ST.XAVIER’S COLLEGE

**Maitighar, Kathmandu**



**DBMS LAB ASSIGNMENT**

**Submitted By:**

**Narayan Panthi**

013BSCCSIT024

**Submitted To:**

**Er. Sanjay Kumar Yadav** ………..…………….

**Lecturer**

**Department of Computer Science**

**Date of Submission:- *2015 -*** *\_\_****\_-*** *\_\_\_*

## Concurrency Control

* + Most DBMS are multi-user systems.
  + The concurrent execution of many different transactions submitted by various users must be organised such that each transaction does not interfere with another transaction with one another in a way that produces incorrect results.
  + The concurrent execution of transactions must be such that each transaction appears to execute in isolation.
  + Recovery
  + System failures, either hardware or software, must not result in an inconsistent database
* If an error or hardware/software crash occurs between the begin and end, the database will be inconsistent
  + Computer Failure (system crash)
  + A transaction or system error
  + Local errors or exception conditions detected by the transaction
  + Concurrency control enforcement
  + Disk failure
  + Physical problems and catastrophes
* The database is restored to some state from the past so that a correct state—close to the time of failure—can be reconstructed from the past state.
* A DBMS ensures that if a transaction executes some updates and then a failure occurs before the transaction reaches normal termination, then those updates are undone.
* The statements COMMIT and ROLLBACK (or their equivalent) ensure Transaction Atomicity

## PURPOSE OF CONCURRENCY CONTROL

* To enforce Isolation (through mutual exclusion) among conflicting transactions.
* To preserve database consistency through consistency preserving execution of transactions.
* To resolve read-write and write-write conflicts.

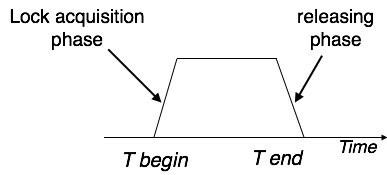
## TWO PHASE LOCKING

In [databases](https://en.wikipedia.org/wiki/Database) and [transaction processing](https://en.wikipedia.org/wiki/Transaction_processing), **two-phase locking** (**2PL**) is a [concurrency control](https://en.wikipedia.org/wiki/Concurrency_control) method that guarantees [serializability](https://en.wikipedia.org/wiki/Serializability). It is also the name of the resulting set of [database transaction](https://en.wikipedia.org/wiki/Database_transaction) [schedules](https://en.wikipedia.org/wiki/Schedule_%28computer_science%29) (histories). The protocol utilizes [locks](https://en.wikipedia.org/wiki/Lock_%28computer_science%29), applied by a transaction to data, which may block (interpreted as signals to stop) other transactions from accessing the same data during the transaction's life.

By the 2PL protocol locks are applied and removed in two phases:

1. Expanding phase: locks are acquired and no locks are released.
2. Shrinking phase: locks are released and no locks are acquired.

This locking protocol divides the execution phase of a transaction into three parts. In the first part, when the transaction starts executing, it seeks permission for the locks it requires. The second part is where the transaction acquires all the locks. As soon as the transaction releases its first lock, the third phase starts. In this phase, the transaction cannot demand any new locks; it only releases the acquired locks.



Two-phase locking has two phases, one is **growing**, where all the locks are being acquired by the transaction; and the second phase is shrinking, where the locks held by the transaction are being released.

To claim an exclusive (write) lock, a transaction must first acquire a shared (read) lock and then upgrade it to an exclusive lock.

## LIMITATION OF CCMS

* Threads can be expensive. Overhead of scheduling, context-switching and synchronization.
* Concurrent programs can run slower than their sequential counterparts even with multiple CPUs

## TIME-STAMP-BASED PROTOCOL

The most commonly used concurrency protocol is the timestamp based protocol. This protocol uses either system time or logical counter as a timestamp.

Lock-based protocols manage the order between the conflicting pairs among transactions at the time of execution, whereas timestamp-based protocols start working as soon as a transaction is created.

Every transaction has a timestamp associated with it, and the ordering is determined by the age of the transaction. A transaction created at 0002 clock time would be older than all other transactions that come after it. For example, any transaction 'y' entering the system at 0004 is two seconds younger and the priority would be given to the older one.

In addition, every data item is given the latest read and write-timestamp. This lets the system know when the last ‘read and write’ operation was performed on the data item.

### Time stamp based protocols

The most commonly used concurrency protocol is the timestamp based protocol. This protocol uses either system time or logical counter as a timestamp.

Lock-based protocols manage the order between the conflicting pairs among transactions at the time of execution, whereas timestamp-based protocols start working as soon as a transaction is created.

Every transaction has a timestamp associated with it, and the ordering is determined by the age of the transaction. A transaction created at 0002 clock time would be older than all other transactions that come after it. For example, any transaction 'y' entering the system at 0004 is two seconds younger and the priority would be given to the older one.

In addition, every data item is given the latest read and write-timestamp. This lets the system know when the last ‘read and write’ operation was performed on the data item.

# INDEX LOCKING

* Every relation must have at least one index. Access to a relation must be made only through one of the indices on the relation.
* A transaction *Ti* that performs a lookup must lock all the index buckets that it accesses, in S-mode.
* A transaction *Ti* may not insert a tuple *ti* into a relation *r*  without updating all indices to *r*.
* *Ti* must perform a lookup on every index to find all index buckets that could have possibly contained a pointer to tuple *ti*, had it existed already, and obtain locks in X-mode on all these index buckets. *Ti* must also obtain locks in X-mode on all index buckets that it modifies.
* The rules of the two-phase locking protocol must be observed.

## LOCK GRANULARITY

The granularity of locks in a database refers to how much of the data is locked at one time. In theory, a database server can lock as much as the entire database or as little as one column of data. Such extremes affect the concurrency (number of users that can access the data) and locking overhead (amount of work to process lock requests) in the server. Adaptive Server supports locking at the table, page, and row level.

By locking at higher levels of granularity, the amount of work required to obtain and manage locks is reduced. If a query needs to read or update many rows in a table:

* It can acquire just one table-level lock
* It can acquire a lock for each page that contained one of the required rows
* It can acquire a lock on each row

Less overall work is required to use a table-level lock, but large-scale locks can degrade performance, by making other users wait until locks are released. Decreasing the lock size makes more of the data accessible to other users. However, finer granularity locks can also degrade performance, since more work is necessary to maintain and coordinate the increased number of locks. To achieve optimum performance, a locking scheme must balance the needs of concurrency and overhead.

## DEADLOCK HANDLING DETECTION AND RESOLUTION

* System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
* *Deadlock prevention* protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
  + Require that each transaction locks all its data items before it begins execution (predeclaration).
  + Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

**Deadlock Prevention Strategies**

* **wait-die** scheme — non-preemptive
  + older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
  + a transaction may die several times before acquiring needed data item
* **wound-wait** scheme — preemptive
  + older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  + may be fewer rollbacks than *wait-die* scheme.
* Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.
* Timeout-Based Schemes :
  + a transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  + thus deadlocks are not possible
  + simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

**Lock Granularity**

The granularity of locks in a database refers to how much of the data is locked at one time. In theory, a database server can lock as much as the entire database or as little as one column of data. Such extremes affect the concurrency (number of users that can access the data) and locking overhead (amount of work to process lock requests) in the server. Adaptive Server supports locking at the table, page, and row level.

By locking at higher levels of granularity, the amount of work required to obtain and manage locks is reduced. If a query needs to read or update many rows in a table:

* It can acquire just one table-level lock
* It can acquire a lock for each page that contained one of the required rows
* It can acquire a lock on each row