IMAGE Locking: Practices and Pitfalls

by Michael A. Casteel
Vice President
Computing Capabilities Corporation
Mountain View, California, USA

The HP3000 literature includes a number of contributions which address the design and implementation of IMAGE data bases and applications using them (see [1] and its bibliography). Unfortunately, with the notable exception of [2], these contributions give short shrift to one of the most critical aspects of data base design and programming: the correct application of IMAGE locking. While this is not such an important consideration for single-user systems or for large batch, "byte-banging" applications, a properly designed and implemented locking strategy is crucial for modern on-line systems.

If a suitable locking strategy is not included in the data base design, on-line performance can suffer due to locking conflicts between users. More importantly, improper design or programming of a locking strategy directly affects the reliability and correctness of the on-line system.

This article describes two common approaches to locking and reveals some hidden flaws in major implementations. As we shall see, it is not only application programmers who may err in their locking strategies: even some commercial transaction processors can damage your data base due to flaws in their application of locking.

Locking-related "bugs" are among the most insidious, since all the programs can appear to work properly but the data base on occasion becomes logically inconsistent. These bugs typically show up when two users try to update the same record at the same time, with the result that only one of the updates is actually recorded in the data base. Both users think their transactions were completed correctly, but at least part of one has been lost.

For a simple example, consider an on-line parts inventory system. Suppose that two users simultaneously issue the same part from inventory. For each user, the program first reads the current inventory record showing, say, a stock on hand of 25, and displays it on the screen. When the first user completes the transaction, the program locks and updates the record, subtracting one part issued from 25, leaving 24. A short time later, as the second user completes the transaction, the program again locks and updates the same record, again subtracts one from 25, leaving 24. As a result, the next physical inventory will show a "shrinkage" of one part due solely to improper locking. While two parts were actually issued, the computer only subtracted one.

Weak Locking

This is an example of what I call "weak" locking, where locks are only placed just prior to the update. In particular, locks are not held during interactions with the user. The relatively short duration of these locks tends to minimize the possibility of contention between users. However, this example raises the more serious question of how to guarantee data base integrity using weak locking. While there is nothing inherently wrong with this approach, the evidence shows that it is very easy to make a mistake (see below). What is required to do it right?

Referring to the simple example above, the program made the fatal assumption that the inventory record being updated had not changed since it was read. The problem here is that IMAGE only guarantees the record will not change AFTER you lock it. In this example, the first user coincidentally changed the record after it was read, but before it was locked by the second user. The probability of such an occurrence depends on the frequency with which different users access the same part, and the length of time between reading the record and locking it. Naturally, this probability is near zero during testing!

For correct operation with weak locking, follow this procedure:

1) Read the record(s) which you will be updating. Show them to the user, think about them, calculate with them, etc. When ready to update, proceed to step 2.

2) Lock all the record(s) which you will be updating.

3) Re-read the record(s). You may use mode 1 (re-read) to minimize overhead, with the risk of an occasional transaction cancellation (see below) due to migrating secondaries on manual masters. This risk won't be high if your data set capacities are adequate.

4) Compare each record you re-read with the contents originally read in step 1. THIS STEP IS ESSENTIAL. You must make sure, one, that the record you originally read is still there, and, two, that it hasn't changed. If the record has been changed or deleted (note that the re-read in step 3 may fail in this case), cancel the transaction. Your thoughts and calculations in step 1 were based on obsolete data.

5) Once that you are assured that a record is still the same, you may update or delete it. If you haven't yet re-read and compared all the records, be prepared to undo (back out) this update/delete if you subsequently find that the transaction must be cancelled.

6) After updating all the records, unlock them.

If you apply these steps to the inventory update example, you will see that they would have prevented the problem. In this example, step 4 would have detected the first user's update and cancelled the second user's update before damaging the data base. But this is not the only pitfall to avoid when using weak locking. The other steps in this procedure also have their purpose. By precisely following the steps, your applications should work correctly. Apparently slight variations can invalidate the whole thing. For example:

1) Not bothering to re-read the records. This gives you no assurance that the original record is still there, much less unchanged. Some other user may have deleted the record, and another user added a completely unrelated record in the same slot where your record used to be.

   This error appears to be made by the TRANSACT processor, as described in [3]. TRANSACT is the transaction processing component of Hewlett Packard's RAPID/3000 package. Although it has been criticized for its use of data base level locking [4], there have been no warnings of this fundamental omission in its locking implementation. Once the problem has been recognized, however, it is possible for the programmer to implement correct locking by coding explicit calls to DBLOCK.

2) Not bothering to compare the record you re-read with the original. Once again, you have no assurance that the record you re-read has anything in common with the original record.

   This step was left out of item 6 under "Database Programming" in [1]. The erroneous inventory count in the above example is one possible consequence of this error. Perhaps more surprising is the possibility of updating or deleting the wrong record. Suppose in our example that the first user actually deleted the record instead of updating it. After the record has been deleted, a third user may add a completely different record in the same slot. When the second user re-reads the record to update it, he reads and updates this new record instead. Now things will really be fouled up!

3) Not keeping the entire original contents for comparison. As this can add up to a lot of memory when many records are to be updated, you might try to compress the record down to a few words, such as a hash total. However, there is a risk with this approach: Because each hash total value equates to a very large number of record values, changes can be made to the record which the hash totaling scheme will fail to detect.

   This method appears to be used in the QUICK processor, as described in [5]. QUICK is the on-line transaction processor in Quasar Systems' POWERHOUSE family. There seems to be no simple way for the programmer to correct this problem within the QUICK environment.

   The degree of risk in this approach depends on how well the hash totaling scheme matches the changes being made. For example, a simple hash totaling scheme might be to take the algebraic sum of the contents of each word in the record. This scheme would be foiled by a typical inventory record update which adds one to Quantity Issued at the same time it subtracts one from Quantity On Hand, with the result that the (algebraic) hash total would not change!

A variation which will work is to keep and compare the original contents of only those items to be updated. But be very careful not to change any other items; another user may have been using or changing them.

4) Only locking, re-reading, updating and unlocking one record at a time can also be a problem. Several records could be updated before discovering one that has been changed or deleted. The transaction must then be cancelled and all the updates undone. But if one of the records that was updated and then unlocked has since been updated by another user, a logically inconsistent data base results. In order to cancel your transaction you must back out the other user's update. If you don't, you haven't completely cancelled your transaction.

   This also appears to be the case in QUICK [5].

As you can see, it is easy to make a mistake in programming weak locking; a lot of people have. While there are variations which work correctly, any mistake places your system at risk of creating a logically inconsistent data base. The four errors listed here might only damage your data base infrequently, but you would naturally prefer not to leave such holes in the logic of your system.

Strong Locking

An alternative to weak locking is what I call "strong" locking, wherein all the records to be updated are locked before they are read, and not unlocked until after they have been updated. This approach has the advantage of being simpler and less error-prone. The original procedure becomes:

1) Lock the record(s) which you will be updating.

2) Read the record(s) which you will be updating. Show them to the user, think about them, calculate with them, etc. When you are ready to update, proceed to step 3.

3) You know that the records you originally read haven't been changed, deleted or moved, since you have kept them locked. Just go ahead and do your updates/deletes, and unlock when you are done.

While this method is clearly easier to implement and maintain, thus promising more reliable software, it is not without its price. The use of strong locking often involves holding locks while interacting with the user, possibly for long periods. This raises the possibility that conflicting locks could delay one user while another finishes a transaction. On the plus side, there is less risk of damage to your data base due to subtle errors in implementation. At the same time, strong locking also reduces overhead by eliminating the need to re-read and compare every record before updating.

The usual practice when using strong locking is to minimize the likelihood of conflicting locks by using entry level locks, with a judicious choice of the data item to be used for locking each data set (see [2] and below). Using entry level locks with strong locking usually works quite well for all but a few complex transactions, provided that the data base and locking strategy have been carefully designed.

Choosing a Locking Strategy

The choice between weak and strong locking methods is not the only decision to be made in regard to locking. There is also the question of the detailed locking strategy, i.e. which levels of locking to use: data base, data set, or entry level?

At one time, data base locking was the only option IMAGE offered. Even with the relatively short duration of weak locks, using data base locking will limit throughput by allowing only one user to be performing updates at any time. Data set locking is better, where the user only locks the data sets which are to be updated. While this allows other users to be updating other data sets, it still delays users who wish to update other records in any of the locked data sets.

The option with the least likelihood of delaying users is entry level locking. In IMAGE, this is the logical equivalent of record locking. Rather than locking specified records, however, IMAGE locks any records within a particular data set which contain certain data item values. For example, you can lock records with PART-NO equal to 312, or DATE-ADDED equal to December 31, 1982. It is also possible to lock using other comparison relations, such as DATE-ADDED greater than December 31, 1982.

With entry level locking, no user need wait for a lock unless another user holds a similar lock, e.g. with the same PART-NO, or an incompatible lock. An entry level lock is incompatible with a data base or data set lock, or with an entry level lock on the same data set using a different data item. If one user is delayed by an incompatible lock, subsequent users will also be delayed, even if their lock requests are compatible with locks already held.

IMAGE manual master data sets often pose such locking conflicts. IMAGE will not add or

delete records in manual master data sets without a data set lock. This restriction conflicts with the user of entry level locking.

For example, one user locks a record while entering changes to a manual master. When the next user tries to add a record to that data set, IMAGE requires that the entire data set be locked. Since the first user has a record locked, IMAGE makes the second user wait until the first one is finished. All subsequent users also have to wait, even if they only want to lock a record. As a result, the entire data set is "locked up" until the first user unlocks the one record. This result is the same when different programs use different items for entry level locks.

If on-line adds and deletes to the manual master are infrequent, this may not be a problem. On the other hand, if your data base design includes manual masters which you expect to be subject to a significant number of on-line adds, deletes and updates, then you may wish to consider making it a detail data set instead.

This shows how important it is to consider locking strategy during the design of the data base. You can minimize locking delays by using entry level locking choosing a single item in each data set to be used as the lock item. (This is usually a search item, although it need not be.) It is very important that all programs use the same data item for locking. If any program (or transaction) uses a different item, the resulting conflicts in locking can be as bad as when using data set level locks.

In a purchasing system, for example, data sets containing order headers, order lines and receiving transactions could all be locked using the Purchase Order Number data item. Since it is unlikely that two users would be dealing with the same order simultaneously, there is little risk of delay due to lock contention.

Now, consider the likely on-line usage: If you are using strong locking, will the program know the data item value in order to place the lock before reading the record? If not, it will have to read the record, place the lock, then read it again, just as in weak locking.

For example, suppose that the receiving transaction in our purchasing system retrieves order lines by part number without knowing the purchase order number. The part number must not be used for locking; this would conflict with locking on purchase order number. Having retrieved the proper order line record, place the lock using the purchase order number from the record. Re-read the record and test to make sure that it's still the proper record. Remember that it may have been changed (or even deleted) since it was read before locking.

Finally, is it likely that more than one user at a time will attempt to access and update records in the same data set with the same value for this lock item? If so, your users could experience delays due to lock contention. You may wish to consider using a different data item for locking.

Combining Strong and Weak Locking

As mentioned earlier, there are some complex transactions which do not lend themselves to strong locking. Perhaps every user is expected to update a single control record in this transaction. Keeping this record locked for any length of time will surely delay other users.

In cases like this you may wish to combine the two types of locking in a given application: use strong locking where possible, for simplicity and reliability; use weak locking on transactions where you need short lock duration. This will work fine, PROVIDED that you:

1) Define and use an appropriate entry level locking strategy. This is important in order to avoid delaying weak locking transactions at long-held strong locks.

2) Correctly program your weak locks.

In the example where many users need to update a single control record, strong locking is fine for most of the transaction, to hold locks over all the other (data) records to be updated. Then, when it's time to update, lock, re-read, update and unlock the control record. But be careful to avoid the following pitfalls:

First of all, placing the control record lock while other records are already locked will require that your programs possess MR (Multiple RIN) capability. Don't take this capability lightly, as the risk of deadlocking your application is very real. In this example, you must make sure that no one who already has the control record locked tries to lock a data record. This will deadlock with a user who has that data record locked while waiting for the control record.

Finally, IMAGE doesn't allow you to unlock just the control record; when you unlock, all your strong locks will be removed as well. This is fine if you have updated all the data records as well as the control record, just before unlocking everything. If you need to keep your other locks in order to continue the transaction, then you must place the control record in another data base, or access it using a second DBOPEN.

Locking and MPE/KSAM Files

These locking principles apply to MPE files as well as to IMAGE data bases. Since the MPE file system does not provide record locking, it is harder to use strong locking without encountering locking conflicts among users. Even weak locking is more complicated, since locking all the files to be updated (in order to avoid weak locking problem (4) above) requires the use of MR capability, with special care to avoid deadlock.

One scheme to support strong locking when using MPE files is to create a dummy IMAGE data set corresponding to each MPE file, and use entry level locking on the dummy data set to effect record locking. This scheme is used in the INSIGHT transaction processor from Computing Capabilities Corporation, as described in [6].

MR capability must still be used in order to lock each file around the actual update (FLOCK / FUPDATE / FUNLOCK) while holding the IMAGE entry locks. With KSAM files, you should re-read by key to reposition in the file before update. Note: This action is advised in the KSAM manual, but I have not yet observed the situations in which it is actually needed. No doubt these situations involve restructuring of the KSAM index.

Conclusion

Although it is essential to the correct operation of multi-user on-line systems, the subject of locking strategy and implementation has been largely ignored in the literature. It is in fact not a simple subject. This article has described two common approaches to locking and pointed out subtle flaws in some major implementations.

While the technique of strong locking has been criticized for potential delays due to locking conflicts, it is easier to guarantee data base integrity with strong locking than with weak locking. When combined with the proper use of entry level locking, strong locking need not delay users except when they attempt to update the same record at the same time. In such an event, a proper implementation of weak locking will not delay a user, but is likely to produce a transaction abort instead.

REFERENCES

[1] David J. Greer, "IMAGE/COBOL: Practical Guidelines", Journal of the HPIUG, Vol. 6 No. 1, Jan/Mar 1983.

[2] Gerald W. Davidson, "IMAGE Locking and Application Design", Journal of the HPGSUG, Vol. IV, No. 1, First Quarter 1981.

[3] Hewlett-Packard Company, "TRANSACT/3000 Reference Manual", Second Edition, December 1982, Appendix C (flowcharts).

[4] M.P. Ashdown, "Developing Large Integrated Systems Using RAPID/3000", Proceedings of the HPIUG Conference, Edinburgh, 1983.

[5] Quasar Systems Ltd, "You are the QUICK Screen Designer", Second Edition, December 1982, pp. 184, 186-7.

[6] Computing Capabilities Corporation, "INSIGHT II Reference Manual", Fifth Edition, December 1982, p. 113.