CHICAGO STATE UNIVERSITY

**MATHEMATICS AND COMPUTER SCIENCE DEPARTMENT**

CPTR 5600-01

**ADVANCED DATA BASE DESIGN AND IMPLEMENTATION**

Topic –TRANSACTION MANAGEMENT AND CONCURRENCY CONTROL

**Submitted By:**

DRONAMRAJU VENKATA SESHA SURYA SAI MANI PREETHI

900726470

Instructor:

DR. LUIS VIDAL

Fall-2016

ABSTARCT

Later now, Many real-world applications taking time consuming to access data from database, For example, consider a navigation systems, control systems, command systems, stock trading, networking management and telephonic switching system. These applications needed gathering data from sources, processing information in the context of information obtained in the past, and contributing response in timely manner. Hence, these applications need a database system where transactions are associated with deadlines on their completion times.

In order to exploit parallel transactions, database management systems must achieve a high level of concurrency when executing transactions. Many real-time concurrency control methods are based on two-phase locking (2PL). For every success full transaction, it must maintain Atomicity, Consistency, Isolation, and Durability commonly known as ACID properties.

In this paper, we focus on transaction management and how transaction management maintains ACID properties in concurrency controlling system. We also discuss transaction management for web based applications, and cover topics such as spring transaction management. We then discuss current challenges for database parallel transactions and some preliminary approaches that address some of these challenges.

TABLES OF CONTENTS

1. Introduction

1.1 Introduction to Transaction Management

1.2 Concurrency Control in Database

2. Database Transaction Management

2.1 Transaction properties (ACID)

2.2 Isolation of Transactions

2.3 Transaction states

2.4 Rollback/Aborting a Transaction

3. Concurrency Control

3.1 Concurrency Control mechanisms

3.2 Concurrency Control Mechanisms and Protocols

3.2.1 Lock-Based Concurrency Control Protocols

3.2.2 Strict Two Phase locking (2PL) protocol

3.3 Concurrency control in Real-time web applications

4. Deadlock Principles

5. Optimistic and Pessimistic Concurrency Control

6. Conclusion

7. References

INTRODUCTION

As organizations increase their adoption of database systems as the key data management technology for day-to-day operations and decision making, processing of transactions in parallel improves the application performance and productivity. In Advanced data base, there are following two primary processes.

1. Transaction management (TM) handles all transactions properly in DBMS. Database transactions are the events or activities such as series of data read/write operations on data object(s) stored in database system
2. Concurrency control (CC) is a process to ensure that data is updated correctly and appropriately when multiple transactions are concurrently executed in DBMS.

In general, concurrency control is an essential part of Transaction Management. It is a mechanism for correctness when two or more database transactions that access the same data or data set are executed concurrently with time overlap. If multiple transactions are executed serially or sequentially, data is consistent in a database. However, if concurrent transactions with interleaving operations are executed, some unexpected data and inconsistent result may occur. Data interference is usually caused by a write operation among transactions on the same set of data in DBMS. For example, the lost update problem may occur when a second transaction writes a second value of a data content on top of the first value written by a first concurrent transaction.

Concurrency control is one of the primary mechanisms in transaction management to provide integrity of data and safety in DBMS. Today, with hundred thousand or more transactions in a few minutes, transaction management and concurrency control become much more complex and sophisticated. Two pessimistic and optimistic mechanisms are still popular, but other techniques such as semi-optimistic are also applied in DBMS for higher performance, better throughput, more accurate results, and faster run time.

**Database Transaction Management**

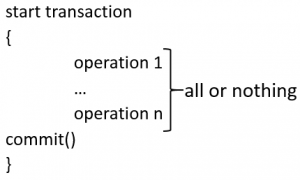
**2.1 Transaction properties (ACID)**

A transaction is a very small unit of a task. It may contain several lower level sub tasks. In order to data accuracy, integrity and completeness, a transaction in database system must maintain Atomicity, Consistency, Durability and Isolation.

**Atomicity**: The database transaction should be atomic, that is, either all operations should execute or none. Any transaction should be left with partially completed. States should be defined either before execution of the transaction or after the execution of the transaction.

Typically, systems implement Atomicity by providing some mechanism to indicate which transactions have started and which finished; or by keeping a copy of the data before any changes occurred. Several file systems have developed methods for avoiding the need to keep multiple copies of data, using journaling. Databases usually implement this using some form of logging/journaling to track changes. The system synchronizes the logs as necessary once the actual changes have successfully taken place. Afterwards, crash recovery simply ignores incomplete entries. Although implementations vary depending on factors such as concurrency issues, the principle of atomicity, that is, complete success or complete failure remain.

An example of an atomic transaction is a monetary transfer from bank account A to account B. It consists of two operations, withdrawing the money from account A and saving it to account B. Performing these operations in an atomic transaction ensures that the database remains in a consistent state, that is, money is not lost nor created if either of those two operations fail.



If for any reason an error occurs and the transaction is unable to complete all of its steps, then the system is returned to the state it was in before the transaction was started.

**Consistency:** The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.

Consistency is one of the four guarantees that define ACID transactions. However, significant ambiguity exists about the nature of this guarantee. It is defined variously as:

* The guarantee that any transactions started in the future necessarily see the effects of other transactions committed in the past
* The guarantee that database constraints are not violated, particularly once a transaction commits
* The guarantee that operations in transactions are performed accurately, correctly, and with validity, with respect to application semantics

Consistency is a very general term, which demands that the data must meet all validation rules. In the previous example, the validation is a requirement that A + B = 100. Also, it may be inferred that both A and B must be integers. A valid range for A and B may also be inferred. All validation rules must be checked to ensure consistency. Assume that a transaction attempts to subtract 10 from A without altering B. Because consistency is checked after each transaction, it is known that A + B = 100 before the transaction begins. If the transaction removes 10 from A successfully, atomicity will be achieved. However, a validation check will show that A + B = 90, which is inconsistent with the rules of the database. The entire transaction must be cancelled and the affected rows rolled back to their pre-transaction state. If there had been other constraints, triggers, or cascades, every single change operation would have been checked in the same way as above before the transaction was committed.

**Durability**: The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.

The durability property ensures that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). To defend against power loss, transactions must be recorded in a non-volatile memory.

Consider a transaction that transfers 10 from A to B. First it removes 10 from A. then it adds 10 to B. At this point, the user is told the transaction was a success. however the changes are still queued in the disk buffer waiting to be committed to disk. Power fails and the changes are lost. The user assumes that the changes have been persisted.

**Isolation:** In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, that is, one after the other. Providing isolation is the main goal of concurrency control. Depending on the concurrency control method, the effects of an incomplete transaction might not even be visible to another transaction.

To demonstrate isolation, we assume two transactions execute at the same time, each attempting to modify the same data. One of the two must wait until the other completes in order to maintain isolation. Consider two transactions. T1 transfers 10 from A to B. T2 transfers 10 from B to A. Combined, there are four actions:

T1 subtracts 10 from A.

T1 adds 10 to B.

T2 subtracts 10 from B.

T2 adds 10 to A.

If these operations are performed in order, isolation is maintained, although T2 must wait. Consider what happens if T1 fails halfway through. The database eliminates T1's effects, and T2 sees only valid data.

By interleaving the transactions, the actual order of actions might be:

T1 subtracts 10 from A.

T2 subtracts 10 from B.

T2 adds 10 to A.

T1 adds 10 to B.

Again, consider what happens if T1 fails halfway through. By the time T1 fails, T2 has already modified A; it cannot be restored to the value it had before T1 without leaving an invalid database. This is known as a write-write failure,[citation needed] because two transactions attempted to write to the same data field. In a typical system, the problem would be resolved by reverting to the last known good state, canceling the failed transaction T1, and restarting the interrupted transaction T2 from the good state.

**2.2Transaction Isolation Levels**

Transaction isolation levels are a measure of the extent to which transaction isolation succeeds. In particular, transaction isolation levels are defined by the presence or absence of the following phenomena:

**Dirty Reads** A dirty *read* occurs when a transaction reads data that has not yet been committed. For example, suppose transaction 1 updates a row. Transaction 2 reads the updated row before transaction 1 commits the update. If transaction 1 rolls back the change, transaction 2 will have read data that is considered never to have existed.

**Nonrepeatable Reads** A nonrepeatable *read* occurs when a transaction reads the same row twice but gets different data each time. For example, suppose transaction 1 reads a row. Transaction 2 updates or deletes that row and commits the update or delete. If transaction 1 rereads the row, it retrieves different row values or discovers that the row has been deleted.

**Phantoms** A *phantom* is a row that matches the search criteria but is not initially seen. For example, suppose transaction 1 reads a set of rows that satisfy some search criteria. Transaction 2 generates a new row (through either an update or an insert) that matches the search criteria for transaction 1. If transaction 1 reexecutes the statement that reads the rows, it gets a different set of rows.

The four transaction isolation levels are defined in terms of these phenomena. In the following table, an "X" marks each phenomenon that can occur.

|  |  |  |  |
| --- | --- | --- | --- |
| Transaction isolation level | Dirty reads | Nonrepeatable reads | Phantoms |
| Read uncommitted | X | X | X |
| Read committed | -- | X | X |
| Repeatable read | -- | -- | X |
| Serializable | -- | -- | -- |

**Read uncommitted:** Transactions are not isolated from each other. If the DBMS supports other transaction isolation levels, it ignores whatever mechanism it uses to implement those levels. So that they do not adversely affect other transactions, transactions running at the Read Uncommitted level are usually read-only.

**Read committed:** The transaction waits until rows write-locked by other transactions are unlocked; this prevents it from reading any "dirty" data.

The transaction holds a read lock (if it only reads the row) or write lock (if it updates or deletes the row) on the current row to prevent other transactions from updating or deleting it. The transaction releases read locks when it moves off the current row. It holds write locks until it is committed or rolled back.

**Repeatable read:** The transaction waits until rows write-locked by other transactions are unlocked; this prevents it from reading any "dirty" data.

The transaction holds read locks on all rows it returns to the application and write locks on all rows it inserts, updates, or deletes. For example, if the transaction includes the SQL statement **SELECT \* FROM Orders**, the transaction read-locks rows as the application fetches them. If the transaction includes the SQL statement **DELETE FROM Orders WHERE Status = 'CLOSED'**, the transaction write-locks rows as it deletes them.

Because other transactions cannot update or delete these rows, the current transaction avoids any nonrepeatable reads. The transaction releases its locks when it is committed or rolled back.

**Serializable:** The transaction waits until rows write-locked by other transactions are unlocked; this prevents it from reading any "dirty" data.

The transaction holds a read lock (if it only reads rows) or write lock (if it can update or delete rows) on the range of rows it affects. For example, if the transaction includes the SQL statement **SELECT \* FROM Orders**, the range is the entire Orders table; the transaction read-locks the table and does not allow any new rows to be inserted into it. If the transaction includes the SQL statement **DELETE FROM Orders WHERE Status = 'CLOSED'**, the range is all rows with a Status of "CLOSED"; the transaction write-locks all rows in the Orders table with a Status of "CLOSED" and does not allow any rows to be inserted or updated such that the resulting row has a Status of "CLOSED".

Because other transactions cannot update or delete the rows in the range, the current transaction avoids any nonrepeatable reads. Because other transactions cannot insert any rows in the range, the current transaction avoids any phantoms. The transaction releases its lock when it is committed or rolled back.

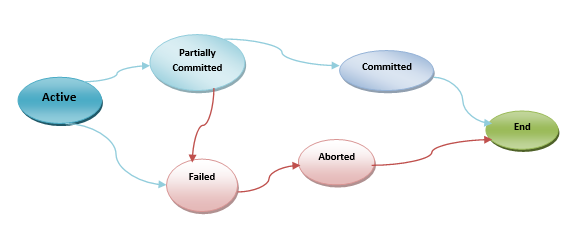
**2.3 Transaction states:** A transaction in a database can be one of the following 5 stages.

* **Active**
* **Partially committed**
* **Failed**
* **Aborted**
* **Committed**

**Active:** This is the first state of transaction and here the transaction is being executed. For example, updating or inserting or deleting a record is done here. But it is still not saved to the database. When we say transaction it will have set of small steps, and those steps will be executed here.

**Partially Committed:** This is also an execution phase where last step in the transaction is executed. But data is still not saved to the database. In our example of calculating total marks, final display the total marks step is executed in this state.

**Committed**: In this state, all the transactions are permanently saved to the database. This step is the last step of a transaction, if it executes without fail.



**Failed**: If a transaction cannot proceed to the execution state because of the failure of the system or database, then the transaction is said to be in failed state. In the total mark calculation example, if the database is not able fire a query to fetch the marks, i.e.; very first step of transaction, then the transaction will fail to execute.

**Aborted**: If a transaction is failed to execute, then the database recovery system will make sure that the database is in its previous consistent state. If not, it brings the database to consistent state by aborting or rolling back the transaction. If the transaction fails in the middle of the transaction, all the executed transactions are rolled back to it consistent state before executing the transaction. Once the transaction is aborted it is either restarted to execute again or fully killed by the DBMS.

**2.4 Transaction Control**

There are three transaction controls described below:

**COMMIT:** The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command.

**ROLLBACK:** The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database.

The ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

**SAVEPOINT:** A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

**Aborting a transaction**

A multi-user transaction processing system must provide the atomic property of transactions even in the presence of various types of failures. This property is referred to as failure atomicity.  A transaction could be aborted by its initiator.

For example, if the withdrawal of two bank accounts would result in a negative balance in one of them then the transaction may be aborted.  Aborts may also occur as a result of other system related faults such as divide-by-zero, memory fault, or deadlock.  There can be other kinds of more drastic failures that may occur in a system, e.g., crash failures, disk or media failure, network failure, etc.  In such situations the operations executed by interrupted transactions must be “undone”. That is, the effect of aborted transactions should be as if those transactions were never executed.  Similarly, when a system failure occurs we must ensure that the effects of committed transactions are not lost.

Failure atomicity of transactions in the absence of concurrency can be easily accomplished by using a simple bookkeeping technique.  Read operations do not require any undo action whereas write operations require restoration of the before-image of the data objects involved in the write operations.  This can be accomplished by requiring that transactions save the before-image of each object on nonvolatile storage before performing a write operation on that object.  Before-images are usually stored in a structure called a log on nonvolatile storage.  Guaranteeing failure atomicity of transactions in a concurrent or distributed environment is more complex.  A distributed transaction uses an atomic commit protocol to ensure that all sites involved commit the transaction or all sites abort the transaction.  In a concurrent environment, when a transaction aborts, all its effects on the data as well as other concurrently executing transactions should be eliminated.

Now we consider the problem of ensuring failure atomicity of transactions in the presence of system failures.  Transaction processing systems usually use two types of memory, volatile, which is usually small but has fast access time, and nonvolatile, such as a disk, which has slower access time, but has much larger capacity.  The data is stored in nonvolatile memory, and part of the data is cached or buffered in the volatile memory.  A transaction usually executes by reading and updating the cache.  Ideally, when a transaction commits, all its changes should be incorporated into the data on nonvolatile memory.  Since disk accesses are generally slower, coherency between the cache and the data itself is not strictly maintained.  Hence, the nonvolatile data may contain data written by uncommitted transactions and the cache may contain data, which has not yet been incorporated into the nonvolatile memory, but that was written by committed transactions.  A system failure is modeled by assuming that the state of the system in the volatile memory is lost, but the state in the nonvolatile memory survives following the system failure.  The recovery from system failure involves the following:

1.    The effects of transactions that were in committed state at the time of the system failure must be incorporated into the data.

2.    The effects of transactions that were aborted or were active at the time of the system failure must be eliminated. Note that transactions that were active when the failure occurs are considered aborted since their internal state is lost due to the failure.

Recovery from system failures can be categorized into two processes:

Redo Process -- For each committed transaction, redo the write operations that are not incorporated into the nonvolatile data.

 Undo Process -- For each active and aborted transaction, undo the write operations that were incorporated into the nonvolatile data.

In order to successfully recover from system failures, additional bookkeeping needs to be performed.  In particular, the system maintains on nonvolatile storage, a list of committed transactions, a list of aborted transactions, and a log of write operations performed on each data object.  A restart procedure is invoked when the system recovers after a system failure and the procedure recovers the persistent state by using the two lists and the log.

**3. Concurrency Control**

In a multiprogramming environment where multiple transactions can be executed simultaneously, it is highly important to control the concurrency of transactions. We have concurrency control protocols to ensure atomicity, isolation, and serializability of concurrent transactions.

**3.1** Concurrency Control protocols

A lock is a mechanism to control concurrent access to a data item. Lock requests are made to the concurrency-control manager by the programmer. Transaction can proceed only after request is granted.

Data items can be locked in two modes:

1. Exclusive (X) mode. Data item can be both read as well as written. X-lock is requested using lock-X instruction.

2. Shared (S) mode. Data item can only be read. S-lock is requested using lock-S instruction.

A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions. If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted. Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

Example of a transaction performing locking:

**T2:** lock-S(A);

read (A);

unlock(A);

lock-S(B);

read (B);

unlock(B);

display(A+B)

3.2.1 Lock-Based Concurrency Control Protocols

A locking protocol is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Let us see how these locking mechanisms help us to create error free schedules. An example of an erroneous schedule:

|  |  |
| --- | --- |
| T1 | T2 |
|  |  |
| Read A; |  |
| A = A - 100; |  |
|  | Read A; |
|  | Temp = A \* 0.1; |
|  | Read C; |
|  | C = C + Temp; |
|  | Write C; |
| Write A; |  |
| Read B; |  |
| B = B + 100; |  |
| Write B; |  |

We detected the error based on common sense only that the Context Switching is being performed before the new value has been updated in A. T2 reads the old value of A, and thus deposits a wrong amount in C. Had we used the locking mechanism, this error could never have occurred. Let us rewrite the schedule using the locks.

|  |  |
| --- | --- |
| T1 | T2 |
|  |  |
| Lock-X (A) |  |
| Read A; |  |
| A = A - 100; |  |
| Write A; |  |
| Unlock (A) |  |
|  | Lock-S (A) |
|  | Read A; |
|  | Temp = A \* 0.1; |
|  | Unlock (A) |
|  | Lock-X(C) |
|  | Read C; |
|  | C = C + Temp; |
|  | Write C; |
|  | Unlock (C) |
| Lock-X (B) |  |
| Read B; |  |
| B = B + 100; |  |
| Write B; |  |
| Unlock (B) |  |

And this automatically becomes a very correct schedule. We need not apply any manual effort to detect or correct the errors that may creep into the schedule if locks are not used in them.

3.2.2 The two phase locking protocol

A transaction is two-phase locked if:

* before reading x, it sets a read lock on x
* before writing x, it sets a write lock on x
* it holds each lock until after it executes the corresponding operation
* after its first unlock operation, it requests no new locks

This protocol ensures conflict-serializable schedules.

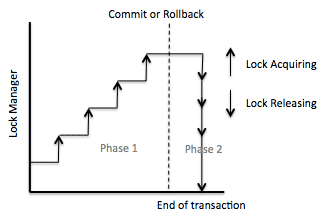
**Growing Phase:** In this phase the transaction can only acquire locks, but cannot release any lock. The transaction enters the growing phase as soon as it acquires the first lock it wants. From now on it has no option but to keep acquiring all the locks it would need. It cannot release any lock at this phase even if it has finished working with a locked data item. Ultimately the transaction reaches a point where all the lock it may need has been acquired. This point is called Lock Point.

**Shrinking Phase:** After Lock Point has been reached, the transaction enters the shrinking phase. In this phase the transaction can only release locks, but cannot acquire any new lock. The transaction enters the shrinking phase as soon as it releases the first lock after crossing the Lock Point. From now on it has no option but to keep releasing all the acquired locks.  
There are two different versions of the Two Phase Locking Protocol. One is called the Strict Two Phase Locking Protocol and the other one is called the Rigorous Two Phase Locking Protocol.

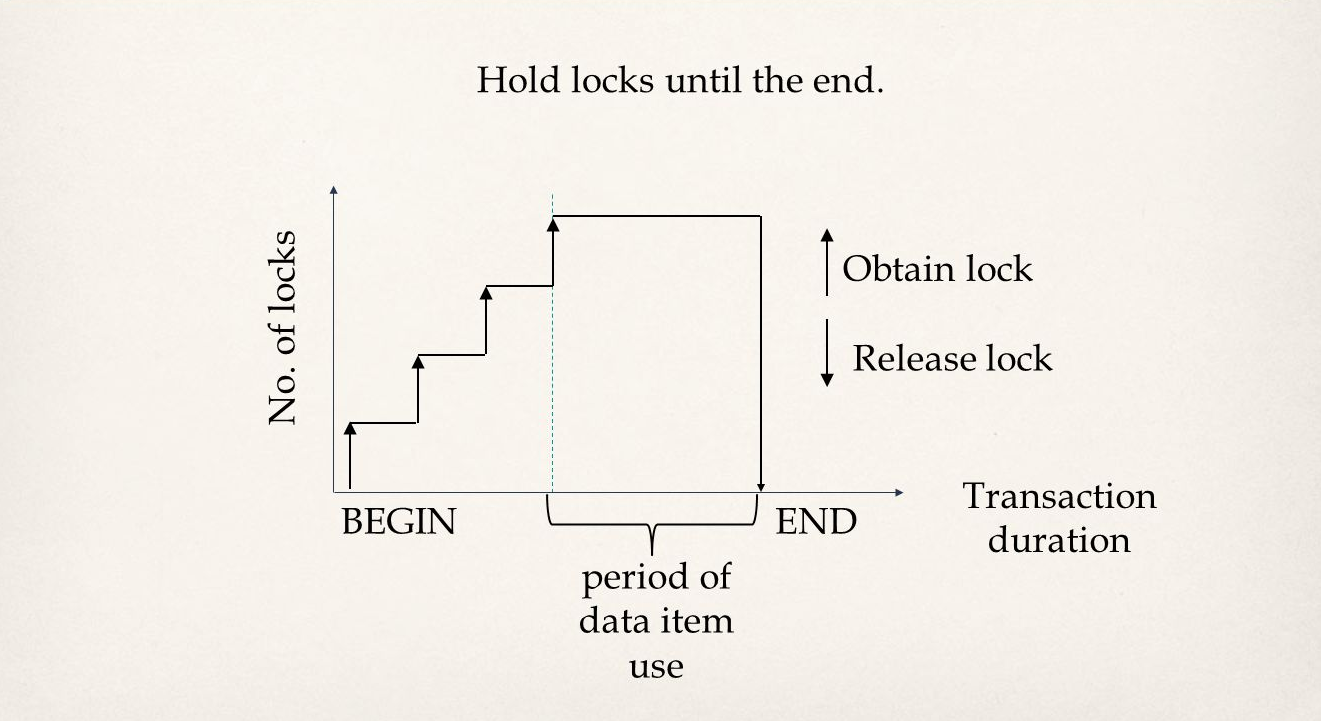
There can be conflict serializable schedules that cannot be obtained if two-phase locking is used. However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:

Given a transaction Ti that does not follow two-phase locking, we can find a transaction Tj that uses two-phase locking, and a schedule for Ti and Tj that is not conflict serializable.

The lock point is the moment when transitioning from the growing phase to th e shrinking phase.



strict 2PL locking protocol Holds the locks till the end of the transaction and Cascading aborts are avoided



**Example:** The schedule S of the previous example is not valid in the 2PL protocol:

S = {wl 1 ( x ), R 1 ( x ), W1 ( x ), lr 1 ( x )

wl 2 ( x ), R 2 ( x ), W2 ( x ), lr 2 ( x )

wl 2 ( y ), R 2 ( y ), W2 ( y ), lr 2 ( y )

wl 1 ( y ), R 1 ( y ), W1 ( y ), lr 1 ( y ) }

after lr 1 ( x ) (in line 1) transaction T1 cannot request the lock wl 1 ( y ) (in line 4), Valid schedule in the 2PL protocol.

S = {wl 1 ( x ), R 1 ( x ), W1 ( x ),

wl 1 ( y ), R 1 ( y ), W1 ( y ), lr 1 ( x ), lr 1 ( y )

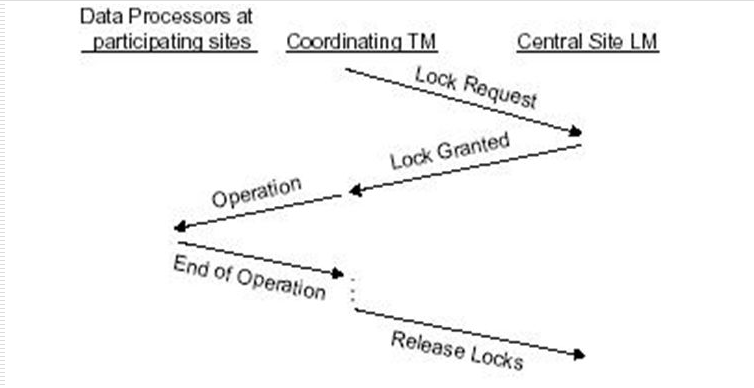
wl 2 ( x ), R 2 ( x ), W2 ( x ),

wl 2 ( y ), R 2 ( y ), W2 ( y ), lr 2 ( x ), lr 2 ( y ) }

**Various extensions of the 2PL:**

**Centralized 2PL**

* A single site is responsible for the lock management, i.e., one lock manager for the whole DDBMS
* Lock requests are issued to the lock manager
* Coordinating transaction manager (TM at site where the transaction is initiated) can make all locking requests on behalf of local transaction managers

****

**Advantage:** Easy to implement

**Disadvantages:**

* Bottlenecks and lower reliability
* Replica control protocol is additionally needed if data are replicated (see also primary copy 2PL)

**Primary Copy 2PL**

* Several lock managers are distributed to a number of sites
* Each lock manager is responsible for managing the locks for a set of data items
* For replicated data items, one copy is chosen as primary copy, others are slave copies
* Only the primary copy of a data item that is updated needs to be write-locked
* Once primary copy has been updated, the change is propagated to the slaves

**Advantages**

* Lower communication costs and better performance than the centralized 2PL

**Disadvantages**

* Deadlock handling is more complex

**Distributed 2PL**

* Lock managers are distributed to all sites
* Each lock manager responsible for locks for data at that site
* If data is not replicated, it is equivalent to primary copy 2PL
* If data is replicated, the Read-One-Write-All (ROWA) replica control protocol is implemented

Read(x): Any copy of a replicated item x can be read by obtaining a read lock on the copy

Write(x): All copies of x must be write-locked before x can be updated

**Disadvantages**

Deadlock handling more complex – Communication costs higher than primary copy 2PL

**Communication structure of the distributed 2PL**

* The coordinating TM sends the lock request to the lock managers of all participating sites
* The LMs pass the operations to the data processors
* The end of the operation is signaled to the coordinating TM

**Rigorous Two Phase Locking Protocol**

In Rigorous Two Phase Locking Protocol, a transaction is not allowed to release any lock (either shared or exclusive) until it commits. This means that until the transaction commits, other transaction might acquire a shared lock on a data item on which the uncommitted transaction has a shared lock; but cannot acquire any lock on a data item on which the uncommitted transaction has an exclusive lock.

**Timestamp Ordering Protocol**

A timestamp is a tag that can be attached to any transaction or any data item, which denotes a specific time on which the transaction or data item had been activated in any way. We, who use computers, must all be familiar with the concepts of “Date Created” or “Last Modified” properties of files and folders. Well, timestamps are things like that.

A timestamp can be implemented in two ways. The simplest one is to directly assign the current value of the clock to the transaction or the data item. The other policy is to attach the value of a logical counter that keeps incrementing as new timestamps are required.  
The timestamp of a transaction denotes the time when it was first activated. The timestamp of a data item can be of the following two types:

**W-timestamp (Q)**: This means the latest time when the data item Q has been written into.  
**R-timestamp (Q)**: This means the latest time when the data item Q has been read from.

These two timestamps are updated each time a successful read/write operation is performed on the data item Q.

The timestamp ordering protocol ensures that any pair of conflicting read/write operations will be executed in their respective timestamp order. This is an alternative solution to using locks.

**For Read operations:**

1. If TS (T) < W-timestamp (Q), then the transaction T is trying to read a value of data item Q which has already been overwritten by some other transaction. Hence the value which T wanted to read from Q does not exist there anymore, and T would be rolled back.
2. If TS (T) >= W-timestamp (Q), then the transaction T is trying to read a value of data item Q which has been written and committed by some other transaction earlier. Hence T will be allowed to read the value of Q, and the R-timestamp of Q should be updated to TS (T).

**For Write operations:**

1. If TS (T) < R-timestamp (Q), then it means that the system has waited too long for transaction T to write its value, and the delay has become so great that it has allowed another transaction to read the old value of data item Q. In such a case T has lost its relevance and will be rolled back.
2. Else if TS (T) < W-timestamp (Q), then transaction T has delayed so much that  the system has allowed another transaction to write into the data item Q. in such a case too, T has lost its relevance and will be rolled back.
3. Otherwise the system executes transaction T and updates the W-timestamp of Q to TS (T).

3.2.2 Concurrency control in Real-time web applications

**4. Deadlock Principles**

Deadlock is a state when set of process depending on each other and awaiting for-ever. A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set. Deadlocks are not healthy for a system. In case a system is stuck in a deadlock, the transactions involved in the deadlock are either rolled back or restarted.

For example, assume a set of transactions {T0, T1, T2, ...,Tn}. T0 needs a resource X to complete its task. Resource X is held by T1, and T1 is waiting for a resource Y, which is held by T2. T2 is waiting for resource Z, which is held by T0. Thus, all the processes wait for each other to release resources. In this situation, none of the processes can finish their task. This situation is known as a deadlock.

There are three general approaches dealing with deadlock:

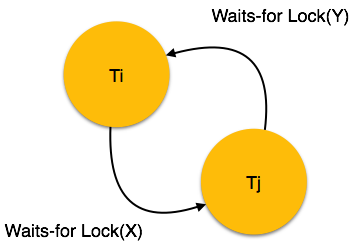
* Avoid deadlock
* Prevent deadlock
* Detect deadlock

**4.1 Deadlock Avoidance**

Aborting a transaction is not always a practical approach. Instead, deadlock avoidance mechanisms can be used to detect any deadlock situation in advance. Methods like "wait-for graph" are available but they are suitable for only those systems where transactions are lightweight having fewer instances of resource. In a bulky system, deadlock prevention techniques may work well.

Wait-for graph is a simple method available to track if any deadlock situation may arise. For each transaction entering into the system, a node is created. When a transaction Ti requests for a lock on an item, say X, which is held by some other transaction Tj, a directed edge is created from Ti to Tj. If Tj releases item X, the edge between them is dropped and Ti locks the data item.

The system maintains this wait-for graph for every transaction waiting for some data items held by others. The system keeps checking if there's any cycle in the graph.



Here, we can use any of the two following approaches −

First, do not allow any request for an item, which is already locked by another transaction. This is not always feasible and may cause starvation, where a transaction indefinitely waits for a data item and can never acquire it.

The second option is to roll back one of the transactions. It is not always feasible to roll back the younger transaction, as it may be important than the older one. With the help of some relative algorithm, a transaction is chosen, which is to be aborted. This transaction is known as the victim and the process is known as victim selection.

**4.2 Deadlock Prevention**

To prevent any deadlock situation in the system, the DBMS aggressively inspects all the operations, where transactions are about to execute. The DBMS inspects the operations and analyzes if they can create a deadlock situation. If it finds that a deadlock situation might occur, then that transaction is never allowed to be executed.

There are deadlock prevention schemes that use timestamp ordering mechanism of transactions in order to predetermine a deadlock situation.

**Wait-Die Scheme**

In this scheme, if a transaction requests to lock a resource (data item), which is already held with a conflicting lock by another transaction, then one of the two possibilities may occur

* If TS(Ti) < TS(Tj) − that is Ti, which is requesting a conflicting lock, is older than Tj − then Ti is allowed to wait until the data-item is available.
* If TS(Ti) > TS(tj) − that is Ti is younger than Tj − then Ti dies. Ti is restarted later with a random delay but with the same timestamp.

This scheme allows the older transaction to wait but kills the younger one.

**Wound-Wait Scheme**

In this scheme, if a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur −

* If TS(Ti) < TS(Tj), then Ti forces Tj to be rolled back − that is Ti wounds Tj. Tj is restarted later with a random delay but with the same timestamp.
* If TS(Ti) > TS(Tj), then Ti is forced to wait until the resource is available.

This scheme, allows the younger transaction to wait; but when an older transaction requests an item held by a younger one, the older transaction forces the younger one to abort and release the item.

In both the cases, the transaction that enters the system at a later stage is aborted.

**4.1 Deadlock Detection**

Deadlock detection strategies have limit access to resources and impose restrictions on processes. These strategies oppose resource requests are granted whenever possible and regularly check for deadlock.

**Deadlock Detection Algorithm**

1. Let Work and Finish be vectors of length m and n, respectively

Initialize:

(a) Work = Available

(b) For i = 1,2, …, n, if Allocationi ≠ 0, then

Finish[i] = false;otherwise, Finish[i] = true.

1. Find an index i such that both:

(a) Finish[i] == false

(b) Requesti ≤ Work

If no such i exists, go to step 4.

1. Work = Work + Allocationi

Finish[i] = true

go to step 2.

1. If Finish[i] == false, for some i, 1 ≤ i ≤ n, then the system is in deadlock state. Moreover, if Finish[i] == false, then Pi is deadlocked.

Example of Detection Algorithm

* Five processes P0 through P4. Three resource types A (7 instances), B (2 instances), and C (6 instances).
* Snapshot at time T0:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Allocation** | **Request** | **Available** |
|  | A B C | A B C | A B C |
| P0 | 0 1 0 | 0 0 0 | 0 0 0 |
| P1 | 2 0 0 | 2 0 2 |  |
| P2 | 3 0 3 | 0 0 0 |  |
| P3 | 2 1 1 | 1 0 0 |  |
| P4 | 0 0 2 | 0 0 2 |  |

* Sequence <P0, P2, P3, P1, P4> will result in Finish[i] = true for all i.
* P2 requests an additional instance of type C.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Request** |  |  |
|  | A B C |  |  |
| P0 | 0 0 0 |  |  |
| P1 | 2 0 1 |  |  |
| P2 | 0 0 1 |  |  |
| P3 | 1 0 0 |  |  |
| P4 | 1. 0 2 |  |  |

* State of system can reclaim resources held by process P0, but insufficient resources to fulfill other processes; requests.
* Deadlock exists, consisting of processes P1, P2, P3, and P4.

Recovery from Deadlock

* Abort all deadlocked processes
* Abort one process at a time until the deadlock cycle is eliminated.
* In which order should we choose to abort?
* Priority of the process.
* How long process has computed, and how much longer to completion.
* Resources the process has used.
* Resources process needs to complete.
* How many processes will need to be terminated.
* Is process interactive or batch?
* Selecting a victim – minimize cost.
* Rollback – return to some safe state, restart process for that state.
* Starvation – same process may always be picked as victim, include number of rollback in cost factor.

**Advantages and Disadvantages of Deadlock strategies**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Approach | Resource Allocation Policy | Different Schemes |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

1. **Optimistic & Pessimistic concurrency control**

**Optimistic concurrency control** (OCC) is a concurrency control method applied to transactional systems such as relational database management systems and software transactional memory. OCC assumes that multiple transactions can frequently complete without interfering with each other. While running, transactions use data resources without acquiring locks on those resources. Before committing, each transaction verifies that no other transaction has modified the data it has read. If the check reveals conflicting modifications, the committing transaction rolls back and can be restarted.

OCC is generally used in environments with low data contention. When conflicts are rare, transactions can complete without the expense of managing locks and without having transactions wait for other transactions locks to clear, leading to higher throughput than other concurrency control methods. However, if contention for data resources is frequent, the cost of repeatedly restarting transactions hurts performance significantly. it is commonly thought that other concurrency control methods have better performance under these conditions. However, locking-based ("pessimistic") methods also can deliver poor performance because locking can drastically limit effective concurrency even when deadlocks are avoided.

More specifically, OCC transactions involve these phases:

**Begin**: Record a timestamp marking the transaction's beginning.

**Modify**: Read database values, and tentatively write changes.

**Validate**: Check whether other transactions have modified data that this transaction has used (read or written). This includes transactions that completed after this transaction's start time, and optionally, transactions that are still active at validation time.

* It uses transaction timestamps
* write\_sets and read\_sets maintained
* To check that TransA does not interfere with TransB the following must hold:
  + TransB completes its write phase before TransA starts its reads phase
  + TransA starts its write phase after TransB completes its write phase, and the read set of TransA has no items in common with the write set of TransB
  + Both the read set and the write set of TransA have no items in common with the write set of TransB, and TransB completes its read phase before TransA completes its read phase.

**Commit/Rollback**: If there is no conflict, make all changes take effect. If there is a conflict, resolve it, typically by aborting the transaction, although other resolution schemes are possible. Care must be taken to avoid a TOCTTOU bug, particularly if this phase and the previous one are not performed as a single atomic operation.

OCC assumes that although conflicts are possible, they will be very rare. Instead of locking every record every time that it is used, the system merely looks for indications that two users actually did try to update the same record at the same time. If that evidence is found, then one user's updates are discarded and the user is informed.

For example, if User1 updates a record and User2 only wants to read it, then User2 simply reads whatever data is on the disk and then proceeds, without checking whether the data is locked. User2 might see slightly out-of-date information if User1 has read the data and updated it, but has not yet committed the transaction.

Elastic search is distributed. When documents are created, updated, or deleted, the new version of the document has to be replicated to other nodes in the cluster. Elastic search is also asynchronous and concurrent, meaning that these replication requests are sent in parallel, and may arrive at their destination out of sequence. Elastic search needs a way of ensuring that an older version of a document never overwrites a newer version.

**Pessimistic concurrency control** (or pessimistic locking) is called "pessimistic" because the system assumes the worst. It assumes that two or more users will want to update the same record at the same time, and then prevents that possibility by locking the record, no matter how unlikely conflicts actually are.

The locks are placed as soon as any piece of the row is accessed, making it impossible for two or more users to update the row at the same time. Depending on the lock mode (shared, exclusive, or update), other users might be able to read the data even though a lock has been placed.

**Choosing concurrency control mechanism**:

In most scenarios, optimistic concurrency control is more efficient and offers higher performance. When choosing between pessimistic and optimistic locking, consider the following:

Pessimistic locking is useful if there are a lot of updates and relatively high chances of users trying to update data at the same time.

For example, if each operation can update a large number of records at a time (the bank might add interest earnings to every account at the end of each month), and two applications are running such operations at the same time, they will have conflicts.

Pessimistic concurrency control is also more appropriate in applications that contain small tables that are frequently updated. In the case of these so-called hotspots, conflicts are so probable that optimistic concurrency control wastes effort in rolling back conflicting transactions.

If you are migrating an application from another DBMS that uses pessimistic locking, you should use the pessimistic mode with solidDB too. Using pessimistic mode in solidDB means that you do not have make changes to the application.

Optimistic locking is useful if the possibility for conflicts is very low – there are many records but relatively few users, or very few updates and mostly read-type operations.

**Conclusion**

In conclusion, concurrency control is one of the primary mechanisms in transaction management to provide integrity of data and safety in DBMS. Today, with hundred thousand or more transactions in a few minutes, transaction management and concurrency control become much more complex and sophisticated. Two pessimistic and optimistic mechanisms are still popular, but other techniques such as semi-optimistic are also applied in DBMS for higher performance, better throughput, more accurate results, and faster run time.

Optimistic Concurrency Control is superior to locking methods for systems where transaction conflict is highly unlikely. It avoids locking overhead and use parallel validation OCC can take full advantage of multiprocessor environment.

Transaction management deals with two key requirements of any database system:

**Resilience**

In the ability of data surviving hardware crashes and software errors without sustaining loss or becoming inconsistent

**Access Control**

In the ability to permit simultaneous access of data multiple users in a consistent manner and assuring only authorized access

**References**

Connolly, T. M., & Begg, C. E. (2014). Database Systems: A Practical Approach to Design, Implementation and Management. New Jersey, NJ: Pearson.

Larson, P. Å., Blanas, S., Diaconu, C., Freedman, C., Patel, J. M., & Zwilling, M. (2011). High-performance concurrency control mechanisms for main-memory databases. Proceedings of the VLDB Endowment, 5(4), 298-309.

Kung, H. T., & Robinson, J. (1981). Carnegie-Mellon University, Retrieved from http://www.csd.uoc.gr/~hy460/pdf/kung.pdf.

Vallejo, E., Sanyal, S., Harris, T., Vallejo, F., Beivide, R., Unsal, O., & ... Valero, M. (2011). Hybrid Transactional Memory with Pessimistic Concurrency Control. International Journal of Parallel Programming, 39(3), 375-396. doi:10.1007/s10766-010-0158-x

Wikipedia (2014). Optimistic Concurrency Control. Retrieved from http://en.wikipedia.org/wiki/Optimistic\_concurrency\_control.