Optimistic Concurrency Control

Anastassia Ailamaki
http://www.cs.cmu.edu/~natassa

Optimistic CC (Kung&Robinson)

- Assumption: conflicts are rare
- Optimize for the no-conflict case.
- All transactions consist of three phases
  - **Read**: Here, all writes are to private storage.
  - **Validation**: Make sure no conflicts have occurred.
  - **Write**: If Validation was successful, make writes public. (If not, 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>Check for conflicts</td>
<td>Make local writes public</td>
</tr>
</tbody>
</table>

Why Might this Make Sense?

- All transactions are readers
  - The system will be setting and releasing locks for no reason at all
- Lots of transactions, each accessing/modifying only a small amount of data, large total amount of data
  - Low probability of conflict, so again locking is wasted
- Fraction of transaction execution in which conflicts “really take place” is small compared to total path length
  - Locks until end of Xtion are way too restrictive most of the time
Validation Phase (1)

- Goal: guarantee only serializable schedules result.
- Technique:
  - Assign each transaction a TN (transaction number)
  - Require TN order to be the serialization order
  - If TN(Ti) < TN(Tj) \Rightarrow \text{ONE of the following must hold:}
  1. Ti completes W before Tj starts R
     \[ \text{Ti} \quad W \quad V \quad R \quad Tj \quad V \quad W \]

Validation Phase (2)

2. WS(Ti) \cap RS(Tj) = \emptyset \text{ and } Ti completes W before Tj starts W
   \[ \text{Ti} \quad W \quad V \quad R \quad Tj \quad V \quad W \]

Comments:
- No problem with Tj reading values previous to Ti's writes (nothing in common there)
- No problem with Ti overwriting Tj's writes (no overlap in time)

Validation Phase (3)

3. WS(Ti) \cap RS(Tj) = \emptyset \text{ and } WS(Ti) \cap WS(Tj) = \emptyset \text{ and } Ti completes its R before Tj completes its R
   \[ \text{Ti} \quad W \quad V \quad R \quad Tj \quad V \quad W \]

Comments:
- No problem with Tj getting (or missing) input from Ti, as there is nothing that Ti writes that Tj touches
- Since Ti finishes its R before Tj finishes its R, Ti won’t read any output from Tj either
- No overwrite problems as write-sets are disjoint
Correctness

All of conflict types (WR, RW, WW) go one way
- Condition 1: true serial execution
- Condition 2
  - No W-R conflicts since WS(T_i) intersect RS(T_j) = NULL
  - In R-W conflicts, T_i precedes T_j, since T_i's W (and hence R) of T_i
    precedes that of T_j
  - In W-W conflicts, T_i precedes T_j by definition
- Condition 3
  - No W-R conflicts since WS(T_i) intersect RS(T_j) = NULL
  - No W-W conflicts since WS(T_i) intersect WS(T_j) = NULL
  - In all R-W conflicts, T_i precedes T_j, since the T_i's R precedes T_j's W

Observations

- When to better assign TN's?
  - at beginning of validation phase
- T with very long R: Problem?
  - check ALL T's within its lifetime!!!
  - Requires unbounded buffer space. Solution?
  - Bound buffer, toss out when full, abort possibly affected T's
  - Starvation!
- Serial/Parallel validation – Pros & cons?

A Serial Validation Technique

**Goal:** to ensure conditions one and/or two above.

Requires that write phases be done serially.
Serial Validation Algorithm

1. Record \texttt{start\_tn} when Xtion starts (to identify active Xtions later)
2. Obtain the Xtion'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

Serial Validation: Critical Section

\begin{verbatim}
beginCriticalSection
\texttt{finish\_tn} := \texttt{currentTN}; /* tentatively assign tn */
\texttt{valid} := \texttt{true};
\texttt{for T from start\_tn + 1 to finish\_tn do}
  \texttt{if (write set of Xtion T intersects read set)}
    \texttt{then valid := false;}
  \texttt{if valid}
    \texttt{then \{ write phase; currentTN++; \texttt{tn} := currentTN \}}
\texttt{endCriticalSection}
\texttt{if valid then cleanup() else backup();}
\end{verbatim}

Serial Validation (cont.)

\textbf{Optimization}: Do not assign TN (TID) unless success!

Informally,
1. check current TN;
2. check everything from start until current TN;
3. then enter critical region and do the rest.

- Read-only Xtions are not assigned TNs; just check write sets of Xtions with \texttt{start\_tn} < TN < \texttt{finish\_tn}
Parallel Validation

Only real difference:
now must check condition 3, using *active*, the set of Xtions that have finished their read phase but have not yet completed their write phases.

Algorithm on the board

Q: What’s wrong with this picture?
A Xtion may cause another Xtion to abort and then abort itself!!!

Performance: Opt CC vs. Locking

- With optimistic, conflicts are
  - found when the transaction is basically done
  - resolved by aborts/restarts (that waste CPU & I/O resources)
- With locking, conflicts are resolved by waits
- With optimistic, updates incur a copy.
- With locking, updates are performed in place
- “Optimistic works well when conflicts are rare”
- In that case, smart locking works well too
- Optimistic incurs non-trivial cost of maintaining read and write sets.