**Chapter 3: concurrency control**

## Concurrency Control (CC) Techniques

Concurrency Control is the 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.

In a multiprogramming environment where multiple transactions can be executed simultaneously, it is highly important to control the concurrency of transactions. We have concurrency control protocols to ensure atomicity, isolation, and Serializability of concurrent transactions.

**Why Concurrency Control is needed?**

* + The lost update problem
  + The temporary update (dirty read) problem
  + The incorrect summary problem
  + The unrepeated read

Concurrency control protocols are (lock, time stamp, and optimistic) can be broadly divided into two categories:

* Pessimistic concurrency control
* Optimistic concurrency control

**Pessimistic concurrency control**

A system of locks prevents users from modifying data in a way that affects other users. After a user performs an action that causes a lock to be applied, other users cannot perform actions that would conflict with the lock until the owner releases it. This is called pessimistic control because it is mainly used in environments where there is high contention for data, where the cost of protecting data with locks is less than the cost of rolling back transactions if concurrency conflicts occur. Locking and Time stamping are conservative approaches: delay transactions in case they conflict with other transactions.

**Optimistic concurrency control**

In optimistic concurrency control, users do not lock data when they read it. When a user updates data, the system checks to see if another user changed the data after it was read. If another user updated the data, an error is raised. Typically, the user receiving the error rolls back the transaction and starts over. This is called optimistic because it is mainly used in environments where there is low contention for data, and where the cost of occasionally rolling back a transaction is lower than the cost of locking data when read. The optimistic approach allows us to proceed and check conflicts at the end. Optimistic methods assume conflict is rare and only check for conflicts at commit.

**The pros and cons of the pessimistic and optimistic concurrency control mechanisms**

Both pessimistic and optimistic concurrency control mechanisms provide different performance, e.g., the different average transaction completion rates or throughput, depending on transaction type’s mix, computing level of parallelism, and other events. Both have pros and cons as shown below:

For pessimistic concurrency control, the strength is:

* Guarantee that all transactions can be executed correctly.
* Data is properly consistent by either rolling back to the previous state (Abort operation) or new content (Commit operation) when the transaction conflict is cleared.
* Database is relatively stable and reliable.

Its weakness is:

* Transactions are slow due to the delay by locking or time-stamping event.
* Runtime is longer. Transaction latency increases significantly.
* Throughput or the amount of work (e.g. read/write, update, rollback operations, etc.)is reduced.

For optimistic concurrency control, the strength is:

* Transactions are executed more efficiently.
* Data content is relatively safe.
* Throughput is much higher.

Its weakness is:

* There is a risk of data interference among concurrent transactions since it transactions conflict may occur during execution. In this case, data is no longer correct.
* Database may have some hidden errors with inconsistent data; even conflict check is performed at the end of transactions.
* Transactions may be in deadlock that causes the system to hang.

## Locking Techniques for Concurrency Control

A **LOCK** is a mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution. Locks are one way of enforcing concurrency control policies. Transaction uses locks to deny access to other transactions and so prevent incorrect updates.

Lock prevents another transaction from modifying item or even reading it, in the case of a write lock.Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

**Types of a Lock**

**Binary Locks:** Have only two states: locked (1) or unlocked (0)

* **Lock (X):** If a transaction T1 applies Lock on data item X, then X is locked and it is not available to any other transaction.
* **Unlock (X):** T1 Unlocks X. X is available to other transactions.

**Shared/Exclusive Locks**

***Shared lock****:* A Read operation does not change the value of a data item. Hence a data item can be read by two different transactions simultaneously under share lock mode. So only to read a data item T1 will do: *Share lock (X), then Read (X), and finally Unlock (X).*shared lock is also called read lock.

***Exclusive lock:***A write operation changes the value of the data item. Hence two write operations from two different transactions or a write from T1 and a read from T2 are not allowed. A data item can be modified only under Exclusive lock. To modify a data item T1 will do: *Exclusive lock (X), then Write (X) and finally Unlock (X).*Exclusive lock is also called write lock.

* The relationship between Shared and Exclusive Lock can be represented by the following table which is known as **Lock Matrix**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Lock to be Granted** | **Locks already Existing** | | | * A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions. * Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item. * If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted |
|  | **S** | **X** |
| **S** | True | False |
| **X** | False | False |

When these locks are applied, then a transaction must behave in a special way. This special behavior of a transaction is referred to as *well-formed*.

***Well-formed:*** A transaction is well- formed if it does not lock a locked data item and it does not try to unlock an unlocked data item.

***Locking - Basic Rules***

* If transaction has shared lock on item, can read but not update item.
* If transaction has exclusive lock on item, can both read and update item.
* Reads cannot conflict, so more than one transaction can hold shared locks simultaneously on same item.
* Exclusive lock gives transaction exclusive access to that item.
* Some systems allow transaction to upgrade a shared lock to an exclusive lock, or vice-versa.

**Examples:** T1 and T2 are two transactions. They are executed under locking as follows. T1 locks A in exclusive mode. When T2 want s to lock A, it finds it locked by T1 so T2 waits for Unlock on A by T1. When A is released then T2 locks A and begins execution.

Example of a transaction performing locking:

|  |  |
| --- | --- |
| **T1:**  lock-S(A);  read (A);  unlock(A);  lock-S(B);  read (B);  unlock(B);  display(A+B) | Locking as shown here is not sufficient to guarantee serializability — if A and B get updated in-between the read of A and B, the displayed sum would be wrong. |

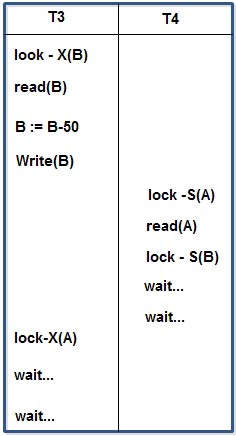
Suppose a lock on a data item is applied, the data item is processed and it is unlocked immediately after reading/writing is completed as follows.

**Locking methods: problems**

**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. A deadlock that may result when two (or more) transactions are each waiting for locks held by the other to be released.

|  |  |
| --- | --- |
| **T1** | **T2** |
| lock-X(B);  read (B);  B =B-50;  WRITE(B); |  |
|  | lock-S(A);  read (A);  lock-S(B); |
| lock-X(A); |  |

Neither T1 nor T2 can make progress — executing lock-S(B) causes T2 to wait for T1 to release its lock on B, while executing lock-X(A) causes T1 to wait for T2 to release its lock on A.

Such a situation is called a deadlock. To handle a deadlock one of T1 or T2 must be rolled back and its locks released

**Deadlock - possible solutions**

Only one way to break deadlock: abort one or more of the transactions in the deadlock.Deadlock should be transparent to user, so DBMS should restart transaction(s).

Three genseral techniques for handling deadlock:

* Deadlock prevention.
* Timeout
* Deadlock detection and recovery.

**Timeout**

The deadlock detection could be done using the technique of TIMEOUT. Every transaction will be given a time to wait in case of deadlock. If a transaction waits for the predefined period of time in idle mode, the DBMS will assume that deadlock occurred and it will abort and restart the transaction.

## Concurrency Control Based on Timestamp Ordering

Timestamp: a unique identifier created by DBMS that indicates relative starting time of a transaction.This protocol uses either system time or logical counter as a timestamp. Therefore, can be generated by:

* using system clock at time transaction started, or
* Incrementing a logical counter every time a new transaction starts.

Time-stamping 🡪 A concurrency control protocol that orders transactions in such a way that older transactions, transactions with smaller time stamps, get priority in the event of conflict.

* Transactions ordered globally base do their timestamp so that older transactions, transactions with earlier timestamps, get priority in the event of conflict.
* Conflict is resolved by rolling back and restarting transaction.
* Since there is no need to use lock there will be ***No Deadlock***.

In timestamp ordering, the schedule is equivalent to the particular serial order that corresponds to the order of the transaction timestamps. To implement this scheme, every transaction will be given a timestamp which is a unique identifier of a transaction. If Ti came to processing prior to Tj then TS of Tj will be larger than TS of Ti. Again, each data item will have a timestamp for Read and Write.

* ***WTS(A)*** which denotes the largest timestamp of any transaction that successfully executed ***Write(A)***
* ***RTS(A)*** which denotes the largest timestamp of any transaction that successfully executed ***Read(A)***

These timestamps are updated whenever a new Read(A) or Write(A) instruction is executed.

Read/write proceeds only if last update on that data item was carried out by an older transaction. Otherwise, transaction requesting read/write is restarted and given a new 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.

The timestamp ordering protocol ensures that any conflicting read and write operations are executed in the timestamp order.

* a transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time, the transaction is rolled back and restarted
* Thus, deadlocks are not possible
* simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

Cascading Rollback

Whenever some transaction T tries to issue a Read\_Item(X) or a Write\_Item(X) operation, the basic timestamp ordering algorithm compares the timestamp of T with the read timestamp and the write timestamp of X to ensure that the timestamp order of execution of the transactions is not violated. If the timestamp order is violated by the operation, then transaction T will violate the equivalent serial schedule, so T is aborted. Then T is resubmitted to the system as a new transaction with new timestamp. If T is aborted and rolled back, any transaction Ti that may have used a value written by T must also be rolled back. Similarly, any transaction Tj that may have used a value written by Ti must also be rolled back, and so on. This effect is known as ***cascading rollback***.

## Validation (Optimistic) Concurrency Control Technique

***Optimistic Technique***

* Locking and assigning and checking timestamp values may be unnecessary for some transactions
* Assumes that conflict is rare.
* When transaction reaches the level of executing commit, a check is performed to determine whether conflict has occurred. If there is a conflict, transaction is rolled back and restarted.
* Based on assumption that conflict is rare and more efficient to let transactions proceed without delays to ensure serializability.
* At commit, check is made to determine whether conflict has occurred.
* If there is a conflict, transaction must be rolled back and restarted.
* Potentially allows greater concurrency than traditional protocols.
* Three phases:
  1. Read
  2. Validation
  3. Write

1. **Optimistic Techniques - Read Phase**

* Extends from start until immediately before commit. Also, called **Execution Phase** − A transaction fetches data items to memory and performs operations upon them.
* Transaction reads values from database and stores them in local variables. Updates are applied to a local copy of the data.

1. **Optimistic Techniques - Validation Phase**

* Follows the read phase.
* For read-only transaction, checks that data read are still current values. If no interference, transaction is committed, else aborted and restarted.
* For update transaction, checks transaction leaves database in a consistent state, with serializability maintained.

1. **Optimistic Techniques - Write Phase**

* Follows successful validation phase for update transactions.
* Updates made to local copy are applied to the database. Also called **Commit Phase** − A transaction writes back modified data item in memory to the disk

|  |  |  |
| --- | --- | --- |
| **Schedule** | | |
| **T1** | **T2** | **Concurrency control Manger** |
| Lock-X(B)  Read (B,b)  b= b-50  Write(B,b)  Unlock(B) |  | Gant-X(B,T1) |
|  | Lock-S(A)  Read (A,a)  Unlock (A)  b= b-50 | Grant-S(A,T2) |
|  | Lock-S(B)  Read (B,b)  Unlock(B)  display(a+b) | Grant-S(B,T2) |
| Lock-X(A)  Read (A,a)  a= a+50  Write(A,a)  Unlock(A) |  | Gant-X(A,T1) |