

# Database Administration

José Orlando Pereira

Departamento de Informática  
Universidade do Minho



# Roadmap

- How to implement each solution?
- What solution for each problem?

# 2-Phase Locking

- Acquire a lock for an item before reading or writing it
- All lock requests precede all unlock requests
  - This means unlocking only on transaction commit
- Equivalent to acquiring all locks upfront

# Deadlocks

- Cannot easily be avoided:
  - Interactive transactions
  - Plan selected by the optimizer
- Can be detected:
  - Wait-for graph
  - Time-out
- Resolved by aborting one transaction

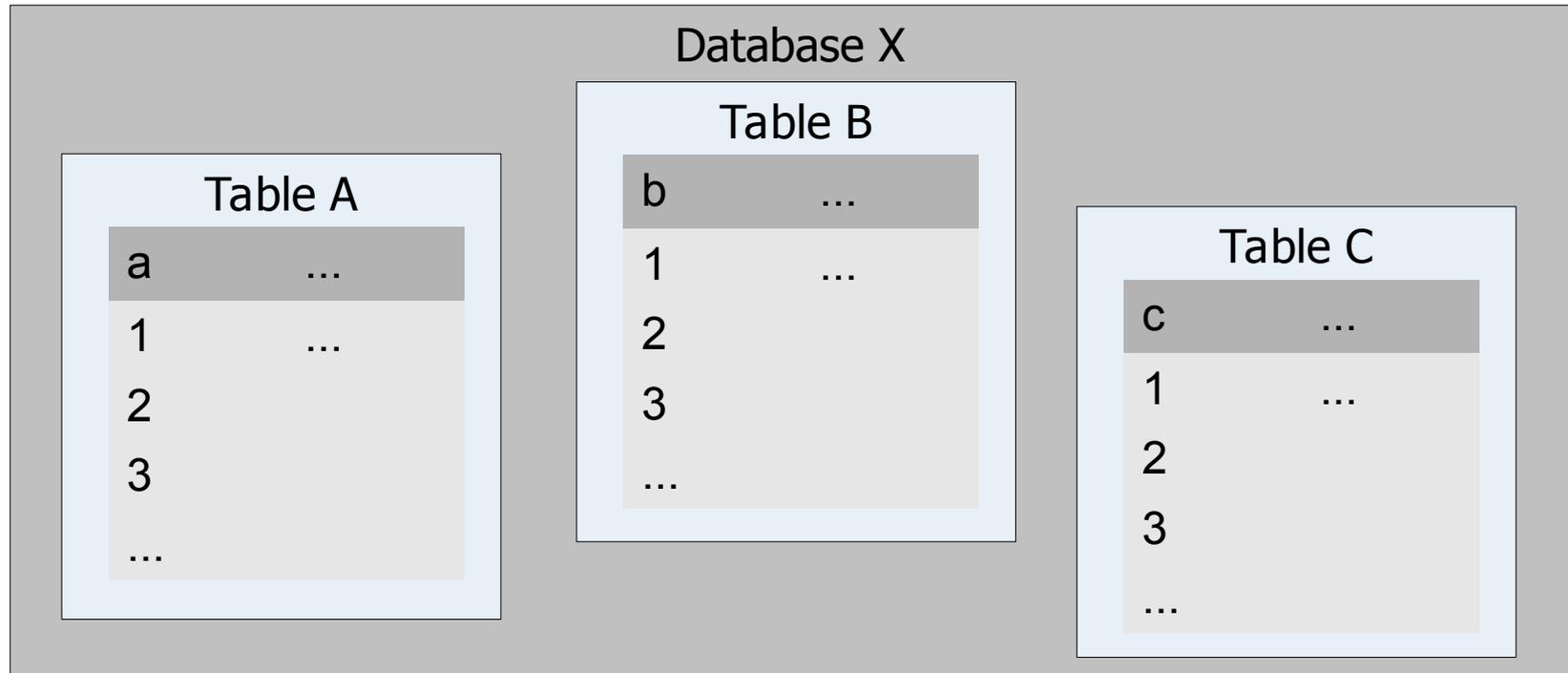
# Shared vs exclusive

Compatible?	Shared	Exclusive
Shared	<b>Yes</b>	No
Exclusive	No	No

- As read/read does not cause anomalies:
  - Exclusive locks for writing
  - Shared locks for reading
- More concurrency is possible

# Multi-level locking

- Row locks vs table locks



# Locking Protocols

- Multiple dimensions:
  - 2-phase locking
  - Shared vs exclusive
  - Granularity
- What combinations?
- How to select them?

# Read uncommitted (aka “browse”)

- Protocol:
  - Exclusive locks on INSERT/UPDATE/DELETE until transaction complete
  - No shared locks
- Allows:
  - ~~Lost update~~ (in a single UPDATE statement)
  - Dirty read
  - Non-repeatable read
  - Phantoms
  - Write skew

# Read committed (aka “cursor stability”)

- Protocol:
  - Exclusive locks on INSERT/UPDATE/DELETE until transaction complete
  - Shared locks on each SELECT statement
- Allows:
  - ~~Lost update~~
  - ~~Dirty read~~
  - Non-repeatable read
  - Phantoms
  - Write skew

# Repeatable read

- Protocol:
  - Exclusive locks on INSERT/UPDATE/DELETE until transaction complete
  - Shared locks on SELECT until transaction complete
- Allows:
  - ~~Lost update~~
  - ~~Dirty read~~
  - ~~Non-repeatable read~~
  - Phantoms
  - Write skew (on collections)

# Serializable

- Protocol:
  - Exclusive locks on INSERT/UPDATE/DELETE until transaction complete
  - Shared range/table locks on SELECT until trans. complete
- Allows:
  - ~~Lost update~~
  - ~~Dirty read~~
  - ~~Non-repeatable read~~
  - ~~Phantoms~~
  - ~~Write skew~~

# Serializable

- A lot of trouble to avoid phantoms:
  - Table locking has a large impact in concurrency
  - Predicate locking or range locking using indexes adds complexity/overhead

# Multi-version

- Never overwrite, always create a new version
- Example transaction 1:
  - insert (aa,11)
  - insert (bb,22)

k	v	from	to
aa	11	1	
bb	22	1	

# Multi-version

- Transaction 2:
  - insert (cc,33)
  - update (bb,44)

k	v	from	to
aa	11	1	
bb	22	1	2
cc	33	2	
bb	44	2	

# Multi-version

- Transaction 3:
  - delete (aa)

k	v	from	to
aa	11	1	3
bb	22	1	2
cc	33	2	
bb	44	2	

# Snapshot isolation

- Protocol:
  - Read from version that existed when the transaction started (or local writes)
  - On I/U/D, lock exclusive and first committer wins, others rollback
- Allows:
  - ~~Lost update~~
  - ~~Dirty read~~
  - ~~Non-repeatable read~~
  - ~~Phantoms~~
  - Write skew

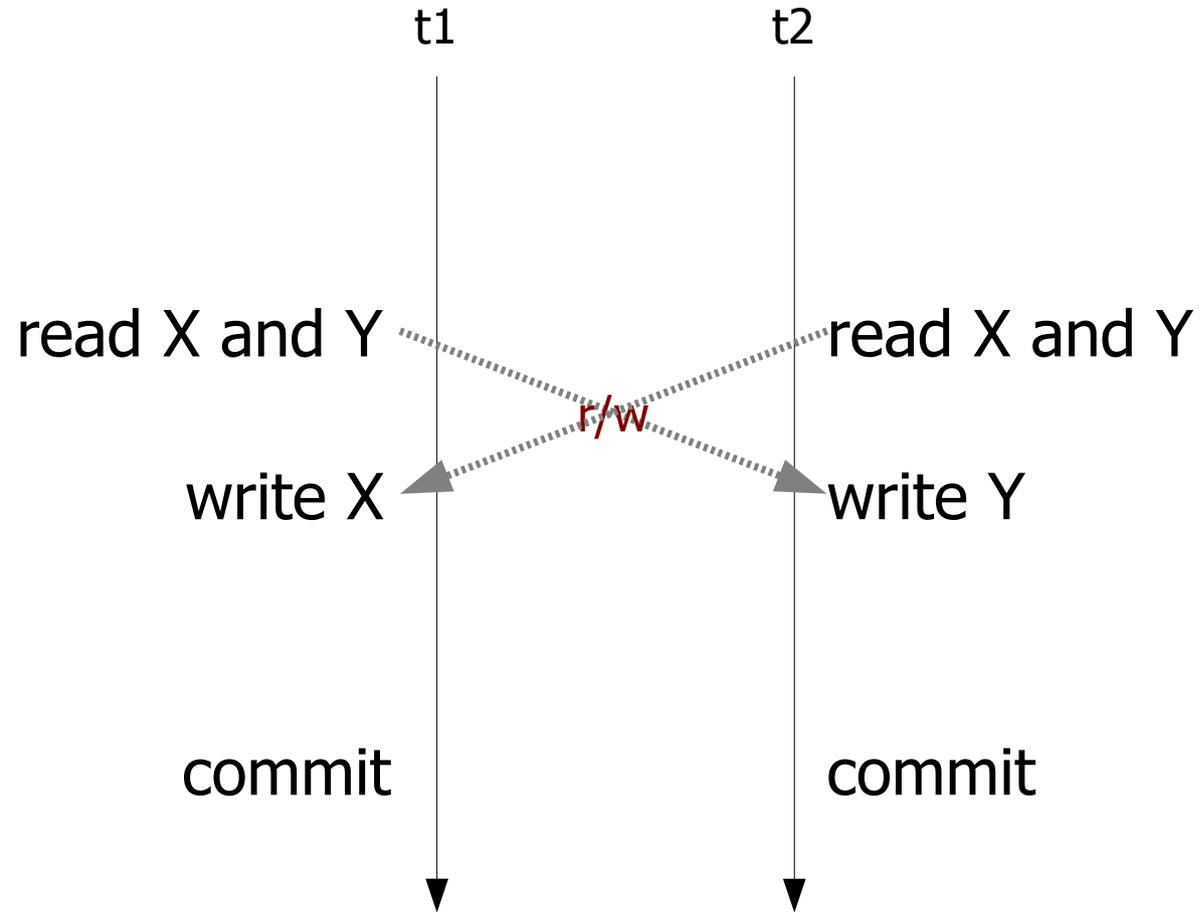
# Snapshot isolation

- To avoid that two transactions execute concurrently:
  - Use explicit locking (SELECT FOR UPDATE)
  - Make them write on the same data item
- If two transactions update the same item, they cannot execute concurrently:
  - Prefer inserts to updates
  - Be careful with:
    - Counters
    - Materialized views of aggregates

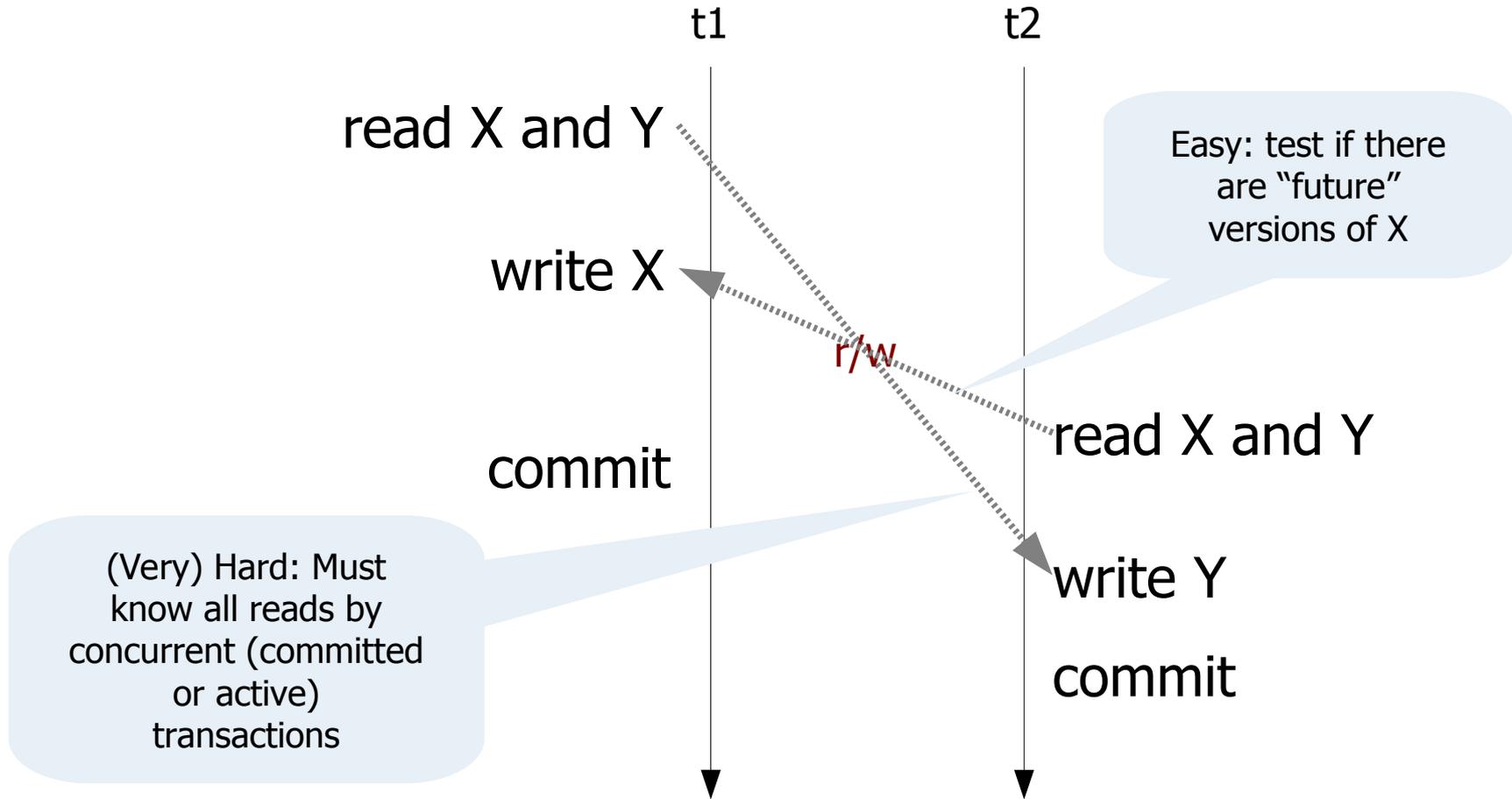
# Serializable snapshot isolation

- Protocol:
  - Everything in SI plus...
  - Detect RW dependencies and abort transactions when two consecutive found (might not be a cycle: false positive!)
- Allows:
  - ~~Lost update~~
  - ~~Dirty read~~
  - ~~Non-repeatable read~~
  - ~~Phantoms~~
  - ~~Write skew~~

# Serializable snapshot isolation



# Serializable snapshot isolation



# Conclusions

- Rollback is not a convenience!
- Snapshot isolation combines all approaches
- Snapshot isolation is now preferred:
  - Never blocks reads
  - Easily becomes serializable
- Must avoid update hot-spots