Degrees of Consistency

1 Introduction to Degree 2 Consistency

We have concentrated our study of concurrency control on a level of consistency called “repeatable reads”. This is the most restrictive form of concurrency, as it places the greatest number of restrictions on transactions. (This is sometimes called *degree 3 isolation* or *repeatable reads (RR)*).

Not all transactions care about repeatable reads. They are willing to forego problems arising with phantoms. We’ll call this level “nonrepeatable reads”. (This is called *degree 2 isolation* also called *cursor stability (CS)*). CS differs from RR in that read locks are discarded once a tuple has been read. Write locks on tuples, like RR, are held to the end of the transaction. Note that CS executions are not truely serializable (because of the nonrepeatability of reads).

Here’s a simple protocol for achieving degree 2 isolation. Suppose the data items in a database are tuples and index records. We will ignore the relationships between tuples and index records. A MGL dag for such a database could be:

$$\begin{tikzcd}
 & \text{database} \arrow[d] & \\
 & \text{volume} \arrow[d] & \\
 & \text{relation} \arrow[dl] \arrow[dr] & \\
\text{tuples} & & \text{index\_records}
\end{tikzcd}$$

Note that this DAG is different than the one used in the index-record-locking protocol we considered before; there is no direct MGL relationship between index_records and tuples. Using the above DAG, let’s see what a reader would lock. Suppose relation R has indices for attributes A-Z. Consider the SQL query:

```
select *
from R                                                        (1)
where A=a and B=b
```

The locks that would be taken would be read locks on each object (tuple or index record) that was touched:

```
IR( database )
IR( volume )
IR( relation )
R( A=a ) // we're processing the query
         // using a single index and we're locking each tuple
foreach qualified tuple t: R(t)
```

Thus, if there are *n* qualified tuples, the number of records that we lock is: 4 + *n*. Note: had we applied our degree 3 multigranularity index-locking protocol (*RR3*), we would have stopped taking locks at `R(A=a)` and would not have taken locks on individual tuples. *Note: the “foreach qualified tuple t” phrase is loaded: we will examine all tuples where (A=a), however we will only lock tuples where (A=a and B=b). (See the last section of these notes for an elaboration of this idea).*

Now consider a delete:

```
delete from R                                                   (2)
where A=a and B=b
```

The write locks that are taken would again be over records that read and written:

```
IW( database )
IW( volume )
IW( relation )
W( A=a )     // we're updating both
W( B=a )    // index records
foreach qualified tuple t: W(t) W(C=t[C]) ... w(Z=t[Z])
```

The total number of locks issued would be: 5 + 25**n*, where *n* is the number of qualified tuples. Note: had we applied our RR3 protocol, we would not have taken locks on individual tuples, and the number of locks would have been 5+ 24**n*.

Now consider an update:

```
update R
set B=b                                                        (3)
where C=c
```

The locks that would be taken are:

```
IW( database )
IW( volume )
IW( relation )
R( C=c ) // we read but don't update
         // the C index record. Note
         // that the IW locks are OK
        // since they are stronger than IR locks
W(B=b)
foreach qualified tuple t: W(t) W(B=t[B])
```

The number of locks taken is: 5 + 2**n*. Note: had we applied our RR3 protocol, we would have placed intent-write locks on all index paths `(A=t[A])`, `(D=t[D])` ... `(Z=t[Z])` and would have place a write lock on `(C=c)`. The net number of locks would have been 5 + 25**n*.

In this scheme, which we will call the *CS2 protocol*, we are taking more locks for selects and deletes, but many fewer for updates, as compared with the way we were doing repeatable reads previously.

Note also why this protocol works: the data items in the database are records (i.e., tuples and index records). It works because a transaction locks every record that it reads/writes; if it only reads an data item, only a read lock is issued. If it updates the data item, a write lock is issued. This is required by 2PL.

2 Phantoms Revisited

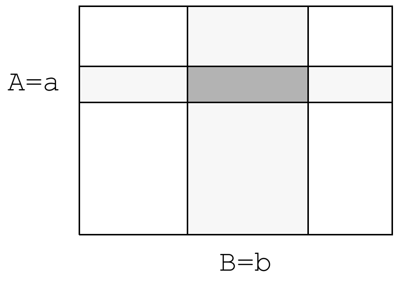
Note that the phantom problem is not solved by our CS2 protocol. Suppose transaction T1 executed the select (1) twice, and transaction T2 executed the update (3) in between T1's selects. Suppose further that there exists a tuple t with the following values:

```
t[A] = a; t[B] = b'; t[C] = c;
```

Tuple t will not be qualified in the first select of T1 (because t[B] = b'). However, t will be updated by T2 and t will become visible to T1's second select. Hence phantoms.

Suppose transactions with degree 2 consistency are running in the same mix as degree 3 consistency. How can we achieve degree 3 consistency using this MGL DAG?

Recall why locking only the `(A=a)` index record in RR3 was sufficient (when we were not locking individual tuples). It worked because *all* records that satisfied predicate `(A=a)` were locked; we ended up read-locking many more records than we had to.



So, to eliminate phantoms, *our repeatable read protocol, which we will call RR2 — to differentiate it from RR3, for readers is to read-lock every tuple that is accessable via the retrieving index*. (If there is no retrieving index, the entire relation is read-locked). By doing so, it follows that transactions `T1` and `T2` below will conflict if either `T1` or `T2` is RR , otherwise they will not conflict:

```
T1: select *                      T2: update R
    from   R                          set    B=b
    where  A=a and B=b                where  C=c and B>b
```

The set of tuples locked by `T1` in CS2 mode is:

```
IR(database)
IR(volume)
IR(relation-R)
R(A=a)
foreach tuple t that satisfies (A=a and B=b), R(t)
```

The set of tuples locked by `T1` in RR2 mode is:

```
IR(database)
IR(volume)
IR(relation-R)
R(A=a)
foreach tuple t that satisfies (A=a), R(t)
```

The set of tuples locked by `T2` in CS2 mode is:

```
IW(database)
IW(volume)
IW(relation-R)
R(C=c)    // assumes C is more selective than A
W(B=b)
foreach tuple t that satisfies (A=a and B>b): W(t), W(B=t{B])
```

The set of tuples locked by `T2` in RR2 mode is:

```
IW(database)
IW(volume)
IW(relation-R)
R(C=c)    // assumes C is more selective than A
W(B=b)
foreach tuple t that satisfies (C=c): W(t), W(B=t{B])
```

Suppose `T1` (in RR2) mode executes, followed by `T2`. Suppose further there is a tuple `t'` where:

```
t'[A]=a; t'[B]>b; t'[C]=c
```

Thus, `T2` will try to update `t'` to be in the write set of `T1`. Note that `T2` will be blocked from doing so. `T1` already has locked all tuples where `(A=a)`.

Our RR2 protocol for writers is similar to readers: *write-lock every tuple that is accessable via the retrieving index*. (If there is no retrieving index, then the entire relation is write-locked). By doing so, all possible records that could satisfy the predicate have been locked. As an example, consider `T1'`:

```
T1': delete from R
     where A=a and B=b
```

Suppose `T1'` executed, and then `T2` executed. `T2` would be blocked from updating `t'` because `t'` would be write-locked by `T1'` (even though `t'` itself was never "deleted" by `T1'`.).

Note our RR2 protocol for readers and writers can be proven using a case analysis. Insertions cannot be made into read/write sets because there would be a direct conflict when updating index records. (Ex: if `(A=a)` is read locked, then no tuple can be inserted into the `(A=a)` set because there is a read or write lock on `(A=a)`). In the case of indirect conflicts (i.e., updates of nonqualified tuples in the read/write set that ultimately become "qualified"), updates are precluded since nonqualified tuples have already been locked.

In this way, degree 2 and degree 3 locking protocols can be intermixed without interference.

3 Implementation Notes

How exactly are locks taken on individual tuples? In particular, if only qualified tuples are to be locked, how is locking done? The answer can be simple[1]: before any tuple is examined, it must be locked. Normally, the lock is in read mode (as exclusive access is not needed for query evaluation). If the tuple is found to satisfy the query, the lock remains. Otherwise, the lock is removed. Note: these are in effect temporary locks that are issued to ensure consistency.

1. Under certain conditions, it may be possible to examine "dirty" data. We'll see this in a homework assignment.

What would happen if a tuple was examined but a lock was not placed on it? Recall the requirements of avoiding cascading aborts and strict executions: never examine dirty data. To look at a tuple and make a decision on whether or not to process it (without first locking the tuple), opens up all the problems of casacading aborts. So, again, even if locks are "temporary", they must be taken to ensure recovery. (Again, see footnote 1).

On another topic, consider the following transactions (assuming A and B are indexed):

```
T1: update R
    set    A = 5
    where  B <= 7


T2: update R
    set    A = 4
    where  B > 7
```

Note that T1 and T2 could actually run concurrently, as they update disjoint sets of tuples *(but not necessarily disjoint sets of index records)*. It is not the case, however, that T1 and T2 can simply skip over each other's tuples. Consider the set of MGL locks that would be taken in each case. Both would issue:

```
IW(database)
W(relation) or RIW(relation)
```

Which means that only T1 or T2 (but not both) could ever execute at one time.

In general, a scanning transaction (one that examines all the tuples of a relation for reading or writing) will either have exclusive access or shared access to a relation. (Exclusive if writing, shared if reading).