Carnegie Mellon Univ.  
Dept. of Computer Science  
15-415/615 - DB Applications

Lecture #23: Alternative Concurrency Control Methods (R&G ch. 17)

Outline

• serializability; 2PL; deadlocks
• Locking granularity
• Tree locking protocols
• Phantoms & predicate locking
• Optimistic CC
• Timestamp based methods
• Multiversion CC

very popular – used in all commercial systems

Optimistic CC (Kung&Robinson)

• Assumption: conflicts are rare
• Optimize for the no-conflict case.
Optimistic CC (Kung&Robinson)

• All transactions consist of three phases
  – **Read**: all writes are to **private** storage.
  – **Validation**: check for no conflicts
  – **Write**: flush ‘writes’ (or abort!)

<table>
<thead>
<tr>
<th>Read Phase</th>
<th>Validation</th>
<th>Write Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>All writes private</td>
<td>Make local writes public</td>
<td></td>
</tr>
</tbody>
</table>

Why Might this Make Sense?

• All transactions are readers
• Many transactions,
  – each accessing/modifying few tuples
  – from many tuples
  – Low probability of conflict, so again locking is wasted
Validation Phase

- Goal: guarantee only serializable schedules
- Intuitively: at validation, Tj checks its ‘elders’ for RW and WW conflicts
- and makes sure that all conflicts go one way (from elder to younger)

Specifically:
- Assign each transaction a TN (transaction number)
- Require TN order to be the serialization order
- If TN(Ti) < TN(Tj) ⇒ ONE of the following must hold:

Validation Phase (1)

1. Ti completes W before Tj starts R
Correctness

1. Ti completes W before Tj starts R

\[ \text{ok W-R} \]
\[ \text{ok W-W} \]
\[ \text{ok R-W} \]

Correctness

• In fact, this is a true serial execution

Validation Phase (2)

2. \( WS(Ti) \cap RS(Tj) = \emptyset \) and

Ti completes W before Tj starts W
Correctness

2. $\text{WS}(T_i) \cap \text{RS}(T_j) = \emptyset$ and
   $T_i$ completes $W$ before $T_j$ starts $W$

   \[ \begin{array}{c}
   T_i \quad R \quad V \quad W \\
   T_j \quad R \quad V \quad W \\
   \end{array} \]

   no W-R
   ok W-W
   ok R-W

Validation Phase (3)

3. $\text{WS}(T_i) \cap \text{RS}(T_j) = \emptyset$ and
   $\text{WS}(T_i) \cap \text{WS}(T_j) = \emptyset$ and
   $T_i$ completes its $R$ before $T_j$ completes its $R$

   \[ \begin{array}{c}
   T_i \quad R \quad V \quad W \\
   T_j \quad R \quad V \quad W \\
   \end{array} \]

Correctness:

3. $\text{WS}(T_i) \cap \text{RS}(T_j) = \emptyset$ and
   $\text{WS}(T_i) \cap \text{WS}(T_j) = \emptyset$ and
   $T_i$ completes its $R$ before $T_j$ completes its $R$

   \[ \begin{array}{c}
   T_i \quad R \quad V \quad W \\
   T_j \quad R \quad V \quad W \\
   \end{array} \]

   no W-R
   no W-W
   ok R-W
Observations

• When to better assign TN’s?
• at beginning of read phase: Tj has to wait...

Tj has to wait for W(Ti)

Observations

• When to better assign TN’s?
• at beginning of validation phase:
  – Tj can start
  – condition (3): automatic!

Tj has to wait for W(Ti)

A Serial Validation Technique

Goal: to ensure conditions 1 and/or 2 above.
• Requires that write phases be done serially
• Validation + Write: in a ‘critical section’
Serial Validation Algorithm

1. Record \textit{start\_tn} when Xact starts (to identify active Xacts later)
2. Obtain the Xact’s real Transaction Number (TN) at the start of validation phase
3. Record read set and write set while running and write into local copy
4. Do validation and write phase inside a critical section

Opt CC vs. Locking

Locking:
- order is of first lock;
- wait
- on deadlock, abort

Optimistic cc:
- order is of \textit{TN(i)}
- abort
- on starvation, lock

Conclusions

- Analysis [Agrawal, Carey, Livny, ‘87]:
  - locking performs well
- All vendors use locking
- Optimistic cc: promising when resource utilization is low.
Outline

- serializability; 2PL; deadlocks
- Locking granularity
- Tree locking protocols
- Phantoms & predicate locking
- Optimistic CC
- Timestamp based methods
  - Multiversion CC

Timestamp based

Motivation:
- can we avoid locks
- AND also avoid the 'critical section' of optimistic CC?

Main idea
- each xact goes ahead reading and writing
- if it tries to access an object 'from the future', it aborts
  (Resembles 'optimistic cc', but writes go directly on the db)
Timestamp CC:

- each xact gets a timestamp (TS)
- each object has
  - a read-timestamp (RTS) (latest xact that read it)
  - and a write-timestamp (WTS) (latest xact that wrote it)

Timestamp CC

- If action ai of Xact Ti conflicts with action aj of Xact Tj, and TS(Ti) < TS (Tj), then ai must occur before aj. Otherwise, restart the offending Xact.
- Specifically:

On ‘reads’:

```
<table>
<thead>
<tr>
<th>Time</th>
<th>T1:&lt;1&gt;</th>
<th>T2:&lt;2&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(O)</td>
<td>W(O)</td>
</tr>
<tr>
<td></td>
<td>....</td>
<td>....</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>RTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>-&lt;2&gt;</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>WTS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>object</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
</tr>
</tbody>
</table>
```

On ‘reads’:

<table>
<thead>
<tr>
<th>Time</th>
<th>T1:≤1&gt;</th>
<th>R(O)</th>
<th>T1 ABORTS!</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:≤2&gt;</td>
<td>W(O)</td>
<td>....</td>
<td>...</td>
</tr>
</tbody>
</table>

Object O

- RTS
- WTS

Timestamp CC – Reads:

- If TS(T) < WTS(O), this violates timestamp order of T w.r.t. writer of O.
  - So, abort T and restart it (with same TS? why?)
- Else
  - Allow T to read O.
  - Update RTS(O) to max(RTS(O), TS(T))

Timestamp CC - Reads

Notice: Change to RTS(O) on reads must be written to disk! This and restarts represent overheads.
On ‘writes’: RW conflict

<table>
<thead>
<tr>
<th>time</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: &lt;1&gt;</td>
<td>....</td>
<td>W(O)</td>
</tr>
<tr>
<td>T2: &lt;2&gt;</td>
<td>....</td>
<td>R(O)</td>
</tr>
</tbody>
</table>

Object: O

RTS
WTS

T1: ABORTS

On ‘writes’: WW conflict

<table>
<thead>
<tr>
<th>time</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: &lt;1&gt;</td>
<td>....</td>
<td>W(O)</td>
</tr>
<tr>
<td>T2: &lt;2&gt;</td>
<td>....</td>
<td>W(O)</td>
</tr>
</tbody>
</table>

Object: O

RTS
WTS

--

11
On ‘writes’: WW conflict

<table>
<thead>
<tr>
<th>Time</th>
<th>T1: W(O)</th>
<th>T2: W(O)</th>
<th>T1: STAYS!!</th>
<th>(Thomas rule: ignore the W of T1)</th>
</tr>
</thead>
</table>

Timestamp CC: Writes

- If TS(T) < RTS(O), abort and restart T.
- If TS(T) < WTS(O), violates timestamp order of T w.r.t. writer of O.
  - Thomas Write Rule: ignore W op, and continue with T
- Else, allow T to write O.
  - and update the WTS(O)

Digging deeper:

- How about recoverability (ie, cascading aborts?)
- Can they appear, under timestamp CC?

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>Commit</td>
</tr>
<tr>
<td>Abort</td>
<td>BAD</td>
</tr>
</tbody>
</table>
Digging deeper:

• How about recoverability (i.e., cascading aborts?)
• Can they appear, under timestamp CC?
• Yes!

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>Commit</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Abort</td>
<td>BAD</td>
</tr>
</tbody>
</table>

Timestamp CC and Recoverability

**Recoverable schedule:** xacts commit only after (and if) all xacts whose changes they read commit

- Unrecoverable schedules are allowed by Timestamp CC!
- (Explain why?)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>Commit</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Abort</td>
<td>BAD</td>
</tr>
</tbody>
</table>

Timestamp CC and Recoverability

- Timestamp CC can be modified, to give recoverable schedules — how?
Timestamp CC and Recoverability

• Timestamp CC can be modified, to give recoverable schedules – how?
• A:
  – Buffer all writes until writer commits (but update WTS(O) when the write is allowed.)
  – Block readers T (where TS(T) > WTS(O)) until writer of O commits.

Similar to writers holding X locks until commit, (but not =2PL).

Outline

• serializability; 2PL; deadlocks
• Locking granularity
• Tree locking protocols
• Phantoms & predicate locking
• Optimistic CC
• Timestamp based methods
  • Multiversion CC

Multiversion CC

• Readers need NO LOCKS!
  – How would you do it?
Multiversion CC

- Readers need NO LOCKS!
  - keep a history of all attribute values
  - give each reader the appropriate version
  - (abort the belated writers)

Multiversion Timestamp CC

- Idea: Let writers make a “new” copy while readers use an appropriate “old” copy:

  MAIN SEGMENT
  (Current versions of DB objects)

  VERSION POOL
  (Older versions that may be useful for some active readers.)

  Readers are always allowed to proceed.
  - But may be blocked until writer commits.

Multiversion CC (Contd.)

- Each Xact is classified as Reader or Writer.
  - Writer may write some object; Reader never will.
  - Xact declares whether it is a Reader when it begins.
- Each version of an object has its writer’s TS as its WTS, and the TS of the Xact that most recently read this version as its RTS.
- Versions are chained backward; we can discard versions that are “too old to be of interest”.
Reader Xact

- Find **newest version** with WTS < TS(T).
- Reader Xacts are never restarted.
  - However, might block until writer of the appropriate version commits.

Writer Xact

- try to insert/append a new version
- abort if there is a reader ‘from the future’, that read an older version
- Specifically:

```
T1:<1>  .... W(O) ........
T2:<2>                                                        .... W(O) ........
T3:<3>                             .... R(O) ........
```

T1 creates the first version of object O
Writer Xact

- To read an object, follows reader protocol.
- To write an object:
  - Finds newest version V s.t. WTS < TS(T).
  - If RTS(V) < TS(T), T makes a copy CV of V, with a pointer to V, with WTS(CV) = TS(T), RTS(CV) = TS(T). (Write is buffered until T commits; other Xacts can see TS values but can’t read version CV.)
  - Else, reject write.
Writer Xact

- To read an object, follows reader protocol.
- To write an object:
  - Finds newest version $V$ s.t. $\text{WTS} < \text{TS}(T)$.
  - If $\text{RTS}(V) < \text{TS}(T)$, $T$ makes a copy $CV$ of $V$, with a pointer to $V$, with $\text{WTS}(CV) = \text{TS}(T)$, $\text{RTS}(CV) = \text{TS}(T)$. (Write is buffered until $T$ commits; other Xacts can see TS values but can’t read version $CV$.)
  - Else, reject write.

Summary – optimistic CC

- Optimistic CC (using a posteriori “validation”) aims to minimize CC overheads in an “optimistic” environment in which reads are common and writes are rare.
- Optimistic CC has its own overheads however; most real systems use locking.

Summary – timestamp based

- Timestamp CC allows some serializable schedules that 2PL does not (although converse is also true).
- Ensuring recoverability requires ability to block Xacts, which is similar to locking.
Summary - multiversion

- read-only Xacts are never restarted; they can always read a suitable older version.
- Has additional overhead of version maintenance.
  - Oracle uses a flavor of multiversion CC

Overall summary of CC

- Most commercial systems use
  - Locking AND
  - with wait-for graphs for deadlock detection AND
  - multiple granularity locking (table, page, row)