ST. XAVIER’S COLLEGE

**Maitighar,Kathmandu**

**(Affiliated to Tribhuvan University)**



**Database Management System**

**Lab Assignment #11**

**Submitted By**

Arun Sanjel

B.Sc. CSIT

Year II/IV Semester

013BSCIT010

**Submitted To**

Er. Sanjay Kumar Yadav

Lecturer,

Department of Computer Science

St. Xavier’s College

Maitighar, Kathmandu

**Submitted On**

October 30, 2015

**Database Concurrency Control**

1. **Purpose of Concurrency Control**

Concurrency control is a database management systems (DBMS) concept that is used to address conflicts with the simultaneous accessing or altering of data that can occur with a multi-user system. Concurrency control, when applied to a DBMS, is meant to coordinate simultaneous transactions while preserving data integrity. The Concurrency is about to control the multi-user access of Database.

**Why do we need a Concurrency Model?**

Pessimistic Locking: This concurrency control strategy involves keeping an entity in a database locked the entire time it exists in the database's memory. This limits or prevents users from altering the data entity that is locked. There are two types of locks that fall under the category of pessimistic locking: write lock and read lock.

With write lock, everyone but the holder of the lock is prevented from reading, updating, or deleting the entity. With read lock, other users can read the entity, but no one except for the lock holder can update or delete it.

Optimistic Locking: This strategy can be used when instances of simultaneous transactions, or collisions, are expected to be infrequent. In contrast with pessimistic locking, optimistic locking doesn't try to prevent the collisions from occurring. Instead, it aims to detect these collisions and resolve them on the chance occasions when they occur.

Pessimistic locking provides a guarantee that database changes are made safely. However, it becomes less viable as the number of simultaneous users or the number of entities involved in a transaction increase because the potential for having to wait for a lock to release will increase.

Optimistic locking can alleviate the problem of waiting for locks to release, but then users have the potential to experience collisions when attempting to update the database.

1. **Two Phase Locking**

Two phase locking is a process used to gain ownership of shared resources without creating the possibility for deadlock. The technique is extremely simple, and breaks up the modification of shared data into "two phases", this is what gives the process its name.

There are actually three activities that take place in the "two phase" update algorithm:

* Lock Acquisition
* Modification of Data
* Release Locks

The modification of data, and the subsequent release of the locks that protected the data are generally grouped together and called the second phase.

Two phase locking prevents deadlock from occurring in distributed systems by releasing all the resources it has acquired, if it is not possible to obtain all the resources required without waiting for another process to finish using a lock. This means that no process is ever in a state where it is holding some shared resources, and waiting for another process to release a shared resource which it requires. This means that deadlock cannot occur due to resource contention.

The resource (or lock) acquisition phase of a "two phase" shared data access protocol is usually implemented as a loop within which all the locks required to access the shared data are acquired one by one. If any lock is not acquired on the first attempt the algorithm gives up all the locks it had previously been able to get, and starts to try to get all the locks again.

This "back-off and re-try" strategy can be a problem in distributed systems. It is not guaranteed to give access to the desired resources within a finite time. This can lead to process starvation, if a single process never acquires all the locks needed for it to continue execution. This is a problem for real-time systems. Consequently, two phase locking protocols cannot be used in hard real-time applications.

1. **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.

1. **Commit Protocols**

Commit protocols are used to ensure atomicity across sites

* 1. a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
  2. not acceptable to have a transaction committed at one site and aborted at another

The *two-phase commit* (2 *PC*) protocol is widely used .The *three-phase commit* (3 *PC*) protocol is more complicated and more expensive, but avoids some drawbacks of two-phase commit protocol.

1. **Index Locking**

Every relation must have at least one index.

A transaction can access tuples only after finding them through one or more indices on the relation

A transaction *Ti* that performs a lookup must lock all the index leaf nodes that it accesses, in S-mode

* Even if the leaf node does not contain any tuple satisfying the index lookup (e.g. for a range query, no tuple in a leaf is in the range)

A transaction *Ti* that inserts, updates or deletes a tuple *ti* in a relation *r*

* must update all indices to *r*
* must obtain exclusive locks on all index leaf nodes affected by the insert/update/delete

The rules of the two-phase locking protocol must be observed

Guarantees that phantom phenomenon won’t occur

1. **Lock Granularity**

* Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
* Can be represented graphically as a tree (but don't confuse with tree-locking protocol)
* When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.
* Granularity of locking (level in tree where locking is done):
  + **fine granularity** (lower in tree): high concurrency, high locking overhead
  + **coarse granularity** (higher in tree): low locking overhead, low concurrency

**Multiple Granularities Locking Scheme**

* Transaction *Ti* can lock a node *Q*, using the following rules:
  1. The lock compatibility matrix must be observed.
  2. The root of the tree must be locked first, and may be locked in any mode.
  3. A node *Q* can be locked by *Ti* in S or IS mode only if the parent of *Q* is currently locked by *Ti* in either IX or IS mode.
  4. A node *Q* can be locked by *Ti* in X, SIX, or IX mode only if the parent of *Q* is currently locked by *Ti* in either IX or SIX mode.
  5. *Ti* can lock a node only if it has not previously unlocked any node (that is, *Ti* is two-phase).
  6. *Ti* can unlock a node *Q* only if none of the children of *Q* are currently locked by *Ti.*
* Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.
* **Lock granularity escalation**: in case there are too many locks at a particular level, switch to higher granularity S or X lock

1. **Time Stamp Ordering Multi-version Concurrency Control**

Basic time stamping is a concurrency control mechanism that eliminates deadlock. This method doesn’t use locks to control concurrency, so it is impossible for deadlock to occur. According to this method a unique timestamp is assigned to each transaction, usually showing when it was started. This effectively allows an age to be assigned to transactions and an order to be assigned. Data items have both a read-timestamp and a write-timestamp. These timestamps are updated each time the data item is read or updated respectively.

Problems arise in this system when a transaction tries to read a data item which has been written by a younger transaction. This is called a late read. This means that the data item has changed since the initial transaction start time and the solution is to roll back the timestamp and acquire a new one. Another problem occurs when a transaction tries to write a data item which has been read by a younger transaction. This is called a late write. This means that the data item has been read by another transaction since the start time of the transaction that is altering it. The solution for this problem is the same as for the late read problem. The timestamp must be rolled back and a new one acquired .

Adhering to the rules of the basic time stamping process allows the transactions to be serialized and a chronological schedule of transactions can then be created. Time stamping may not be practical in the case of larger databases with high levels of transactions. A large amount of storage space would have to be dedicated to storing the timestamps in these cases .

1. **DEADLOCK HANDLING DETCTION AND RESOLUTON**

When dealing with locks two problems can arise, the first of which being deadlock. Deadlock refers to a particular situation where two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain. Deadlock is a common problem in multiprocessing where many processes share a specific type of mutually exclusive resource. Some computers, usually those intended for the time-sharing and/or real-time markets, are often equipped with a hardware lock, or hard lock, which guarantees exclusive access to processes, forcing serialization. Deadlocks are particularly disconcerting because there is no general solution to avoid them.

A fitting analogy of the deadlock problem could be a situation like when you go to unlock your car door and your passenger pulls the handle at the exact same time, leaving the door still locked. If you have ever been in a situation where the passenger is impatient and keeps trying to open the door, it can be very frustrating. Basically you can get stuck in an endless cycle, and since both actions cannot be satisfied, deadlock occurs.