Demo: Pyrrho DBMS and concurrency

Malcolm Crowe, 25 Oct 2022

This demo is updated from a tutorial at DBKDA 2021.

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

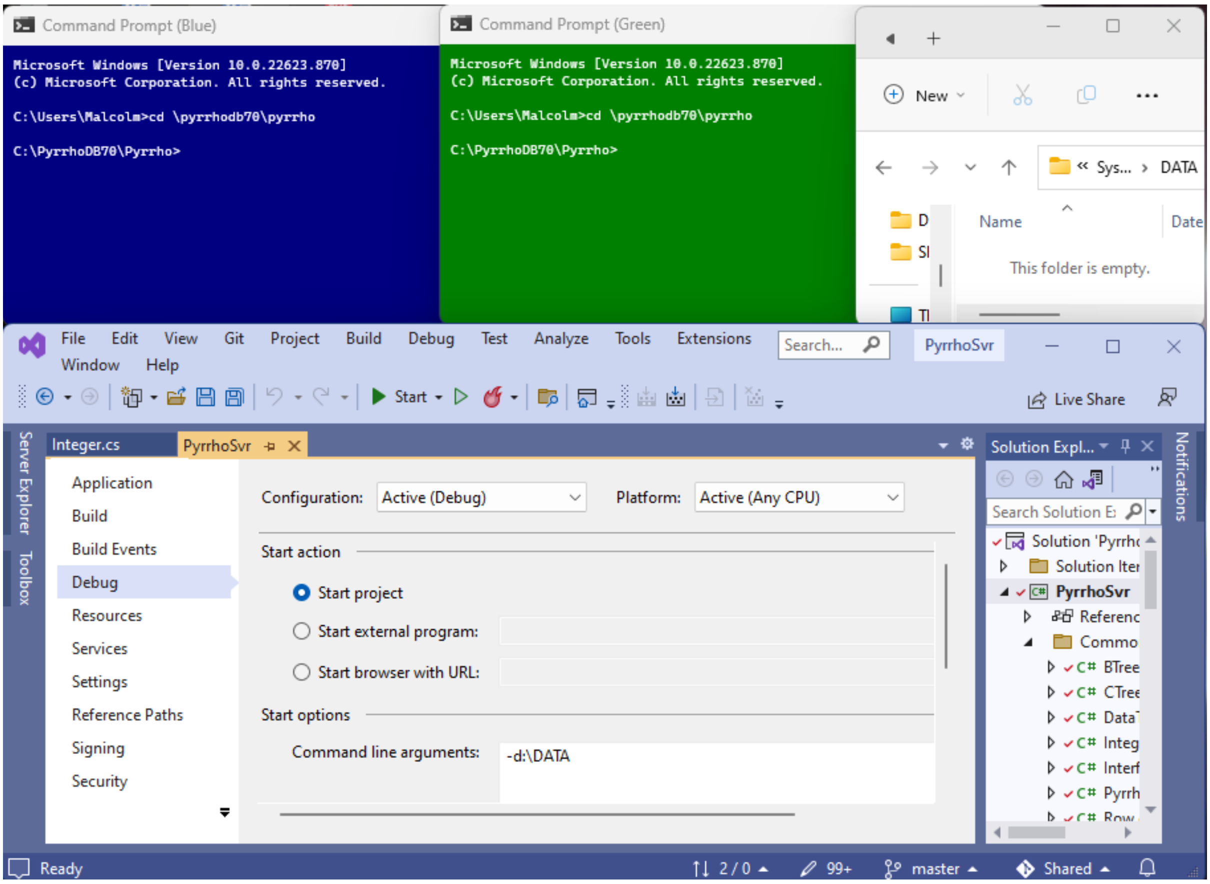
At the start of Transaction Commit, there is a validation check, to ensure that the transaction still fits on the current shared state of the database, that is, that we have no conflict with transaction that committed since our transaction started.

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

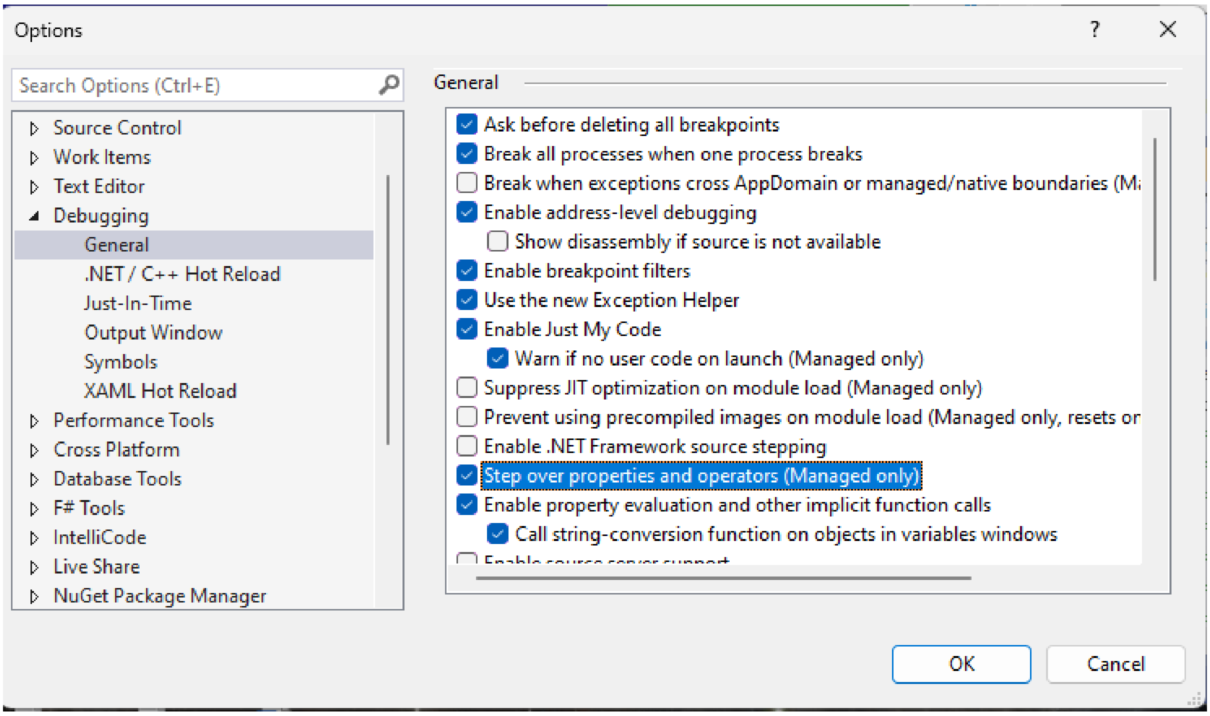
The tests for write-write conflicts involve comparing our list of physicals with those of the other transactions. For checking read-write conflicts, we collect "read constraints" when we are making Cursors.

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. (The command windows are coloured for clarity in these notes: you don't need to colour your windows.)

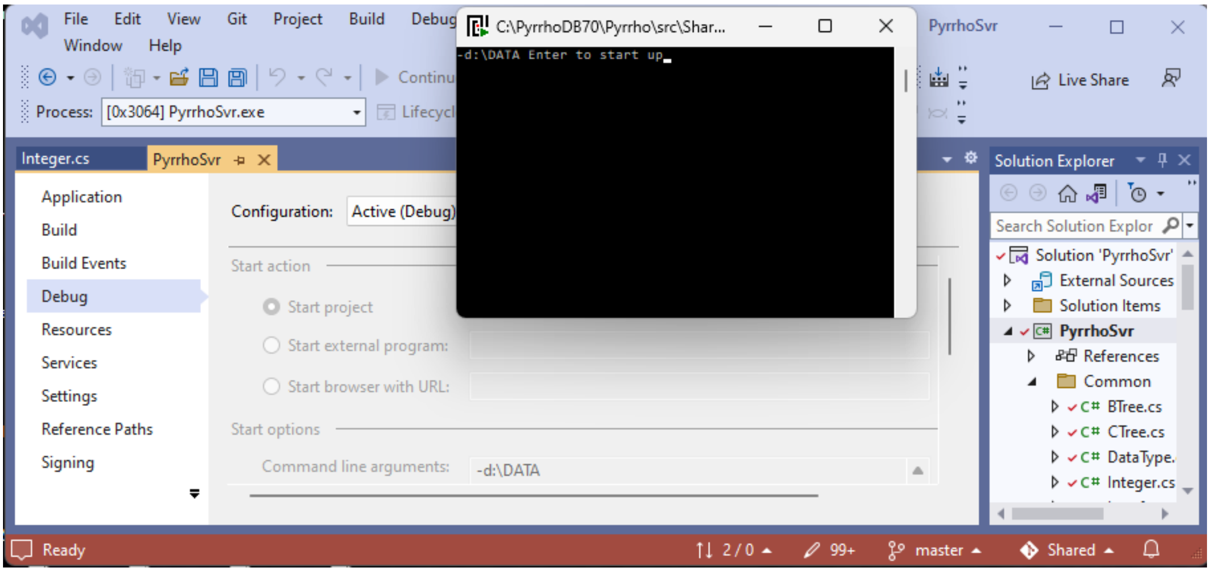


Also ensure that in the Debug>Options.. window,

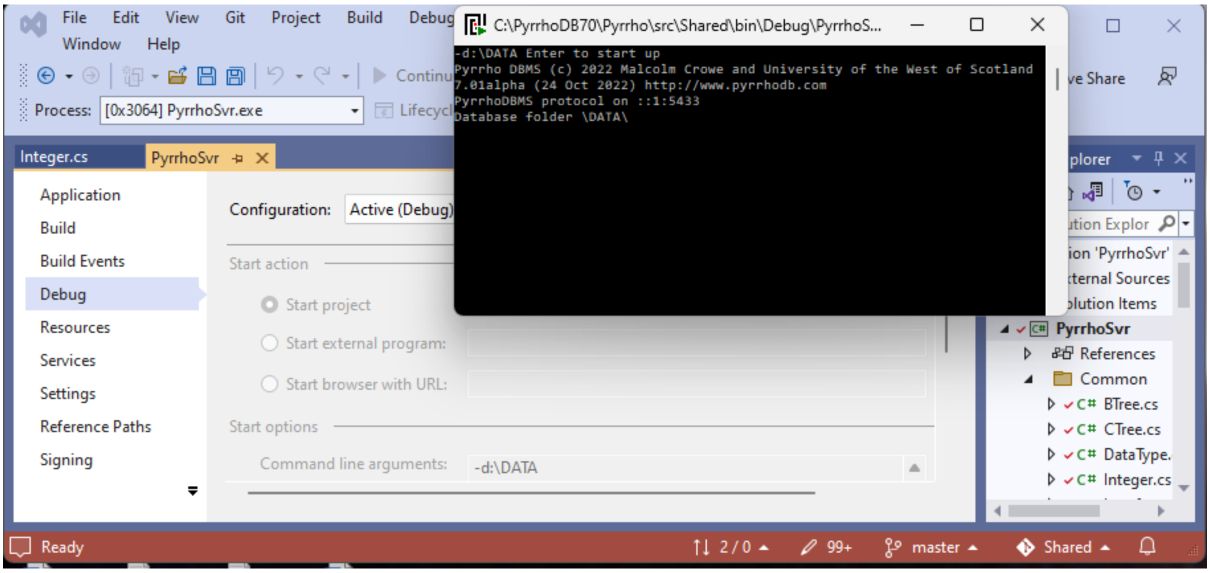


the checkbox Step over properties and operators (managed only) is checked. Click OK.

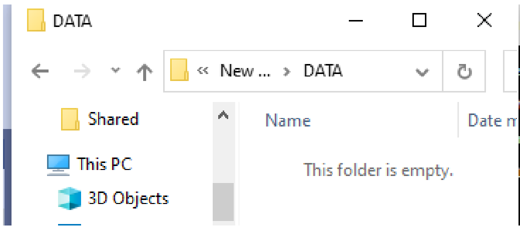
Click Start.



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



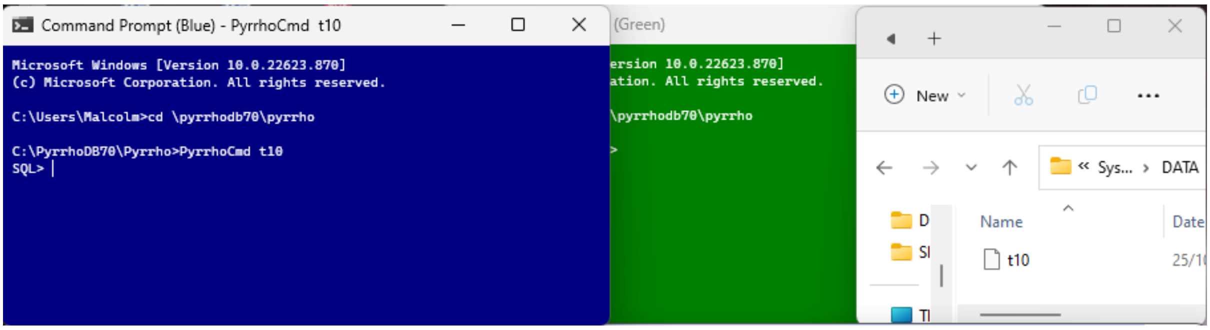
Minimise the pop-up command window.



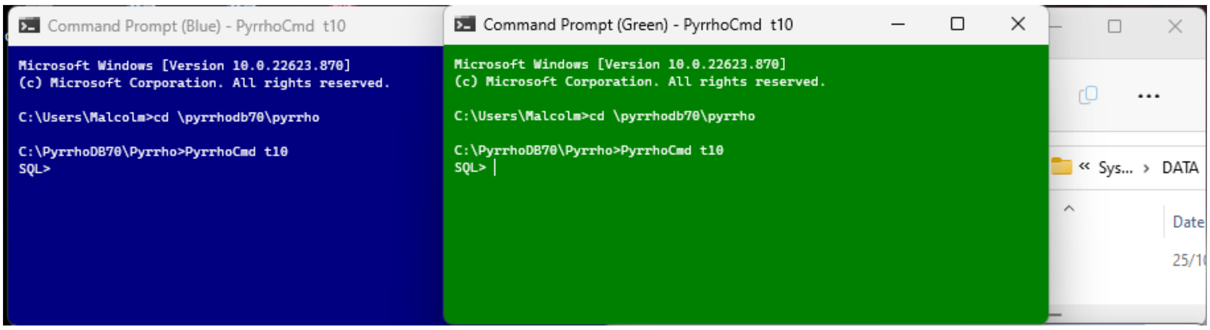
At this point ensure there is no database t10 in the nominated folder.

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).



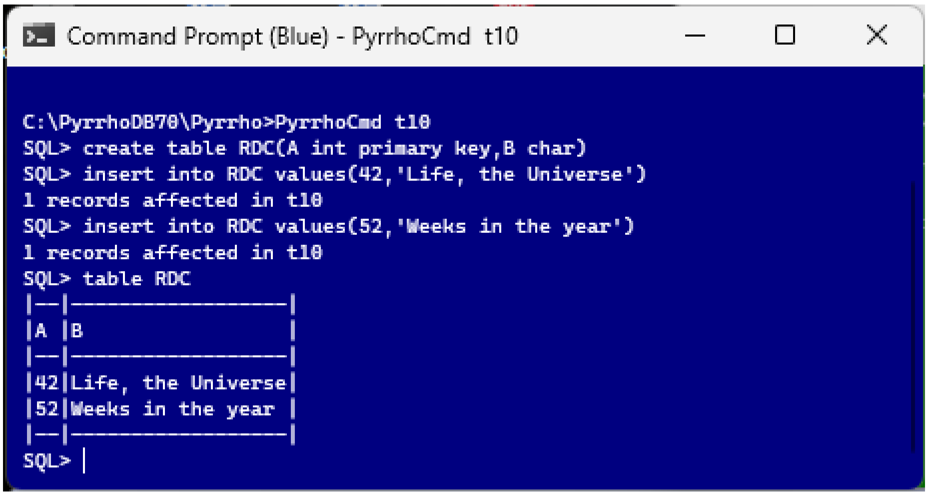
In the blue window, let us create a table (it is called RDC because we will demonstrate read-write conflicts), by giving the following commands:

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

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

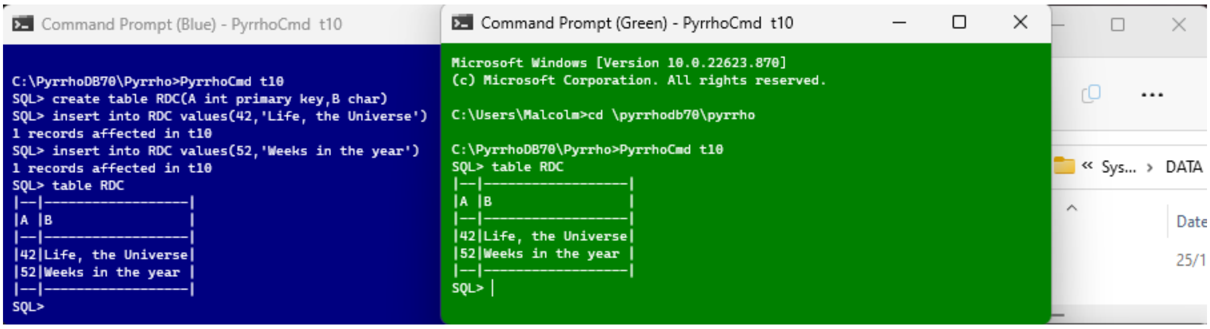
insert into RDC values(52,'Weeks in the year')

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

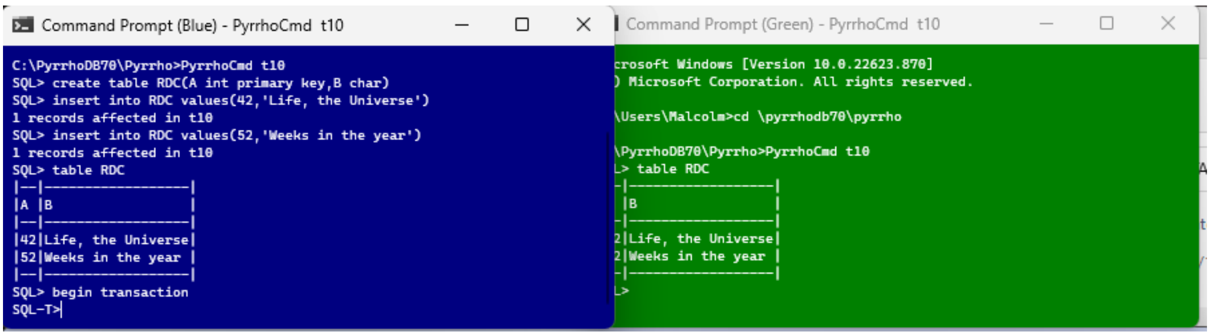


This completes the set up for this demo. We will repeat this part of the demo later.

Part 2: A Write-write conflict

Now we start an explicit transaction in the blue window.

begin transaction

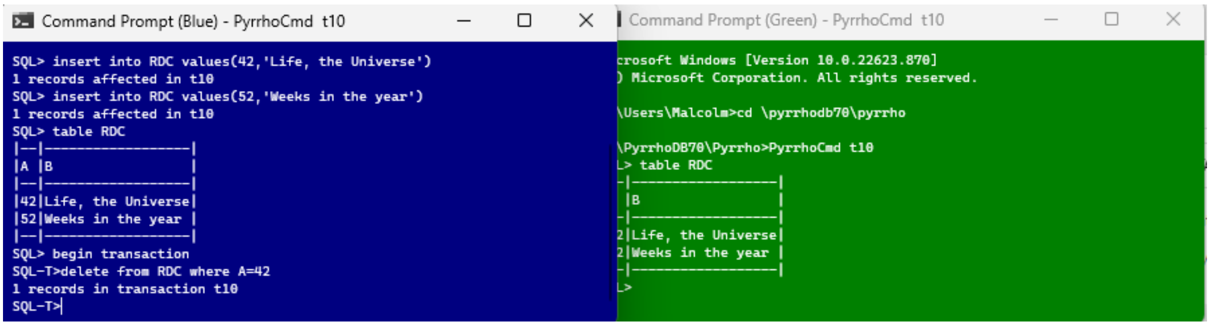


If we just relied on auto-commit mode it is very difficult to synchronise an overlap of transactions. For a demo, it works very well to use explicit transactions.

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
```

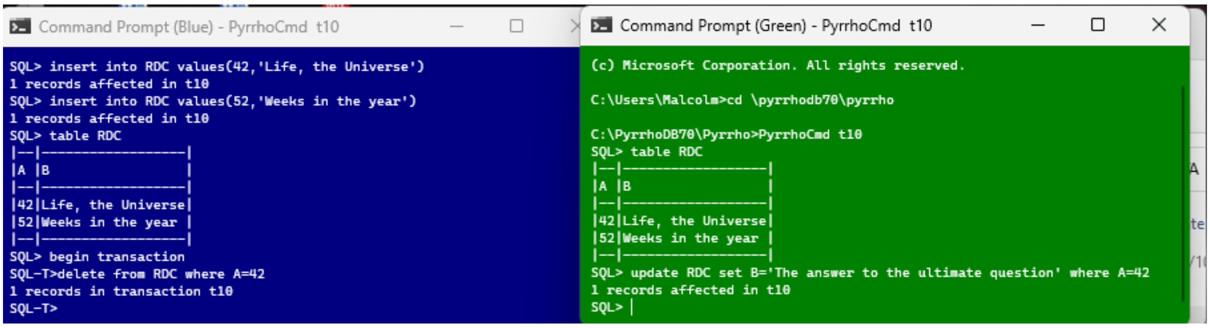


That's in a transaction, so nothing has been written to disk yet.

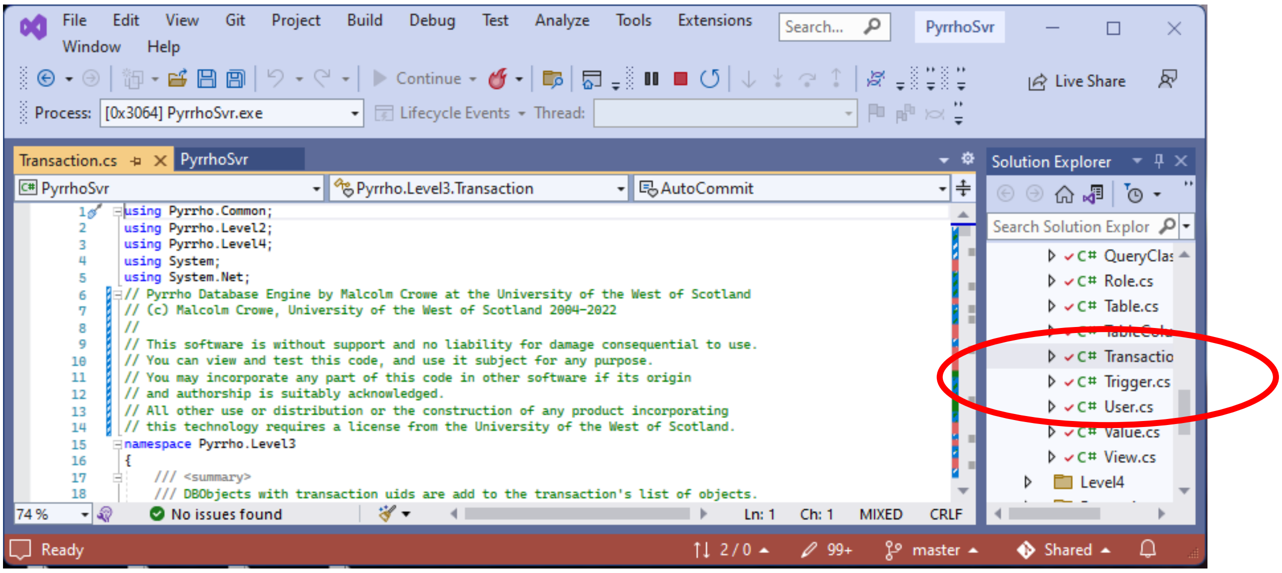
In the green window, we are still in auto-commit mode and we are going to make an update to the same row, which will be committed straightaway to disk.

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

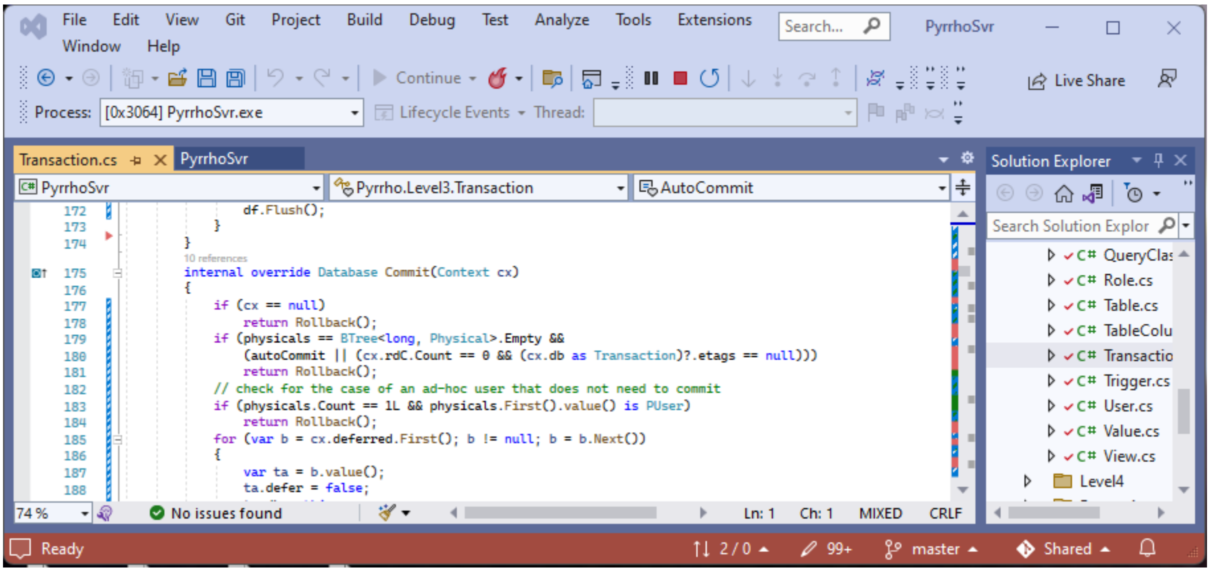
This should make the blue window unable to commit its transaction because of the conflict.



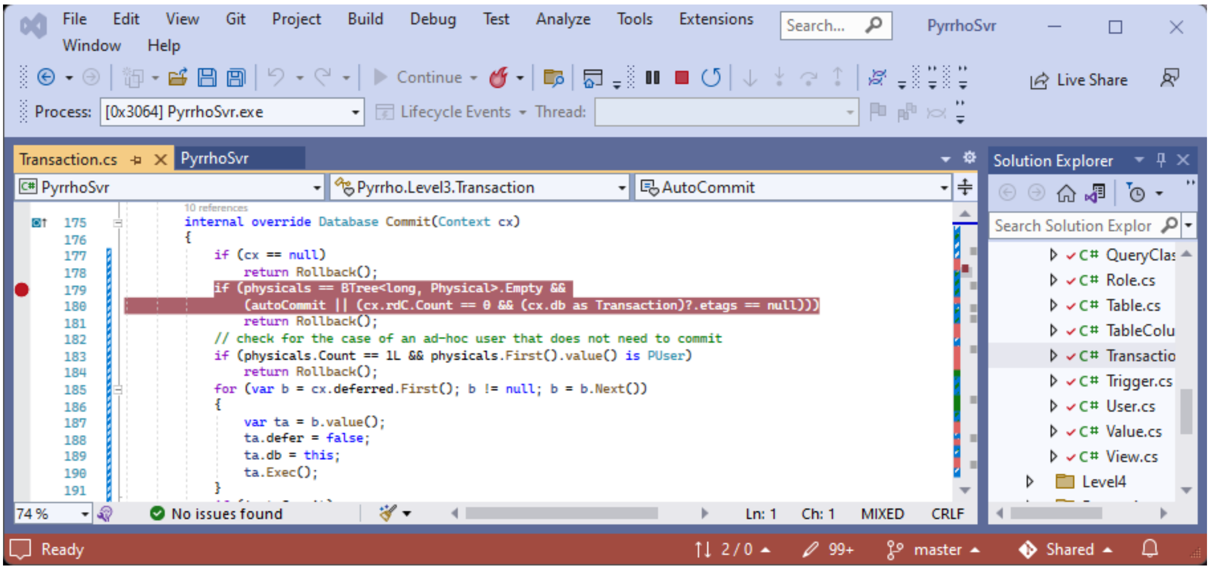
In order to see what happens, let us set a breakpoint in the debugger. In Solution Explorer, in the Level3 folder, locate file Transaction.cs and double-click.



Scroll down to line 175 at the start of the Transaction.Commit() method.



Click the mouse in the left margin to set a break point at line 179 of the Transaction.cs file.

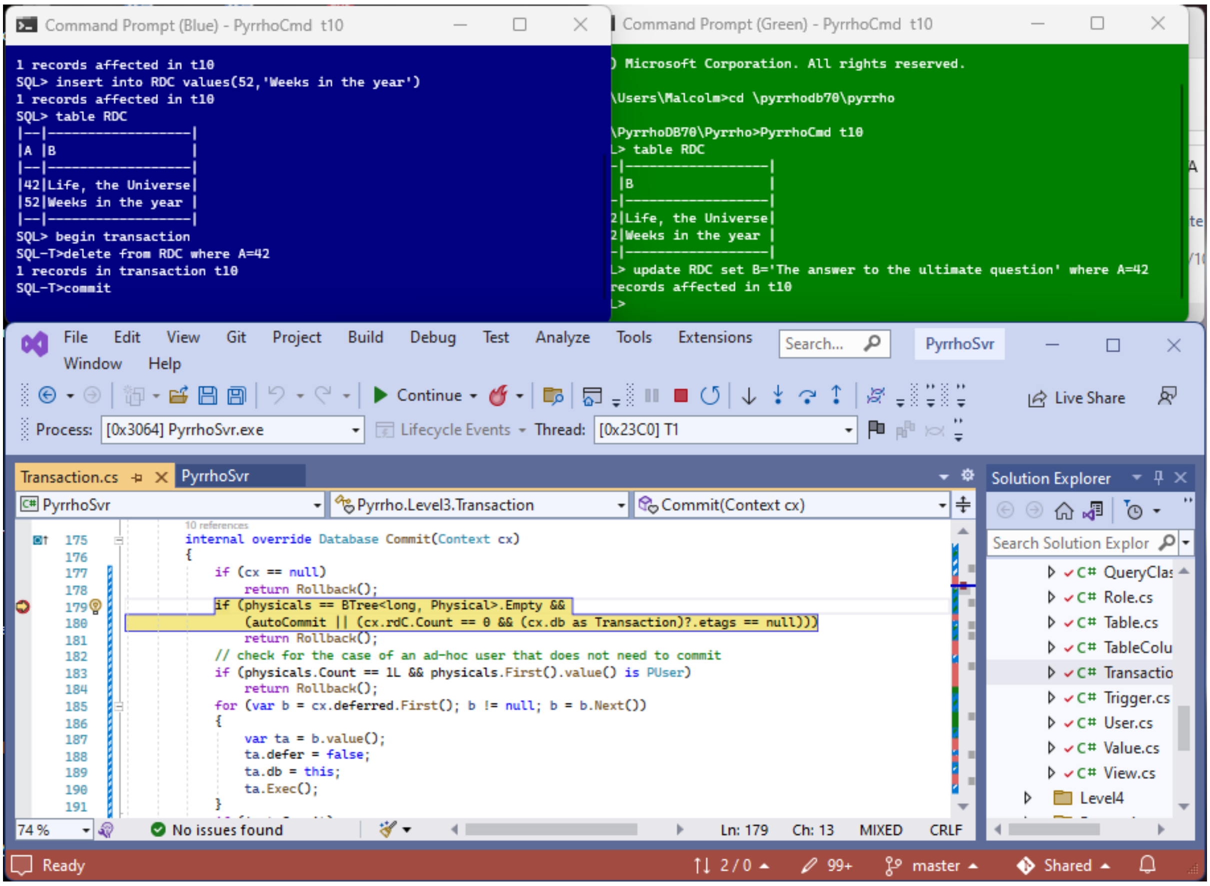


[39 @ 19:17]

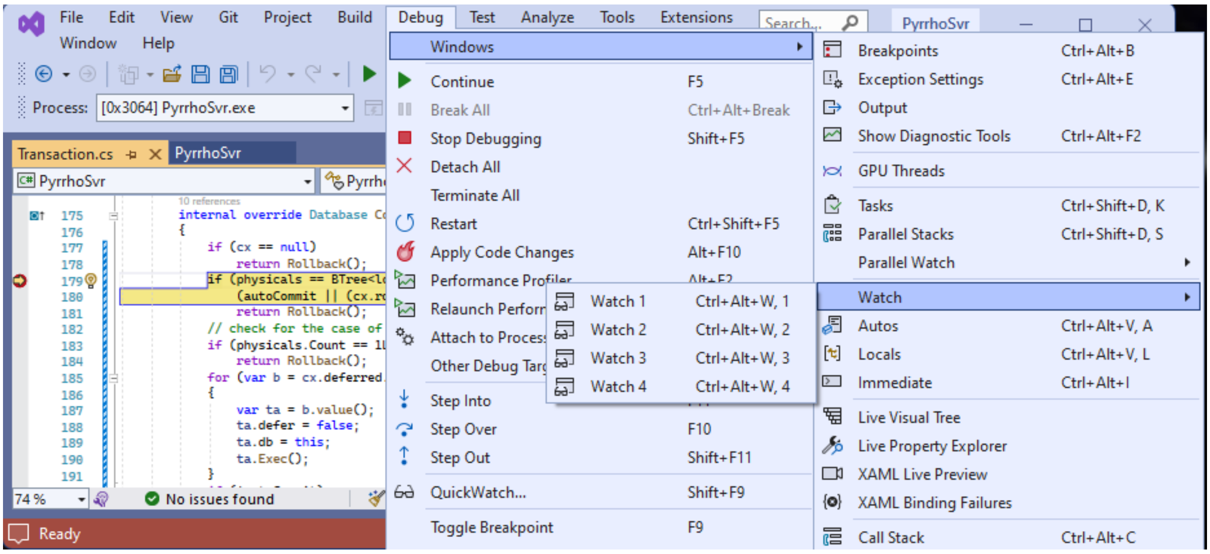
Then when we issue the commit command in the blue window,

`commit`

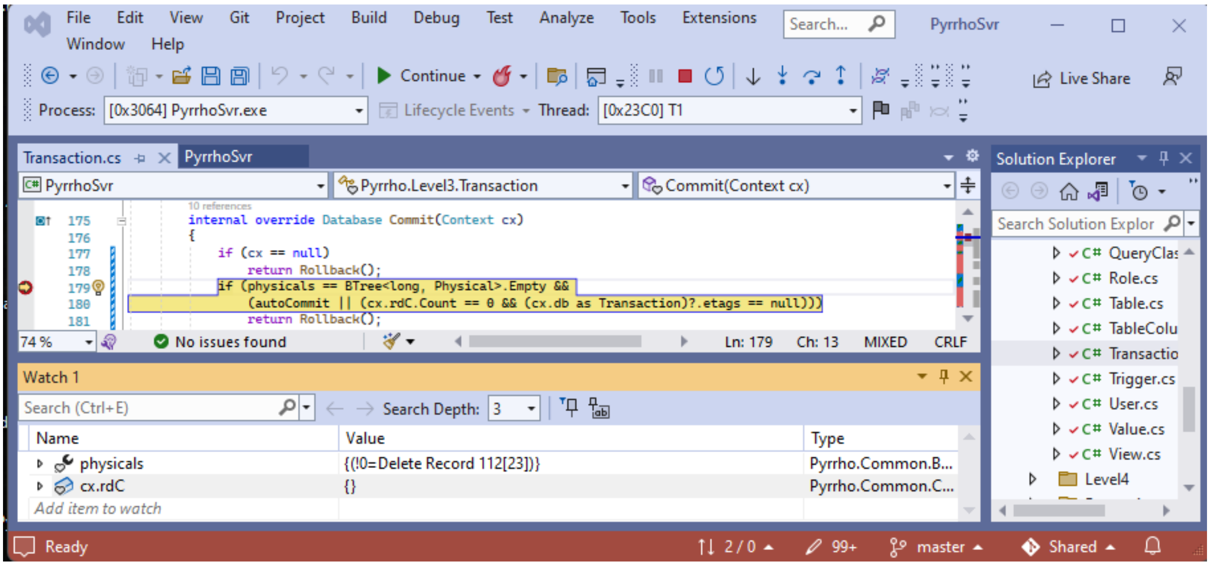
Visual Studio stops at the breakpoint.



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



and Dock it below the text window. Use it to examine `physicals` and `cx.rdC`:



[41 @ 19:33]

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 112 in the database which is "Life, the Universe".

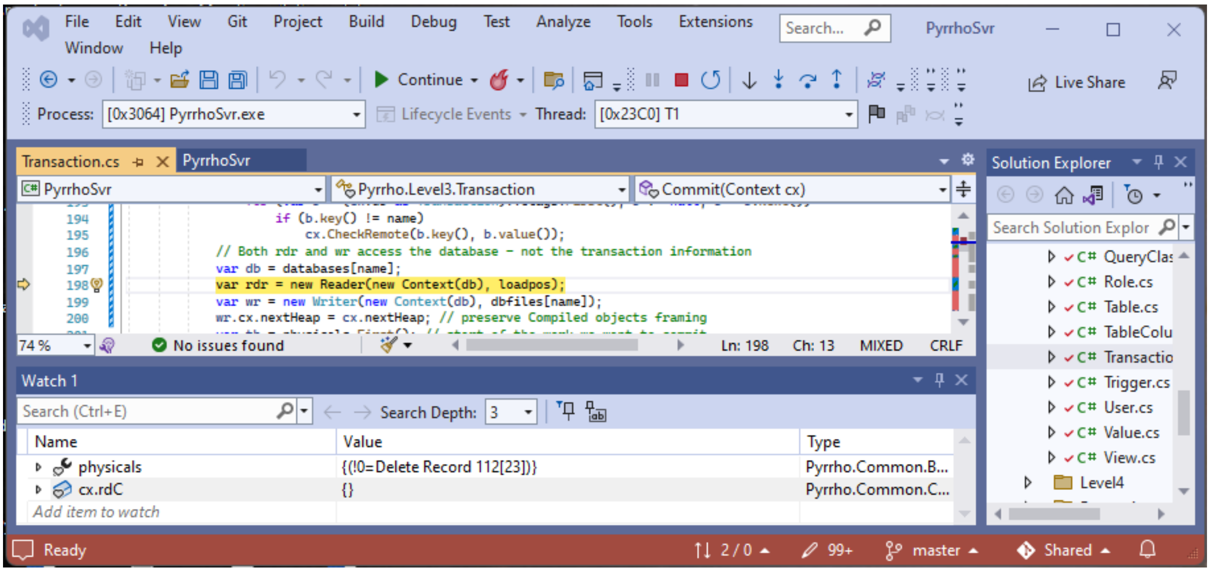
[42 @ 19:50]

Also look at cx.rdS[1], which is to do with the ReadConstraints for this particular Transaction. In this version of Pyrrho, it is currently empty because we have not performed any selects.

[43 @ 20:12]

Step Over a few times to get to line 198.

[1] The illustration shows rdC instead. We will see in the next section that we really want rdS here.

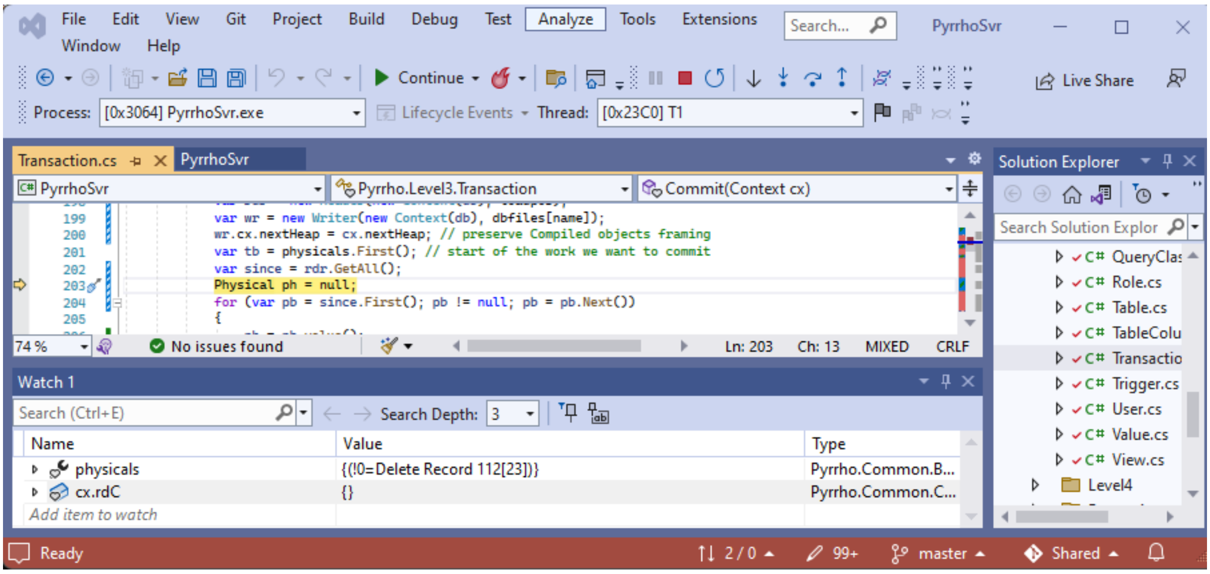


The validation step will use the up-to-date copy of the database, as it was left by the green window. We also open a Reader and a Writer: a Reader to look at this database, specifically at the records that have been committed since the start of our Transaction; and a Writer, where we will prepare the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

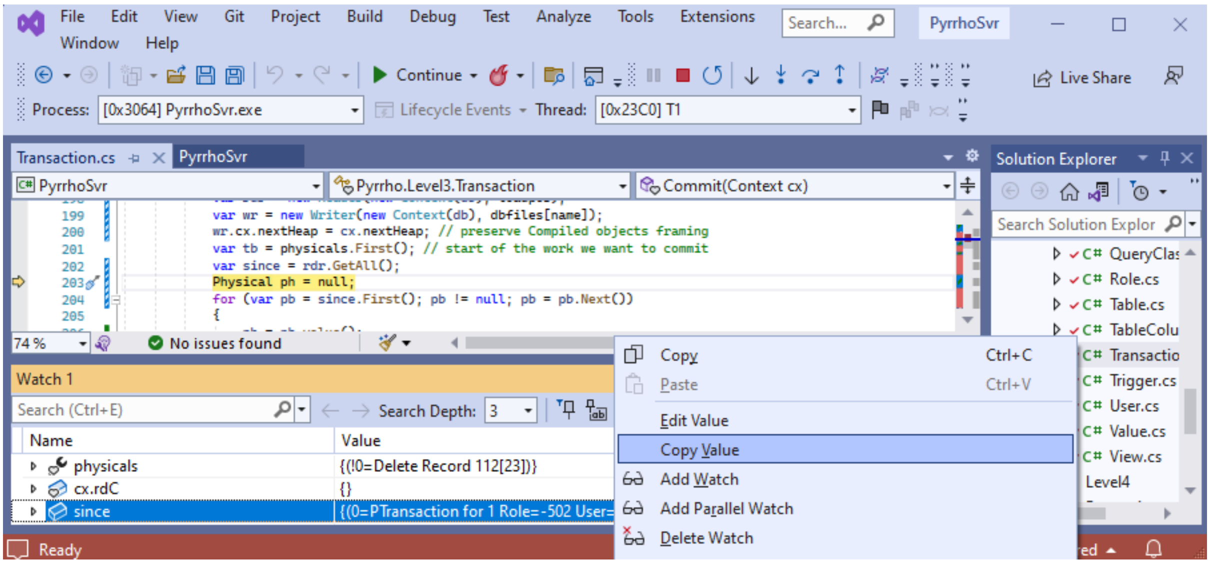
These are not shareable and are subject to locking protocols. They also work on the same FileStream. (Transaction Commit() repeats the validation step after the locking the FilleStream.)

[44 @ 20:27]

Step Over to line203, 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.

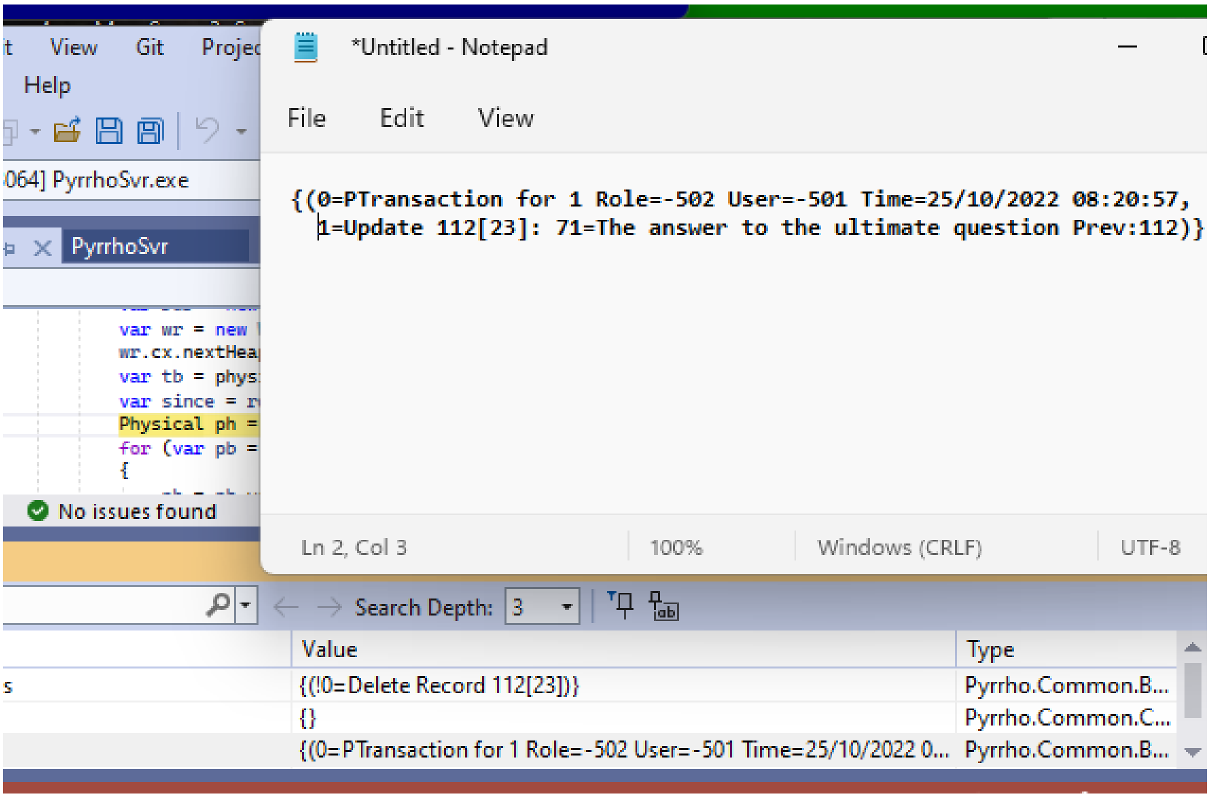


Use the Watch window to examine **since**.



[45 @ 21:23]

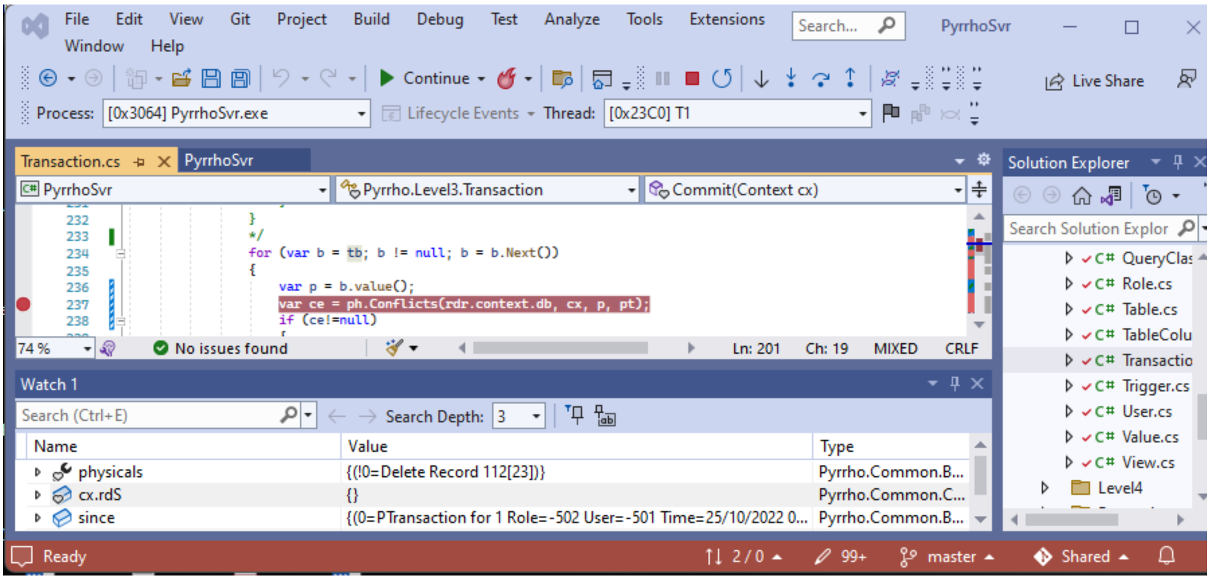
Right-click and copy the value into a Notepad. Add a little extra white space to make it more readable.



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

We have just (at line 201) set tb = physicals.First(), a bookmark for the first physical item that our transaction is wanting to commit. We will check our physicals with anything that has happened since the start of our transaction.

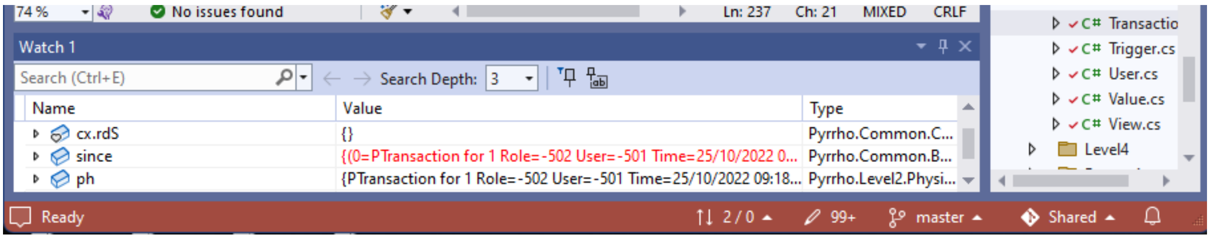
Set a breakpoint at line 237.



Click Continue

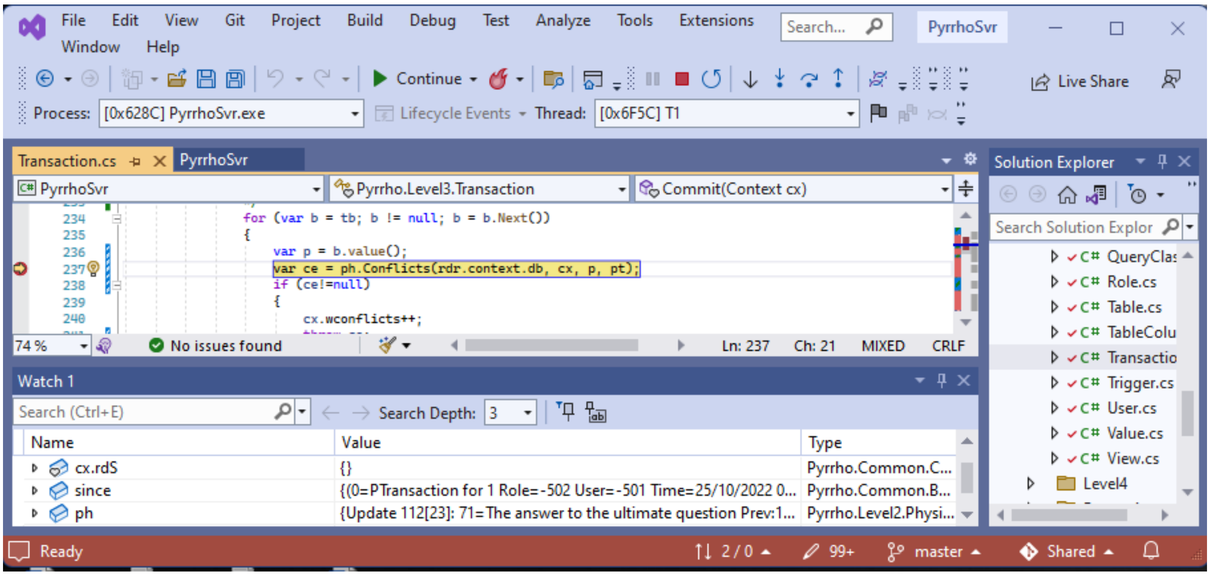
[47 @ 21:41]

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

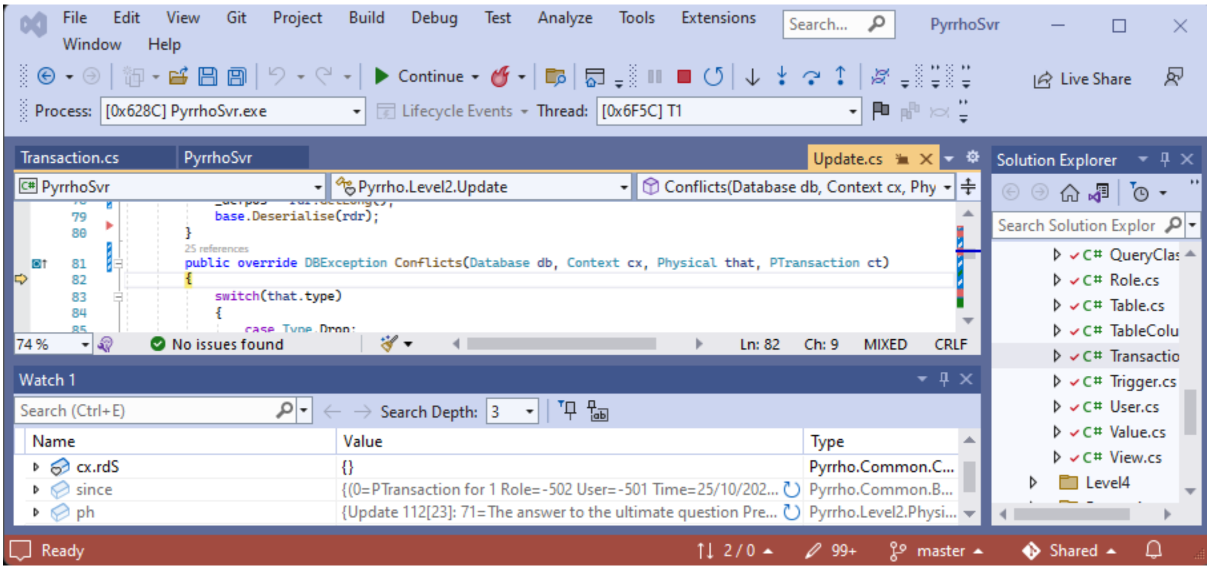


which is not very interesting.

Click Continue

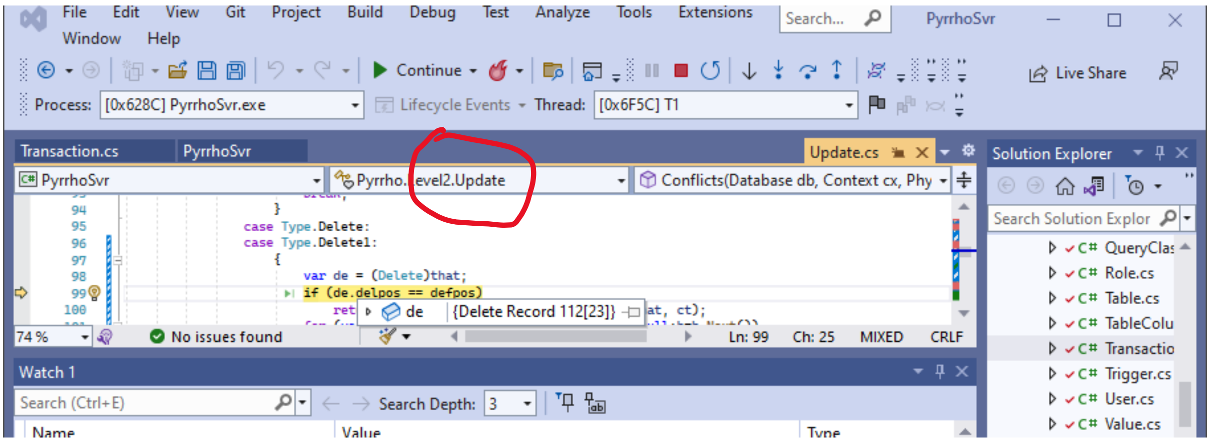


The second time, ph is the Update from the green window, and we Step into the call to Conflicts:

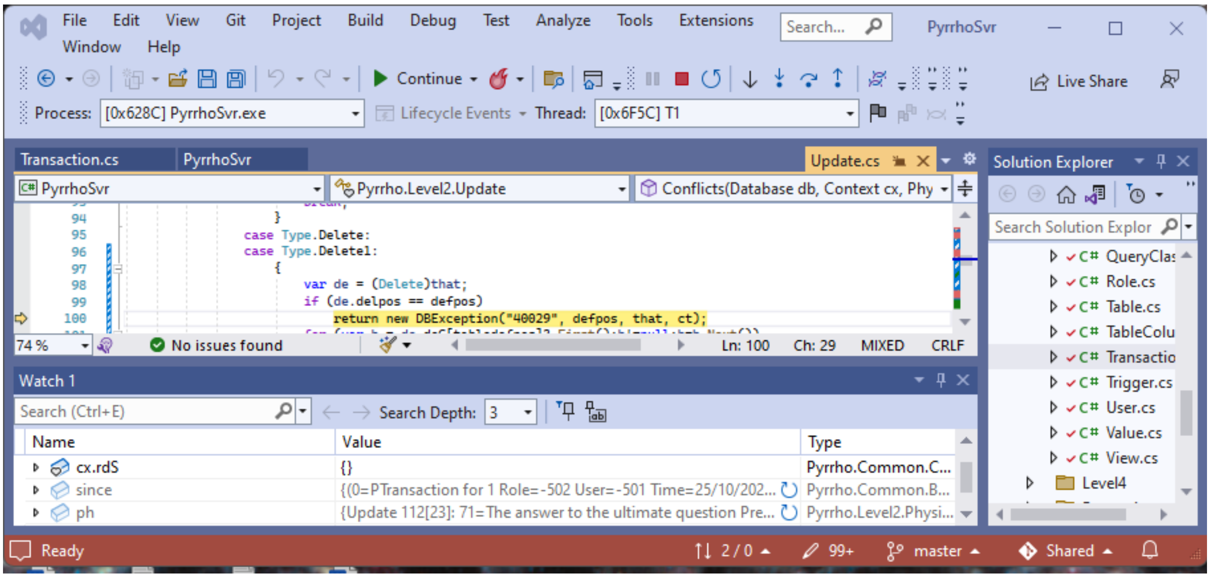


[49 @22.04]

Click StepOver: stop at line 99, where our Delete is tested against their update:

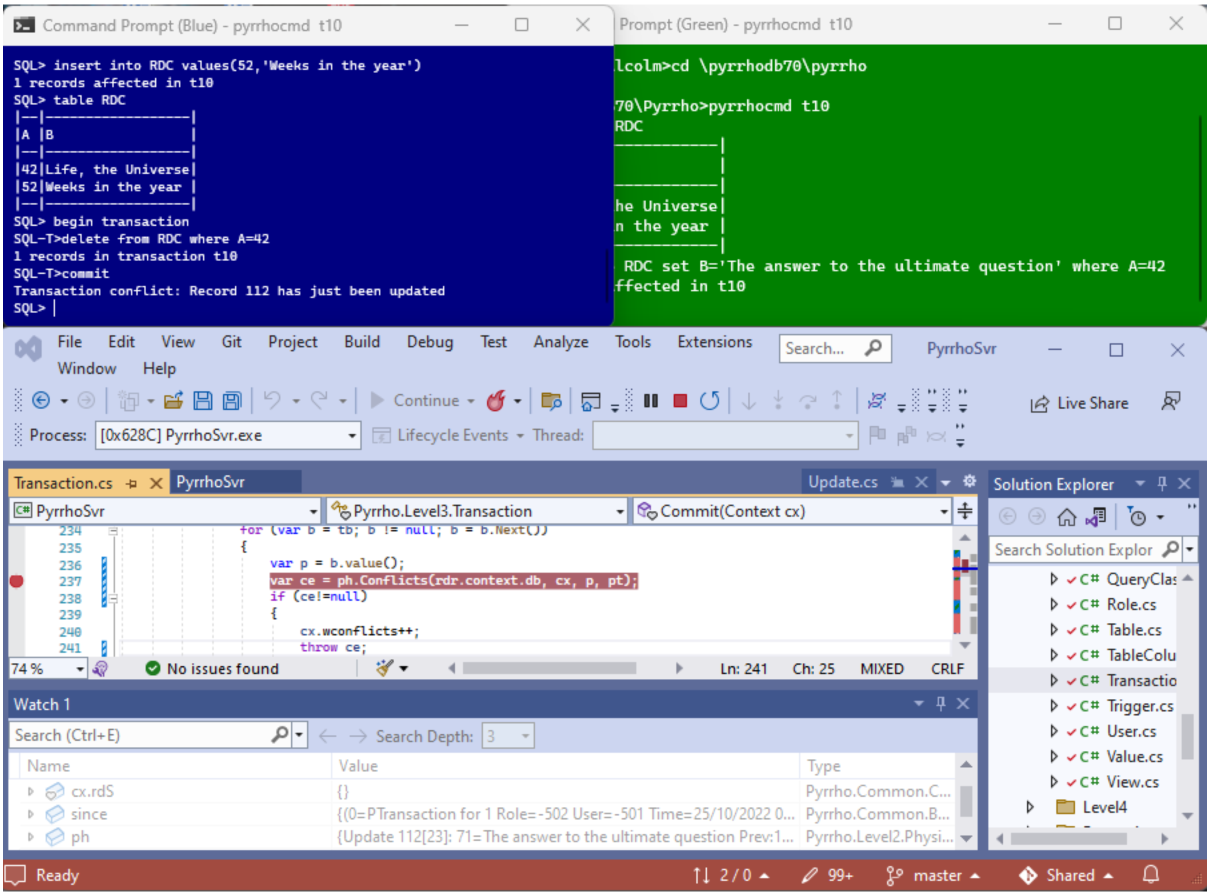


and since these both are for the same defining position 112, we will raise an exception.



[51 @ 22:35]

Click Continue twice, and we get the resulting message in the blue window:



We see that we get a complaint back in the blue window that record 112 has just been updated. The transaction has been rolled back, as we see from the prompt.

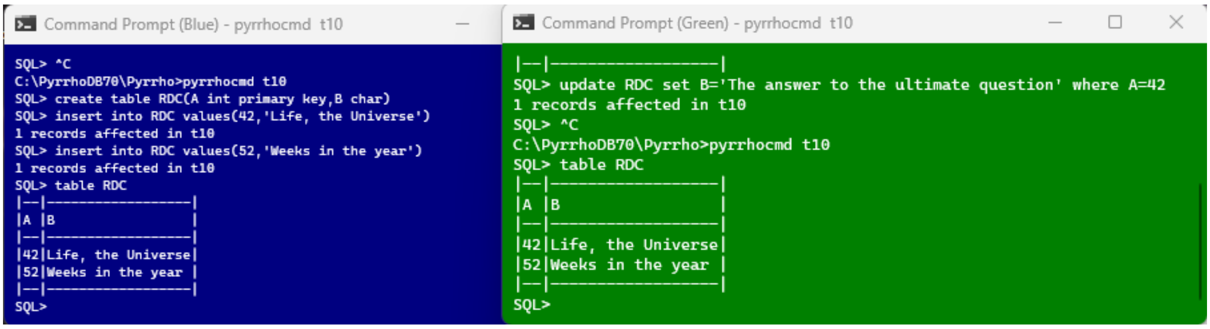
That completes the first experiment that we want to do in this demonstration.

[53 @ 23:01]

Part 3: A Read-Write Transaction

Stop the server, stop the programs in both command windows with control-C, delete the database t10, restart the server, and recreate the database as in part 1 above, so that we have the same situation as slide 35.

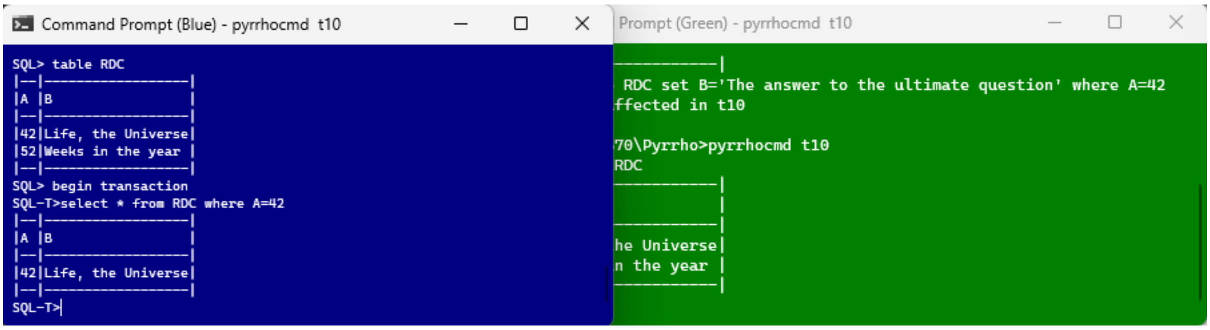
[54 @ 23:18]



This time, when we start an explicit transaction in the blue window, instead of deleting something, we are going to select a single row from the RDC table, and we stick with A=42 again.

begin transaction

select * from RDC where A=42

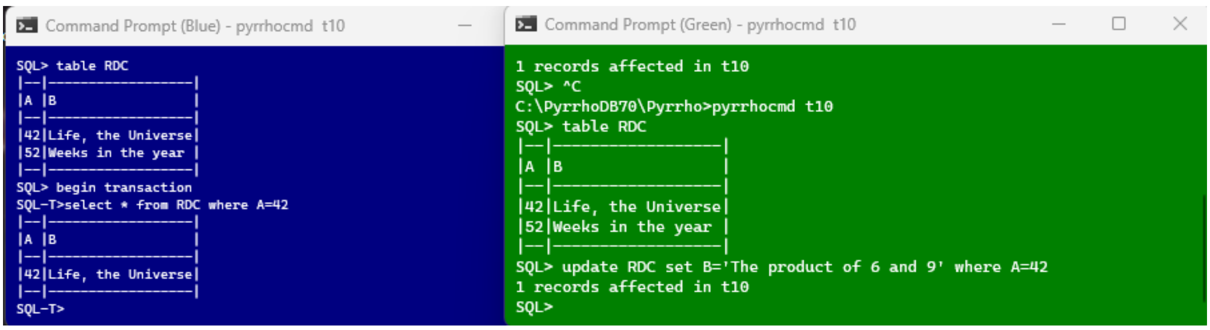


[55 @ 23:33]

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

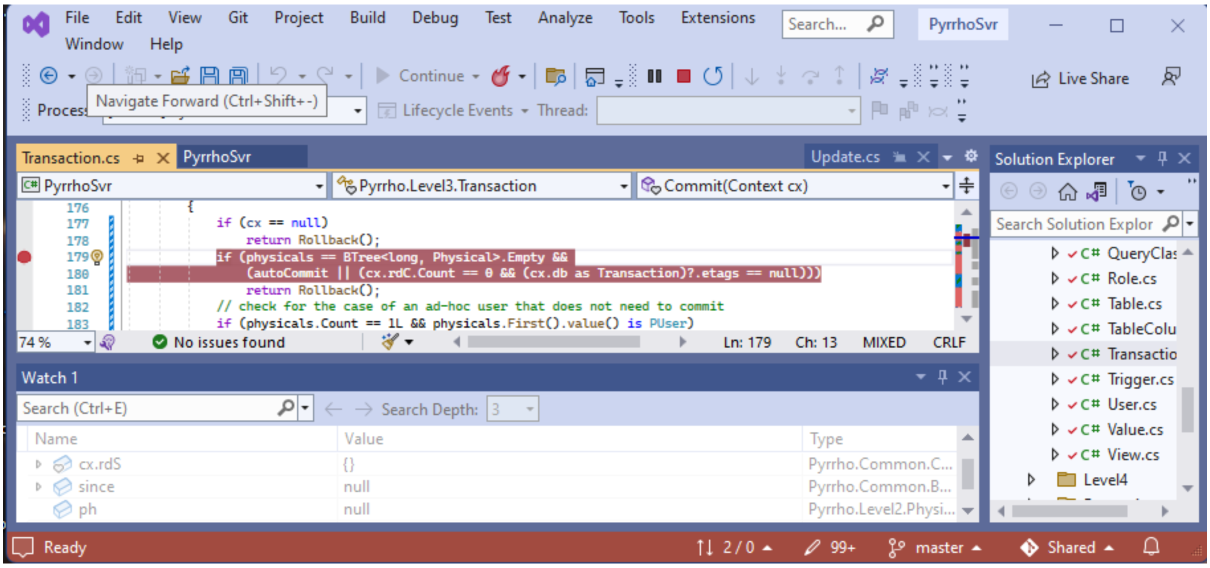
update RDC set B='The product of 6 and 9' where A=42

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



[56 @ 23:48]

Add or restore the break point in the Transaction.Commit() method.

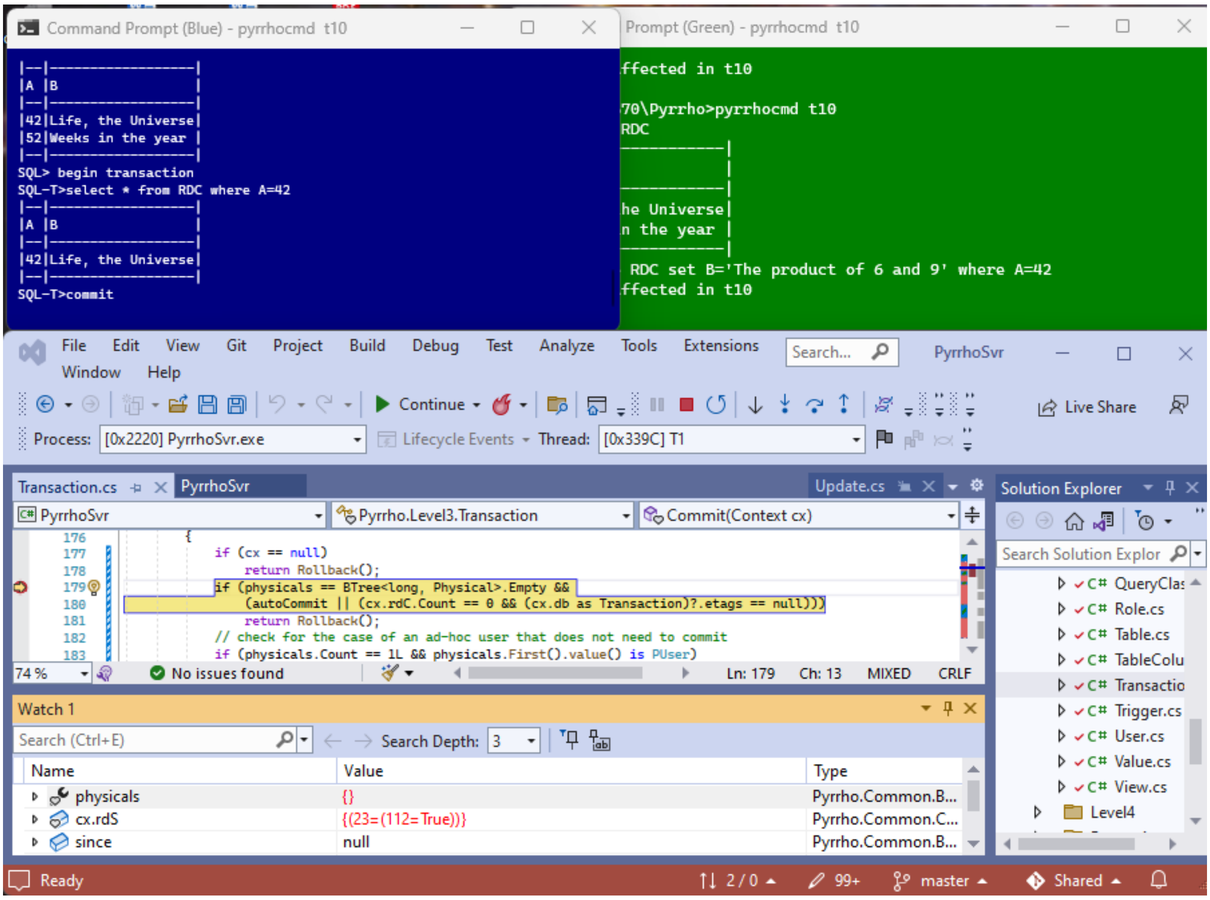


[57 @ 23:25]

and in the blue window,

commit

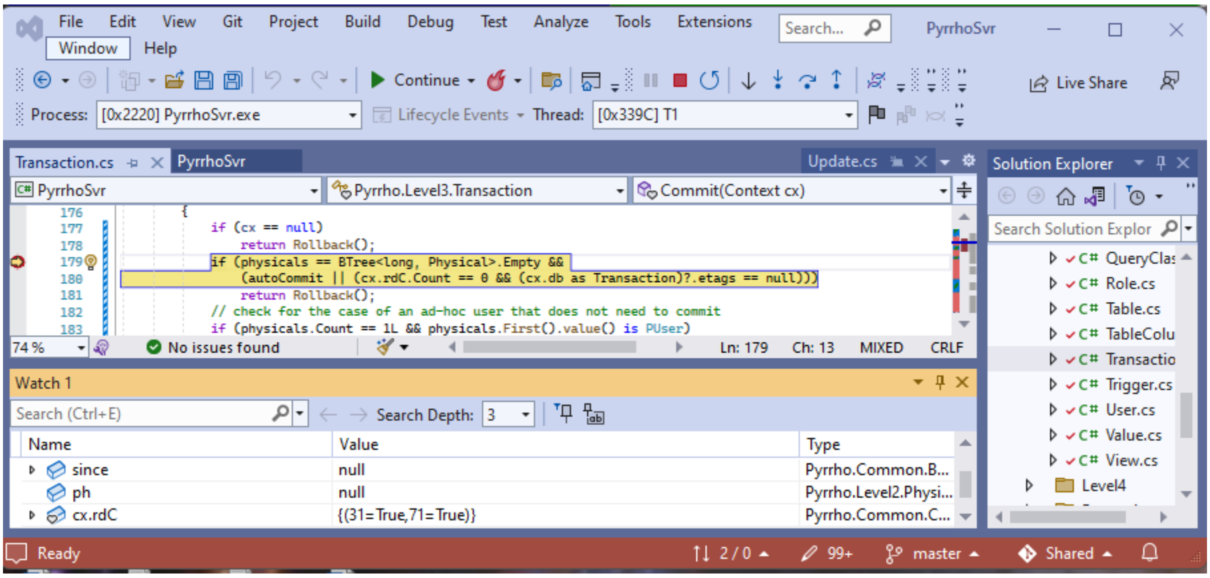
On the commit command, we hit the break point.



[58 @ 24:03]

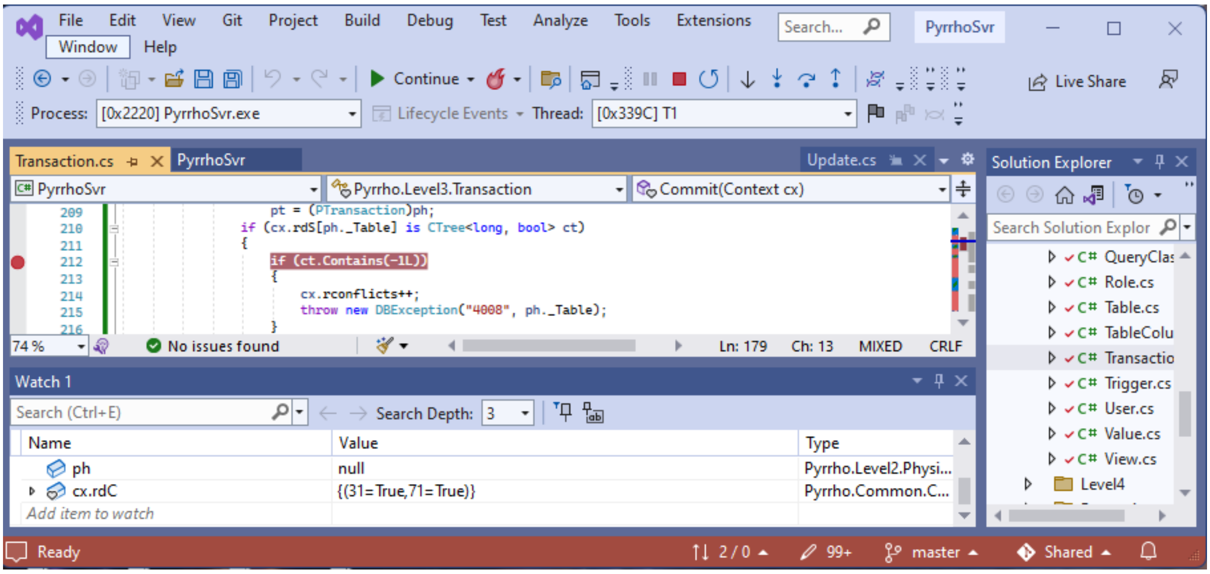
In the Watch window, we can see that this time the physicals list for the blue window is empty, but it has a readConstraint in rdS, for table 23 and the single record 112 that it has read. (We haven't fetched

since yet.) If you also look at cx.rdC, you will see that it tells us that both columns 31 and 71 have been read.

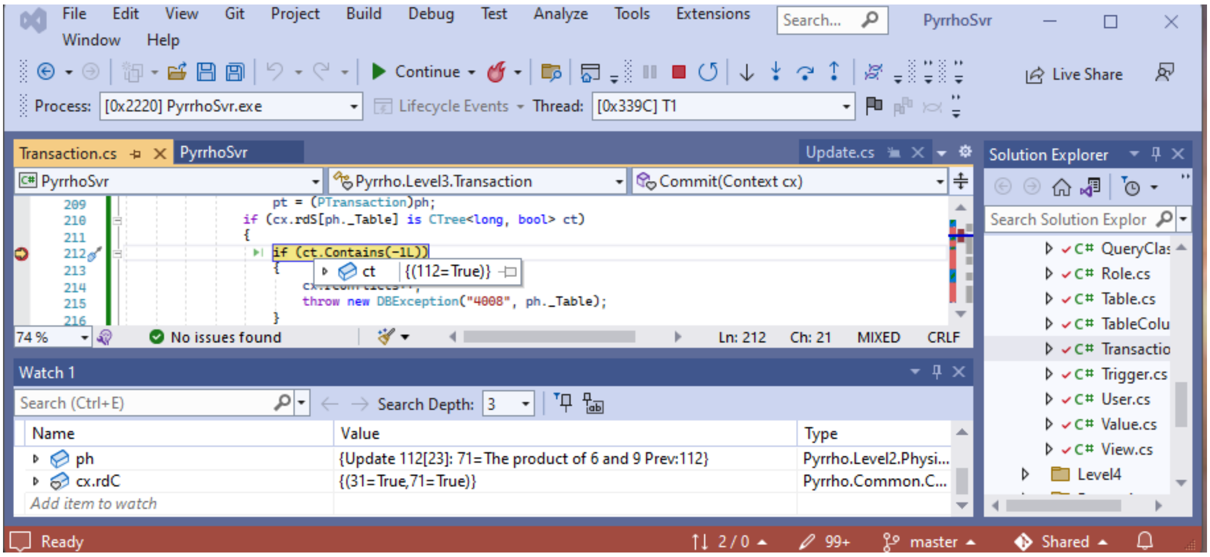


[60 @ 24:31]

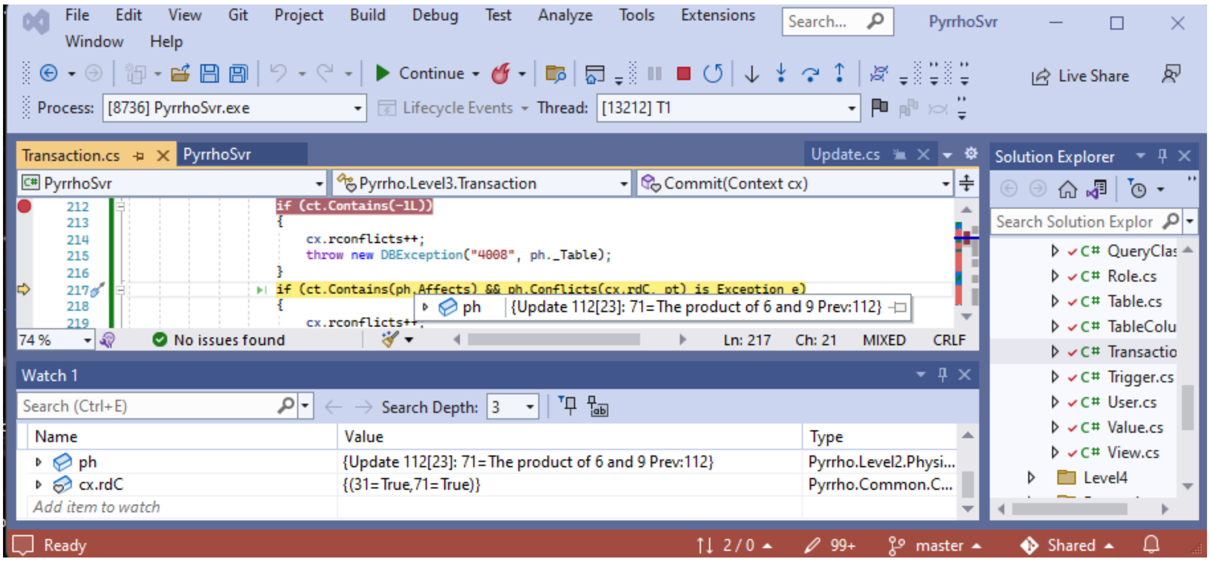
Since rdS is not empty, let's put a break point at line 212:



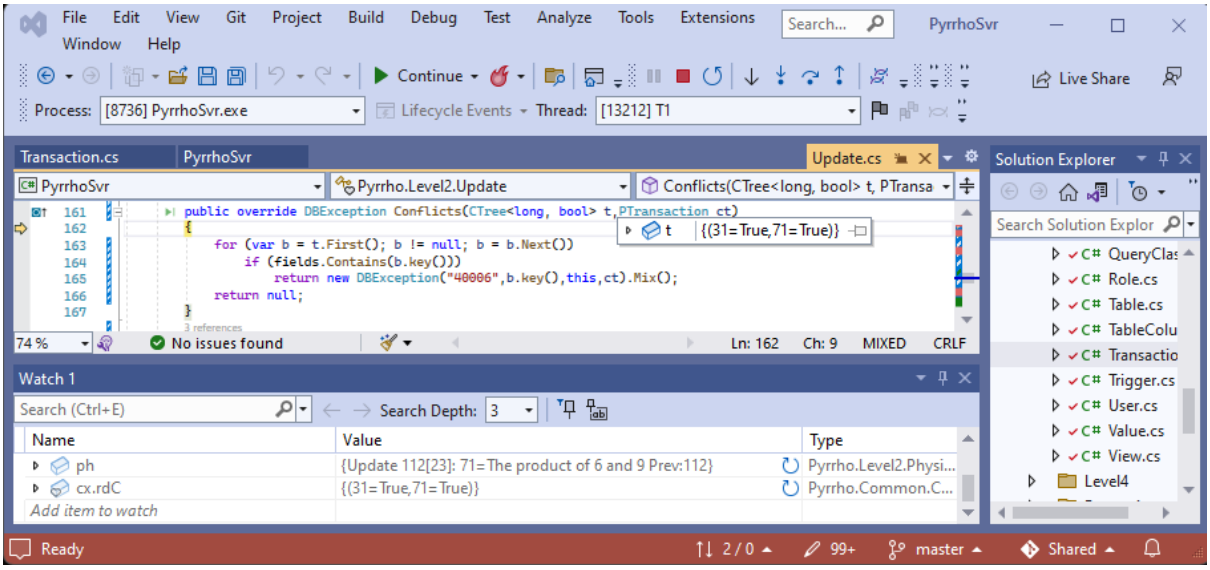
and click Continue to reach this point:



We see the readConstraint for our transaction is for a specific row 112. Step over to line 217,

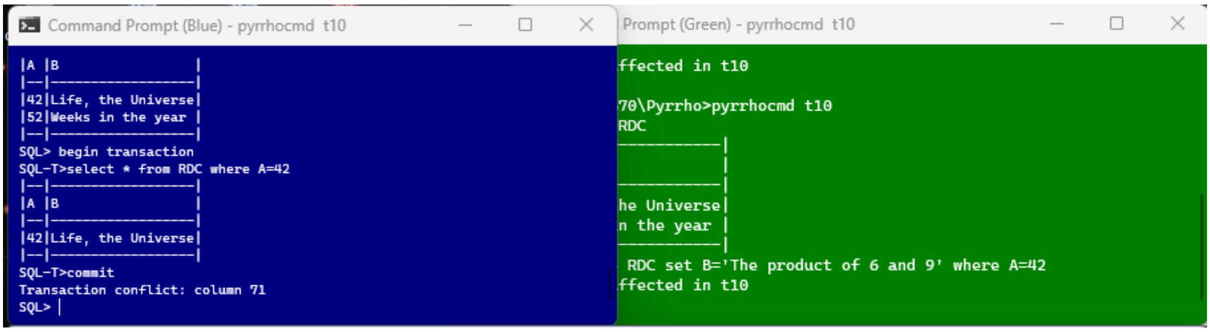


where we see that the green window’s update is for the same row. Step into the Conflicts call, which receives the list of columns we have read:



[64 @ 25:28]

and the exception is raised, terminating the blue transaction (click Continue twice).



This completes the demostration of a read-write conflict.

[Ends]

As an extra (optional) experiment, we can 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

We can check that selects and updates even for the same row will not conflict if the columns affected are different.

These tests are left for you as an exercise.