AUTONOMOUS TRANSACTIONS: EXTENDING THE POSSIBILITIES

Michael Rosenblum, Dulcian, Inc.

"If you are not allowed to do something, but really want it – sometimes you can"
Jewish proverb

Introduction

Rules are created to be broken. This axiom is true for almost any activity, even database development. Any complex system has a basic set of rules. However, in real life these rules may become limitations. Therefore, in order to ensure a successful implementation, the rules must either be changed or weakened (the rule is still in place but sometimes can be avoided). Changing a rule in an information system may be the best alternative conceptually, but doing this is often the most expensive since it entails modifications to the way of thinking and working for every person linked to the defined system. Weakening the existing rules is usually easier but includes some risks since the new meaning of the rules may not be readily embraced or properly understood by the developers and users who are comfortable with the old definitions.

This paper discusses advanced transaction control. Historically, the autonomous transaction functionality was an internal Oracle tool, which circumvented Oracle's own restrictions, such as making sequence changes permanent without influencing users' sessions. As a production feature, it became available starting with version 8.1.5 by providing "back doors." The extremely simple syntax led many people to underestimate the consequences of using autonomous transactions. This created a bad reputation for what was essentially an extremely useful and powerful feature. The goal of this paper is to change readers' minds about using autonomous transactions and explain why these can and should be used in certain situations.

I. Important Concepts

It is important to explain the following concepts in order to understand the processes explained later in this paper.

A. Definition

Autonomous transactions are independent transactions that can be called from within other transactions.

B. Syntax

```
declare
       Pragma autonomous_transaction;
  Begin
       ……
     commit;(or rollback;)
  End;
```

C. Language elements

`Pragma autonomous_transaction` – defines a specific transaction as autonomous.

This syntax can be used in the declaration part of the following:

- Top-level anonymous blocks
- Local, standalone, or packaged functions and procedures
- Methods of object types
- Database triggers

However, this syntax cannot be used in any of the following situations:

- Outside of a declaration section
- Within the declaration section of a nested block (a block within a block)
- In a package specification
- In a package body outside of a procedure or function definition
- In a type body outside of a method definition

`Begin ... end;` – An autonomous transaction starts from the `begin` command of the block, where the defining statement is found. The corresponding `end` command does not close the autonomous transaction.

`commit;` (or `rollback;`) – Data changes made in an autonomous transaction must be committed or rolled back. If such activity has not happened and the block defined as an autonomous transaction has ended, the Oracle DBMS will rollback the entire transaction and display the following message "ORA-06519: active autonomous transaction detected and rolled back."

An autonomous transaction allows you to do the following:

- Leave the context of the calling transaction (parent)
- Perform SQL operations
- Commit or rollback those operations
- Return to the calling transaction's context
- Continue with the parent transaction

D. Database objects and conventions

The database objects used throughout this paper are defined below:

1. EMP – main data source table

```
Create table emp (empno number primary key,
                  ename varchar2(2000),
                  deptno number,
                  mgr number,
                  job varchar2(255),
                  sal number)
```

2. AUDIT_EMP – special table that contains information about EMP table activity

```
create table Audit_emp (action_nr number,
                   action_cd varchar2(2000),
                   descr_tx varchar2(2000),
                   user_cd varchar2(2000),
                   date_dt date)
```

3. AUDIT_SEQ – source of primary keys for AUDIT_EMP

```
create sequence audit_seq
```

II. Basic Example

The example used here is designed to track any salary changes in the system (whether they are committed or not). Therefore a BEFORE UPDATE trigger is placed on the column Sal.

```
create or replace trigger Bu_emp
 before update of sal on Emp
referencing new as new old as old
 for each row
begin
    p_log_audit (user,
                 'update',
                 'update of emp.salary',
                  sysdate);
end;
```

There is also a generic function to log activities, which will be used for different examples later:

```
create or replace procedure p_log_audit (
        who varchar2, what varchar2,
        descr_tx varchar2, when_dt date)
is
    pragma autonomous_transaction;
begin
    insert into Audit_emp
    values(audit_seq.nextval,
           what,
           descr_tx,
           who,
           when_dt);

     commit;
end;
```

The code works through 4 steps:

1. The trigger calls the procedure P_LOG_AUDIT (still in the same transaction).
2. The declaration block of the function still belongs to the main transaction; however, the Oracle engine encountered the line PRAGMA AUTONOMOUS_TRANSACTION. This means that from the following BEGIN statement, it should start a new transaction in the current session.
3. Inside of the autonomous transaction, a new record was inserted into the table AUDIT_EMP and the change was committed. Note that the commit happened only for changes in this transaction and is completely independent of the parent transaction. Any unsaved data will be still unsaved. Also, it does not matter what happens with the update statement. The log information was already sent to the database. Nothing in the parent transaction removes the record. The key point to remember is that COMMIT/ROLLBACK statements in autonomous transactions are absolutely independent from ones in other transactions. They only affect changes in the specified transaction, not in any others.
4. When the autonomous transaction ends, since the INSERT has been committed, the PL/SQL engine can return to the main transaction (the trigger), from which the procedure has been called.

Therefore, it can be concluded that some activities should be handled differently from the point of transaction control as described in the next section.

III. Nested vs. Autonomous Transactions

To be able to properly describe an autonomous transaction, it is useful to make a comparison to a more familiar concept, namely, nested transactions. As defined in "Nested Transactions: An Approach to Reliable Distributed Computing" by J. Moss at M.I.T, a nested transaction is “a tree of transactions, the sub-trees of which are either nested or flat transactions.”

In the context of the Oracle DBMS, this means that each time a function, procedure, method, or anonymous block is called within another block or trigger, it spawns a sub-transaction of the main transaction. Everything in this list (except anonymous sub-blocks) can be also defined as an autonomous transaction. Understanding the difference requires the introduction of a new concept:

The **SCOPE** of the objects defines visibility of the object within the database. Scope can be applied to any of the following:

- Variables
- Session settings/parameters
- Data changes
- Locks
- Exceptions

The following sections describe a number of test cases for all of these objects of interest.

A. Locks

Locks represent a fairly a simple problem to resolve. In the parent transaction, the row from the EMP table is locked for update. Exactly the same action is done in `LOCK_TEST`. Locks are a good example of transactional resources. The following test shows how autonomous transactions work with them.

```
procedure lock_test is
     v varchar2(2000);
     -- pragma autonomous_transaction; -- with/without that line
begin
     select ename into v from emp
     where ename = 'SCOTT' for update;
     -- commit; -- with/without that line
end;
```

1. Nested transaction

```
SQL> declare
  2    v varchar2(2000);
  3  begin
  4     select ename into v from emp
  5     where ename = 'SCOTT' for update;
  6     lock_test;
  7     commit;
  8  End;
  9  /
PL/SQL procedure successfully completed.
SQL>
```

2. Autonomous transaction

```
SQL> declare
  2    v varchar2(2000);
  3  begin
  4     select ename into v from emp
  5     where ename = 'SCOTT' for update;
  6     lock_test;
  7     commit;
  8  End;
  9  /
declare
*
ERROR at line 1:
ORA-00060: deadlock detected while waiting for resource
ORA-06512: at "SCOTT.LOCK_TEST", line 5
ORA-06512: at line 6
SQL>
```

These results are extremely interesting. The first (nested) version of the procedure worked without question. But in the second case, the result was a deadlock. This leads to the formulation of the following rule:

Rule #1: An autonomous transaction does not share transactional resources (such as locks) with the main transaction.

B. Session Resources

Since transactional resources are processed differently in autonomous transactions than they are in nested ones, it does make sense to test session-level resources within the same context. The most common resources (packaged variables) have been used for this purpose. The test attempted to update the variable from both sides. In the procedure defined as an autonomous transaction, the value of the variable VAR_TEST.GLOBAL_NR (already changed in the parent transaction) is displayed first. Next, the variable is updated and, the value from the parent will be checked after ending the autonomous transaction.

```
SQL> create or replace package var_test
  2  As
  3      global_nr number :=0;
  4  end;
  5  /
Package created.
SQL> create or replace procedure p_var_test  (v_nr number) is
  2      pragma autonomous_transaction;
```

```
  3  Begin
  4      dbms_output.put_line(' Before Auto value: '||var_test.global_nr );
  5      var_test.global_nr := v_nr;
  6    commit;
  7  end;
  8  /
Procedure created.
SQL> Begin
  2    dbms_output.put_line('Start value: '||var_test.global_nr );
  3     var_test.global_nr := 10;
  4    p_var_test (20);
  5      dbms_output.put_line('After Auto value: '||var_test.global_nr );
  6  End;
  7  /
Start value: 0
Before Auto value: 10
After Auto value: 20
PL/SQL procedure successfully completed.
SQL>
```

In examining the three output values, the first is different from second in that the autonomous transaction can see the change made by the parent one. The second value is different from the third in that the parent transaction can see the change made by the child one. This works in the expected way and leads to the formulation of a second rule:

Rule #2: Autonomous transactions and main transactions belong to the same session and share the same session resources.

C. Changes in Parent Transactions

So far, the sections above discussed uncommitted changes in autonomous transactions. At this point, a logical question to ask is: What would happen with uncommitted changes in the parent transaction? This can be answered using the following test:

- In an anonymous block, count the records from the table AUDIT_EMP (it is empty for now).
- Insert a new record into that table without a COMMIT and call the procedure DATA_CHANGE_TEST, which will try to count the records in the same table.

```
procedure data_change_test is
     v_nr number;
 pragma
    -- autonomous_transaction; -- with/without that line
begin
     select count(1) into v_nr
     from audit_emp;
     dbms_output.put_line ('Count#2='||v_nr);
end;
```

1. Nested transaction

```
SQL> Declare
  2   v_nr number;
  3  Begin
  4   Select count(1) into v_nr from audit_emp;
  5   insert into audit_emp values (audit_seq.nextval,'Test','Test',user,sysdate);
  6   dbms_output.put_line('Count#1='||v_nr);
  7   data_change_test;
  8  End;
  9  /
Count#1=0
Count#2=1
PL/SQL procedure successfully completed.
SQL>
```

2. Autonomous transaction

```
SQL> Declare
  2   v_nr number;
  3  Begin
```

```
  4   Select count(1) into v_nr from audit_emp;
  5   insert into audit_emp values (audit_seq.nextval,'Test','Test',user,sysdate);
  6   dbms_output.put_line('Count#1='||v_nr);
  7   data_change_test;
  8  End;
  9  /
Count#1=0
Count#2=0
PL/SQL procedure successfully completed.
SQL>
```

In this case, the autonomous transaction does not recognize the new record. It simply doesn't exist for it leading to the formulation of a third rule:

Rule#3: Non-committed changes of parent transactions are not immediately visible to autonomous transactions, but are visible for nested ones.

D. Changes in Child Transactions

The test described in the previous section indicated what happens with records that have been inserted from the parent transaction. What are the changes going to be in the child transactions? To answer this question, some additional definitions are needed.

Isolation level is defined as the degree to which the intermediate state of the data being modified by a transaction is visible to other concurrent transactions; and, the data being modified by other transactions is visible to it. There are two supported isolation levels in the Oracle DBMS (other database manufacturers may have different isolation levels):

- Read committed: a transaction rereads data that it has previously read and finds that another committed transaction has modified or deleted the data. A transaction re-executes a query, returning a set of rows that satisfies a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition
- Serializable: the transaction cannot see any changes that happened in other transactions, that have been processed AFTER it started

It is clear that this parameter does not have any connection with nested transactions. These transactions are part of the main one which is why changes made in the main transaction must be visible throughout the whole tree. But for autonomous transactions, the question remains: Are there any data visibility differences depending upon the isolation level? This can be tested using an old pattern where one INSERT is done in the parent transaction and another in the autonomous transaction. Before each test table, AUDIT_EMP is truncated.

```
procedure commit_test
is
pragma autonomous_transaction;
begin
  insert into audit_emp values (1,'Test','Test',  user,sysdate );
commit;
end;

SQL> declare
  2    v_nr number;
  3  Begin
  4    set transaction isolation level read committed;
  5    insert into audit_emp values (1,'Test','Test', user,sysdate );
  6    commit_test;
  7    select count(1) into v_nr from audit_emp;
  8    dbms_output.put_line   ('Count read committed='||v_nr);
  9  end;
 10  /
Count read committed=2
PL/SQL procedure successfully completed.
SQL> declare
  2    v_nr number;
  3  Begin
  4    set transaction isolation level serializable;
```

```
  5    insert into audit_emp values (1,'Test','Test', user,sysdate );
  6    commit_test;
  7    select count(1) into v_nr from audit_emp;
  8    dbms_output.put_line   ('Count serializable='||v_nr);
  9  end;
 10  /
Count serializable=1
PL/SQL procedure successfully completed.
SQL>
```

From these results, it seems that for the Oracle DBMS, there is no difference between autonomous transactions and transactions from another session in the context of data visibility. This leads to the formulation of a fourth rule:

Rule#4: Changes made by autonomous transactions may or may not be visible to the parent one depending upon the isolation level, while changes made by nested transactions are always visible to the parent one.

E. Exceptions

As mentioned previously, if changes in the autonomous transaction are not committed or rolled back, the Oracle DBMS will raise an error and rollback the whole transaction. It is possible that something in the autonomous transaction went wrong. What would happen with uncommitted changes? To test this, the procedure `ROLLBACK_TEST` was created. There are two INSERT statements. The second one tries to place text data in a numeric field. In the parent transaction, an exception handler catches the raised exception and counts the number of records that went to the table AUDIT_EMP (as usual, the table is truncated before the test) as shown here:

```
procedure rollback_test is
     -- pragma autonomous_transaction; -- with/without this line
begin
     insert into audit_emp values (1,'Test','Test', user,sysdate );
     insert into audit_emp values ('Wrong Data','Test', 'Test', user,sysdate );
     -- commit; -- with/without this line
end;
```

1. Nested transaction

```
SQL> Declare
  2       v_nr number;
  3  Begin
  4   rollback_test;
  5  Exception
  6      when others then
  7         select count(1) into v_nr from audit_emp;
  8         dbms_output.put_line ('Count='||v_nr);
  9  end;
 10  /
Count=1
PL/SQL procedure successfully completed.
SQL>
```

2. Autonomous transaction

```
SQL> Declare
  2       v_nr number;
  3  Begin
  4   rollback_test;
  5  Exception
  6      when others then
  7         select count(1) into v_nr from audit_emp;
  8         dbms_output.put_line ('Count='||v_nr);
  9  end;
 10  /
Count=0
PL/SQL procedure successfully completed.
SQL>
```

In the second case (autonomous transaction), both records were lost. This leads to the formulation of the next rule:

Rule #5: Exceptions raised in an autonomous transaction cause a transaction-level rollback, not a statement-level rollback.

Summary

The first part of this article attempted to explain the differences between nested transactions and autonomous transactions with the goal of clarifying what autonomous transactions are. Using a set of rules, all results should be predictable. There is no place for surprises in database development. When working with autonomous transactions, it is important to understand how the Oracle DBMS processes them in the following contexts:

- Transactional resources
- Session-level resources
- Data changes of parent transaction
- Data changes of autonomous transaction
- Exceptions

IV. How to Use Autonomous Transactions

The following are some basic examples covering autonomous transactions that might be found in many reference books. The purpose of summarizing them here is to provide a pattern of thinking about autonomous transactions and use some common real world situations to make the suggestions relevant.

There are three major areas where autonomous transactions could be helpful:

1. Security Subsystems: Since this feature allows for working independently with multiple sets of data, it is relevant for security subsystems
2. Advanced control of transaction-level resources allows some structural optimization
3. Resolving non-standard PL/SQL problems

A. Security: Query Audit

The current trend towards more robust security has also influenced the database development industry. This section will discuss how to implement some security using autonomous transactions.

Business Rule: Each request by a user to view the Salary column should be recorded.

```
create or replace package audit$pkg is
 function f_record (in_id number, in_tx varchar2, in_value_nr number) return number;
end;

create or replace package body audit$pkg as
 function f_record (in_id number, in_tx varchar2, in_value_nr number) return number
 return number
 is
         pragma  autonomous_transaction;
 begin
    insert into audit_emp values
       (audit_seq.nextval,
       'VIEW',
       'Request of '||in_tx||'='
          ||in_value_nr|| ' from emp by pk='
          ||in_id,
       user,
       sysdate );
    commit;
    return v_value_nr;
 end;
End;

create or replace view v_emp
As
Select empno,
          ename,
          audit$pkg.f_record(empno, 'sal', sal) sal
From emp
```

The idea is very simple. There is a view, V_EMP, that has exactly the same columns, as the table EMP. But for the column Sal, the packaged function AUDIT$PKG.F_RECORD is used. The function has the value of the column Sal as an incoming parameter so that it can be returned properly. The user will see the value he/she wanted. But, at the same time, this function logs a querying request to the table AUDIT_EMP. This log record is committed at the moment when the user retrieves the data, just as it was specified.

Advanced query audit

In the previous example a fairly interesting technique was used, namely, sending the value of a column to a function to be returned. This trick is used to solve the next problem.

Business Rule: A user can query specific data only once per session from the temporary dataset that is created each time a session starts.

```
create or replace package audit$pkg is
 function f_clean (in_id number, in_nr number) return number
end;

create or replace package body audit$pkg as
 function f_clean (in_id number, in_nr number) return number
 is
         pragma  autonomous_transaction;
 begin
     delete from temp_emp where empno=in_id;
     commit;
     return in_nr;
 end;
End;

create or replace view v_emp
As
Select empno,
          ename,
          audit$pkg.f_clean(empno, sal) sal
From temp_emp
```

Although is may appear to be a strange concept, the solution is based on the set of assumptions that at the moment of execution of the AUDIT$PKG.F_CLEAN command, all data is already cached and available for querying. A this point, it will not interrupt the executed statement. The function AUDIT$PKG.F_CLEAN has two parameters: the primary key of the table EMP, and value of the column SAL. As in the first example, SAL is needed only to be returned (as temporary storage), while the primary key is used to delete the record from table TEMP_EMP. The algorithm is clear. A selection from the view causes the return of the data from the requested record in addition to removing all records that have been requested. It absolutely satisfies the specified business rule and the problem is solved.

B. Activity Audit

The problem of performing a generic audit was discussed in the first autonomous transaction example In addition, there are some other interesting tricks that can be used involving autonomous transactions.

Business Rules to be implemented:

1. A user-executed update on any salary should be recorded, even if the update failed. A user can update Salary only if he/she and his/her direct manager both have the job “MANAGER.”
2. The second part of the rule is more complex since it is based on the same table, which is updated. The job of the manager is not known, only the ID. But since Oracle does not allow it (ORA-04091: table SCOTT.EMP is a mutating trigger/function and may not see it), there is no way to implement the specified rule in conventional trigger.
3. The generic function CHECK_PRIVILEGES is used in many different places. It cannot be modified.

```
Function check_privileges
                    (in_mgr number,
                     in_job_emp varchar2)
Return boolean is
    v_job_mgr varchar2(2000);
```

```
Begin
     select job into v_job_mgr from emp where empno = in_mgr;

     if  in_job_emp = 'MANAGER' and v_job_mgr = 'MANAGER'
      then return TRUE;
      else return FALSE;
     end if;
End;
```

Since the trigger could be defined as an autonomous transaction, all nested transactions spawned by it are members of the one started in the trigger. Therefore, they are inherently autonomous in terms of the main transaction (in which the update happened). This avoids the Oracle exception and places the business rule as specified.

```
Trigger emp_audit
 Before update on emp
 For each row
Declare
   pragma autonomous_transaction;
Begin
   if (check_privileges
            (:new.mgr :new.job))
   then
       p_log_audit (user,
                    'update: rule succeeded',
                    'update of   emp.salary',
                    sysdate);
          commit;
   else
       p_log_audit (user,
                    'update: rule failed',
                    'update of emp.salary',
                     sysdate);
      commit;
      raise_application_error
                (-2001, 'Access denied!');
   end if;
End;
```

C. Modular code: Consistency of environment

The Oracle DBMS sometimes has a set of implicit activities that cannot be separated. For example, any DDL statement forces a COMMIT for the whole transaction. But there are some cases, when this feature interferes with desired functionality and autonomous transactions can provide a solution.

Business Rule: Committing changes in a subroutine should not force any activity in other routines.

This problem is reasonably common. For example, in a large migration, parts of it are tables A and B. Table B is a child of A, but the foreign key is deferred.

```
Create table A (a number primary key);
Create table B (a number, b number);

Alter table B add constraint a_fk
foreign key (a)  references A(a) deferrable initially deferred;
```

Because of other data, table B must be migrated before table A. Before migrating table B, a script has to generate a brand new copy of the future table A (from a database link) with the current day in the table name.

```
Begin
 populate_b;
 copy_link_a (sysdate);
 populate_a;
End;
```

This may look simple, but without autonomous transactions it would not be possible since the foreign key is deferred. The creation of the table would force a COMMIT before table A has been migrated. The procedure copy_link_a is autonomous and does not do anything in the main transaction. Thus the business rule can be implemented as shown here:

```
Procedure copy_link_a (v_dt date)
```

```
Is
     pragma autonomous_transaction;
Begin
    execute immediate 'create table a_copy_'||to_char (sysdate,'ddmmyyyy')||' as select * from a@link';
End;
```

D. Non-standard PL/SQL: DDL in triggers

The problem of DDL in triggers is a subset of COMMIT statements in triggers that are allowed in autonomous transactions. But the following case is extremely useful. For example, there is a need to implement real-time communication between a data modeling front-end and an Oracle repository through the view, which displays all attributes of existing objects (physically implemented as columns of tables).

Business Rule: The insertion of a record in the view creates the new column in the specified table.

```
trigger u_uml_attrib
 Instead of Insert on uml_attrib
 For each row
Declare
          pragma autonomous_transaction;
Begin
     if check(:new.attrib_cd)='Y'
     then
        execute immediate
          ' alter table '||:new.class_cd ||' add column '||:new.attrib_cd ||' '||:new.datatype;
     end if;
End;
```

E. Non-standard PL/SQL: SELECT-only environment

Another simple example regarding security settings occurs when users have been allowed to only select data from the database (reporting utilities). However, there is also a need to execute a set of procedures at user logon. The solution is clear: create a view that will call a function with all logon procedures and query this view in each report at the very beginning. Since the function is defined as an autonomous transaction, all log values are already committed.

Business Rule: The system needs to register a user while the tool allows only SELECT statements.

```
function start_session
   (in_dt date,
    in_user varchar2)
    return varchar2
    is
        pragma autonomous_transaction;
Begin
        log_user (in_user, in_dt);
        set_system_defaults;
        populate_temp(in_dt, in_user);
        commit;
       return 'Y'
Exception when others return 'N';
End;

Create or replace view v_log
As
Select start_session
   (sysdate, user) flag
From dual
```

F. Non-standard PL/SQL: Self-mutation

This paper has already discussed how triggers defined as autonomous transaction allow querying of the same table that the trigger created. This situation can be extended farther.

Business Rule: The Rule for UPDATE is based on the same column that is updated. "The Average salary of an employee cannot be less than half of the maximum salary in his/her department."

The problem only appears simple. The salary could be updated for one employee only or for the set of employees. In the first case, row-level triggers could capture old and new values. Since all other data is static, the trigger could be defined as an autonomous transaction, query the maximum and average salaries in the department of the employee, and correct these aggregate values within the change in the current update.

For the set of rows, the problem lies in the nature of row-level triggers. Only the current update is known, but it is not possible to detect the generic influence of the whole statement on aggregates. Statement-level triggers do not have the ability to detect old and new values. If the AFTER-UPDATE trigger is defined as an autonomous transaction. nothing will be visible because the changes have not yet been committed in the parent transaction.

This solution is based on another advanced Oracle feature: object collections. The corresponding types are defined as follows:

```
create type emp_t as object
(empno number,  deptno number,
  old_sal number,  new_sal number);

create type emp_tt as table of emp_t;
```

Collection EMP_TEMP is instantiated inside of the package. For this reason, it is consistent and accessible for any transaction inside of the session.

```
create package obj
as
   emp_temp emp_tt := emp_tt();
end;
```

EMP_TEMP works as a buffer for changes. It is populated at the row-level BEFORE-UPDATE trigger as shown here:

```
Create or replace trigger BU_EMP
before update on EMP
begin
      obj.emp_temp.delete;
end;

Create or replace trigger BU_EMP_ROW
    before update on EMP
    for each row
Begin
     obj.emp_temp.extend;
     obj.emp_temp(obj.emp_temp.last)
              := emp_t (:new.empno,
                        :new.deptno,
                        :old.sal,
                        :new.sal);
End;
```

All prepared values are processed in the AFTER-UPDATE statement level trigger as shown here:

```
Create or replace trigger AU_EMP
After update on EMP
  pragma autonomous_transaction;
  cursor cDept
  is
  select t.deptno,
         sum(t.new_sal) -
                sum(t.old_sal) DeptDif,
          max(new_sal) MaxDif
  from table(
            cast(obj.emp_temp as emp_tt)
            ) t
  group by t.deptno;

  v_max number;
  v_avg number;
  v_count number;
Begin
  for cD in cDept
  loop
```

```
        select max(sal), avg(sal),count(1)
        into v_max, v_avg, v_count
        from emp
        where deptno = cd.Deptno;

        if (greatest (v_max, cd.MaxDif)/2)
             > ((v_avg*v_count +
                cd.DeptDif)/v_count)
        then
            raise_application_error
               (-20001, 'Rule Violated!');
        end if;
    end loop;
End;
```

Only the last trigger is defined as an autonomous transaction. It is necessary to query the state of the table EMP at the very beginning of the update. There is now enough information to validate the rule, namely preexisting data and all modifications. It does not matter how many records have been involved. The algorithm is 100% generic. The error raised in the AFTER-UPDATE trigger will be sent to the main transaction. This means that the update statement will be rolled back if the checking fails.

Conclusions

It is very easy to type a line of code, but it is not always that simple to figure out the effect of your code changes on the existing code. As with any other advanced feature, it is important to be aware of the restrictions, limits, and proper patterns of autonomous transactions.

In general, there are three important points to remember:

1. Autonomous transactions are powerful tools that allow us to solve old problems in new ways.
2. They are complex tools requiring a high level of familiarity with the Oracle DBMS.
3. They are sensitive tools that may result in unexpected (or even catastrophic) results if used improperly.

In summary, autonomous transactions are an interesting feature that can be very useful, but must be handled carefully.

About the Author

Michael Rosenblum is a Development DBA at Dulcian, Inc. He is responsible for system tuning and application architecture. He supports Dulcian developers by writing complex PL/SQL routines and researching new features. Mr. Rosenblum is the co-author of *PL/SQL for Dummies* (Wiley Press, 2006). Michael is a frequent presenter at various regional and national Oracle user group conferences. In his native Ukraine, he received the scholarship of the President of Ukraine, a Masters Degree in Information Systems, and a Diploma with Honors from the Kiev National University of Economics.