# Analysis and Evaluation of Two-Phase Locking (2PL)

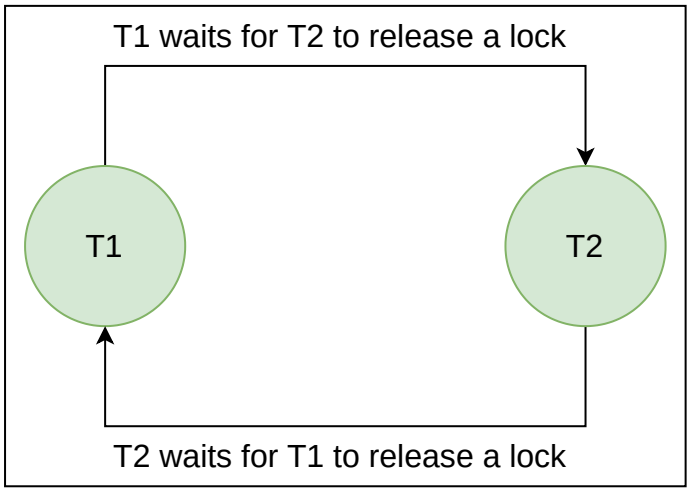
Let's analyze and evaluate the two-phase locking mechanism for concurrency management.

**We'll cover the following**



## Handling of deadlocks

A situation might arise where transaction A is waiting for a long time for transaction B to finish, since we use a lot of locks simultaneously. In general, we refer to the situation where a cycle of transactions waits for one another to release locks as a **deadlock**.

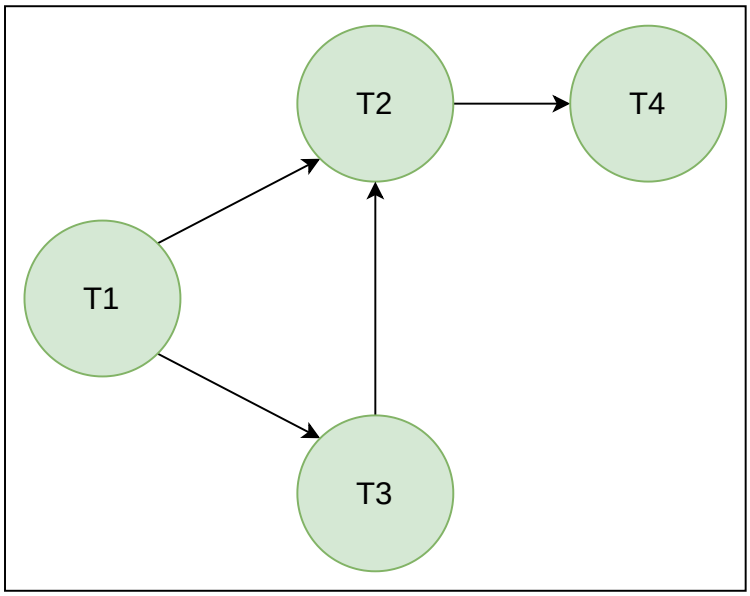



A deadlock

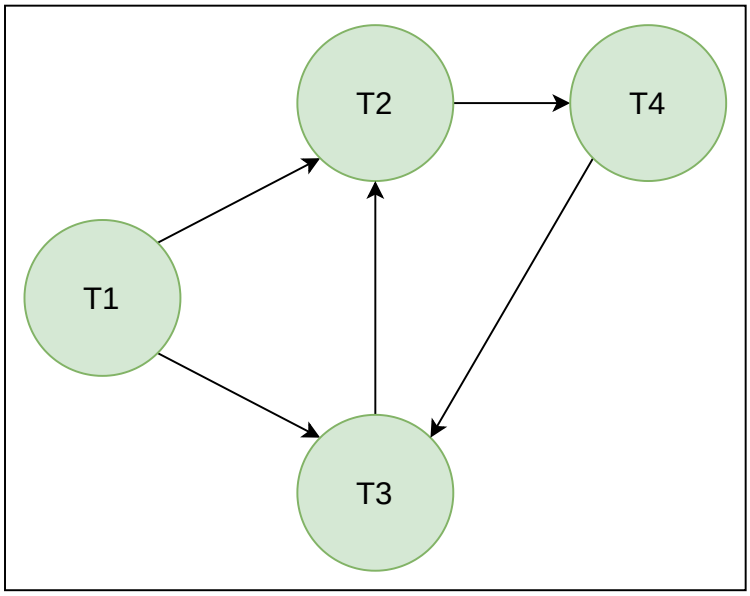
The database automatically identifies deadlocks among the transactions. To resolve them, it relies on two approaches—detection and prevention.

## Deadlocks detection

The database management system (DBMS) produces a waits-for graph, where transactions will form the nodes. Consequently, an edge will form between two nodes, `Ti` and `Tj`, if `Ti` is waiting for the release of a lock held by `Tj`. The system will frequently inspect the waits-for graph for cycles before determining how to break them. The system will have a deadlock only if there is a cycle in the waits-for graph.




Waits-for graph without a cycle—no deadlock




Waits-for graph with a cycle—deadlock



- The DBMS will choose a victim transaction to roll back when it notices a deadlock to break the loop.
- Depending on how the application invoked the transaction, the victim transaction will either restart or abort.
- When choosing a victim, there are many transaction properties to consider. There isn't a decision that is superior to another. All 2PL DBMSs carry out several tasks in the following ways:
    1. By age (newest or oldest timestamp)
    2. In order of query execution progress (least/most)
    3. Based on the number of locked objects
    4. Based on the number of transactions, we need to roll back along with it
    5. The number of times a transaction has previously been restarted

## Deadlock prevention

If another transaction currently holds the lock when a transaction tries to acquire it, we must take some action to avoid a deadlock. In general, to prevent a deadlock,

the lock manager checks whether or not adding a new lock request results in a deadlock. If it does, it simply doesn't execute that transaction.

While waiting for a lock, a transaction can take only one direction—wait or abort. There are some specific schemes to achieve deadlock prevention. They operate based on the timestamps assigned to the transactions, defining priorities, e.g., older means higher priority.

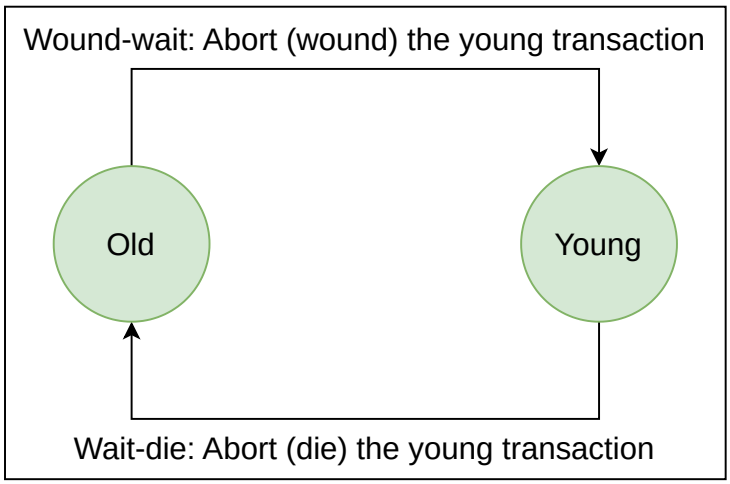
- **Wait-Die ("Old waits for Young")**: If T1 has a higher priority, T1 waits for T2. If not, T1 terminates.

  An older transaction must **wait** before attempting to lock a database object already locked by a younger transaction. A younger transaction **dies** when it attempts to lock a database object that an older transaction has already locked.
- **Wound-Wait ("Young waits for Old")**: If T1 has a higher priority, T2 aborts. If not, T1 waits.

  An older transaction **wounds** (meaning aborts) a younger transaction when attempting to lock a database object already locked by the younger one. A younger transaction **waits** when it attempts to lock a database object that an older transaction has already locked.

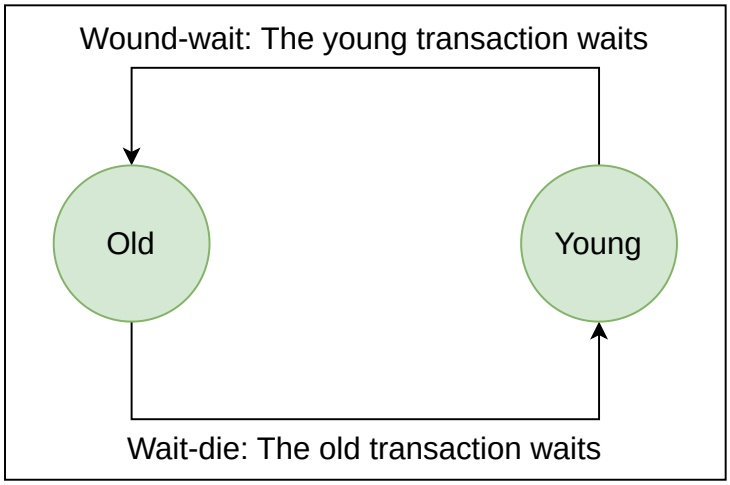
> Wound-wait can provide low latency for high-priority transactions because they don't have to wait for completion for lower-priority transactions. However, it comes at the cost of wasted computation, which happens when an aborting transaction throws away its work and re-does it later.

Only the younger transaction (the one entering the system with a later timestamp) may be terminated and restarted in both scenarios. When a transaction restarts, its old timestamp becomes its (new) priority. This ensures that some transaction is not disadvantaged by always waiting or aborting when there is a flurry of high-priority / old transactions, and gets executed.




Scenario I of deadlock prevention

**1** of 2




Scenario II of deadlock prevention

**2** of 2

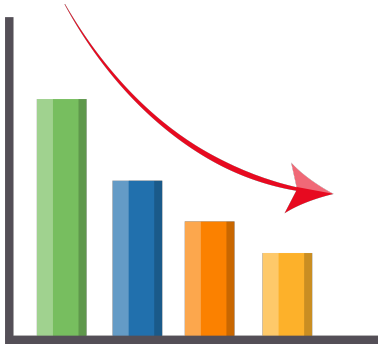
## Shortcomings

Even though 2PL has been in practice for decades and serving well, it has some associated limitations. Let's analyze a few of them.

### Poor throughput and high query response time

In any system's performance, throughput is a prominent indicator that analyzes the system's request-handling capabilities. The adoption rate of 2PL is not as expected, since everyone does not widely use it as a solution to race conditions because of its poor transaction throughput and high query response times. There are many underlying factors to this poor performance. Let's highlight the main ones.

1. **Reduced concurrency:** One of the prominent factors contributing to the poor transaction throughput is **reduced concurrency** where one transaction must wait for the preceding transaction to finish if their actions result in a race condition. It adds distinguished overheads of acquiring and releasing all these locks for the transactions, producing poor throughput performance.



2. **Unlimited transaction time by databases:** By design, the traditional relational databases have transactions with unlimited durations to make the user-interactivity their main priority. As it might lead to scenarios where transactions might have to wait for an extended time period to access the same object, it results in a long queue of unprocessed transactions even if we keep our transactions short. Consequently, a transaction might have to wait a considerable time before others finish their turns, and it can access the desired object.

> **Note:** 2PL helps us achieve serializability but it comes at the cost of lower throughput and higher latency.

### Unstable latencies

The time of every transaction may vary, and it might just take one slow transaction or a transaction that accesses numerous objects and acquire multiple locks simultaneously to clog the whole transaction pipeline. Such situations may lead to unstable latencies in databases running 2PL, which can become very slow at high percentiles if the workload has a lot of variances.

### Frequent deadlocks

Although deadlocks can happen in any other locking mechanism, they are more frequent in 2PL. These deadlocks give rise to additional performance bottlenecks since the application will have to retry the failed transactions from scratch, which is a lot of wasted effort. The higher the frequency of deadlocks, the greater the amount of wasted effort.

## Conclusion