers
  - conservative (favour delaying)
  - aggressive (favour rejected operations)

2 phase locking (2PL)

- common example of conservative concurrency algo
- tx must have a lock on objects before access:
  - shared lock to read an object
  - an exclusive lock to write an object



2PL protocol: tx must acquire all locks before it releases any of them

- if you log lock requests, tx in 1 of 2 phases:
  1) locks granted
  2) locks released
  - cannot acq → release → acq (needs to be 2-phase)

Theorem

2PL guarantees the produced tx schedules are conflict serializable.

In practice: Strict 2PL (locks held till commit, this guarantees ACA). e.g. DB2

Snapshot isolation:

- for OLTP workloads (e.g. Amazon), the probability of 2 txs buying same items (conflict) is low
  ⇒ optimistic concurrency control
- stores versioning path (Multi-version concurrency control, MVCC)

Strict 2PL problem: Deadlock

$r_1[x]$, $r_2[y]$, $w_2[x]$, blocked by T_1,
$w_1[y]$, blocked by T_2

T_1 Sh[x], r[x], Ex(y)

T_2 Sh[y], r[y], Ex(x)
B

deadlock prevention:

⇒ locks granted only if deadlock impossible
⇒ ordered data items and locks granted in this order, uncommon

deadlock detection:

⇒ wait-for-graphs and cycle detection
⇒ resolution: involuntary abort of one of the offending transactions

Strict 2PL: return code, tx could have been aborted, app needs to deal

2PL does not handle insert / update
- 1 tx counts records
- 2nd adds/deletes record
⇒ Phantom problem
- ops for all records lock against insertion/deletion of a qualifying record
  ⇒ locks on table
  ⇒ index locking + other techniques

R

A	B
1	1
2	2
2	2

T_1: update R set B=2 where B=1
T_2: update R set B=1 where B=2

SQL isolation levels

level 3 (serializable): → OLTP
- essentially table-level strict 2PL, full isolation

level 2 (repeatable read):
- tuple-level strict 2PL - phantom tuples may occur
- no locks on table/files

level 1 (cursor stability):
- tuple-level exclusive-lock only strict 2PL
  read same obj twice: diff values

level 0: neither read nor write locks are acquired, tx may read uncommitted updates
↳ OLAP workloads

snapshot isolation ≈ 2.5 level (write-write control but can read stale value)
↳ to avoid consistency issues (online stealing) null writes to objects needed
↳ Amazon lost $billions due to 2 general bugs:
  - inappropriate use of snapshot isolation
  - no app code usage of begin/end trans

Durability

2 goals
1) allow txs to be
   a) committed (effects permanent)
   b) aborted (⇒ effects disappear)
2) allow db to be recovered to a consistent state in case of hw failure

assume: disk drives and flash memory are okay

input: 2PL ACA schedule of operations produced by TM

output: schedule of R/W/forced writes

2 essential approaches:

1) shadowing

- update copy, copy-on-write and merge-on-commit approach
- poor clustering, used in system R, not in modern

2) logging

- use of LOG (separate disk) to avoid forced writes
- good use of buffers
- preserves original clusters
- record db twice: latest state (primary), + log for deltas