Assignment 3.1

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.
If the record is dirty (i.e. a different version to yours) you abort the transaction and the user can restart it.

This strategy is most applicable to high-volume systems and three-tier architectures where you do not necessarily maintain a connection to the database for your session. In this situation the client cannot actually maintain database locks as the connections are taken from a pool and you may not be using the same connection from one access to the next.

Optimistic locking is used when you don't expect many collisions. It costs less to do a normal operation but if the collision DOES occur you would pay a higher price to resolve it as the transaction is aborted.

Pessimistic Locking is when you lock the record for your exclusive use until you have finished with it. It has much better integrity than optimistic locking but requires you to be careful with your application design to avoid Deadlocks. To use pessimistic locking you need either a direct connection to the database (as would typically be the case in a two tier client server application) or an externally available transaction ID that can be used independently of the connection.
In the latter case you open the transaction with the TxID and then reconnect using that ID. The DBMS maintains the locks and allows you to pick the session back up through the TxID. This is how distributed transactions using two-phase commit protocols (such as XA or COM+ Transactions) work.

Pessimistic locking is used when a collision is anticipated. The transactions which would violate synchronization are simply blocked.

To select a proper locking mechanism you have to estimate the amount of reads and writes and plan accordingly.

Reference:
https://stackoverflow.com/questions/129329/optimistic-vs-pessimistic-locking

Assignment 3.2
Conservative 2-PL A.K.A Static 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.

Conservative 2-PL is Deadlock free and but it does not ensure a Strict schedule(More about this here!). However, it is difficult to use in practice because of the need to predeclare the read-set and the write-set which is not possible in many situations. In practice, the most popular variation of 2-PL is Strict 2-PL.

Wait-Die Scheme –
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.

Suppose, there are two transactions T1 and T2, and Let the timestamp of any transaction T be TS (T). Now, If there is a lock on T2 by some other transaction and T1 is requesting for resources held by T2, then DBMS performs the following actions:

Checks if TS (T1) < TS (T2) – if T1 is the older transaction and T2 has held some resource, then it allows T1 to wait until resource is available for execution. That means if a younger transaction has locked some resource and an older transaction is waiting for it, then an older transaction is allowed to wait for it till it is available. If T1 is an older transaction and has held some resource with it and if T2 is waiting for it, then T2 is killed and restarted later with random delay but with the same timestamp. i.e. if the older transaction has held some resource and the younger transaction waits for the resource, then the younger transaction is killed and restarted with a very minute delay with the same timestamp.

This scheme allows the older transaction to wait but kills the younger one.

Wound Wait Scheme –
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.

Livelock occurs when two or more processes continually repeat the same interaction in response to changes in the other processes without doing any useful work. These processes are not in the waiting state, and they are running concurrently. This is different from a deadlock because in a deadlock all processes are in the waiting state.

A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing. Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.

Reference:
https://www.geeksforgeeks.org/deadlock-in-dbms/#:~:text=Wait%2DDie%20Scheme%20%E2%80%93,resource%20is%20available%20for%20execution.
https://www.geeksforgeeks.org/categories-of-two-phase-locking-strict-rigorous-conservative/
https://www.geeksforgeeks.org/deadlock-starvation-and-livelock/

Assignment 3.3

Saga distributed transactions 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.

The Saga pattern provides transaction management using a sequence of local transactions. A local transaction is the atomic work effort performed by a saga participant. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails, the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.

In Saga patterns:

- Compensable transactions are transactions that can potentially be reversed by processing another transaction with the opposite effect.
- A pivot transaction is the go/no-go point in a saga. If the pivot transaction commits, the saga runs until completion. A pivot transaction can be a transaction that is neither compensable nor retryable, or it can be the last compensable transaction or the first retryable transaction in the saga.
- Retryable transactions are transactions that follow the pivot transaction and are guaranteed to succeed.

There are two common saga implementation approaches, choreography and orchestration.

Choreography

Choreography is a way to coordinate sagas where participants exchange events without a centralized point of control. With choreography, each local transaction publishes domain events that trigger local transactions in other services.

Pros	Cons
Good for simple workflows that require few participants and don't need a coordination logic.	Workflow can become confusing when adding new steps, as it's difficult to track which saga participants listen to which commands.
Doesn't require additional service implementation and maintenance.	There's a risk of cyclic dependency between saga participants because they have to consume each other's commands.
Doesn't introduce a single point of failure, since the responsibilities are distributed	Integration testing is difficult because all services must be running to simulate a

across the saga participants.	transaction.

Orchestration

Orchestration is a way to coordinate sagas where a centralized controller tells the saga participants what local transactions to execute. The saga orchestrator handles all the transactions and tells the participants which operation to perform based on events. The orchestrator executes saga requests, stores and interprets the states of each task, and handles failure recovery with compensating transactions.

Pros	Cons
Good for complex workflows involving many participants or new participants added over time.	Additional design complexity requires an implementation of a coordination logic.
Suitable when there is control over every participant in the process, and control over the flow of activities.	There's an additional point of failure, because the orchestrator manages the complete workflow.
Doesn't introduce cyclical dependencies, because the orchestrator unilaterally depends on the saga participants.	
Saga participants don't need to know about commands for other participants. Clear separation of concerns simplifies business logic.	

Use the Saga pattern when you need to:

- Ensure data consistency in a distributed system without tight coupling.
- Roll back or compensate if one of the operations in the sequence fails.

The Saga pattern is less suitable for:

- Tightly coupled transactions.
- Compensating transactions that occur in earlier participants.
- Cyclic dependencies.

Reference:
https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga