Optimistic lock

Optimistic lock allows multiple users to attempt to update the record. The record changes are validated only when the record is committed. If one user successfully updates the record, the other users attempting to commit their concurrent updates are informed that a conflict exists.

The advantage of optimistic lock is that it avoids the overhead of locking a record for the duration of the actions. If there are no simultaneous updates, then this model provides fast updates. Optimistic lock is a useful approach when concurrent record updates are expected to be infrequent, or the locking overhead is high.

Pessimistic lock

Pessimistic lock prevents simultaneous updates to records. As soon as one user starts to update a record, a lock is placed to it. Other users who attempt to update this record are informed that another use 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 use make changes based on the previous user’s changes.

The advantage of the pessimistic lock is that it avoids the issue of conflict resolution by preventing conflicts. Updates are serialized and each subsequent update start with the committed record changes from the previous user. Pessimistic lock is a useful approach when subsequent updates can be delayed until a previous update is completed. This usually implies that updates occur in a short time interval.

Transaction

A transaction is defined as a group of tasks. A single task is the minimum processing unit which cannot be divided further.

A transaction is a very small unit of a program and it may contain several lowlevel tasks. A transaction in a database system must maintain the ACID properties to ensure accuracy, completeness, and data integrity.

When multiple transactions are begin executed by the operation system in a multiprogramming environment, there are possibilities that instructions of one transaction are interleaved with some other transaction. A chronological execution sequence of a transaction is called a schedule. A schedule can have many transactions in it, each comprising a number of instructions/tasks. A schedule in which transactions are aligned in such a way that one transaction is executed first is serial schedule. When the first transaction completes its cycle, then the next transaction is executed. Transactions are ordered one after the other.

In a multi-transaction environment, serial schedules are considered as a benchmark. The execution sequence of an instruction in a transaction cannot be changed, but two transactions can have their instructions executed in a random fashion. The execution does no harm if two transactions are mutually independent and working on different segments of data; but in case these two transactions are working on the same data, then the results may vary. This ever-varying result may bring the database to an inconsistent state. To resolve this problem, we allow parallel execution of a transaction schedule, if its transactions are either serializable or have some equivalence relation among them.

If two schedule produce the same result after execution, they are said to be result equivalent. They may yield the same result for some value and different results for another set of values. That’s why this equivalence is not generally considered significant.

If the transactions in both the schedules perform similar actions in a similar manner, the two schedules would be view equivalence.

Two schedules would be conflicting if they have following properties: both belong to separate transactions, both accesses the same data item, at least one of them is “write” operation. Two schedules having multiple transactions with conflicting operations are said to be conflict equivalent if and only if: both the schedules contain the same set of Transactions, the order of conflicting pairs of operation is maintained in both the schedules.

A transaction can be in one of the following states: active, partially committed, failed, aborted, and committed.

Active: in this state, transaction is being executed. This is the initial state of every transaction

Partially committed: When a transaction executes its final operation, it is said to be in partially committed state.

Failed: A transaction is said to be in failed state if any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further.

Aborted: If any of the checks fails and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state where it was prior to the execution of the transaction. Transactions in this state are called aborted. The database recovery module can select one of the two operations after a transaction aborts: re-start the transaction and kill the transaction.

Committed: If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.

Deadlock prevention:

Eliminate hold and wait: Allocate call required resources to the process before the start of its execution, this way hold and wait condition is eliminated but it will lead to low device utilization.

Eliminate No Preemption

Preempt resources from the process when resources required by other high priority processes.

Eliminate Circular wait

Each resource will be assigned with a numerical number. A process can request the resources increasing/decreasing order of numbering.

Banker’s algorithm

Banker’s algorithm is used to avoid deadlock. Banker’s algorithm is a resource allocation and deadlock avoidance algorithm which test all the request made by processes for resources, if checks for the safe state, if after granting request system remains in the safe state is allows the request and if there is no safe state it doesn’t allow the request made by the process.

Live lock

A livelock is a situation where a request for an exclusive lock is denied repeatedly, as many overlapping shared lock keep on interfering each other. The processes keep on changing their status, which further prevents them from completing the task. Livelock occurs when the total number of allowed processes in a specific system should be defined by the total number of entries in the process table. Therefore, process table slots should be referred to as Finite Resources.