U-Locks

The most frequent reason for deadlocks is conversion from S-locks to X-locks.

Suppose T1 has an S-lock on A and T2 has an S-lock on A and T1 requests an X-lock on A and T2 requests an X-lock on A. You get a deadlock. U-locks solve this problem.

U-locks would be used in update statements where the system must read each record first to see if it satisfies a "where" clause. Would U-locks be used in a transaction where a record is read in an SQL Select statement first, and then later updated in an Update Statement?

U-Locks

Salzberg's U-locks:

Clearing up the discussion on U locks. Do not pay attention to what is in the text on U locks. It does not make sense. Here is the story: A **U-lock** is exactly the same as an S-lock in its compatibility. The only new information to remember is that two U-locks are not compatible with each other. U is compatible with S and IS.

When a U lock wants to convert to an X-lock, it must obtain the necessary new higher granularity locks first. And replace page 416 with the following:

Partial Order on Locks

Partial order on locks:

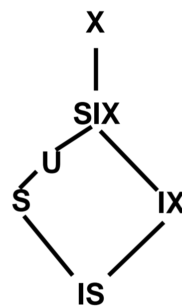
- IS $<$ S (or U)
- IS $<$ IX
- IX $<$ SIX
- S (or U) $<$ SIX
- SIX $<$ X

If $A < B$ then any lock L which conflicts with A conflicts with B.

Compatability Matrix for locks

	IS	IX	S	U	SIX	X
IS	yes	yes	yes	yes	yes	no
IX	yes	yes	no	no	no	no
S	yes	no	yes	yes	no	no
U	yes	no	yes	no	no	no
SIX	yes	no	no	no	no	no
X	no	no	no	no	no	no

partial order of locks



Rules for requesting locks:

- Obtain an IS lock or stronger on all ancestors before obtaining an S or IS lock on descendents in the object hierarchy. Reason: IS prevents X.
- Obtain a IX lock or stronger on all ancestors before obtaining an X, IX or SIX **or U** lock on descendents in the object hierarchy. Reason: IX prevents S (or U). (This will prevent a deadlock if a lower granularity and higher granularity U are both requested.)

S and X locks create implicit locks on objects lower in the object hierarchy. If a file is S-locked, it is the same as if every record in the file were S-locked. If a file is X-locked, it is the same as if every record were X-locked.

DAG locks

A little on DAG locks. What if there are two indexes? (This part uses key-range locks also.)

Example: T1 wants to update E and X-locks [E G) as before. T2 wants to read the record with socsecno = 333-33-3444. But this record has alphakey = E. If T2 does not also locking the alphakey, or T1 does not also lock the socsecno, there is a problem.

DAG locks continued

Answer: To update (or insert or delete), X-lock all paths through indexes and to read, set an S lock on one of the paths.

Problem: What about scanning the table without using an index? (ok if T1 IX-locks the table and the scanner tries to get an S lock on the table.)

Another problem: suppose there are ten indexes and there are 100 records with alphakey = E. Answer: for each record updated, inserted or deleted, all indexes must be key-range locked.

Instead of DAG locks

you can lock a record ID (RID or primary key) when it is read or updated, even if you look it up by a secondary key. This will cause the correct conflicts. This is probably what systems do when they lock records. You still need the table lock or the key-range locks to prevent phantoms.

Locking Heuristics (sec 7.9, text)

- **lock conversion** T1 has an S-lock or a U-lock (compatible with S but not with X) and wants to convert to an X-lock. T1 keeps the S-lock and requests the X-lock. (We will see T1 goes to front of lock-wait queue for a lock conversion request)
- **lock escalation** When a transaction has requested hundreds of record-granularity locks on the same table, most systems will escalate the lock to a table-level lock, dropping all the record-level locks.

- **lock de-escalation** Some systems begin by making all locks table-granularity locks. Then if there is a conflicting request, the lock is de-escalated to record-level locks. Transactions must maintain lists of the locks they would have requested if record-level locks were used.

Nested Transaction Locks (sec 7.10)

Studying how nested transaction locks work explains what nested transactions do. The semantics of nesting can be derived from the mechanics of locking.

Sequential nested transactions are implemented in many systems, such as DB2 and Sybase. Sybase has nested pairs of begin and commit transaction statements. Only the outermost commit commits the transaction. The inner pairs just keep track of the nesting level. This nesting has no effect on locks.

If you have savepoints and rollback to a savepoint, the locks acquired after the savepoint are released (after the updates made since the savepoint are UNDONE).

Parallel Nesting

Parallel nested transactions allow any child to inherit a lock from a parent (but not from a sibling). If a child subtransaction rolls back, the inherited locks are returned to the parent and the acquired locks are dropped. If a child commits, both inherited and acquired locks go to the parent. If a sibling wants a lock another sibling has, it has to wait. Most systems do not support parallel nesting.

Deadlocks

The order transactions request locks cannot be predicted, so deadlocks are always possible. Most systems use timeout to resolve possible deadlocks. (If you have to wait a long time, you are aborted.)

An alternative is to construct a waits-for graph. Transactions which are waiting are nodes and there is an edge from T1 to T2 if T1 is waiting for T2. If there is a cycle in the graph, there is a deadlock. The transaction with the shortest number of log records might be chosen to be aborted.

Deadlock is rare.

Only timeout is used for *distributed* deadlock.

The Convoy Phenomenon

Suppose T1 is running in a high-priority process and T2 has a low priority. When doing process switches, T1 should be chosen more often than T2. But suppose T2 acquires a lock on object X. T1 will have to wait for X.

Suppose T3 and T4 are also high priority and wait for X. When T2 finally gets some time, (because higher priority processes are no longer ready to run because they are waiting) T2 may run, release the lock and get swapped out and then T1 runs, releases the lock and goes to the end of the queue. T1, T3, T4 cycle forever. (The lock in the text example is the end-of-log lock, which must be acquired for each update.) Solution: don't use a queue for hot spots lock waiting, or spin rather than wait, or don't allow lock holders to be pre-empted.

Mixed Multiversion Concurrency

Mixed multiversion concurrency: Read-only transactions read the most recent version valid before their begin time. Read/Write transactions keep locks. and put commit-time timestamps on the records they update (or the TID and system keeps table correlating TID and commit time).

DEC's Rdb used this. Most systems do not.

A new transaction copies a TID vector: TIDs increase monotonically, so all you need is the TID of the earliest non-committed transaction T1 (All TIDs less than T1s are from committed transactions) and the TIDs of any transactions which have already committed and are greater than T1's TID.

Snapshot Isolation (from Berenson et al.)

- Readers can read data from the most recent version committed before the begin time (start timestamp)
- Updating transactions: When T1 is ready to commit, if gets a commit timestamp. Success if no other transaction T2 with commit timestamp in T1's (start, commit) interval wrote data that T1 also wrote (*first committer*) Otherwise T1 is aborted.

No locks. But also it does not guarantee isolation.

Examples for SI

T1 moves $40 from x to y; T2 reads an earlier version

```
H1SI: r1[x0=50] w1[x1=10]
    r2[x0=50] r2 [y0=50] c2
      r1[y0=50] w1[y1=90] c1
```

Here, T2<<< T1 and this is equivalent to a serial history. So we avoid using locks (we keep track of when transactions commit) and we get a consistent view.

```
A5A: r1[x]...w2[x]...w2[y] c2
       r1[y] (c1 or a1)
```

Snapshot Isolation does not allow T1 to read the update T2 made in A5A, so this kind of inconsistent read is prevented. A5A is allowed in READ COMMITTED.

```
H5: r1[x=50]r1[y=50]r2[x=50]r2[y=50]
        w1[y= -40] w2[x= -40] c1 c2
```

H5 is not equivalent to a serial history. T1 and T2 write different items and neither reads the update of the other. But they may both be written to preserve the constraint x + y positive. Interleaved like this, they do not preserve this constraint.

```
A5B: r1[x]...r2[y]...
  w1[y]...w2[x] (c1 and c2 occur)
```

Snapshot Isolation does not prevent A5B, so it allows non-serializable histories.

Oracle???

According to the 1989 Oracle manual cited by Berenson, Oracle read consistency reads the most recent committed version. (It does not depend on the start time of the reading transaction.) This gives unrepeatable reads. It also allows A5A.

Field calls

1. Check predicate: are there enough milk bottles so I can buy one? Unlock milk, but now milk record is in the database cache.
2. Write REDO log record in memory, but not in log cache.
3. At commit time, share lock all predicates and x-lock all transforms.
4. If ok, perform transforms, move log records to log and unlock
5. If not ok, abort (no updates to be made).

Locks held only for a short time (I/Os and Log writes have already been performed).

Escrow Locks

With each pending update, keep range of values: highest and lowest values if pending updates succeed or fail.

Example: current value m, T1 would subtract 7 and T2 would add 8. The range of possible values is (m-7, m+8).

New field calls must check that if they succeeded, the range would not be illegal (less than 0 milk bottles, for example).

"Optimistic Concurrency"

T1 keeps a list of updates to be made. If any other transaction has changed one of the objects since it was read, T1 aborts. (Look at list of updates committed since T1 started.) What Gray says: Predicate is value was the same as the one read.

Adya et al.

Want to allow

`HP1':`

`r1[x=50]w1[x=10]r2[x=10]c1c2`

This is a serializable schedule that is not possible if long locks are kept. T2 reads an uncommitted update. But both transactions commit. What if T1 had aborted after T2 read the update? In an "optimistic" concurrency control system, the system would abort transactions that read dirty data. This is called *cascading abort.*

Adya et al.

this paper proposes isolation level definitions which could be used for optimistic systems as well as for locking systems. It does this by not allowing certain cycles in graphs of committed transactions. It leaves it to the implementor to decide how to prevent such cycles. One way is to abort a transaction if committing it would create a cycle. Remember that aborts are at least 100 times more expensive than lock waits. Lock waits are on the order of 1000 to 2000 instructions. Aborts are on the order of 100,000 or more instructions. None of the major relational systems use optimistic concurrency.

relationship of locking to Adya paper

- long write locks prevent ww cycles in committed transactions and prevent

  `w1[x]w2[x]c1c2`

  which is serializable

- long write locks and short read locks prevent cycles with wr or ww edges and prevent

  `w1[x]r2[x]c1c2`

  which is serializable

- long write locks and long read locks prevent cycles which contain ww, wr and rw edges and prevent

  `r1[x]w2[x]c1c2`

  which is serializable

Notes from Adya Liskov, Gruber and Maheshwari SIGMOD 1995

A scenario where optimistic concurrency might be a good idea, but even here the authors say it does not work well if there are a lot of conflicts (which would produce a lot of aborts).

In this paper, a distributed object oriented database is described.

- primary copy is at server, clients fetch copies of objects
- incoming commit requests are checked against committed transactions they could have conflicted with. If there are problems, requester is aborted.
- after T commits, the server sends invalidation messages to clients other than C that are caching objects installed by T. If any transaction has read this now changed item, it is aborted.

This is just a part of the algorithm. The claim is that if there few conflicts, not setting locks saves enough to offset the occasional aborts.

"Timestamp Concurrency"

T1 reads the timestamp. If the timestamp changes at the second predicate check, it aborts. If not, it puts its own timestamp on the record (even if it makes no update). Locks are used at the second predicate check.

Time domain addressing

Time domain addressing. T1 is assigned a timestamp when it begins. It puts the timestamp on each object it reads. If it reads an object with a timestamp greater than its own, it reads a previous version. If it reads an object with a timestamp less than its own, it writes the same value with its timestamp. If T1 is going to write, T1's timestamp must be higher than the most recent timestamp on the record. If not, T1 aborts. Remember aborts are 100 times more expensive than waits. Also, reads become writes.

Conclusions

If you do any kind of "optimistic" concurrency, you have to keep track of which transactions read and write which items and check things at the end of the transaction.

- If there are very few conflicts, you pay for keeping track, probably about the same as setting unconflicting locks
- If there are many conflicts, you pay by aborting instead of waiting

- Only mixed multiversion concurrency (no locks for read-only transactions) seems to have been implemented in relational systems. Here, you need an extra column on each record for the TID of the last updater. You also keep a few extra copies of recently updated records for transactions whose begin time is before the commit time of the updating TID. But this has benefits for making consistent dumps without locking, and other large read-only operations. This system uses write-locks for read/write transactions.