IMPROVING OPTIMISTIC CONCURRENCY CONTROL USING HYBRID TECHNIQUES OF SNAPSHOT ISOLATION AND ROCC

Samidip Basu and Syed M. Rahman
Department of Computer Science
North Dakota State University
258 IACC, Fargo, ND 58105
Email: {Samidip.Basu, Syed.Rahman}@ndsu.edu

Abstract

Concurrent access to shared entities is common for popular database applications and a hotspot for arising problems in the database management system. By far, there are many techniques to overcome the problems; unfortunately, none of them can handle all possible situations. In this paper, we propose a hybrid approach combining two concurrency control mechanisms - Snapshot Isolation (SI) and Read-commit Order Concurrency Control (ROCC). We try to take the best of each method and implement concurrency control with the addition of a simple data structure. Our approach should reduce the stress on developers and on the client-side application.

[*Key words: Visual Studio.NET, ADO.NET, Snapshot Isolation, Read-commit Order Concurrency Control*]

1. Introduction

Concurrency control deals with the issues involved in allowing multiple simultaneous accesses to shared entities. Atomicity, consistency, and isolation of transactions are achieved in the database through concurrency control mechanisms. However, concurrency will probably emerge as a bigger problem as we move towards using more and more database applications. Although much of this paper talks about concurrency control in the .NET perspective, the concepts discussed should be applicable to other environments as well.

Visual Studio.NET is an Integrated Development Environment (IDE) [3], which makes application development very easy. The underlying .NET framework provides the common environment for the execution of applications. Visual Studio.NET allows development of Windows or Web-based or other applications, all of which can communicate with a database; local or remote. The component of the .NET framework that allows applications to connect to a database is called ADO.NET (ActiveX Data Objects) [3]. This technique uses classes that are pre-defined in the .NET framework class library. The basic data structures/objects used in connecting to a database are described through the following diagram (Fig 1):

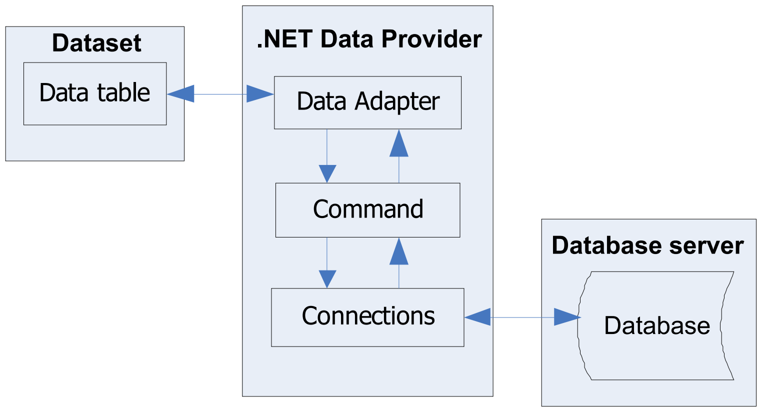


Figure 1: ADO.NET Architecture

The .NET trademark in Database connectivity is the Dataset [3]. The key feature of the ADO.NET Dataset is that it can be used as a self-contained disconnected data store. The users of the applications only deal with the dataset, while it in turn, interacts with the underlying database, when required. This reduces frequent visits to the database and speeds up applications. When the user makes any changes to the data he/she has read from the database, the changes are stored in the dataset until the user commits, at which point, the changes are written back to the database.

Datasets are the biggest strength of ADO.NET [3], as they implement the *disconnected data architecture*, so typical to .NET. The connection object only maintains connection to the database for as long it takes to populate the dataset with data or to write back changes to the database. After that, the connection is severed and the dataset is on its own, acting as a disconnected data store.

2. Concurrency Problems

The vast popularity of windows/web-based database applications demands that they should be scalable and should support multiple users simultaneously. More than one user could access the same data item simultaneously, if no locking is used. This is particularly true for any popular web-application where users access *hotspot* [3] data items most frequently. If multiple users want to make changes to the same data item simultaneously, whom do we allow first? Do we allow over-writes? Hence, *concurrency control* becomes an issue of paramount importance.

Concurrency control (CC) is a vital issue for either windows or web-based database applications. The problem is even more severe when the Internet has been growing so fast that the number of users accessing online databases doubles every year. This calls for higher system throughput [2]. There are many (such as [1, 2, 6]) concurrency control techniques out there, which handle concurrent access and solve deadlock problems. The . NET framework, for example, supports optimistic concurrency control at the database side.

2.1 Concurrency Control Techniques

There are three common ways to handle concurrency control in a database [2, 6]:

- **Pessimistic** – This approach says: “Nobody can cause a concurrency violation with my data if I do not let them get at the data while I have it” [4]. This tactic prevents concurrency in the first place but it limits scalability because it prevents all concurrent access. Pessimistic concurrency generally *locks a row* from the time it is retrieved until the time updates are flushed to the database. Since this requires a connection to remain open during the entire process, pessimistic concurrency *cannot be successfully implemented* in a disconnected model like the ADO.NET dataset, which opens a connection only long enough to populate the dataset [3]; so a database lock cannot be held.

- **Last-in-wins** – This method is pretty straight-forward and easy to implement – whatever data modification was made last, is what gets written to the database [5]. No matter what is changed, the Update statement will overwrite the changes with its own changes. Unlike the pessimistic model, last-in-wins approach allows users to read the data while it is being edited on screen. However, problems can occur when users try to modify the same data at the same time, because users can overwrite each other’s changes without being notified of the collision.

- **Optimistic –** In optimistic concurrency models, a row is *only locked during the update* to the database. [4]. Therefore, the data can be retrieved and updated by other users at any time other than during the actual row update operation. Optimistic concurrency allows the data to be read simultaneously by multiple users and blocks other users less often than its pessimistic counterpart, making it a good choice for ADO.NET. This type of concurrency is also more suitable for the disconnected data architecture of ADO.NET.

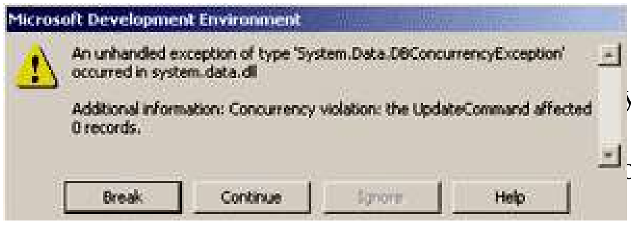
In fact, ADO.NET uses optimistic concurrency mechanism to detect concurrency violations; however, it does not do much with the detection and leaves much of the job to the programmer. In this paper, we presented an approach to share the responsibilities of the programmers.

In .NET, once the dataset has been filled with data, users only interact with the dataset. Users are free to read data, add new data or modify existing data. In ADO.NET, the dataset maintains two copies of data – *original*, the one that was derived from the database and the *current*, which contains the changes/additions made by the user [3]. Whenever the user wants to commit his/her changes, the contents of the dataset are written back to the database. While doing this update, *the dataset checks the data in the database with its original version*. If there is no difference, the underlying data has not been changed from the time it was last retrieved and updates are written to the database. However, if the underlying data does not match with the original version in the dataset, then ADO.NET detects a concurrency violation. Writing changes to the database will over-write the changes made by the previous user. We thus have a concurrency problem at hand.

2.2 Concurrency Problems in .NET

ADO.NET is equipped with classes and methods (*DbConcurrencyException*) to detect concurrency violations when the current database records do not match with the original version of the data in the dataset. However, not much is achieved by the detection. If the programmer does not handle concurrency violation exceptions explicitly, the application simply *blows up* by throwing an error message. This could be disastrous for popular web applications that are being used by hundreds or thousands of concurrent users.

We developed a simple web application prototype that accessed a database. We created two users with two separate sessions and opened two instances of a browser. Both instances read the same data (row); the first one made few changes and committed to the database successfully. Now, the other instance changed the same values (record) in the dataset from the second user's session and tried to commit them back to the database, ADO.NET did not allow the changes and threw up the following (Fig 2) concurrency violation error message.



An unhandled exception of type 'System.Data.DBConcurrencyException' occurred in system.data.dll

Additional information: Concurrency violation: the UpdateCommand affected 0 records.

y error detection

ons in the specified situation:

- The programmer could simply over-write whatever changes have been made to the database. This could work, if one is ready to accept lost updates.
- Otherwise, the programmer could handle the situation by showing a message-box to the user. This way, someone could take permission of the user before over-writing changes to the database, or have the user *refresh* his/her out-of-sync data (by re-loading his dataset with new data) and asking him/her to start editing again.

We believe none of these two are satisfactory. Also, different database management systems handle concurrency control in different ways. This means that applications could behave differently for different back-end systems, making it difficult for the programmer to code for exception handling. A standardized approach is thus needed, that would be consistent at both the application and the database side. This mechanism must also be consistent with the .NET framework architecture.

3. Improvements in Optimistic Concurrency Control

We propose a *combination of two concurrency control mechanisms – Snapshot Isolation* (SI) and *Read-commit Order Concurrency Control* (ROCC), in order to handle concurrency issues in ADO.NET. In our approach, the chances of a concurrency violation will be reduced, as best as feasible, reducing the programmer's responsibility to code for exception handling.

Snapshot Isolation (***SI***) is a multi-version concurrency control algorithm that has widespread industrial use [2]. SI avoids many concurrency errors and it never delays read-only transactions [1]. A transaction T_i executing under SI conceptually reads data from the committed state of the database as of time *start* (T_i), the time when T_i started. Hence, T_i basically reads a *snapshot* of the data as of that time. This snapshot contains all writes of committed transaction and no writes of uncommitted transactions. T_i then holds the results its own writes in local memory store until it commits at time *commit* (T_i). Snapshot Isolation (also called Row Versioning) is optimistic locking, but it is completely transparent to the users and is handled by the database. The database keeps a copy of the original data while one user is changing it, and serves up the original data to anybody who wants to read it in the interim [9].

We consider SI because of its great similarity to the data architecture used by the datasets in ADO.NET. The datasets read the present content of the database, as of the time when the request comes in. Then, it holds all user modifications until the user wants to commit, at which point, changes are flushed to the database. However, there are some inherent weaknesses in SI [1]. It uses a "First Committer Wins" policy that could lead to frequent aborts among transactions, it does not guarantee serializability, it could lead to lost updates and SI also suffers from "Write skew" errors.

Read-commit Order Concurrency Control (ROCC) is a deadlock-free concurrency control method based on optimistic mechanisms [2]. It maintains a single data structure called the Read-Commit queue (RC-queue) that records the access order of transactions. This single structure is all it needs to record all transactions in order of their arrival. It then uses an intervening validation algorithm to ensure execution validations. This algorithm uses a new concept of element conflict instead of operation conflict to reduce validation failures. ROCC only aborts transactions when two or more intervening conflicts occur. This double-checking on conflicts significantly reduces restarts among transactions. Transactions can be controlled to complete successfully at any execution phase by using over-declaration technique or access invariance property [2]. It can also be proved that ROCC produces serializable execution of transactions.

3.1 Explanations of Our Approach

We propose a combination of SI and ROCC in handling much of concurrency control for applications that are developed to run over the .NET framework. To accommodate this change, the back-end database needs to run ROCC-C to maintain transaction serializability. The front-end application will read and write data using rules of SI. The . NET framework will need to maintain an extra data structure (RC-queue) at the application side in the form of a linked list, that will be passed on to the underlying database, one element at a time. However, the load on the dataset will be reduced to a great extent as the versioning (holding on to old value of records) will be done at the database side.

We assume that multiple users are using the application simultaneously. Each user has a dedicated dataset and a data adapter [3]. Whenever a user wants to read/write data from/to a database, it comes in the form of a request to the data adapter. Such requests could be treated as *transactions*. If the request is for one read/write, then it is the only data access request in the transaction. Transactions could also contain multiple data access requests from the user. This gives ADO.NET the flexibility of bundling user actions into a single transaction if the actions constitute an atomic set. Now, when the system (data adapter) receives a request message from the client (user), it will generate the corresponding *element* and post it to the RC-queue. Hence, every transaction (and its sub-actions) is posted as an element(s) on the RC-queue. The format of an RC-queue element is as given below:

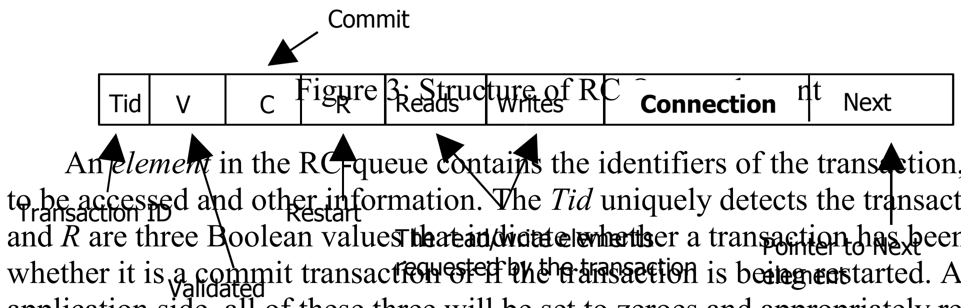


Figure 3: Structure of RC [illegible]nt

An *element* in the RC-queue contains the identifiers of the transaction, the data items to be accessed and other information. The *Tid* uniquely detects the transaction ID. *V*, *C* and *R* are three Boolean values that indicate whether a transaction has been validated, whether it is a commit transaction or if the transaction is being restarted. At the application side, all of these three will be set to zeroes and appropriately reset at the database side. The element also contains a set of *read/write* data entities as requested by the transaction and also, a pointer to the *next* element in the RC-queue. All these are part of a standard RC-queue element as described in [2]. We add one other component – the *connection*. It is a pointer to the connection object of the .NET data provider. This way, each element of the RC-queue will have information as to which connection it is supposed to use, when they are being copied out to the RC-queue of the database. It is particularly important in the case of a popular web application, where the server often uses *connection pooling* to support multiple users [3].

We propose a change in the *content of the read/write elements* for the RC-queue at the application side. The read/write components here store information on *individual attributes* of a row. This is, by far, the most important change in the RC-queue structure. Thus, everything comes down to the level of individual data items. This is possible solely because of the fact that the whole underlying structure of ADO.NET is based on XML. Based on the *Select* command statement, the data adapter knows the structure of the tables that are involved in the SQL query it supports. This is stored in an XML file that the data adapter uses to generate *Update, Insert* and *Delete* statements on the fly. We propose that instead of generating these statements on the fly, the data adapter generate elements of the RC-queue, one for each data item involved in a read/write. The load on the dataset is reduced to a great extent as it does not have to hold on to the original versions of the records that it read. Such versioning is done at the database side by the application of SI. Temporary versions of original records are kept and served up for reads, until the transaction commits when the changes are flushed to the database.

At the application side, corresponding to each read/write operation, a number of RC queue elements will be generated, one for each data item. The generated RC-queue at the application side will then be transported to the RC-queue at the database side, one element at a time using XML. At the database side, the order of elements in the RC-queue represents the real order in which transactions are executed. This is because the database scheduler reads elements directly out of one end of the RC-queue. Because the intervening algorithm is applied for incoming elements at the RC-queue, the real order of elements may be different from their arrival order.

Now, let us consider an example to see how the application of these techniques makes a difference to the way the application behaves. Say, at a given instance, a data row has the following values (say):

ID	Name	Phone	Address	Zip
20	Sam	231-4341	ABC	58102

Transaction T1 changes the address of this record to XYZ. So, now the data row is

ID	Name	Phone	Address	Zip
20	Sam	231-4341	XYZ	58102

Now, T2 is another transaction that began before T1. So, T2 read the original data row. However, T1 commits first. Now, T2 changes the phone number of the record to 231-6729 and commits. The existing ADO.NET will detect a concurrency violation as the underlying database has changed. So, it throws up an exception. If the application is to continue, the programmer will have to either ask T2 to refresh his dataset and start editing again, or overwrite the database. This problem arises because T2 is unaware of the changes T1 is making. Asking T2 to restart editing could be frustrating for the user. Also, there is no guarantee that next time T2 commits; someone else has not already made some other changes. If the programmer chooses to dump the changes to the database, the data row will now be

ID	Name	Phone	Address	Zip
20	Sam	231-6729	ABC	58102

Hence, T1's changes are lost. It is thus evident that the programmer does not have many options.

In our approach, all the data items that the transaction requests to write are actually contained in the Commit element of the RC-queue at the database side, since we are going to use deferred writes to avoid cascading abort [2]. The data manager of the database is only left to perform data accesses in the same order as they appear in the RC-queue. Hence, the element order of in the RC-queue represents the real execution order of data access. Of course, the element order in the RC-queue could be different from the arrival order after we apply the intervening validation algorithm on the elements of the queue.

After all elements have been posted on the RC-queue at the database side, we apply the "*intervening*" algorithm to ensure execution correctness. The algorithm, when applied on the elements of the RC-queue will ensure serializable execution of transactions [2]. The way the application-side RC-queue fits into the overall .NET architecture is shown below:

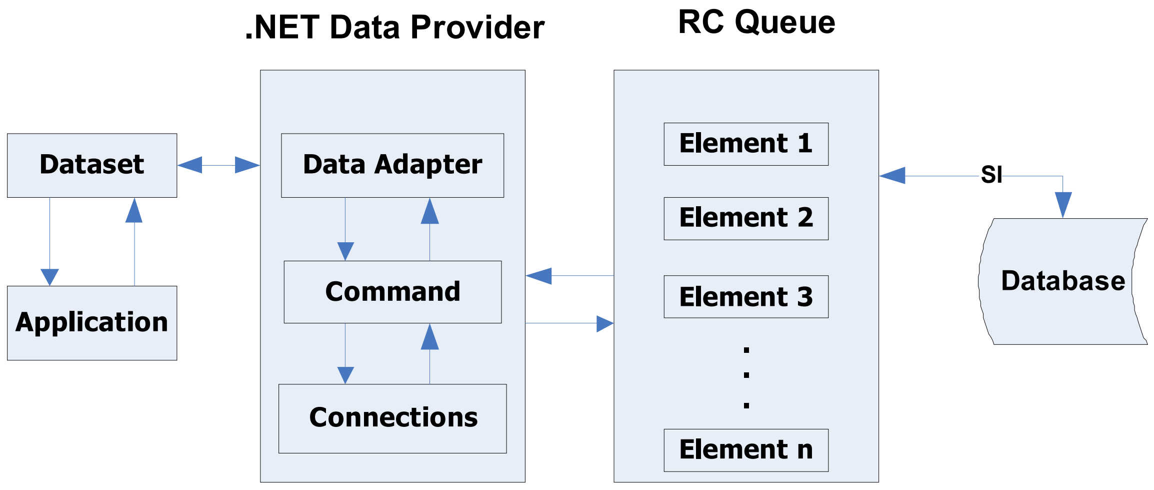


Figure 4: RC Queue in the .NET architecture

At the database side, all transactions from all users broken down into reads/writes of one data item at a time and are posted as elements in the RC-queue. No read elements are ever blocked. Only when a transaction commits, do we apply the intervening validation algorithm [2]. Whenever, a transaction element wants to read data, we do what SI would do – read a snapshot of the system as of that time. This *snapshot* then retraces its path through the objects of the .NET data provider before getting dumped into the dataset. This dataset is what users connect to and read their data from. All changes made by the user are stored in the dataset, until the user commits. At that point, all the writes are bundled into a single transaction element – the *commit element* and posted on the RC-queue. Note that this commit element could be a combination of number of elements based on how many data items are being changed. But, the net effect is that of one commit element, which contains all the writes. Whenever we reach the commit element of a transaction, we apply the "intervening" validation algorithm [2] to check for execution validation.

The algorithm uses a new concept [2] – *element conflict*, to determine whether or not a transaction passes validation. An element conflicts with another element if any of the operations they represent are in conflict (read-write, write-read or write-write). The algorithm essentially checks for element conflicts among existing elements of the RC-queue. This algorithm is the backbone of ROCC and works as follows:

Given the commit element of a transaction, the algorithm starts from the *last commit element* and moves up through the elements of the queue until it finds the *first read element* of the same transaction. Let us call it "first". Now, from "first", we move down until we reach the next element of the same transaction. The elements in between are the "first" element's intervening elements. If "first" does not conflict with the intervening elements, then we combine "first" with the first down-reached-element and call this combined element as "first". Now, we continue to check if the new "first" conflicts with its intervening elements. Such a check process proceeds until it finds that "first" conflicts with its intervening elements or reaches the last commit element of the same transaction and still finds no conflicts. In the latter case, the transaction passes the validation. If a conflict was found, let "second" be the last commit element. We now check if "second" conflicts with its intervening elements in the same way as before, this time though moving up. If both "first" and "second" conflict with their intervening elements, the validation fails.

Such two-prong search for element conflicts should drastically reduce restarts [2]. We only abort a transaction, if it has both upper-sided and lower-sided conflicts. Also, after the application of the algorithm on the existing elements in the queue, it can be proved that the transactions are serializable. When a transaction fails and the client still wants to access the same data (access invariance property), the system will generate a *restart* element [2]. This restart element will contain all the identifiers of data items and the operations that the failed transaction intended to perform. However, this time, the entire restart transaction is posted as a single unit on the RC-queue. Because a single element does not have any intervening elements, this time the transaction is sure to succeed. Hence, even if a transaction fails the first time, it is bound to pass the second time. This way, we never have perennially failed transactions. If the user intends to access a completely new set of data after a validation failure, then the system will abort the failed transaction and treat access requests from the user as if they were a part of a newly arriving transaction.

Essentially, if a user has been making changes to a specific data item in a row of a table, the underlying data could have changed meanwhile. In that case, ADO.NET would generally detect a concurrency violation and throw an exception. With this approach, even if the underlying data has changed, we check if the *particular data item* the user is concerned with has been changed or not. If it does, the user has to refresh his out-of-sync data and start editing again. However, if it has not been changed, then we might be able to save the user's changes along with the changes that have already been made. In that case, we may inform the user that the underlying data has been changed, but for his field of interest. We may show the user the changes that have taken place in the other fields and then ask if he/she still wants to proceed with his changes or not. If yes, we save his changes to the DB.

Let us consider the previous example again. After T1 commits, the data row looks like the following, with the address field changed:

ID	Name	Phone	Address	Zip
20	Sam	231-4341	XYZ	58102

Now, T2 wants to change the phone number item of this data row. Ideally, there should not be any conflict and both the changes can co-exist. This is only possible if we go down to the individual data item level. When T2 commits, his/her writes will be posted as RC queue elements. There will be one write element involving the data item phone number. Under normal situation, ADO.NET would detect a concurrency violation because the underlying row has changed from the time it was last read and throw up an exception. This should not happen with the use of the RC queue. Because the RC queue elements now deal with individual data items, the intervening validation algorithm also works at the data item level. Hence, the commit element involving the write to the data item phone number does not conflict with any writes of T1, because T1 never changed the phone number. Thus, T2 can commit safely and its writes are flushed to the database. This means that both T1 and T2's changes coexist and the data row now looks like:

ID	Name	Phone	Address	Zip
20	Sam	231-6729	XYZ	58102

Of course, if the item phone number had changed prior to T2's commit, then T2 has no option other than to refresh his dataset and start editing again. This however has much less probability than checking to see if anything in the underlying row has changed or not. This will also reduce the frequency of aborted transactions.

Although we show that our approach is feasible, however, we realize that there may be legitimate concerns about allowing updates to data items in a row when the other data items have been changed, as in the case of banking transactions. In such a case, the programmer may catch the concurrency exception and throw up a dialog to the user telling him exactly what has changed in the data row and whether he/she still wants to proceed with his transaction. This could be done using a simple data row object and reading the latest snapshot values into it from the database. If the user says that he/she wants to proceed, his/her transaction is posted as RC queue elements and things proceed as usual. If the user decides not to go with it, the transaction is rolled back and the RC queue elements are deleted. Now, whether the programmer wants to implement this second layer of checking is entirely dependent on him/her and the type of application he/she is building.

We ran a number of simulations to test if our approach can handle on different cases or not. All of our test cases have given satisfactory results; however, the RC queue structure has not been implemented within the .NET architecture. This is because we need to be maintaining a data structure (RC-queue) at the application side itself; something that is beyond our scope. We do hope to see this implemented, while we keep working further on our idea.

4. Conclusion

Concurrency is bound to emerge as a major challenge as build more and more popular scalable database applications. Within the .NET framework, there are methods which can detect concurrency violations, but on detection, it leaves much of the job to the user/developer. At present, ADO.NET uses optimistic concurrency to handle concurrent access. We propose a hybrid concurrency control mechanisms that combines both the SI and the ROCC methods. Using our approach, we believe that we can relieve the programmer stress or user of much work. Concurrency violations will be less probable and database efficiency should increase with the use of latest SI techniques. Oracle has been using SI for a long time; now, SQL Server 2005 also uses SI. We see the trend continuing.

Using the RC-queue data structure, we can have all transactions posted as elements on the queue and then we can apply the "intervening" validation algorithm on the elements. This algorithm is going to ensure serializable execution of transactions. The algorithm checks for both upper and lower-sided conflicts and thus, drastically reduces transaction restarts. Further, the read/write components in the elements are actually the attributes in a row that the transaction is interested in. This should greatly reduce the chance of conflicts among elements. The intervening validation algorithm is easy to implement and has been tested to show good performance [2]. Moreover, the datasets in ADO.NET do not need to maintain multiple copies of same data, which should significantly increase their performance.

References

1. Alan Fekete, Elizabeth O'Neil and Patrick O'Neil: "A read-only transaction anomaly under Snapshot Isolation", ACM SIGMOD Record, Volume 33, Issue 3 (September 2004), Pages: 12 - 14, 2004.
2. Shi, Victor and Perrizo, William: "A new method for Concurrency Control in centralized High Performance Database Systems" ISCA Computers and Their Applications Conference - April, 2002.

3. Anne Prince, Doug Lowe: “VB.NET Database Programming with ADO.NET”, Mike Murach & Associates Inc, 2003.
4. Wayne Plourde: “Handling Concurrency Issues in .NET”, http://www.15seconds.com/issue/030604.htm , Web retrieve on February 19, 2005.
5. Rick Dobson: “Coding for Concurrency in ADO.NET”, http://www.serverintellect.com/pdf/ServerIntellect.pdf , Web retrieve on July 17, 2005.
6. “Introduction to Data Concurrency in ADO.NET”, MSDN Library, http://msdn.microsoft.com/library/default.asp?url=/library/enus/vbcon/html/vbtskperformingoptimisticconcurrencychecking.asp , Web retrieve on February 3, 2005.
7. Nair, Shrijeet ”Transactions and Concurrency Control using ADO.NET”, http://www.c-sharpcorner.com/Code/2002/Aug/TransactionsNConcurr.asp .
8. Benton, Nick; Cardelli, Luca and Fournet, Cédric; “Modern concurrency abstractions for C#”, ACM Transactions on Programming Languages and Systems, Volume 26 , Issue 5, Pages: 769 – 804, NY, USA, September 2004.
9. Ravindra Okade: “SQL Server 2005’s Snapshot Isolation”, http://www.informit.com/articles/article.asp?p=357098&rl=1, Web retrieve on March 5, 2006.