Lecture 46

Multiversion Concurrency Control Techniques

Multiversion Concurrency Control Technique

- These protocols for concurrency control keep copies of the old values of a data item when the item is updated (written);
- 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.
- One reason for keeping multiple versions is that some read operations that would be rejected in other techniques can still be accepted by reading an *older version* of the item to maintain serializability.
- When a transaction writes an item, it writes a *new version* and the old version(s) of the item is retained.
- Some multiversion concurrency control algorithms use the concept of view serializability rather than conflict serializability.

Multiversion Concurrency Control Technique

- An obvious drawback of multiversion techniques is that more storage is needed to maintain multiple versions of the database items.
- In some cases, older versions can be kept in a temporary store.
- It is also possible that older versions may have to be maintained anyway—for example, for recovery purposes.
- Some database applications may require older versions to be kept to maintain a history of the changes of data item values.
- The extreme case is a *temporal database*, which keeps track of all changes and the times at which they occurred.
- In such cases, there is no additional storage penalty for multiversion techniques, since older versions are already maintained.

Multiversion Timestamp Ordering Protocol

- In this method, several versions X_1, X_2, … , 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:
    1. **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.
    2. **write_TS(X_i)** → The **write timestamp** of X_i is the timestamp of the transaction that wrote the value of version X_i.
- Whenever a transaction T is allowed to execute a write_item(X) operation, a new version X_{k+1} of item X is created, with both the write_TS(X_{k+1}) and the read_TS(X_{k+1}) set to TS(T). Correspondingly, when a transaction T is allowed to read the value of version
- X_i, the value of read_TS(X_i) is set to the larger of the current read_TS(X_i) and TS(T).

Multiversion Timestamp Ordering Protocol

- To ensure serializability, the following rules are used:
  1. 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).
  2. 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 Timestamp Ordering Protocol

- As we can see in case 2, a read_item(X) *is always successful*, since it finds the appropriate version X_i to read based on the write_TS of the various existing versions of X.
- In case 1, however, transaction T may be aborted and rolled back. This happens if T attempts to write a version of X that should have been read by another transaction T' whose timestamp is read_TS(X_i); however, T' has already read version Xi, which was written by the transaction with timestamp equal to write_TS(X_i). If this conflict occurs, T is rolled back; otherwise, a new version of X, written by transaction T, is created.
- Note that if T is rolled back, cascading rollback may occur. Hence, to ensure recoverability, a transaction T should not be allowed to commit until after all the transactions that have written some version that T has read have committed.

Multiversion Two-Phase Locking Using Certify Locks

- In this multiple-mode locking scheme, there are *three locking modes* for an item— read, write, and *certify*—instead of just the two modes (read, write).
- Hence, the state of LOCK(*X*) for an item *X* can be one of read-locked, write-locked, certify-locked, or unlocked.
- In the standard locking scheme, with only read and write locks, a write lock is an exclusive lock. We can describe the relationship between read and write locks in the standard scheme by means of the **lock compatibility table** shown below.

	Read	**Write**
Read	**Yes**	**No**
Write	**No**	**No**

Multiversion Two-Phase Locking Using Certify Locks

- An entry of *Yes* means that if a transaction *T* holds the type of lock specified in the column header on item *X* and if transaction *T′* requests the type of lock specified in the row header on the same item *X*, then *T′ can obtain the lock* because the locking modes are compatible.
- On the other hand, an entry of *No* in the table indicates that the locks are not compatible, so *T′ must wait* until *T releases* the lock.

	Read	**Write**
Read	**Yes**	**No**
Write	**No**	**No**

Multiversion Two-Phase Locking Using Certify Locks

- The lock compatibility table for this scheme is shown in Figure

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

Multiversion Two-Phase Locking Using Certify Locks

- In the standard locking scheme, once a transaction obtains a write lock on an item, no other transactions can access that item.
- 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 version, 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*. Other transactions can continue to read the *committed version* of *X* while *T* holds the write lock.

Multiversion Two-Phase Locking Using Certify Locks

- Transaction *T* can write the value of *X*′ as needed, without affecting the value of the committed version *X*. However, 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; this is another form of **lock upgrading**.
- The certify lock is not compatible with read locks, so the transaction may have to delay its commit until all its write-locked items are released by any reading transactions in order to obtain the certify locks.
- Once the certify locks—which are exclusive 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.

For Video lecture on this topic please subscribe to my youtube channel.

The link for my youtube channel is

https://www.youtube.com/channel/UCRWGtE76JlTp1iim6aOTRuw?sub_confirmation=1