Demo 3: Updatable Views

Malcolm Crowe, 25 Oct 2022

[Slide 78 @ 36:05 of the video]

This demonstration explores the operation of RowSets, by considering the implementation of Views. [The process has been greatly streamlined in version 7.01, so that there are fewer stages in the account below than in the tutorial video, although the overall approach is similar. A more detailed account of this example is contained in the SourceIntro document in section 6.6.]

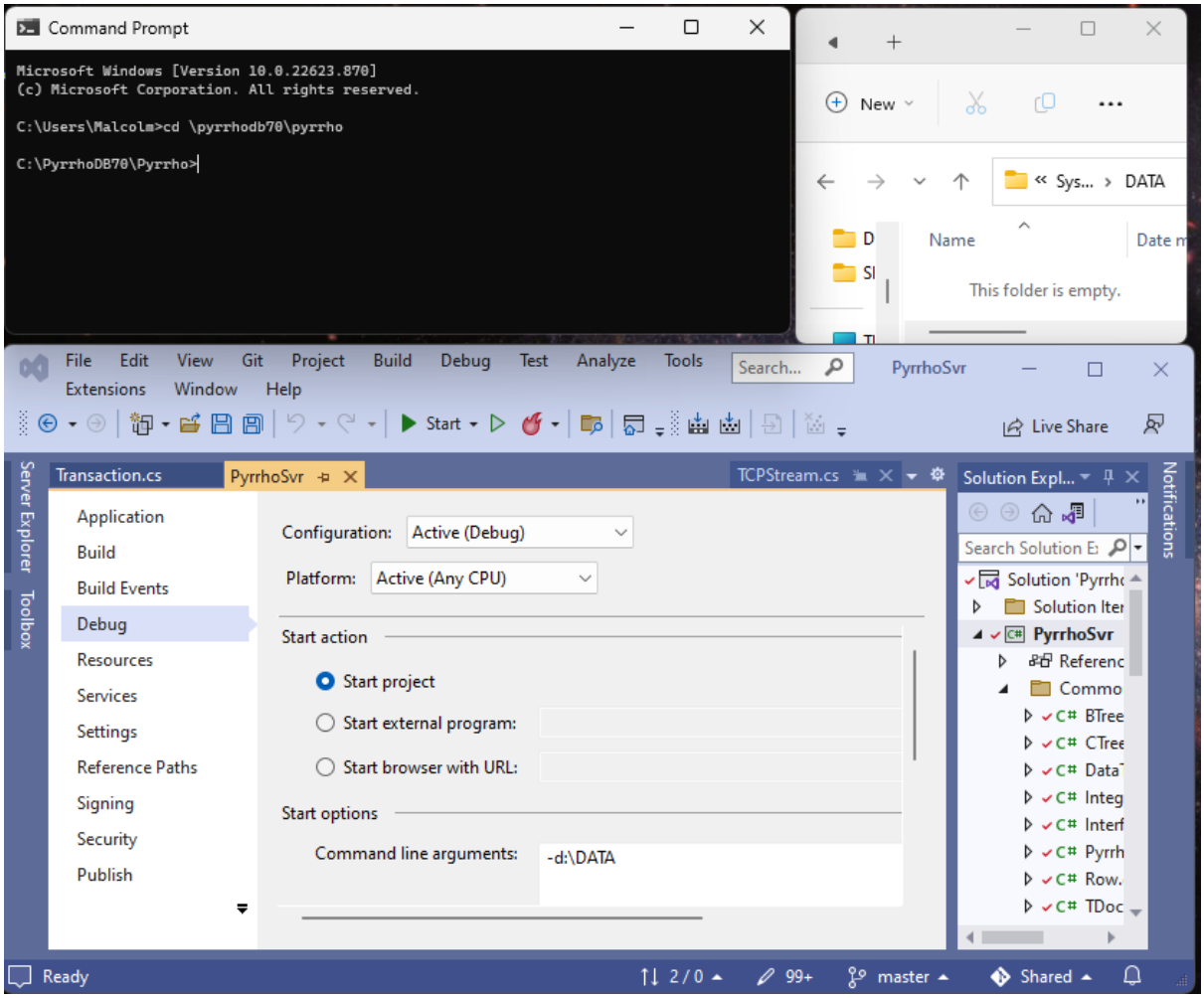
We will see that RowSets can be used for update, insert and delete actions in addition to queries

We will see the use of precompilation of complex database objects

We will see that RowSet analysis helps ensure that operations on Views are implemented as operations on the (possibly remote) base tables, but only if the user has the right permissions. (We consider remote data sets in Demo 4.)

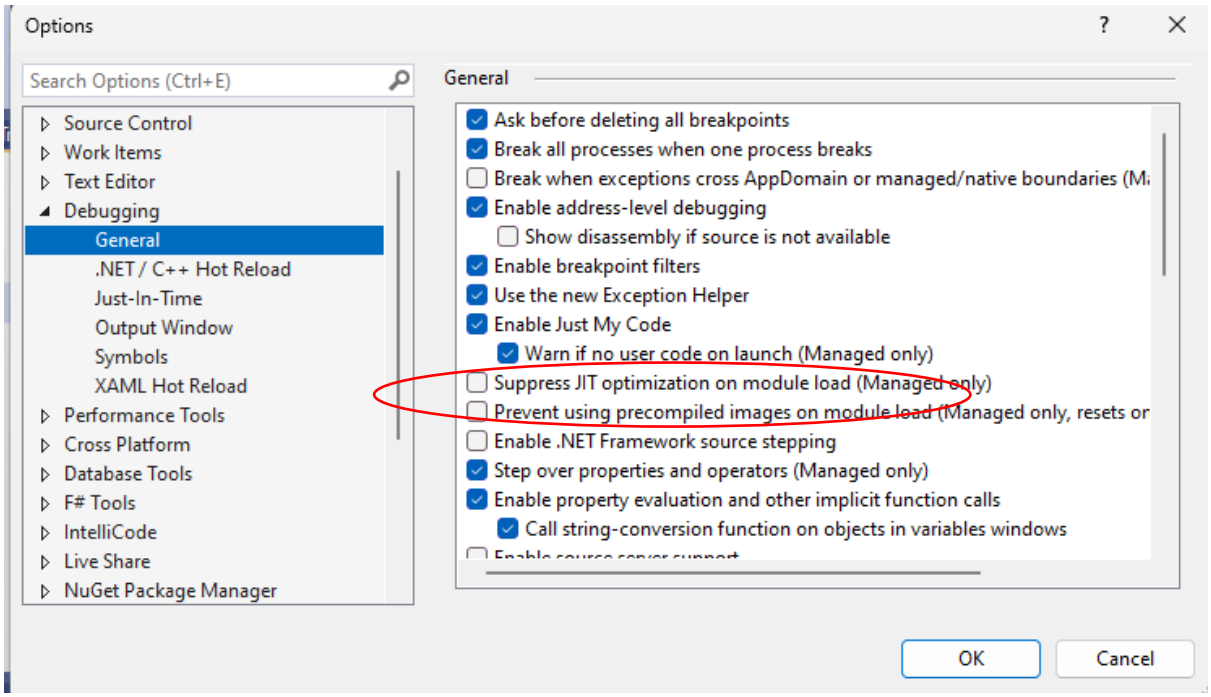
[78 @ 36:37]

This demonstration traces through part of test12 of the PyrrhoTest program using the Visual Studio debugger. In Visual Studio, open the PyrrhoSvr solution in the src\Shared folder of the distribution. Set the debug properties of the PyrrhoSvr project to have -d:\DATA in the Command line arguments.



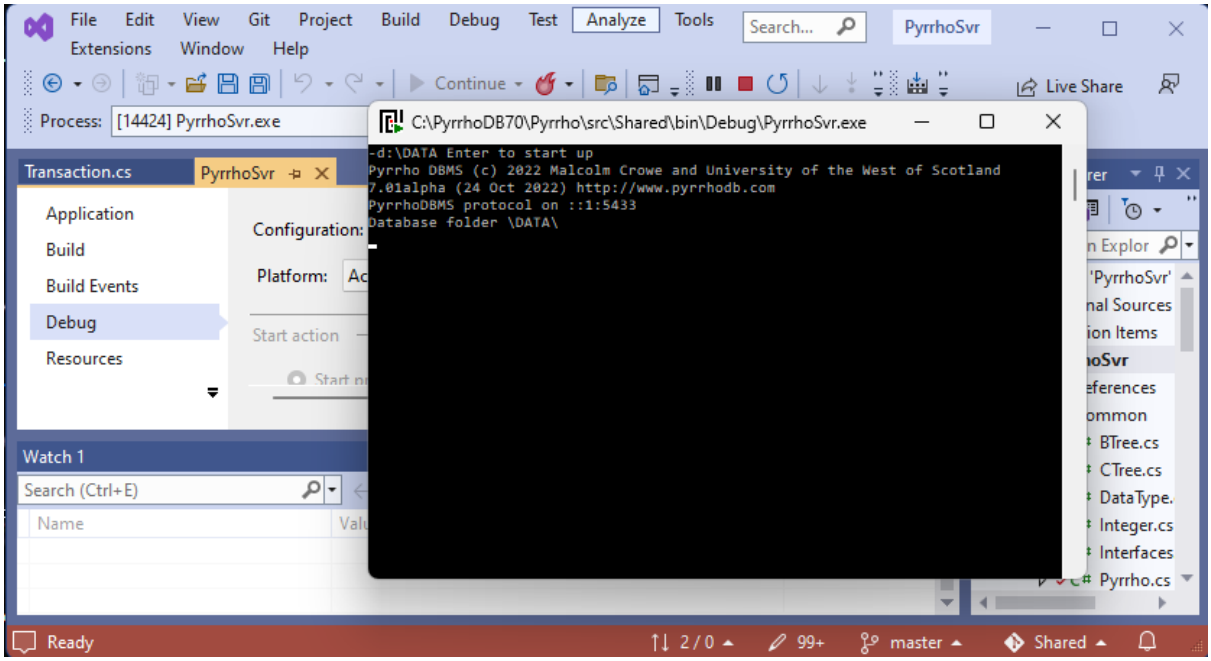
[79 @ 37:11]

From the Debug menu, select Options.. and ensure that the “Step over properties and operators (Managed only)” in Debugging/General is checked. Click OK.

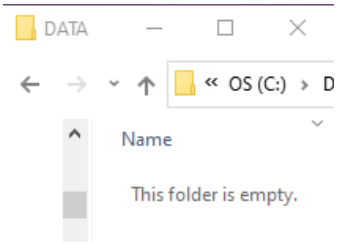


[80 @ 37:26]

Now click Start in the debugger, and in the popup command window, click Enter. We hide this window because it is not going to do anything interesting during this demo.



We want to ensure that the database folder that we are using for our databases is currently empty before we start the demonstration.

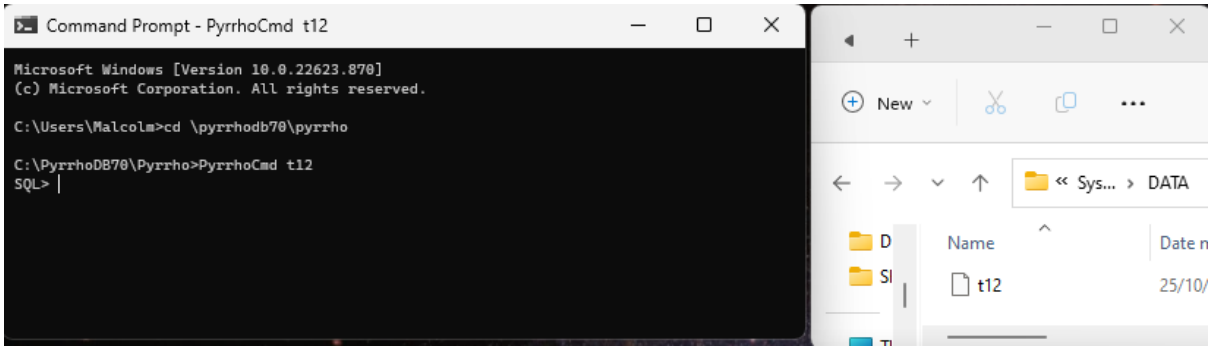


Creating a View

[81 @ 37:48]

In a command window set to the distribution folder, start the command line client PyrrhoCmd for database t12. We see that the database is created by the server.

`PyrrhoCmd t12`



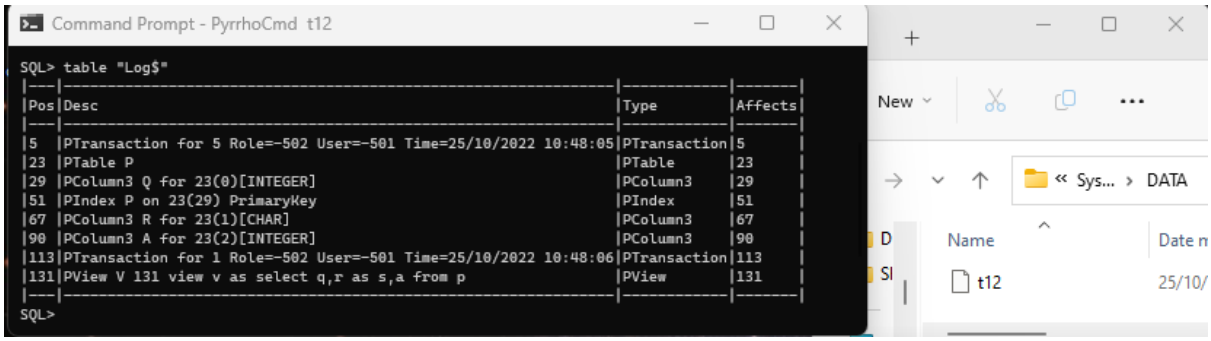
[82 @ 38:00]

In the command window, enter commands to create a table P with three columns, and a View which uses this table, renaming a column.

```
create table p(q int primary key,r char,a int)
create view v as select q,r as s,a from p
table "Log$"
```

We also want to look at the transaction log.

[83 @ 382:21]

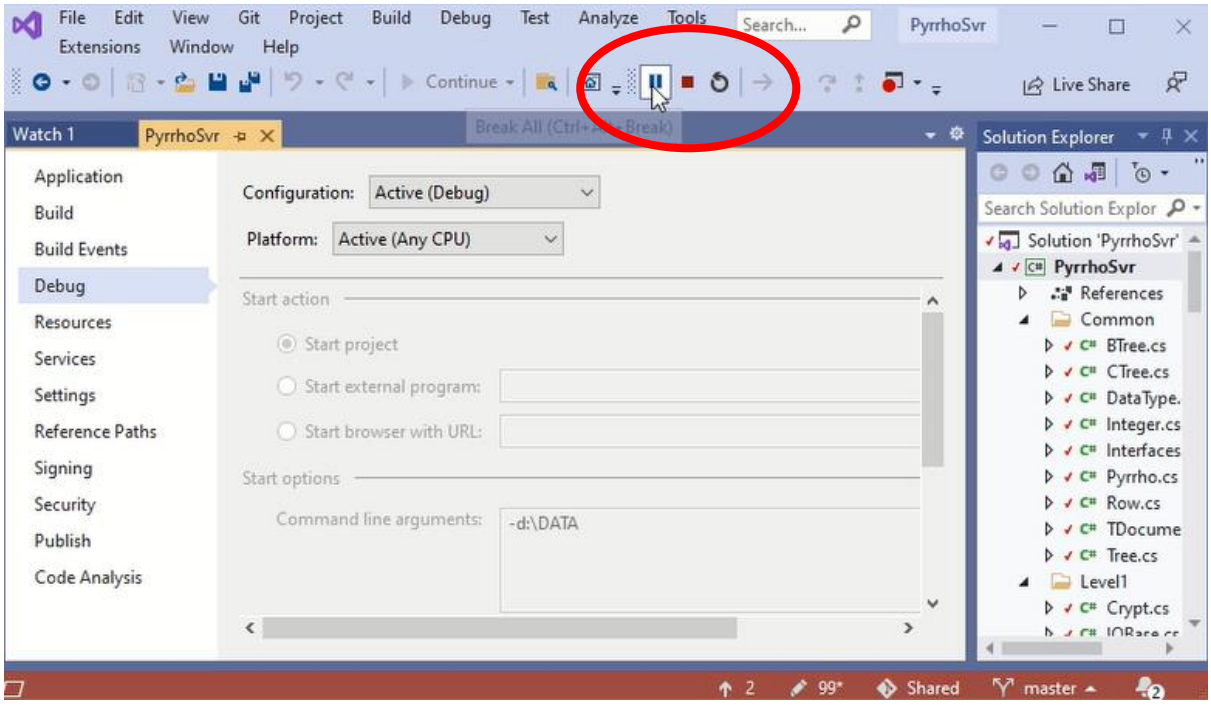


In the log file, we see that the View definition in the database file is just recorded as the source of the select definition of the view.

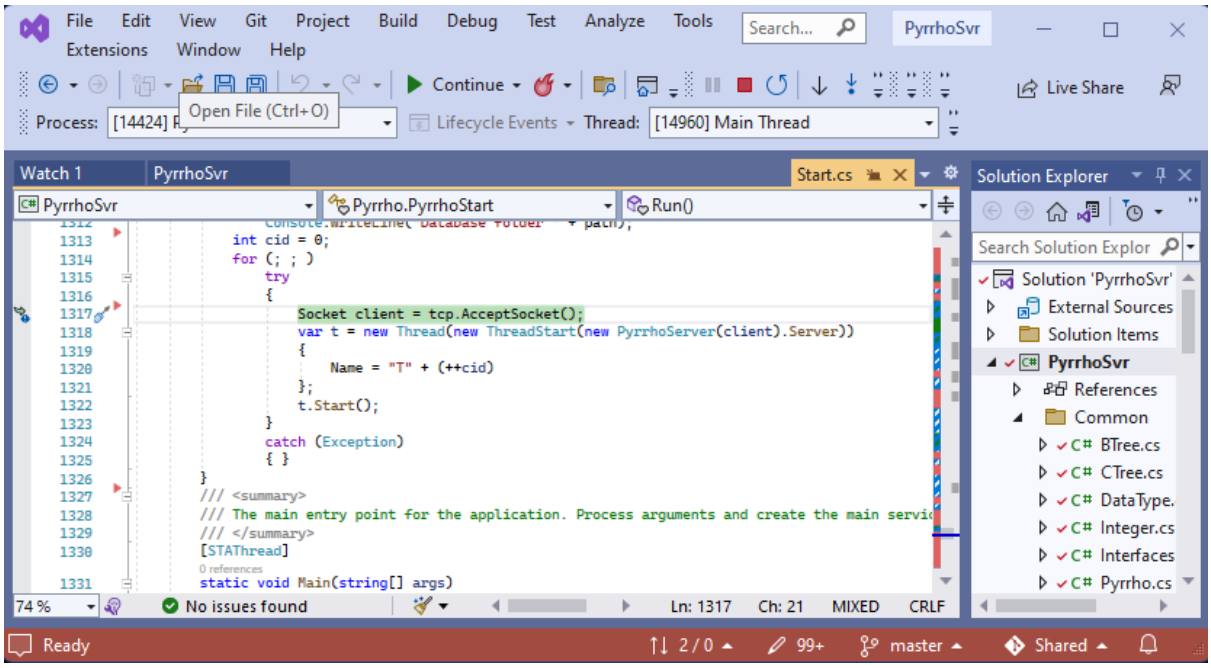
When the server processes this, on load or on definition, it creates some compiled components, so that this statement doesn't have to be parsed every time it is used.

[84 @ 38:44]

Let's pause the server so we can look at the precompiled objects.

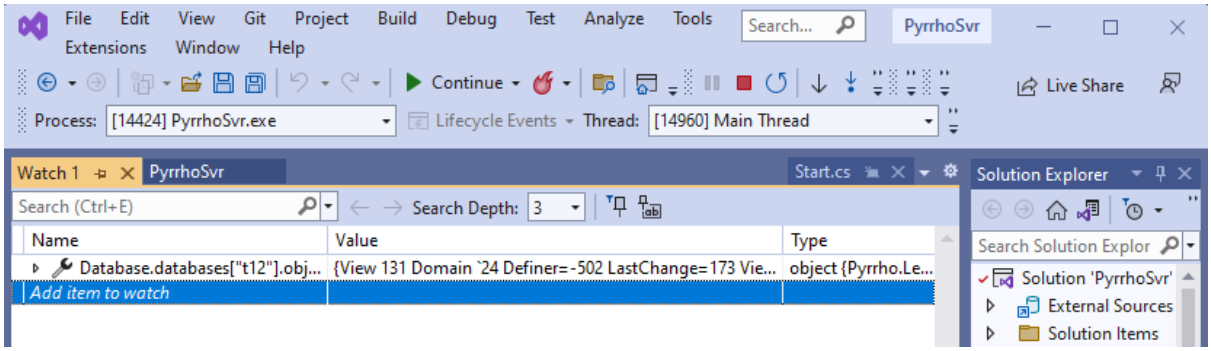


Notice I have docked the Watch window.



[85 @ 38:54]

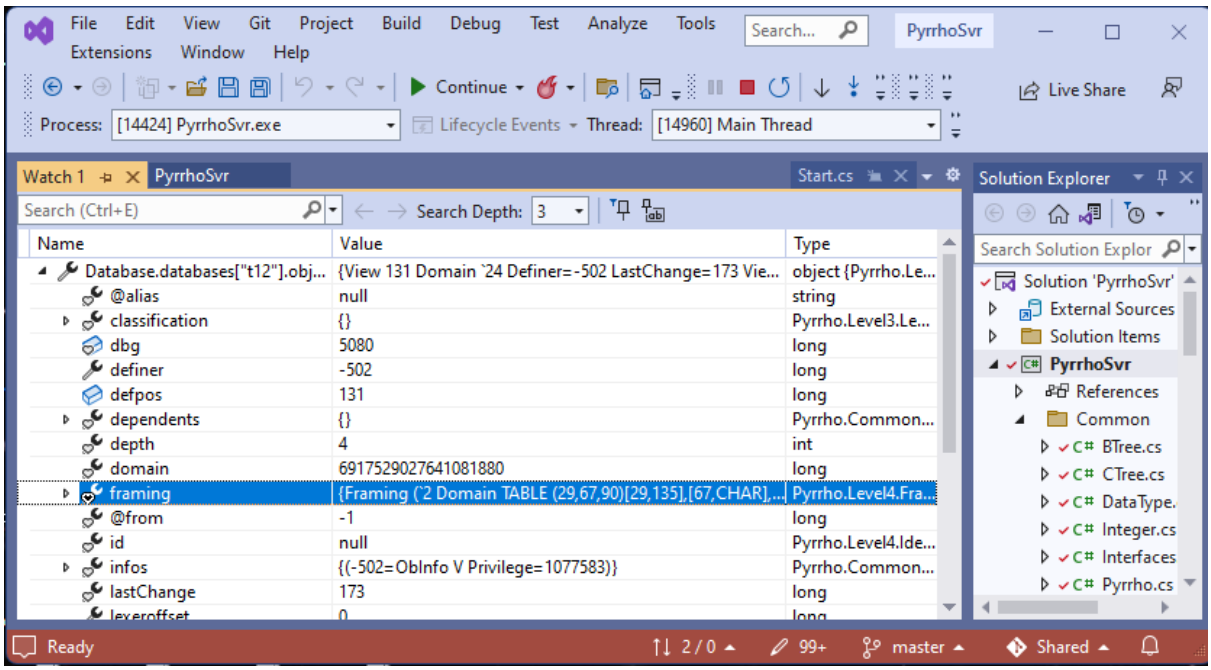
In the Watch window, examine Database.databases["t12"].objects[131] the View position.



We see that there is a View defined in the Value.

[86 @ 39:11]

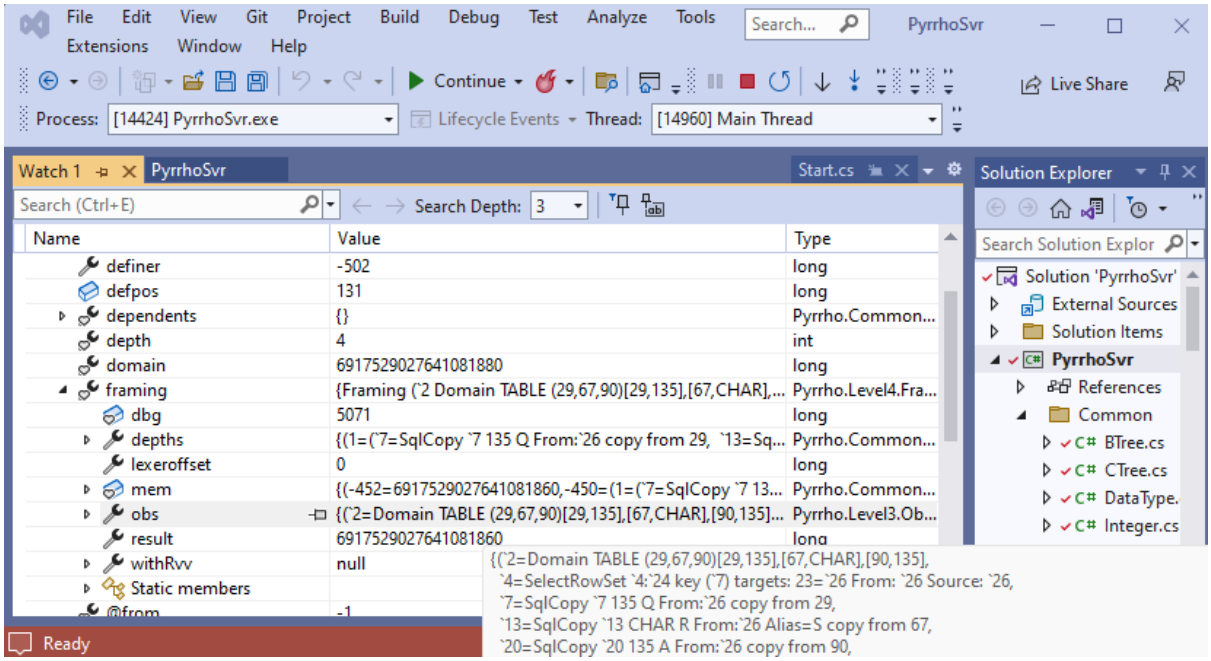
Let's expand that (it takes the server a moment to do it),



and we see that there is a Framing field, and this contains the compiled components of the definition.

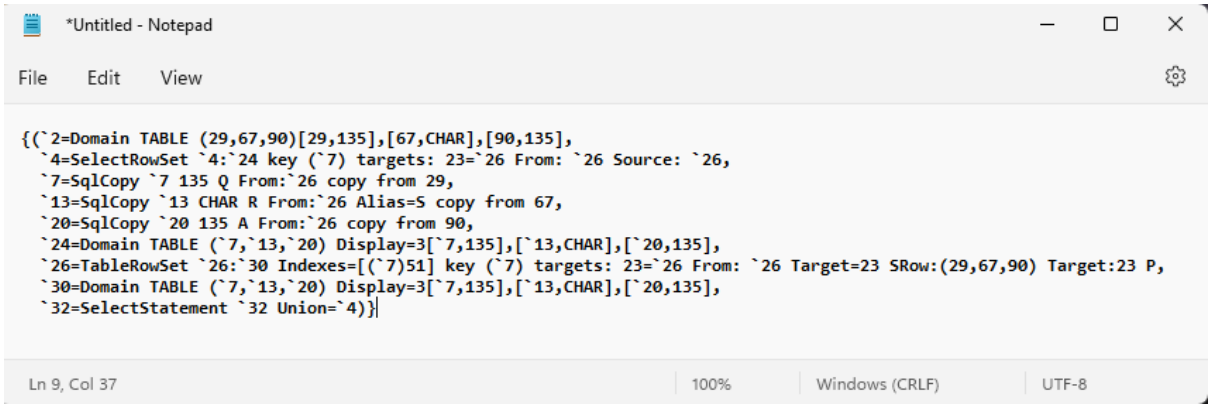
[87 @ 39:25]

Expand the framing, Right click on obs, and Copy its value.



[88 @ 39:33]

Open a Notepad window, and click Paste. We see there is a fair bit in the framing field. We don't need to look at all this detail.

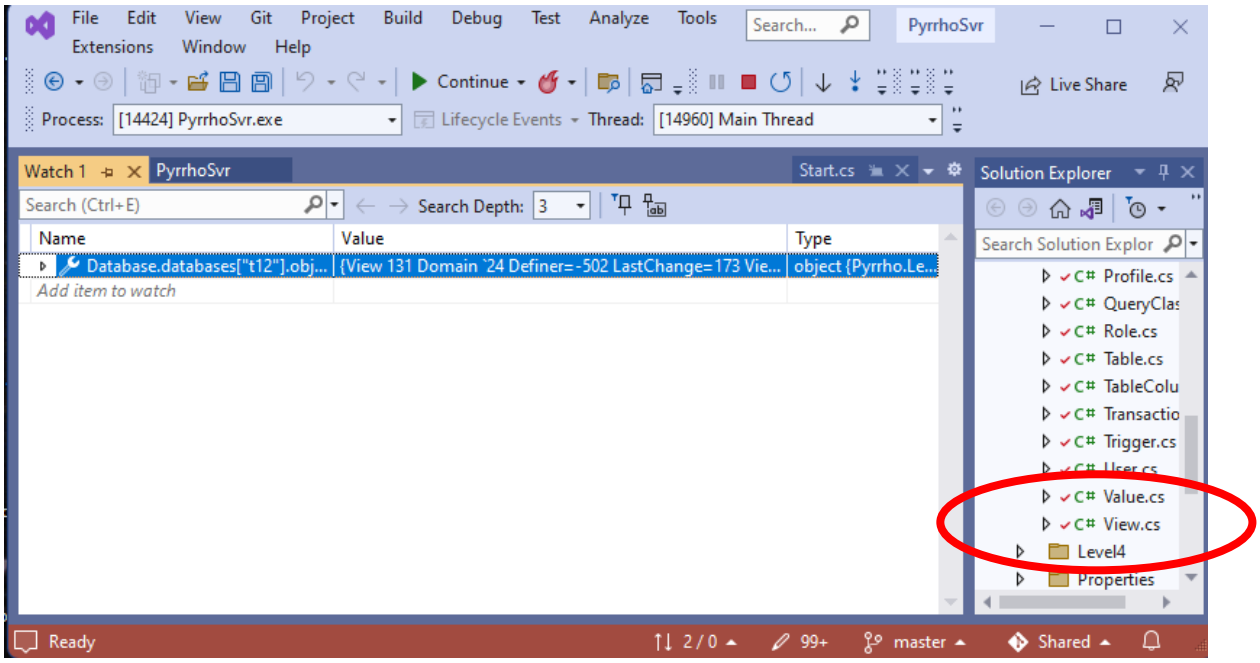


Objects in the framing have positions notated as `1, `2 etc. These are in the uid range reserved for compiled objects: from 0x6000000000000000 to 0x6fffffffffffffff. `1 is a readable version of 0x6000000000000001, etc.

We notice straightaway that the Domain `24 of the View and the domain `2 of the table don't match. There is a relationship, as the types of the columns match, but the uids are all different. In the table, they are of the columns, but in the View `7, `13, and `20 are some of the compiled objects. These positions (like file positions) are assigned when the database is loaded. However, a view might be referenced in more than one place in a complex SQL statement, so each reference will be instanced, to use per-reference uids in the heap uid range.

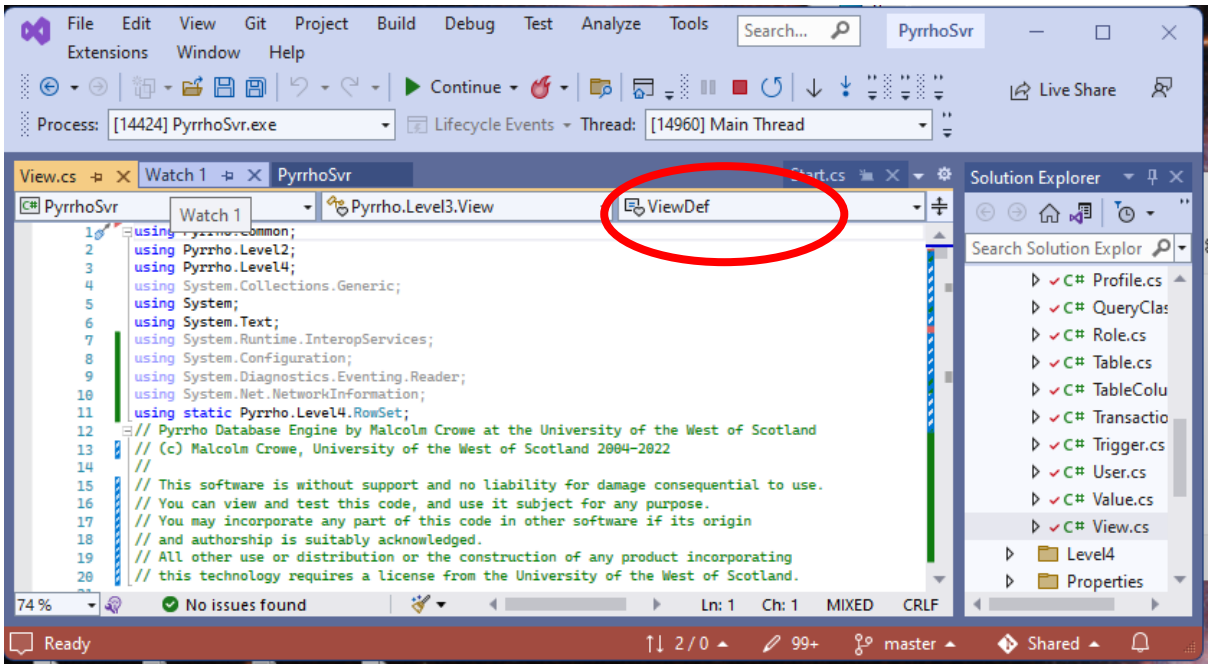
[92 @ 40:41]

Now, in Solution Explorer, scroll down to the end of the Level 3 folder, (I have collapsed the View object in the Watch window to make space for the next bit.)

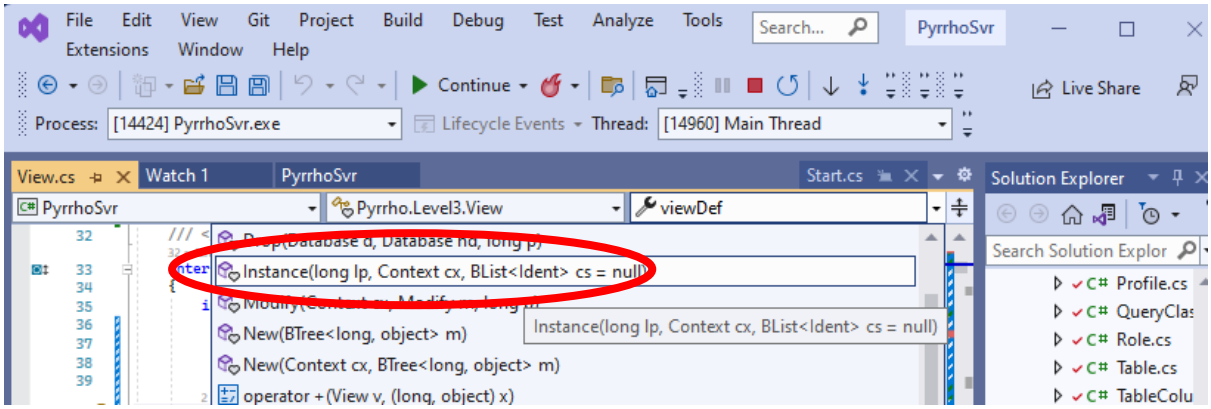


Double-click View.cs

[93 @ 40:54]

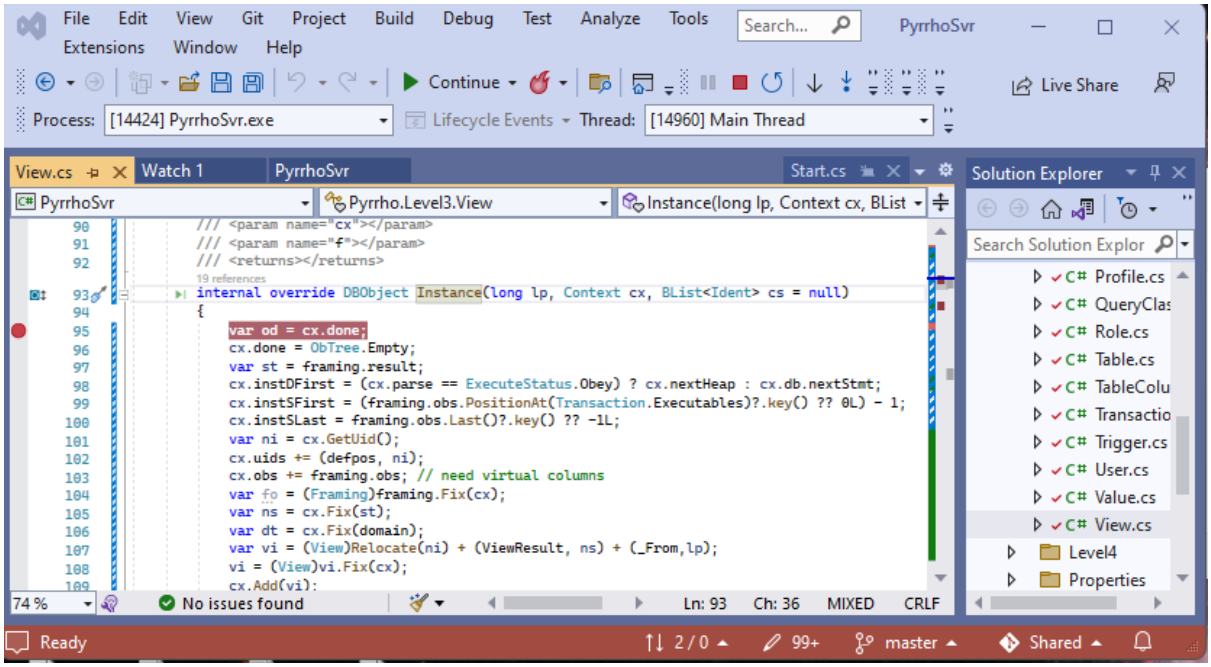


Find the Instance() method in the Targets list, and click on it,



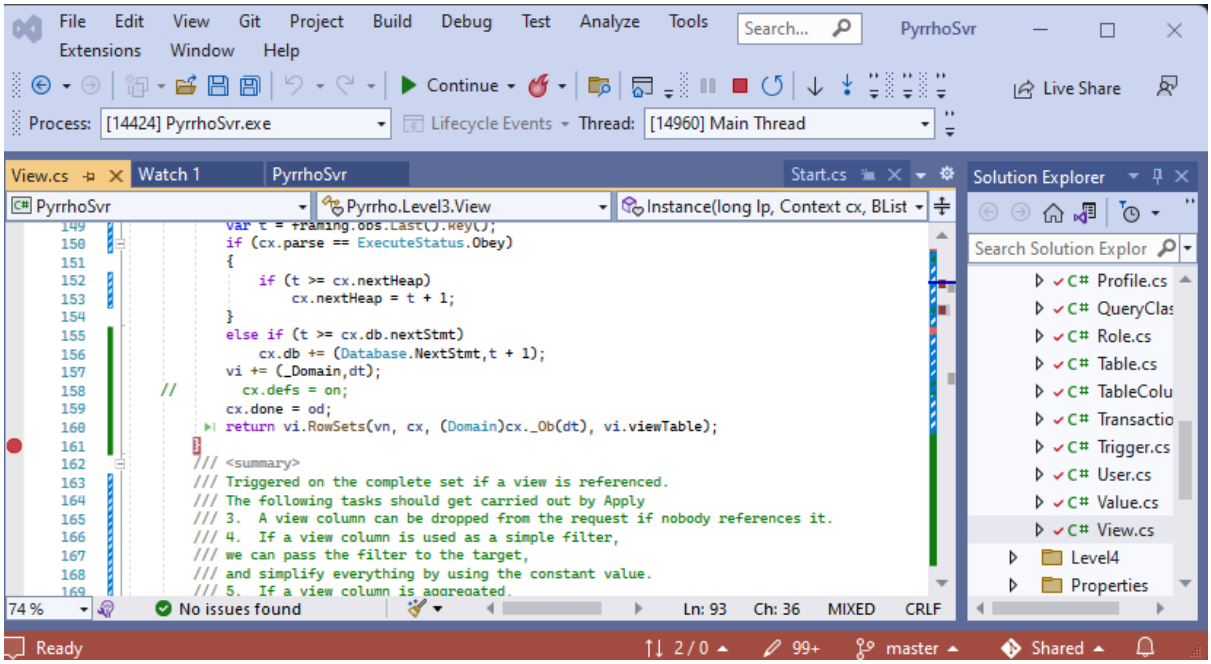
[94 @ 40:59]

and set breakpoints in Visual Studio at the start and end of the View.Instance() method (line 95 and of View.cs):



[95 @ 41:09]

and line 161:



at the end of the Instance method. The View.Instance() method responds to a reference to the View.

[96 @ 41:22]

Allow the server to Continue execution.

[97 @ 41:29]

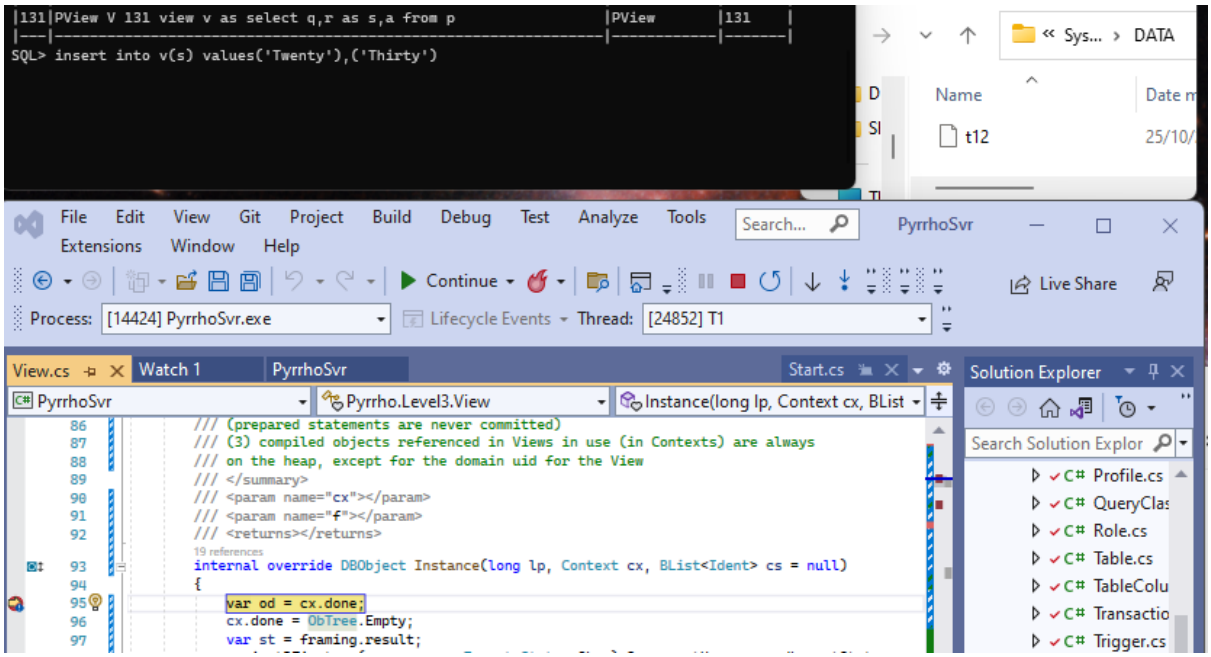
Let us use the following insert statement to add some rows *to the table P*:

```
insert into v(s) values('Twenty'),('Thirty')
```

This is obviously not what views are normally used for, but in this demonstration, we show that we have updatable views. Views can be used, provided the permissions are set correctly, to make modifications to the underlying tables.

[98 @ 41:52]

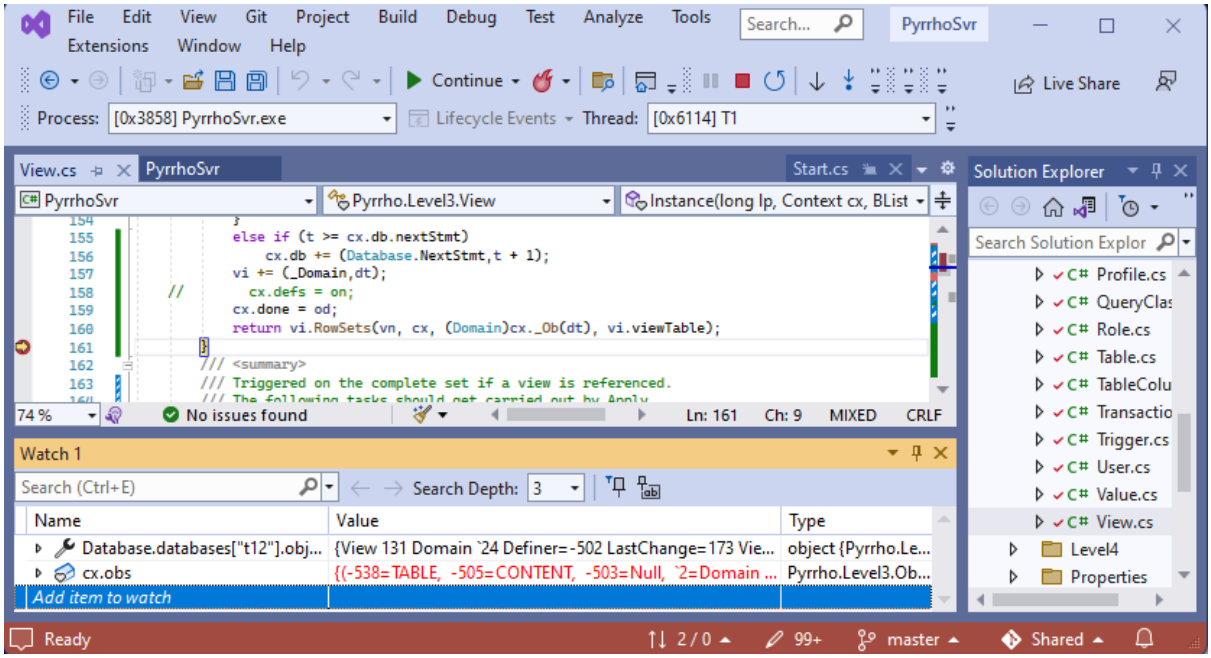
When we click Enter for this insert statement, Visual Studio stops at the break point.



[99 @ 42:11]

As implied above, instancing of the view occurs when the Parser encounters a reference to the view. The only interesting parameter here is lp, the lexical position of the reference. Selecting hexadecimal display in the right-click menu on lp, we see it is 0x500000000000000f which we write as #15. The reference to v in the insert statement occurred at character position 15.

It is the very first object encountered by the parser, so the context is currently empty (cx.obs is {}). We will use the Watch window to see what objects the parser constructs during the Instance method: click continue:

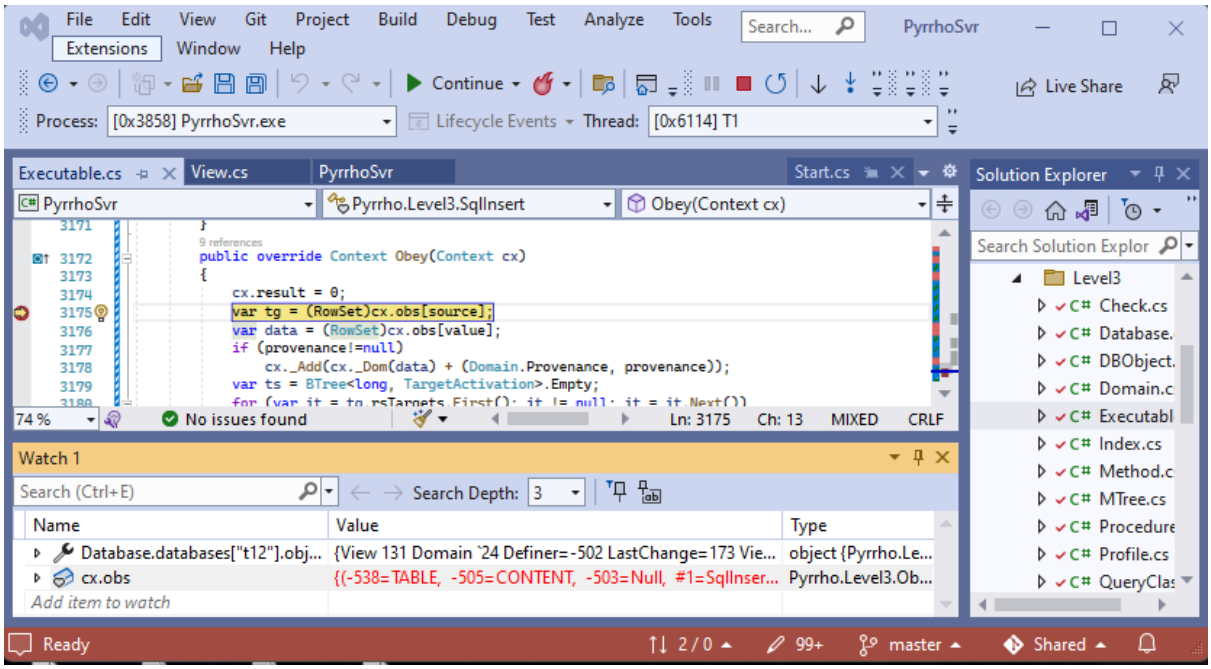


Right-click the cx.obs value, and copy the value to replace the contents of our Notepad:

In addition to the compiled objects we now have the instanced view. %0 and %1 are 0x7000000000000000 and 0x7000000000000001 respectively. (We will soon see !0 which is 0x4000000000000000.) These are long integers unlikely to clash with file positions. It appears so far that everything has been set up for a SELECT operation. We are performing an insert on V, but the above work makes it easy to insert directly into the TableRowSet %25, as we will see.

[100 @ 42:24] move on in the video to slide 112!

Place a breakpoint in Executable.cs at line 3175: in SqlInsert.Obey().



Again copy the value of cx.obs to replace the contents of our Notepad: the new lines are

```
#1=SqlInsert #1 Target: %3 Value: %33 Columns: [%12],
#18=#24,#35,
#24=SqlRow #24 Domain %34 ,
#25=Twenty,
#35=SqlRow #35 Domain %35 ,
#36=Thirty,
```

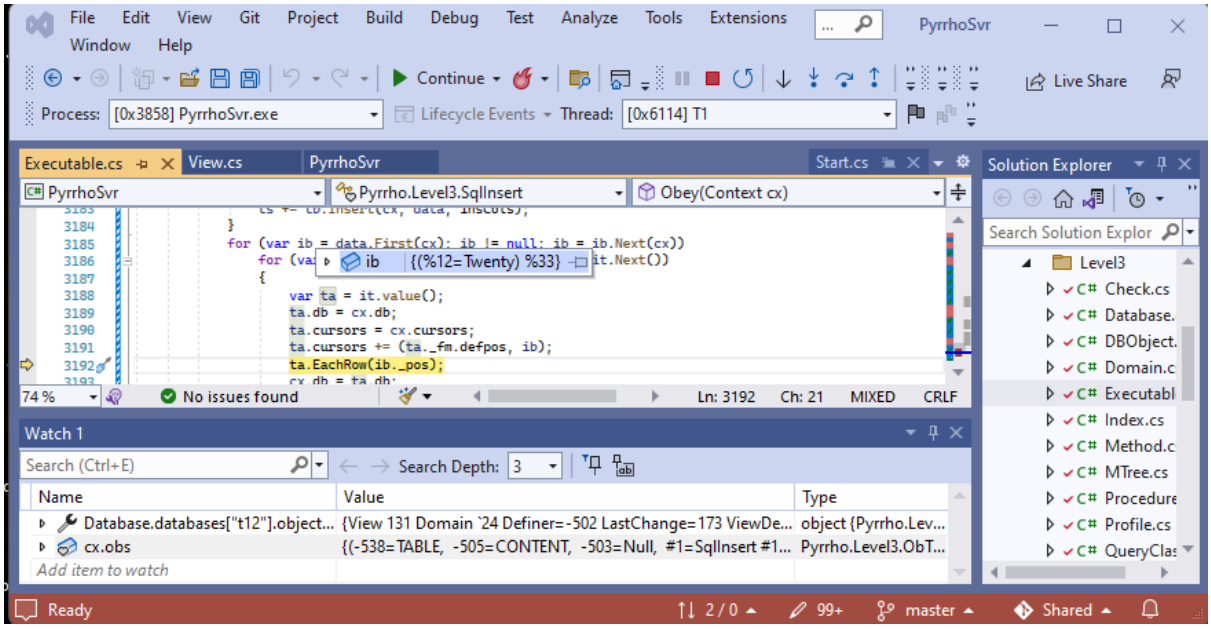
```
%32=Domain TABLE (%12) Display=1[%6,INTEGER],[%12,CHAR],[%19,INTEGER],
%33=SqlRowSet %33:%36 targets: 23=%25 SqlRows [#24,#35],
%34=Domain ROW (#25)[#25,CHAR],
%35=Domain ROW (#36)[#36,CHAR],
%36=Domain TABLE (%12) Display=1[%6,INTEGER],[%12,CHAR],[%19,INTEGER])}
```

It is not hard to see that the SqlInsert statement targets the SelectRowSet %3 from before, and has two SqlRows to insert into it.

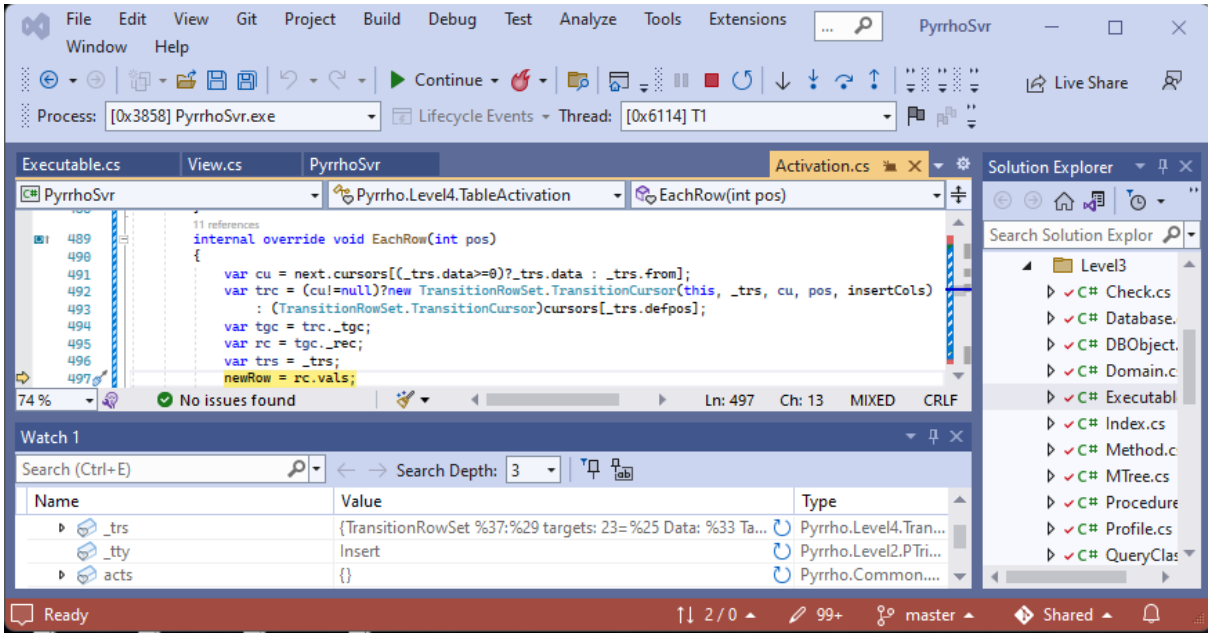
[113 @ 45:25]

StepOver to line 3177. We see that tg is the SelectRowSet %3, and data is the SqlRowSet %33. In case the trigger machinery is needed, a TableActivation is constructed at line 3183 which we will see in action at line 3192. So set the next breakpoint at line 3192.

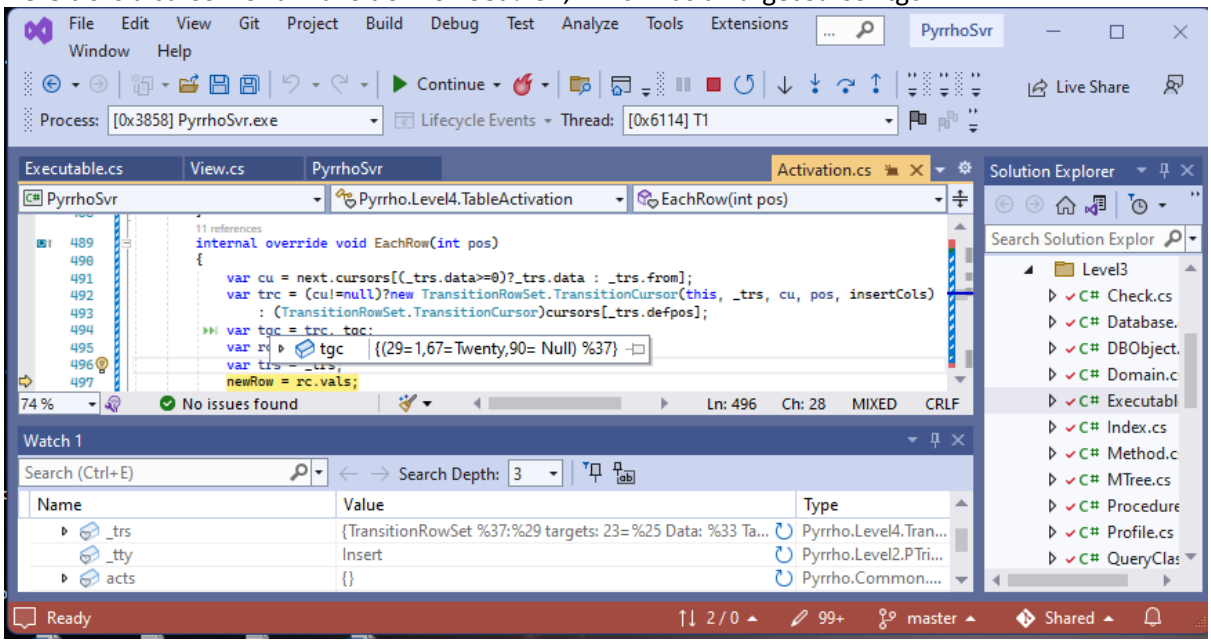
[117 @ 46:18]



ib is a cursor in the data rowset, and the TableActivation ta calls EachRow to insert a row into the target. StepInto this, and step over to line 497 in Activation.cs:

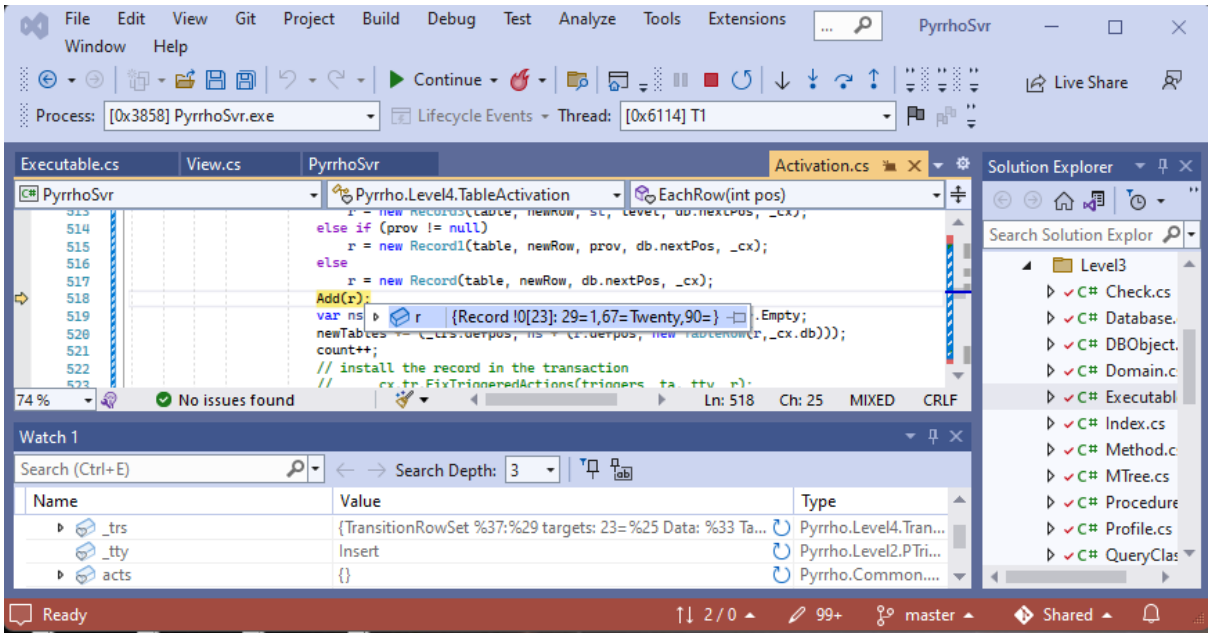


The TableActivation uses a thing called a TransitionRowSet, a concept defined in the SQL standard. Here trs is a cursor for a TransitionRowSet %37, which has a TargetCursor tgc:

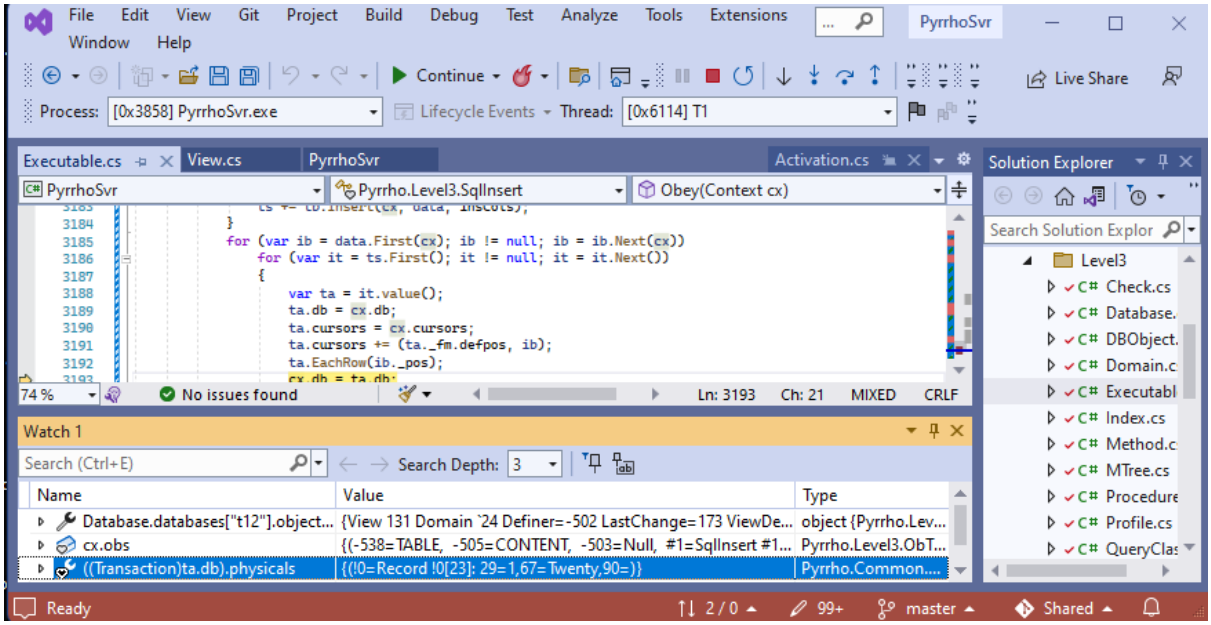


[114 @ 45:35]

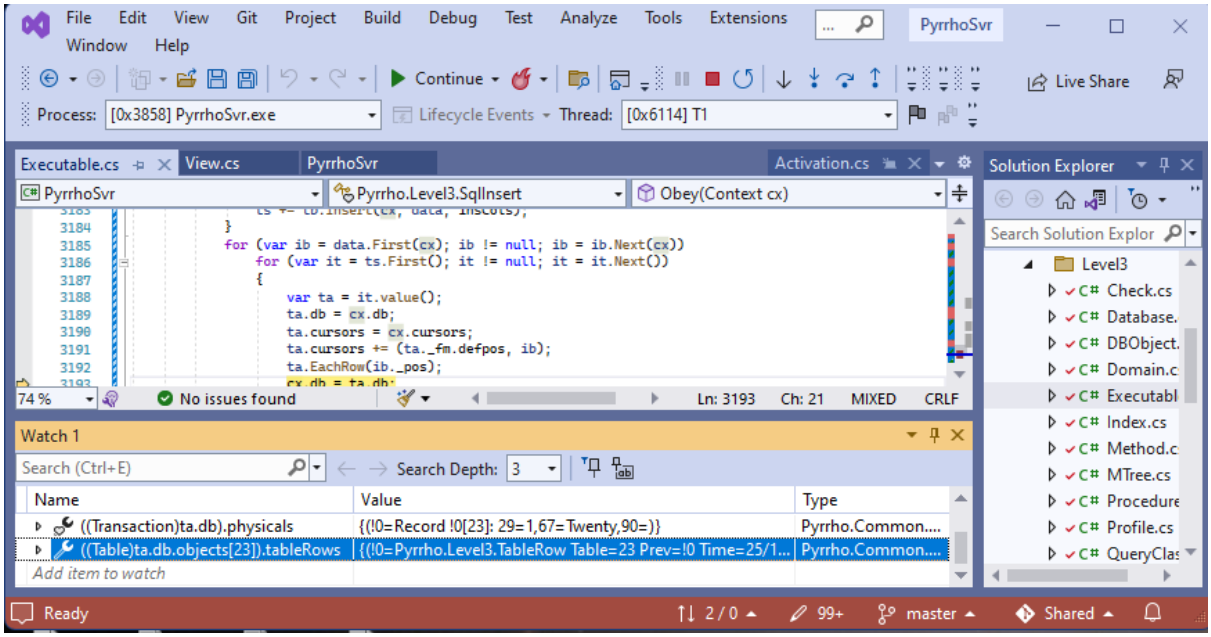
The TargetCursor has the column uids for the target table P, and column 67 holds the value Twenty from the SQL insert command. The server has supplied an autokey value 1 for the primary key column 29. At line 517 the new Record is constructed:



From now on it is all about installing this record in the Transaction. We are at line 518, where Add() adds the new record r to the TableActivation, so that when we Step Out to line 3193 of Executable.cx we can see in the Watch window that that the physicals now include the new Record, awaiting a future Commit().



But the Record has already been Installed in the table and its index in transaction ta.db, and the new table is also installed in the transaction as object[23]. Remember that this does not alter the Database t12, which will only be updated on commit. Line 3193 gets the context to adopt the new transaction value.



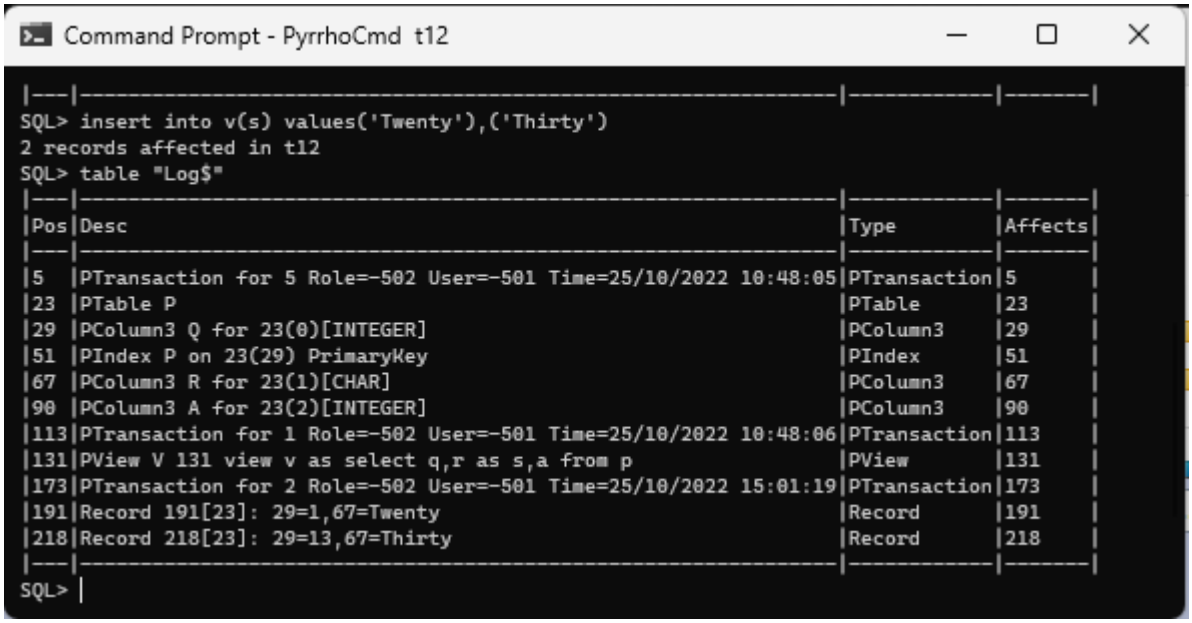
[135 @ 49:26]

We are still in the loop in SqlInsert, and the next tiume around the loop we will add the second new Record, to the physicals, and the transaction's structures. Continue to Step over as you wish, or just click Continue;

[149 @ 52:05]

The successful commit is reported to the client. Check the new entries in the transaction log with

`table "Log"`

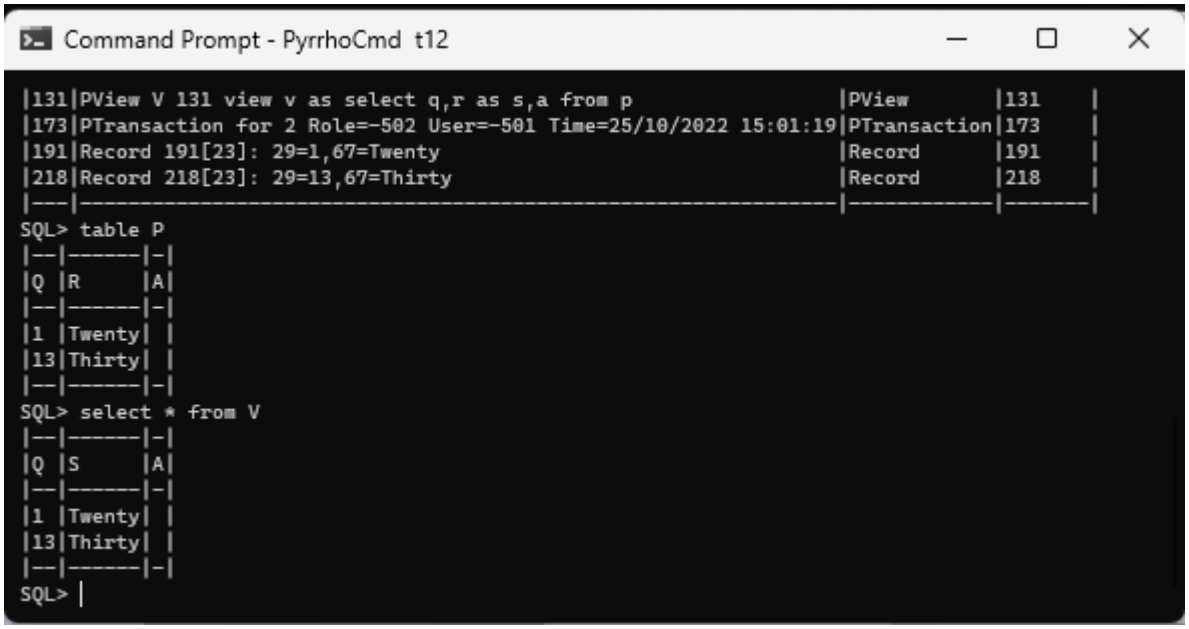


The important aspect in the above is that the changes are made to the Table, not the View. The View contains no data.

[150 @ 52:12]

Confirm the new contents of the table P (this is the real target of the INSERT statement, not V!)

`table P`



This concludes the demonstration.

As an exercise, trace through the operation of updating a join.