Lecture 4

Type	Lecture

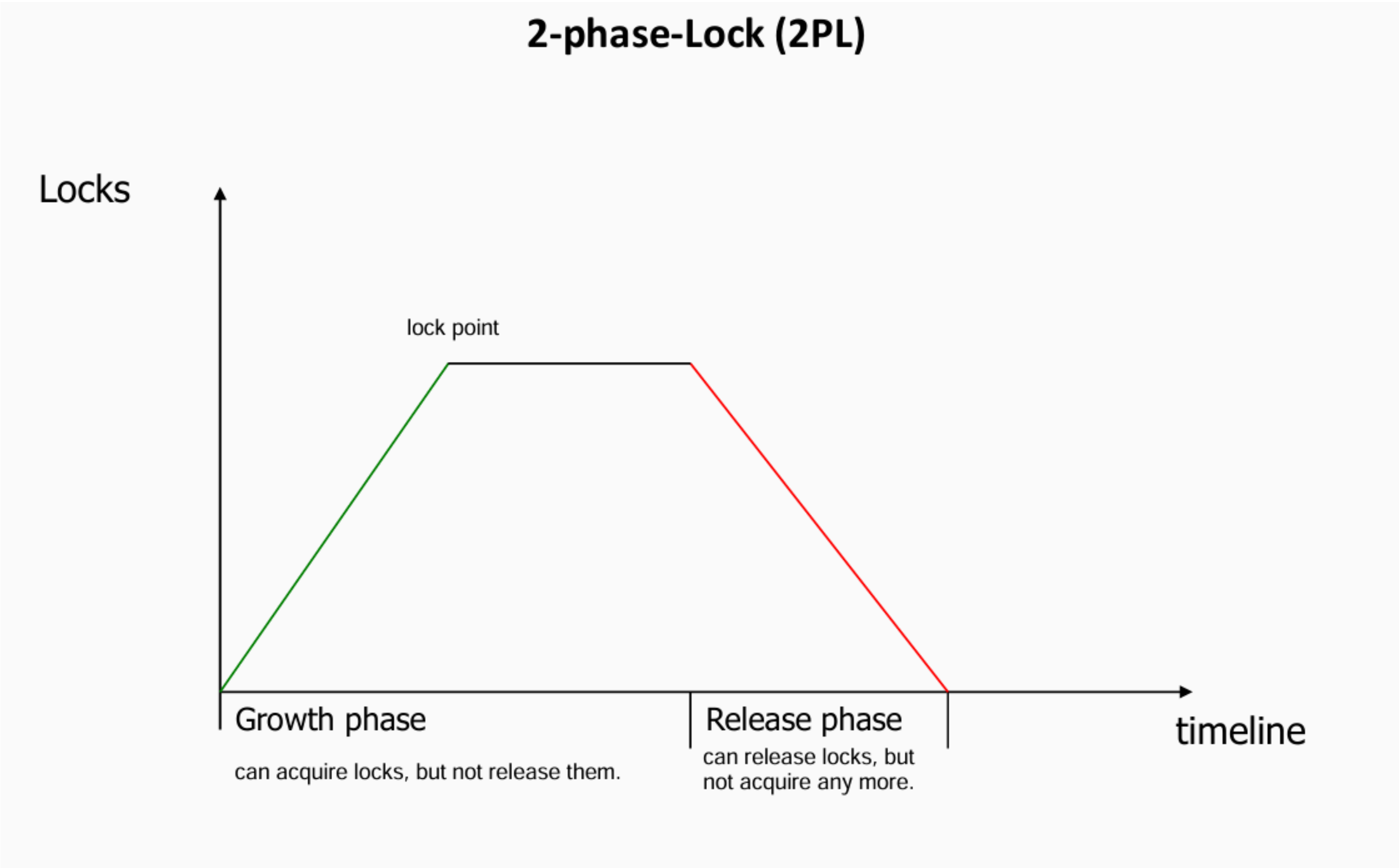
2-phase-Lock (2PL)

The two-phase locking protocol (2PL) guarantees serializability.

The 2PL protocol splits each transaction in two phases:

1. lock phase: all locks are set
2. release phase: locks are released.

No lock after unlock, once you do unlock you can't acquire another lock.



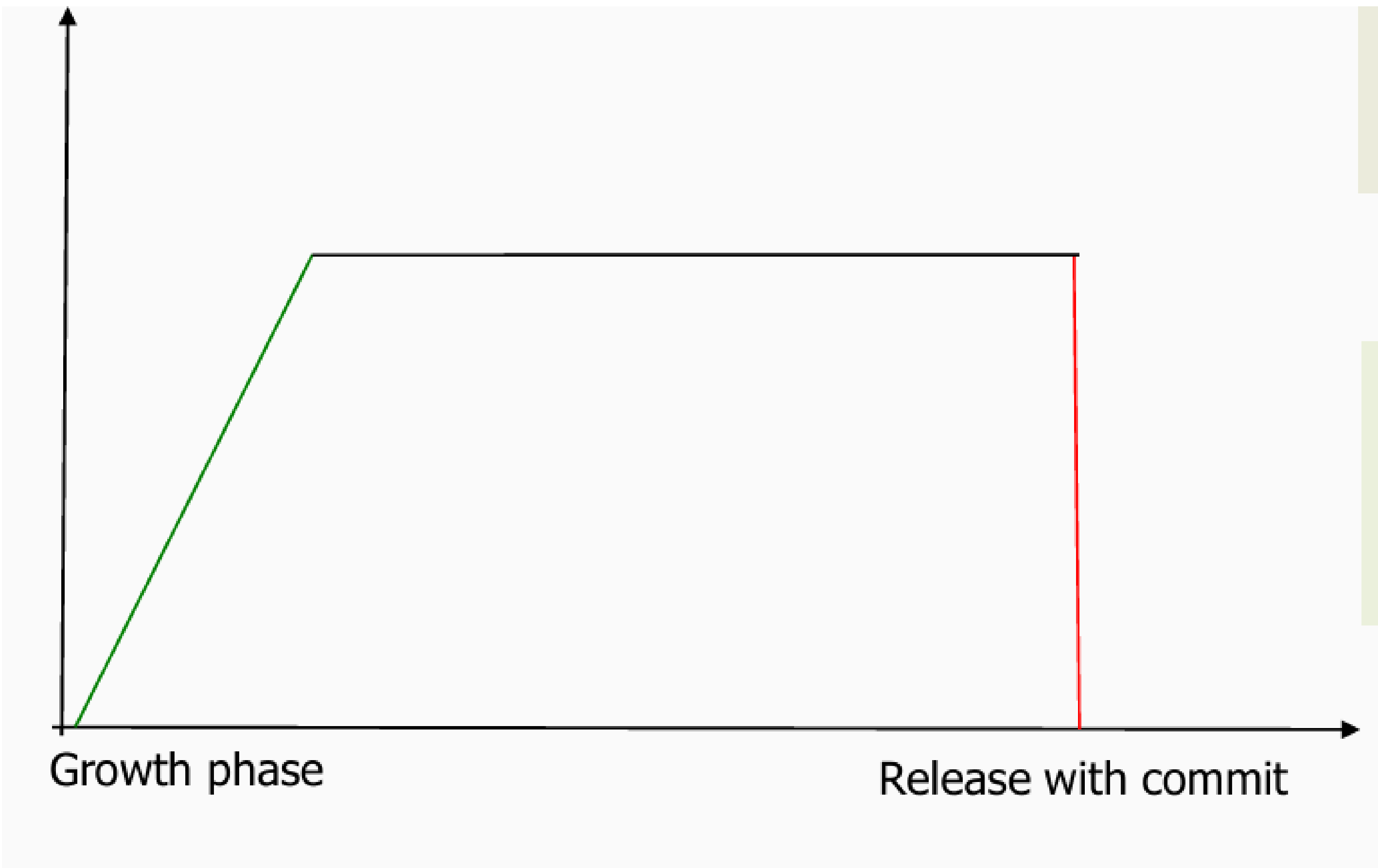
If every transaction uses 2PL, then their interleaving (**Interleaving** refers to how the operations of **multiple transactions** are mixed together during concurrent execution) is guaranteed to be serializable — no need to check separately. That's why 2PL became the standard for decades.

Strict 2PL

To avoid cascading rollbacks, use **strict 2PL**:

- **Locks are only released at commit time**.
- Prevents other transactions from reading or writing uncommitted data.

Most DBMS (like PostgreSQL) implement **strict 2PL** (or similar behavior).



Frequent Problems

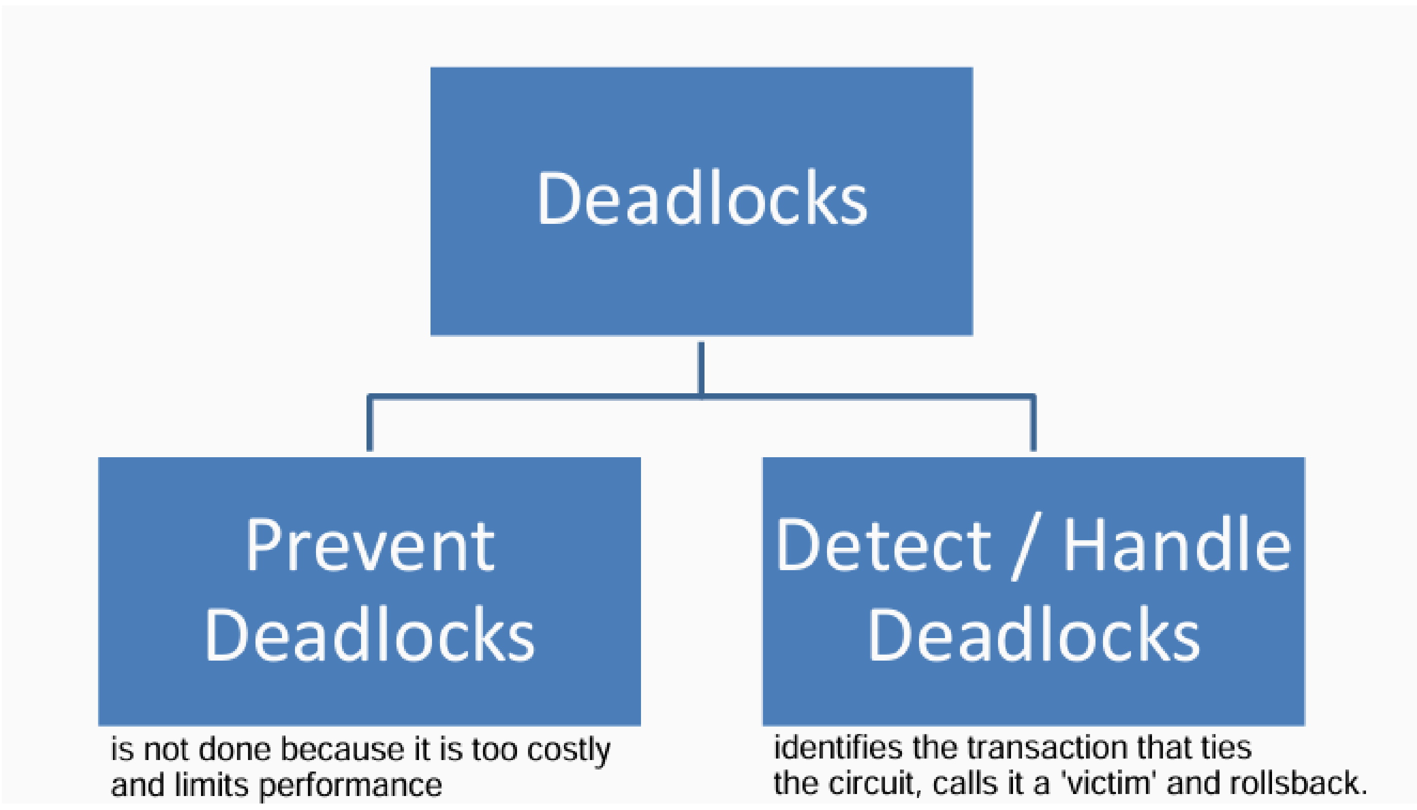
- **Null values** (logic problems)
- **Deadlocks** (transactions waiting on each other)
- **Date/time** confusion (e.g. time zones)
- **Duplicate Data** in Database, e.g. Customer is stored twice
- **Character encoding issues** (e.g. München vs M?nchen)

Deadlocks

Transactions hold locks and request locks held by each other. They're stuck waiting in a loop — no progress. Deadlocks not only **happen in 2PL but also in MVCC**.

▼ Example

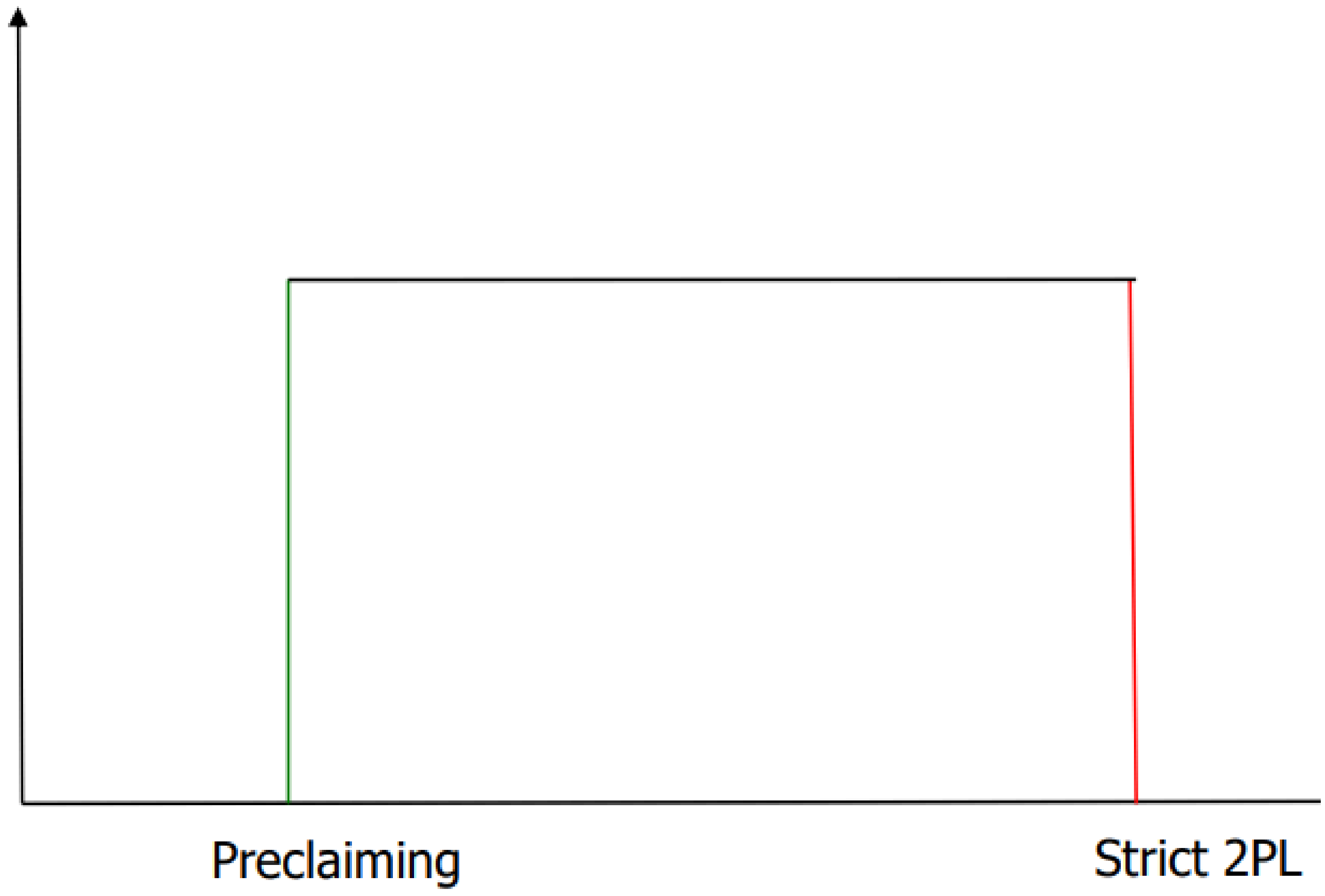
T1 locks A and wants B. T2 locks B and wants A.



Preventing Deadlocks: Preclaiming (Conservative)

Preclaiming 2PL = transaction locks **everything it might need** at the beginning. If it can't, it doesn't proceed. Deadlocks are avoided.

❗ Downside: **Low concurrency** — transactions hold many locks unnecessarily for a long time (limits parallel processing considerably).



Detecting Deadlocks: Waits-for Graph

Deadlocks can be detected using a waits-for graph, each active transaction is represented by a node, a deadlock exists if the waits-for graph contains a cycle and the node that ties the cycle is the one that gets aborted and rolledback.

❗ Downside: The Waits-for-Graph Detection is is computationally expensive.

Since building the graph constantly is expensive, PostgreSQL:

- Waits a short time (default: **1 second**) before checking.
- If no progress, it runs the graph-based deadlock detection.

Preventing Deadlocks: Timestamp methods

Wait-Die:

- Older transactions wait.
- Younger ones are killed ("die") and restarted.

Wound-Wait:

- Older transactions kill ("wound") younger ones.
- Younger ones wait.

These use transaction
timestamps to decide who waits and who gets rolled back.

Dealing with Deadlocks from the application side

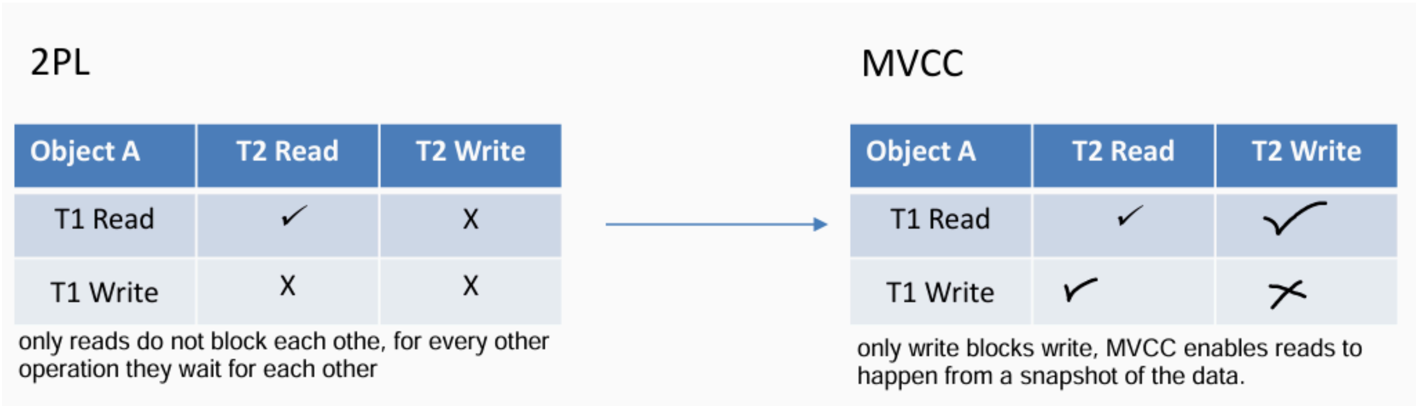
How to **deal with deadlocks in your code**:

- Retry transactions if they fail.
- Keep transactions **short**.
- **Avoid** locking too many resources.

- Don’t leave sessions open with uncommitted changes.
- Don’t overuse `SELECT ... FOR UPDATE`.

Apps should be designed to tolerate occasional rollbacks due to deadlocks.

MVCC (Multiversion Concurrency Control)



2PL

Object A	T2 Read	T2 Write
T1 Read	✓	X
T1 Write	X	X

only reads do not block each othe, for every other operation they wait for each other

MVCC

Object A	T2 Read	T2 Write
T1 Read	✓	✓
T1 Write	✓	✗

only write blocks write, MVCC enables reads to happen from a snapshot of the data.

On **write**, a new version of the row is created.

On **read**, the transaction sees:

- **REPEATABLE READ**: Snapshot of a database from the **start** of the transaction.
- **READ COMMITTED**: Snapshot of a row from the **start of the statement**.

▼ Example

T1 reads values from a table and inserts them into `transaction_log`.

T2 updates the same data concurrently.

T1 reads:

- `REPEATABLE READ`: T1 sees **same** value throughout.
- `READ COMMITTED`: T1 sees **updated value** if T2 commits before T1 reads the second time.

Visibility Rules

For REPEATABLE READ:

1. Ignores writes by active or aborted transactions.
2. Ignores changes from newer transactions (with a higher ID).
3. Only sees committed changes from transactions that started earlier.

For READ COMMITTED:

1. Ignores active transactions.
2. Sees changes from committed transactions at **query execution time**.
3. More relaxed, more up-to-date, but less consistent.

MVCC - Update

Update = delete old version + insert new version

PostgreSQL uses internal fields:

- `Xmin`: Transaction that created the row.
- `Xmax`: Transaction that marked it as deleted.

MVCC - Delete

Deleting a row marks it with `Xmax = transaction ID`.

Until committed:

- The deleting session doesn't see the row.
- Others still see the old version.
- After commit, it's invisible to everyone.

MVCC - writes

Summary of how PostgreSQL tracks version history:

Operation	Created_by (Xmin)	Deleted_by (Xmax)

Insert	Ti	0
Delete	Ti	Td
Update	Ti (old)	Tu (old) → new version created

Garbage Collection:

- PostgreSQL periodically removes old versions using the **VACUUM** process.
- Use `pgstattuple` to check for dead rows.

MVCC – Write – Write Conflict

MVCC doesn't solve **write-write conflicts** on its own. MVCC **alone** is **not serializable**. Needs **additional protocol** to detect/handle conflicts.

▼ Example

T1 writes A → version A1

T2 writes A before T1 commits → conflict