ST. XAVIER’S COLLEGE

**(Affiliated to Tribhuvan University)**

Maitighar, Kathmandu



**Database Management System Assignment #**

**Submitted by:**

BIRAJ POUDYAL  
013BSCCSIT015

**Submitted to:**

|  |  |
| --- | --- |
| Er. Sanjay Kumar Yadav Lecturer, St. Xavier’s College |  |

**Date of Submission:October30, 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.

**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.

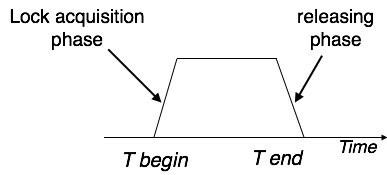
Optimistic Locking: This strategy can be used when instances of simultaneous transactions, or collisions, are expected to be infrequent. In contrast with pessimistic locking, optimistic locking doesn't try to prevent the collisions from occurring. Instead, it aims to detect these collisions and resolve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



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. **Limitations 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

1. **Time-stamp-based protocols**

The most commonly used concurrency protocol is the timestampbased 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**

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. **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 ti 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**

It deals with the cost of implementing locks depending upon the space and time. Here, space refers to data structure in DBMS for each lock and time refers to handling of lock request and release.

The cost of implementing locks depends on the size of data items. There are two types of lock granularity:

• Fine granularity

• Coarse granularity

Fine granularity refers for small item sizes and coarse granularity refers for large item Sizes.

Here, Sizes decides on the basis:

• a database record

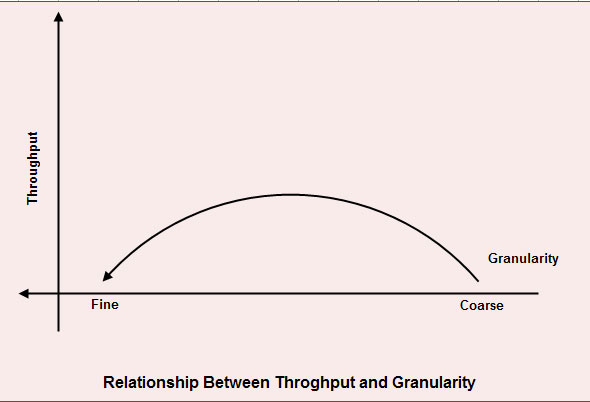
• a field value of a database record

• a disk block

• a whole file

• the whole database

If a typical transaction accesses a small number of records it is advantageous that the data item granularity is one record. If a transaction typically accesses many records of the same file it is better to have block or file granularity so that the transaction will consider all those records as one data item.

[](http://ecomputernotes.com/images/RELATIONSHIP-BETWEEN-THROUGHOUT-AND-GRAvity.jpg)

A too-fine granularity will increase the frequency of locks requests and locks releases, which therefore will add additional instructions. You must locate a balance between a too-fine and too-coarse granularity. The figure shows the relation between the throughput and the granularity of locks.

This illustration is a simple two axis chart. The vertical, or y axis, represents throughput.

The horizontal, or x axis, represents granularity going from fine to coarse as it moves out on the scale. An elongated bell curve shows the relationship of granularity on throughput. As granularity goes from fine to coarse, throughput gradually increases to a maximum level and, then slowly starts to decline. It shows that a compromise in granularity is necessary to reach maximum throughput.

1. **Time stamp ordering multiversion 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).

1. **Deadlock handling detection and resolution**

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.

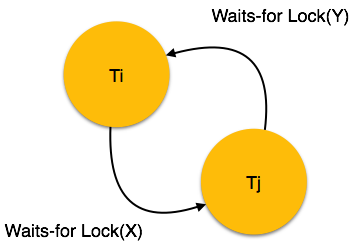
**Deadlock Avoidance**

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**.