**St. Xavier’s College**

****

**Database Management System**

**Lab assignment 10 #2**

**Submitted to:**

**..............................**

**Er. Sanjay Yadav**

**Department of Computer Science**

**Submitted by:**

**Pankaj Niroula**

**BSCCSIT 4th Sem**

**Roll number:25**

Purpose of Concurrency control

**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 subesquent release of the locks that protected the data are generally grouped together and called the second phase.

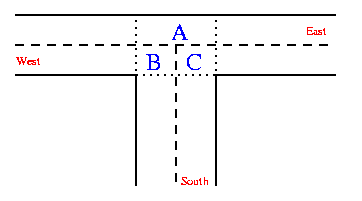
Two phase locking prevents deadlock from occuring in distributed systems by releasing all the resources it has acquired, if it is not possible to obtain all the resources required without waiting for another process to finish using a lock. This means that no process is ever in a state where it is holding 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 exectuion. This is a problem for real-time systems. Consequently, two phase locking protocols cannot be used in hard realtime applications.

In addition, more complicated problems arise when two processes compete over locks in such a way that the pattern of back-off and re-try attempts always leads them to conflict over locks in a sequence that means that neither process will ever get all the locks it needs to commit a transaction. One way to prevent this from happening is to order the locks in a global sequence, and require the processes to acquire the locks in that sequence, then both livelock and deadlock are eliminated from the algorithm.

Consider the example where processes gain access to the shared space in a traffic intersection using semaphores that represent ownership of spaces in the intersection. Each car has two possible direction choices as it enters the intersection and the algorithms executed by cars entering the intersection from different directions are shown in the following table. Refer to the picture for and image of the intersection and the allocation of compass points and semphores to areas.

  
A sample intersection

A set of access algorithms that uses incremental locking and allows deadlock.

|  |  |  |
| --- | --- | --- |
| Car from South | Car from West | Car from East |
| Go\_East:  Wait(C)  Turn\_Right  Signal(C)  Go\_West:  Wait(C)  Wait(A)  Turn\_Left  Signal(A)  Signal(C) | Go\_East:  Wait(B)  Wait(C)  Drive\_Straight  Signal(C)  Signal(B)  Go\_South:  Wait(B)  Turn\_Right  Signal(B) | Go\_West:  Wait(A)  Drive\_Straight  Signal(A)  Go\_South:  Wait(A)  Wait(B)  Turn\_Left  Signal(B)  Signal(A) |

A set of access algorithms that uses incremental locking and prevents deadlock.

|  |  |  |
| --- | --- | --- |
| Car from South | Car from West | Car from East |
| Go\_East:  Wait(C)  Turn\_Right  Signal(C)  Go\_West:  Wait(A)  Wait(C)  Turn\_Left  Signal(A)  Signal(C) | Go\_East:  Wait(B)  Wait(C)  Drive\_Straight  Signal(B)  Signal(C)  Go\_South:  Wait(B)  Turn\_Right  Signal(B) | Go\_West:  Wait(A)  Drive\_Straight  Signal(A)  Go\_South:  Wait(A)  Wait(B)  Turn\_Left  Signal(B)  Signal(A) |

Limitations of CCMs

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**

       Basic Timestamp Ordering

       Strict Timestamp Ordering

       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. **Read\_TS(*X*):**The **read timestamp**of item *X;*this is the largest timestamp among all the timestamps of transactions that have successfully read item *X*—that is, read\_TS(*X*) = TS(T), where T is the *youngest*transaction that has read *X*successfully.

2. **Write\_TS**(***X***)**:**The **write timestamp**of item *X;*this is the largest of all the timestamps of transactions that have successfully written item *X*—that is, write\_TS(*X*) = TS(T), where T is the *youngest*transaction that has written *X*successfully.

**Basic Timestamp Ordering**

Whenever some transaction T tries to issue a read\_item(*X*) or a write\_item(*X*) operation, the **basic TO**algorithm compares the timestamp of T with read\_TS(*X*) and write\_TS(*X*) to ensure that the timestamp order of transaction execution is not violated. If this order is violated, then transaction T is aborted and resubmitted to the system as a new transaction with a *new timestamp.*If T is aborted and rolled back, any transaction T1 that may have used a value written by T must also be rolled back. Similarly, any transaction T2 that may have used a value written by T1 must also be rolled back, and so on. This effect is known as **cascading rollback**and is one of the problems associated with basic TO, since the schedules produced are not recoverable. An *additional protocol*must be enforced to ensure that the schedules are recoverable, cascadeless, or strict. We first describe the basic TO algorithm here. The concurrency control algorithm must check whether conflicting operations violate the timestamp ordering in the following two cases:

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

a. If read\_TS(*X*) > TS(T) or if write\_TS(*X*) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with a timestamp greater than TS(T)—and hence *after*T in the timestamp ordering—has already read or written the value of item *X*before T had a chance to write *X*, thus violating the timestamp ordering.

b. If the condition in part (a) does not occur, then execute the write\_item(*X*) operation of T and set write\_TS(*X*) to TS(T).

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

a. If write\_TS(*X*) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with timestamp greater than TS(T)—and hence *after*T in the timestamp ordering—has already written the value of item *X*before T had a chance to read *X*.

b. If write\_TS(*X*) <= TS(T), then execute the read\_item(*X*) operation of T and set read\_TS(*X*) to the *larger*of TS(T) and the current read\_TS(*X*).

Hence, whenever the basic TO algorithm detects two *conflicting operations*that occur in the incorrect order, it rejects the later of the two operations by aborting the transaction that issued it. The schedules produced by basic TO are hence guaranteed to be conflict serializable, like the 2PL protocol. However, some schedules are possible under each protocol that is not allowed under the other. Hence, neither protocol allows *all possible*serializable schedules. As mentioned earlier, deadlock does not occur with timestamp ordering. However, cyclic restart (and hence starvation) may occur if a transaction is continually aborted and restarted.

**Strict Timestamp Ordering**

A variation of basic TO called **strict TO**ensures that the schedules are both **strict**(for easy recoverability) and (conflict) serializable. In this variation, a transaction T that issues a read\_item(*X*) or write\_item(*X*) such that TS(T) > write\_TS(*X*) has its read or write operation *delayed*until the transaction T that *wrote*the value of*X*(hence TS(T) = write\_TS(*X*)) has committed or aborted. To implement this algorithm, it is necessary to simulate the locking of an item *X*that has been written by transaction T until T is either committed or aborted. This algorithm does not cause deadlock, since T waits for T only if TS(T) > TS(T).

**Thomas's Write Rule**

A modification of the basic TO algorithm, known as **Thomas’s write rule,**does not enforce conflict serializability; but it rejects fewer write operations, by modifying the checks for the write\_item(*X*) operation as follows:

1. If read\_TS(*X*) > TS(T), then abort and roll back T and reject the operation.

2. If write\_TS(*X*) > TS(T), then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already written the value of *X*. Hence, we must ignore the write\_item(*X*) operation of T because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case (1).

3. If neither the condition in part (1) nor the condition in part (2) occurs, then execute the write\_item(*X*) operation of T and set write\_TS(*X*) to TS(T).

**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.

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 refered following publications and have significantly influence further work in commit protocols.

**Index Locking**

An index access method can choose whether it supports concurrent updates of the index by multiple processes. If the method's pg\_am.amconcurrent flag is true, then the core PostgreSQL system obtainsAccessShareLock on the index during an index scan, and RowExclusiveLock when updating the index. Since these lock types do not conflict, the access method is responsible for handling any fine-grained locking it may need. An exclusive lock on the index as a whole will be taken only during index creation, destruction, or REINDEX. When amconcurrent is false, PostgreSQL still obtains AccessShareLock during index scans, but it obtains AccessExclusiveLock during any update. This ensures that updaters have sole use of the index. Note that this implicitly assumes that index scans are read-only; an access method that might modify the index during a scan will still have to do its own locking to handle the case of concurrent scans.

Recall that a backend's own locks never conflict; therefore, even a non-concurrent index type must be prepared to handle the case where a backend is inserting or deleting entries in an index that it is itself scanning. (This is of course necessary to support an UPDATE that uses the index to find the rows to be updated.)

Building an index type that supports concurrent updates usually requires extensive and subtle analysis of the required behavior. For the b-tree and hash index types, you can read about the design decisions involved in src/backend/access/nbtree/README and src/backend/access/hash/README.

Aside from the index's own internal consistency requirements, concurrent updates create issues about consistency between the parent table (the *heap*) and the index. Because PostgreSQL separates accesses and updates of the heap from those of the index, there are windows in which the index may be inconsistent with the heap. We handle this problem with the following rules:

* A new heap entry is made before making its index entries. (Therefore a concurrent index scan is likely to fail to see the heap entry. This is okay because the index reader would be uninterested in an uncommitted row anyway. But see [Section 48.5](http://www.postgresql.org/docs/8.1/static/index-unique-checks.html).)
* When a heap entry is to be deleted (by VACUUM), all its index entries must be removed first.
* For concurrent index types, an index scan must maintain a pin on the index page holding the item last returned by amgettuple, and ambulkdelete cannot delete entries from pages that are pinned by other backends. The need for this rule is explained below.

If an index is concurrent then it is possible for an index reader to see an index entry just before it is removed by VACUUM, and then to arrive at the corresponding heap entry after that was removed by VACUUM. (With a nonconcurrent index, this is not possible because of the conflicting index-level locks that will be taken out.) This creates no serious problems if that item number is still unused when the reader reaches it, since an empty item slot will be ignored by heap\_fetch(). But what if a third backend has already re-used the item slot for something else? When using an MVCC-compliant snapshot, there is no problem because the new occupant of the slot is certain to be too new to pass the snapshot test. However, with a non-MVCC-compliant snapshot (such as SnapshotNow), it would be possible to accept and return a row that does not in fact match the scan keys. We could defend against this scenario by requiring the scan keys to be rechecked against the heap row in all cases, but that is too expensive. Instead, we use a pin on an index page as a proxy to indicate that the reader may still be "in flight" from the index entry to the matching heap entry. Making ambulkdelete block on such a pin ensures that VACUUM cannot delete the heap entry before the reader is done with it. This solution costs little in run time, and adds blocking overhead only in the rare cases where there actually is a conflict.

This solution requires that index scans be "synchronous": we have to fetch each heap tuple immediately after scanning the corresponding index entry. This is expensive for a number of reasons. An"asynchronous" scan in which we collect many TIDs from the index, and only visit the heap tuples sometime later, requires much less index locking overhead and may allow a more efficient heap access pattern. Per the above analysis, we must use the synchronous approach for non-MVCC-compliant snapshots, but an asynchronous scan is workable for a query using an MVCC snapshot.

In an amgetmulti index scan, the access method need not guarantee to keep an index pin on any of the returned tuples. (It would be impractical to pin more than the last one anyway.) Therefore it is only safe to use such scans with MVCC-compliant snapshots.

**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

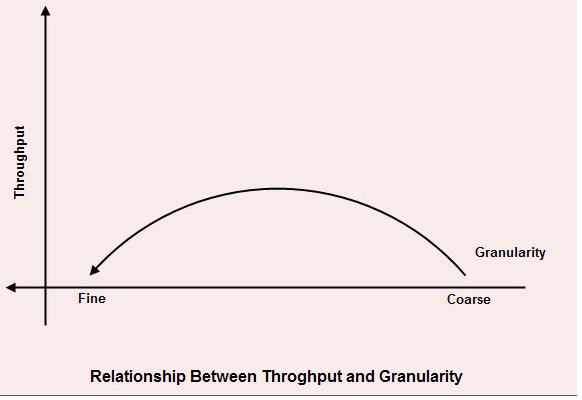
• the whole database

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.

**RELATIONSHIP-BETWEEN-THROUGHOUT-AND-GRAVITY**

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 control**

As we have seen above, a problem with 2PL is that it can lead to deadlocks. Reed's multiversion timestamp ordering scheme solves this problem by ordering transactions and aborting transactions that access data out of order. It also increases the concurrency in the system by never making an operation block (though it does abort transactions.)

The basic idea in this scheme is to assign transactions timestamps when they are started, which are used to order these transactions. If two transactions access data items in an order that is inconsistent with their time stamps, then one of them is aborted.

**Deadlock Handling & Resolution**