**ST.XAVIER,S COLLEGE**

**Maitighar, Kathmandu**



DataBase Management System Assignment #14

**.**

Yub Raj Basnet

013BScCSIT048 (4th Semester)

**Submitted to**

|  |  |
| --- | --- |
| Er. Sanjya Kumar Yadav  (Lecturer, St.Xavier’s College ) |  |

Concurrency control is a database management system (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.

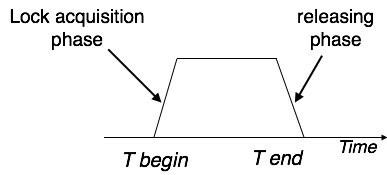
**Purpose of concurrency control**

In information technology and computer science, especially in the fields of computer programming, operating systems, multiprocessors, and databases, **concurrency control** ensures that correct results for concurrent operations are generated, while getting those results as quickly as possible.

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.

**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.

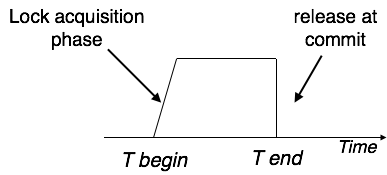


Two-phase locking has two phases, one is **growing**, where all the locks are being acquired by the transaction; and the second phase is shrinking, where the locks held by the transaction are being released.

To claim an exclusive (write) lock, a transaction must first acquire a shared (read) lock and then upgrade it to an exclusive lock.

### Strict Two-Phase Locking

The first phase of Strict-2PL is same as 2PL. After acquiring all the locks in the first phase, the transaction continues to execute normally. But in contrast to 2PL, Strict-2PL does not release a lock after using it. Strict-2PL holds all the locks until the commit point and releases all the locks at a time.



Strict-2PL does not have cascading abort as 2PL does.

**limitation of CCMs**

Concurrency Control is a type of management style where employers or supervisors constantly monitor how employees are working while the work is still in progress. This kind of management makes employees feel like slaves and lowers their morale to work, which lowers production. It also creates a sense of mistrust between the employers and the employees.

**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.

## Timestamp Ordering Protocol

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.

### Thomas' Write Rule

This rule states if TS(Ti) < W-timestamp(X), then the operation is rejected and Ti is rolled back.

Time-stamp ordering rules can be modified to make the schedule view serializable.

Instead of making Ti rolled back, the 'write' operation itself is ignored.

**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.

**index locking**

In [databases](https://en.wikipedia.org/wiki/Database) an [*index*](https://en.wikipedia.org/wiki/Index_(database)) 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](https://en.wikipedia.org/wiki/Pointer_(computer_programming)). Changes in a database (by insert, delete, or modify operations), may require indexes to be updated to maintain accurate user data accesses.[[1]](https://en.wikipedia.org/wiki/Index_locking#cite_note-Weikum01-1) **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 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](https://en.wikipedia.org/wiki/Two-phase_locking) protocol must be observed.

Specialized [concurrency control](https://en.wikipedia.org/wiki/Concurrency_control) techniques exist for accessing indexes. These techniques depend on the index type, and take advantage of its structure. They are typically much more effective than applying to indexes common concurrency control methods applied to user data. Notable and widely researched are specialized techniques for [B-trees](https://en.wikipedia.org/wiki/B-tree) ([B-Tree concurrency control](https://en.wikipedia.org/w/index.php?title=B-Tree_concurrency_control&action=edit&redlink=1)) which are regularly used as database indexes.

Index locks are used to coordinate [threads](https://en.wikipedia.org/wiki/Thread_(computer_science)) accessing indexes concurrently, and typically shorter-lived than the common transaction locks on user data. In professional literature, they are often called [*latches*](https://en.wikipedia.org/w/index.php?title=Latch_(computer_science)&action=edit&redlink=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](http://ecomputernotes.com/fundamental/what-is-a-database/advantages-and-disadvantages-of-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

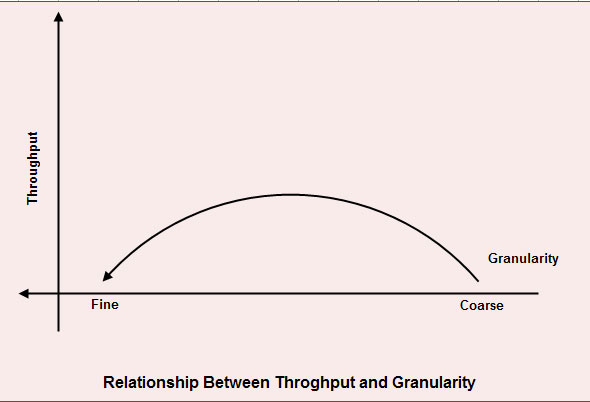
• a field value of a database record

• 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.

**time stamp ordering multi-version concurrency model**

**deadlock handling detection and resolution**

In a multi-process system, deadlock is an unwanted situation that arises in a shared resource environment, where a process indefinitely waits for a resource that is held by another process.

For example, assume a set of transactions {T0, T1, T2, ...,Tn}. T0 needs a resource X to complete its task. Resource X is held by T1, and T1 is waiting for a resource Y, which is held by T2. T2 is waiting for resource Z, which is held by T0. Thus, all the processes wait for each other to release resources. In this situation, none of the processes can finish their task. This situation is known as a deadlock.

Deadlocks are not healthy for a system. In case a system is stuck in a deadlock, the transactions involved in the deadlock are either rolled back or restarted.

## Deadlock Prevention

To prevent any deadlock situation in the system, the DBMS aggressively inspects all the operations, where transactions are about to execute. The DBMS inspects the operations and analyzes if they can create a deadlock situation. If it finds that a deadlock situation might occur, then that transaction is never allowed to be executed.

There are deadlock prevention schemes that use timestamp ordering mechanism of transactions in order to predetermine a deadlock situation.

### Wait-Die Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with a conflicting lock by another transaction, then one of the two possibilities may occur −

* If TS(Ti) < TS(Tj) − that is Ti, which is requesting a conflicting lock, is older than Tj − then Ti is allowed to wait until the data-item is available.
* If TS(Ti) > TS(tj) − that is Ti is younger than Tj − then Ti dies. Ti is restarted later with a random delay but with the same timestamp.

This scheme allows the older transaction to wait but kills the younger one.

### Wound-Wait Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur −

* If TS(Ti) < TS(Tj), then Ti forces Tj to be rolled back − that is TiwoundsTj. Tj is restarted later with a random delay but with the same timestamp.
* If TS(Ti) > TS(Tj), then Ti is forced to wait until the resource is available.

This scheme, allows the younger transaction to wait; but when an older transaction requests an item held by a younger one, the older transaction forces the younger one to abort and release the item.

In both the cases, the transaction that enters the system at a later stage is aborted.

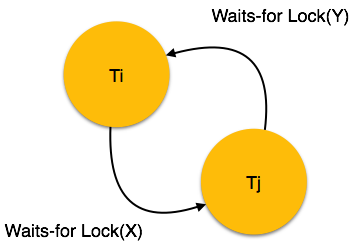
## Deadlock Avoidance

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**.