ST. XAVIER’S COLLEGE

**(Affiliated to Tribhuvan University)**

**Maitighar, Kathmandu**

****

**Database Management System**

**TheoryAssignment #12**

**SUBMITTED BY:**

**Apil Neupane**

**013BSCCSIT008**

**SUBMITTED TO:**

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

Submission Date: 30th October 2015

**DATABASE CONCURRENCY MODEL**

1. **PURPOSE OF CONCURRENCY MODEL**

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

Process of managing simultaneous operations on the database without having them interfere with one another.

• Prevents interference when two or more users are accessing database simultaneously and at least one is updating data.

• Although two transactions may be correct in themselves, interleaving of operations may produce

an incorrect result.

**PURPOSE:**

Several problems can occur when concurrent transactions execute in an uncontrolled manner.

1) **The Lost Update Problem**

This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.

Successfully completed update is overridden by another user.

Example:

• T1 withdraws £10 from an account with balx, initially £100.

• T2 deposits £100 into same account.

• Serially, final balance would be £190.

**2) The Temporary Update (or Dirty Read) Problem**

This problem occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed by another transaction before it is changed back to its original value.

Occurs when one transaction can see intermediate results of another transaction before it has committed.

Example:

• T4 updates balx to £200 but it aborts, so balx should be back at original value of £100.

• T3 has read new value of balx (£200) and uses value as basis of £10 reduction, giving a new balance of £190, instead of £90.

**3)The Incorrect Summary Problem**

If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

Occurs when transaction reads several values but second transaction updates some of them during execution of first.

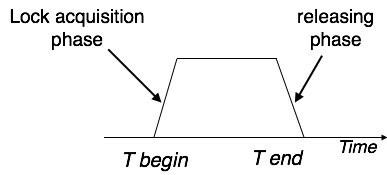
Example:

• T6 is totaling balances of account x (£100), account y (£50), and account z (£25).

• Meantime, T5 has transferred £10 from balx to balz, so T6 now has wrong result (£10 too high).

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**

### 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.

1. **TIME-STAMP-BASED PROTOCOLS**

* Each transaction is issued a timestamp when it enters the system. If an old transaction *Ti* has time-stamp TS(*Ti*), a new transaction *Tj* is assigned time-stamp TS(*Tj*) such that TS(*Ti*) <TS(*Tj*).
* The protocol manages concurrent execution such that the time-stamps determine the serializability order.
* In order to assure such behavior, the protocol maintains for each data *Q* two timestamp values:
  + **W-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **write**(*Q*) successfully.
  + **R-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **read**(*Q*) successfully.
* The timestamp ordering protocol ensures that any conflicting  **read** and **write** operations are executed in timestamp order.
* Suppose a transaction Ti issues a **read**(*Q*)

1. If TS(*Ti*) ≤ **W**-timestamp(*Q*), then *Ti* needs to read a value of *Q* that was already overwritten. Hence, the **read** operation is rejected, and *Ti*  is rolled back.

2. If TS(*Ti*)≥ **W**-timestamp(*Q*), then the **read** operation is executed, and R-timestamp(*Q*) is set to the maximum of R-timestamp(*Q*) and TS(*Ti*).

* Suppose that transaction *Ti* issues **write**(*Q*).
* If TS(*Ti*) < R-timestamp(*Q*), then the value of *Q* that *Ti* is producing was needed previously, and the system assumed that that value would never be produced. Hence, the **write** operation is rejected, and *Ti* is rolled back.
* If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*. Hence, this **write** operation is rejected, and *Ti* is rolled back.
* Otherwise, the  **write** operation is executed, and W-timestamp(*Q*) is set to TS(*Ti*).

1. **COMMIT PROTOCOLS**

In transaction processing, databases, and [computer networking](https://en.wikipedia.org/wiki/Computer_networking), the **two-phase commit protocol** (**2PC**) is a type of atomic commitment protocol (ACP). It is a distributed algorithm that coordinates all the processes that participate in a distributed atomic transaction on whether to *commit* or *abort* (*roll back*) the transaction (it is a specialized type of consensus protocol). The protocol achieves its goal even in many cases of temporary system failure (involving either 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 of the protocol's states. Log records, which are typically slow to generate but survive failures, are used by the protocol's recovery procedures. 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, 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).
3. **INDEX LOCKING**

* 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.

1. **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

Less overall work is required to use a table-level lock, but large-scale locks can degrade performance, by making other users wait until locks are released. Decreasing the lock size makes more of the data accessible to other users. However, finer granularity locks can also degrade performance, since more work is necessary to maintain and coordinate the increased number of locks. To achieve optimum performance, a locking scheme must balance the needs of concurrency and overhead.

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

**Multiversion concurrency control** (**MCC** or **MVCC**), is a concurrency control method commonly used by database management systems to provide concurrent access to the database and in programming languages to implement transactional memory.

If someone is reading from a database at the same time as someone else is writing to it, it is possible that the reader will see a half-written or inconsistent piece of data. There are several ways of solving this problem, known as concurrency control methods. The simplest way is to make all readers wait until the writer is done, which is known as a lock. This can be very slow, so MVCC takes a different approach: each user connected to the database sees a *snapshot* of the database at a particular instant in time. Any changes made by a writer will not be seen by other users of the database until the changes have been completed (or, in database terms: until the transaction has been committed.)

When an MVCC database needs to update an item of data, it will not overwrite the old data with new data, but instead mark the old data as obsolete and add the newer version elsewhere. Thus there are multiple versions stored, but only one is the latest. This allows readers to access the data that was there when they began reading, even if it was modified or deleted part way through by someone else. It also allows the database to avoid the overhead of filling in holes in memory or disk structures but requires (generally) the system to periodically sweep through and delete the old, obsolete data objects. For a document-oriented database it also allows the system to optimize documents by writing entire documents onto contiguous sections of disk—when updated, the entire document can be re-written rather than bits and pieces cut out or maintained in a linked, non-contiguous database structure.

MVCC provides *point in time* consistent views. Read transactions under MVCC typically use a timestamp or transaction ID to determine what state of the DB to read, and read these versions of the data. Read and write transactions are thus isolated from each other without any need for locking. Writes create a newer version, while concurrent reads access the older version.

1. **DEADLOCK HANDLING DETECTION AND RESOLUTION**

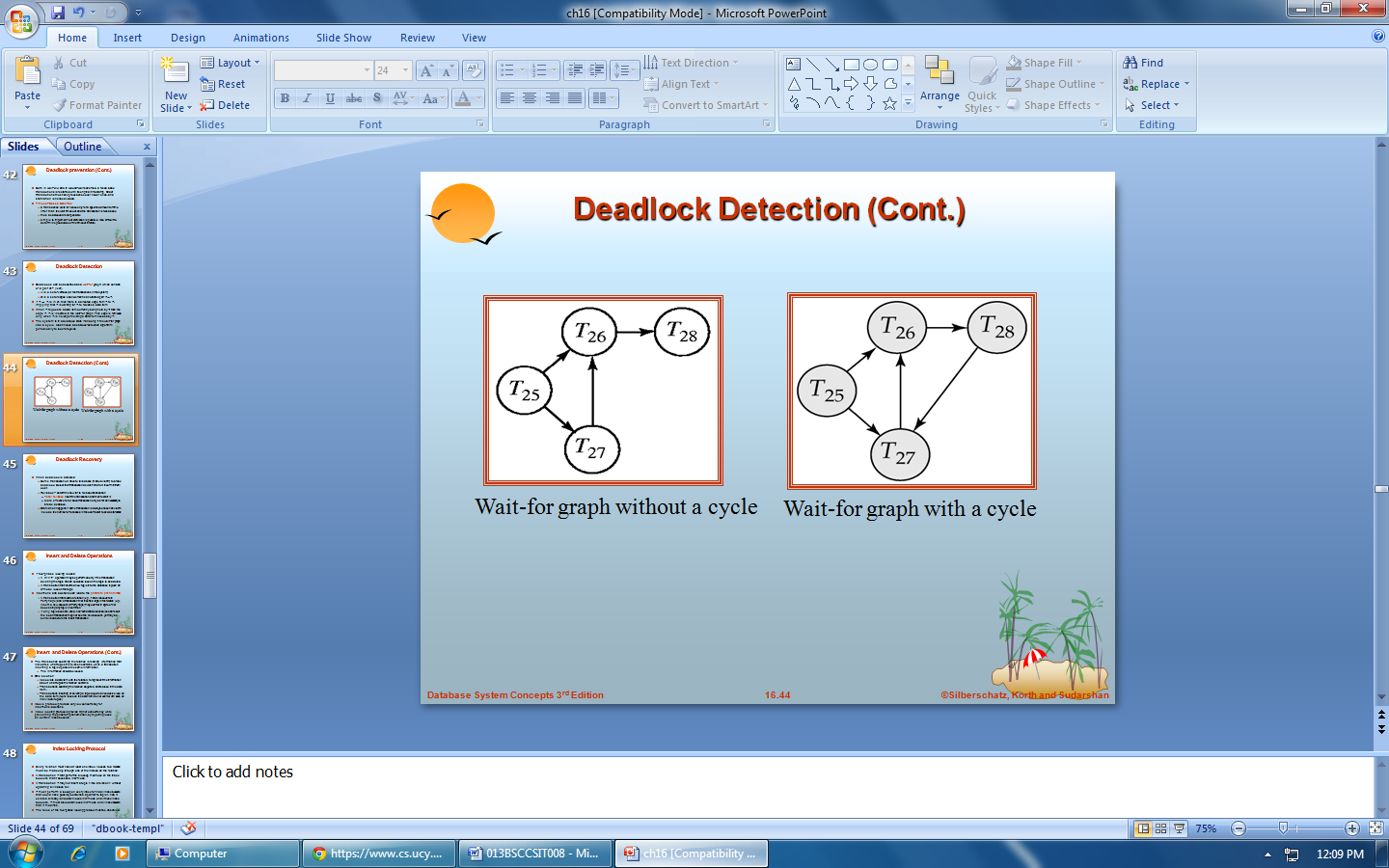
* System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
* *Deadlock prevention* protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
  + Require that each transaction locks all its data items before it begins execution (predeclaration).
  + Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

**Deadlock Prevention Strategies**

* **wait-die** scheme — non-preemptive
  + older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
  + a transaction may die several times before acquiring needed data item
* **wound-wait** scheme — preemptive
  + older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  + may be fewer rollbacks than *wait-die* scheme.
* Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.
* Timeout-Based Schemes :
  + a transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  + thus deadlocks are not possible
  + simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

**Deadlock Detection:**

* Deadlocks can be described as a *wait-for graph*, which consists of a pair *G* = (*V*,*E*),
  + *V* is a set of vertices (all the transactions in the system)
  + *E* is a set of edges; each element is an ordered pair *Ti* →*Tj*.
* If *Ti →* *Tj*is in *E*, then there is a directed edge from *Ti* to *Tj*, implying that *Ti* is waiting for *Tj* to release a data item.
* When *Ti* requests a data item currently being held by *Tj*, then the edge *Ti* *Tj* is inserted in the wait-for graph. This edge is removed only when *Tj* is no longer holding a data item needed by *Ti*.
* The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.



**DEADLOCK RECOVERY:**

* When deadlock is detected :
  + Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
  + Rollback -- determine how far to roll back transaction
    - Total rollback: Abort the transaction and then restart it.
    - More effective to roll back transaction only as far as necessary to break deadlock.
  + Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation