Time-Stamp based Concurrency Control

- In a time-stamp based concurrency control protocol, we associate an unique time-stamp to each transaction denoted as $TS(t_i)$ where t_i is the transaction.
  - Either the value of the system clock is taken as the time-stamp when the transaction entered the system.
  - Or a logical counter is used that is incremented after the entry of every transaction into the system. The transaction is assigned with the counter's present value when the transaction is entered into the system.
  - The timestamps of the transactions determine the serializability order.
- To implement this scheme, two times-tamp values are associated with each data item also. Let's take a data item (A) and see:-
  - W-timestamp(A) is the largest of the times-tamps of any transaction that successfully has written to (A).
  - R-timestamp(A) is the largest timestamp of any transactions that successfully reads the (A).
- A transaction with a smaller time-stamp value is considered to be an older transaction whereas a transaction with a larger time-stamp is considered to be a younger one.
- A conflict is said to occur when an older transaction tries to read a value that is written by a younger transaction.
  If $TS(T_i) < W\text{-timestamp}(A)$ then the the read operation of T_i is rejected and T_i is rolled back. Otherwise;
  If $TS(T_i) \geq W\text{-timestamp}(A)$ then the the read operation is accepted.
- A conflict also occurs when an older transaction tries to modify a value already read or written by a younger transaction.
  If $TS(T_i) < R\text{-timestamp}(A)$ or If $TS(T_i) < W\text{-timestamp}(A)$ here T_i has a Write operation in conflict.
- These time-stamp ordering protocols ensure conflict serializability because conflicting operations are processed in time-stamp order.
- This protocol ensures freedom from deadlock, since no transaction ever waits as the conflict is resolved by rolling back the conflicting transaction(s).
- The only demerit of this protocol is that if there is a long transaction conflicting with a sequence of short transactions then the long transaction may be repeatedly restarted causing a starvation.

Multiversion schemes

The Multiversion schemes get their name from keeping multiple versions of the data in the database. In this scheme each write operation creates a new version of data (A). When any transaction issues a read on (A) then the concurrency control manager selects one version of (A) and allows it to be read.

- The concurrency control manager during selection of the version of (A) must ensure that the serializability is maintained and
- The process of determining the proper version of (A) should be easy and quick.

We can apply the multiversion scheme to both time-stamp and 2PL schemes. However **Multiversion timestamp ordering** is most common.

The transactions have timestamps associated with them as $TS(t_i)$. Further for a data item (A) there is a sequence of versions of (A) like (A_1), (A_2), (A_3) etc. Each (A_i) contains three data fields:-

1. Content (value of the version of A)
2. W-timestamp(A_i) is the timestamp of the transaction which created this version.
3. R-timestamp(A_i) is the largest timestamp from the transactions which successfully read the version.

If transaction (t_{11}) creates a version of (A) by a write operation let's take (A_3) then the content is the value written and W-timestamp(A_3) and R-timestamp(A_3) is initialized to the timestamp of (t_{11}) i.e. $TS(t_{11})$. Whenever any transaction (t_i) reads (A_3) then the $TS(t_i)$ will be checked with R-timestamp(A_3) and if the current timestamp is less (R-timestamp(A_3)<$TS(t_i)$) then the R-timestamp will be updated to $TS(t_i)$.

To ensure serializability:

1. If any transaction wants to read (A) then the value of (A_i) whose W-timestamp is largest is provided for read[1]. If the timestamp of the transaction is $TS(t_i)$ then:
   W-timestamp(A_i)$\leq TS(t_i)$ has to be true too.
2. If any transaction wants to write (A) then[2]
   1. if ($TS(t_i)$<R-timestamp(A_i)) then
      Rollback(t_i)
   2. if ($TS(t_i)$=W-timestamp(A_i)) then
      the value is overwritten
   3. if ($TS(t_i)$>W-timestamp(A_i)) then
      a new version of (A) is created with the timestamp of (t_i) that is $TS(t_i)$.

1 The rule-1 says that only the recent version of (A) must be read.

2 The second rule forces a transaction to abort if a newer transaction has read the value. Otherwise the other conditions are evaluated to compare the value of the write timestamp of the data item (A_i) and appropriate decision is taken. If other two conditions also not satisfied then Rollback(t_i) is performed.

Multiversion Locking

This attempts to combine the benefits of multiversion concurrency control along with the advantages of two-phase locking. This protocol differentiates between Read-only and Update transactions.

For Update transactions there is rigorous 2phase Locking (all locks are held till completion of transaction) and so they are serializable. Each version of the data item is has a single timestamp. The timestamp in this case is not a real clock-based one but rather a counter (ts-counter). When the Update transaction(t_i) wants to read it gets a shared lock and reads the recent most value. When it want to execute an update on the database it is allowed an exclusive lock. It creates a new version with timestamp (∞) which is bigger then any possible timestamp. When it completes all the actions it carrys out the commit process.

- it sets the timestamp of each version it created to $TS(t_i)+1$
- the ts-counter is incremented

  Only one Update transaction is permitted at a time.

Read-only transactions are assigned a timestamp by reading the current ts-counter. They follow the multiversion timestamp ordering protocol for read.

Read-only transactions will read the value before update if they are executed before an update is performed or will see an updated value if they are executed after an update.