IBM DB2® 9.7

Data Concurrency
Hands-On Lab

Information Management Cloud Computing Center of Competence

IBM Canada Lab
1. Introduction to Data Concurrency

In this lab you will practice with data concurrency and concurrency control in DB2.
2. Objectives of This Lab

After completion of this lab, the student should be able to:

- Understand the semantic differences between Cursor Stability and Currently Committed.
- Understand the differences between Repeatable Read, Read Stability, Cursor Stability and Uncommitted Read.
- Be able to specify different isolation levels for a database at run time using the CLP.

3. Setup and Start DB2

3.1 Environment Setup Requirements

To complete this lab you will need the following:

- DB2 Academic Workshop VMware® image
- VMware Player 2.x or VMware Workstation 5.x or later

For help on how to obtain these components please follow the instructions specified in the VMware Basics and Introduction module.

3.2 Login to the Virtual Machine

1. Login to the VMware virtual machine using the following information:
   User: db2inst1
   Password: password

2. Open a terminal window as by right-clicking on the Desktop area and choose the “Open Terminal” item.
3. Start up DB2 Server by typing “**db2start**” in the terminal window.

```plaintext
db2start
```

### 3.3 SAMPLE Database

For executing this lab, you will need the DB2’s sample database created in its original format.

Execute the commands below to drop (if it already exists) and recreate the **SAMPLE** database:

```plaintext
db2 force applications all
db2 drop db sample
db2samp1
```
3.4 Create and populate a table

We will create a simple table that will be updated during this lab session. The table named “tb1” will be created with a single column named “column1”. We will then populate it with 9 rows with the same value “10”.

1. Run the following commands.

```sql
db2 connect to SAMPLE
db2 "create table TB1 (COLUMN1 integer)"
```

```sql
db2 "insert into TB1 (select 10 from syscat.tables fetch first 9 rows only)"
```

```sql
db2 terminate
```

4. Cursor Stability with Currently Committed

We will now demonstrate the effect of the currently committed feature. To do so, we will simulate a scenario where a potential read / write block can happen when 2 queries are running concurrently. Then, we compare the difference in results and execution time when we toggle the parameter `cur_commit`.

We will use DB2’s command line processor (CLP) to simulate the applications accessing the database at the same time.

4.1 The “Before” scenario: without Currently Committed

4.1.1 Turning off Currently Committed

1. First, we will examine the existing setting for currently committed. Using the terminal, type in the following command. Since we will be using more than one terminal, we'll refer to this terminal as Terminal A.

```sql
db2 get db cfg for sample
```
The `cur_commit` parameter is located near the end of the list. It should display as ON for now, as this is the default for new databases in DB2 9.7.

2. The next step is to disable the Currently Committed semantics. For that, change the value of `cur_commit` to DISABLED using the following command:

   ```
   db2 update db cfg for sample using cur_commit disabled
   ```

4.1.2 Execute a write query in Terminal A

1. In order to mimic the behaviour of a long running transaction, we first need to disable the auto-commit feature, which is ON by default in CLP. When auto-commit is active, CLP automatically issues a COMMIT after every
executed SQL statement. Therefore, we need to disable it so we are able to specify when the transaction will be committed. Enter the CLP prompt by typing the command below. The “+c” option will disable the auto-commit feature for this session.

```
db2 +c
```

2. You can check that the auto-commit feature is off by executing the command below. Since auto-commit is OFF, from now on all SQL statements that you execute will be part of the same transaction until you issue a “commit” or “rollback”.

```
list command options
```

```
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Current Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a</td>
<td>Display SQLCA</td>
<td>OFF</td>
</tr>
<tr>
<td>-c</td>
<td>Auto-Commit</td>
<td>OFF</td>
</tr>
<tr>
<td>-d</td>
<td>Retrieve and display XML declarations</td>
<td>OFF</td>
</tr>
<tr>
<td>-e</td>
<td>Display SQLCODE/SQLOSTATE</td>
<td>OFF</td>
</tr>
<tr>
<td>-f</td>
<td>Read from input file</td>
<td>OFF</td>
</tr>
<tr>
<td>-i</td>
<td>Display XML data with indentation</td>
<td>OFF</td>
</tr>
<tr>
<td>-l</td>
<td>Log commands in history file</td>
<td>OFF</td>
</tr>
<tr>
<td>-m</td>
<td>Display the number of rows affected</td>
<td>OFF</td>
</tr>
<tr>
<td>-n</td>
<td>Remove new line character</td>
<td>OFF</td>
</tr>
<tr>
<td>-o</td>
<td>Display output</td>
<td>ON</td>
</tr>
<tr>
<td>-p</td>
<td>Display interactive input prompt</td>
<td>ON</td>
</tr>
<tr>
<td>-q</td>
<td>Preserve whitespaces &amp; linefeeds</td>
<td>OFF</td>
</tr>
<tr>
<td>-r</td>
<td>Save output to report file</td>
<td>OFF</td>
</tr>
<tr>
<td>-s</td>
<td>Stop execution on command error</td>
<td>OFF</td>
</tr>
<tr>
<td>-t</td>
<td>Set statement termination character</td>
<td>OFF</td>
</tr>
<tr>
<td>-v</td>
<td>Echo current command</td>
<td>OFF</td>
</tr>
<tr>
<td>-w</td>
<td>Display FETCH/SELECT warning messages</td>
<td>ON</td>
</tr>
<tr>
<td>-x</td>
<td>Suppress printing of column headings</td>
<td>OFF</td>
</tr>
<tr>
<td>-z</td>
<td>Save all output to output file</td>
<td>OFF</td>
</tr>
</tbody>
</table>
```

db2 =>

3. Connect to database “sample”.

```
connect to sample
```
4. Before we make any updates to the table, we will do a quick query to observe the current values for column “column1”.

```sql
select * from tbl
```

5. We will then execute an update query which will put a lock on the rows for as long as the transaction is not committed. We will execute a simple update query which will change all the values to 20.

```sql
update tbl set column1 = 20
```

4.1.3 Execute a read query in Terminal B

1. We will open up another terminal window that will act as the second application trying to access the table. Open a terminal window as by right-clicking on the Desktop area and choose the “Open Terminal” item. This new terminal will be designated as Terminal B.
2. Similar to the first terminal, we will connect to the database “sample” as user “db2inst1” with password “password” by typing in the command

```
db2 connect to sample
```

3. Next, we will launch a query that will read the data locked by Terminal A.

```
time db2 "select * from tbl"
```

The `time` command will allow us to quantify the wait time. We can see that the query waits and does not return any result. In fact, it is being blocked by Terminal A’s query.

```
<table>
<thead>
<tr>
<th>COLUMNS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

Terminal A

```
db2 => update tbl set column1 = 20
DB200001 The SQL command completed successfully.
db2 =>
```

```
db2inst1@db2rules:~
```

```
<table>
<thead>
<tr>
<th>COLUMNS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

Terminal B

4.1.4 Releasing the lock

1. With the 2 terminals open beside each other, we will observe the effect of committing the query in Terminal A. In Terminal A, commit the transaction by executing the following command

```
commit
```
We can see that terminal B's query instantly returned with the updated values. The block by terminal A has been released and the transaction on terminal B was allowed to continue and access the values.

**4.2 The “After” scenario: With Currently Committed**

We will repeat the procedure again but this time with the Currently Committed feature turned on. The objective is to see the difference in the time it took for the second query to return and the actual values being returned.
4.2.1 Turning on Currently Committed

1. In Terminal A, we will use the command to turn on currently committed:

   ```
   update db cfg for sample using cur_commit on
   ```

2. After changing the value, we need to disconnect the database connection for the new value to take effect. In terminal A, execute:

   ```
   connect reset
   ```

3. In terminal B, execute:

   ```
   db2 connect reset
   ```

4.2.2 Execute a write query in Terminal A

1. Similar to the previous section, we will update the values in the table from 20 to 30.

   ```
   connect to sample
   update tbl set column1 = 30
   ```

   You should see that the query has been executed successfully.

4.2.3 Execute a read query in Terminal B

1. In Terminal B, reconnect to the database and try to retrieve the values from table tbl1.

   ```
   db2 connect to sample
   time db2 "select * from tbl"
   ```
Notice the amount of time the query took to return this time. The query returned instantly because there was no access block to the data. Also, notice the values returned were not from the most recent update since we have not committed it yet.

2. In Terminal A, commit the update by typing in the command

```
commit
```

3. Switch the focus back to Terminal B. We want to execute the selection query again by pressing the up arrow button once to retrieve the last executed command, and then press Enter. If you cannot find the last command, type in

```
time db2 "select * from tbl"
```

Notice the values returned this time reflects our last update since the transaction in terminal A has ended and the updates committed to the database.
4. Terminate the database connection in terminal A:

```
connect reset
```

5. Then, terminate the database connection in terminal B:

```
db2 connect reset
```

---

5. **Repeatable Read**

Now that we have demonstrated the effect of cursor stability and the currently committed feature, we will take a look at repeatable read. To do so, we will simulate a scenario to show how repeatable read isolates each transaction to prevent phantom read concurrency issues.

Application A will execute a query that reads a set of rows based on some search criterion. Application B will try to insert new data that would satisfy application A’s query.

We will use DB2’s command line processor (CLP) to simulate the applications accessing the database at the same time.
5.1 The “Phantom Read” scenario: Repeatable Read

5.1.1 Execute a read query in Terminal A

1. We need to change the isolation of the current CLP session of Terminal A to repeatable read. This must be done before connecting to a database.

   change isolation to RR

2. Connect to database “sample”.

   connect to sample

3. Now we can perform a quick query to observe the current values for column “column1” based on some criteria.

   select * from tb1 where column1 = 30

5.1.2 Execute a write query in Terminal B

1. We will launch a query that will attempt to insert data into tb1 which is locked by Terminal A.

   db2 connect to sample
   db2 "insert into tb1 values (30)"
We can see that the operation waits and does not return any result. In fact, it is being blocked by Terminal A’s query.

### Terminal A

```
select * from tbl where column1 = 30
```

```
COLUMN1
--------------
  30
  30
  30
  30
  30
  30
  30
  30
  30
9 record(s) selected.
```

```
db2 =>
```

### Terminal B

```
connect to sample
```

```
Database Connection Information

Database server = DB2/LINUX 9.7.1
SQL authorization ID = DB2INST1
Local database alias = SAMPLE
```

```
insert into tbl values (30)
```

#### 5.1.3 Releasing the lock

1. With the 2 terminals open beside each other, we will observe the effect of committing the query in Terminal A. In Terminal A, commit the transaction by executing the following command

```
commit
```
We can see that terminal B’s query instantly completed. The block by Terminal A has been released and the transaction on Terminal B was allowed to insert the new values.

Here we can see that with the Repeatable Read isolation level, phantom read scenarios do not occur because the rows read by the application are locked and cannot be updated by other transactions.

What if we perform the same scenario with the read stability isolation level instead?

2. Terminate the database connection in terminal A:

```sql
connect reset
```

3. Then, terminate the database connection in terminal B:

```sql
db2 connect reset
```
6. Read Stability

We have previously determined that phantom reads cannot occur with the repeatable read isolation level. They are possible, however, when using the read stability isolation level. We will simulate a scenario to show how read stability differs from repeatable read in terms of isolating transactions.

Application A will execute a query that reads a set of rows based on some search criterion. Application B will insert new data that would satisfy application A's query.

We will use DB2’s command line processor (CLP) to simulate the applications accessing the database at the same time.

6.1 The “Phantom Read” scenario: Read Stability

6.1.1 Execute a read query in Terminal A

1. We need to change the isolation of the current CLP session of Terminal A to read stability. This must be done before connecting to a database.

```
change isolation to RS
```

2. Connect to database “sample”.

```
connect to sample
```

3. Now we can perform a quick query to observe the current values for column “column1” using some criteria.

```
select * from tbl where column1 = 30
```
The number of record(s) selected is currently 10.

6.1.2 Execute a write query in Terminal B

1. Terminal B will insert data matching the criteria of the query by Terminal A.

```
db2 connect to sample
db2 "insert into tb1 values (30)"
```

We can see that the query does not wait for Terminal A to commit and inserts data into tb1.
6.1.3 Execute another read query in Terminal A

1. Now we can perform another quick query to observe the current values for column “column1” before committing.

\[
\text{select } * \text{ from tbl where column1 = 30}
\]
Notice the query now returns 11 rows of data instead of 10. One additional row has appeared even though we executed the same SQL query inside the same transaction. This is because the Read Stability isolation level does not prevent the appearance of phantom rows.

2. In Terminal A, commit the update by typing in the command

   ```
   commit
   ```

3. Terminate the database connection in terminal A:

   ```
   connect reset
   ```

4. Then, terminate the database connection in terminal B:

   ```
   db2 connect reset
   ```

7. **Uncommitted Read**

Now that we know what the difference between repeatable read and read stability is, we can see how the lowest isolation level functions. The uncommitted read isolation level can be useful when using read-only tables or only select statements. When using uncommitted read, uncommitted data from other transactions is read.

Application A will execute a query that updates a row using RR. Application B will attempt to read the same row using CS and UR.
7.1 The “Uncommitted Read” scenario: Cursor Stability

7.1.1 Execute an update query in Terminal A

1. We need to change the isolation of the current CLP session of Terminal A to repeatable read. This must be done before connecting to a database.

```
change isolation to RR
```

2. Connect to database “sample”.

```
connect to sample
```

3. Now we can perform a quick query to update the current values for column “column1”.

```
update tb1 set column1 = 40
```

```
File Edit View Terminal Tabs Help

db2 => change isolation to RR
DB21053W Automatic escalation will occur when you connect to a database that does not support RR.
DB20000I The CHANGE ISOLATION command completed successfully.
```

```
db2 => connect to sample
```

```
Database Connection Information
Database server = DB2/LINUX 9.7.1
SQL authorization ID = DB2INST1
Local database alias = SAMPLE
```

```
db2 => update tb1 set column1 = 40
DB20000I The SQL command completed successfully.
```

```
db2 =>
```

7.1.2 Execute a read query in Terminal B

1. Using CS, Terminal B will attempt to read the data being locked by Terminal A.

```
db2 connect to sample
db2 "select * from tb1"
```

We can see that the select query waits for Terminal A to commit before reading the data.
7.1.3 Releasing the lock

1. With the 2 terminals open beside each other, we will observe the effect of committing the query in Terminal A. In Terminal A, commit the transaction by executing the following command

commit
We can see that terminal B’s query instantly completed. The block by Terminal A has been released and the transaction on Terminal B was allowed to read the committed data.

2. Terminate the database connection in terminal B:

```
db2 connect reset
```
7.2 The “Uncommitted Read” scenario: Uncommitted Read

7.2.1 Execute an update query in Terminal A

1. We will perform a quick query to update the current values for column “column1”.

```
update tbl set column1 = 50
```

7.2.2 Execute a read query in Terminal B

1. Terminal B will attempt to read the data being locked by Terminal A using UR.

```
db2 change isolation to UR
db2 connect to sample
db2 "select * from tbl"
```

We can see that the select query under the uncommitted read isolation level does not wait for Terminal A to commit before reading the data. Instead the values returned are from the uncommitted transaction from Terminal A.

If the transaction from Terminal A executes a rollback, the data listed in Terminal B does not reflect the actual data in TB1. This phenomenon is called a “dirty read”.
2. In Terminal A, commit the update by typing in the command:

```
commit
```

3. Terminate the database connection in terminal A:

```
connect reset
```

4. Then, terminate the database connection in terminal B:

```
db2 connect reset
```