INTRODUCTION TO TRANSACTION PROCESSING

One criterion for classifying a database system is according to the number of users who can use the system concurrently—that is, at the same time. A DBMS is single-user if at most one user at a time can use the system, and it is multiuser if many users can use the system—and hence access the database-concurrently. Single-user DBMSs are mostly restricted to personal computer systems; most other DBMSs are multiuser. For example, an airline reservations system is used by hundreds of travel agents and reservation clerks concurrently.

Multiple users can access databases—and use computer systems-simultaneously because of the concept of multiprogramming, which allows the computer to execute multiple programs-or processes-at the same time. If only a single central processing unit (CPU) exists, it can actually execute at most one process at a time. However, multiprogramming operating systems execute some commands from one process, then suspend that process and execute some commands from the next process, and so on. A process is resumed at the point where it was suspended whenever it gets its turn to use the CPU again. Hence, concurrent execution of processes is actually interleaved. Figure 1, shows two processes A and B executing concurrently in an interleaved fashion. Interleaving keeps the CPU busy when a process requires an input or output (r/o) operation, such as reading a block from disk. The CPU is switched to execute another process rather than remaining idle during r/o time. Interleaving also prevents a long process from delaying other processes.

![Figure 1: Interleaved processing versus parallel processing of concurrent](image)

If the computer system has multiple hardware processors (CPUs), parallel processing of multiple processes is possible, as illustrated by processes C and D in Figure 1.

Transactions, Read and Write Operations, and DBMS Buffers

A transaction is an executing program that forms a logical unit of database processing. A transaction includes one or more database access operations—these can include insertion, deletion, modification, or retrieval operations. The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL.
Chapter 9: Introduction To Transaction Processing

One way of specifying the transaction boundaries is by specifying explicit `begin transaction` and `end transaction` statements in an application program; in this case, all database access operations between the two are considered as forming one transaction. A single application program may contain more than one transaction if it contains several transaction boundaries. If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction**.

The basic database access operations that a transaction can include are as follows:

- **read_item(X)**: Reads a database item named X into a program variable.
- **write_item(X)**: Writes the value of program variable X into the database item named X.

Executing a `read_item(X)` command includes the following steps:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the buffer to the program variable named X.

Executing a `write_item(X)` command includes the following steps:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the program variable named X into its correct location in the buffer.
4. Store the updated block from the buffer back to disk (either immediately or at some later point in time).

The **DBMS** will generally maintain a number of buffers in main memory that hold database disk blocks containing the database items being processed. When these buffers are all occupied, and additional database blocks must be copied into memory, some buffer replacement policy is used to choose which of the current buffers is to be replaced. If the chosen buffer has been modified, it must be written back to disk before it is reused.

A transaction includes `read_item` and `write_item` operations to access and update the database. Figure 2 shows examples of two very simple transactions. The read-set of a transaction is the set of all items that the transaction reads, and the write-set is the set of all items that the transaction writes. For example, the read-set of T1 in Figure 2 is \{X, Y\} and its write-set is also \{X, Y\}.
Why Concurrency Control Is Needed

Several problems can occur when concurrent transactions execute in an uncontrolled manner. Figure 2(a) shows a transaction $T_1$ that transfers $N$ reservations from one flight whose number of reserved seats is stored in the database item named $X$ to another flight whose number of reserved seats is stored in the database item named $Y$. Figure 2(b) shows a simpler transaction $T_2$ that just reserves $M$ seats on the first flight ($X$) referenced in transaction $T_1$.

The Lost Update Problem: This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect. Suppose that transactions $T_1$ and $T_2$ are submitted at approximately the same time, and suppose that their operations are interleaved as shown in Figure 3a, then the final value of item $X$ is incorrect, because $T_2$ reads the value of $X$ before $T_1$ changes it in the database, and hence the updated value resulting from $T_1$ is lost.
Chapter 9: Introduction To Transaction Processing

The Temporary Update (or Dirty Read) Problem. This problem occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed by another transaction before it is changed back to its original value. Figure 3b shows an example where \( T_1 \) updates item \( X \) and then fails before completion, so the system must change \( X \) back to its original value. Before it can do so, however, transaction \( T_2 \) reads the "temporary" value of \( X \), which will not be recorded permanently in the database because of the failure of \( T_1 \). The value of item \( X \) that is read by \( T_2 \) is called dirty data, because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the dirty read problem.

The Incorrect Summary Problem: If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated. For example, suppose that a transaction \( T_3 \) is calculating the total number of reservations on all the flights; meanwhile, transaction \( T_1 \) is executing. If the interleaving of operations shown in Figure 3c occurs, the result of \( T_3 \) will be off by an amount \( N \) because \( T_3 \) reads the value of \( X \) after \( N \) seats have been subtracted from it but reads the value of \( Y \) before those \( N \) seats have been added to it.

Another problem that may occur is called unrepeatable read, where a transaction \( T \) reads an item twice and the item is changed by another transaction \( T' \) between the two reads. Hence, \( T \) receives different values for its two reads of the same item.

Why Recovery Is Needed
Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either (1) all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or (2) the transaction has no effect whatsoever on the database.
or on any other transactions. The DBMS must not permit some operations of a transaction $T$ to be applied to the database while other operations of $T$ are not. This may happen if a transaction fails after executing some of its operations but before executing all of them.

**Types of Failures:** Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:

- **A computer failure (system crash):** A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—e.g., main memory failure.
- **A transaction or system error:** Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.
- **Local errors or exception conditions detected by the transaction:** During transaction execution, certain conditions may occur that necessitate cancellation of the transaction. For example, data for the transaction may not be found. Notice that an exception condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled. This exception should be programmed in the transaction itself, and hence would not be considered a failure.
- **Concurrency control enforcement:** The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock.
- **Disk failure:** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.
- **Physical problems and catastrophes:** This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

**TRANSACTION AND SYSTEM CONCEPTS**

**Transaction States and Additional Operations**

A transaction is an atomic unit of work that is either completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts. Hence, the recovery manager keeps track of the following operations:

- **BEGIN_TRANSACTION:** This marks the beginning of transaction execution.
- **READ or WRITE:** These specify read or write operations on the database items that are executed as part of a transaction.
- **END_TRANSACTION:** This specifies that **READ** and **WRITE** transaction operations have ended and marks the end of transaction execution. However, at this point it may be necessary to check
whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability or for some other reason.

- **COMMIT_TRANSACTION**: This signals a *successful end of* the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.
- **ROLLBACK (OR ABORT)**: This signals that the transaction has ended *unsuccessfully*, so that any changes or effects that the transaction may have applied to the database must be *undone*.

Figure 4 shows a state transition diagram that describes how a transaction moves through its execution states. A transaction goes into an active state immediately after it starts execution, where it can issue **READ** and **WRITE** operations. When the transaction ends, it moves to the partially committed state. At this point, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently.

Once this check is successful, the transaction is said to have reached its commit point and enters the committed state. Once a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database. However, a transaction can go to the failed state if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its **WRITE** operations on the database. The terminated state corresponds to the transaction leaving the system.

![State transition diagram illustrating the states for transaction execution](image_url)

**The System Log**

To be able to recover from failures that affect transactions, the system maintains a log to keep track of all transaction operations that affect the values of database items. This information may be needed to permit recovery from failures. We now list the types of entries-called log records-that are written to the log and the action each performs. In these entries, T refers to a unique transaction-id that is generated automatically by the system and is used to identify each transaction:
Chapter 9: Introduction To Transaction Processing

- **[start-transaction, T]**: Indicates that transaction T has started execution.
- **[write_item, T, X, old_value, new_value]**: Indicates that transaction T has changed the value of database item X from old_value to new_value.
- **[read_item, T, X]**: Indicates that transaction T has read the value of database item X.
- **[commit, T]**: Indicates that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
- **[abort, T]**: Indicates that transaction T has been aborted.

**Commit Point of a Transaction**

A transaction T reaches its commit point when all its operations that access the database have been executed successfully and the effect of all the transaction operations on the database have been recorded in the log. Beyond the commit point, the transaction is said to be committed, and its effect is assumed to be permanently recorded in the database. The transaction then writes a commit record [commit, T] into the log. If a system failure occurs, we search back in the log for all transactions T that have written a [start_transaction, T] record into the log but have not written their [commit, T] record yet; these transactions may have to be rolled back to undo their effect on the database during the recovery process.

**DESIRABLE PROPERTIES OF TRANSACTIONS**

Transactions should possess several properties. These are often called the ACID properties, and they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

1. **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
2. **Consistency preservation**: A transaction is consistency preserving if its complete execution takes the database from one consistent state to another.
3. **Isolation**: A transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.
4. **Durability or permanency**: The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

**CHARACTERIZING SCHEDULES BASED ON RECOVERABILITY**

When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from the various transactions is known as a schedule (or history).
Schedules (Histories) of Transactions

A schedule (or history) $S$ of $n$ transactions $T_1, T_2, \ldots, T_n$ is an ordering of the operations of the transactions subject to the constraint that, for each transaction $T_i$ that participates in $S$, the operations of $T_i$ in $S$ must appear in the same order in which they occur in $T_i$.

For the purpose of recovery and concurrency control, we are mainly interested in the read, item and write item operations of the transactions, as well as the commit and abort operations. A shorthand notation for describing a schedule uses the symbols $r$, $w$, $c$, and $a$ for the operations read item, write item, commit, and abort, respectively, and appends as subscript the transaction id (transaction number) to each operation in the schedule. In this notation, the database item 'X that is read or written follows the rand $w$ operations in parentheses. For example, the schedule of Figure 3(a), which we shall call $S_a$, can be written as follows in this notation:

$S_a$: $r1(X)$; $r2(X)$; $W1(X)$; $r1(Y)$; $w2(X)$; $W1(Y)$;

Two operations in a schedule are said to conflict if they satisfy all three of the following conditions: (1) they belong to different transactions; (2) they access the same item $X$; and (3) at least one of the operations is a write item($X$).

A schedule $S$ of $n$ transactions $T_1, T_2, \ldots, T_n$, is said to be a complete schedule if the following conditions hold:

1. The operations in $S$ are exactly those operations in $T_1, T_2, \ldots, T_n$, including a commit or abort operation as the last operation for each transaction in the schedule.
2. For any pair of operations from the same transaction $T_i$, their order of appearance in $S$ is the same as their order of appearance in $T$;
3. For any two conflicting operations, one of the two must occur before the other in the schedule.

For some schedules it is easy to recover from transaction failures, whereas for other schedules the recovery process can be quite involved. Hence, it is important to characterize the types of schedules for which recovery is possible, as well as those for which recovery is relatively simple. First, we would like to ensure that, once a transaction $T$ is committed, it should never be necessary to roll back $T$. The schedules that theoretically meet this criterion are called recoverable schedules and those that do not are called non recoverable, and hence should not be permitted. A schedule $S$ is recoverable if no transaction $T$ in $S$ commits until all transactions $T'$ that have written an item that $T$ reads have committed.

Recoverable schedules require a complex recovery process, but if sufficient information is kept (in the log), a recovery algorithm can be devised. In a recoverable schedule, no committed transaction ever needs to be rolled back. However, it is possible for a phenomenon known as cascading rollback (or cascading abort) to occur, where an uncommitted transaction has to be rolled back because it read an item from a transaction that failed. Because cascading rollback can be quite time-consuming—since numerous transactions can be rolled back it is important to characterize the schedules where this
phenomenon is guaranteed not to occur. A schedule is said to be **cascadeless**, or to avoid cascading **rollback**, if every transaction in the schedule reads only items that were written by committed transactions.

Finally, there is a third, more restrictive type of schedule, called a strict schedule, in which transactions can neither read nor write an item X until the last transaction that wrote X has committed (or aborted). Strict schedules simplify the recovery process. In a strict schedule, the process of undoing a write_item(X) operation of an aborted transaction is simply to restore the before image (old_value or BFIM) of data item X. This simple procedure always works correctly for strict schedules, but it may not work for recoverable or cascadeless schedules.

**TRANSACTION SUPPORT IN SQL**

The definition of an SQL-transaction is similar to our already defined concept of a transaction. That is, it is a logical unit of work and is guaranteed to be atomic. A single SQL statement is always considered to be atomic—either it completes execution without error or it fails and leaves the database unchanged. With SQL, there is no explicit Begin_Transaction statement. Transaction initiation is done implicitly when particular SQL statements are encountered. However, every transaction must have an explicit end statement, which is either a COMMIT or a ROLLBACK. Every transaction has certain characteristics attributed to it SQL. The characteristics are the **access mode**, the **diagnostic area size**, and the isolation **level**.

The **access mode** can be specified as READ ONLY or READ WRITE. The default is READ WRITE, unless the isolation level of READ UNCOMMITTED is specified, in which case READ ONLY is assumed. A mode of READ WRITE allows update, insert, delete and create commands to be executed. A mode of READ ONLY, as the name implies, is simply for data retrieval.

The **diagnostic area** size option, DIAGNOSTIC SIZE n, specifies an integer value n, indicating the number of conditions that can be held simultaneously in the diagnostic area. These conditions supply feedback information (errors or exceptions) to the user or program on the most recently executed SQL statement.

The **isolation level** option is specified using the statement ISOLATION LEVEL <isolation>, where the value for <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE.

If a transaction executes at a lower isolation level than SERIALIZABLE, then one or more of the following three violations may occur:

1. **Dirty read**: A transaction T1 may read the update of a transaction T2, which has not yet committed. If T2 fails and is aborted, then T1 would have read a value that does not exist and is incorrect.
2. **Non repeatable read**: A transaction T1 may read a given value from a table. If another transaction T2 later updates that value and T1 reads that value again, T1 will see a different value.
3. **Phantoms**: A transaction T1 may read a set of rows from a table perhaps based on some condition specified in the SQL WHERE-clause. Now suppose that a transaction T2 inserts a new
row that also satisfies the WHERE-clause condition used in \(T_1\), into the table used by \(T_1\). If \(T_1\) is repeated, then \(T_1\) will see a phantom, a row that previously did not exist.

LOCKING TECHNIQUES FOR CONCURRENCY CONTROL

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. Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

**Types of Locks and System Lock Tables**

Several types of locks are used in concurrency control.

**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. We refer to the current value (or state) of the lock associated with item \(X\) as \(\text{LOCK}(X)\).

Two operations, lock\(_{(\text{item})}\) and unlock\(_{(\text{item})}\), are used with binary locking. A transaction requests access to an item \(X\) by first issuing a lock\(_{(\text{item})}(X)\) operation. If \(\text{LOCK}(X) = 1\), the transaction is forced to wait. If \(\text{LOCK}(X) = 0\), it is set to 1 (the transaction locks the item) and the transaction is allowed to access item \(X\). When the transaction is through using the item, it issues an unlock\(_{(\text{item})}(X)\) operation, which sets \(\text{LOCK}(X)\) to 0(unlocks the item) so that \(X\) may be accessed by other transactions. Hence, a binary lock enforces mutual exclusion on the data item.

If the simple binary locking scheme described here is used, every transaction must obey the following rules:

1. A transaction \(T\) must issue the operation lock\(_{(\text{item})}(X)\) before any read\(_{(\text{item})}(X)\) or write\(_{(\text{item})}(X)\) operations are performed in \(T\).

2. A transaction \(T\) must issue the operation unlock\(_{(\text{item})}(X)\) after all read\(_{(\text{item})}(X)\) and write\(_{(\text{item})}(X)\) operations are completed in \(T\).

3. A transaction \(T\) will not issue a lock\(_{(\text{item})}(X)\) operation if it already holds the lock on item \(X\).

4. A transaction \(T\) will not issue an unlock\(_{(\text{item})}(X)\) operation unless it already holds the lock on item \(X\).

**Shared/Exclusive (or Read/Write) Locks**: The preceding binary locking scheme is too restrictive for database items, because at most one transaction can hold a lock on a given item. We should allow several transactions to access the same item \(X\) if they all access \(X\) for reading purposes only. However, if a transaction is to write an item \(X\), it must have exclusive access to \(X\). For this purpose, a different type of lock called a **multiple mode lock** is used. In this scheme-called **shared/exclusive** or **read/write locks**-there are three locking operations: read\(_{(\text{lock})}(X)\), write\(_{(\text{lock})}(X)\), and unlock\(_{(\text{lock})}(X)\). A lock associated with an
item X, LOCK(X), now has three possible states: "read-locked," "writelocked," 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.

When we use the shared/exclusive locking scheme, the system must enforce the following rules:

1. A transaction T must issue the operation read_lock(X) or write_lock(X) before any read-item(X) operation is performed in T.
2. A transaction T must issue the operation write_lock(X) before any write-item(X) operation is performed in T.
3. A transaction T must issue the operation unlock(X) after all read-item(X) and write-item(X) operations are completed in T.
4. A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X.
5. A transaction T will not issue a write_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X. This rule may be relaxed.
6. A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

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 (first) phase, during which new locks on items can be acquired but none can be released; and a shrinking (second) phase, during which existing locks can be released but no new locks can be acquired.

Basic, Conservative, Strict, and Rigorous Two-Phase Locking: There are a number of variations of two-phase locking (2PL). The technique just described is known as basic 2PL. A variation known as conservative 2PL (or static 2PL) requires a transaction to lock all the items it accesses before the transaction begins execution, by predeclaring its readset and write-set.

The read-set of a transaction is the set of all items that the transaction reads, and the write-set is the set of all items that it writes. If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking.

A more restrictive variation of strict 2PL is rigorous 2PL, which also guarantees strict schedules. In this variation, a transaction T does not release any of its locks (exclusive or shared) until after it commits or aborts, and so it is easier to implement than strict 2PL. Notice the difference between conservative and rigorous 2PL; the former must lock all its items before it starts so once the transaction starts it is in its shrinking phase, whereas the latter does not unlock any of its items until after it terminates (by committing or aborting) so the transaction is in its expanding phase until it ends.
Dealing with Deadlock and Starvation

Deadlock occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T’ in the set. Hence, each transaction in the set is on a waiting queue, waiting for one of the other transactions in the set to release the lock on an item.

Deadlock Prevention Protocols: One way to prevent deadlock is to use a deadlock prevention protocol. One deadlock prevention protocol, which is used in conservative two-phase locking, requires that every transaction lock all the items it needs in advance (which is generally not a practical assumption)—if any of the items cannot be obtained, none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs. This solution obviously further limits concurrency.

A number of other deadlock prevention schemes have been proposed that make a decision about what to do with a transaction involved in a possible deadlock situation: Should it be blocked and made to wait or should it be aborted, or should the transaction preempt and abort another transaction?

The rules followed by these schemes are as follows:

- **Wait-die**: If $TS(T) < TS(T_j)$, then (Tj older than Tj) Tj is allowed to wait; otherwise (T, younger than T) abort Tj (T, dies) and restart it later with the same timestamp.
- **Wound-wait**: If $TS(T) < TS(T_j)$, then (T, older than Tj) abort Tj (T, wounds Tj) and restart it later with the same timestamp; otherwise (T, younger than T) Tj is allowed to wait.

A second more practical approach to dealing with deadlock is deadlock detection, where the system checks if a state of deadlock actually exists. This solution is attractive if we know there will be little interference among the transactions—that is, if different transactions will rarely access the same items at the same time.

A simple scheme to deal with deadlock is the use of timeouts. 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 or not.

**Starvation**: Another problem that may occur when we use locking is starvation, which occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally. This may occur if the waiting scheme for locked items is unfair, giving priority to some transactions over others. One solution for starvation is to have a fair waiting scheme, such as using a first-come-first-served queue; transactions are enabled to lock an item in the order in which they originally requested the lock.

**CONCURRENCY CONTROL BASED ON TIMESTAMP ORDERING**

A timestamp is a unique identifier created by the DBMS to identify a transaction. Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the transaction start time. We will refer to the timestamp of transaction T as $TS(T)$.
Concurrency control techniques based on timestamp ordering do not use locks; hence, deadlocks cannot occur.

The idea for this scheme is to order the transactions based on their timestamps. A schedule in which the transactions participate is then serializable, and the equivalent serial schedule has the transactions in order of their timestamp values. This is called timestamp ordering (TO).

The timestamp algorithm must ensure that, for each item accessed by conflicting operations in the schedule, the order in which the item is accessed does not violate the serializability order. To do this, the algorithm associates with each database item X two timestamp (TS) values:

1. Read_T(S)(X): The read timestamp of item X is the largest timestamp among all the timestamps of transactions that have successfully read item X—that is, read_T(S)(X) = TS(T), where T is the youngest transaction that has read X successfully.
2. Write_T(S)(X): The write timestamp of item X is the largest of all the timestamps of transactions that have successfully written item X—that is, write_T(S)(X) = TS(T), where T is the youngest transaction that has written X successfully.

Whenever some transaction T times to issue a read_item(X) or a write_item(X) operation, the basic TO algorithm compares the timestamp of T with read_T(S)(X) and write_T(S)(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. If T is aborted and rolled back, any transaction T1 that may have used a value written by T must also be rolled back. Similarly, any transaction T2 that may have used a value written by T1 must also be rolled back, and so on. This effect is known as cascading rollback and is one of the problems associated with basic TO, since the schedules produced are not guaranteed to be recoverable.

We first describe the basic TO algorithm here. The concurrency control algorithm must check whether conflicting operations violate the timestamp ordering in the following two cases:

1. Transaction T issues a write_item(X) operation:
   a) If read_T(S)(X) > TS(T) or if write_T(S)(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with a timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.
   b) If the condition in part (a) does not occur, then execute the write_item(X) operation of T and set write_T(S)(X) to TS(T).

2. Transaction T issues a read_item(X) operation:
Chapter 9: Introduction To Transaction Processing

a) If write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with timestamp greater than TS(T)-and hence after T in the timestamp ordering-has already written the value of item X before T had a chance to read X.

b) If write_TS(X) ≤TS(T), then execute the read_item(X) operation of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

OPTIMISTIC CONCURRENCY CONTROL TECHNIQUES

In optimistic concurrency control techniques, also known as validation or certification techniques, no checking is done while the transaction is executing. Several proposed concurrency control methods use the validation technique. In this scheme, updates in the transaction are not applied directly to the database items until the transaction reaches its end. 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.

USING LOCKS FOR CONCURRENCY CONTROL IN INDEXES

Two-phase locking can also be applied to indexes, where the nodes of an index correspond to disk pages. However, holding locks on index pages until the shrinking phase of 2PL could cause an undue amount of transaction blocking. This is because searching an index always starts at the root, so if a transaction wants to insert a record (write operation), the root would be locked in exclusive mode, so all other conflicting lock requests for the index must wait until the transaction enters its shrinking phase. This blocks all other transactions from accessing the index, so in practice other approaches to locking an index must be used.

The tree structure of the index can be taken advantage of when developing a concurrency control scheme. For example, when an index search (read operation) is being executed, a path in the tree is traversed from the root to a leaf. Once a lower-level node in the path has been accessed, the higher-level nodes in that path will not be used again. So once a read lock on a child node is obtained, the lock on the parent can be released. Second, when an insertion is being applied to a leaf node (that is, when a key and a pointer are inserted), then a specific leaf node must be locked in exclusive mode. However, if that
node is not full, the insertion will not cause changes to higher-level index nodes, which implies that they need not be locked exclusively.