Learn about autonomous transactions, a feature introduced in IBM DB2® Version 9.7 for Linux®, UNIX®, and Windows®. You will get an overview and example of what an autonomous transaction is and how to create one.

View more content in this series

Introduction

This article introduces autonomous transactions, a feature introduced in DB2 9.7. Autonomous transactions are of interest to both database administrators and application developers.

This article discusses DB2 9.7 for Linux, UNIX, and Windows. Knowledge of the DB2 command line processor (CLP) and SQL PL are beneficial.

To run the examples, you need access to a DB2 9.7 for Linux, UNIX, and Windows database. See Related topics to download a trial copy of DB2.

Understanding a little background about transactions

Transactions are real world entities that can be expressed in the form of text, numbers, or both, to be processed by a database management system. They should make sense as actions against the database, and they must be performed as a group.

For example, a request to transfer $X$ amount from user A's account to user B's account is a simple transaction. This transaction can be broken into two SQL statements, as shown in Listing 1.

Listing 1. Example of simple transaction

```
Update table AccountInfo set CurrentBalance = CurrentBalance - X, where UserName=A
Update table AccountInfo set CurrentBalance = CurrentBalance + X, where UserName=B
```
This transaction can only be called successful if both SQL statements were able to update the table successfully. To make sure that either both statements take effect or none, the applications are run in such a manner that no changes are made to the database until a COMMIT occurs. On a COMMIT, all the uncommitted statements (statements after last COMMIT) take effect together, ensuring data integrity. This is similar to turning OFF the AUTO COMMIT behavior of the command line processor (CLP), then issuing a set of statements and then manually giving a COMMIT when done. A ROLLBACK removes all uncommitted changes. So COMMIT and ROLLBACK statements are important building blocks of a transaction implementation.

**Introducing an autonomous transaction**

An autonomous transaction has its own COMMIT and ROLLBACK scope to ensure that its outcome does not affect the caller's uncommitted changes. Additionally, the COMMITs and ROLLBACKs in the calling session should not affect the changes that were finalized on the completion of the autonomous transaction itself.

Note that the calling session is suspended until the called session returns control. Support for autonomous transactions in no way should be seen as a support to have parallel executing sessions.

**Creating an autonomous transaction**

In DB2, autonomous transactions are implemented through autonomous procedures. A stored procedure provides a natural means to bundle statements into a block. To create an autonomous procedure, the keyword AUTONOMOUS is specified on the CREATE PROCEDURE statement, as shown in Listing 2.

**Listing 2. Example of CREATE PROCEDURE statement**

```
CREATE OR REPLACE your_procedure_name
LANGUAGE SQL
AUTONOMOUS
BEGIN
  do autonomous work ;
END
```

When an autonomous procedure is invoked, it is executed within its own session to provide the transaction independence required. Successful autonomous procedures implicitly commit, while unsuccessful autonomous procedures rollback. In either case, the invoking transaction is untouched.

**Seeing a real-life usage example**

Bank B wants to make sure each query to a table containing customer sensitive data is logged properly. To achieve this objective, application developers at Bank B are provided with a set of interfaces they may use to access the sensitive data. Each interface is implemented as a stored procedure. The stored procedure returns the information required from the table, while at the
same time logging the user ID of the employee making the query and the account number of the customer record being queried with date and time.

**Assumptions**

The SQL assumes all of the following:

- A database connection exists
- Auto commit is off
- The statement terminator is set to `%` which is done by entering the DB2 CLP statement `db2 +c -td%` while starting a new DB2 CLP session

**Getting started**

First create the necessary tables. You'll need one table to hold the customer-sensitive information and another to hold the information to be logged each time the sensitive information is accessed. Listing 3 shows an example.

**Listing 3. Creating two example tables**

```sql
DROP TABLE customerSensitiveInfo %
CREATE TABLE customerSensitiveInfo(customerAccountNumber integer, amountOverdue integer) %

DROP TABLE log_table %
CREATE TABLE log_table(queryingEmployeeID varchar(100),
  customerAccNumber integer, when timestamp) %
COMMIT %
```

Next, create a procedure to write to the log_table when the sensitive information is accessed, as shown in Listing 4.

**Listing 4. Writing to the log table**

```sql
CREATE OR REPLACE PROCEDURE log_query (in queryingEmployee varchar(100),
  in accNumber integer, in when timestamp)
LANGUAGE SQL
BEGIN
  insert into log_table values (queryingEmployee, accNumber, when);
END %
COMMIT %
```

One of the interfaces that Bank B provides its application developers is a means to query by how much a given account is overdue. The procedure `get_AmountOverdue` first makes a call to the `log_query` procedure to record that it will be accessing sensitive data. It then does a `select` from the `customerSensitiveInfo` table to retrieve the amount overdue for the given account number. Listing 5 shows an example.
Listing 5. get_AmountOverdue procedure

```sql
CREATE OR REPLACE PROCEDURE get_AmountOverdue(in accountNumber integer, out overdue integer)
LANGUAGE SQL
BEGIN
  DECLARE due integer;
  DECLARE currentTime timestamp;
  SET currentTime= CURRENT TIMESTAMP;
  CALL log_query(CURRENT USER, accountNumber, currentTime);
  SELECT amountOverdue INTO due FROM customerSensitiveInfo
    WHERE customerAccountNumber= accountNumber;
  SET overdue=due;
END %
COMMIT %
```

After you create the interface get_AmountOverdue, add some customer data to the customerSensitiveInfo table. Next, execute the statement in Listing 6 to create the table.

Listing 6. Statement to create the example table

```sql
INSERT INTO customerSensitiveInfo VALUES(12345, 10000),(12346,20000) %
COMMIT %
```

The resulting table contains the information as shown in Table 1.

Table 1. CustomerSensitiveInfo

<table>
<thead>
<tr>
<th>CustomerAccountNumber</th>
<th>AmountOverdue</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>10,000</td>
</tr>
<tr>
<td>12346</td>
<td>20,000</td>
</tr>
</tbody>
</table>

With the table now populated with data and a means to access it, retrieve the amount overdue from account 12345. Because you are only interested in viewing the data, you would prefer to do so anonymously, issue a rollback statement immediately after the call to cover your tracks, as shown in Listing 7.

Listing 7. Add a rollback statement to get_AmountOverdue code

```sql
CALL get_AmountOverdue(12345, ?) %

Value of output parameters
--------------------------
Parameter Name : OVERDUE
Parameter Value : 10,000

return Status = 0
ROLLBACK %
```

Check the state of the log_table, which looks like Listing 8.
Listing 8. The log_table

```
SELECT * FROM log_table %
QUERYINGEMPLOYEEID   CUSTOMERACCNUMBER   WHEN
0 record(s) selected.
```

As expected, the log table is empty because the transaction containing both the actual access to the table and the insert into the log table was rolled back. This is definitely not the behavior desired. Add the AUTONOMOUS keyword to the log_query procedure, as shown in Listing 9.

Listing 9. The log_query procedure with the AUTONOMOUS statement

```
CREATE OR REPLACE PROCEDURE log_query (in queryingEmployee varchar(100),
                                       in accNumber integer, in when timestamp)
LANGUAGE SQL
AUTONOMOUS
BEGIN
    INSERT INTO log_table values (queryingEmployee, accNumber, when);
END %
COMMIT %
```

Now retrieve the amount overdue from the account 12345, and rollback the transaction after doing so again, as shown in Listing 10.

Listing 10. The get_AmountOverdue code with rollback statement

```
CALL get_AmountOverdue(12345, ?) %
Value of output parameters
--------------------------
Parameter Name : OVERDUE
Parameter Value : 10,000
return Status = 0
ROLLBACK %
```

Again, check the state of the log_table, which now looks like Listing 11.

Listing 11. The log_table after adding the AUTONOMOUS statement

```
SELECT * FROM log_table %
QUERYINGEMPLOYEEID   CUSTOMERACCNUMBER   WHEN
98765                 12345                   2009:05:25:12.00.00.000000
1 record(s) selected.
```

This time, the desired result is achieved. Even though the transaction that read the sensitive information was itself rolledback, the entries to the log_table are committed. This way, you can maintain a history of who had access to the data, even if the access itself was not committed.
Conclusion

This article introduced the concept of autonomous transactions. You should now understand what an autonomous transaction is as well as how to create and use one within DB2.
Related topics

- Use an **RSS feed** to request notification for the upcoming articles in this series. (Find out more about **RSS feeds of developerWorks content**.)
- Download **DB2 Express-C 9.7**, which is a no-charge version of DB2 Express database server.
- Download a no-charge trial version of **DB2 9.7 for Linux, UNIX, and Windows**.
- In the **DB2 for Linux, UNIX, and Windows area on developerWorks**, get the resources you need to advance your skills in DB2.
- Find information describing how to use DB2 in the **DB2 for Linux, UNIX, and Windows Information Center**.

© Copyright IBM Corporation 2009
**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)