Record Locking and Updating Efficiencies

Peter Vogel

95 97 2000 2002

Creating an effective Update statement in SQL can make all the difference in what data gets saved in a multi-user environment and what sort of performance you'll get from your application. Peter Vogel discusses record locking and SQL statements.

READER, beware! This article is a long, involved, detailed, and tremendously boring discussion of record locking. However, while this is more about record locking than you ever wanted to know, I'll also discusses the most efficient way to handle updates in a multi-user environment.

The problem

The problem with updates in a multi-user environment occurs when two users access the same record. For my first example, I'll assume a simple record with three fields (A, B, C) with the numbers 1, 2, and 3 in each field:

A	B	C
1	2	3

My first user retrieves the record and changes field A from 1 to 4. While my first user is doing this, a second user retrieves the record and changes field C from 3 to 5. At this point, both users' fingers are reaching for the button that will start the updates. The second user is faster and updates the record in the database:

A	B	C
1	2	5

A millisecond later, the first user presses the update button and changes the same record. The result is:

A	B	C
4	2	3

As you can see, the second user's change to field C has been lost. The change was lost because the SQL statement behind the button looks like this:

```
Update tblExample
   Set A = Me.txtField1,
        B = Me.txtField2,
        C = Me.txtField3
  Where PrimaryKeyField = Me.txtCustId
```

The SQL statement updates every field in the record, regardless of whether or not the field was changed. How can this problem be avoided?

Locking is evil

One solution is to lock records as you retrieve them. Locking schemes basically fall into two patterns: pessimistic and optimistic.

Pessimistic locking assumes that having multiple users updating the same record is a frequent occurrence. To prevent two users making changes at the same time, the application attempts to lock a record against changes when the user retrieves it. When a record is locked and another user tries to access the record, the database system or the application may simply have the user wait (in hopes that the record will be unlocked), return an error to the user, or display a read-only version of the record.

Optimistic locking assumes that it's very unlikely that two or more users will ever access the same record at the same time: Locking the record when it's retrieved is unnecessary. In optimistic locking, a record is locked for only the time that it takes to write out changes to it. Records are locked when being written just to make sure that two users don't try to write the record at the same time and that a user doesn't retrieve a record with some fields updated and other fields not yet updated. When a user finds a record locked in these situations, the system typically waits until the write operation is complete and then accesses the record. Remember, though, that the assumption is that two users will never access the same record at the same time, so this situation should never occur.

In optimistic-locking systems, during the update, the application can check to see if the record has been changed since it was retrieved. If the record has been changed, the application can abandon the update and notify the user (Access works this way, for instance).

Pessimistic locking is certainly the safe choice, but it incurs at least three costs:

1. *Pessimistic locking isn't free.* Attempting to lock a record, locking a record, and notifying other processes that a record is locked all take processing time. Maintaining lock status information also requires some space in memory, on disk, or both (for

example, the Access ldb files). Pessimistic locking typically costs more to perform these tasks than does optimistic locking.

2. *Pessimistic locking hurts performance.* Denying access to records slows down your system, over and above the costs associated with locking. Users have to wait for data to be released from locks when pessimistic locking is in place. Users typically make repeated attempts to access locked records, trying to see if the records have become available (or the application makes repeated attempts on behalf of the user). These attempts also increase the load on the system. Pessimistic-locking systems are less scaleable than optimistic-locking systems.
3. *Pessimistic-locking applications take longer to create.* In a system with pessimistic locking, you have to test to see if your request for a record has been denied because of locking. If it's denied, you have to decide what to do about it. Should you, for instance, wait some appropriate period of time and try again? Or should you just return an error to the user? Or should you, perhaps, write a loop that makes several attempts to retrieve the record before returning an error? You also have to determine when to release your locks. If the user moves from one form to another, without doing an update, should you release your lock on the record? Failure to release locks as early as possible will further degrade your application's scalability. All of this code takes longer to write than the equivalent code in optimistic-locking scenarios.

The differences between optimistic and pessimistic locking are so great that many developers don't worry about optimistic locking. When they discuss "locking," they mean "pessimistic locking."

I've only discussed updating a single record. The problem is aggravated when more than one record is involved. If you're updating several records in a process, should you lock all of them before updating any? In a sales order system, for instance, should you lock the sales order header record and the sales order detail records when a user chooses to modify a sales order? If so, performance and scalability will be degraded even more than when a single record is locked. However, if you don't lock all of the records involved you may end up with a situation where the user successfully updates the header but is then unable to update the detail records.

Two-part solution

The solution to the problem comes in two parts: the SQL used to update the tables and the database design. In your SQL, you must only update the data that changes. In your database design you have to eliminate dependencies between your data.

The SQL solution

Let me return to my example. If the first user's Update statement had looked like the following code, only one field would've been updated:

```
Update tblExample
  Set A = Me.txtField1,
   Where PrimaryKeyField = Me.txtCustId
```

The corresponding update statement for the second user would've looked like this:

```
Update tblExample
  Set C = Me.txtField3,
   Where PrimaryKeyField = Me.txtCustId
```

Since each user has updated only the fields that have changed, the final state of the record would've looked like the following, and no data would've been lost:

A	B	C
4	2	5

What if both users had updated the first field? What if the first user had changed field A to 4, and the second user had changed field A to 5? Since the first user was the last to save, the result would've looked like this:

A	B	C
4	2	3

Well, so what? Yes, the second user's change has been lost. But people get their changes overwritten all the time. If my second user had made the change on Tuesday and the first user had made the change on Wednesday, the second user's change would've been lost, but no one would've complained. Why does it matter if the changes overlap?

There's a problem here but, on most occasions, it can be avoided through proper database design, which is the second part of the solution.

The data design solution

There's a problem with two users updating the same record simultaneously if two fields on the record are related in some way. This table design actually violates the second and third normal forms. In database design, fields in the same record aren't supposed to depend on each other—fields are only to depend on the whole of the primary key. You shouldn't have two fields that depend upon each other in the same record.

Just because it's a rule, though, doesn't mean that it's always followed. For instance, a table of employee data would probably put both the employee's salutation ("Mrs.," "Mr.," "Miss") and the employee's last name on the same record. However, at least some female employees, when they get married, will choose to change both their salutation ("Miss" to "Mrs.") and their last name. Should the salutation be in a separate table? Yes. Will it be? No.

However, if the definition of "related" means that the two fields are always updated together, then there's no problem. And if it's possible for one field to be changed without a corresponding change in the other field, then can you really say that the fields are related? If a female employee gets married, she will either change only her last name or both her last name and salutation. Either one field is updated, or both fields are updated together, so pessimistic locking isn't required. Pessimistic locking would be required only if it were possible for one user to change the last name and another user to change the salutation in such a way that the two related fields are out of whack.

Summing up (so far): Pessimistic locking is required only when there are two fields on the record that are related in some way but that can be changed independently of each other.

My salutation / last name scenario can provide an example of this. At the same time that a user is changing the salutation and last name, another user is also updating the same employee's record. The second user, however, is only changing the salutation so that it reflects that the employee has just had her doctorate granted—her salutation goes from either "Miss" or "Mrs." to "Dr." Either one of two scenarios results:

1. User number one saves first and is followed by user number two. User number two updates the salutation only, so the final version of the record has the correct salutation and last name.
2. User number two saves first and is followed by user number one. User number one updates the salutation and the last name, so the final version of the record doesn't include "Dr."

The question is, "Is this wrong?" Here's the test: If user number one (who's responsible for changing the salutation and last name) had brought up the record and has seen the salutation as "Dr.," would the user have changed the "Dr." to "Mrs.?" If the user isn't allowed to make that change, then the result of my scenario is wrong. To put it another way: If the change in the salutation's state from "Dr." to "Mrs." is prohibited, then this update is wrong.

It's not a coincidence that the second field isn't involved in making the decision. The problem only occurs if some state changes on the field, considered all by itself, aren't allowed. Of course, if this change isn't allowed, then it's a business rule and could be placed in the database as a validation rule to prevent the update from occurring.

Summing up (again): Pessimistic locking is required only when:

- There are two fields on the record that are related in some way.
- Both fields can be changed independently of each other.
- There exists a business rule that excludes some changes in one of the fields from one state to another.

As you can see, the scenarios when pessimistic locking is required are very few. With proper data design and a thorough understanding of the potential scenarios, it's possible to avoid pessimistic locking in most single record scenarios.

Handling multiple records

Unfortunately, most business transactions require more than one record to be updated. For this discussion, I'll return to my sales order example. In this sales order system, sales order information is stored in a sales order header table and a sales order detail table. Again, in theory, there should be no relationships between the two tables. Changes to the sales order detail records should be independent of the sales order header. In practice, this isn't always the case.

As an example, imagine that the customer gets a discount if the total value of the order is over some limit. The discount code would be stored on the sales order header, which must, therefore, be updated as items are added and removed from the sales order. If multiple users are adding and deleting detail records, it's entirely possible that the discount code could be set incorrectly.

The correct design (from the point of view of database design theory) would be to not put the discount code on any record in the sales order. The discount code is a calculated field and should be determined by examining the sales order detail records. However, the impact on reporting and calculating the value of orders would be so great that it's hard to believe that anyone would calculate the discount code every time that it was needed.

One solution is to lock the sales order header and all of its detail records every time a user makes any change to the detail records. However, this isn't necessary—you can use transactions instead.

In this plan, you retrieve all the records that make up the sales order and allow the user to make updates to them. No records are locked. When it comes time to make updates, you begin by starting a transaction. You then retrieve the sales order header and lock it. If the record hasn't been changed since you first retrieved it, you make the updates and release the lock.

The next step is to retrieve the first sales order detail record, lock it, and check to see if any changes have been made since you first retrieved the record. If changes have been made, you roll back your transaction, which will throw away your sales order header changes. If no changes have been made, you make your updates, release the lock, and continue to the next sales order detail record.

Yes, backing out a transaction is expensive, but you need to recognize that you'll only have to do it infrequently. Rather than lock every record on every update just in case there's a problem, you incur the rollback costs only if you actually have a problem.

When a rollback does occur you will, of course, notify your user. Since you still have your copy of the records and can retrieve the new versions from the database, you're in a position to show the user the current version of the record in conjunction with the user's changes. Or you could just throw away your user's changes and make him start over.

Other combinations

So far, I've been discussing multiple users updating the same record. Four other combinations are possible:

- Two users can insert the same record.
- Two users can delete the same record.
- One user can insert a record while another updates it.
- One user can delete a record while another updates it.

Most of these combinations aren't an issue. With the two simultaneous deletes, it (presumably) doesn't matter which user succeeds. With the combinations involving an insert, the record doesn't exist, and so can't be locked.

However, a user who is updating a record that someone else is deleting is certainly possible. The two scenarios are (I'm assuming that the application retrieves the record and checks it before deciding to delete it):

1. The user who's deleting the record finishes before the user doing the update does. The record is deleted, and the subsequent update fails.
2. The user doing the update finishes before the user doing the delete. The update succeeds but, almost immediately, the record is deleted.

In both scenarios, your initial reaction may be "Who cares?" The record is deleted and, while it's interesting that there was an update happening at the same time, it's not a problem. For most cases, that's probably true. However, if the update user is entering information that would've prevented the record from being deleted, then there's a problem.

In the first scenario, it's too bad that the record has been deleted, but the timing just wasn't right. The information that would've saved the record should've been entered earlier. The key point here is that the updating user will be informed that the record has been deleted so the user can take corrective action.

The second case is more serious because the user will successfully update the record and be convinced that the record is now safe from deletion. The updating user won't be informed that the record has been almost immediately deleted.

If this scenario is possible, the delete operation must lock the record before deleting it. Either the lock will fail (because the record is being updated), or the updating user will be prevented from making the change that would've saved the record. If the delete fails because the record couldn't be locked, the delete operation can retrieve the record after the lock is released and recheck it to see if the record can still be deleted. If the update fails, the user is notified that the record that he wanted to update has been deleted.

This has been a long discussion and you're to be commended for finishing it. Here's your reward: Next month, I'll look at the code that will allow you to implement these solutions. As I'll show you, even in the "multiple records with transactions" scenario, you won't need to use pessimistic locking. ▲

Peter Vogel (MBA, MCSD) is a principal in PH&V Information Services. PH&V specializes in system design and development for systems that use Microsoft technologies. Peter has designed, built, and installed intranet and component-based systems for Bayer AG, Exxon, Christie Digital, and the Canadian Imperial Bank of Commerce. He's also the editor of Pinnacle's *Smart Access* and *XML Developer* newsletters and wrote *The Visual Basic Object and Component Handbook* (Prentice Hall, currently being revised for .NET). In addition to teaching for Learning Tree International, Peter wrote its Web application development, ASP.NET, and technical writing courses, along with being technical editor of its COM+ course. His articles have appeared in every major magazine devoted to VB-based development, can be found in the Microsoft Developer Network libraries, and will be included in Visual Studio .NET. Peter also presents at conferences around the world. peter.vogel@phvis.com.