8.1 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.

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.

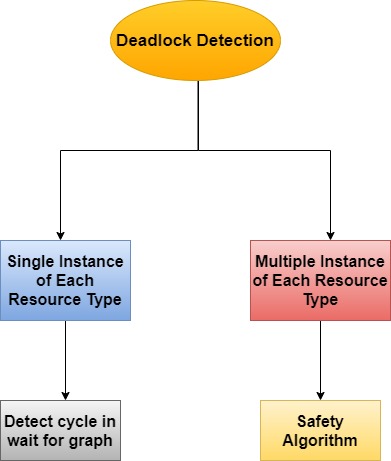
8.2

transaction -

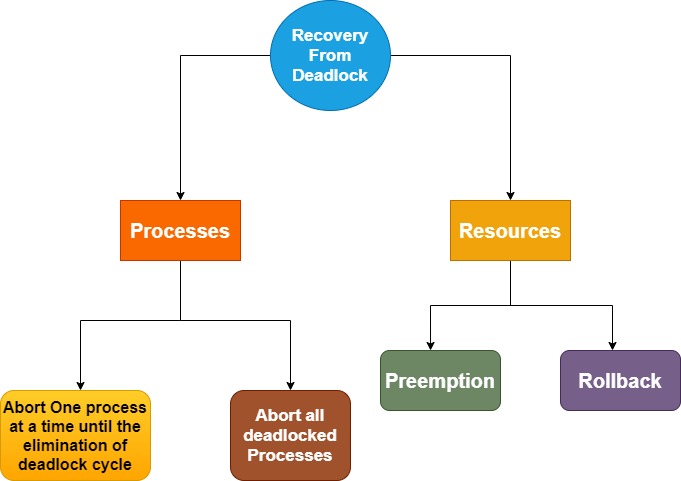
A transaction is an action, or a series of actions, carried out by a single user or an application program. Following ACID.

how to solve the deadlock -

First of all, we should detect deadlock, the process below:



Then, we can follow the process to recovery from deadlock:



live lock -

A Livelock is a situation where a request for an exclusive lock is denied repeatedly, as many overlapping shared locks keep on interfering each other. The processes keep on changing their status, which further prevents them from completing the task. This further prevents them from completing the task.

8.3

Saga -

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.

2PC (two phase commit) –

The two-phase commit protocol breaks a database commit into two phases to ensure correctness and fault tolerance in a distributed database system.

Prepare phase

in which a coordinator process attempts to prepare all the transaction's participating processes (named participants, cohorts, or workers) to take the necessary steps for either committing or aborting the transaction and to vote, either "Yes": commit (if the transaction participant's local portion execution has ended properly), or "No": abort (if a problem has been detected with the local portion)

Commit phase

in which, based on voting of the participants, the coordinator decides whether to commit (only if all have voted "Yes") or abort the transaction (otherwise), and notifies the result to all the participants. The participants then follow with the needed actions (commit or abort) with their local transactional resources (also called recoverable resources; e.g., database data) and their respective portions in the transaction's other output (if applicable).