ST. XAVIER’S COLLEGE

**Maitighar, Kathmandu**

**(Affiliated to Tribhuvan University)**



**Database Management System**

**Theory Assignment #10**

**Submitted By**

Binod Paneru

B.Sc. CSIT

Year II/IV Semester

013BSCIT014

**Submitted To**

Er. Sanjay Yadav

Lecturer

Department of Computer Science

St. Xavier’s College

Maitighar, Kathmandu

**Submitted On**

30 October 2015

**Database Concurrency Model**

* 1. **Purpose of Concurrency model**

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.

**Lock Problems:**

Deadlock:

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.

Live lock:

Live lock is a special case of resource starvation. A livelock is similar to a deadlock, except that the states of the processes involved constantly change with regard to one another wile never progressing. The general definition only states that a specific process is not progressing. For example, the system keeps selecting the same transaction for rollback causing the transaction to never finish executing. Another livelock situation can come about when the system is deciding which transaction gets a lock and which waits in a conflict situation.

An illustration of livelock occurs when numerous people arrive at a four way stop, and are not quite sure who should proceed next. If no one makes a solid decision to go, and all the cars just keep creeping into the intersection afraid that someone else will possibly hit them, then a kind of live lock can happen.

* 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:

1. Lock Acquisition
2. Modification of Data
3. 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. **Limitations of CCMs**
  2. **Time-Stamp-based Protocols**

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.

Generation of Timestamp:

Timestamps can be generated in several ways. One possibility is to use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered 1, 2, 3, . . . in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time. Another way to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

Timestamp Ordering Algorithm

ü       Basic Timestamp Ordering

ü       Strict Timestamp Ordering

ü       Thomas's Write Rule

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). Notice how this differs from 2PL, where a schedule is serializable by being equivalent to some serial schedule allowed by the locking protocols. In timestamp ordering, however, the schedule is equivalent to the particular serial order corresponding to the order of the transaction timestamps. The 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. **Commit protocols**

In distributed data base and transaction systems a distributed commit protocol is required to ensure that the effects of a distributed transaction are atomic, that is, either all the effects of the transaction persist or none persist, whether or not failures occur. Several commit protocols have been proposed in the literature. These are variations of what has become a standard and known as the two-phase commit (2PC) protocol.

**Optimization of Commit Protocol**

Much of the literature focuses on improving performance in failure cases by providing a non-blocking 2PC that streamlines recovery processing at the expense of extra processing in the normal case.  We focused on improving performance in the normal case based on two assumptions: first, that networks and systems are becoming increasingly reliable, and second, that the need to support high-volume transactions requires a streamlined protocol for the normal case. Our work resulted in a number of optimizations most of which have been *incorporated in IBM and non-IBM transactional offerings*. These optimizations were presented and analyzed in terms of reliability, savings in log writes and network traffic, and reduction in resource lock time.  Our work's unique contributions include the description of some optimizations not described elsewhere in the literature and a systematic comparison of the optimizations and the environments where they cause the most benefit. Furthermore, it analyzed the feasibility and performance of several combinations of the optimizations and identifies situations where optimizations can be combined effectively. Optimizing for the non-failure case has been, also, demonstrated through this work as the correct approach towards commit optimization. These results have been published in the referred following publications and have significantly influence further work in commit protocols.

* 1. **Index Locking**

In databases an index is a data structure, part of the database, used by a database system to effectively navigate access to user data. Index data are system data distinct from user data, and consist primarily of pointers. Changes in a database (by insert, delete, or modify operations), may require indexes to be updated to maintain accurate user data accesses.[1] Index locking is a technique used to maintain index integrity. A portion of an index is locked during a database transaction when this portion is being accessed by the transaction as a result of attempt to access related user data. Additionally, special database system transactions (not user-invoked transactions) may be invoked to maintain and modify an index, as part of a system's self-maintenance activities. When a portion of an index is locked by a transaction, other transactions may be blocked from accessing this index portion (blocked from modifying, and even from reading it, depending on lock type and needed operation). Index Locking Protocol guarantees that Phantom Phenomenon won't occur. Index locking protocol states:

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 in a relation r must update all indices to r and it 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.

* 1. **Lock Granularity**

The Microsoft SQL Server Database Engine has multigranular locking that allows different types of resources to be locked by a transaction. To minimize the cost of locking, the Database Engine locks resources automatically at a level appropriate to the task. Locking at a smaller granularity, such as rows, increases concurrency but has a higher overhead because more locks must be held if many rows are locked. Locking at a larger granularity, such as tables, are expensive in terms of concurrency because locking an entire table restricts access to any part of the table by other transactions. However, it has a lower overhead because fewer locks are being maintained.

* 1. **Time Stamp ordering Multi-version Concurrency Model**
  2. **Deadlock handling detection and resolution.**

**Deadlock Handling**

! Consider the following two transactions:

T1: write (X) T2: write(Y)

write(Y) write(X)

! Schedule with deadlock

T1 T2

lock-X on X

write (X) lock-X on Y

write (X)

wait for lock-X on X

wait for lock-X on Y

Database System Concepts 3 16.40 ©Silberschatz, Korth and Sudarshan rd Edition

Deadlock Handling Deadlock Handling

! 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 Detection**

! Deadlocks can be described as a wait-for graph, which consists

of a pair G = (V,E),

! V is a set of vertices (all the transactions in the system)

! E is a set of edges; each element is an ordered pair Ti →Tj

.

! If Ti → Tj is in E, then there is a directed edge from Ti to Tj

,

implying that Ti is waiting for Tj to release a data item.

! When Ti requests a data item currently being held by Tj

, then the

edge Ti Tj is inserted in the wait-for graph. This edge is removed

only when Tj is no longer holding a data item needed by Ti

.

! The system is in a deadlock state if and only if the wait-for graph

has a cycle. Must invoke a deadlock-detection algorithm

periodically to look for cycles.

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | **Deadlock resolution clauses**  In the current economic times, potential purchasers and sellers in acquisition transactions often take a more exploratory position and are more focused on starting or continuing a strategic alliance. As a result, we see fewer acquisitions with 100% of the shares in a company being sold, and more frequently transactions in which purchasers choose to take over a limited interest in a target. This results in the formation of a joint venture with shareholders who each hold a certain percentage of shares. Common joint ventures include “50/50” joint ventures and “49/51” joint ventures. | |  |  |  | | --- | |  | |

1. **Reference**
   1. “Two Phase Locking ” <http://user.it.uu.se/~arnoldp/distrib/TwoPhase.html> “Friday 30th October 30, 2015”
   2. “Purpose of Concurrency ” <http://sce.uhcl.edu/helm/rationalunifiedprocess/process/workflow/ana_desi/co_cncry.htm> “Friday 30th October 30, 2015”
   3. “Time-Stamp-based Protocols” <https://www.classle.net/book/timestamp-based-protocol> “Friday 30th October 30, 2015”
   4. “Commit Protocols” <https://www.classle.net/book/timestamp-based-protocol> “October 30, 2015”
   5. “Index Locking” <https://en.wikipedia.org/wiki/Index_locking> “October 30, 2015”
   6. “Lock Granularity ” <https://technet.microsoft.com/en-us/library/ms189849(v=sql.105).aspx> “October 30, 2015”