# Demo: Pyrrho DBMS and concurrency

Malcolm Crowe, 8 June 2021

This demo is updated from a [tutorial](https://www.iaria.org/conferences2021/filesDBKDA21/MalcolmCrowe_FritzLaux_Tutorial_ImplementingTrueSerializable_Demo2.pdf) at DBKDA 2021.

We look in detail at Pyrrho’s Commit() method for a transaction, and the detection of conflicts.

At the start of Transaction Commit, there is a validation check, to ensure that the transaction still fits on the current shared state of the database, that is, that we have no conflict with transaction that committed since our transaction started.

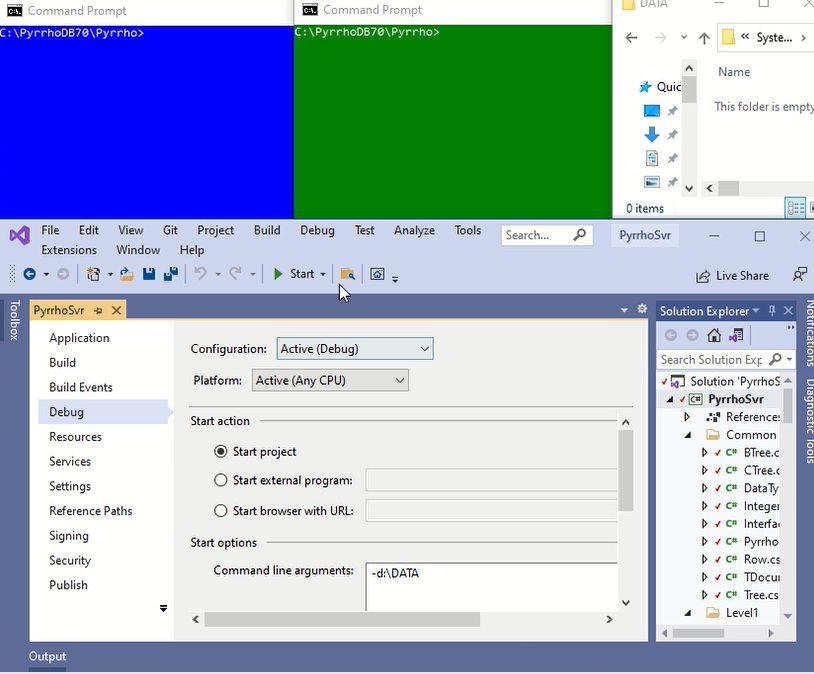
If that is the case, we can relocate all our proposed changes to come after the committed transactions.

The tests for write-write conflicts involve comparing our list of physicals with those of the other transactions.

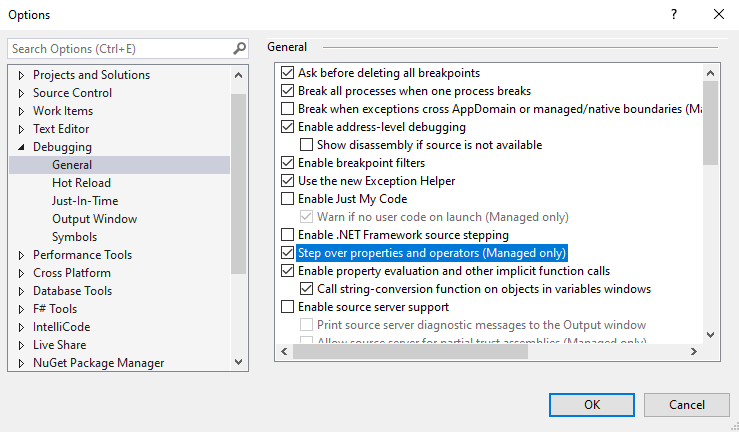
For checking read-write conflicts, we collect “read constraints” when we are making Cursors.

## Part 1: Setting up the demo

Use Visual Studio to open the solution in Pyrrho’s src\Shared folder. In the Solution properties, under Debug>Command Line Arguments, enter -d: followed by a suitable database folder such as \DATA. The folder should not contain a file t10.

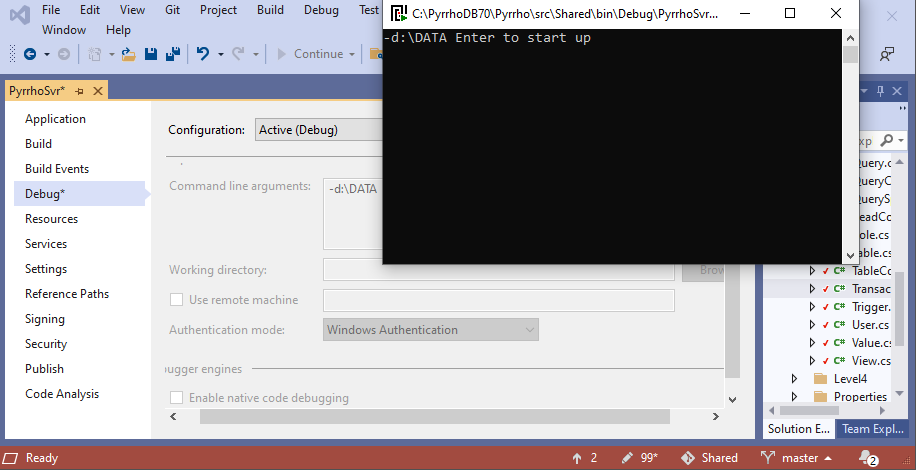


Also ensure that in the Debug>Options.. window,

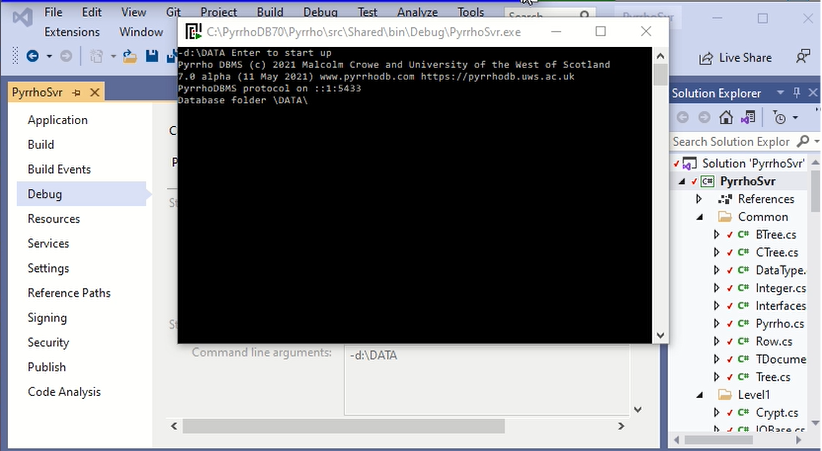


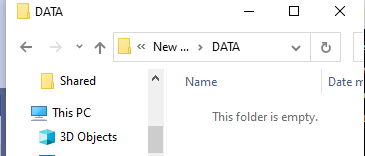
the checkbox Step over properties and operators (managed only) is checked.

Click Start.



In the pop-up command window, hit the enter key to start up the server:

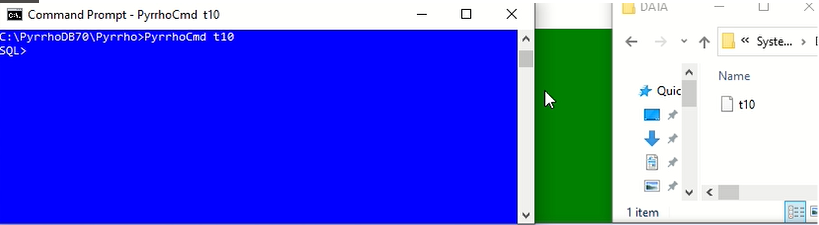


Minimise the pop-up command window.

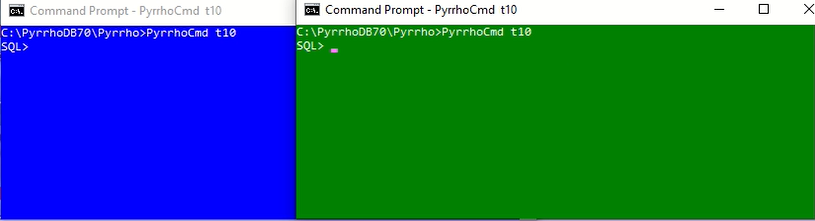
At this point ensure there is no database t10 in the nominated folder.

The command we want to run in both windows is the same: PyrrhoCmd t10. Once we do one of them, the DBMS immediately creates the database as we have seen in demo 1.

**PyrrhoCmd t10**



From now on, the folder window can be hidden (it does not change).



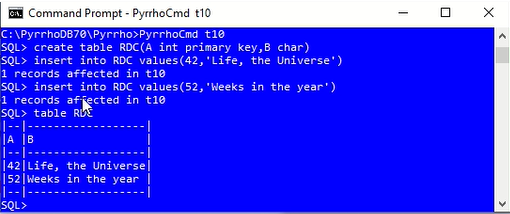
In the blue window, let us create a table (it is called RDC because we will demonstrate read-write conflicts), by giving the following commands:

**create table RDC(A int primary key,B char)**

**insert into RDC values(42,'Life, the Universe')**

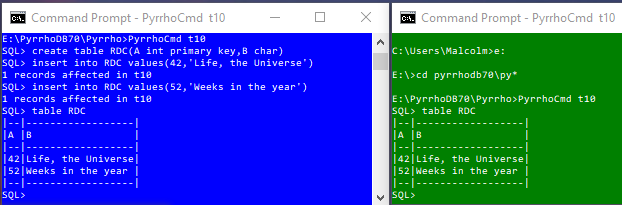
**insert into RDC values(52,'Weeks in the year')**

**table RDC**



That has all been committed, because we are running in auto-commit mode, so the green window will pick it up. It is also running in auto-commit mode, so a new command starts a new transaction, and it will check the current state of the database.

**table RDC**

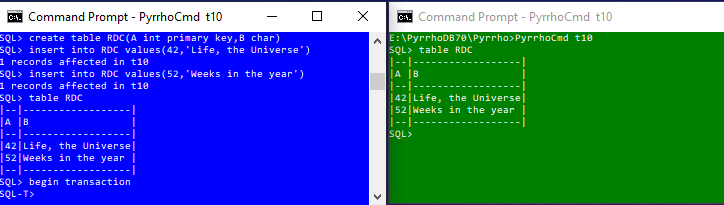


This completes the set up for this demo. We will repeat this part of the demo later.

## Part 2: A Write-write conflict

Now we start an explicit transaction in the blue window.

**begin transaction**

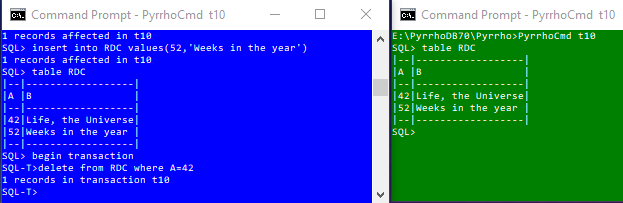


If we just relied on auto-commit mode it is very difficult to synchronise an overlap of transactions. For a demo, it works very well to use explicit transactions.

Notice that in an explicit transaction the prompt changes to SQL-T>. As long as the transaction is running, we will get these prompts. When the transaction is over, either because of commit, or rollback, or because we have done something wrong, it will go back to SQL>.

We will request conflicting changes to table RDC in these two clients. In the blue window, let’s “delete from RDC where A=42”, to delete the first row.

**delete from RDC where A=42**

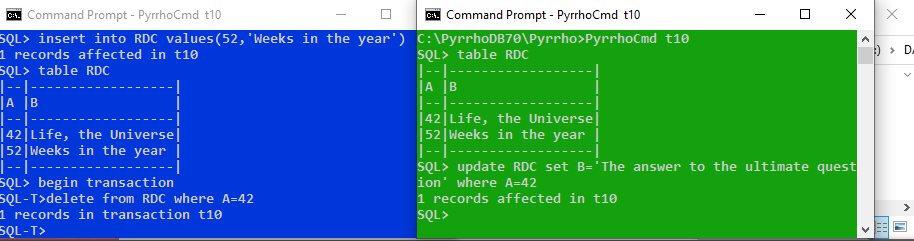


That’s in a transaction, so nothing has been written to disk yet.

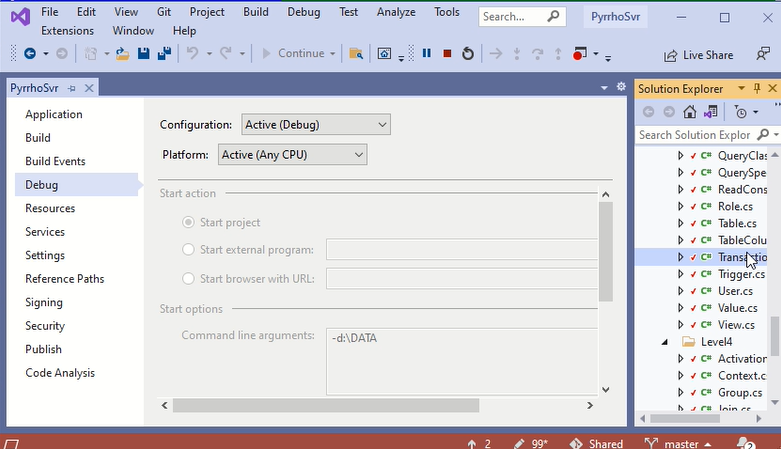
In the green window, we are still in auto-commit mode and we are going to make an update to the same row, which will be committed straightaway to disk.

**update RDC set B='The answer to the ultimate question' where A=42**

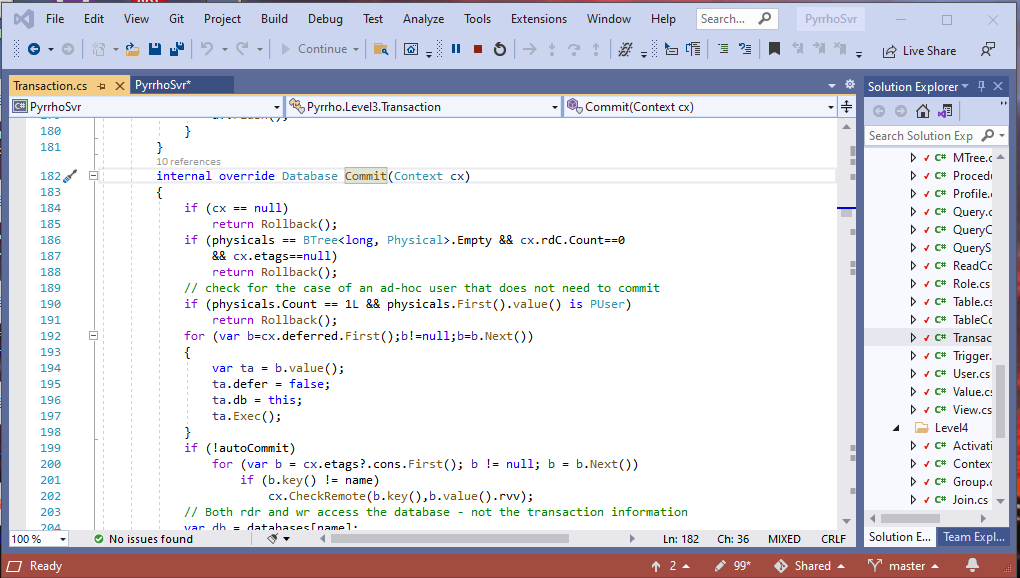
This should make the blue window unable to commit its transaction because of the conflict.



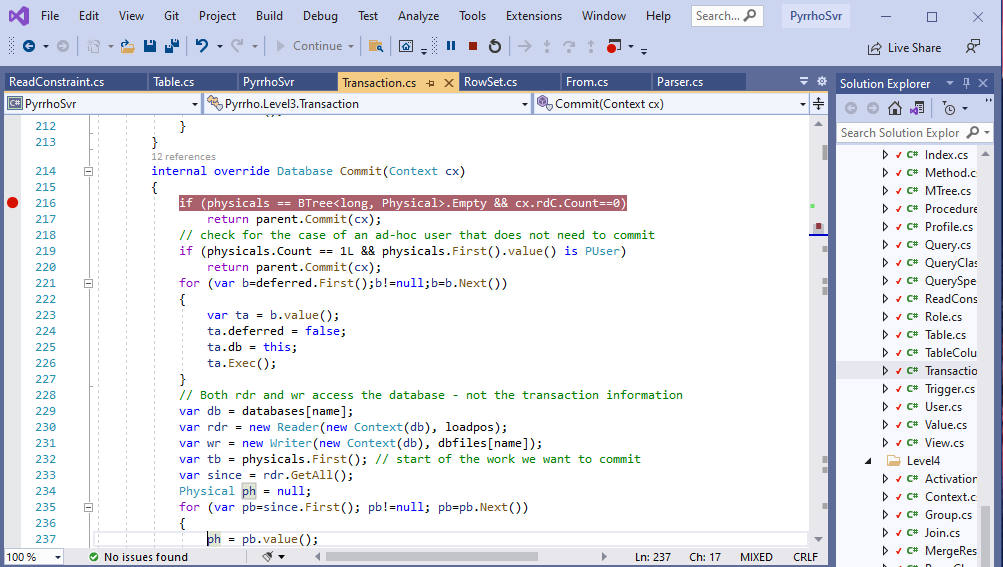
In order to see what happens, let us set a breakpoint in the debugger. In Solution Explorer, in the Level3 folder, locate file Transaction.cs and double-click.



Scroll down to line 182 at the start of the Transaction.Commit() method.



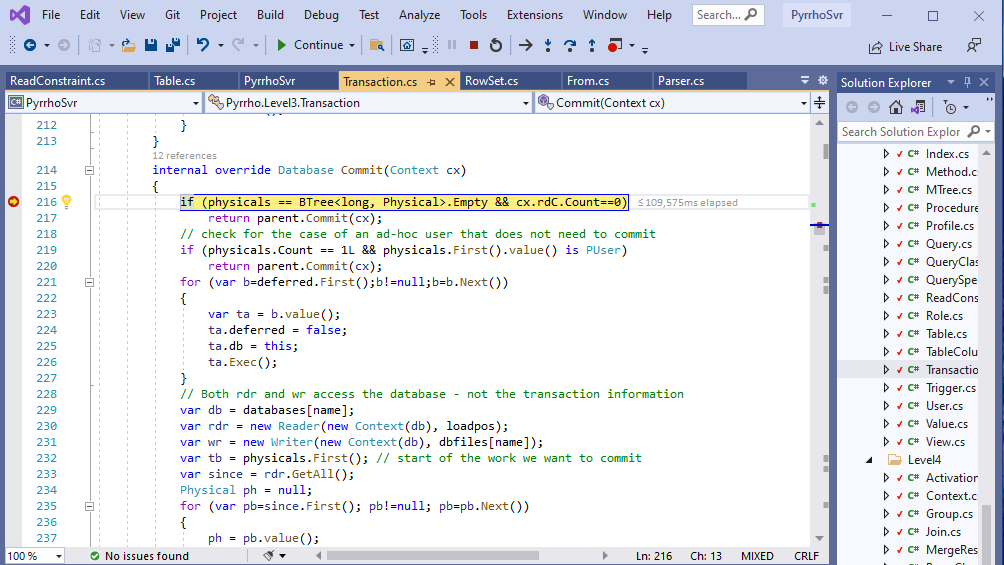
Click the mouse in the left margin to set a break point at line 184 of the Transaction.cs file.



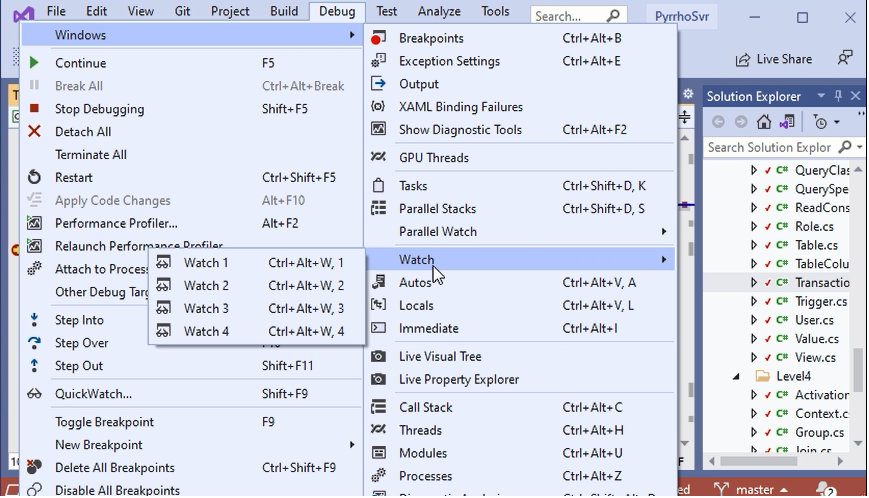
Then when we issue the commit command in the blue window,

**commit**

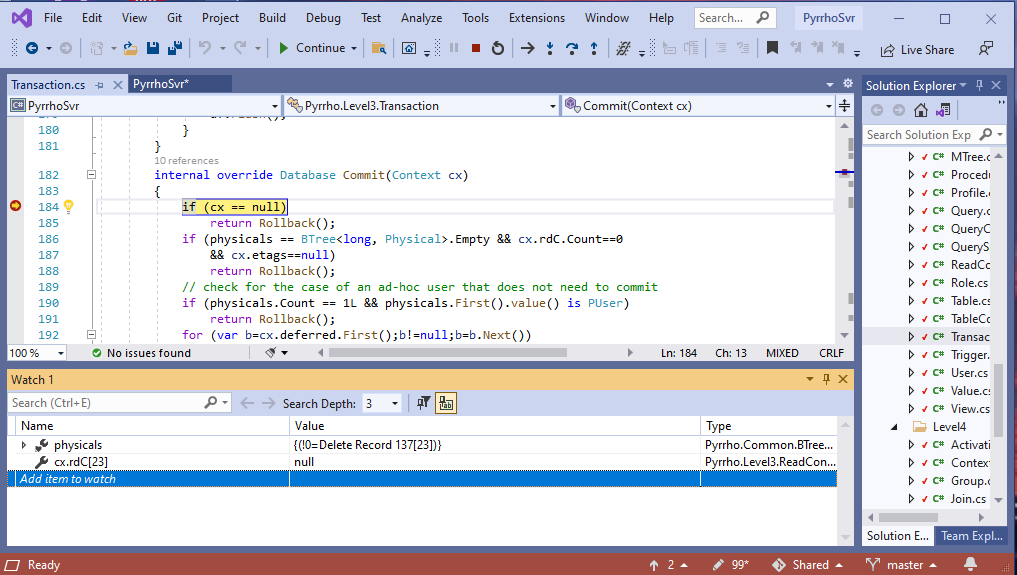
Visual Studio stops at the breakpoint.



Make the Debug>Windows>Watch>Watch 1 window visible,



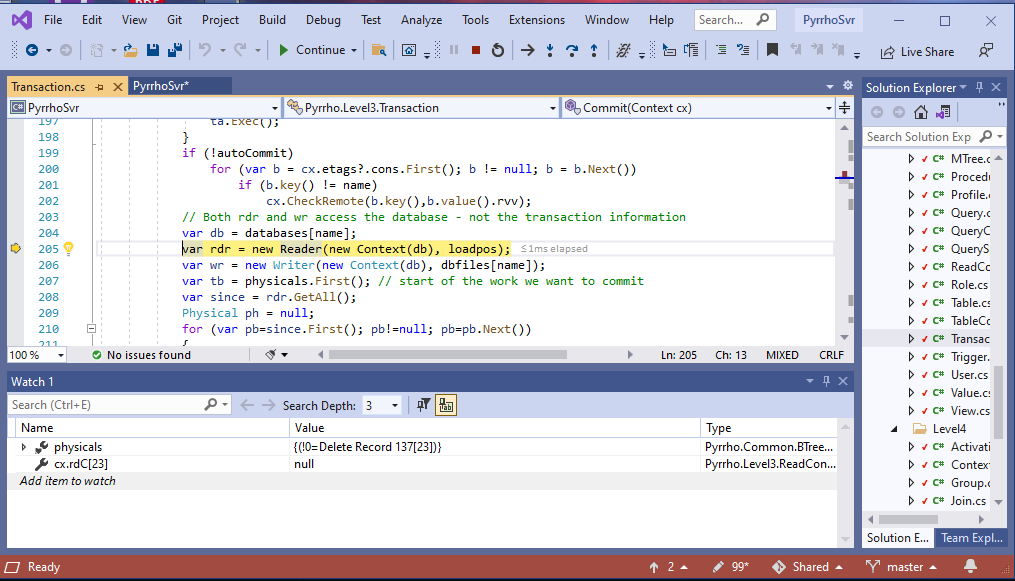
and Dock it below the text window. Use it to examine physicals and cx.rdC[23] :



Let’s look at the physicals. **physicals** is the list of Physical records that the Transaction wishes to commit, and it’s just the single Physical record to delete row 137 in the database which is “Life, the Universe”.

Also look at cx.rdC[23], which is to do with the ReadConstraints for this particular Transaction. It is null, as we have read no rows. (In the next experiment it will be the other way round.)

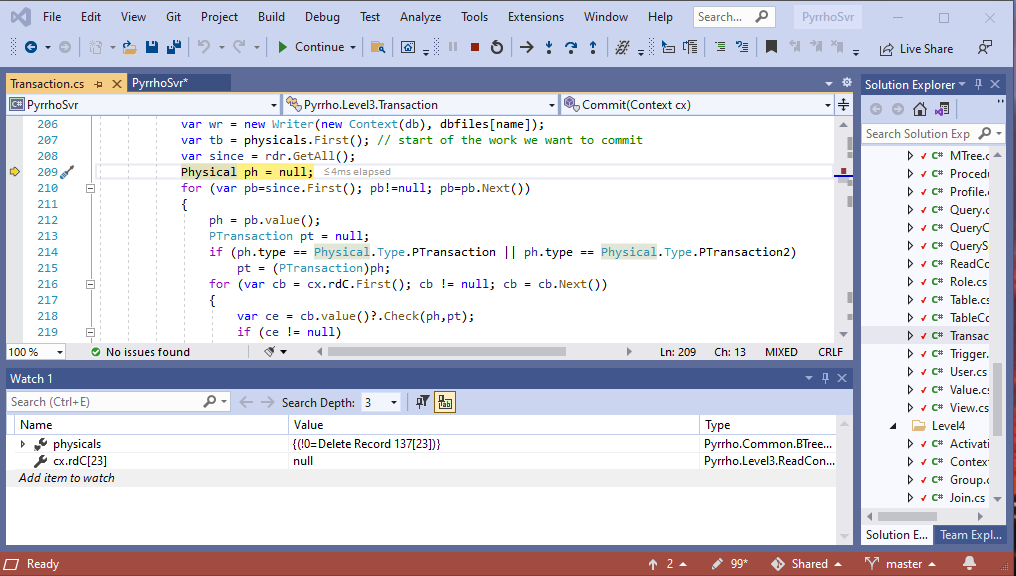
Step Over a few times to get to line 205.



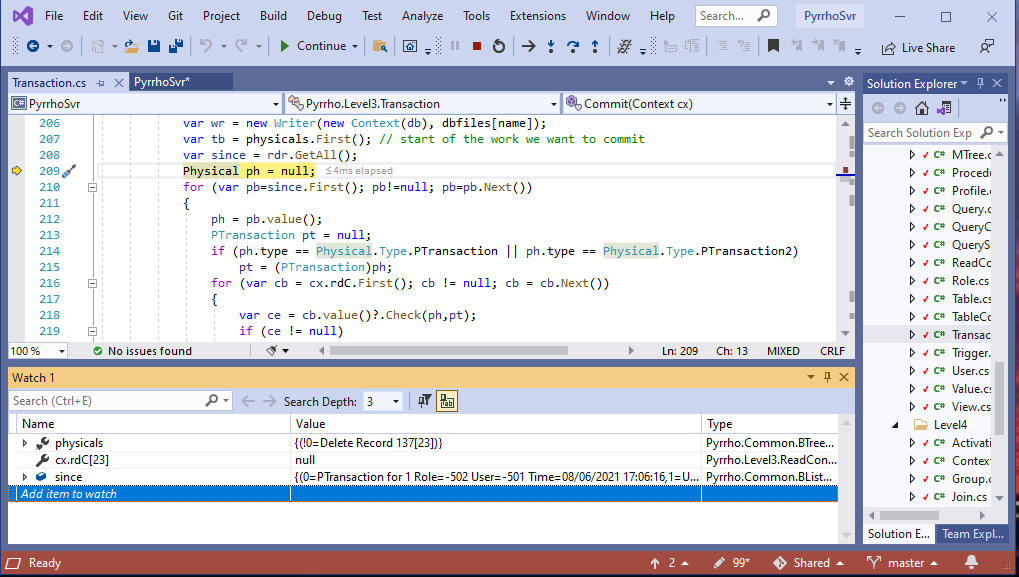
The validation step will use the up-to-date copy of the database, as it was left by the green window. We also open a Reader and a Writer: a Reader to look at this database, specifically at the records that have been committed since the start of our Transaction; and a Writer, where we will prepare the records that we are going to add to the database if our validation succeeds.

These are not shareable and are subject to locking protocols. They also work on the same FileStream. (Transaction Commit() repeats the validation step after locking the FilleStream.)

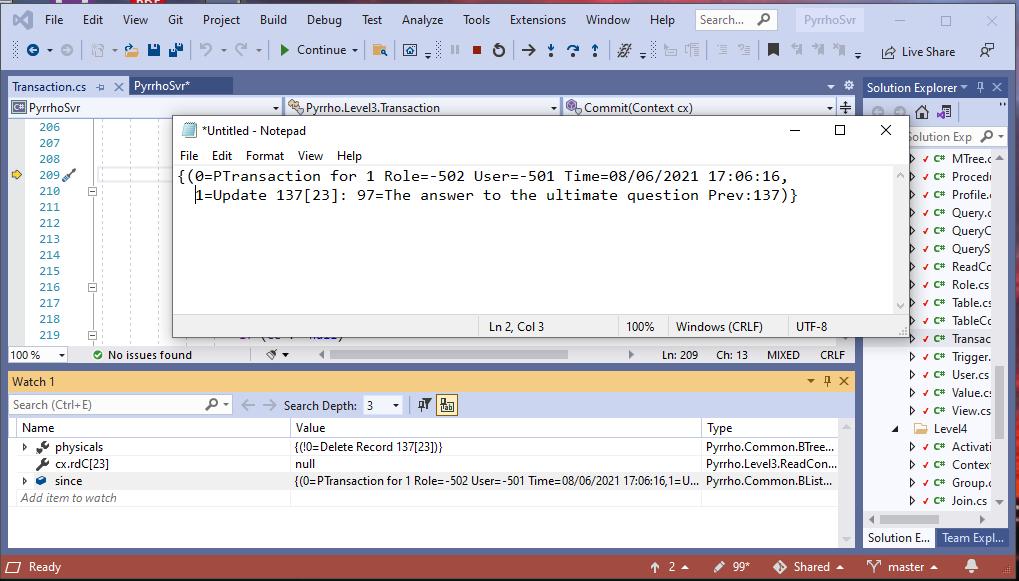
Step Over to line 209, just after a line saying “since=rdr.GetAll();” This gets the records that have been committed to the database since the start of our transaction.



Use the Watch window to examine **since**.

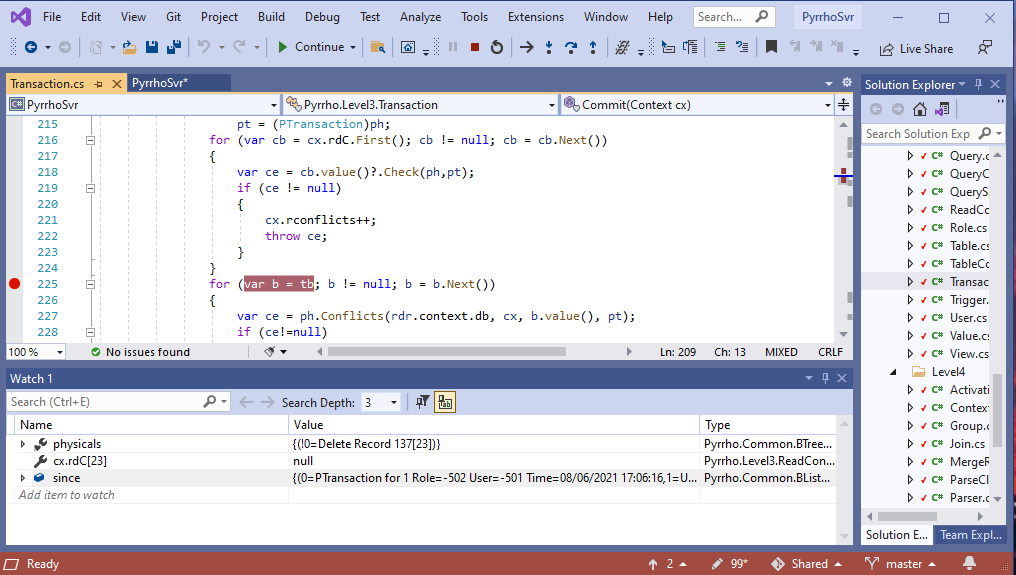


Right-click and copy the value into a Notepad. Add a little extra white space to make it more readable.

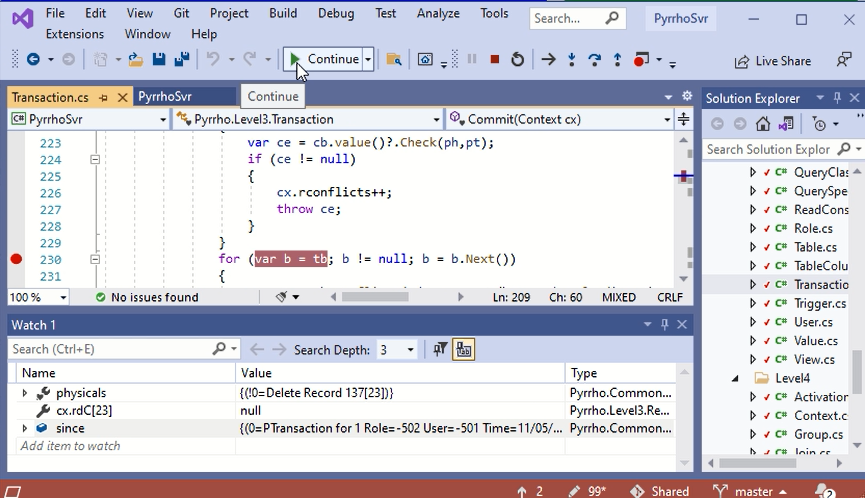


We can see we have got the transaction marker and the update that the green window has made.

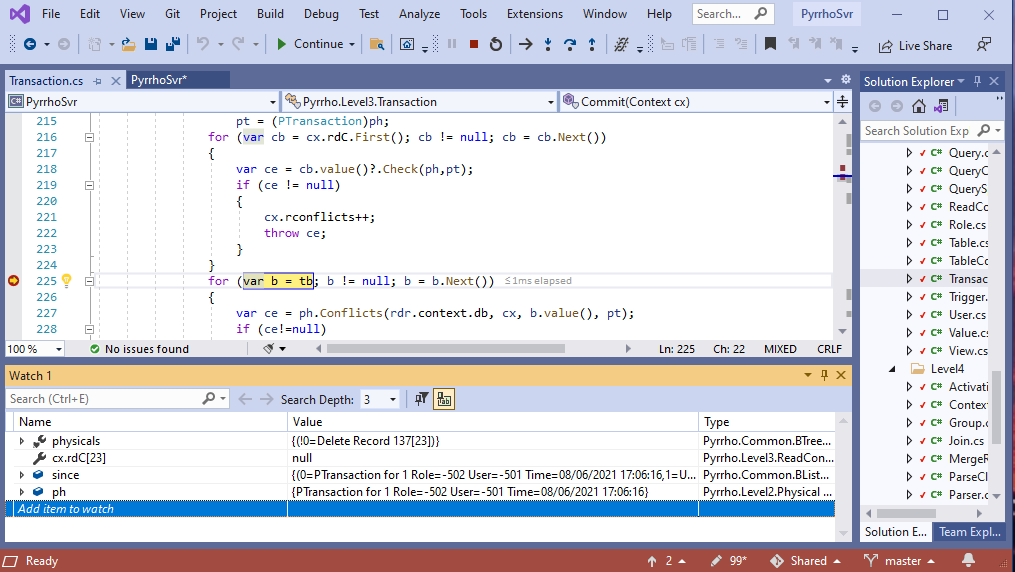
Set a breakpoint at line 225.



Click Continue

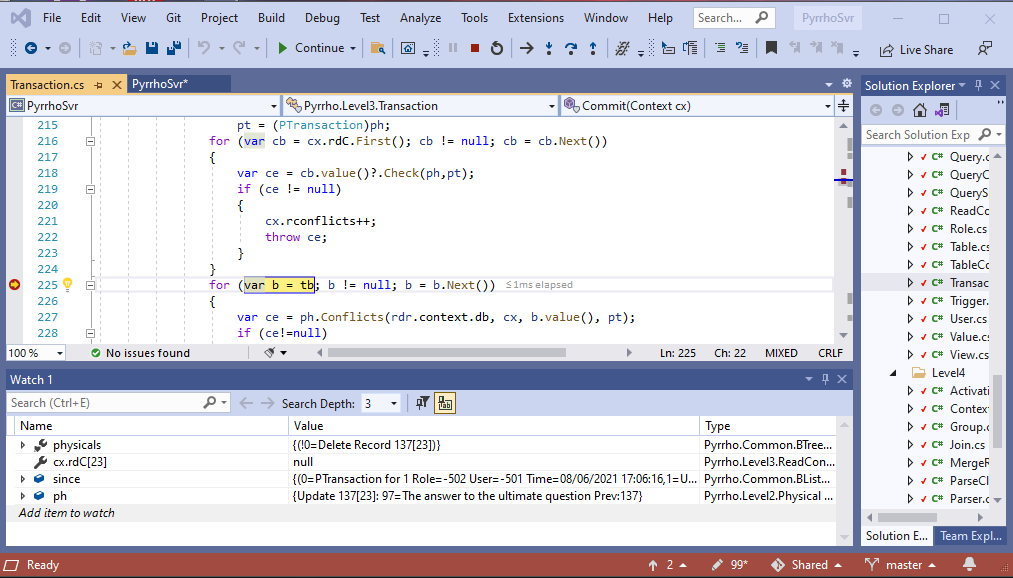


The first time we hit this particular line, ph is the transaction record, as we can see in the Watch window,

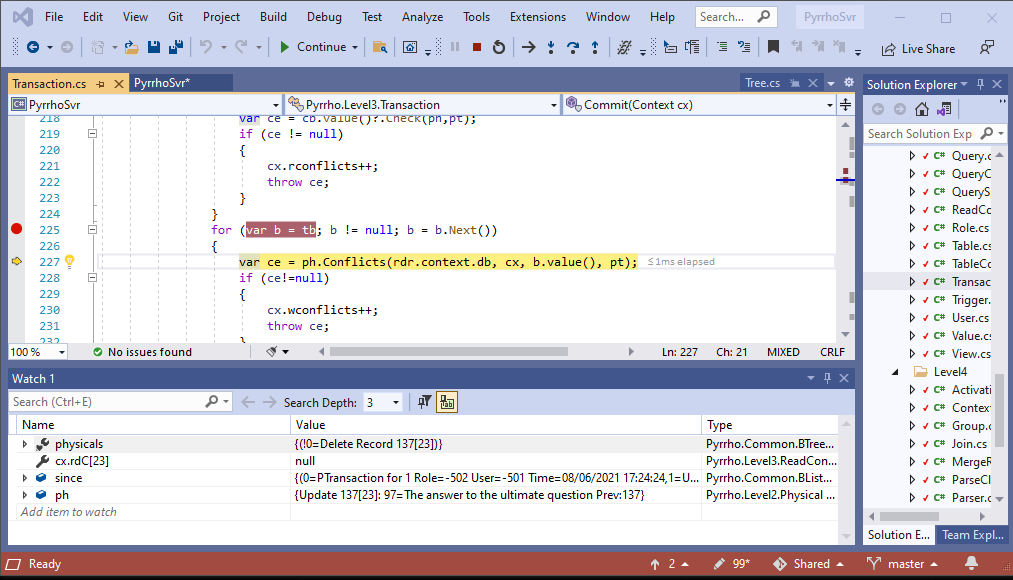


which is not very interesting.

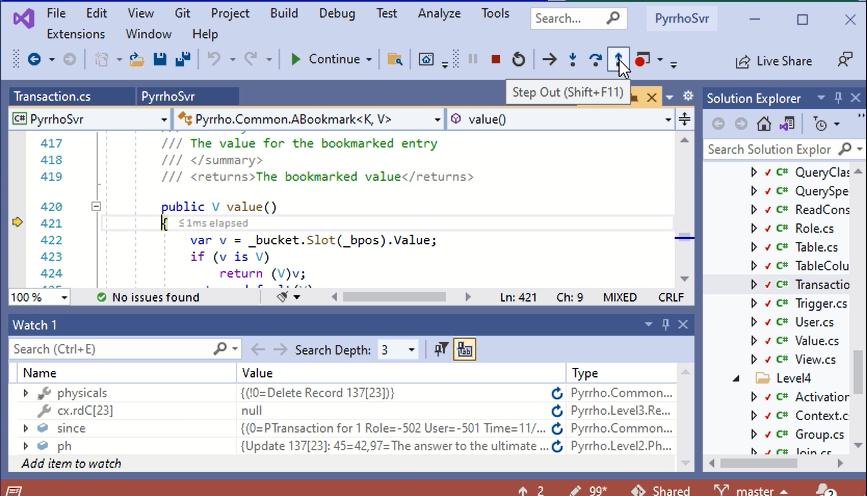
Click Continue



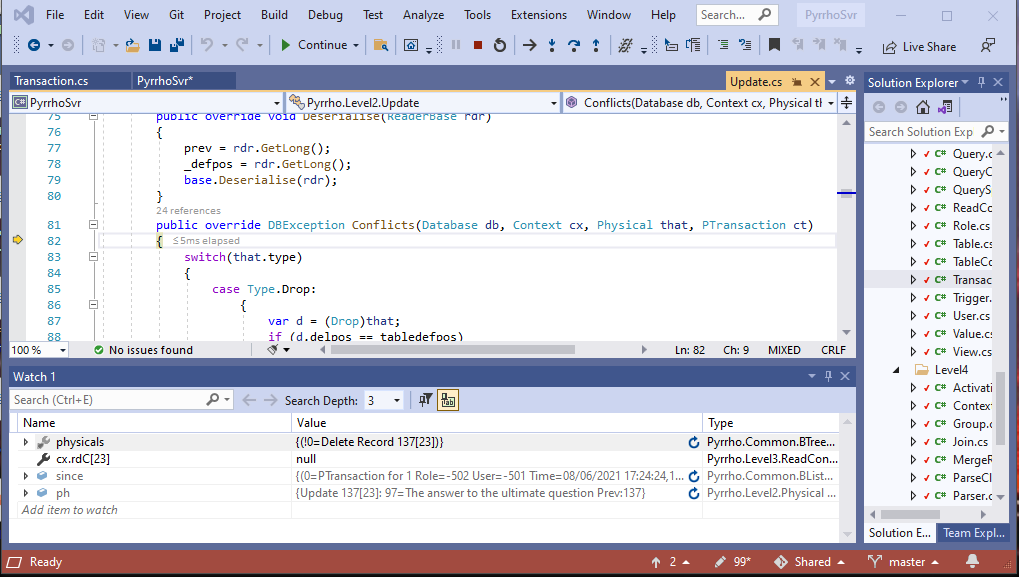
The second time, ph is the Update, and we Step over three times:



Click StepInto: the debugger stops during the evaluation of cb.value(),



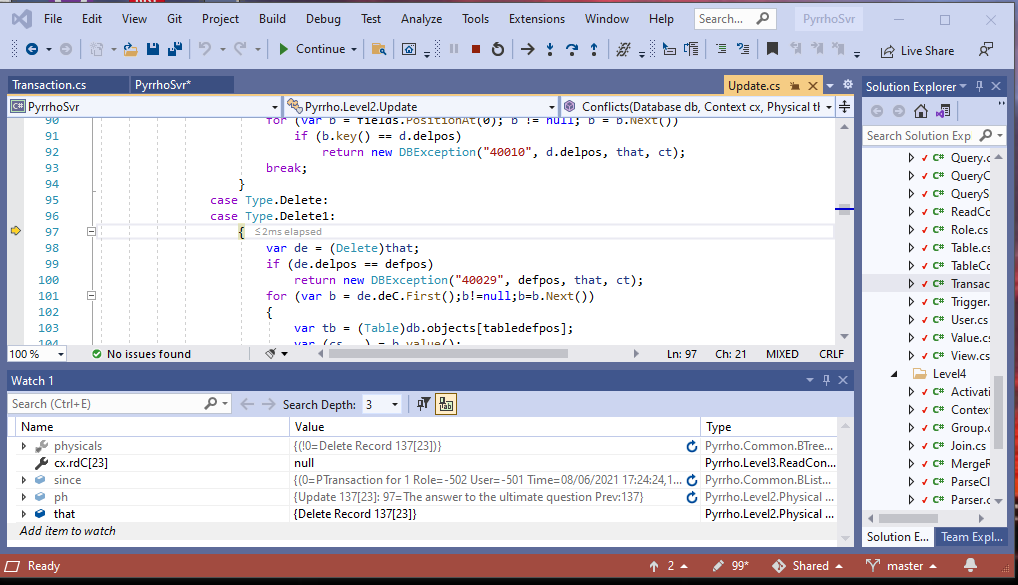
so Step Out, and them immediately Step Into:



In Update.Conflicts() examine that in the Watch window: it is our Delete.

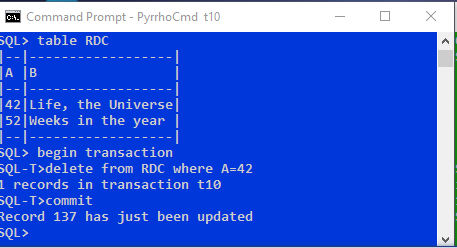


Step Over twice to see the case Delete:



We can see that if the Delete’s delpos matches the Update’s defpos, there will be a conflict. (We are trying to delete a record that another transaction has updated.)

In this case they are both 137 and will conflict, so let’s just click Continue.



We see that we get a complaint back in the blue window that record 137 has just been updated. The transaction has been rolled back, as we see from the prompt.

That completes the first experiment that we want to do in this demonstration.

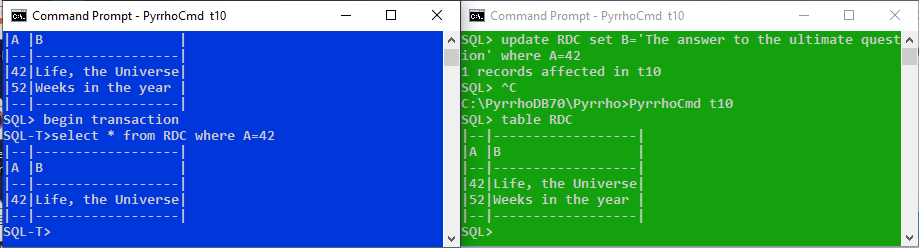
## Part 3: A Read-Write Transaction

Stop the server, delete the file t10 in \DATA, remove the breakpoints, and repeat the steps in part 1.

In the blue window

**begin transaction**

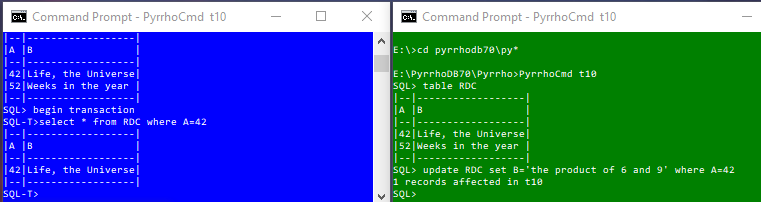
**select \* from RDC where A=42**



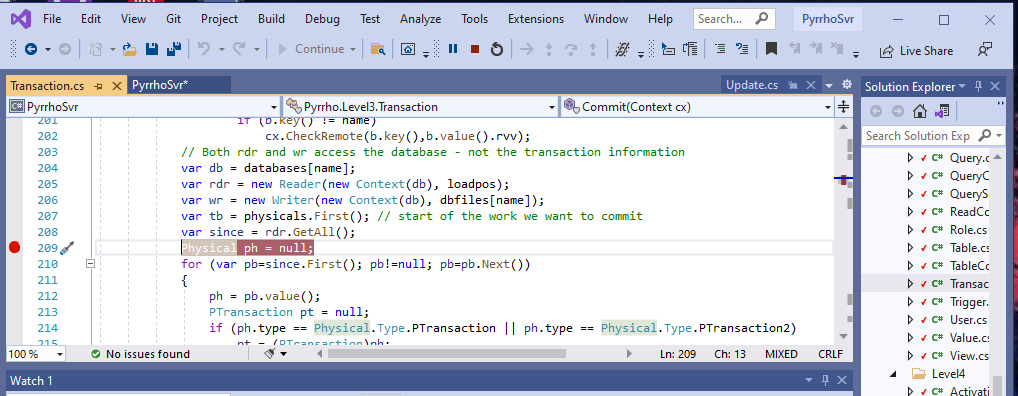
Just as before, the green window makes an update to the same row,

**update RDC set B='The product of 6 and 9' where A=42**

which is auto-committed. For the reasons explained earlier, we expect that the blue window will now be unable to commit.

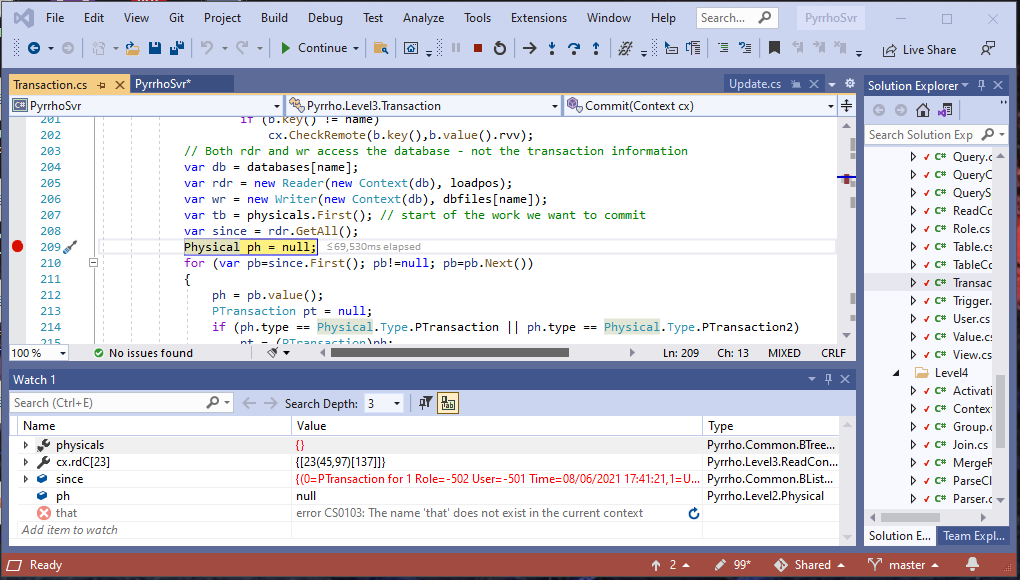


Again we stop in Transaction.Commit to see what happens. Place a breakpoint at line 209 of Transaction.cs:

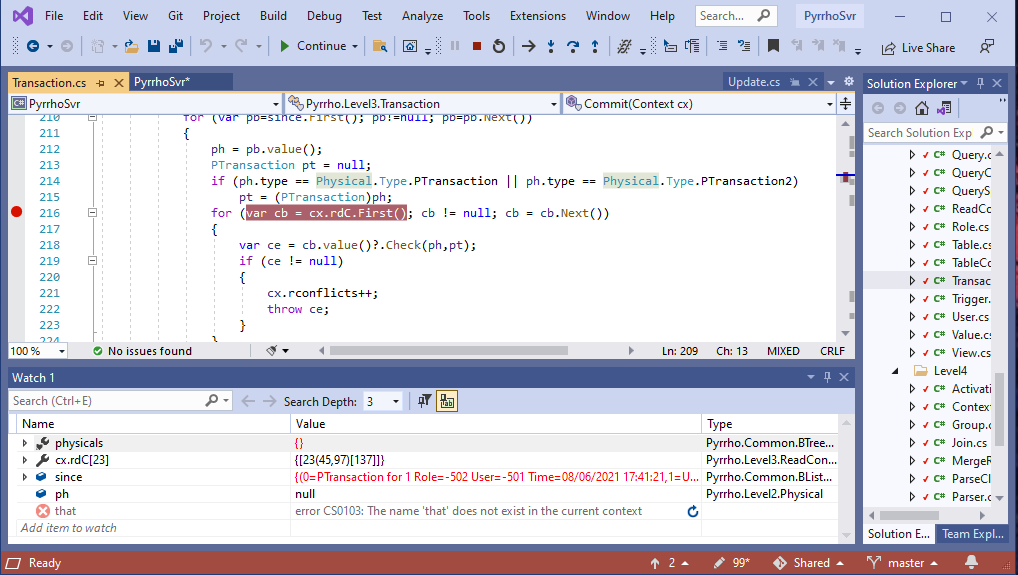


and in the blue window

**commit**



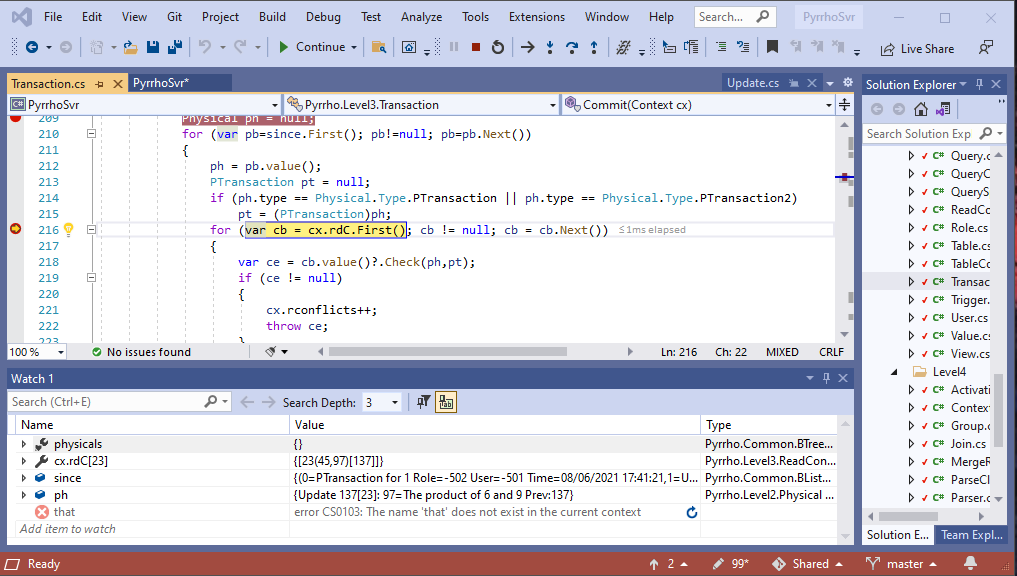
This time, we see that physicals is empty, and instead we have a readConstraint. Set a breakpoint at line 216, to see the effect of this rdC:



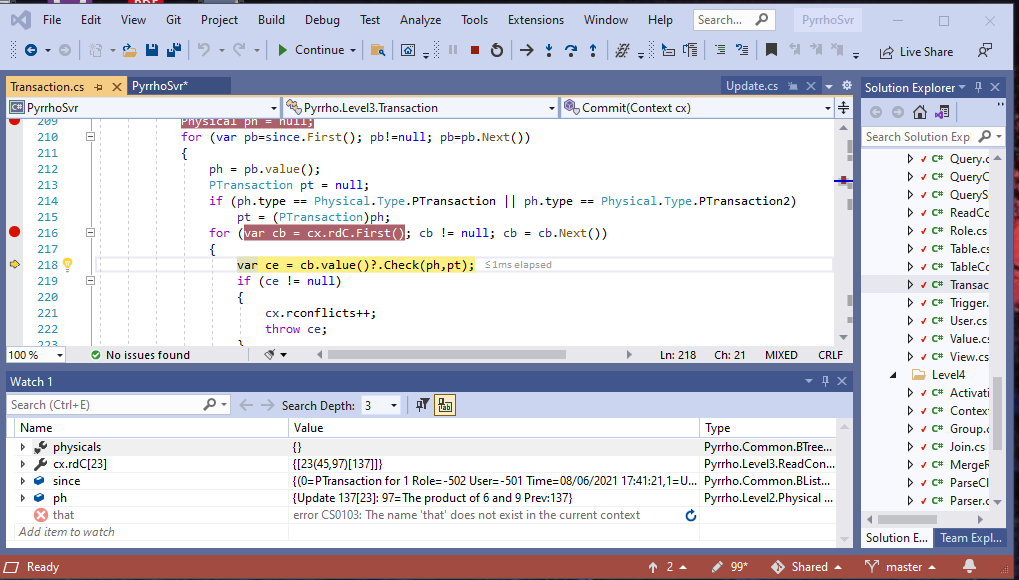
and click Continue, to reach this point.



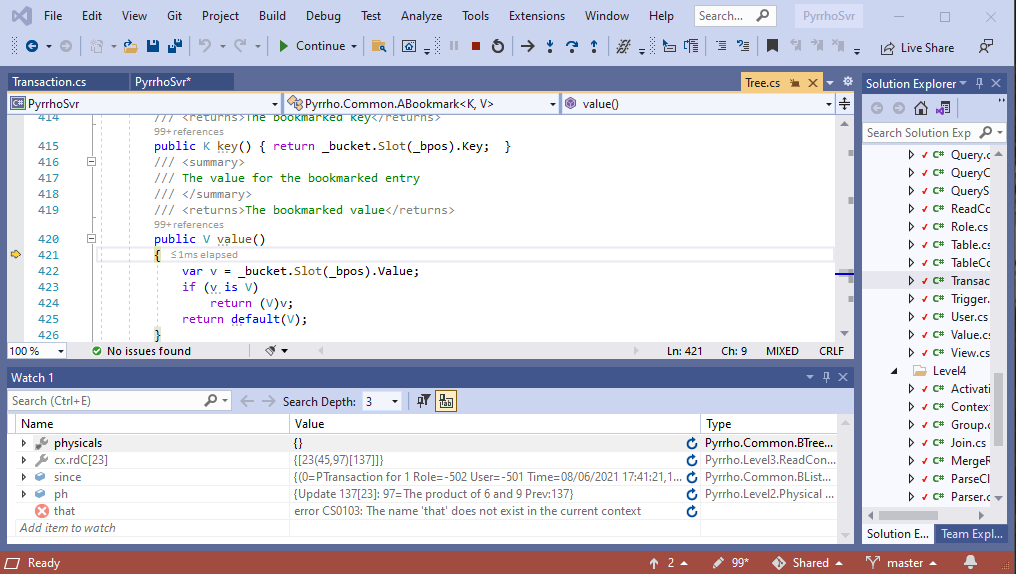
Again, we see that ph is the PTransaction, which is not interesting. Click Continue again:



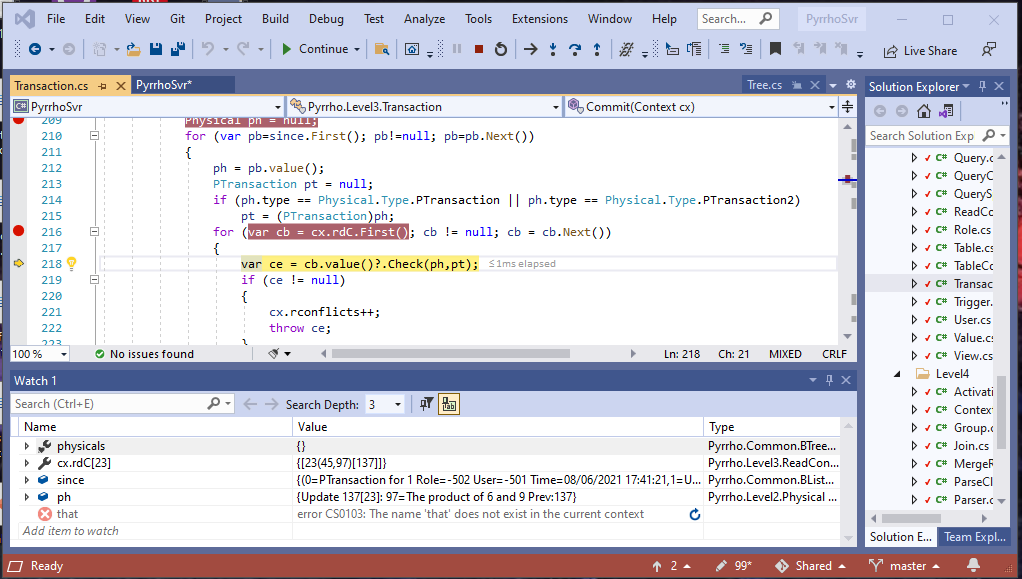
This time, ph is the Update. We want to Step into the Check method here. So Step Over to get to line 218.



At line 218, Step Into, b.value():



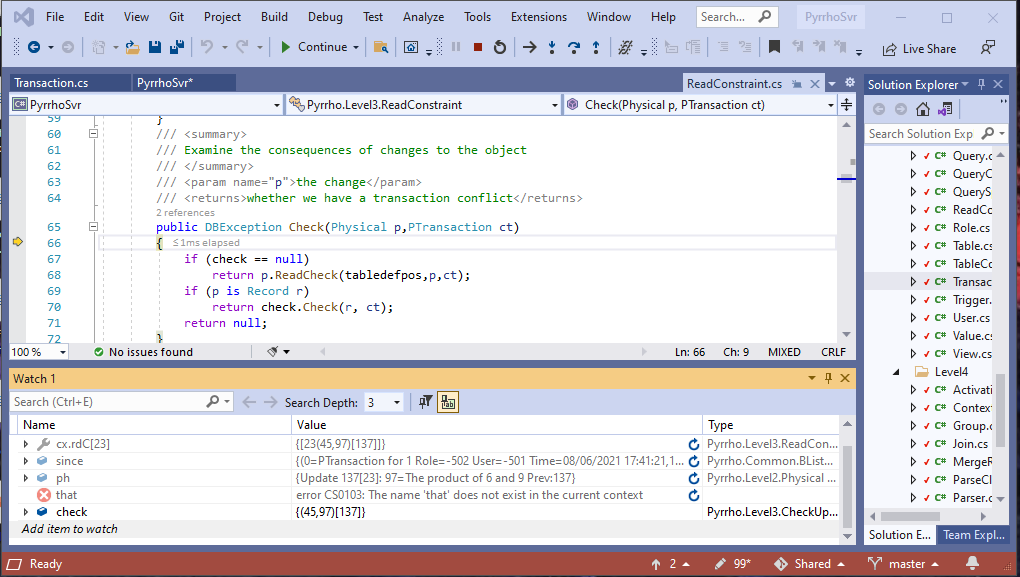
Step Out, (apparently back to where we were):



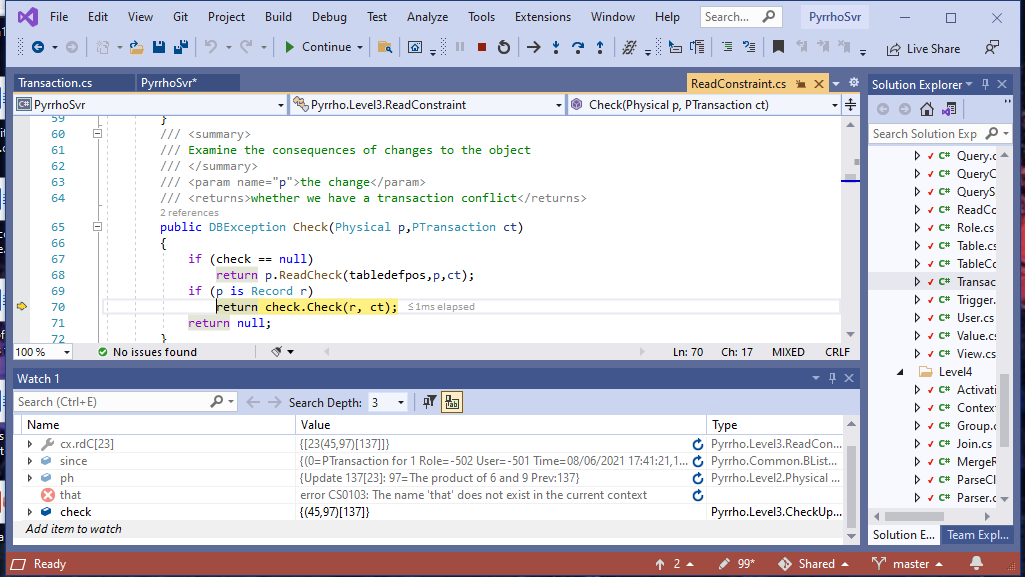
Step Into Check():



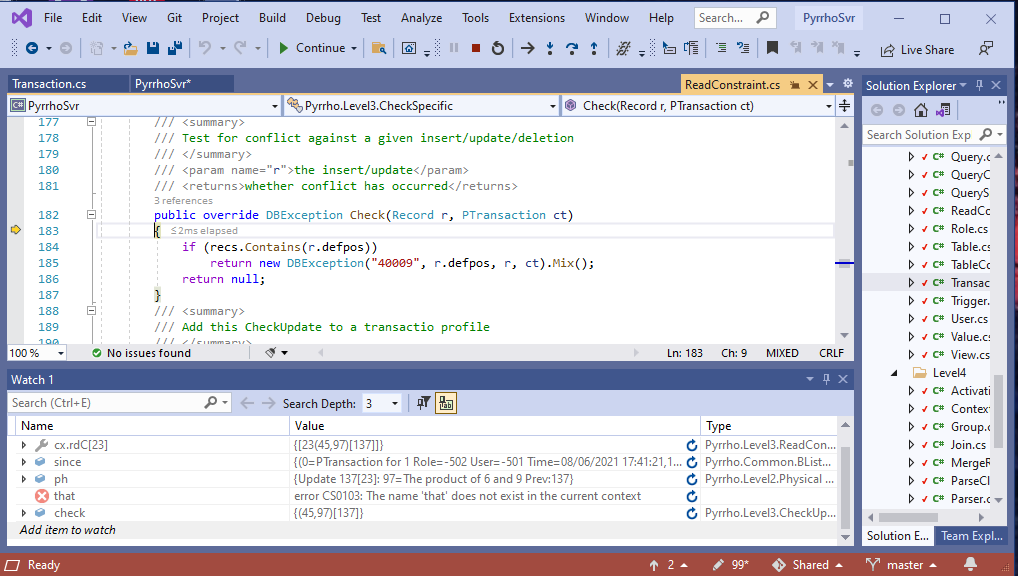
and examine check in the Watch window:



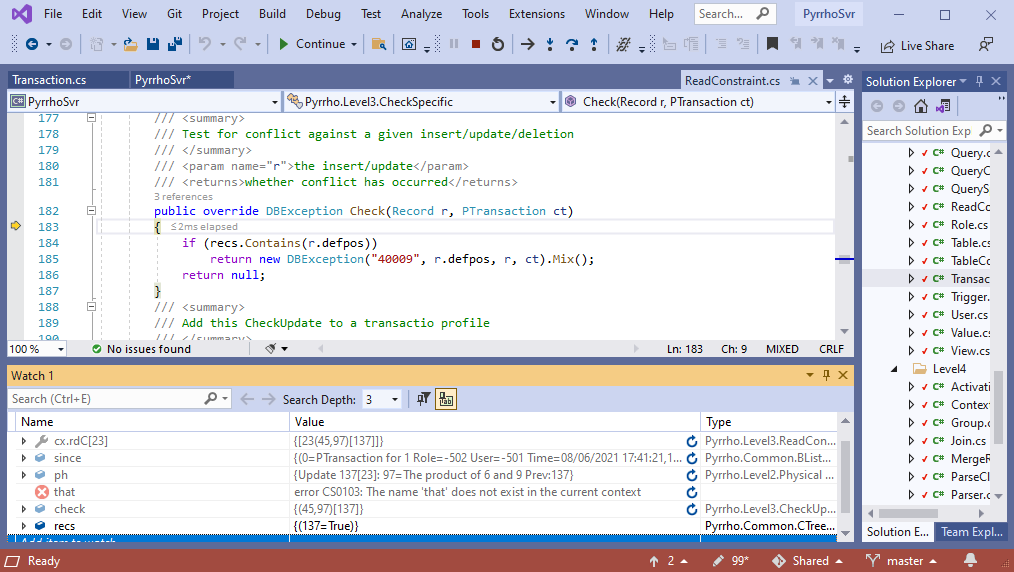
And p is the Update, which is a subclass of Record. Step Over to line 70:



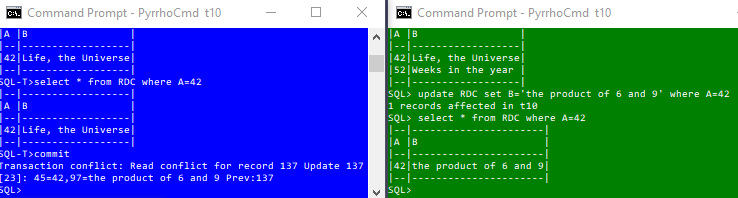
and Step Into check.Check():



Examine recs in the Watch window:



recs contains defpos (that is, the row being updated is one we read), so there is a conflict. Click Continue.

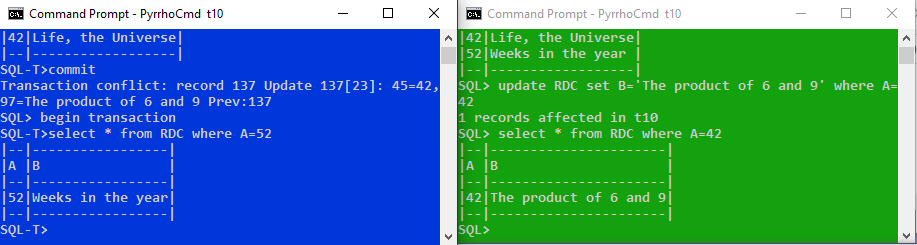


This concludes the second experiment.

As an extra (optional) experiment, let us show that read-write conflicts operate at the row level by using different rows. We can continue in the current state, or repeat the setting up stage. In the blue window:

**begin transaction**

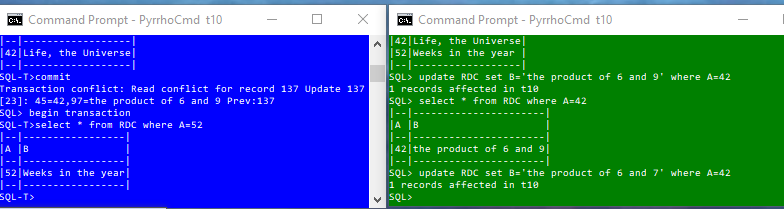
**select \* from RDC where A=52**



and in the green window:

**update RDC set B='the product of 6 and 7' where A=42**

If you still have the breakpoint in Transaction.Commit, just Continue



Now in the blue window everything is fine on commit:

**commit**

