# Course ISE 326: Database Systems Engineering
# Recitation 2 Exercise: Serializability

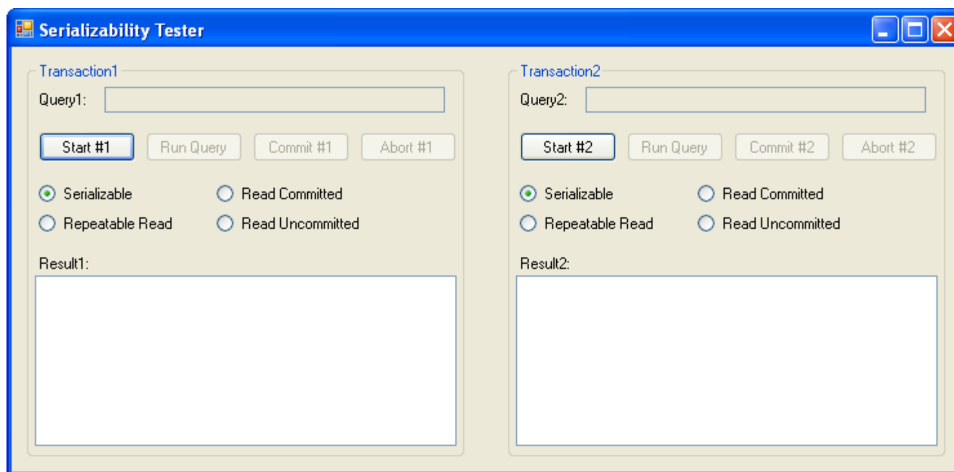
Michael J. May

March 9, 2010

The purpose of this lab is to learn about how the MS SQL Server database provides concurrency control to its transactions. In the lab, you will write code to run two simultaneous transactions with the SQL Server and try different combinations of transactions and protection levels.

## 1 The Tester Window

You will start by making a tester window which looks like the following:




## 2 The Connection

As we saw in the previous lab, the connection string for the MS SQL Server database is:

```
SqlConnection connection1 =
    new SqlConnection("user id=XXXXXX;"
                    + "password=YYYYYY;"
                    + "server=DUGIT;"
                    + "database=ZZZZZZZZZ");
```

Use the connection string to make two concurrent transactions with the database server when the user clicks on either of the Start buttons.

You will need two connection objects.

## 3 Creating a Transaction

You can create a transaction for the connection by using the SQL Server command:

```
SqlCommand command =
    new SqlCommand("SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; BEGIN TRANSACTION",
                   connection1);
```

You can change the isolation level from SERIALIZABLE to other values as we have discussed in class using the radio buttons on the window.

Don't forget to run the command using the following lines:

```
int result =  command.ExecuteNonQuery();
```

Since you run the command on each connection, you will need to do the above twice, once for each connection you created.

## 4 Running Queries

When the user enters a query in either of the connections, you will need to make a new command for the query and read the results. For example, the following code sends a query to the transaction on connection1 and shows its results in the results box. Keep in mind that the code is hard coded for a result with two columns.

```
SqlDataReader reader = null;
SqlCommand command =
    new SqlCommand(tb1Query.Text, connection1);

try
{
    reader = command.ExecuteReader();

    while (reader.Read())
    {
       for (int i = 0; i < reader.FieldCount; i++)
       {
          tb1Result.Text += reader[i] + "   ";
       }
       tb1Result.Text += Environment.NewLine;
    }
    reader.Dispose();
}
catch (SqlException sqle)
{
    MessageBox.Show("Couldn't run the query on Transaction1: "
                    + sqle.ToString());
}
```

The catch is important since not all of the queries that you will attempt in the lab will work.

# 5 Commit and Abort

The Commit and Abort buttons should send Commit and Rollback commands to each connection and close them using the function (for instance on connection1):

```
connection1.Close();
```

# 6 Experiments

When you have written your serializability analyzer, you can perform experiments with the server to see how it behaves.

First, use the Excel file found on the web page of the course to create the Sailors, Boats, and Reserves tables that we have used in the previous semester. If the tables are still in your database from last year, you can skip this step.

Now, attempt the following experiments, first all on Serializable, then on Repeatable Read, Read Committed, and Read Uncommitted:

## 6.1 Experiment 1

1. Start Transaction 1 and Transaction 2
2. Perform on Transaction 1:

   ```
   SELECT * FROM Sailors
   ```

3. Perform on Transaction 2:

   ```
   SELECT * FROM Sailors
   ```

   What happens?

## 6.2 Experiment 2

1. Start Transaction 1 and Transaction 2
2. Perform on Transaction 1:

   ```
   SELECT * FROM Sailors;
   INSERT INTO Sailors (sid, sname, rating, age) VALUES (55, 'Norbert', 8, 30)
   ```

3. Perform on Transaction 2:

   ```
   SELECT * FROM Sailors
   ```

   What happens?

### 6.3 Experiment 3

1. Start Transaction 1 and Transaction 2
2. Perform on Transaction 1:

```
SELECT * FROM Sailors;
INSERT INTO Sailors (sid, sname, rating, age) VALUES (56, 'Artimus', 9, 20)
```

3. Commit Transaction 1
4. Perform on Transaction 2:

```
SELECT * FROM Sailors
```

What happens?

### 6.4 Experiment 4

1. Start Transaction 1 and Transaction 2
2. Perform on Transaction 1:

```
SELECT * FROM Sailors;
INSERT INTO Sailors (sid, sname, rating, age) VALUES (57, 'Tolouse', 10, 50)
```

3. Perform on Transaction 2:

```
SELECT * FROM Boats;
INSERT INTO Boats (bid, bname, color) VALUES (20, 'Tika', 'Fuschia')
```

4. Perform on Transaction 1:

```
SELECT * FROM Boats
```

What happens?

### 6.5 Experiment 5

1. Start Transaction 1 and Transaction 2
2. Perform on Transaction 1:

```
SELECT sname, bid, day FROM Sailors S, Reserves R WHERE S.sid = R.sid;
```

3. Perform on Transaction 2:

```
INSERT INTO Reserves (sid, bid, day) VALUES (85, 102, '1/1/2010')
```

4. Perform on Transaction 1:

```
SELECT * FROM Reserves
```

What happens?

## 6.6 Experiment 6

- Start Transaction 1 and Transaction 2
- Perform on Transaction 1:

  ```
  SELECT * FROM Reserves
  ```

- Perform on Transaction 2:

  ```
  UPDATE Boats set bid = 59 WHERE bid = 58
  ```

- Perform on Transaction 1:

  ```
  SELECT * FROM Reserves
  ```

## 6.7 More Experiments

Try some more experiments on your own to explore the locking policy of the SQL Server instance.

Use the tool to test the definitions of Serializable, Repeatable Read, Read Committed, and Read Uncommitted from class.

**Cascading Aborts** Does MS SQL Server perform cascading aborts? Try to answer this question by setting the transaction isolation levels appropriately and performing a dirty read followed by an abort.