5.9 MULTIVERSION CONCURRENCY CONTROL TECHNIQUES

- 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.
- Drawback of multiversion techniques: 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.

1. Multiversion Technique Based on Timestamp Ordering:

- In this method, several versions $X_1, X_2, \ldots, X_k$ 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)`.
- 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(Xi) > 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`)`.

A `read_item(X)` is always successful as seen in case 2, 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 X_i, 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. 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.

2. Multiversion Two-Phase Locking Using Certify Locks:

- In the multiple-mode locking 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.
- In the standard locking scheme, with only read and write locks, a write lock is an exclusive lock. The relationship between read and write locks in the standard scheme can be described by means of the ***lock compatibility table*** shown in Figure 19(a).

(a)

	Read	Write
Read	Yes	No
Write	No	No

(b)

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

Figure 19: Lock compatibility tables. (a) Lock compatibility table for read/write locking scheme. (b) Lock compatibility table for read/write/certify locking scheme.

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. An entry of `No` in the table indicates that the locks are not compatible, so T′ must wait until T releases the lock. In the standard locking scheme, once a transaction obtains a write lock on an item, no other transactions can access that item.

- Multiversion 2PL will 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-
  - 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. Transaction T can write the value of X′ as needed, without affecting the value of the committed version X. But 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. The lock compatibility table for this scheme is shown in Figure 19(b).

- In the multiversion 2PL scheme, reads can proceed concurrently with a single write operation which is not permitted under the standard 2PL schemes. The cost is that a transaction may have to delay its commit until it obtains exclusive certify locks on all the items it has updated.
- This scheme avoids cascading aborts, since transactions are only allowed to read the version X that was written by a committed transaction. But deadlocks may occur and must be handled by various techniques.

5.10 VALIDATION (OPTIMISTIC) TECHNIQUES AND SNAPSHOT ISOLATION CONCURRENCY CONTROL

In locking, a check is done to determine whether the item being accessed is locked. In timestamp ordering, the transaction timestamp is checked against the read and write timestamps of the item. Such checking represents overhead during transaction execution, with the effect of slowing down the transactions. In **optimistic concurrency control techniques**, also known as ***validation or certification techniques***, no checking is done while the transaction is executing. Several concurrency control methods are based on the

validation technique. The implementations of these concurrency control methods can utilize a combination of the concepts from validation-based techniques and versioning techniques, as well as utilizing timestamps. Some of these methods may suffer from anomalies that can violate serializability, but because they generally have lower overhead than 2PL, they have been implemented in several relational DBMSs.

1. Validation-Based (Optimistic) Concurrency Control:

- Updates in the transaction are not applied directly to the database items on disk until the transaction reaches its end and is validated.
- During transaction execution, all updates are applied to local copies of the data items that are kept for the transaction. At the end of transaction execution, a validation phase checks whether any of the transaction's updates violate serializability. Certain information needed by the validation phase must be kept by the system. If serializability is not violated, the transaction is committed and the database is updated from the local copies; otherwise, the transaction is aborted and then restarted later.
- There are three phases for this concurrency control protocol:
  1. Read phase: A transaction can read values of committed data items from the database. However, updates are applied only to local copies (versions) of the data items kept in the transaction workspace.
  2. Validation phase: Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.
  3. Write phase: If the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the transaction is restarted.

  The optimistic concurrency control does all the checks at once. Hence transaction execution proceeds with a minimum of overhead until the validation phase is reached. If there is little interference among transactions, most will be validated successfully. If there is much interference, many transactions that execute to completion will have their results discarded and must be restarted later. Under such circumstances, optimistic techniques do not work well.

- The techniques are called optimistic because they assume that little interference will occur and hence most transaction will be validated successfully, so that there is no need to do checking during transaction execution. This assumption is generally true in many transaction processing workloads.
- The optimistic protocol described uses transaction timestamps and also requires that the `write_sets` and `read_sets` of the transactions be kept by the system. Additionally, start and end times for the three phases need to be kept for each transaction.

- In the validation phase for transaction T_i, the protocol checks that T_i does not interfere with any recently committed transactions or with any other concurrent transactions that have started their validation phase. The validation phase for T_i checks that, for each such transaction T_j that is either recently committed or is in its validation phase, one of the following conditions holds:
  1. Transaction T_j completes its write phase before T_i starts its read phase.
  2. Ti starts its write phase after T_j completes its write phase, and the `read_set` of T_i has no items in common with the `write_set` of T_j.
  3. Both the `read_set` and `write_set` of T_i have no items in common with the `write_set` of T_j, and T_j completes its read phase before T_i completes its read phase.

  When validating transaction T_i against each one of the transactions T_j, the first condition is checked first since (1) is the simplest condition to check. Only if condition 1 is false condition 2 checked, and only if (2) is false condition 3 is checked. If any one of these three conditions holds with each transaction T_j, there is no interference and Ti is validated successfully. If none of these three conditions holds for any one T_j, the validation of transaction T_i fails (because T_i and T_j may violate serializability) and so T_i is aborted and restarted later because interference with T_j may have occurred.

2. Concurrency Control Based on Snapshot Isolation:

- In snapshot isolation a transaction sees the data items that it reads based on the committed values of the items in the database snapshot (or database state) when the transaction starts.
- Snapshot isolation will ensure that the phantom record problem does not occur, since the database transaction, or, in some cases, the database statement, will only see the records that were committed in the database at the time the transaction started. Any insertions, deletions, or updates that occur after the transaction starts will not be seen by the transaction.
- Snapshot isolation does not allow the problems of dirty read and nonrepeatable read to occur. Certain anomalies that violate serializability can occur when snapshot isolation is used as the basis for concurrency control.
- In this scheme, read operations do not require read locks to be applied to the items, thus reducing the overhead associated with two-phase locking. But write operations do require write locks. Thus, for transactions that have many reads, the performance is much better than 2PL. When writes do occur, the system will have to keep track of older versions of the updated items in a **temporary version store** (also known as tempstore), with the timestamps of when the version was created. This is necessary so that a transaction that started before the item was written can still read the value (version) of the item that was in the database snapshot when the transaction started.

- To keep track of versions, items that have been updated will have pointers to a list of recent versions of the item in the tempstore, so that the correct item can be read for each transaction. The tempstore items will be removed when no longer needed, so a method to decide when to remove unneeded versions will be needed.
- Variations of this method have been used in several commercial and open source DBMSs, including Oracle and PostGRES. If the users require guaranteed serializability, then the problems with anomalies that violate serializability will have to be solved by the programmers/software engineers by analyzing the set of transactions to determine which types of anomalies can occur, and adding checks that do not permit these anomalies. This can place a burden on the software developers when compared to the DBMS enforcing serializability in all cases.
- Variations of snapshot isolation (SI) techniques, known as serializable snapshot isolation (SSI), have been proposed and implemented in some of the DBMSs that use SI as their primary concurrency control method. For example, recent versions of the PostGRES DBMS allow the user to choose between basic SI and SSI. The tradeoff is ensuring full serializability with SSI versus living with possible rare anomalies but having better performance with basic SI.

21.5 GRANULARITY OF DATA ITEMS AND MULTIPLE GRANULARITY LOCKING

All concurrency control techniques assume that the database is formed of a number of named data items. A database item could be chosen to be one of the following:

- a database record
- a field value of a database record
- a disk block
- a whole file
- the whole database

The particular choice of data item type can affect the performance of concurrency control and recovery.

1. Granularity Level Considerations for Locking:

- The size of data items is called the data item granularity.
- Fine granularity refers to small item sizes, whereas coarse granularity refers to large item sizes.
- The larger the data item size the lower is the degree of concurrency permitted. For example, if the data item size is a disk block, a transaction T that needs to lock a single record B must lock the whole disk block X that contains B because a lock is associated with the whole data item (block). Now, if another transaction S wants to lock a different record C that happens to reside in the same disk block X in a

conflicting lock mode, it is forced to wait. If the data item size was a single record instead of a disk block, transaction S would be able to proceed, because it would be locking a different data item (record).

- The smaller the data item size, the more is the number of items in the database. Because every item is associated with a lock, the system will have a larger number of active locks to be handled by the lock manager. More lock and unlock operations will be performed, causing a higher overhead. In addition, more storage space will be required for the lock table. For timestamps, storage is required for the `read_TS` and `write_TS` for each data item, and there will be similar overhead for handling a large number of items.
- The best item size depends on the types of transactions involved. If a typical transaction accesses a small number of records, it is advantageous to have the data item granularity be one record. If a transaction typically accesses many records in the same file, it may be better to have block or file granularity so that the transaction will consider all those records as one (or a few) data items.

2. Multiple Granularity Level Locking:

- A database system should support multiple levels of granularity, where the granularity level can be adjusted dynamically for various mixes of transactions.

  Figure 20 shows a simple granularity hierarchy with a database containing two files, each file containing several disk pages, and each page containing several records.

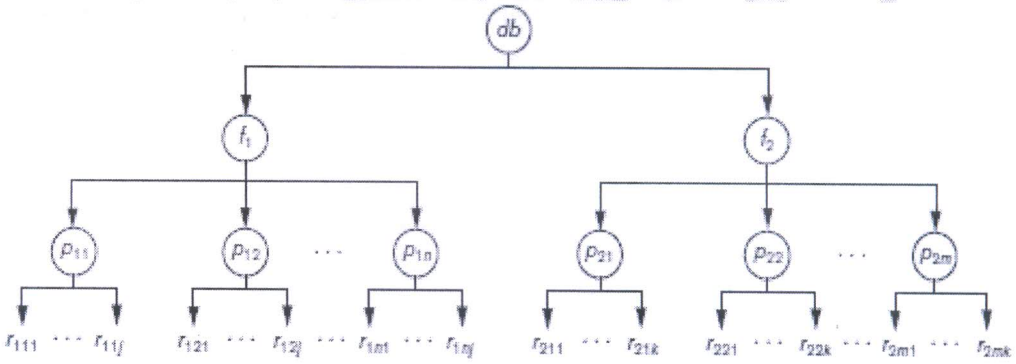

Figure 20: A granularity hierarchy for illustrating multiple granularity level Locking

This can be used to illustrate a multiple granularity level 2PL protocol, with shared/exclusive locking modes, where a lock can be requested at any level. Additional types of locks will be needed to support such a protocol efficiently. Consider the following scenario, which refers to the example in Figure 20. Suppose transaction T_1 wants to update all the records in file f_1, and T_1 requests and is granted an exclusive lock for f_1. Then all of f_1's pages (p_{11} through p_{1n})—and the records contained on those

pages are locked in exclusive mode. This is beneficial for T_1 because setting a single file-level lock is more efficient than setting ***n*** pagelevel locks or having to lock each record individually.

Now suppose another transaction T_2 only wants to read record r_{1nj} from page p_{1n} of file f_1; then T_2 would request a shared record-level lock on r_{1nj}. However, the database system- the transaction manager or the lock manager - must verify the compatibility of the requested lock with already held locks. This can be verified by traversing the tree from the leaf r_{1nj} to p_{1n} to f_1 to db. If at any time a conflicting lock is held on any of those items, then the lock request for r_{1nj} is denied and T_2 is blocked and must wait. This traversal would be fairly efficient.

If transaction T_2's request came before transaction T_1's request, the shared record lock is granted to T_2 for r_{1nj}, but when T_1's file-level lock is requested, it can be time-consuming for the lock manager to check all nodes (pages and records) that are descendants of node f_1 for a lock conflict. This would be very inefficient and would defeat the purpose of having multiple granularity level locks.

- To make multiple granularity level locking practical, ***intention locks*** are needed. The idea behind intention locks is for a transaction to indicate, along the path from the root to the desired node, what type of lock (shared or exclusive) it will require from one of the node's descendants.

- There are <u>three types</u> of intention locks:
  1. Intention-shared (IS) indicates that one or more shared locks will be requested on some descendant node(s).
  2. Intention-exclusive (IX) indicates that one or more exclusive locks will be requested on some descendant node(s).
  3. Shared-intention-exclusive (SIX) indicates that the current node is locked in shared mode but that one or more exclusive locks will be requested on some descendant node(s).

The compatibility table of the three intention locks, and the actual shared and exclusive locks, is shown in Figure 21.

	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

Figure 21: Lock compatibility matrix for multiple granularity locking

- An appropriate locking protocol must be used in addition to the three types of intention locks. The ***multiple granularity locking*** (MGL) protocol consists of the following rules:
  1. The lock compatibility (based on Figure 21) must be adhered to.
  2. The root of the tree must be locked first, in any mode.
  3. A node N can be locked by a transaction T in `S` or `IS` mode only if the parent node N is already locked by transaction T in either `IS` or `IX` mode.
  4. A node N can be locked by a transaction T in `X`, `IX`, or `SIX` mode only if the parent of node N is already locked by transaction T in either `IX` or `SIX` mode.
  5. A transaction T can lock a node only if it has not unlocked any node (to enforce the 2PL protocol).
  6. A transaction T can unlock a node, N, only if none of the children of node N are currently locked by T.

  Rule 1 states that conflicting locks cannot be granted. Rules 2, 3, and 4 state the conditions when a transaction may lock a given node in any of the lock modes. Rules 5 and 6 of the MGL protocol enforce 2PL rules to produce serializable schedules.
  Basically, the locking starts from the root and goes down the tree until the node that needs to be locked is encountered, whereas unlocking starts from the locked node and goes up the tree until the root itself is unlocked.

  Consider the following three transactions:
  1. T_1 wants to update record r_{111} and record r_{211}.
  2. T_2 wants to update all records on page p_{12}.
  3. T_3 wants to read record r_{11j} and the entire f_2 file. Figure 22 shows a possible serializable schedule for these three transactions.

  Only the lock and unlock operations are shown. The notation `<lock_type>(<item>)` is used to display the locking operations in the schedule.

- The multiple granularity level protocol is especially suited when processing a mix of transactions that include:
  (1) short transactions that access only a few items (records or fields) and
  (2) long transactions that access entire files.

  In this environment, less transaction blocking and less locking overhead are incurred by such a protocol when compared to a single-level granularity locking approach.

T_1	T_2	T_3
IX(*db*)		
IX(f_1)		
	IX(*db*)	
		IS(*db*)
		IS(f_1)
		IS(p_{11})
IX(p_{11})		
X(r_{111})		
	IX(f_1)	
	X(p_{12})	
		S(r_{11j})
IX(f_2)		
IX(p_{21})		
X(p_{211})		
unlock(r_{211})		
unlock(p_{21})		
unlock(f_2)		
		S(f_2)
	unlock(p_{12})	
	unlock(f_1)	
	unlock(*db*)	
unlock(r_{111})		
unlock(p_{11})		
unlock(f_1)		
unlock(*db*)		
		unlock(r_{11j})
		unlock(p_{11})
		unlock(f_1)
		unlock(f_2)
		unlock(*db*)

Figure 22: Lock operations to illustrate a serializable schedule

QUESTION BANK:

1. What is snapshot isolation? What are the advantages and disadvantages of concurrency control methods that are based on snapshot isolation?
2. How does the granularity of data items affect the performance of concurrency control? What factors affect selection of granularity size for data items?
3. How do optimistic concurrency control techniques differ from other concurrency control techniques? Why are they called validation or certification techniques? Discuss the typical phases of an optimistic concurrency control method.