ST. XAVIER’S COLLEGE

**(Affiliated to Tribhuvan University)**

**Maitighar, Kathmandu**

****

**Database Management System**

**Theory Assignment #12**

**SUBMITTED BY:**

Anlisha Maharjan

013BSCCSIT006

**SUBMITTED TO:**

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

Submission Date: 30th October 2015

**DATABASE CONCURRENCY CONTROL**

**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. Concurrency control, when applied to a DBMS, is meant to coordinate simultaneous transactions while preserving data integrity.

The purposes of concurrency problem are:

1. The lost update problem: A second transaction writes a second value of a data-item (datum) on top of a first value written by a first concurrent transaction, and the first value is lost to other transactions running concurrently which need, by their precedence, to read the first value. The transactions that have read the wrong value end with incorrect results.
2. The dirty read problem: Transactions read a value written by a transaction that has been later aborted. This value disappears from the database upon abort, and should not have been read by any transaction ("dirty read"). The reading transactions end with incorrect results.
3. The incorrect summary problem: While one transaction takes a summary over the values of all the instances of a repeated data-item, a second transaction updates some instances of that data-item. The resulting summary does not reflect a correct result for any (usually needed for correctness) precedence order between the two transactions (if one is executed before the other), but rather some random result, depending on the timing of the updates, and whether certain update results have been included in the summary or not.

**TWO PHASE LOCKING**

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.

**LIMITATIONS OF CCMS**

**TIME-STAMP-BASED PROTOCOLS**

A timestamp-based concurrency control algorithm is a non-lock concurrency control method. It is used in some databases to safely handle transactions, using timestamps.

Assumptions

* Every timestamp value is unique and accurately represents an instant in time.
* No two timestamps can be the same.
* A higher-valued timestamp occurs later in time than a lower-valued timestamp.

Generating a Timestamp

* 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.

**COMMIT PROTOCOLS**

In [transaction processing](https://en.wikipedia.org/wiki/Transaction_processing), [databases](https://en.wikipedia.org/wiki/Database), and [computer networking](https://en.wikipedia.org/wiki/Computer_networking), the two-phase commit protocol (2PC) is a type of [atomic commitment protocol](https://en.wikipedia.org/wiki/Atomic_commit) (ACP). It is a [distributed algorithm](https://en.wikipedia.org/wiki/Distributed_algorithm) that coordinates all the processes that participate in a [distributed atomic transaction](https://en.wikipedia.org/wiki/Distributed_transaction) on whether to [commit](https://en.wikipedia.org/wiki/Commit_(data_management)) or abort (roll back) the transaction (it is a specialized type of [consensus](https://en.wikipedia.org/wiki/Consensus_(computer_science)) protocol). The protocol achieves its goal even in many cases of temporary system failure (involving process, network node, communication, etc. failures), and is thus widely utilized. However, it is not resilient to all possible failure configurations, and in rare cases, user (e.g., a system's administrator) intervention is needed to remedy an outcome. To accommodate recovery from failure (automatic in most cases) the protocol's participants use [logging](https://en.wikipedia.org/wiki/Server_log) of the protocol's states. Log records, which are typically slow to generate but survive failures, are used by the protocol's [recovery procedures](https://en.wikipedia.org/wiki/Recovery_procedure). Many protocol variants exist that primarily differ in logging strategies and recovery mechanisms. Though usually intended to be used infrequently, recovery procedures compose a substantial portion of the protocol, due to many possible failure scenarios to be considered and supported by the protocol.

In a "normal execution" of any single [distributed transaction](https://en.wikipedia.org/wiki/Distributed_transaction), i.e., when no failure occurs, which is typically the most frequent situation; the protocol consists of two phases:

1. The commit-request phase (or voting phase), in which a coordinator process attempts to prepare all the transaction's participating processes (named participants, cohorts, or workers) to take the necessary steps for either committing or aborting the transaction and to vote, either "Yes": commit (if the transaction participant's local portion execution has ended properly), or "No": abort (if a problem has been detected with the local portion), and
2. The commit phase, in which, based on voting of the cohorts, the coordinator decides whether to commit (only if all have voted "Yes") or abort the transaction (otherwise), and notifies the result to all the cohorts. The cohorts then follow with the needed actions (commit or abort) with their local transactional resources (also called recoverable resources; e.g., database data) and their respective portions in the transaction's other output (if applicable).

**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. 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.

**LOCK GRANULARITY**

The granularity of locks in a database refers to how much of the data is locked at one time. In theory, a database server can lock as much as the entire database or as little as one column of data. Such extremes affect the concurrency (number of users that can access the data) and locking overhead (amount of work to process lock requests) in the server. Adaptive Server supports locking at the table, page, and row level.

By locking at higher levels of granularity, the amount of work required to obtain and manage locks is reduced. If a query needs to read or update many rows in a table:

* It can acquire just one table-level lock
* It can acquire a lock for each page that contained one of the required rows
* It can acquire a lock on each row

**TIME STAMP ORDERING MULTI-VERSION CONCURRENCY MODEL**

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 DETECTION AND RESOLUTION**

Most current operating systems cannot prevent a deadlock from occurring.[10] When a deadlock occurs, different operating systems respond to them in different non-standard manners. Major approaches are as follows.

**Ignoring deadlock**

In this approach, it is assumed that a deadlock will never occur. This is also an application of the Ostrich algorithm. This approach was initially used by MINIX and UNIX. This is used when the time intervals between occurrences of deadlocks are large and the data loss incurred each time is tolerable.

**Detection**

Under deadlock detection, deadlocks are allowed to occur. Then the state of the system is examined to detect that a deadlock has occurred and subsequently it is corrected. An algorithm is employed that tracks resource allocation and process states, it rolls back and restarts one or more of the processes in order to remove the detected deadlock. Detecting a deadlock that has already occurred is easily possible since the resources that each process has locked and/or currently requested are known to the resource scheduler of the operating system. Deadlock detection techniques include, but are not limited to, model checking. This approach constructs a finite state-model on which it performs a progress analysis and finds all possible terminal sets in the model.

**After a deadlock is detected, it can be corrected by using one of the following methods:**

Process termination: one or more processes involved in the deadlock may be aborted. We can choose to abort all processes involved in the deadlock. This ensures that deadlock is resolved with certainty and speed. But the expense is high as partial computations will be lost. Or, we can choose to abort one process at a time until the deadlock is resolved. This approach has high overheads because after each abort an algorithm must determine whether the system is still in deadlock. Several factors must be considered while choosing a candidate for termination, such as priority and age of the process.

Resource preemption: resources allocated to various processes may be successively preempted and allocated to other processes until the deadlock is broken.

**Prevention**

**Removing the mutual exclusion** condition means that no process will have exclusive access to a resource. This proves impossible for resources that cannot be spooled. But even with spooled resources, deadlock could still occur. Algorithms that avoid mutual exclusion are called non-blocking synchronization algorithms.

The **hold and wait** or resource holding conditions may be removed by requiring processes to request all the resources they will need before starting up (or before embarking upon a particular set of operations). This advance knowledge is frequently difficult to satisfy and, in any case, is an inefficient use of resources. Another way is to require processes to request resources only when it has none. Thus, first they must release all their currently held resources before requesting all the resources they will need from scratch. This too is often impractical. It is so because resources may be allocated and remain unused for long periods. Also, a process requiring a popular resource may have to wait indefinitely, as such a resource may always be allocated to some process, resulting in resource starvation.[13] (These algorithms, such as serializing tokens, are known as the all-or-none algorithms.)

The **no preemption condition** may also be difficult or impossible to avoid as a process has to be able to have a resource for a certain amount of time, or the processing outcome may be inconsistent or thrashing may occur. However, inability to enforce preemption may interfere with a priority algorithm. Preemption of a "locked out" resource generally implies a rollback, and is to be avoided, since it is very costly in overhead. Algorithms that allow preemption include lock-free and wait-free algorithms and optimistic concurrency control.

The final condition is the **circular wait condition**. Approaches that avoid circular waits include disabling interrupts during critical sections and using a hierarchy to determine a partial ordering of resources.