
# *Returning effected rows by DML statements in Oracle and SQL Server*

---

White Paper

Last revised: Jan. 2007

# Table of Contents

# Returning effected rows by DML statements in Oracle and SQL Server

## Abstract

The new release of SQL Server 2005 came with a wide variety of features. Apart from the pivot/unpivot commands that we had discussed in one of the previous whitepapers on our site and analytical functions (ROW_NUMBER, RANK, DENSE_RANK and NTILE), another new helpful feature is the inclusion of an OUTPUT clause in the DML statements. It returns information from each row effected by an INSERT, UPDATE, or a DELETE statement. This new enhancement can help us in archiving and also in auditing of data into separate tables. The results can also be sent back to the calling application for further actions. Major benefit in using this clause is we can avoid triggers and thus the extra overhead that comes with the triggers. Oracle already has similar functionality available via the RETURNING clause.

In this whitepaper, we will show you some examples of the OUTPUT and RETURNING clause and how can we use them. We will start with the OUTPUT clause of SQL Server 2005. For basic syntax of OUTPUT clause, please refer to BOL.

## Sample data

```
/*************************************************************************
* Create new table Invoice and populate it with sample data. Also create another empty
*  table Audit_Invoice to store audited data.
*************************************************************************/
CREATE TABLE dbo.Invoice
(
 INVOICE_NUMBER        INT IDENTITY(1,1)             NOT NULL,
 INVOICE_DATE          DATETIME                      NOT NULL,
 CLIENT_ID             INT                           NOT NULL,
 INVOICE_AMT           NUMERIC(9,2) DEFAULT 0        NOT NULL,
 PAID_FLAG             TINYINT DEFAULT 0             NOT NULL, -- 0 Not paid/ 1 paid
 CONSTRAINT PK_INVOICE PRIMARY KEY(INVOICE_NUMBER)
)
-- FILEGROUP Clause
GO

CREATE TABLE dbo.Audit_Invoice
(
 AUDIT_INVOICE_ID      INT IDENTITY(1,1)             NOT NULL,
 ACTION                VARCHAR(2)                    NOT NULL,
 INVOICE_NUMBER        INT                           NOT NULL,
 INVOICE_DATE          DATETIME                      NOT NULL,
 CLIENT_ID             INT                           NOT NULL,
 INVOICE_AMT           NUMERIC(9,2)                  NOT NULL,
 PAID_FLAG             TINYINT                       NOT NULL,
 CONSTRAINT PK_AUDIT_INVOICE PRIMARY KEY(AUDIT_INVOICE_ID)
)
-- FILEGROUP Clause
GO

INSERT INTO INVOICE(Invoice_date, client_ID, Invoice_Amt)
VALUES(getDate()-50,101,1100.00);
```

```
INSERT INTO INVOICE(Invoice_date, client_ID, Invoice_Amt)
VALUES(getDate()-49,102,1100.00);
INSERT INTO INVOICE(Invoice_date, client_ID, Invoice_Amt)
VALUES(getDate()-50,103,1100.00);
INSERT INTO INVOICE(Invoice_date, client_ID, Invoice_Amt)
VALUES(getDate()-40,101,1100.00);
INSERT INTO INVOICE(Invoice_date, client_ID, Invoice_Amt)
VALUES(getDate()-38,101,1500.00);
INSERT INTO INVOICE(Invoice_date, client_ID, Invoice_Amt)
VALUES(getDate()-41,102,1100.00);
INSERT INTO INVOICE(Invoice_date, client_ID, Invoice_Amt)
VALUES(getDate()-41,103,1100.00);
```

## SQL Server 2005

OUTPUT clause makes use of the "magic" tables: deleted and inserted tables. These are special tables and in SQL Server 2000 and below they were accessible only from triggers. Inserted table is populated whenever record is inserted or updated into the table. Similarly deleted table is populated when record is deleted or updated in the table. With the introduction of the OUTPUT clause, now they are accessible outside of the trigger as well.

### Advantages

- One of the advantages of using OUTPUT clause is the reduced database trip and this increase in performance specially, when user is working on larger data set. We can get the values of the columns for the table for which insert took place without going back to the database to query the record we inserted.
- Less database trips result in fewer network trips, fewer cursors and hence less cpu time and less memory requirements.
- It can be used to audit the data for insert/ update/ delete actions without having need of trigger. This reduces the overhead of maintaining the audit triggers.
- Deleted data can be archived to separate tables.
- Dataset can be returned back to calling application for further processing.

Following is the example scenario in which we audit the invoice table for any DML operations. If record gets inserted, updated or deleted we write a record in the audit_invoice table using OUTPUT clause.

### Using OUTPUT clause with a simple INSERT statement

Let us audit all the records which get inserted into Invoice table. Following is the SQL to create record in the Audit_Invoice table when record is inserted in the Invoice table.

```
INSERT dbo.INVOICE(Invoice_date, client_ID, Invoice_Amt, paid_flag)
OUTPUT 'I',inserted.Invoice_number,inserted.invoice_date, inserted.client_id,
       inserted.invoice_amt, inserted.paid_flag
  INTO AUDIT_INVOICE
VALUES(getDate()-2,102,1100.00,1)
```

## Using OUTPUT clause with INSERT INTO SELECT statement

Here we are adding some more records to invoice table from already inserted data to show how INSERT INTO SELECT syntax works.

```
INSERT dbo.INVOICE(Invoice_date, client_ID, Invoice_Amt, paid_flag)
OUTPUT 'I',inserted.Invoice_number,inserted.invoice_date, inserted.client_id,
       inserted.invoice_amt, inserted.paid_flag
  INTO dbo.AUDIT_INVOICE
SELECT Invoice_date, client_ID, Invoice_Amt, paid_flag
  FROM dbo.INVOICE
 WHERE INVOICE_NUMBER <= 5
```

## Using OUTPUT clause with UPDATE statement

Now we will update some records in the Invoice table and will make sure that updated record is inserted into audit_Invoice table. Here we are getting values from only inserted table but one can get values from both inserted and deleted table as per his/her need because as a result of update statement both inserted and deleted tables get populated.

```
UPDATE dbo.INVOICE
   SET paid_flag = 0
OUTPUT 'U',inserted.Invoice_number,inserted.invoice_date, inserted.client_id,
       inserted.invoice_amt, inserted.paid_flag
  INTO dbo.AUDIT_INVOICE
 WHERE invoice_number = 2
```

## Using OUTPUT clause with DELETE statement

Now let us try deleting record from the Invoice table.

```
DELETE FROM dbo.INVOICE
OUTPUT 'D',deleted.Invoice_number,deleted.invoice_date, deleted.client_id,
deleted.invoice_amt,
       deleted.paid_flag
  INTO dbo.AUDIT_INVOICE
 WHERE invoice_number <= 2
```

## Restrictions

Even though the OUTPUT clause is very useful, it has some restrictions as well. Following are some of the restrictions where OUTPUT clause cannot be used.

- When used, OUTPUT Clause is getting fired before trigger is fired so if you have a trigger on the table to alter the column value which is included in the OUTPUT clause, it may not contain correct value. We need to be an extra careful in such cases.
- Table cannot have check constraints.
- Table cannot contain the trigger which is in enabled state. If trigger is in place and is enabled then OUTPUT INTO clause should be used to store output values in either table or table variable.
- OUTPUT clause cannot support the DML statements that reference local or distributed partition views and remote tables.

- OUTPUT INTO clause cannot be used to insert into view.

## Oracle

Oracle was the first one to implement the functionality to return the effected rows by DML statements. Unlike the OUTPUT clause in SQL Server, it is called the RETURNING clause in Oracle. It exists since 8i days with new enhancements of returning single-set aggregation in 10g release. In this we will examine how we can use RETURNING clause to return values/rows affected by DML statements.

We can either return single value or we can bulk collect the affected values in collection variable and return it. Following is the example in PL/SQL block but it can be used in procedures and functions in similar way. Example demonstrates the use of RETURNING clause in INSERT/ UPDATE and DELETE statement. If we are using release 9.2.0.1 and above, instead of declaring individual variables, we can declare record type variable and bulk collect into it. One can specify this clause for tables, materialized views and views having single table. For detailed syntax, please refer SQL Reference manual by oracle.

Let us create empty table with sample data.

```
CREATE TABLE Invoice
(
 INVOICE_NUMBER        NUMBER(9) NOT NULL,
 INVOICE_DATE          DATE NOT NULL,
 CLIENT_ID             NUMBER(9)      NOT NULL,
 INVOICE_AMT           NUMBER(9,2) DEFAULT 0 NOT NULL,
 PAID_FLAG             NUMBER(1) DEFAULT 0 NOT NULL, -- 0 Not paid/ 1 paid
 CONSTRAINT PK_INVOICE PRIMARY KEY(INVOICE_NUMBER)
)
-- TABLESPACE Clause
/

CREATE SEQUENCE INVOICE_SEQ
START WITH 1
CACHE 100
/


CREATE OR REPLACE TRIGGER TIB_INVOICE
BEFORE INSERT ON INVOICE FOR EACH ROW
DECLARE
BEGIN
IF :NEW.INVOICE_NUMBER IS NULL THEN
SELECT INVOICE_SEQ.NEXTVAL INTO :NEW.INVOICE_NUMBER
FROM DUAL;
END IF;

END;
/


INSERT INTO INVOICE(Invoice_date, client_ID, Invoice_Amt) VALUES(sysdate-50,101,1100.00);
INSERT INTO INVOICE(Invoice_date, client_ID, Invoice_Amt) VALUES(sysdate-49,102,1100.00);
INSERT INTO INVOICE(Invoice_date, client_ID, Invoice_Amt) VALUES(sysdate-50,103,1100.00);

INSERT INTO INVOICE(Invoice_date, client_ID, Invoice_Amt) VALUES(sysdate-40,101,1100.00);
INSERT INTO INVOICE(Invoice_date, client_ID, Invoice_Amt) VALUES(sysdate-38,101,1500.00);
INSERT INTO INVOICE(Invoice_date, client_ID, Invoice_Amt) VALUES(sysdate-41,102,1100.00);
INSERT INTO INVOICE(Invoice_date, client_ID, Invoice_Amt) VALUES(sysdate-41,103,1100.00);
```

## Using RETURNING clause in an INSERT statements

In this example, we will see how we can get the value of invoice_number (generated via sequence / trigger combination) and value of paid_flag column which is not part of insert statement and is assigned default value. Run following pl/sql block from SQL*Plus command.

```
SET SERVEROUTPUT ON
DECLARE

   v_InvID   Invoice.invoice_number%type;
   v_InvFlag Invoice.paid_flag%type;

BEGIN

   INSERT INTO Invoice(Invoice_Date,Client_ID, Invoice_Amt)
   VALUES(sysdate, 200, 1000)
      RETURNING invoice_number, paid_flag INTO v_InvID, v_InvFlag;

   COMMIT;
   dbms_output.put_line('Invoice_number = ' || v_InvID || '.. Paid = ' || v_InvFlag);

END;
/
```

One thing to remember for INSERT statement is that we cannot use 'insert into select' statement. It should be VALUES clause only. Also as per documentation, we cannot use single-set aggregation in an INSERT statement.

## Using RETURNING clause in UPDATE statements

In this example, we will see how we can use bulk collect in returning clause.

```
DECLARE
   TYPE IDList IS TABLE OF Invoice.Invoice_Number%TYPE;
   TYPE PList IS TABLE OF Invoice.paid_Flag%TYPE;

   InvID   IdList;
   InvFlag PList;

BEGIN

   UPDATE Invoice
      SET PAID_FLAG = 1
    WHERE Invoice_Number <= 2
      RETURNING invoice_number, paid_flag
           BULK COLLECT INTO InvID, InvFlag;
   COMMIT;
   FOR i IN 1..InvID.count
   LOOP
        dbms_output.put_line('Invoice_number  =  '  ||  InvID(i)  ||  '  Paid  =  '  ||
InvFlag(i));
   END LOOP;

END;
/
```

## Using RETURNING clause in DELETE statements

In this example, we will see how we can use bulk collect in returning clause but this time we will use the dynamic sql.

```
DECLARE
   TYPE IDList IS TABLE OF Invoice.Invoice_Number%TYPE;
   TYPE PList IS TABLE OF Invoice.paid_Flag%TYPE;

   InvID   IdList;
   InvFlag PList;
   v_Sql_Stmt VARCHAR(300);
   v_invoice_number Invoice.invoice_number%type := 2;

BEGIN
   v_sql_Stmt := 'DELETE FROM Invoice WHERE Invoice_Number <= :invoice_number
                     RETURNING invoice_number, paid_flag
                          INTO :InvID, :InvFlag';

   EXECUTE IMMEDIATE v_sql_Stmt USING v_Invoice_number
      RETURNING BULK COLLECT INTO InvID, InvFlag;
   COMMIT;
   FOR i IN 1..InvID.count
   LOOP
      dbms_output.put_line('Invoice_number = ' || InvID(i) || ' Paid = ' || InvFlag(i));
   END LOOP;
END;
/
```

## RETURNING single-set aggregation in UPDATE/ DELETE statements (10g)

In the following example, we will see how we can return aggregated value as a result of update and/or delete statement. One thing to keep in mind is, it will return aggregation of only affected rows.

```
DECLARE
   v_update_paid NUMBER(9,2) := 0;
   v_delete_paid NUMBER(9,2) := 0;

BEGIN
   -- Update
   UPDATE Invoice
      SET paid_flag = 1
    WHERE Invoice_Number <= 5
      RETURNING sum(invoice_Amt) INTO v_update_paid;

   -- Delete
   DELETE Invoice
    WHERE Invoice_Number <= 4
      RETURNING sum(invoice_Amt) INTO v_delete_paid;

   dbms_output.put_line('Update Total = ' || v_update_paid);
   dbms_output.put_line('Delete Total = ' || v_delete_paid);
END;
/
```

## Restrictions

Considering scenarios where RETURNING clause can be used effectively, it comes with certain limitations as well. Some of them are

- This clause cannot be defined for multitable insert.

- When used in UPDATE / DELETE statements, either single-set aggregation function expression or simple expression can be returned but not both.
- Cannot be specified for a view with instead of triggers on it.
- Aggregate functions are not supported in an INSERT statement returning clause.
- Long datatypes cannot be retrieved using this clause.
- Merge statement does not support RETURNING clause.

## Summary

In this whitepaper, we have explored the OUTPUT clause for SQL Server 2005 and the RETURNING clause of Oracle. With examples, we have shown how both can be used and what are the limitations of each one of them. In scenarios where we need to perform certain actions, be it auditing, archiving, knowing database generated values, computed values or populating some other transaction tables based on the rows affected by DML statement, this functionality is very useful. We incur less network round trips, fewer database calls and fewer cursors overall reducing CPU time and memory requirements.