**ST. XAVIER’S COLLEGE**

**MAITIGHAR, KATHMANDU**

****

**Database Management System**

**Theory Assignment #12**

**Submitted by:**

Hemanchal Joshi

013BSCIT021

**Submitted to:**

|  |  |
| --- | --- |
| **Er. Sanjay Kumar Yadav**  Lecturer  Department of Computer Science |  |

Submission Date: 30th October, 2015

**DATABASE CONCURRENCY CONTROL**

Concurrency control is the activity of coordinating concurrent accesses to a data- base in a multiuser database management system (DBMS). Concurrency control permits users to access a database in a multi- programmed fashion while preserving the illusion that each user is executing alone on a dedicated system. Process of managing simultaneous execution of transactions in a shared database, to ensure the serializability of transactions, is known as concurrency control. Simultaneous execution of transactions over a shared database can create several data integrity and consistency problems:

1. **PURPOSE OF CONCURRENCY CONTROL**

To illustrate the concept of concurrency control, consider two travellers who go to electronic kiosks at the same time to purchase a train ticket to the same destination on the same train. There's only one seat left in the coach, but without concurrency control, it's possible that both travellers will end up purchasing a ticket for that one seat. However, with concurrency control, the database wouldn't allow this to happen. Both travellers would still be able to access the train seating database, but concurrency control would preserve data accuracy and allow only one traveller to purchase the seat.

This example also illustrates the importance of addressing this issue in a multi-user database. Obviously, one could quickly run into problems with the inaccurate data that can result from several transactions occurring simultaneously and writing over each other. The following section provides strategies for implementing concurrency control.

• To enforce Isolation (through mutual exclusion) among conflicting transactions.

• To preserve database consistency through consistency preserving execution of transactions.

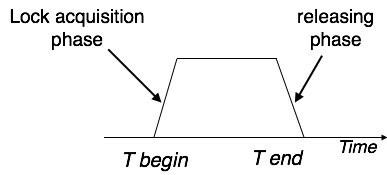
• To resolve read-write and write-write conflicts.

1. **TWO PHASE LOCKING**

Controlling access to data by locks assigned to the data. Access of a transaction to a data item (database object) locked by another transaction may be blocked (depending on lock type and access operation type) until lock release. In databases and transaction processing, two-phase locking (2PL) is a concurrency control method that guarantees serializability. It is also the name of the resulting set of database transaction schedules (histories). The protocol utilizes locks, 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:

* **Expanding phase:** locks are acquired and no locks are released.
* **Shrinking phase:** locks are released and no locks are acquired.



Two types of locks are utilized by the basic protocol: Shared and Exclusive locks. Refinements of the basic protocol may utilize more lock types. Using locks that block processes, 2PL may be subject to deadlocks that result from the mutual blocking of two or more transactions. 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.

1. **LIMITATION OF CCMS**

**Concurrency Control Schemes**

Online Transaction Processing (OLTP) database systems support the part of an application that interacts with end-users in the form of transactions. For example, OLTP may process new orders, respond to a page request, or perform a financial transaction. Each transaction is the execution of a sequence of one or more operations (e.g., SQL queries) on a shared database to perform some higher-level function.

**Many-Core Architecture**

Since many-core chips with 1000 cores do not yet exist, we performed our analysis through computer simulations using Graphite, a multi-core simulator that can scale up to 1024 cores. The simulated architecture is a tiled chip multi-processor where each tile has a small processor core, two levels of cache, and a 2D-mesh network-on-chip for communication between the cores.

**Simulation Results**

We evaluate the performance of different concurrency control schemes on the TPC-C benchmark, which is the current industry standard for evaluating the performance of OLTP systems. It consists of nine tables that simulate a warehouse-centric order processing application. For a concurrency control algorithm that requires data partitioning (i.e., H-STORE), TPC-C is partitioned based on the warehouse id of the WAREHOUSE table. Only two (Payment and New Order) out of the five transactions in TPC-C are modeled in our simulation. Since the two transactions comprise 88% of the total transactions, this is a good approximation.

**4. 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.

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.

The timestamp-ordering protocol ensures serializability among transactions in their conflicting read and write operations. This is the responsibility of the protocol system that the conflicting pair of tasks should be executed according to the timestamp values of the transactions.

* The timestamp of transaction Ti is denoted as TS(Ti).
* Read time-stamp of data-item X is denoted by R-timestamp(X).
* Write time-stamp of data-item X is denoted by W-timestamp(X).

Timestamp ordering protocol works as follows −

* **If a transaction Ti issues a read(X) operation −**
  + If TS(Ti) < W-timestamp(X)
    - Operation rejected.
  + If TS(Ti) >= W-timestamp(X)
    - Operation executed.
  + All data-item timestamps updated.
* **If a transaction Ti issues a write(X) operation −**
  + If TS(Ti) < R-timestamp(X)
    - Operation rejected.
  + If TS(Ti) < W-timestamp(X)
    - Operation rejected and Ti rolled back.
  + Otherwise, operation executed.

**5. COMMIT PROTOCOL**

Commit protocols are used to ensure atomicity across sites

* a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
* 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.

**6. INDEX LOCKING**

Index locking protocol states:

* 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.

**7. LOCK GRANULARITY**

The optimistic concurrency model is handled using row versioning. The system automatically chooses the appropriate lock granularity. A row is the smallest resource that can be locked. The support of row-level locking includes both data rows and index entries.

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

Lock granularity specifies which resource is locked by a single lock attempt. The SQL Server Database Engine can lock the following resources:

* Row
* Page
* Index key or range of index keys
* Table
* Extent
* Database itself

**8. TIME STAMP ORDERING MULTI VERSION CONCURRENCY CONTROL**

A number of different ways have been used to generate timestamp

* Use the value of the system's clock at the start of a transaction as the timestamp.
* Use a thread-safe shared counter that is incremental at the start of a transaction as the timestamp.
* A combination of the above two methods.

**9. 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.

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.

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.