Returning Modified Rows – SELECT Statements with Side Effects

Andreas Behm, Serge Rielau, Richard Swagerman

IBM Toronto Lab
8200 Warden Avenue
Markham, ON
Canada
{abehm, srielau, swagrman}@ca.ibm.com

Abstract

SQL in the IBM® DB2® Universal Database™ for Linux®, UNIX®, and Windows® (DB2 UDB) database management product has been extended to support nested INSERT, UPDATE, and DELETE operations in SELECT statements. This allows database applications additional processing on modified rows. Within a single unit of work, applications can retrieve a result set containing the modified rows from a table or view modified by an SQL data-change operation. This eliminates the need to select the row after an INSERT or UPDATE, or before a DELETE statement. As a result, fewer network round trips, less server CPU time, fewer cursors, and less server memory are required. In addition, deadlocks can be avoided. The proposed approach is integrated with the set semantics of SQL, and does not require any procedural logic or modifications on the underlying relational data model. Pipelining multiple update, insert and delete operations using the same source data provides a very efficient way for multi-table data-change statements typically found in ETL (extraction, transformation, load) applications. We demonstrate significant performance benefit with our experiences in the TPC-C benchmark. Experimental results show that the new SQL is more efficient in query execution compared to classic SQL.

Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004

1. Introduction

Commercial DBMS vendors constantly extend the SQL query language to address emerging business demands for increasing functionality and improving performance. One aspect of new features is to push more processing into the database engine. While most language extensions apply to data retrieval, extending UPDATE, DELETE, and INSERT statements (henceforth called *data-change statements*) has found less attention.

Prominent language extensions in the context of data-change statements are triggers, MERGE statement, identity columns using sequences, expression-generated columns, and default values. Modern RDBMS can use these features to produce surrogate keys. In doing so it becomes apparent that a way needs to be found to retrieve data back from data-change statements, which traditionally only provide a very limited set of information. The information returned today is usually limited to the number of rows changed.

Another area where performance of data-change statements plays a key role is a data-cleansing environment, where large quantities of data need to be imported into the RDBMS. The data needs to be processed efficiently in several phases, and dispatched into the appropriate tables such as fact and dimension tables in a star-schema scenario.

There are band-aid solutions in the industry today, which allow the return of specific properties. Examples include functions to return the last generated *identity_value* or sequence value to retrieve generated primary keys. Other more general solutions allow for returning data back into the procedural context using temporary tables or set-valued host variables.

We found that the problem of returning data from data-change statements needs to be solved in a more holistic and set-oriented fashion. A general, more relational approach needs to address more complex scenarios such as data cleansing in addition to covering the simple cases described above.

One of the challenges the authors faced was to maintain the spirit of the relational model within the context of standard SQL while allowing for optimal performance through pipelining and parallelization.

In the following sections, we propose a fairly small set of changes SQL, which result in a major extension to the expressive power of SQL. The changes include:

- The exposure of the NEW and OLD transition table as defined for SQL statement triggers in the FROM clause of an SQL SELECT statement.
- A definition of the order of execution for nested queries.
- An extension of the column lists for data-change statements using an include clause.

These three simple changes allow data-change statements to fully integrate with the select statement and hence exploit the expressive power of SQL.

The structure of this paper is as follows. Section 2 introduces our major SQL extensions. In Section 3 we discuss how we approached the implementation of the new statement type. We demonstrate a pipelining mechanism for update, delete and insert operations in Section 4, which leads to significant performance benefit for multi-table data-change statements. In Section 5 we discuss our experiences with the new SQL in the TPC-C benchmark, and the performance benefit of OLTP applications. Related approaches for returning modified data are discussed in Section 6. Section 7 provides a conclusion.

2. SQL Changes

In this section, we introduce the SQL extensions for data-change statements. We show how `insert`, `update`, or `delete` operations can be embedded in the `from` clause of a `select` statement. We enhance common table expressions to provide a way for embedding multiple data-change operations within one SQL statement. Finally, we introduce modifying table functions, which allow more complex scenarios for returning modified data. All language extensions introduced in this chapter are available in DB2 UDB for Linux, UNIX, and Windows Release 8.1.4. The complete new SQL can be found in the SQL Reference [5].

Select From Data-Change Statement

In the simplest form, an SQL data-change statement is characterized as a select statement having an `insert`, `update` or `delete` operation embedded in the `from` clause. The columns of the target object of the specified SQL data-change statement are considered the columns of this intermediate result table and can be referenced by name in the select list of the query.

```
create table orders (
      purchase_date date,
      sales_person varchar(16),
      region varchar(10),
      quantity varchar,
      order_num integer not null
         generated always as identity
         (start with 100, increment by 1))

select * from new table
     (insert into orders
           (pdate, salesp, region, quantity)
      values
       (current date,'Judith','Beijing',6),
       (current date,'Marieke','Medway',5),
       (current date,'Hanneke','Halifax',5))

PDATE       SALESP   REGION    QUANT  ONUM
----------- -------- --------- ------ ----
12/22/2003  Judith   Beijing   6       100
12/22/2003  Marieke  Medway    5       101
12/22/2003  Hanneke  Halifax   5       102
```

Note that the syntax of the `insert` statement is not changed by this approach. The `insert` operation is wrapped in a `from` clause to indicate that the intermediate result table represented by the inserted rows should be returned. A data-change-table-reference can be specified as the only table-reference in the `from` clause of the outer fullselect that is used in a select-statement, a `select into` statement, a common-table expression, or as the only fullselect in an assignment statement. To execute select from data-change statements, the user must have the proper SQL authorization. For example, to perform a select from insert statement on a table `ORDERS`, the user would have to hold both, select and insert privileges on the `ORDERS` table.

The contents of the intermediate result table dependend on the qualifier specified in the `from` clause. If `old table` is specified, the rows in the intermediate result table will contain values of the target table rows at the point immediately preceding the execution of before triggers and the SQL data-change operation. For the `new table` qualifier, the rows in the intermediate result table will contain values of the target table rows at the point immediately after the SQL data-change statement has been executed, but before referential integrity evaluation and the firing of any after triggers. The `old table` qualifier applies to `update` and `delete` operations, the `new table` qualifier applies to `update` and `insert` operations.

Include Columns

We introduce the concept of include columns, which allows you to specify additional columns that do not exist in the target table of a select from data-change statement. These additional columns are available for use in the `select` list or `order by` clause of the query containing the SQL data-change statement in the `from` clause, but have no effect on the SQL data-change of the target table. One typical example for using include columns is to provide a way for update statements to return both, the new and the old value of a column. The following select from update statement defines an include column `old_salary`, and assigns the salary value to `old_salary` in the `set` clause of the `update` operation. Additional examples can be found in chapter 4 and the IBM TPC-C disclosure report [9].

```
create table employee (
  ssn char(10), salary integer);
insert into employee values
  ('1234567890', 90000);

select * from new table
     (update employee
      include (old_salary integer)
      set old_salary = salary,
          salary = salary * 1.2);


ssn        salary   old_salary
---------- -------- ----------
1234567890 108000   90000
```

Views

Special care needs to be taken when the target of a select from data-change statement is a view containing a where clause. A view containing a where clause in its definition is by default non-symmetric. That is, a row modified by an `insert` or `update` operation does not need to remain in the view. In contrast, a symmetric view is defined by specifying the `with cascaded check option` clause during the creation of the view, which indicates that an inserted or updated row has to remain in the view after the modification [4].

Non-symmetric views pose a security issue for select from data-change statements if, for example, a before trigger of an `insert` or `update` operation modifies a value of the row which should not be seen by the user. We concluded that non-symmetric views as the target of a select from data-change statement must satisfy the restrictions of symmetric views, if the qualifier `new table` is used. The following example shows a view containing employees having a salary less than 100.000, and a before trigger modifying the salary if the employee is promoted. The given select from update statement is rejected because the modified row with the updated salary (i.e., 108000) does not remain in the view. No row is updated or returned.

```
create table employee (
   ssn char(10),
  salary integer,
  ranking integer);

insert into employee values
  ('1234567890', 90000, 3);

create trigger promote
before update on employee
referencing old as o new as n for each row
when (n.rank > o.rank)
  set n.salary = n.salary * 1.2;

create view lowemps as
select * from employee
where salary < 100000;

select * from new table (
  update lowemps set rank = rank+1);
```

Common Table Expressions

In addition to simple select from data-change statements, we want to support a way to perform multiple data-change operations within one SQL statement. This is in particular useful for complex data-change scenarios, where the result of one data-change operation is used as input for another one. It allows us to construct better performing plans, since we can see multiple operations together when we rewrite and optimize a statement. Furthermore, it is not necessary to return the result of a select from data-change statement to the client in order to use the data as input for another statement.

We use common table expressions for this purpose, and allow select from data-change statements in the definition of temporary views in a common table expression. The main advantage of using common table expressions for embedding multiple data-change statements is that the syntax specifies an order for temporary views, which determines the semantics of executing a common table expression. All temporary views are executed in the order they occur in the common table expression.

In the following example, we illustrate common table expressions using two tables `Employee (EmpNr, Name)` and `Project (ProjNr, Name, Lead)`. The common table expression replaces an employee '`Old Emp`' with a new employee `'New Emp'`, assigns all projects lead by `'Old Emp'` to `'New Emp'`, and returns the names of the updated projects.

```
with
  NewEmp AS (select EmpNr from new table
             (insert into Employee(name)
              values ('New Emp'))),
  OldEmp AS (select EmpNr from Employee
             where Name = 'Old Emp'),
  UpProj AS (select Name from new table
             (update Project
              set Lead = (select EmpNr
                          from NewEmp))),
  DelEmp AS (select EmpNr from new table
             (delete from Employee
              where EmpNr = (select EmpNr
                             from OldEmp)))
select Name from UpProj
```

Modifying Table Functions

We introduce modifying table functions as another concept for embedding multiple data-change operations in a single statement. In contrast to common table expressions, two additional tasks can be accomplished. First, the body of a modifying table function can be defined as a compound statement containing a sequence of SQL-procedure-statements [5] including data-change statements. Second, it allows applying a data-change operation multiple times for a collection of input data, as shown in the example below. Both features have been widely used in the implementation of our TPC-C benchmark [9].

A select statement can contain one modifying table in the `from` clause. To guarantee order of execution in case the `from` clause contains more than one table reference, the modifying table function has to be the last table reference in the from clause, correlated to all other table references.

The following table function `upsal()` updates the salary of an employee, records the salary change in an `audit` table, and returns the salary increase. The subsequent select statement updates the salary using the `upsal()` function for three employees.

```
create function upsal(upeid int,
                      factor float)
returns table (increase int)
modifies sql data
return
with i1 as (
select eid, old_salary, salary
       from new table
           (update emp
            include (old_salary int)
            set salary = salary * factor,
                old_salary = salary
            where emp.eid=upeid))
select  new_salary-old_salary
from new table (
  insert into audit select
  eid,old_salary,salary from i1);
```

```
select sum (increase)
from table (values (1, 1.1),
                   (2, 1.2),
                   (3, 1.05))
            as upemp(eid,factor),
     table (upsal(upemp.eid,upemp.factor))
            as upsal;
```

The update effect on the `emp` and `audit` table is equivalent to the following sequence of `insert` and `update` statements:

```
insert into audit
with i1(eid,factor) as
  (values (1, 1.1),
          (2, 1.2),
          (3, 1.05))
select emp.eid, emp.salary,
       emp.salary * factor
from emp, i1
where emp.eid = i1.eid;

update emp set salary = salary * 1.1
           where eid=1;
update emp set salary = salary * 1.2
           where eid=2;
update emp set salary = salary * 1.05
           where eid=3;
```

3. Implementation

In this section, we provide a high-level summary of some of the key design issues we faced during implementation of the new SQL features. The proposed approach mostly takes advantage of existing "tooling" within and around the relational database engine. In the query compiler, we mostly applied "plug-and-play" of existing infrastructure, that is, combining constructs used for representing `select` statements as well as `insert`, `update` and `delete` statements. The optimizer is already capable handling data-change operations in a complex statement to ensure, for example, that predicates are not pushed down through data-change operations. We did not need to implement any changes for the client infrastructure.

The first interesting problem when executing a select from data-change statement is to get hold on the new and old values of the rows modified by an `insert`, `update` or `delete` operation. Fortunately, this functionality has already been implemented for after statement triggers [3]. An after statement trigger for data-change statements can access the new and old transition table containing all modified rows. The following example shows an after update statement trigger for the `emp` table. The trigger inserts the content of both, old and new transition table into the `audit` table.

```
create table emp
(eid int, name varchar(10), salary int);
insert into emp values (1, 'Peter',50000),
                       (2, 'Paul',60000),
                       (3, 'Mary',70000);
create table audit(eid int, salary int);

create trigger audtrig
after update on emp
referencing old_table as old
            new_table as new
for each statement
insert into audit
  select eid,salary from old union all
  select eid,salary from new);

update emp set salary=salary*1.1
where eid<3;

select * from audit;

EID         SALARY
----------- -----------
          1       55000
          2       66000
          1       50000
          2       60000
```

The data flow for select from data-change statements has to be constructed in a way that either the new or old transition table is returned to the client instead of flowing into a trigger.

A table that has been modified by a data-change operation within a complex statement can be accessed in the same statement through a subquery or another data-change operation. Consequently, all read and write operations on the same data within one statement need to be synchronized. The infrastructure for handling read/write conflicts is already required for standard SQL. For example, a before trigger or an after trigger can contain a subquery over the target table. The same concepts have been extended to ensure correct semantics for select from data-change statements. In the following, some examples for read/write conflicts are shown.

```
select c1, (select sum(salary) from emp)
from new table (insert into emp(eid,salary)
                values (1,50000));

with v1 as (select * from new table
             (update emp
              set salary=salary+1000)),
     v2 as (select * from new table
             (update emp
              set salary=salary*1.1))
select * from emp
```

In the first example, the subquery `select sum(salary) from emp` includes the salary 50.000 inserted in the nested `insert` operation. In the second example, the table `emp` is updated twice. First, the salary is increased by 1000 for each row, the intermediate result is then multiplied by 1.1. The fullselect `select * from emp` returns the final result in the `salary` column after executing both `update` operations.

The most fundamental change in the infrastructure we made was the execution of select from data-change statements at runtime. In general, a select statement returning more than one row is performed using a cursor. The typical execution of a cursor consists of three consecutive tasks: declare, open, and fetch. The statement is compiled when the cursor is declared. In theory, opening the cursor completes the operation and positions the cursor on to the first row of the result set. Each row of the result set can be accessed through the fetch command, which moves the cursor towards the end of the result set. However, in practice we are trying to avoid executing a statement completely at one time, as it requires temporary storing the result.

Instead, we execute the statement as we fetch the rows, such that no temporary storage for the result set needs to be used, unless, for example, the statement contains an `order by` clause and the statement needs to be completely executed in order to determine the first row.

For select from data-change statements, this strategy does not work. Any `insert`, `update` or `delete` operation is an atomic operation which needs to be driven to completion at one time. Data-change operations cannot be executed on a row-by-row basis. Instead, they are processed as follows: first, the before triggers for all modified rows are performed. Second, all insert, update, and delete operations are applied. Third, constraints for all modified rows are checked. Finally, all after statement triggers are executed.

We have chosen to completely execute select from data-change statements at cursor open time. This allows applications to keep X locks for a minimum amount of time when using cursors specified as `with hold`, and committing the transaction right after opening the cursor. The following example shows typical use of cursors for select from data-change statements:

```
declare emp_cur cursor with hold for
   select id from new table
     (insert into emp(name)
      values 'Peter', 'Paul', 'Mary');

open emp_cur;
commit;

fetch emp_cur;
...
```

All X locks are acquired at cursor open time, and immediately released at commit. The application can then fetch the result set without blocking other transactions.

We call the execution at cursor open time *do-at-open* semantics. In order to achieve do-at-open semantics, we need to teach the optimizer to include an additional operator at the top of the generated plan, which is called a zero-key sort operator. A zero-key sort is a specialized sort operator where the set of keys is empty. This concept has several advantages. First, it is an elegant way to drive execution of a statement to completion using an existing, slightly modified operator. Second, we are using existing infrastructure in terms of the sort buffer to efficiently store the result set. Note that a zero-key sort does not actually sort any data, it only indicates that runtime has to drive the execution of the statement to completion.

In the following, we illustrate the do-at-open semantics using two interleaving transactions T1 and T2 in isolation level *uncommitted read*. T1 declares a cursor for a select from insert statement modifying a table `emp`, T2 reads the content of table `emp`. T2 can see all the rows inserted by T1 as soon as the cursor is opened.

T1	T2

`declare emp_cur`
`cursor for`
`select id from new table`
`  (insert into emp(name)`
`   values 'Peter',`
`          'Paul',`
`          'Mary');`	

	`select id from emp;`
	`0 rows selected`

`open emp_cur;`	

	`select * from emp;`
	`1`
	`2`
	`3`
	`3 rows selected`

`fetch emp_cur;`
`1`
`fetch emp_cur;`
`2`
`...`	

Since our approach for returning modified rows makes use of select statements as the data carrier, no change was needed for the client infrastructure. Regardless of which client is used to perform select from data-change statements, the statement is always considered a select statement, and the APIs can be used respectively. Typically, a cursor containing a select from data-change statement can be declared, and after opening the cursor the modified rows are available in the result set for fetch.

4. Pipelining

The fact that multiple data-change operations can be embedded in a single SQL statement gives the compiler a chance to optimize the data flow between operations. In this section, we describe how performance of multi-table data-change statements can be significantly increased when constructing a plan as a pipeline of insert, update, and delete operations.

In the following example, we introduce two tables `emp` and `mgr` used as target for insert operations, and a table `src` containing some source data for both, employees and managers.

```
create table emp (
  eid    int not null primary key,
  salary int);

create table mgr (
  eid   int not null primary key,
  bonus int);

create table src (
  eid    int not null primary key,
  salary int,
  bonus  int,
  ismgr  char(1));
```

We are now looking for an SQL statement which transfers data from the source table to the two target tables. The first example shows a classic SQL solution with two `insert` statements, embedded in an atomic compound statement. The plan shows that the two `insert` statements are executed in sequential order, reading the data from the source table twice.

```
begin atomic
insert into emp select eid,salary from src;
insert into mgr select eid,bonus from src
 where ismgr='Y';
end;
```

```
                  RETURN
                    |
                  FILTER
                    |
  +-------------+---+------------+
TBSCAN       INSERT           INSERT
  |             |                |
  |         /---+---\        /---+---\
TABFNC:  TBSCAN   TABLE:  TBSCAN   TABLE:
GENROW     |      MGR       |      EMP
           |                |
         TABLE:           TABLE:
         SRC              SRC
```

With a common table expression and two nested select from insert operations, the query can be expressed in a way that the result of the first insert operation is used as source for the second one. In the generated plan, the two insert operations are stacked upon each other, which allows pipelining the data flow through both operations.

```
with i1 as (
  select eid,bonus,ismgr
  from new table (
         insert into emp
         include (bonus int, ismgr char(1))
         select eid,salary,bonus,ismgr
         from src))
select count(*) from new table (
 insert into mgr
 select eid,bonus from i1 where ismgr='Y');
```

```
                  RETURN
                    |
                  TBSCAN
                    |
                  SORT
                    |
                  GRPBY
                    |
                  INSERT
                    |
               /---+---\
            FILTER   TABLE: MGR
              |
            INSERT
              |
          /---+---\
     TBSCAN   TABLE: EMP
       |
 TABLE: SRC
```

With the above approach, the total cost of the query is reduced by 20% and the CPU cost is reduced by 40%, independent of the number of rows in the source table.

The example above has the property that all rows inserted into the `mgr` table must be inserted into the `emp` table as well. An insert operator is not able to pass rows not participating in the insert operation to the following operator. Consider a revised example, where the source data is partitioned into employees and manager tables. Rows in the `src` table are inserted into either `emp` or `mgr` table, but not into both.

```
create table emp (
  eid    int not null primary key,
  salary int);

create table mgr (
  eid    int not null primary key,
  salary int,
  bonus  int);
```

In order to solve this problem, we propose an extension of our approach to support the `merge` statement in the `from` clause of a `select` statement. The `merge` statement is a combination of `insert`, `update`, and `delete` statements, and allows conditional `insert`, `update`, and `delete` operations on a table. We need to extend the `merge` statement with a `set` clause, so that rows not participating in any data-change operation can be returned.

The following SQL statement contains nested select from merge and select from insert operations. The merge operation inserts source rows into the `emp` table where `ismgr=’N’`, and includes source rows where `ismgr=’Y’` in the result of the inline view `i1`. The subsequent insert operation inserts the rows from `i1` where `ismgr=’Y’` into the `mgr` table.

```
with i1 as (
  select eid,salary,bonus from new table (
    merge into emp
    include (bonus int, ismgr char(1))
    using src on (1=0)
    when not matched and ismgr=’N’ then
      insert (eid,salary)
      values (src.eid,src.salary)
    when not matched and ismgr=’Y’ then
      set eid    = src.eid,
          salary = src.salary,
          bonus  = src.bonus,
          ismgr  = src.ismgr)
select count(*) from new table (
  insert into mgr
    select eid,salary,bonus
    from i1 where ismgr=’Y’)
```

SQL queries such as above are very common in the area of ETL (Extraction, Transformation, Load), for transforming, cleansing and integrating data from operational databases to a data warehouse. With the full support the merge statement in the `from` clause of a select statement, we are able to provide efficient plans for multi-table insert, update, and delete statements.

5. TPC-C Benchmark

In this section, we describe the impact of the new SQL statement type on the result of the TPC-C benchmark. The TPC-C benchmark [10] represents a typical workload for online transaction processing, based on an order-entry application for a wholesale supplier. The logical database design is composed of 9 relations: Warehouse, District, Customer, Stock, Item, Order, New_Order, Order_Line, and History. A more detailed description of the schema can be found in [7][10].

The application defines five types of short, moderately complex transactions: entering and delivering orders, recording payments, checking the status of orders, and monitoring stock level. Three of these transactions modify data: entering and delivering orders, and recording payments.

In the following sections, we introduce how the three modifying transactions are implemented using classic

SQL statements. We illustrate how we rewrite the statements using the new select from data-change statement type. For better readability, we will use simplified examples out of the three transactions to illustrate the benefit of the new SQL. The complete SQL statements we used for the transactions can be found in the disclosure report [9].

Generally, several assumptions are made for improving performance of OLTP applications. First, the codepath in the database engine for executing transactions should be reduced. Second, network traffic can be reduced through decreasing the number of I/O operations between client and database. Finally, lock contention should be minimized and deadlocks avoided.

New Order Transaction

The New Order transaction places an order for an average 10 items from a warehouse. The following database operations (in a simplified pseudocode) are required: first, the new order ID is retrieved from the District table; for each item, the stock level will be updated; the order is stored in two tables Order and New_Order (containing pending orders), and each item is stored in the table Order_Line. The Stock table is updated 10 times, so in total 17 statements are executed in classic SQL.

1. Select from District
2. Update District
3. Select from Item
4. for each item: Update Stock
5. Insert into Order_Line
6. Insert into Orders
7. Insert into New_Order
8. Select from Warehouse, Customer

Our New Order transaction contains only three SQL statements. The first statement implements the first two operations, select and update the District table. This table contains an ID and an order number, which represents the next order number to be used for a new order in this district. The new order transaction needs to read and increment the next order ID from the DISTRICT table. With classic SQL, two separate statements are required for this task:

```
select d_next_o_id into :new_id
from district
where d_id = :district_id;

update district
set d_next_o_id = d_next_o_id+1
where d_id = :district_id;
```

The approach above has two problems. First, we observe two I/O operations, one for each statement, such that the row in the district table is fetched twice. Second, the fact that we first read the district row and subsequently update the same row, is causing lock conversion. This is a typical scenario for a deadlock, where two transactions concurrently read the same row, and later on attempt to write it, requesting to upgrade a read lock to a write lock [1].

With the new SQL, the scenario can be implemented in one single SQL statement containing a nested update operation. The plan of the statement shows only one table scan for the District table.

```
select d_next_o_id from old table
     (update district
      set d_next_o_id = d_next_o_id+1
      where d_id = :district_id))
```

```
           RETURN
               |
           UPDATE
               |
          /---+--\
      TBSCAN     TABLE:
        |        DISTRICT
     TABLE:
     DISTRICT
```

The second statement implements the operations 3 to 5. A table function is provided which retrieves the price of the item, updates the stock level and inserts a row into the Order_Line table. The following example shows a (simplified) table function, and a select statement referencing the order items and the table function in the from clause.

```
create function new_ol
(i_id int, i_qty int, o_id int, ol_nr int)
returns table (i_price int)
modifies sql data
begin atomic
  declare i_price int;
  set i_price =
    (select i_price from item
     where item.i_id = new_ol.i_id);
  update stock set qty = qty - i_qty
    where s_i_id = new_ol.i_id;
  insert into order_line
    values (o_id, ol_nr, i_id, i_qty,
            (i_price * i_qty));
  return values (i_price * i_qty);
end
```

In the select statement calling the table function, we define a table `ol` containing the new order line items. Then, we join this table with the function `new_ol`, such that every row in `ol` is used as input for `new_ol`. Finally, we return the sum of the total price for each order line.

```
select sum(i_price) into :total_price
from table (values (1, 15, 200),
                   (2, 31, 150),
                   (3, 47, 250))
        as ol(ol_nr, i_id, i_qty),
     table(new_ol(i_id, i_qty, 17, ol_nr))
        as new_ol(i_price);
```

As shown below, the plan of the new SQL statement contains only one update and one insert operation. The subtree below the UNION box represents the table function.

```
          RETURN
            |
          GRPBY
            |
          NLJOIN
            |
   /----------+--------\
TBSCAN               NLJOIN
  |           /---------+---------\
TABFNC:     NLJOIN             TBSCAN
GENROW        |                   |
              |                   |
   /----------+-----------\    TABFNC:
TBSCAN                 UNION   GENROW
  |                      |
  |     +-----------+---+--------------+
TABFNC:  TBSCAN     UPDATE          INSERT
GENROW     |          |                |
           |      /---+---\       /---+--\
         TABLE:  TBSCAN  TABLE:  TBSCAN  TABLE:
         ITEM      |     STOCK     |  ORDER_LINE
                   |               |
                 TABLE:          TABFNC:
                 STOCK           GENROW
```

The third statement implements operations 6 to 8 in another table function.

Payment Transaction

The Payment transaction processes a payment for a customer and updates the Warehouse, District and Customer tables. A History table contains the history of payment transactions. Seven statements are executed for the payment transaction in classic SQL.

1. Select from Customer
2. Update Customer
3. Select from District
4. Update District
5. Select from Warehouse
6. Update Warehouse
7. Insert into History

We managed to collapse all 7 operations into one new SQL statement, providing one table function containing a sequence of 4 select from update and insert statements. The benefit is similar to the New Order transaction, avoiding separate select and update statements.

Delivery Transaction

The Delivery transaction processes one order per district. The next order to be processed is identified by the oldest order number in the New_Order table.

1. Select min(order_id) from New_Order
2. Delete from New_Order
3. Update Order
4. Select sum(ol_amount) from Order_Line
5. Update Order_Line
6. Update Customer

The table function we used for the delivery transaction contains one select from delete and three select from update statements. The deletion of a new order shows a select from delete statement with other SQL features like order by and fetch first in subqueries, and fullselect as target of delete statements. In order to deliver a new order, the oldest order of a given district (i.e., the smallest order id), needs to be deleted and returned. All this can be contained in one SQL statement, which we call a *destructive read*. The plan shows that the New_Order table needs to be scanned only once in order to find and delete the oldest order.

```
select no_o_id, … into :no_o_id, …
from old table (
      delete from (select * from new_order
                   order by no_o_id
                   fetch first row only));
```

```
               RETURN
                 |
               DELETE
                 |
              /---+---\
        FETCH      TABLE:
          |        NEW_ORDERS
       /---+---\
  TBSCAN     TABLE:
    |        NEW_ORDERS
  SORT
    |
  TBSCAN
    |
TABLE:
NEW_ORDERS
```

Summary

In our implementation of the TPC-C transactions, all insert, update, and delete operations are embedded in the from clause of a select statement, or in a modifying table function. As a consequence, we have been able to significantly reduce the total number of SQL statements executed for each transaction. In addition, less data has to be bound into and out of the database server. We summarized the result for the three modifying transactions in Fig. 1 and Fig. 2. The first row shows a remote New Order transaction for a remote good customer, the second row shows a Payment transaction for a good customer identified by ID, and the third row shows a Delivery transaction. For the new order transaction we assume 10 order lines associated.

Fig. 1 shows the number of SQL statements that are executed in one transaction for the classic case and for using the new SQL statement type. The last column shows the total codepath reduction of the transaction on the server achieved by having less runtime overhead for executing SQL statements.

	Stmts classic	Stmts new	Codepath reduction
New Order (remote good)	17	3	11%
Payment (by CID)	7	1	16%
Delivery	61	11	3%

Figure 1

Fig. 2 shows the number of rows passed between application and database server. The first two columns denote how many times we bind in rows from client to server for the classic case and for using the new SQL statement type. The last two columns show the same information for rows returned from server to client.

	Bindin classic	Bindin new	Bindout classic	Bindout new
New Order (remote good)	28	4	24	14
Payment (by CID)	8	1	4	1
Delivery	61	11	21	11

Figure 2

6. Related Work

The idea of returning modified rows has already been adopted by the JDBC^{TM} specification and other database vendors. The JDBC 3.0 Specification [6] introduces in chapter 13.6 an interface for retrieving auto-generated columns of insert statements. A method `Statement.getGeneratedKeys()` is provided, which produces a result set containing the generated key values for all inserted rows. The following example inserts a row into an `ORDERS` table, and returns the generated `ORDER_ID` value:

```
Statement stmt = conn.createStatement();
int rows = stmt.executeUpdate
            ("INSERT INTO ORDERS " +
             "(ISBN, CUSTOMERID) " +
             "VALUES (966431502, 'SAMP')",
             "ORDER_ID");
ResultSet rs = stmt.getGeneratedKeys();
boolean b = rs.next();
if (b == true) {
     // retrieve the new key value
     ...
}
```

Oracle has introduced a `returning` clause for `insert`, `update` and `delete` statements [8]. The `returning` clause specifies which columns are returned, followed by an `into` clause and a set of host variables in which the values are stored. The approach allows returning more than one row, in which case the host variables need to be declared as arrays. Applications require PL/SQL extensions to access the returned data; they are not returned as a result set to the client. Oracle is using the `returning` clause for `update` and `delete` statements in the Delivery and Payment transactions of the TPC-C benchmark. In contrast to the DB2 approach, an `insert` or `update` statement always returns all modified rows, even if the target is a view with a where clause, and a before trigger modifies a value so that it violates the view predicate. The following is an example of an `insert` with `returning` clause (the bind variables must first be declared).

```
INSERT INTO employees
      (employee_id, last_name, email,
       hire_date, job_id, salary)
   VALUES
   (employees_seq.nextval, 'Doe',
    'john.doe@oracle.com',
    SYSDATE, 'SH_CLERK', 2400)
   RETURNING salary*12, job_id
        INTO :bnd1, :bnd2;
```

7. Conclusions

We have proposed and implemented a clean SQL extension to provide applications a way to return values that have been modified by `insert`, `update`, or `delete` operations. The main contributions of our implementation are as follows.

- We provide a way for applications to find out the value of an automatically generated column when a new value is inserted into a column. Similarly, default values or values changed by a before insert or update trigger can be returned to the application.
- We support `insert`, `update` and `delete` operations embedded in the `from` clause of a `select` statement. For more complex scenarios

of data-change operations, modifying table functions can be used.

- The approach is fully embedded in SQL and solves the problem in a set-oriented fashion, using select statements as carrier for returned values, using respective client APIs.
- Common table expressions containing multiple data-change operations provide an efficient way for multi-table `insert`, `update` and `delete` operations using the same source data. The generated plans create a single data flow through all data-change operations, such that source data needs to be read only once.
- Our experiences in the TPC-C benchmark have proven the performance of select from data-change statements. The main reasons are significantly reduced number of SQL statements, generation of better performing plans, less data to be transferred between client and server, and deadlock avoidance.

The idea of returning modified data can be applied to other query languages besides SQL. In particular, the new XQuery language [2] will address update capabilities for XML documents in the future. We expect that update composability and the ability to return updated data will be addressed when designing the language.

References

[1] P. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 1997.

[2] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, Jerome Simeon, and Mugur Stefanescu. XQuery 1.0: An XML Query Language. *World Wide Web Consortium*, Working Draft WD-xquery-2003.

[3] D. Chamberlin. *Using the New DB2*. Morgan Kaufmann, 1996.

[4] R. Cochrane, H. Pirahesh, and N. Mattos. Integrating triggers and declarative constraints in SQL database systems. *In Proc. Inl. Conf. on Very Large Databases*, 1996.

[5] IBM. *DB2 Universal Database$^{(TM)}$ Version 8.1.4*. 2003

[6] John Ellis, Linda Ho, and Maydene Fisher. *JDBC^{TM} 3.0 Specification*, 2001.

[7] Scott T. Leutenegger and Daniel Dias. A Modeling Study of the TPC-C Benchmark. *In Proc. ACM SIGMOD Int. Conf. On Management of Data*, 1993.

[8] Oracle^{TM} Database. *SQL Reference 10g Release 1*.

[9] TPC Benchmark C Full Disclosure Report. *IBM eServer™ pSeries® 690 Model 7040-681 Using AIX® 5L V5.2 and DB2 Universal Database 8.1*, 2004.

[10] Transaction Processing Council. *TPC Benchmark C*, *Standard Specification*, Rev. 5.2, Dec. 2003.

Trademarks