11 - DB Concurrency

Title: CS61 Lecture 11
Author: Charles C. Palmer
Date: May 6, 2015

ACID+S

The A,C, and D principles apply to all databases, while the I and the S only applies to multi-user databases (ok, or really busy single-user DBs).

SLIDE 11–1
Take a look at the Transaction log and see how it really works.

Concurrency control is all about making each user/client feel like their transaction(s) are running alone on the system.

Transactions can get in each other’s way, often without either knowing it!

Review the ways this can happen.

SLIDE 11–2..11–8

It’s quite a challenge to come up with a Scheduler that can determine the best way to interleave transactions without allowing these interferences.

The Scheduler determines the order in which the operations of the transactions are executed, aiming to maintain serializability while supporting concurrency.

The scheduler also tries to optimize the DB’s use of the host system.

Concurrency using Locks

A lock guarantees exclusive use of the things it locks up. Things like:
* a section of memory, a network adapter, the CPU cache, etc.

The size of the thing a lock can protect determines the granularity of that lock.

Locking can take place at the database, table, page, row, or even field level, depending upon the DB being used.
Field level locks are the lowest you can go … concurrent transactions can access the same row IFF they don’t access the same attribute (column). This is individual cell protection, if you think of the DB table as a spreadsheet.

It can be lead to a lot of overhead, but you can’t beat the level of control!
Most DB implementations stop at row-level access.

**Kinds of locks**

You need to be able to TEST the lock, SET the lock, and OPEN the lock.

**Binary Lock Example**

Exclusive locks are just that - exclusive.
Shared locks are used with concurrent transactions are both allowed READ access.

Locks can operate in two modes:

* **exclusive (X) mode.** The protected item can be read and written. `lock_x`
* **shared (S) mode.** The protected items can only be read. `lock_s`.

Either way, locks are requested from and managed by the DBMS.

When requesting a lock, the DBMS either grants the request or not based on what other locks already exist on the item. If it cannot grant the request, the requesting client is **blocked** on the lock. *This is an important point.*

Silberschatz summarizes the valid lock transitions in a table:

<table>
<thead>
<tr>
<th></th>
<th>lock_s</th>
<th>lock_x</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock_s</td>
<td>OK</td>
<td>NOT OK</td>
</tr>
<tr>
<td>lock_x</td>
<td>NOT OK</td>
<td>NOT OK</td>
</tr>
</tbody>
</table>

There can be zero or more locks on a given item, held by 1 or more clients.

A lock request may be granted if the request is compatible with other locks already on the item.

A simple example (also from Silberschatz):
This seems good, but does it enforce serializability?
No, since the update of A or B in between the read of A or B (respectively) could lead to an error.

**Two-Phase locking (2PL)**

This is an orderly way of acquiring and freeing locks.

**SLIDE 10–15**

Two phases

* **growing phase**
  - a transaction acquires all the required locks without unlocking any data.
  - once all locks have been acquired, the transaction is in the locked state.

* **shrinking phase**
  - a transaction releases all its locks, and cannot obtain a new one.
In addition the 2PL protocol has three rules:
1. Two transactions can not have conflicting locks.
2. No unlock operation can precede a lock operation in the same transaction
3. No data are affected until all locks are obtained - until the transaction is in the locked state.

**Deadlocks**

Now, just because you use locks doesn’t mean the resulting schedule is serializable … and locks can sometimes create DEADLOCKS or a deadly embrace.

A related challenge is starvation
* as in T1 waiting for a lock_x(A) while many lock_s(A) requests are granted to a long line of other transactions.

A deadlock occurs when two transactions are waiting for each other to unlock data before proceeding.

**SLIDE 10–16**

We have various techniques for preventing deadlocks.
* avoidance schemes like 2PL work well
* detection schemes that periodically check the DB for deadlocks and aborts any deadlocked transactions and reschedules them
* prevention schemes wherein a transaction requesting a new lock is aborted if there is a chance that a deadlock would occur. All its previously obtained locks are freed and the transaction is rescheduled.

**Time Stamping methods of concurrency control**

A time stamping approach uses system wide, guaranteed unique time stamps to order transactions.
These timestamps have two required properties:
* unique
* monotonic increasing

All the DB operations of a transaction must have the same timestamp. The DB executes transactions in strict timestamp order, ensuring serializability.

The disadvantage is that they introduce more overhead in both storage and computation, since two timestamps are required:
* last time read
* last time written

**wait/die and wound/wait schemes**

**SLIDE 10–17**

In the wait/die scheme, the older transaction (lower timestamp) waits for the younger one to complete and release its locks.
1. If the T requesting the L is older, it will wait for the younger T to complete and release its locks.
2. If the T requesting the L is younger, it will ROLLBACK, giving up its locks, and be rescheduled.

In the wound/wait scheme, the older transaction rolls back the younger transaction and reschedules it.
1. If the T requesting the lock is the older of the two, it will preempt (wound) the younger transaction by rolling it back. The younger T is rescheduled using the same time stamp.
2. If the T requesting the lock is the younger of the two, it will wait until the other T is completed and its locks released.
Other factors affecting recovery

- Deferred-write technique or deferred update
  - Ensures that transaction logs are always written before the data are updated
- Redundant transaction logs
  - Ensure that a physical disk failure will not impair the DBMS’s ability to recover data
- Buffers
  - Temporary storage areas in a primary memory
- Checkpoints
  - Allows DBMS to write all its updated buffers in memory to disk

Checkpointing

Periodically a DB will stop processing transactions and write everything that’s pending to the disk. This is checkpointing, and it’s noted in the log.

Thus, after a failure the DB recovers by
1. find the previous checkpoint in the log
2. assume everything before that was written to disk.
3. Go through and redo everything after that checkpoint.

Example of a recovery

SLIDE 10–18

1. Lecture notes based on texts by Coronel, Widom, Ullman, Jukic, and Silberschatz.