Homework 3.1
Optimistic lock and pessimistic lock
Optimistic Locking is a strategy where you read a record, take note of a version number (other methods to do this involve dates, timestamps or checksums/hashes) and check that the version hasn't changed before you write the record back. When you write the record back you filter the update on the version to make sure it's atomic. (i.e. hasn't been updated between when you check the version and write the record to the disk) and update the version in one hit.
The pessimistic locking model prevents simultaneous updates to records. As soon as one user starts to update a record, a lock is placed on it. Other users who attempt to update this record are informed that another user has an update in progress. The other users must wait until the first user has finished committing their changes, thereby releasing the record lock. Only then can another user make changes based on the previous user's changes.

Homework 3.2
How to solve the deadlock?
Conservative 2-PL
this protocol requires the transaction to lock all the items it access before the Transaction begins execution by predeclaring its read-set and write-set. If any of the predeclared items needed cannot be locked, the transaction does not lock any of the items, instead, it waits until all the items are available for locking.
Wait-Die
In this scheme, If a transaction requests a resource that is locked by another transaction, then the DBMS simply checks the timestamp of both transactions and allows the older transaction to wait until the resource is available for execution.

Wound Wait
In this scheme, if an older transaction requests for a resource held by a younger transaction, then an older transaction forces a younger transaction to kill the transaction and release the resource. The younger transaction is restarted with a minute delay but with the same timestamp. If the younger transaction is requesting a resource that is held by an older one, then the younger transaction is asked to wait till the older one releases it.

Homework 3.3
Saga design pattern

The Saga design pattern is a way to manage data consistency across microservices in distributed transaction scenarios. A saga is a sequence of transactions that updates each service and publishes a message or event to trigger the next transaction step. If a step fails, the saga executes compensating transactions that counteract the preceding transactions.

SAGA vs. 2PC

2PC works as a single commit and aims to perform ACID transactions on distributed systems. It is used wherever strong consistency is important. On the other hand, SAGA works sequentially, not as a single commit. Each operation gets committed before the subsequent one, and this makes the data eventually consistent. Thus, Saga consists of multiple steps whereas 2PC acts like a single request.