**Concurrency Control**

When more than one transactions are running simultaneously there are chances of a conflict to occur which can leave database to an inconsistent state. To handle these conflicts we need concurrency control in DBMS, which allows transactions to run simultaneously but handles them in such a way so that the integrity of data remains intact.

Let’s take an example to understand what I’m talking here.

## Conflict Example

You and your brother have a joint bank account, from which you both can withdraw money. Now let’s say you both go to different branches of the same bank at the same time and try to withdraw 5000, your joint account has only 6000 balance. Now if we don’t have concurrency control in place you both can get 5000 at the same time but once both the transactions finish the account balance would be -4000 which is not possible and leaves the database in inconsistent state.  
We need something that controls the transactions in such a way that allows the transaction to run concurrently but maintaining the consistency of data to avoid such issues.

## Solution of Conflicts: Locks

A lock is kind of a mechanism that ensures that the integrity of data is maintained. There are two types of a lock that can be placed while accessing the data so that the concurrent transaction can not alter the data while we are processing it.

1. Shared Lock(S)
2. Exclusive Lock(X)
3. **Shared Lock(S)**: Shared lock is placed when we are reading the data, multiple shared locks can be placed on the data but when a shared lock is placed no exclusive lock can be placed.

|  |
| --- |
| T1 |
| S(A)  R(A)  U(A) |

For example, when two transactions are reading account balance, let them read by placing shared lock but at the same time if another transaction wants to update the account balance by placing Exclusive lock, do not allow it until reading is finished.

1. **Exclusive Lock(X)**: Exclusive lock is placed when we want to read and write the data. This lock allows both the read and write operation, Once this lock is placed on the data no other lock (shared or Exclusive) can be placed on the data until Exclusive lock is released.

|  |
| --- |
| T1 |
| X(A)  R(A)  W(A)  U(A) |

For example, when a transaction wants to update the account balance, let it do by placing X lock on it but if a second transaction wants to read the data(S lock) don’t allow it, if another transaction wants to write the data(X lock) don’t allow that either.

So based on this we can create a table like this:

### Lock Compatibility Matrix

Request

|  |  |  |
| --- | --- | --- |
|  | **S** | **X** |
| **S** | True | False |
| **X** | False | False |

**R\_W**

|  |  |
| --- | --- |
| T1 | T2 |
| R(A)---1000 | R(A)  W(A)----50 |

**W-R**

|  |  |
| --- | --- |
| T1 | T2 |
| W(A) | R(A) |

**How to read this matrix?:**  
There are two rows, first row says that when S lock is placed, another S lock can be acquired so it is marked true but no Exclusive locks can be acquired so marked False.  
In second row, When X lock is acquired neither S nor X lock can be acquired so both marked false.

Problems in Shared Exclusive Locking

1. May not sufficient to produce only serializability schedule.

T1-🡪T2

OR  
T2--🡪T1

|  |  |
| --- | --- |
| T1 | T2 |
| X(A) G  R(A)  W(A)  U(A)  X(B) G  R(B) W(B)  U(B) | S(A) wait-🡪G  R(A)  U(A) |

1. May not free from irrecoverable.

|  |  |
| --- | --- |
| T1 | T2 |
| X(A) G  R(A)  W(A)  U(A)  .  .  .  .  .  .  \*  Fail | S(A) wait--🡪G  R(A)  .  .  .  Commit; |

1. May not free from deadlock.

|  |  |
| --- | --- |
| T1 | T2 |
| X(A) G  X(B) wait | X(B) G  X(A) wait |

1. May not free from starvation.

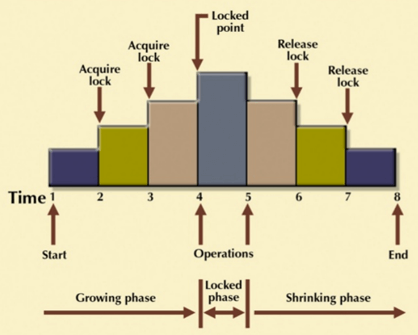
|  |  |  |  |
| --- | --- | --- | --- |
| T1 | T2 | T3 | T4 |
| X(A) wait—G  -  -  -  -  -  -  -  --  - | S(A) G  .  .  .  U(A) | S(A) G  .  .  .  .  U(A) | S(A) G  .  .  .  U(A) |

## Two Phase Locking Protocol

**Two Phase Locking Protocol** also known as 2PL protocol is a method of concurrency control in DBMS that ensures serializability by applying a lock to the transaction data which blocks other transactions to access the same data simultaneously. Two Phase Locking protocol helps to eliminate the concurrency problem in DBMS.

This locking protocol divides the execution phase of a transaction into three different parts.

* In the first phase, when the transaction begins to execute, it requires permission for the locks it needs.
* The second part is where the transaction obtains all the locks. When a transaction releases its first lock, the third phase starts.
* In this third phase, the transaction cannot demand any new locks. Instead, it only releases the acquired locks.



The Two-Phase Locking protocol allows each transaction to make a lock or unlock request in two steps:

* **Growing Phase**: In this phase transaction may obtain locks but may not release any locks.
* **Shrinking Phase**: In this phase, a transaction may release locks but not obtain any new lock

It is true that the 2PL protocol offers serializability. However, it does not ensure that deadlocks do not happen.

Example:

|  |
| --- |
| T1 |
| X(A)G  S(B)G  R(A) W(A)  R(B)  S(C)G  R(C)  S(D)G  R(D)  U(A)  U(B) U(C) U(D) |

Example:

If we want to achieve serializability. We will go with 2PL.

|  |  |
| --- | --- |
| T1 | T2 |
| X(A) G  R(A)  W(A)  S(B) G  R(B)  @  U(A)  R(B)  U(B)  . | S(A)wait🡪G  R(A) |

T1-🡪T2

Example:

|  |  |
| --- | --- |
| T1 | T2 |
| S(A)  X(B)  U(A)  U(B) | S(A)  X(D)  U(A)  U(D) |

## Drawbacks In 2PL OR Problems in 2PL:

**Advantage:**

Always ensure serializability.

**Disadvantage:**

1. May not free from ir-recoverability.
2. May free from deadlock.
3. May free from starvation.
4. May free from cascading Rollback.

Types of 2PL:

1. **Strict 2PL:**

It should satisfy the basic 2PL and all exclusive locks should hold until commit/ abort.

Cascade problem

|  |  |  |
| --- | --- | --- |
| T1 | T2 | T3 |
| X(A)  R(A)  W(A)  .  .  .  U(A)  .  .  \*  fail | S(A)  R(A) | S(A)  R(A) |

Cascade-less

|  |  |  |
| --- | --- | --- |
| T1 | T2 | T3 |
| X(A)  R(A)  W(A)  .  .  .  Commit;  U(A) | S(A)  R(A) | S(A)  R(A) |

Ir-recoverable problem:

|  |  |
| --- | --- |
| T1 | T2 |
| X(A)  R(A)  W(A)  U(A)  .  .  .  .  .  .  \*  fail | S(A)  R(A)  .  .  .  Commit |

Recoverable schedule:

|  |  |
| --- | --- |
| T1 | T2 |
| X(A)  R(A)  W(A)  .  .  .  .  .  .  \*  fail | S(A)  R(A)  .  .  .  Commit |

|  |
| --- |
| Advantage of Strict Recoverable:   * Cascade-less * Strict recoverable |

1. **Rigorous 2PL:**

It should satisfy the basic 2PL and all shared and exclusive locks should hold until commit/ abort.

|  |
| --- |
| Advantage of Rigorous Recoverable:   * Cascade-less * Strict recoverable   Disadvantages of Rigorous Recoverable:   * Deadlock * starvation |

Basic 2PL

Strict 2PL

Rigorous 2PL

1. **Conservative 2PL:**

|  |  |
| --- | --- |
| T1 | T2 |
| X(A)  X(B) | X(B)  X(A) |

Timestamp Ordering Protocol:

What is timestamp?

Unique values assign to every transaction.

What is the significance?

It tells the order (When they enter in the system.).

Example:

|  |  |  |
| --- | --- | --- |
| Time | 10:00AM | 10:10AM |
| Transactions | T1 | T2 |
| Timestamp (in increasing order) | 100 | 200 |
|  | Old | Young |

**Notation:**

TS(Ti): TS(T2)=200

**Read\_TS(RTS):** Last transactions number which performed read successfully.

Example:

|  |  |  |
| --- | --- | --- |
| T1 | T2 | T3 |
| 10 | 20 | 30 |
| R(A) | R(A) | R(A) |

RTS(A)=30(Last time A is read by T3)

**Write\_TS(WTS):** Last transactions number which performed write successfully.

Example:

|  |  |  |
| --- | --- | --- |
| T1 | T2 | T3 |
| 10 | 20 | 30 |
| W(A) | W(A) | W(A) |

WTS(A)=20(Last time A is written by T2)

In timestamp all transaction executes in sequences.

Case1: RW--🡪Allowed

|  |  |
| --- | --- |
| T1 | T2 |
| 100 | 200 |
| R(A) |  |
|  | W(A) |

Case2: WR---🡪Allowed

|  |  |
| --- | --- |
| T1 | T2 |
| 100 | 200 |
| W(A) |  |
|  | R(A) |

Case3: WW--🡪Allowed

|  |  |
| --- | --- |
| T1 | T2 |
| 100 | 200 |
| W(A) |  |
|  | W(A) |

Rules:

Timestamp ordering protocol works as follows:

* **If a transaction Ti issues a read(A) operation −**
  + If W-timestamp(A) > TS(Ti)
    - Rollback Ti
  + Otherwise, execute R(A) operation
    - Set RTS(A) = max {RTS(A), TS(Ti)}
* **If a transaction Ti issues a write(A) operation −**
  + If R-timestamp(A) > TS(Ti)
    - Rollback Ti
  + If W-timestamp(A) > TS(Ti)
    - Rollback Ti
  + Otherwise, execute W(A) operation
    - Set WTS(A) = TS(Ti)

In timestamp all transaction executes in sequences.

Case1: RW--🡪Not Allowed

|  |  |
| --- | --- |
| T1 | T2 |
| 100 | 200 |
|  | R(A) |
| W(A) |  |

Case2: WR---🡪Not Allowed

|  |  |
| --- | --- |
| T1 | T2 |
| 100 | 200 |
|  | W(A) |
| R(A) |  |

Case3: WW--🡪Not Allowed

|  |  |
| --- | --- |
| T1 | T2 |
| 100 | 200 |
|  | W(A) |
| W(A) |  |

Exercise:

|  |  |  |
| --- | --- | --- |
| 100 | 200 | 300 |
| T1 | T2 | T3 |
| R(A)  W(C)  R(C) | R(B)  W(B) | R(B)  W(A) |

|  |  |  |  |
| --- | --- | --- | --- |
|  | A | B | C |
| RTS | 0 | 0 | 0 |
| WTS | 0 | 0 | 0 |

T1---------🡪 R(A)

According to Rule1: