In the second demonstration, we look in detail at the Commit() method for a transaction, and the detection of conflicts.

At the start of Transaction Commit, there is 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**

The slide shows two client command windows, the database folder, and Visual Studio. We will get the Visual Studio debugger to run the server for us, as this will enable us to look at the validation step of Transaction.Commit().

Visual Studio has opened the solution in Pyrrho’s src\Shared folder, and I have set the database folder to \DATA for convenience, using the Debug properties of the PyrrhoSvr project. In the command window that the start-up creates, I clicked the Enter key to start the server as in demo 1. I have minimised the server window to get it out of the way.

We confirm that the database folder I have nominated is empty.

I have used Visual Studio to open the PyrrhoSvr solution in the src\Shared folder of the distribution, setting the PyrrhoSvr>Properties>Debug>Command Line Arguments for the PyrrhoSvr project to -d:\DATA.
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.

[32 @ 16:44]

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.
From now on, the folder window is 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:

```sql
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.

[35 @ 17:44]

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>.

[36 @ 18:21]

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.

[37 @ 18:40]

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.

[38 @ 19:00]

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 188 at the start of the Transaction.Commit() method.

Click the mouse in the left margin to set a break point at line 190 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”.

We also look at `cx.rdC[23]`, which is to do with the ReadConstraints for this particular Transaction. In this case there aren’t any, as this transaction hasn’t read anything. (In the next experiment we will see it is the other way round.) Step Over a few times to get to line 208.

Step Over a few times to get to line 208.
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 the locking the FileStream.)

[44 @ 20:27]

Step Over to line 213, 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.

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

Set a breakpoint at line 230.
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 Into the Conflicts() method
When you click Step Into the debugger stops during the evaluation of `b.value()`, so Step Out, and then immediately Step Into:
In the Conflicts method, we see that **that** is our Delete, and **this** is the Update. We are looking at the Update that was done by the green window, and comparing it with our Delete record.
Step Over: we reach line 95.

For our delete to conflict with an update, it just means that the position we are trying to delete matches the position that has been updated.

In this case we find there is a 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.

[53 @ 23:01]

Part 3: A Read-Write Transaction

For the second part of the demonstration, for simplicity we will restore the state we had at slide 35: stop the server, delete the database, and repeat the set-up to that point.

[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
```

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.

```
Add or restore the break point in the Transaction.Commit() method.
```
and in the blue window,

commit

On the commit command, we hit the break point.
Open the Watch window again.

[58 @ 24:03]

If we look at the Watch window that we had before, we can see that the physicals list is empty this time, but the read constraint has got something in it. (We haven’t fetched since yet.)
Expand the watch entry for cx.rdc[23], and we see the check here is a CheckSpecific, that will detect any change to a set of specified rows:

Since rdC is not empty, let’s put a break point at line 223
The second time this is hit, ph is the Update that was made by the green window. (The blue window just made a select.)

Step Into the Check() call.

[62 @ 24:55]
Step Over a few times to get to line 63:

and Step Into the check.Check() method:

[63 @ 25:12]
This looks at the list of records of the CheckSpecific, the records that were read by the blue window, and it contains the record that has just been updated, so an exception occurs. Continue from this point.

[64 @ 25:28]

A suitable report is sent to the blue client, and the transaction is rolled back.
This concludes the demonstration of transaction conflict.

[Ends]

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
```
SQL> select * from KOL where A=32
A | B
---|---
1 | 2

1 record(s) affected in 0.000 sec

SQL> select * from KOL where A=42
A | B
---|---
1 | 2

1 record(s) affected in 0.000 sec

SQL> update KOL set B = the product of 6 and 9 where A=42
1 record(s) affected in 0.000 sec