Multiversion Database Systems

and

Multiversion Concurrency Control

Monoversion database systems

- Update_in_place (disks) – it violates accounting practice that has been observed for centuries. It destroys data (the old version), and if data has value, then update_in_place destroys value.
- Degree of concurrency – problem of the so called audit transaction.

S: T1: r(x) T2: r(x, y) T2: w(x, y) T2: c T1: r(y) T1: c

The schedule is not serializable in monoversion's environment.

Compatibility matrix:

operation \ operation	***read(x)***	***write(x)***
read(x)	*4*	–
write(x)	–	–

Multiversion data item

There have been several proposals for systems in which data items are never altered; rather, a data item is considered to have a **history**.

A data item is not updated but evolves as information is added. The process consists in creating a new value and appending it to the data item history as the data item's current value. Such a data item is called ***a multiversion data item.***

A multiversion data item can be defined as follows (a sequence of versions):

$x = (n, (v^0, v^1, ..., v^k))$

where v^i denotes an i-th version of the data item x (a value)

A multiversion data item can be also defined as follows (a history):

$x = (n, (v^0[to, t1]), (v^1[t1, t2]), ..., (v^k[tk, *]))$

where v^0, v^1, ..., v^k is a sequence of values of consecutive versions, and to, t1, t2, tk is an increasing time sequence.

This is interpreted as follows: between time to and t1, the data item x had value v^0. At time t1 it was updated to value v^1, and at time tk the most recent update was made.

Each transaction is assigned a time at which it ran, that is, a time at which all its reads and writes are interpreted.

Reading a data item, transaction T, assigned to run at time t3, gets the value of the data item at that time.

Example:

S: T1: r(x) T2: r(x, y) T2: w(x, y) T2: c T1: r(y) T1: c

Assume that at the beginning data items x and y have the following histories:

x = (x0); y = (y0)

The schedule is not serializable in a monoversion database system; however, it is correct if both data items x and y are multiversioned.

The equivalent serial schedule is the following:

S_{serial}: T1: r(x, y) T1: c T2: r(x, y) T2: w(x, y) T2: c

Multiversion Schedule

A multiversion schedule mvs of a set of transactions τ is a triple:

$$mvs(\tau) = (\overline{T}_{mvs}(\tau), \prec_{mvs}, h),$$

where:

1. $\overline{T_{mvs}}(\tau) = \coprod_{T_i \in \tau} \overline{T_i} \cup T_0 \cup T_f$;
2. $\prec_{mvs} \supseteq \coprod_{T_i \in \tau} \prec_i$;
3. $h : \overline{T}mvs \rightarrow \overline{T}mvs$ is a function that maps each read operation $\{r_i(x) \in \overline{T}mvs(\tau)\}$ into a write operation $\{w_j(x) \in \overline{T}mvs(\tau)\}$, in such a way that for each read operation $r_i(x) \in \overline{T}mvs(\tau)$, if $h(r_i(x)) = w_j(x)$, then $w_j(x) \prec_{mvs} r_i(x)$.

According to the monoversion serializability criterion, a given concurrent monoversion schedule is correct if it is equivalent to a serial monoversion schedule.

Is it possible to extend this definition for multiversion schedules?

Example:

Assume that DB consists of a single data item x. Assume initially that x=(x0=1000).
Given 3 transactions increasing the value of x by 1000 – Ti = { x= x + 1000}, i=1, 2, 3.

Consider the following serial schedule of T1, T2, and T3.

S_{serial}: T1: r(x0) T1: w(x0+1000) T1: c
T2: r(x0) T2: w(x0+1000) T2: c
T3: r(x0) T3: w(x0+1000) T3: c

The final DB state is the following:

x = (x0, x1, x2, x3)

where x0=1000; x1=2000; x2=2000; x3=2000

The multiversion serial schedule is incorrect!

Standard serial multiversion schedule

A serial multiversion schedule mvs(τ) is ***standard*** if each read operation $T_i : r(x)$ accesses the version of a data item x created by the last write operation $T_j : w(x)$ preceding $T_i : r(x)$. In other words, a serial multiversion schedule is standard if there is no such write operation $T_k : w(x)$ that the following condition is fulfilled:

$$h(T_i : r(x)) \prec_{mvs} T_k : w(x) \prec_{mvs} T_i : r(x)$$

Multiversion serializability criterion

A multiversion schedule *mvs*(τ) of a set of transactions τ is ***multiversion serializable (correct)*** if it is equivalent to any standard serial multiversion schedule of the set of transactions τ.

Multiversion Two-Phase Locking Algorithm

(two-version data model)

The concept of multiversion 2PL is broader than the concept of monoversion 2PL.

Each transaction initiated in the system and each version of a data item is *certified* or *uncertified*. When a transaction begins, it is uncertified. Similarly, each new version prepared in the transaction's workspace is uncertified.

A certify operation is introduced and a new lock mode – the certify lock is introduced. Cerify locks are mutually incompatible.

Any read operation r(x) concerns the last certified version of a data item x or any uncertified version of this data item. Any write operation w(x) prepares a new version of x in the workspace of the transaction. At the end of transaction execution the transaction and new versions of data items prepared are being certified. The certification procedure consists of certify-locking of all data items that the transaction accessed to write. The Ti certification is

completed when all certify locks are set and the following conditions are satisfied:

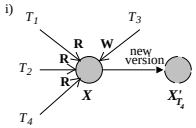
- At the moment of ti's certification the versions of all data items read by Ti are certified;
- For each data item x that Ti wrote, all transactions that read certified versions of x are also certified.

Inorder to satisfy the second condition a certify token is allocated to each data item x to forbid reading certified versions of x other than the last one.

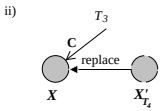
Compatibility matrix

Current lock / Requested lock	***R***	***W***	***C***
R	*4*	*4*	–
W	*4*	–	–
C	–	–	–

i)



ii)



Read*(x, tid) begin*
*B: if (LOCK(x, tid) = **C**)*
then begin
< insert into queue(x) and wait (until lock manager wakes up the transaction)>;
go to B;
end;
else begin
*LOCK(x, tid) ← **R**;*
< read x >;
end;
end Read;

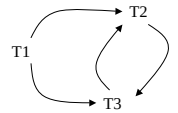
Write*(x, tid) begin*
*B: if (LOCK(x, tid) = **0** or LOCK(x, tid) = **R**)*
then begin
*LOCK(X, tid) ← **W**;*
< create new version x' >;
end;
else begin
< insert into queue(X) and wait (until lock manager wakes up the transaction)>;
go to B;
end;
end W_lock;

Certify*(x, tid) begin*
B: if (number_of_read_locks_on_x = 0) then begin
*LOCK(X, tid) ← **C**;*
< replace old version x with x' >;
end;
else begin
< wait (until lock manager wakes up the transaction)>;
go to B;
end;
end Certify;

Example:

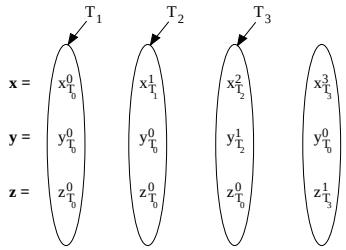
S: T1: r(x) T1: r(y) T1: r(z) T2: r(y) T1: w(z) T1:c
T3: r(x) T3: w(y) T2: w(y) T2:c T3: r(z) T3:w(z) T3: c

The schedule is incorrect in monoversion database system due to cycle



However, the algorithm correctly serialize the operations and the schedule is multiversion serializable.

Multiversion Timestamp Ordering Method



Each multiversion data item $x=<x_0, x_1, ..., x_n>$ is represented by its history:

$H(x) = \{(Read\_TS(x_0), Write\_TS(x_0), ..., (Read\_TS(x_n), Write\_TS(x_n)\}$,

The pair $\{(Read\_TS(x_i), Write\_TS(x_i)\}$ is called a ***history of a version x_i***. A history consists of two timestamps:

- ***Read_TS(xi)*** – is assigned the largest timestamp of a completed transaction that read this version; until the first transaction that has read the version completes, the timestamp Read_TS(xi) has the value Write_TS(xi).
- ***Write_TS(xi)*** - is assigned the value of the timestamp of the transaction that created (wrote) this version.

The history $H(x)$={(1, 5), (8, 10), (13, 18), (19, 19)} means that the data item x has four versions (x0, x1, x2, x3), created at times 1, 8, 13, and 19.

Reed Algorithm

Read(T_i, x) *begin*
  < read x_k , such that ***Write_TS***(x_k) *= max {*(x_j) *:*
    Write_TS$(x_j) \le$ ***TS***(T_i)*} >;*
  *if (***Read_TS***$(x_k) <$ ***TS***(T_i)*) then*
    Read_TS$(x_k) \leftarrow$ ***TS***(T_i)*;*
end ***Read****;*

Write(T_i, x) *begin*
  if (exists x_k , such that
    Write_TS$(x_k) \le$***TS***$(T_i) \le$ ***Read_TS***(x_k)*)*
  then
    < abort T_i and restart it with a new timestamp >;
  else begin
    < write x_k >;
    Write_TS(x_k), ***Read_TS***$(x_k) \leftarrow$ ***TS***(T_i)*;*
  end;
end ***Write****;*

Example:

Consider the execution of four operations performed on a data item x: two read operations by transactions T1 and T2, and two write operations by transactions T3 and T4. Let TS(T1)=3, TS(T2)=11, TS(T3)=15 and TS(T4)=20.

Assume that the history of x is the following:
$H(x)$={(1, 5), (8, 10), (13, 18), (19, 19)}