
Optimistic CC

Christos Faloutsos
www.cs.cmu.edu/~christos

Roadmap
1) Roots: System R and Ingres
2) Implementation: buffering, indexing, q-opt
3) Transactions: locking, recovery
granularity of locks
optimistic CC
B-trees

4) Distributed DBMSs
5) Parallel DBMSs: Gamma, Alphasort

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!)

Write Phase

Why Might this Make Sense?

• All transactions are readers
• 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 Xact are way too restrictive most of the time

Validation Phase (1)

• Goal: guarantee only serializable schedules
• intuitively: at validation, Tj checks it’s ‘elders’ for RW and WW conflicts - specifically:
• Technique:
  – 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:

Why Might this Make Sense?
Validation Phase

1. Ti completes W before Tj starts R

Validation Phase (2)

2. WS(Ti) ∩ RS(Tj) = ∅ and Ti completes W before Tj starts 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) ∩ RS(Tj) = ∅ and
   WS(Ti) ∩ WS(Tj) = ∅ and
   Ti completes its R before Tj completes its R

Validation Phase (cont’d)

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(Ti) intersect RS(Tj) = NULL
  - In R-W conflicts, Ti precedes Tj, since Ti’s W (and hence R) of Ti precedes that of Tj
  - In W-W conflicts, Ti precedes Tj by definition

Correctness (cont’d)

- Condition 3
  - No W-R conflicts since WS(Ti) intersect RS(Tj) = NULL
  - No W-W conflicts since WS(Ti) intersect WS(Tj) = NULL
  - In all R-W conflicts, Ti precedes Tj, since the Ti’s R precedes Tj’s W
Observations

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

\[ \text{Tj has to wait for } W(Ti) \]

Observations (cont’d)

- BUT: subtle problem: T with very long R!
  - must 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 1 and/or 2 above.

Requires that write phases be done serially.

Serial Validation: Critical Section

beginCriticalSection
\[ \text{finish}_tn := \text{currentTN}; \]
valid := true;
for T from \( \text{start}_tn + 1 \) to \( \text{finish}_tn \) do
  if (write set of Xact T intersects read set)
    then valid := false;
  if valid
    then \{ write phase; currentTN++; tn := currentTN \}
endCriticalSection
if valid then cleanup() else backup();
Serial Validation (cont.)

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 Xacts are not assigned TNs; just check write sets
of Xacts with \( \text{start}_tn < TN < \text{finish}_tn \)

A Serial Validation Technique

Optimization: move some of the validation
outside the critical section.

Parallel Validation

Only real difference:
now must check condition 3, using active,
the set of Xacts that have finished their read
phase but have not yet completed their write
phases.
Algorithm: in paper
Subtlety: An aborting Xact may cause another
Xact to abort!

Opt CC vs. Locking

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

Optimistic cc
- order is of \( t(i) \)
- abort
- on starvation, lock

Conclusions

- Analysis [Agrawal, Carey, Livny, ‘87]:
  – dynamic locking performs very well, in most
cases
- All vendors use locking
- optimistic cc: promising for OO systems, or
when resource utilization is low.