Consider

Begin Xact
Select avg (sal) from emp
End xact

Begin xact
Update emp set …
End xact

Lock escalation will occur;

Either reader will starve (without scheduling help)
Or
Writers will starve (while reader is doing a long query)

Both undesirable…… So what to do:

1) nearly always done – run analytics on a companion data warehouse
2) take the stall (rarely done)
3) run less than serializability

Hence:

Degree 0: (read uncommitted) no locks (guarantees nothing). Can read uncommitted data

Degree 1: (read committed) X locks held till EOT – R locks obtained only temporarily. (can only read committed data – however subsequent reads of the same thing can produce different answers

Degree 2: (repeatable read) X, S locks held till XOT – serializable unless you squint.

Degree 3: (serializable) S and X locks held till EOT plus solve the phantom problem. Really do the right thing

Choose what you want to pay for. Degree 0 solves big read problem –at the expense of getting the wrong answer.

4) use multi-version system

read is given a time stamp (TS). A write installs new timestamp and keeps old data for a while. Read is given the biggest TS less than his. I.e. read is "as of a time".

Reads set no locks. However, get a historical (consistent) answer. After a while can garbage collect old values – when they are no longer needed – i.e. there is no running xact older than the next guy in line.

Can be turned into a full cc system

MVCC. Give every Xact a timestamp

Have a read TS and a write TS for every "granule"

Read-only xact: get a TS.. Read whatever you want. If multiple versions, then read the one written just earlier than your time stamp. If reading the current version, install your TS if greater than the one that is there.

Update xact: given a TS. Read whatever you want, installing a read TS as above. Write a new value with your timestamp, keeping the old value, as above. Do this only if your timestamp greater than both ones there. Otherwise, commit suicide.

******

Locking is pessimistic – i.e. ensure no conflict by assuming the worst case. Other approaches to concurrency control are more aggressive:

Optimistic concurrency control (Kung and Robinson – late '70s)

Run transaction to completion. Check at end if there was a problem.

3 phases: read/write, validate, commit

Read/write: do logic normally. Any write goes to a private copy (think of it as an update list). Keep track of read-set and write-set. R(Ti) W (Ti)

At end, enter the validate phase:

For all xacts, Tj, which commited after I started:

W(Tj) intersect R (Ti) empty, if fail then abort (and restart) if succeed, then enter commit phase.

Commit: install updates

Issues: one validator at a time! One commiter at a time! Bottleneck

Issues: lose all work on a abort
Issues: starvation (cyclic restart)
Issues: a bit pessimistic – possible to restart when there is not a conflict.
****

So which one wins?

Several simulation studies in the 80's. Most have Mike Carey as an author or co-author.

Variables:

Prob (contention)
concurrent xacts
Resources available (disks, CPU)

Locking wins, except in corner cases.

If no conflict, then nobody waits and it is a wash

If lots of conflicts, then locking wastes less work

*****
Modern day OLTP:

Main memory problem
No-disk stalls in a Xact
Do not allow user-stalls in a Xact (Aunt Millie will go out for lunch)
--- hence no stalls.

A heavy Xact is 200 record touches – less than 1 Msec.

Why not run Xact to completion – single threaded! No latches, no issues, no nothing.
Basically TS order !!!

Problem: multiprocessor support – we will come back to this.

6.830 / 6.814 Database Systems

Fall 2010