ST. XAVIER’S COLLEGE

**Maitighar,Kathmandu**

**(Affiliated to Tribhuvan University)**



**Database Management System**

**Lab Assignment #12**

**Submitted By**

Pratik Gautam

B.Sc. CSIT

Year II/IV Semester

013BSCIT029

**Submitted To**

Er. Sanjay Kumar Yadav

Lecturer,

Department of Computer Science

St. Xavier’s College

Maitighar, Kathmandu

**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.

**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**

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.

**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.

**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. These are variations of what has become a standard and known as the two-phase commit (2PC) protocol

**Two-phase Commit**

The database ensures the integrity of data in a distributed transaction using the **two-phase commit mechanism**. In the **prepare phase**, the initiating node in the transaction asks the other participating nodes to promise to commit or roll back the transaction. During the **commit phase**, the initiating node asks all participating nodes to commit the transaction. If this outcome is not possible, then all nodes are asked to roll back.

All participating nodes in a distributed transaction should perform the same action: they should either all commit or all perform a rollback of the transaction.

1. **Index Locking**

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 [2].

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 [3].

**Basic Timestamp Ordering**

1. Transaction T issues a write\_item(X) operation:

* + - If read\_TS(X) > TS(T) or if write\_TS(X) > TS(T), then an younger transaction has already read the data item so abort and roll-back T and reject the operation.
    - If the condition in part (a) does not exist, then execute write\_item(X) of T and set write\_TS(X) to TS(T).

2. Transaction T issues a read\_item(X) operation:

* + - If write\_TS(X) > TS(T), then an younger transaction has already written to the data item so abort and roll-back T and reject the operation.
    - If write\_TS(X) ≤ TS(T), then execute read\_item(X) of T and set read\_TS(X) to the larger of TS(T) and the current read\_TS(X).

**Strict Timestamp Ordering**

1. Transaction T issues a write\_item(X) operation:

* + - If TS(T) > read\_TS(X), then delay T until the transaction T’ that wrote or read X has terminated (committed or aborted).

2. Transaction T issues a read\_item(X) operation:

* + - If TS (T) > write\_TS (X), then delay T until the transaction T’ that wrote or read X has terminated (committed or aborted).

**Thomas’s Write Rule**

* + If read\_TS(X) > TS (T) then abort and roll-back T and reject the operation.
  + If write\_TS(X) > TS (T), then just ignore the write operation and continue execution. This is because the most recent writes counts in case of two consecutive writes.
  + If the conditions given in 1 and 2 above do not occur, then execute write\_item(X) of T and set write\_TS(X) to TS (T).
* 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
* This 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.