UNIT=9

Concurrency Control Techniques

Concurrency control is the procedure in DBMS for managing simultaneous operations without conflicting with eachother. Different concurrency control protocols/techniques offer different benefits between the amount of concurrency they allow and the overhead they impose.

1) Two-Phase Locking Technique:

for concept, if felt lengthy can be escaped.

A lock is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the database.

Locking is an operation which secures: permission to read, or permission to write a data item. Two phase locking is a process used to gain ownership of shared resources without creating the possibility of deadlock. The 3 activities taking place in the two phase update algorithm are:-

→ lock acquisition
→ Modification of data
→ Release lock

Two phase locking prevents deadlock from occuring in distributed systems by releasing all the resources it has acquired, if it is not possible to acquire all the resources required without waiting for another process to finish using a lock. A transaction in the Two Phase Locking Protocol can assume one of the two phases:

i) Growing Phase → In this phase a transaction can only acquire locks but cannot release any lock. The point when a transaction acquires all the locks it needs is called the lock Point.

ii) Shrinking Phase → In this phase a transaction can only release locks but cannot acquire any.

The two main modes in which a data item may be locked are;

i) exclusive (X) mode → Data item can be both read as well as written.

ii) Shared (S) mode → Data item can only be read.

	S	X
S	True	False
X	False	False

Fig. Lock-compatibility Matrix

⊛ Types of locks and System lock tables:

i) Binary Locks → A binary lock can have two states or values: locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item X. If the value of the lock on X is 1, item X cannot be accessed by a database operation that requests the item. If the value of the lock on X is 0, the item can be accessed when requested, and the lock value is changed to 1. We refer to the current state of the lock associated with item X as lock(X). Two operations, lock_item(X) and unlock_item(X) are used with binary locking.

ii) Shared/Exclusive (or Read/Write) Locks → In this lock there are three locking operations: read_lock(X), write_lock(X) and unlock(X). A lock associated with an item X, LOCK(X), now has three possible states: read-locked, write-locked, or unlocked. A read-locked item is also called share-locked because other transactions are allowed to read the item, whereas a write-locked item is called exclusive-locked because a single transaction exclusively holds the lock on the item.

⊛. Conversion (Upgrading, Downgrading) of Locks

A transaction that already holds a lock on item X is allowed under certain conditions to convert the lock from one locked state to another. For example, it is possible for a transaction T to issue a read_lock(X) and then later to upgrade the lock by issuing a write_lock(X) operation. Similarly it is also possible for a transaction T to issue a write_lock(X) and then later to downgrade the lock by issuing a read_lock(X) operation. When upgrading and downgrading of locks is used, the lock table must include transaction identifiers in the record structure for each lock to store the information on which transactions hold locks on the item.

⊛. Guaranteeing Serializability by Two-Phase Locking:

A transaction is said to follow the two-phase locking protocol if all locking operations (read_lock, write_lock) precede the first unlock operation in the transaction. Such a transaction can be divided into two phases: an expanding or growing phase, during which new locks on items can be acquired but none can be released; and a shrinking phase during which existing locks can be released but no new locks can be acquired. If lock conversion is allowed, then upgrading of locks must be done during the expanding phase and downgrading of locks must be done in the shrinking phase.

⊛. Basic, Conservative, Strict and Rigorous Two-Phase Locking:

OR Categories of two-phase locking (2-PL):

i) Basic 2-PL → Two-phase locking that we studied before is the basic 2-PL.

ii) Strict 2-PL → This requires in addition to basic 2-PL that all Exclusive (X) locks held by the transaction be released until after the transaction commits. Following strict 2-PL ensures that our schedule is: Recoverable and Cascadeless. Hence, it gives us freedom from Cascading Abort which was in Basic 2-PL, but still deadlocks are possible.

iii) Rigorous 2-PL → This requires in addition to basic 2-PL that all Exclusive (X) and Shared (S) locks held by the transaction be released until after the transaction commits. The difference between Strict 2-PL and Rigorous 2-PL is that, Rigorous is more restrictive, it requires both Exclusive and Shared locks to be held until after transaction commits.

iv) Conservative 2-PL → This requires 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.

⊗. Dealing with Deadlock and Starvation:

Deadlock and Starvation both are the conditions where the processes requesting for a resource has been delayed for a long. Although deadlock and starvation both are different from each other in many aspects. Deadlock is a condition where no process proceeds for execution and each waits for resources that have been acquired by the other processes. On the other hand, Starvation is a condition where process with higher priorities continuously uses the resources preventing low priority process to acquire the resources.

i.e. Handling

Dealing with Deadlock:

One way to prevent deadlock is to use a deadlock prevention protocol. A number of other deadlock prevention schemes have been proposed. Some of these techniques use the concept of transaction timestamp, which is a unique identifier assigned to each transaction. An alternative approach to deal with deadlock is deadlock detection, where the system checks if a state of deadlock actually exists. This solution is attractive if different transactions will rarely access the same items at the same time. Timeouts is the another scheme to deal with deadlock. This method is practical because of its low overhead and simplicity. In this method, if a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it — regardless of whether a deadlock actually exists.

Dealing with Starvation: One solution for starvation is to have a fair waiting scheme, such as using a first-come-first-served queue; where transactions are enabled to lock an item in order in which they originally requested to lock. Another scheme allows some transactions to have higher priority over others. The wait-die and wound-wait avoid starvation, because they restart a transaction that have been aborted.

Deadlock vs Starvation

Deadlock	Starvation
i) Deadlock is a condition where no process proceeds for execution and each ~~waits~~ waits for resources that have been acquired by other processes.	i) Starvation is a condition where process with higher priorities continiously uses the resources preventing low priority process to acquire the resources.
ii) It happens if two or more transaction is waiting for each other.	ii) It happens if the waiting scheme for locked items is in unfair.
iii) Avoidance: ~~→Switch priorities so that every~~ → Acquire locks at once before starting. →Acquire locks with predefined order.	iii) Avoidance: →Switch priorities so that every thread has a chance to have high priority. → Use FIFO order among competing request.
iv) Deadlock is also known as circular waiting.	iv) Starvation is also known as lived lock.
v) Resources are blocked by the processes.	v) Resources are continiously utilized by high priority processes.

2) Timestamp Ordering:

A timestamp is a unique identifier created by the DBMS to identify a transaction. Timestamp Ordering is a schedule in which the transactions participate is then serializable, and the only equivalent serial schedule permitted has the transactions in order of their timestamp values. A timestamp can be implemented in 2 ways. One is to directly assign the current value of the clock to the transaction or data item. The other is to attach the value of a logical counter that keeps increment as new timestamps are required. The timestamp of a data item can be of 2 types:

i) W-timestamp(X) → This means the latest time when the data item X has been written into.

ii) R-timestamp(X) → This means the latest time when the data item X has been read from.

*. Basic Timestamp Ordering:

Whenever some transaction T tries to issue a read_item(X) or a write_item(X) operation, then this algorithm compares the timestamp of T with R_TS(X) and W_TS(X) to ensure that the timestamp order of transaction execution is not violated. If this order is violated, then transaction T is aborted and resubmitted to the system as a new transaction with a new timestamp.

1. Check the following condition whenever a transaction T_i issues a read_item(X) operation:
   → If W_TS(X) > TS(T_i) then the operation is rejected.
   → If W_TS(X) <= TS(T_i) then the operation is executed.
   → Timestamps of all the data items are updated.
2. Check the following condition whenever a transaction T_i issues a write_item(X) operation:
   → If TS(T_i) < R_TS(X) then the operation is rejected.
   → If TS(T_i) < W_TS(X) then the operation is rejected and T_i is rolled back otherwise the operation is executed.

> where,
> TS(T_i) denotes the timestamp of transaction T_i.
> R_TS(X) denotes the Read timestamp of data-item X.
> W_TS(X) denotes the Write timestamp of data-item X.

*. Strict Timestamp Ordering:

A variation of basic TO called strict TO (timestamp ordering) ensure that the schedules are both strict (for easy recoverability) and (conflict) serializable.

1. Transaction T issues a write_item(X) operation:
   → If TS(T) > R_TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).
2. Transaction T issues a read_item(X) operation:
   → If TS(T) > W_TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

⊛. Thoma's Write Rule:

A modification of a basic TO algorithm, known as Thoma's write rule, does not enfore conflict serializability, but it rejects fewer write operations by modifying the checks for the write_item(X) operation as follows;

1. If $R\_TS(X) > TS(T)$ then abort and roll-back T and reject the operation.
2. If $W\_TS(X) > TS(T)$, then just ignore the write operation and continue execution. This is because the most recent writes counts in case of two consecutive writes.
3. If the conditions given in 1 and 2 above do not occur, then execute write_item(X) of T and set $W\_TS(X)$ to $TS(T)$.

3. Multiversion Concurrency Control:

These concurrency control keep copies of the old values of a data item when the item is updated. They are known as multiversion concurrency control because several versions (values) of an item are kept by the system. When a transaction requests to read an item, the appropriate version is chosen to maintain the serializability of the currently executing schedule. When a transaction writes an item, it writes a new version and the old version(s) of the item is retained. An obvious drawback of multiversion techniques is that more storage is needed to maintain multiple versions of the database items.

⊛ Multiversion technique based on timestamp ordering:

In this method, several versions $X_1, X_2, \dots, X_K$ of each data item X are maintained. For each version, the value of version X_i and the following two timestamps associated with version X_i are kept:

i) read_TS(X_i) → The read timestamp of X_i is the largest of all the timestamps of transactions that have successfully read version X_i.

ii) write_TS(X_i) → The write timestamp of X_i is the timestamp of the transaction that wrote the value of version X_i.

To ensure serializability, the following rules are used:

i) If transaction T issues a write_item(X) operation, and version i of X has the highest write_TS(X_i) of all versions of X that is also less than or equal to TS(T), and read_TS(X_i) > TS(T), then abort and roll back transaction T; otherwise, create a new version X_j of X with read_TS(X_j) = write_TS(X_j) = TS(T).

ii) If transaction T issues a read_item(X) operation, find the version i of X that has the highest write_TS(X_i) of all versions of X that is also less than or equal to TS(T); then return the value of X_i to transaction T, and set the value of read_TS(X_i) to the larger of TS(T) and the current read_TS(X_i).

⊛. Multiversion locking using certify locks:- (OR Multi-version two-phase locking)

In this scheme, there are three locking modes for an item; read, write and certify. Hence the state of LOCK(X) for an item X can be one of read-locked, write-locked, certify-locked or unlocked. The idea behind multiversion 2PL is to allow other transactions T to read an item X while a single transaction T holds a write lock on X. This is accomplished by allowing two versions for each item X; one, the committed version, must always have been written by some committed transaction. The second local version X' can be created when a transaction T acquires a write lock on X.

Once T is ready to commit, it must obtain a certify lock on all items that it currently holds write locks on before it can commit. The certify lock is not compatible with read locks, so the transaction may have to delay it's commit until all its write-locked items are released by any reading transactions in order to obtain the certify locks. Once the certify locks are acquired, the committed version X of the data item is set to the value of version X, version X is discarded, and the certify locks are then released.

Scanned with CamScanner

4. Validation (Optimistic) Techniques:

The optimistic approach is based on the assumption that the majority of the database operations do not conflict. The optimistic approach requires neither locking nor time stamping techniques. Instead, a transaction is executed without restrictions until it is committed. Using an optimistic approach, each transaction moves through 2 or 3 phases, referred to as read, validation and write.

i) Read phase → During read phase, the transaction reads the database, executes the needed computations and makes the updates to a private copy of the database values. All update operations of the transactions are recorded in a temporary update file, which is not accessed by the remaining transactions.

ii) Validation phase → During validation phase, the transaction is validated to ensure that the changes made will not affect the integrity and consistency of the database. If the validation test is positive, the transaction goes to write phase. If the validation test is negative, the transaction is restarted and the changes are discarded.

iii) Write phase → During write phase, the changes are permanently applied to the database.

⊛. Snapshot Isolation Concurrency Control:

In database and transaction processing, snapshot isolation is a guarantee that all reads made in a transaction will see a consistent snapshot of the database and the transaction itself will successfully commit only if no updates it has made conflict with any updates made since that snapshot.

Snapshot isolation has been adopted by several major database management systems, such as SQL, Oracle, MongoDB, Postgree SQL etc. The main reason for it's adoption is that it allows better performance than serializability. In practice snapshot isolation is implemented within multiversion concurrency control (MVCC), where generational values of each data item (versions) are maintained.