Time-stamp protocol – ensures that conflicting read and write operations occur in timestamp order.

If a read (Q) is issued by Ti:

1) If TS(Ti) < W-timestamp(Q) then Ti needs to read a value of Q that was already overwritten. The read operation is rejected and Ti is rolled back.
2) If TS(Ii) >= 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).

If a write(Q) is issued:

1) If TS(Ti) < R-timestamp(Q) then the value of Q that Ti is producing was needed previously, and the system assumed that the value would never be produced. The write is rejected and Ti is rolled back.
2) If TS(Ti) < W-timestamp(Q) then Ti is attempting to write an obsolete value of Q. The write is rejected and Ti is rolled back.
3) Otherwise the write is accepted and W-timestamp(Q) is set to TS(Ti).

Time-stamp protocol ensures conflict serializability – because conflicting operations are processed in timestamp order. It also ensures freedom from deadlock since no transaction ever waits. There is the possibility of starvation if a sequence of short transactions continue to cause repeated restarting of a long transaction.

The protocol can generate schedules that are not recoverable. It can be extended to include recoverability by:

1) (Recoverable and cascadeless) Forcing all the writes together at the end of the transaction. The writes are atomic in that while the writes are in progress, no transaction is permitted to access any of the data items being written.

2) (Recoverable and cascadeless) Using a limited form of locking where the reads of uncommitted items are postponed until the transaction that updated the item commits.
3) (Recoverable) Tracking uncommitted writes and allowing Ti to commit only after the commit of any transaction that write a value the Ti read.

Thomas write rule:

This involves the case that obsolete writes can be ignored under certain circumstances. Read rules remain unchanged, but writes are slightly different.

If a write(Q) is issued:

1) If TS(Ti) < R-timestamp(Q) then the value of Q that Ti is producing was needed previously, and the system assumed that the value would never be produced. The write is rejected and Ti is rolled back.
2) If TS(Ti) < W-timestamp(Q) then Ti is attempting to write an obsolete value of Q. The write is ignored.
3) Otherwise the write is accepted and W-timestamp(Q) is set to TS(Ti).

The Thomas write rule makes use of view serializability by deleting obsolete write operations from the transactions that issue them.

Multiversion schemes

Multiversion schemes keep old versions of a data item to increase concurrency.

Mutiversion timestamp ordering

Each successful write results in the creation of a new version of the data item written
Use timestamps to label versions

When a R(Q) operation is issued, select the appropriate version of Q based on the timestamp of the transaction and return that value. Reads never have to wait as the appropriate version is available immediately.

Each data item Q has a sequence of versions <Q1, Q2, Q3…..Qn>

Each version contains the value, W-timestamp(Qk) – TS of the transaction that wrote Qk, and R-timestamp(Qk) – largest TS of a transaction to successfully read version Qk.

When Ti creates a new version Qk of Q, Qk's W&R-timestamps are initialized to TS(Ti).

R-timestamp of Qk is updated whenever a transaction Tj reads Qk and TS(Tj) > R-timestamp(Qk).

Suppose Ti issues a R(Q) or a W(Q) operation. Let Qk denote the version of Q whose W-timestamp is the largest W-timestamp <= TS(Ti).

1) If Ti issues a R(Q), then the value returned is the content of of version Qk. Reads always succeed.
2) If Ti issues a W(Q)
    a. If TS(Ti) < R-timestamp(Qk), them transaction Ti is rolled back. Some other transaction Tj (as serialization is defined by the timestamp values) should read Tj's write and has already read a version created by a transaction older than Ti.
    b. If TS(Ti) =W-timestamp(Qk), the contents of Qk are overwritten; Qk was written before by Ti.
    c. If (TS(Ti) > W-timestamp(Qk) a new version of Q is created.
    d. If (TS(Ti) < W-timestamp(Qk), write is ignored (Thomas write rule)

Validation protocols

When the majority of the transactions are read-only, rate of conflict is low. In this situation, you may not want to incur the cost of a concurrency control scheme. Instead, we look at a scheme that does monitoring rather than locking.

Each transaction is assumed to execute in three phases whether it is read-only or an update transaction. The phases are:

1) Read phase – all values to be read are read into variable local to Ti. All writes are performed on these local variables rather than the actual database.
2) Validation phase – Ti performs a validation test to determine whether it can copy to the database the results of the writes without causing a violation of serializability.
3) If the validation succeeds, the system applies the changes – otherwise the transaction is rolled back.

Each transaction must go through the three phases, but the three phases can be interleaved by concurrently executing transactions. To perform the validation phase we need a couple of pieces of data. Three timestamps are associated with the transaction.

1) Start (Ti) – the time when Ti started its execution.
2) Validation (Ti) – the time when Ti finished its read phase and started its validation phase.
3) Finish (Ti) – the time when Ti finished its write phase.

The serializability order is determined by the timestamp Validation – i.e. TS(Ti) = Validation(Ti). The validation test for Tj is as follows: For all transactions Ti where TS(Ti)<TS(Tj) one of the two following must hold.

1) Finish(Ti) < Start (Tj). Since Ti completes before Tj starts, the serializability order is maintained.
2) The set of data items written by Ti does not intersect with the set of data items read by Tj; and Ti completes it write phase before Tj starts its validation phase. (Start(Tj) < Finish(Ti) < Validation (Tj)). This condition ensures the writes of Ti and Tj do not overlap. Since the writes of Ti do not affect the read of Tj and since Tj cannot affect the read of Ti – the serializability order is maintained.

This is an example of an optimistic concurrency control.