Contents

CMPE2400 – Databases – Notes VI

Inserting Data into a Table

Row(s) can be added to a table with the INSERT statement.

Inserting data into a table is actually a large topic, as data may be inserted very simply, by providing the data to insert, or with high complexity, by using SELECT statements to bulk insert data.

The simplest syntax specifies the table and ordered values matching the tables columns :

```
insert into db_name.dbo.my_table
values ('One', ‘Two’, 5)
go
```

This syntax is not generally desirable – there are many assumptions. The columns are not obvious in the statement and this can lead to problems. For example, if this table has an identity property for the primary key in the first column, then NO data is expected, and ‘One’ will commence being inserted as the second column. If a column is forgotten or the table restructured with a new column things can go astray quickly.

The preferred syntax is to explicitly indicate the column names and order in which the value clause will provide insertion data :

```
insert into db_name.dbo.my_table( column_one, column_two, column_three )
values ('column_one_data', 'column_two_data', 'column_three_data')
go
```

In this case, you should specify all columns that require data (ie. Non null) and makes the statement easy to read – and this generally is the form taken for data exported as scripts from a database.

```
insert into db_name.dbo.Customers( first_name, last_name, postal_code )
values ('James', 'Kirk', 'T2B4J5')
go
```

If you wish to enter multiple rows of data, just separate the row data with commas:

```
insert into db_name.dbo.Customers( first_name, last_name, postal_code )
values ('James', 'Kirk', 'T2B4J5'), ('Spock', null, 'Z9Z8Y8')
go
```

In the above case, the columns can been specified in any order, but it is very obvious to anyone reading the statement what data is going to which column.

Always investigate the table structure of the insertion table to determine if any identity fields exist, foreign keys and fields that are optional (allow null).

You may insert multiple rows into a table by using a SELECT statement to obtain the data to insert. In this case, the VALUES clause is entirely replaced with the SELECT statement. The subquery you use must match the column order and types for the insertion list. Consider the following table:

```
create table db_Lockers.dbo.Student
(
	StudentID int not null
		constraint PK_StudentID primary key,
	FirstName nvarchar (20) not null,
	LastName nvarchar (20) not null,
	LockerID nvarchar (10) null
		constraint FK_LockerID foreign key
			references db_Lockers.dbo.Locker (LockerID)
)
go
```

If you wanted to take, say AdventureWorksLT customers, and make them students, you could add them with the following statement:

```
insert into db_Lockers.dbo.Student
(FirstName, LastName, StudentID)
	select top 10 C.FirstName, C.LastName, C.CustomerID
	from AdventureWorksLT.SalesLt.Customer as C
	order by C.CustomerID
go

select * from db_Lockers.dbo.Student
go
```

```
StudentID   FirstName            LastName             LockerID
----------- -------------------- -------------------- ----------
1           Orlando              Gee                  NULL
2           Keith                Harris               NULL
3           Donna                Carreras             NULL
4           Janet                Gates                NULL
5           Lucy                 Harrington           NULL
6           Rosmarie             Carroll              NULL
7           Dominic              Gash                 NULL
10          Kathleen             Garza                NULL
11          Katherine            Harding              NULL
12          Johnny               Caprio               NULL

(10 row(s) affected)
```

In this case the names are taken from the source database, and the *CustomerID* is used as the *StudentID* in the target database. Depending on the complexity of the query, you could easily populate an entire database from one or more source databases.

The subquery, provided it supplies the explicitly specified columns in the required order with the required types, will generally convert incoming data to conform with the destination table types. For example if your subquery selects a varchar(24) field and a real from one table to insert into another table the string and numeric value will be saved in the insertion format (ie. could be nvarchar(20) and money). You will be warned if the conversion loses data – ie. string is truncated or real to integer.

@@IDENTITY

What about IDENTITY columns ? Unless a modification is made to the state of a table an insert statement will not allow a column with the IDENTITY property set to accept and insert a value. The IDENTITY property on a table column ensures that an insertion of a record to that table will result in a new incremented value generated and inserted. This is handy for primary keys or a situation where a unique identifier is required – in a multi-user environment you can't rely on simple manual retrieval of the current max, manually increment and use that value for insertion.

The use of the IDENTITY property means an unknown value has been inserted on your behalf (very likely a primary key), and subsequent knowledge of that value is critical for continued processing. For example, adding an author to the authors table in Publishers, then adding the title they wrote – the author's primary key is required.

The @@IDENTITY system function returns the last inserted identity value (in your session or stored procedure). Like @@ERROR, immediate retrieval (and potential saving) is important. Other actions (like triggers) may lead to @@IDENTITY being modified and your value is then lost. In the case where an insert used a sub-select to insert multiple records, @@IDENTITY will return the last identity value used.

Assuming the Customers table has an IDENTITY property set on the customer_id column, an insert may look like :

```
declare @last_identity as int

insert into Customers( first_name, last_name, postal_code )
values ('James', 'Kirk', 'T2B4J5')

set @last_identity = @@IDENTITY

-- Use @last_identity to verify insertion, or insert new records in related tables
select * from Customers where customer_id = @last_identity

go
```

The @last_identity variable can be used as required for additional processing, either as a foreign key value for an insertion in a related table, or just as a where restriction on a select to verify/display insertion information.

Deleting Table Data

Delete statements delete entire rows of data, no partial deletion allowed. Delete statements can delete everything, nothing or anything in between. When deleting rows, always ensure the proper restrictions are in place as the DB engine does not warn before deleting rows, and the change is permanent.

The syntax for deletion of rows follows this form: NOTE : this deletes all rows **

```
delete [from] table_name
go
```

RNote: the optional from in designating the deletion table, you will both variations in online references - but there is a second use of from when indicating a source restriction. To minimize confusion, the use of from will be reserved for restriction use.

Restrictions are typical, and allow for simple selective removal of entire rows :

```
delete myDB.dbo.Students
where Students.student_id = '123456'
go
```

Restrictions can be simple where criteria, include subqueries or combinations of both.

```
delete myDB.dbo.Students
where
       LastName like 'G%' and
       FirstName in (
              select FirstName
              from #HipsterNames ) -- previously select into used for temp table
go
```

An alternative to subqueries is when a join is desired to provide restriction for deletion. This takes a subtle but different form. This form uses a from clause to designate a join condition :

```
delete myDB.dbo.Students  -- designate the deletion table
from
       myDB.dbo.Students as s inner join myDB.dbo.class_to_student as cs
       on s.student_id = cs.student_id
where
       cs.class_id = 123
go
```

In this case, the deletion table is indicated, allowing for a “from” clause to perform the required join to gain access to the field in which to specify the restriction.

Note : The DB may restrict your deletions if a foreign key constraint exists that prohibits the deletion of a student that has a foreign key in another table (results ?).

You may join as many tables as necessary, and only the resultset where the student records are included are deleted from the deletion table. This method may be used when your restriction is on a table that differs from the deletion table.

Updating Table Data

You may update the data in a table with the UPDATE statement. Update statements work on an entire row, but like delete can affect all, none or some rows. A where clause is used to restrict the rows which will be updated. The body of the statement consists of the SET statement with column assignments separated by commas where necessary. The basic form is :

```
update myDB.dbo.Students
set LastName = '??'
go
```

Note : Like delete, this update would affect ALL rows, setting LastName to ??
A more typical form would include a restriction, thereby allowing 1 (or more) rows to be updated :

```
update myDB.dbo.Students
set LastName = 'Sulu'
where student_id = 123456
go
```

Or for multiple columns to be affected, comma separate the value assignments

```
update myDB.dbo.Students
set LastName = 'Sulu', FirstName = 'Hikaru'
where student_id = 123456
go
```

Another typical use is to update many rows to a new value, perhaps using the original value

```
update db_Lockers.dbo.Students
set LockerBank = LockerBank + '-XA'
where LockerBank = 'WA%'
go
```

In this case, all rows that are currently in LockerBank that start with WA, to WANNN-XA. Ie. The original LockerBank with a '-XA' appended on the end. Remember types matter in the set operation.

Be careful, update statements with no restrictions will result in ALL rows being updated.

Occasionally, an update is desired that requires data from a different table. This could be due to the set expression. Consider an update to a field that is calculated :

```
update db_Sales.dbo.Customer
set CurrentBalance = CurrentBalance - CustomerBonus
from
       db_Sales.dbo.Customer as c
              inner join db_Sales.dbo.CustomerDetails as cd
              on c.customer_id = cd.customer_id
where
       c.customer_type_id in (
              select customer_type_id
              from db_Sales.dbo.CustomerType as ct
              where ct.customer_type like 'Retail' )
go
```

In this case, a join is performed in the from clause to retrieve the CustomerBonus from the CustomerDetails table. In addition, a subquery is used to restrict the update to only Customer types of 'Retail'. Care must be taken in use of [from joins] – in this case the join MUST result in a single CustomerBonus value from the CustomerDetails table based on the customer_id value.

Put another way – One-to-One (good), Many-to-One(Many side is the update, good) but One-to-Many(One side is the update, BAD), Many-to-Many (don't even try it).

If you are not sure if a join will be a potential issue, try performing a select *, using your join and restriction. Then examine the resultset, if your update has a single row for each record you wish to update, then things should go well, if however you see multiple rows for your customer then you will have an issue. As a work around, try calculating values into variables or temporary tables first, then accessing them in your update. Or, perhaps, a subquery might be a better approach.

Update statements have the potential for much more complexity, but this adequate for most uses.