12.3 Nonlocking schedulers

Freie Universität Berlin

12.3.1 not discussed in class

12.3.1 Time stamp ordering

Basic idea:

- assign **timestamp when transaction starts**
- if ts(t1) < ts(t2) … < ts(tn), then scheduler has to produce history equivalent* to t1, t2, t3, t4, ... tn

Timestamp ordering rule:
If pi(x) and qj(x) are **conflicting** operations,
then pi(x) is executed before qj(x) ⇔ ts(ti) < ts(tj)
or: pi(x) < qj(x) ⇔ ts(ti) < ts(tj)

(*) in case of conflicting operations – otherwise order arbitrary.

12-CC-32

Timestamp ordering

Freie Universität Berlin

TO concurrency control guarantees conflict-serializable schedules

Proof sketch:

Assume not ⇒ cycle in conflict graph (*)
cycle of length 2: ts(t1) < ts(t2) ∧ ts(t2) < ts(t1) #
induction over length of cycle ⇒ #

⇒ No cycle in conflict graph ✓

(*) Do not confuse with Wait-For-Graph – only defined for locking protocols

12-CC-33

TO Scheduler

Freie Universität Berlin

Basic principle:

Abort transaction if its operation is "too late"

Each object x has **two timestamps**
maxW(x): **timestamp of last writer** (TA which wrote x)
maxR(x): **timestamp of last reader**

Whether op(x) of TA t_i is "too late", depends on ts(t_i) and the read / write timestamps of x

12-CC-34

TO Scheduler: read

Freie Universität Berlin

Read: TA t_i with timestamp ts(t_i) wants to read x : r_i(x)

(i) maxW(x) > ts(t_i):
⇨ there is a younger TA which has written x
⇨ contradicts timestamp ordering:
t_i reads too late
⇨ **abort TA t_i , restart t_i**

(ii) maxW(x) < ts(t_i) ⇨ set maxR(x) = ts(t_i), go ahead
example: ------|------|----------- >
w_j(x) r_i(x) ts(t_j) < ts(t_i)

What would happen in a locking scheduler in this case?

12-CC-35

TO Scheduler: write

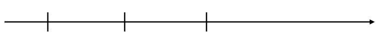
Freie Universität Berlin

Write: TA ti with timestamp ts(ti) wants to write x : wi(x)

(i) maxW(x) > ts(ti) ∨ maxR(x) > ts(ti) :
/* x has been *written or read by younger transaction*:
⇨ contradicts timestamp ordering
⇨ abort TA ti

(ii) otherwise: ⇨ schedule wi(x) for execution
set maxW(x) = ts(ti),

Why abort ?



wi(x) wj(x) abort(i) ts(ti) < ts(tj)

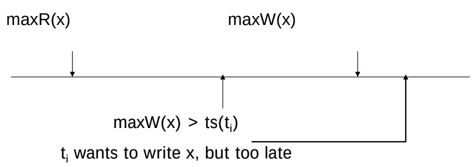
x would have been overwritten in serialization according to timestamp order anyway! ... ti < ...< tj....

12-CC-36

Thomas Write Rule

Freie Universität Berlin

Idea: younger write overwrites older write without changing effect of timestamp ordering



Rules for Writer t with timestamp ts(t):
1. maxR(x) > ts(t) : abort T
2. maxW(x) > ts(t) : skip write // Thomas write rule
3. otherwise write(x), maxW(x) = TS(t)

12-CC-37

Discussion

- Lightweight solution.
  - Serializable? Obvious
  - Why not replace 2PL in DBS?

- Timestamp ordering optimistic or pessimistic??
- There are more protocols using timestamps
  (BOT-timestamp or EOT-timestamp)
  but different from timestamp ordering protocol

12.3.2 Optimistic CC

Optimistic concurrency control

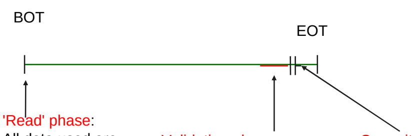
- Locks are expensive
- Few conflicts ⇨ retrospective check for conflicts cheaper

Basic idea: all transactions **work on copies,**
check for conflicts before write into DB

if conflict detected (*): abort TA else commit

(*) how to detect conflicts??

Phases of optimistic cc

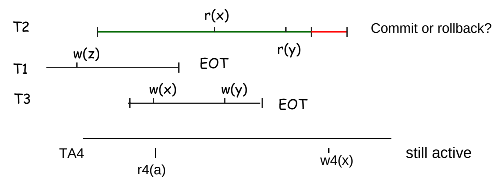


'Read' phase:
All data used are copied to private workspace and used by the application, some are modified, but not written int0 DB.

Validation phase:
any conflicts?
if yes: resolve

Commit phase:
write all (changed) data into DB

Backward oriented concurrency control (BOCC)



- **ReadSet** R(T) = data, transaction T has read in read phase
- **WriteSet** W (T) = data (**on copies**!), T has changed in read phase

Assumption: $W(T) \subseteq R(T)$ - necessary? why?

Example above: $x,y \in R(T2)$, $x,y \in W(T3)$, $z \in W(T1)$

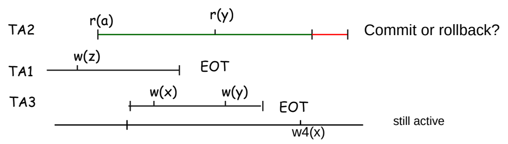
What is a conflict?

- Let $x \in R(T)$. T wants to validate.
- If a transaction **S different from T <u>read</u> x,**
  but did not commit ⇨ no problem

- If a transaction **S different from T** <u>committed</u> after BOT(T),
  DB state of x may be **different from x at BOT(T) ⇨ conflict**

BOCC_validate(T) :
  if for all transactions T' which committed after BOT(T) :
  $R(T) \cap W(T') = \varnothing$ then T.commit // successful validation
  else T.abort

Optimistic CC: BOCC



More aborts than necessary :
$R(TA2) \cap W(TA3) != \varnothing$.
Note: No abort when 2PL synchronization !

Question: Validation - what happens, if more than one TA validates?

Implementation

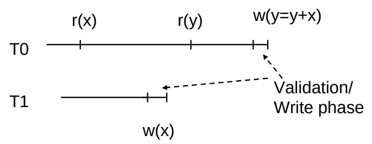
Implementation of backward oriented OCC

- Each object x has a timestamp t , where t is the commit time of the last transaction which modified x
- When T validates, it compares the current timestamp t_{new} of each object x with the timestamp t_{old} of x had when it was read by T.
- if (for all x read by T: $t_{old} = t_{new}$) commit; else abort T; start T again;

These timestamps have NOTHING to do with Concurrency Control using timestamp ordering !!

Implementation

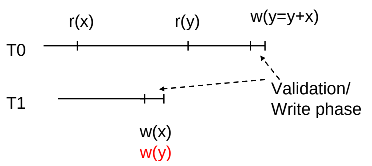
Have timestamps of objects x read but not written by T to be compared during validation?



Serializable: T0; T1

Implementation

Have timestamps of objects x read but not written by T to compared during validation?

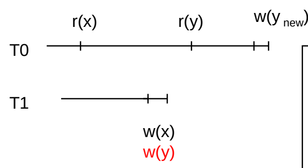


Cycle in conflict graph : T0; T1; T0

Consequence: records have to be checked which T0 read only!

Implementation

... timestamps of objects x read but not written by T have also to be compared during validation.



Cycle in conflict graph : T0; T1; T0

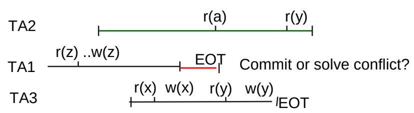
Only a problem, if y_{new} depends on x!

Implementations often assume, that update of **x is only dependent on the old value of x**, e.g. many OR mappers.
SQLServer: cursor can be defined **`OPTIMISTIC WITH VALUE`**,
In case of update of a row compares value read and value in database.
`OPTIMISTIC WITH VERSIONS`

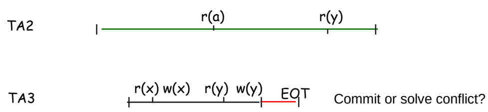
Optimistic CC: FOCC

Forward oriented optimistic Concurrency control (FOCC)
Forward looking validation phase:

If there is a running transaction T' which read data written by the validating transaction T then solve the conflict (e.g. kill T'), else commit



Concurrency: Optimistic CC



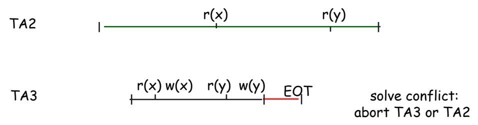
FOCC_validate(T) :
```
if(for all running transactions (T')
   R(T') ∩ W(T) = ∅  )
  T.commit    // successful validation
  else solve_conflict ( T, T')
```

R(T'): Read set of T' at validation time of T (current read set)

Optimistic Concurrency control

Validation of "**read only**" transactions T:
FOCC guarantees **successful** validation !

FOCC has greater flexibility
Validating TA may decide on victims!



- **Issues** for both approaches:
  fast validation – only one TA can validate at a time.
  Fast and atomic commit processing,
- Useful in situation with few expected conflicts.

Implementation of Read / Write sets

Thinkfood:

Is it possible to implement of Read / Write sets used by FOCC by means of **timestamps** ts(x) as BOCC?

- what about committed TA concurrent to validating?
- Important detail: how to avoid that read-timestamps attached to records have to be written back to disk? !

12.3.3 Principle of Multiversion Concurrency control

Arrows from TA2-ops to conflicting TA1-ops

Multiversion CC:
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.

Main idea of **multiversion concurrency control** : **Reads should see a consistent** (and committed) **state**, which might be older than the current object state.

Update strategies and versions

Required:
Different versions of an object
Particular important: 2 versions

Implementation depends on the how DB is updated:
- **update in place**: object is updated in the DB
  (compare: update of copy in optimistic cc)
- **No update** at all:
  each **update is an insert**
  of a new version (Postgres solution).

Isolation levels?

- What does read committed mean exactly?

w0(x0) c0 r2(x0) w1(x1) w1(y1) c1 r2(y1) r2(x1) (*)

TA2 reads only committed data: READ COMMITTED

But not REPEATABLE, not SERIALIZABLE

(*) $w_i(x_i)$ means: TA_i produces version i of x: x_i;
$r_j(y_k)$ means: TA_j reads version of y produced by TA_k

Transaction level consistency

Idea: each **transaction reads only objects from** the **same DB state**

Requirement: **each version** of an object has as a **timestamp the commit time** cts_i of the TAi which produced this version:

e.g.: (x_i, cts_i) means: TA_i produced this version and committed at ts_i

Transaction level consistency

Def.: A Transaction TA_i with BOT time stamp ts(i) is **transaction level consistent** iff for all objects x the version (x_i, cts_i) is read by TA_i which is defined by:
$cts_i = \max \{cts_j : (x_j, cts_j) \text{ is a version and } cts_j < ts_i\}$

Def.: Snapshot number: cts assigned to TA .
Reflects the state of the DB which TA observes at BOT.

If only one version: nothing new – read committed.
Multiple versions: Need Read-only TA read locks at all?

MVCC pragmatics

- Difficult to integrate MVCC into a DBS kernel
- Even difficult protocols in general
- Postgres: The design decision never to update but to append new "record states" greatly alleviates MVC synchronisation,
- Easy:
  Process **Read only transactions** different from R/W transactions.

Read-only Transactions

Assume scheduler knows that TA t will only read, why read-locks?

- Goal: r(x) of t should never be member of a conflict pair
  $\Rightarrow$ no locks, no delay, execute immediately

SQL:
`SET TRANSACTION READ ONLY`
`FOR READ ONLY` in cursor definition

Important examples: e.g. browsing a product catalogue

Read Only transaction

Basic idea of Read-only transactions:
- **several version of x** with **commit-timestamp** of TA which wrote x ("produced this version of x"): (x(1),ts1),..,(x(k),tsk)
- **Read-only TA** t with **begin** timestamp **ts(t)** reads version **(x(i),tsi)** with **tsi = max{tsj: tsj < ts(t)}**

- Why does it work?
- Why is more than one version needed?

Characteristics of RO-TA

- **A RO-Transaction always is (reads) transaction consistent.**
- **No Read locks** !
  Obvious: no conflicts – reads on committed versions
- More than two versions needed.

Issue: management of (in principle) arbitrary many versions

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(y2)c2
r3(x).......blocked..........r3(x2)__w3(x3)c3
R4(z0)____ **R4**(x0)c4
r6(y2)_w6(y6) c6
R5(z0)_________R5(x2)c5

3 version of y needed!

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

Multiple versions?

Assumption: update in place – otherwise next to trivial

Use DBS log for reconstruction of old versions!

Log: all operation of the DBS have to logged in a log file for recovery purposes (see below)

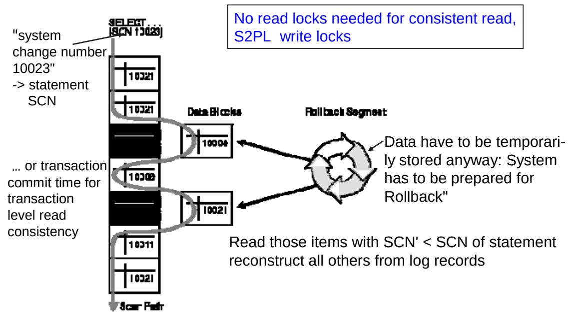
"**Roll back**" for reconstruction past states of object x.

When needed?

MVCC: How to implement versions

Read Only Multiple version CC (used in Oracle)

No read locks needed for consistent read, S2PL write locks



Roadmap MVCC

What we have:

No Read-locks for RO-TA if more than one version per object

What we would like:

- No Read locks at all!?

- No write locks??

Overall goal: **decrease** synchronization (locking) **overhead** if **more than on version** available.

Read Consistency MVCC

- **Combine Read-only TA** and **lock based cc**
  - Read-only 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

Read consistent MVCC

Example

→ t

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

R= Read Only

why?

Remember:
READ_COMMITTED with 2PL requires a (short) read lock on an item x to be read.
Why needed with one version, but not with more than one?

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 transactions

but...
no repeatable read, not serializable

- How to avoid lost updates and guarantee repeatable read without reintroducing read locks?
- Can write locks be avoided? ??

SNAPSHOT Isolation

'writes' are the problem .

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

- **Avoid conflicting writes of concurrent transactions!**

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

... and **Repeatable Read?**

SNAPSHOT isolation

- read(x): versionof x that was current 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 TAi **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?
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!**

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

else...

TA1 reads TA level consistent, i.e. the version of x that was current at BOT of TA1

... →

SNAPSHOT isolation: 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
- **Repeatable Read,** but not Serializable.
- **Compatible with update in place**, if version reconstructed from the log.

Serializability and versions

Disadvantage of snapshot isolation:
- not serializable in all cases
- Abort of a TA in case of w-w conflicts
  Maybe waiting for the release of a lock would be sufficient?

Generalized lock protocol with 2 versions only:

- only one TA can prepare a new version
  ⇒ Standard lock protocol (2 PL)
- Writer wants to publish new version of x:
  no reader of x should still be active.

Multiversion CC: 2 versions (2VMVCC)

2 versions of each object x:

- a consistent one x_j with commit time of last modifying transaction t_j as a timestamp
- a writer t_i may prepare a second version x_i, not visible until commit of writing TA t_i

Restrictions for 2VMCC:
- Never two writers at the same time on the same object
  ⇒ only one new version can be prepared
- New version cannot be published, if a reader of the (consistent) old version is still active

2VMVCC

r1(x0) w1(x1) r1(z0) w1(z1) c1
r2(x0) w2(y2) r2(z1) c2

Suppose z1 = z0+x1: inconsistent – two different states of x in the TA t_2 , read not repeatable – remember: only 2 versions

Delay the commit of t_1 until all readers of objects written by t_1 (i.e. x, z) **have committed:**

r1(x0) w1(x1) r1(z0) w1(z1) (delayed) c1
r2(x0) w2(y2) r2(z0) c2

Multiversion concurrency

Lock based MVCC ("MVCC2PL")

w(x): **write lock x** if not locked, else wait
r(x): **read lock on x always granted** for last consistent version
c(x): acquire **certify lock**, if prepared version of x is to become the current consistent version, granted, if now reader or writer on x active.

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

Compatibility matrix

Multiversion concurrency

Two-version-2PL MVCC
has only **one uncommitted** version, one consistent ("current") version because writes are incompatible
Readers benefit, not writers
- May be generalized to more than one uncommitted
- MVCC is most in practice

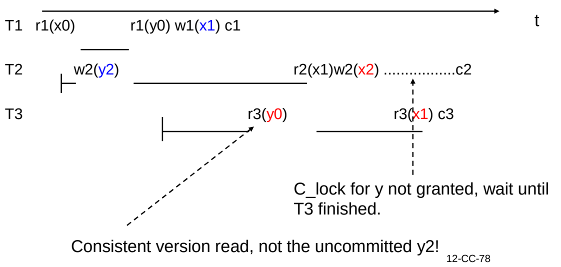
Deadlocks?

Read locks needed why?

Serializable?

2PL-MVCC

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) r2(x) w2(x) c2 r3(x) c3



Update replaced by append

The Postgres solution...

- ... is much trickier
- ... will be presumably analyzed in DB-Tech (winter term)

- **MVCC also employed in non-DB applications**

Summary: Transactions and concurrency

- Transactions: very **import concept**
- Model for **consistent, isolated execution of concurrent TAs**
- Scheduler has to decide on **interleaving of operations**
- **Serializability:** correctness criterion
- Implementation of serializability: **concurrency control:**

  2-phase-locking, time stamping, multiversion cc ...and more
- Strict 2PL restrictive, but employed in many DBS
- **Read-mostly DB** has fostered **MVCC,** today in **most DBS** Oracle, Postgres, SQL-Server and more...

see comprehensive overview of synchronization in DBS in the reader