6 Multiversion Concurrency Control

6.1 Motivation: Versions and serializability
6.2 MVCC using timestamps
6.3 Two Version MV2PL
6.4 MVCC for read only TA
6.5 Read consistency MVCC – no read locks
6.6 Snapshot isolation

based on slides by Weikum / Vossen: Transactional Information Systems; P. Bernstein

Motivation

r1[x] w1[x] r2[x] w2[y] r1[y] w1[z] c1 w2[a] c2
not serializable.

- If r1[y] had arrived at the scheduler before w2[y] the schedule would have been serializable.

Timestamp ordering: scheduler forces serialization according to TA age, Multiversion protocols?

Multiversion CC: Read a consistent version which has not been written by an uncommitted TA:
writers of x : prepare a new version x'
readers: read a committed version

Multiversion schedules

r1[x] w1[x] r2[x] w2[y'] r1[y] w1[z] c1 w2[a] c2
T2 produces version y', but T1 read an old, committed version y.
No conflict any more: T1 read a version of x which was committed before BOT(T1)

T0 -> T1

Notation:
A data object x has **versions** xi, xj such that transactions Ti , Tj have written ("produced") the corresponding versions

e.g. T2 reads version of x which was produced by T0

Serializability

Conflict serializability inappropriate:
MC: $H_1 = w_0(x_0)\ c_0\ w_1(x_1)\ w_1(z_1)\ c_1\ r_2(x_0)\ w_2(y_2)\ c_2$

One copy: $H_1' = w_0(x)\ c_0\ w_1(x)\ w_1(z)\ c_1\ r_2(x)\ w_2(y)\ c_2$

Same conflicts on x, but H_1' does not have the same conflict properties as H_1:
H1: T2 reads from T0
H1' T0 < T1 < T2 since T2 reads from T1

Conflict properties different: w1(x1), r2(x0) **not in conflict**

Correctness reasoning

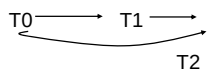
Conflict equivalence not appropriate, but…
View equivalence:
Reads-From relation of schedule H is equivalent to Reads-From relation of some serial schedule S

First attempt:
Scheduler has to produce MV histories H which are view serializable to some **one-version history** (1V history),
e.g. serialization Graph SG(H) is acyclic

Correctness reasoning

Not sufficient to show that MV serialization graph is cycle free !

Example:
w0(x0) w0(y0)c0 r1(x0) r1(y0) w1(x1) w1(y1) c1 r2(x0) r2(y1) c2
not equivalent to any 1V history, but SG acyclic



1 V history (e.g.)
w0(x) w0(y)c0 r2(x) r2(y) c2 r1(x) r1(y) w1(x) w1(y) c1
T2 reads from T0, in the MV history from T0 and from T1

Correctness reasoning

Problem in the example above:

H: w0(x0) w0(y0)c0 …. w1(x1) w1(y1) c1 r2(x0) r2(y1) c2

T2 reads from T0 but the value read has already been modified by a subsequent TA

An **MV history is a <u>1-serial MV history</u>**, if for every read-from triple (Tj, x ,Ti), i.e. Ti reads x from Tj, **Tj is the last TA preceding Ti which produced any version of x**

H not 1-serializable: (T0,x,T2) but T1 produced a newer version

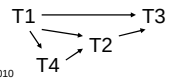
Correctness

1-serial MV histories are **equivalent to serial 1-version histories**

Proof for correctness of a scheduler has to show:
The scheduled MV history is 1-serial

w1(x1) r1(x1) w1(y1) c1 r4(y1) w2(y2) r3(x1)c2 r3(y2)c3 r4(x1)c4 y1 ??

w1(x) r1(x) w1(y) c1 r4(y) w2(y) r3(x)c2 r3(y)c3 r4(x)c4



Projection to 1-version history

MVCC correctness

Deciding if a MV history is 1-serial MV is NP complete :(

But there is a polynominal time characterization of "**multiversion conflicts**"

Intuitively: rl(xi) and wk(xj) are not in conflict, since k writes a different version than l reads.

Gives rise to an MV conflict graph
MV-conflict serializability ⇒ no cycles

6.2 MV timestamp ordering

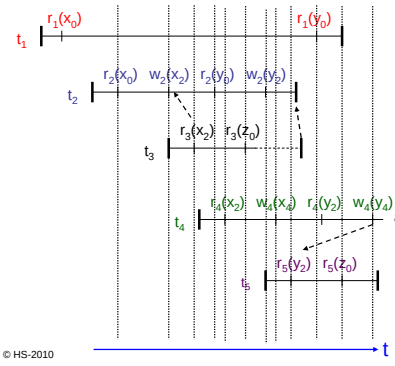
Wanted: **schedulers which guarantee conflict serializability using multiple versions**

MV Timestamp ordering

- each transaction T_i is assigned a unique timestamp $ts(t_i)$
- $r_i(x)$ is mapped to $r_i(x_k)$ where x_k is the version that carries the largest timestamp $\leq ts(t_i)$
- $w_i(x)$ is
  - rejected if there is $r_j(x_k)$ with $ts(t_k) < ts(t_i) < ts(t_j)$
  - mapped into $w_i(x_i)$ otherwise
- c_i is delayed until c_j of all transactions T_j that have written versions read by T_i

Ti wants to write a new version of x, the younger TA Tj has read an older version

MV Timestamp ordering



TO: every step is executed in "call-order" or TA is aborted

Reads always succeed!

How many versions needed?

6.3 MV2PL scheduling

Two Version 2 PL approach

Requirements:

- Exactly **one <u>committed</u> version x** exists, **<u>at most one</u> version x' which is being <u>produced</u>** (written) ⇒ use write locks for w-w conflicts
- **reading** of the committed version x is possible if a new version is produced ⇒ **no r-w conflict**

Critical situation: **commit newly produced version x'**
(⇒ forget old committed version x – only one may exist!)

MV2PL : conversion

- **Conversion** of x to the new version x' has to be done with care:
  - no reader should read x (or x'?) during conversion
  - if reader of x still active, do not convert

Needed: (1) read and write locks
(2) certify lock for the conversion phase

Example 2PL MVCC

Input schedule

s = $r_1(x)\ w_1(x)\ r_2(x)\ \ r_1(y)\ w_2(x)\ c_2\ w_1(y)\ c_1$

Blocked until T1 committed

MV2PL output schedule?

$r_1(x_0)\ w_1(x_1)\ r_2(x_0)\ r_1(y_0)\ w_1(y_1)\ c_1\ w_2(x_2)\ c_2$

y0 committed
y2 is not

2 Version MV2PL

Lock based MVCC ("MV2PL")

	R	W	C
R	+	+	-
W	+	-	-
C	-	-	-

C = Certify

Prevent w-w-conflict

No concurrency on x during certification

r-w-conflict prevented by version selection:
- committed for read
- new version for write

Multiversion concurrency

Two-version-2PL MVCC
- **Readers benefit**, **not writers**
- Certify: **lock conversion** from write lock to certify lock
- May be generalized to more than one uncommitted
- 2 versions important in practice

Scheduler:
- rl (x) : set read lock immediately on consistent version of x
- wl(x) : if not write locked, set write lock on x to produce a new uncommitted version x', else block TA
- cl (x) : if neither read-locked nor write-locked cl(x) is granted to other TAs, x' will be written as the new consistent version by the TA, else block TA

Correctness? Deadlocks? Read locks needed?

2V MV 2PL

x0,y0,,z0 : consistent state of x,y,z
xi := value of x produced by TAi

call sequence:
r1(x) w2(y) r1(y) w1(x) c1 r3(y) r3(z) w3(z) w2(x) c2 w4(z)c4 c3

T1 r1(x0) r1(y0) w1(x1) c1 t

T2 w2(y2) r2(x1)w2(x2) c2

T3 r3(y0) r3(...

w4(z4)..... c4

MVCC for Read only Transactions

Read only transactions occur frequently
- e.g. single SELECT expression,
  - (e.g. Select...from product_catalogue where...)
  - summarization, grouping (decision support, DWH)

⇒ should not be delayed by writers

Read-only transactions should read (the last) consistent state

Last consistent state for R-only TA R:
- last committed value(s) before R starts

Read only MVCC: solution

- **Update transactions** make a new version of updated data x,y,.. with a **<u>timestamp t at commit</u>**, version of x,y,.. is t

- **Read-only TA** with timestamp t' **reads only those values v with version t_v , t_v = max {t''_v , t''_v < t'}**
  ⇒ read the last committed values.
  ⇒ **No read locks** for read only TA

Update TA use conventional 2PL protocol with r and w locks.

MVCC / Read Only TAs: Example

call sequence: TA1, TA4 and TA5 are RO
r1(x) r2(x)w2(x)r3(x)r2(y)r4(z)w2(y)c2r4(x)c4w3(x)r5(z)c3r1(y)c1r5(x)c5

r1(x0)_____r1(y0)c1

r2(x0)w2(x2)__r2(y0)____w2(y)c2

r3(x).....blocked.....r3(x2)__w3(x3)c3

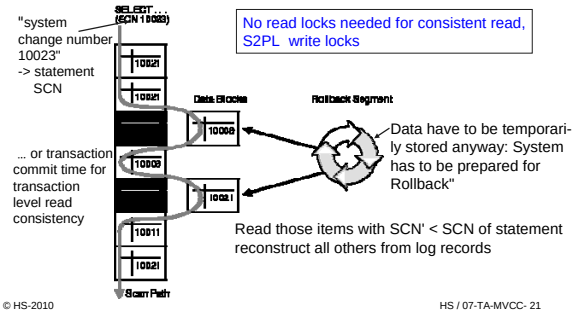
r4(z0)____ r4(x0)c4

r5(z0)_____r5(x2)c5

r1(y0): there exists a newer version y2, but RO_TA1 is older
r5(x2): reads x2 since TA3 which produces x3, commits after TA 5 begins
r4(x0): same with TA2, which produces x2
TA3 has been blocked, since TA2 holds lock on x, r3(x2) after TA2 commited

MVCC: How to implement versions?

Read Only Multiple version CC (used in Oracle)



No read locks needed for consistent read, S2PL write locks

Implementing versions

PostgreSQL

- do not delete!
- update = insert (append) of a new version!
- erase old versions which are not needed any more. (which ones?) – called *vacuum clean* in PG_SQL

6.5 Read Consistency MVCC

- **Combine Read-only TA** and **lock based cc**
  – Read-only TA as above
  – write (x):
    write lock the most current version of x and produce version (x_i, cts_i)
    ⇒ other writers have to wait
  – read(x):
    read **last committed version without locking**(!)
    ⇒ READ COMMITTED , not repeatable

w0(x0) c0 r2(x0) R3(x0) w1(x1) w1(y1) c1 r2(y1) r2(x1) R3(y0) R3(x0)

R= Read Only

why?

Read Consistency MVCC (2)

- Most significant! **No Read locks at all**!
- More than READ COMMITTED
  ... since READ ONLY TA serializable

- Fits to standard 2PL for R/O transaction

but...
no repeatable read, not serializable

Can we do better?
Even no write locks??

6.6 SNAPSHOT Isolation

'writes' are the problem – of course.

Suppose: w0(x0), c0, r1(x0) r2(x0) w1(x1) c1 w2(x2) c2

How can lost update be prevented?

Avoid conflicting writes of concurrent transactions!

⇒ **Write set of concurrent (overlapping!) transactions must be disjoint.**

SNAPSHOT isolation

- READs: version that was current (last committed) when TA started
  e.g. $max\ (x_j, cts_j), cts_j < ts(TA)$ (*)

  ⇒ **transaction level consistent, no read locks**

- if **write set** of TA_j und TA_i **not disjoint**: **abort** one of them!

How to implement with / without(!) write locks??

SNAPSHOT isolation

"First commit wins" implementation.
Transaction T:

1. make updates locally (like optimistic cc)
2. Commit step 1:
   validate: have all updated objects the same version number which T read? condition (*)
3. If yes: commit else abort

No writes locks, no read locks!!

SNAPSHOT isolation

Lock based implementation

Let **snapshot number (*) of TA1 be s**
TA1: write (x)

if **s < current version of x: abort**
Some TA* modified x after BOT(TA1) and **committed!**

expl: **r1(y0) r2(x0) w2(x2) c2 r1(x0) ~~w1(x1)~~**
TA1 aborts

else...

(*)Version that was current when TA started (or: read for the first time)

SNAPSHOT violation: locking

else: TA1 **locks x 2PL** if it wants to produce a new version.

if x already (write) locked by TA* TA1 waits until:
TA* commits ⇒ TA1 aborts
else
TA* aborts ⇒ TA1 commits
else commit.

- **No read locks** needed
- **Compatible with update in place**, if version reconstructed from the log.

Summary

MV concurrency control protocols are **highly relevant** from a practical point of view

- Standard 2V MV2PL and MV TO favor reads
- Read locks in 2V MV2PL needed to protect the consistent version read by a TA (certify lock not granted)
- Read only transactions without locks together with 2PL is standard in serious DBS
- Snapshot isolation: may be implemented without any locks.