Demo 3: Updatable Views
Malcolm Crowe, 11 May 2021

[Slide 78 @ 36:05 of the video]

This demonstration explores the operation of RowSets

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.

[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.
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

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

```sql
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.

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.

Let’s pause the server so we can look at the precompiled objects. Notice I have docked the Watch window.
In the Watch window, examine `Database.databases["t12"].objects[155]` the View position.
We see that there is a View defined in the Value.

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.

Right click, and Copy the value of the framing field.

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.
All we want to do here is to see that the Framing is in four parts. We add a little white space in the next slide.

[89 @ 39:49]

Just now, we will delete everything except the first two lines, which are the details for table P.

[91 @ 40:13]
Let’s also paste the View definition from the Watch window in here, so that we can see what’s going on.
We notice straightaway that the Domain of the View and the domain 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 these are some of the compiled objects.

[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,
and set breakpoints in Visual Studio at the start and end of the View.Instance() method (line 143 of View.cs):

```csharp
    /// in method OnInstance. Once obuids and rsuids are built we call Fix.
    137    /// <summary>
    138    /// The obuids and rsuids are built from 
    139    /// ob = obuids; 
    140    /// rs = rsuids;
    141    internal RowSet Instance(Context cx, RowSet f)
    142    {
    143    var oor = cx.rsuids;
    144    var st = framing_result;
    145    cx.obuids = @tree<long,long?,Empty> from
    146    cx.rsuids = @Tree<long,long?,Empty> from
    148    cx.st = framing_result;
    149    ob = obuids;
    ```
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.
The Instance() method has a RowSet parameter RowSet f (its uid identifies the position of V in the command): it is a VirtualRowSet.

[99 @ 42:11]

Use the Watch window to examine f. Right-click and copy its value into our Notepad.

We see this is a VirtualRowSet. Paste the value in the Notepad,

[100 @ 42:24]
This domain seems different again, and the notation needs explanation. As mentioned, #13 is V’s position in the command, and #15 S’s position. These notations stand for long integers above 2^60 0x500000000000000d and 0x500000000000000f respectively, and %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.

In the Watch window, expand f to get a closer look at the domain. Paste its value into the Notepad.

We notice that not only are the uids different, they are also in a different order. Continue to the next break point.
We are at the end of the Instance() method. Now examine the return value from the Instance() method in the Watch window.

r is a SelectedRowSet. Right-click and copy its value into our Notepad:

This completes the analysis: We can now see that the SelectedRowSet has a map taking the uids %0, #15, %1 to the table columns 43, 93, 115 of P (not V).
As its name implies, SelectedRowSet is normally used for retrieving data but we can use it for insert, update and delete also.

Back in Visual Studio,

We see that Instance() was called from View.RowSets().

We are now back in From.RowSets().
Again we have just finished this RowSets call. Step Over

[109 @ 44:44]

This takes us back to Parser.ParseSqlInsert.

Step Over until we get to line 6626

[110 @ 45:00]
At line 6626, Step Into Transaction.Execute()

[111 @ 45:05]

In Transaction.Execute(), Step Over until line 322

[112 @ 45:17]
At line 322, Step Into Obey()

[113 @ 45:25]

In Sqlinsert.Obey, step into Insert()
This is where the work is done.

Inserting into a base table often involves Triggers. Triggers are managed by Activations. Step Over the creation of the TargetActivation, to line 1328.
At line 1328, use the Watch window to examine ta

The source code said this would be a TargetActivation. It is a subclass, TableActivation. Expand it.

The TableActivation uses a thing called a TransitionRowSet, a concept defined in the SQL standard. Copy its value into our Notepad.
We see the TransitionRowSet knows it has Data, VALUES is at position 18 of the INSERT statement.
Back in the Insert method, start traversal of the data. Step Over the call to First().

and use the Watch window to see the value of Cursor b is \{(#15=Twenty) #18\}. 

[121 @ 47:00]
We recognise this value from the INSERT statement. Copy its value into our Notepad.

Step over to line 1335,
and step into `ta.EachRow()`.

[124 @ 47:21]

In `Activation.cs` `TableActivation.EachRow()`,

Step Over to line 386:

[125 @ 47:29]
and examine tgc in the Watch window.

[126 @ 47:36]

It is a TargetCursor \{(43=1,93=Twenty,115=Null) %10\} that has filled in a suitable value for the primary key column 43.

[127 @ 47:46]

Step Over a bit more. We have no triggers in this example, so after refreshing the value of the target cursor, Pyrrho constructs a record. We will stop at line 411.

[128 @ 48:00]

[129 @ 48:16]
Use the Watch window to examine r.

From now on it is all about installing this record in the Transaction.

Back in Visual Studio, Step Into Add(r):
In Context.Add(), step over a few times,

then step into db.Add() to add our Record to the Transaction.

In Transaction.Add():
ph is the Record we just made. We see how the Physical is added to the physicals list. Step Over line 158.

[134 @ 49:13]

At line 159, we are about to call ph.Install(). But let's use the Watch window to examine ((Transaction)cx.db).physicals.

[135 @ 49:26]
It is a list containing the new Record. Now Step Into Install():

We see we are adding a Record to a base table tb. Step Over to line 241
Note we are adding a row to table `tb`. Step into `AddRow()`

Record.\texttt{AddRow} is all about building a TableRow and adding it to any indexes that are around. Step over down to line 231.

At line 231, we use one of our addition operators to add the key and record position to the Index `x`.

Hover over `x`, to see it has a new value and contains our key.
The next line adds this changed Index to the database.

[141 @ 50:28]

After a journey around the loop we reach the return statement at line 234 with the new TableRow.

Let’s examine now with the Watch window. We see it has our new values vals. The huge numbers called ppos etc are all 10, i.e. 0x4000000000000000. This is a position for an uncommitted database object.

[143 @ 50:55]

After adding this new row to the table, we add the updated table to the transaction.
After adding this new row to the table, we add the updated table to the transaction.

[144 @ 51:06]

Step Over some more,

and a few more times to get back to EachRow.

[145 @ 50:53]
In EachRow() we have just finished Add for our Record. Step over to line 417.

[146 @ 51:01]

At line 417 we copy our transaction from the TargetActivation into the parent Context. (Just an assignment of a 64-bit pointer.)

[147 @ 51:41]

Step over a few more times to return to SelectedRowSet.Insert().
That finishes EachRow for this row of the data. Step Over to see the Cursor for the next row of the data.

[148 @ 51:24]

Hovering over b, we recognise the value Thirty here. The same thing will happen for this row, and then the transaction will Commit(), so just click Continue to finish.

[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.