TAKING ON A DB2 UDB IDENTITY --Kent Milligan

This article was originally published in the November, 2002 issue of System iNEWS magazine (systeminetwork.com)

DB2 UDB identity columns are one of many V5R2 SQL enhancements that make it easier to port databases from other platforms to the iSeries. One characteristic common to many of the tables in a database is that they contain unique identifiers, which are typically used as an index to identify the data. For example, a customer number field uniquely identifies a customer and ties information in the database to that customer.

Some applications require that users assign these identifiers; others generate the identifiers themselves. Of course, developers must write and maintain this code. For such applications, identity columns make this task easy. Let's explore how identity columns work.

Why Identity Columns?
Identity columns provide developers an easy way to automatically generate a unique, numeric value for every row in a table. As new rows are added to a table, DB2 UDB for iSeries is responsible for supplying the next value for an identity column instead of running through application code to generate the value.

The sample employee table below shows identity columns in action.

```
CREATE TABLE employee (
  empid INTEGER GENERATED ALWAYS AS IDENTITY,
  name CHAR(30),   dept#  CHAR(4))
```

In this example, the employee id column (empid) has been created as an identity column, so as new employees are added, DB2 UDB will generate the empid value. The first employee will be assigned an empid value of 1, the next employee will be assigned 2 for it's empid value, and so on. Identity columns are great for columns such as employee id where you simply need logic to assign the next value and the value itself has no inherent meaning. Surrogate keys in data warehousing are another niche that identity columns can fill.

Now that you see what an identity column can do, the next logical question is, "Is it any better than the application code?". DB2 UDB identity columns avoid the concurrency and performance problems that can occur when an application generates its own unique counter outside the database. A common application logic design is to store a counter in an object that can be shared, such as a table, data area, or data queue. Each transaction then locks this counter object, increments the number, and then unlocks the counter object.

Unfortunately, this design also makes other jobs or transactions wait until the counter object has been incremented and unlocked before they can obtain a key value. Although DB2 UDB also must use some type of internal locking to maintain the identity column, the locking performed by DB2 UDB requires fewer system resources. Faster generation of key values by DB2 UDB identity columns means that applications can achieve much higher levels of throughput and scalability.

Identity columns also remove the burden from the application developer of having to remember to write code or call a routine that generates the unique identifier for a table. The integrity of the application and database are enhanced with the automatic generation of unique identifiers by the identity column.

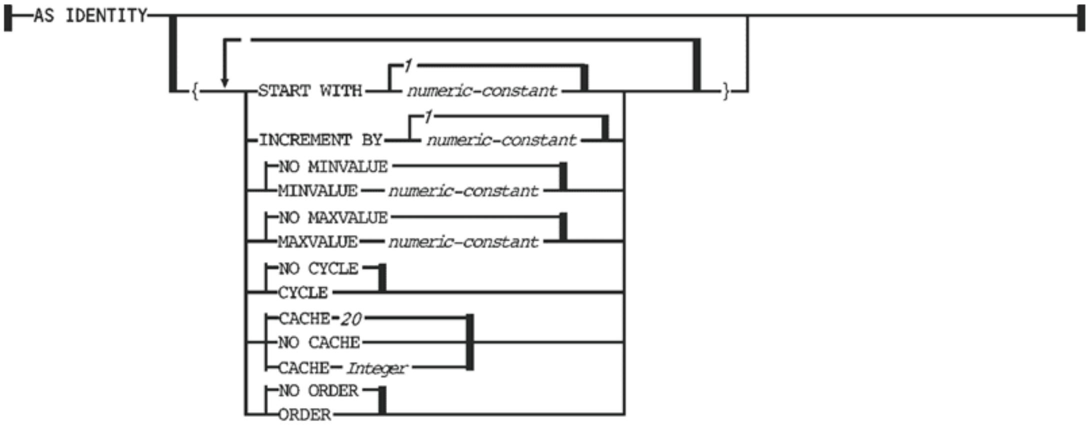
CREATING AN IDENTITY
To create an identity column in DB2 UDB, you include the IDENTITY clause for a numeric column when creating an SQL table. Along with the integer data types, the decimal and numeric data types can also be used for an identity column as the long as the precision for those column types is zero. A table can have a maximum of one identify column.

The IDENTITY clause is only available to SQL tables, which means there's no DDS support for identity columns. This is consistent with past OS/400 releases, in which an SQL interface is required for new features (such as BLOBs), as SQL is the strategic database interface for iSeries. Despite this SQL requirement, native (or non-SQL) applications can benefit from the key-value generation associated with identity columns. If a table has an identity column and a new row is inserted, DB2 UDB generates a key value for that new row regardless of the interface being used. It doesn't matter to DB2 UDB whether the new row was generated with an RPG Write, an SQL INSERT statement, or a CL command such as CPYF (Copy File); a new value is generated for the Identity column.

Although DB2 UDB is responsible for generating the next value for an identity column, it's *not* responsible for enforcing the uniqueness of that next value. The identity column must have a primary key constraint, unique constraint, or unique index created over it to guarantee the values in the identity column are unique. In a moment, I'll introduce several methods that let an identity column be bypassed or overridden, which could cause duplicate values without a unique index in place.

As Figure 1 shows, several identity options are available to define key-generation behavior that best matches your business requirements. By default, DB2 UDB will start an identity column with a value of 1 and increment that value by 1 each time a new row is added to the table. You can specify the START WITH and INCREMENT BY clauses to easily change this default behavior to something else, such as starting with a value of 100 and incrementing by 10. The INCREMENT BY clause also lets you specify a negative value to generate a descending key value.

Figure 1: Available identity options to define key-generation behavior



The MAXVALUE and MINVALUE clauses allow a limited range of values to be generated for an identity column. For example, a company that manufactures lawn furniture in two colors could specify a MAXVALUE of 2 for the colorid column in their colors table to prevent someone from adding a third color to the table:

```
CREATE TABLE colors (
  colorid INTEGER GENERATED ALWAYS AS IDENTITY (MAXVALUE 2),
  color_name CHAR(30))
```

The minimum and maximum values for an identity column default to the minimum and maximum value associated with the column's data type.

When DB2 UDB hits the maximum or minimum value for an identity column, the default behavior of NO CYCLE prevents a new value from being assigned to the identity column. For cases in which you want DB2 UDB to continue assigning new identity column values after the maximum or minimum value is reached, you can specify CYCLE to have the identity column start over at the beginning. For an ascending sequence, DB2 UDB would start assigning with the minimum value; for descending sequences, DB2 UDB would use the maximum value.

The CACHE identity option is the most difficult to understand. To improve the performance of multiple jobs and connections inserting new rows into a table, DB2 UDB reserves the next set (or cache) of identity column values in memory. Instead of having to write the next identity column value out to disk each time a row is inserted, the cache of reserved values allows DB2 UDB to maintain the next identity column value in memory.

The integer value supplied with the keyword specifies the number of identity values that will be reserved. IBM recommends that you simply use the default value(CACHE 20). This means the first time a row is inserted into a table with identity column, DB2 UDB will pre-allocate identity values 1-20 in memory and write to disk that the next identity column value is 21. Values 1 thru 20 would be assigned to the first 20 rows inserted even though the identity column value stored on disk says that 21 is the next identity column value.

If a system crash occurs before all of the pre-allocated values are assigned, those identity columns values that are not yet assigned are lost and will never be used. After the system is restarted, 21 will be the next identity value used. This is one way that gaps can appear in the sequence of values for an identity column. (In a moment, we'll explore other ways that this can happen). If distributed tables are allowed to have identity columns in the future, then the CACHE identity option will also have an affect on how identity values are allocated to each server that the distributed table is spread over with DB2 Multisystem.

The ORDER and NO ORDER options apply only to distributed tables created with the DB2 Multisystem feature. However, they have no effect in V5R2 since distributed tables are not allowed to have identity columns.

To review the identity options for an existing identity column, you can display properties of its table in iSeries Navigator or review the output of the CL command DSPFFD (Display File Field Description).

Identity options such as CACHE and MAXVALUE are actually part of the GENERATED clause, which controls when DB2 generates values for an identity column. GENERATED ALWAYS, which is the default and recommended value, causes DB2 UDB to always generate a value for the column when a row is inserted into the table. GENERATED BY DEFAULT is the other choice that can be specified. It directs DB2 UDB to generate an identity column value only when a value is not specified for the identity column on a row insertion. This is one of the reasons why a unique index or constraint is needed over the identity column to prevent duplicate values.

If a row is added via a non-SQL application interface (e.g., RPG Write), DB2 UDB will always generate the identity column value even if BY DEFAULT has been specified.

GENERATING AN IDENTITY

When you insert a row into an identity column defined with the default option of GENERATED ALWAYS, then a value for the identity column is not specified. The identity column value can be omitted in one of the two ways depicted below:

```
INSERT INTO employee(name, dept#) VALUES('Larry Bird','BC33')

or

INSERT INTO employee VALUES(DEFAULT,'Larry Bird', 'BC33')
```

If the Larry Bird row is the first row added to the employee table, then either INSERT statement would cause an empid value of 1 to be generated for Larry Bird.

If you try to specify a value for an identity column created with the GENERATED ALWAYs option, an error (SQLSTATE 428C9) is returned. The one exception to this rule is if the OVERRIDING SYSTEM VALUE clause has been specified on an INSERT (or UPDATE) statement. If the INSERT contains an OVERRIDING SYSTEM VALUE clause, the new values specified for an identity column (rather than the values generated by DB2) are used.

Here's an example of how you can use this OVERRIDING clause to specify a user-generated employee ID value for Larry Bird.

```
INSERT INTO employee VALUES(33,'Larry Bird', 'BC33')
   OVERRIDING SYSTEM VALUE
```

This example shows how the override option can also introduce gaps in the sequence of values for an identity column.

OVERRIDING USER VALUE is another override option that you can specify on INSERT (and UPDATE) statements. OVERRIDING USER VALUE causes DB2 UDB to always generate the value for an identity column and ignore any user-supplied values for an INSERT or UPDATE statement.

The override option isn't available to non-SQL interfaces. As mentioned earlier, DB2 UDB will always generate the identity column value when rows are added or changed from a non-SQL interface.

The counter for the identity column is incremented or decremented independently of a job or connection. If a given job inserts two rows that increments an identity counter two times, that job may incur a gap in the two numbers that are generated because other jobs may be concurrently incrementing the same identity counter.

An identity column may also appear to have gaps in the sequence of values as the result of an INSERT statement that was rolled back as part of the database transaction. Once an identity column value has been assigned, it won't be reused even if the associated INSERT statement has been rolled back. For example, if the INSERT of the first employee into the employee table is rolled back, the next employee inserted into the employee table will receive an employee id of 2, not 1. Identity column values that have been deleted are also not reused.

FINDING YOUR IDENTITY

One benefit of having application code generate key values for columns such as employee ID is that the generated value is easily accessible to the application to display or include in an email welcoming a new employee. So what does the application do in these cases if DB2 UDB is generating the identity column value? The application could issue a FETCH operation against the row just inserted and extract that identity value manually. But this is a complex and slow-performing solution. A better answer is the IDENTITY_VAL_LOCAL SQL function.

The IDENTITY_VAL_LOCAL function returns the most recently assigned value for an identity column within a job or database connection. If you're using threads within a job or connection, the function always returns the last value inserted into an identity column by *any* thread, not just the thread invoking the IDENTITY_VAL_LOCAL function. The identity value returned by this function can either be a DB2 UDB-generated value or a user-specified value (e.g., INSERT with OVERRIDING SYSTEM VALUE).

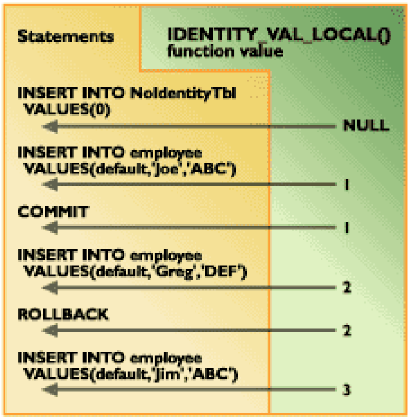
Here's an example of how you can use the VALUES statement to retrieve the most recently assigned identity value into a local host variable:

```
VALUES IDENTITY_VAL_LOCAL() INTO :hostvar.
```

If an SQL INSERT statement is attempted but ends in failure, the IDENTITY_VAL_LOCAL function will return unpredictable results. If no SQL Insert statement has been executed within a job or connection, then the IDENTITY_VAL_LOCAL function will return a null value.

The IDENTITY_VAL_LOCAL function isn't affected by UPDATE, COMMIT, ROLLBACK statements, nor is it affected by an INSERT into a table without identity columns. Figure 2 shows a sequence of statements that demonstrate how the IDENTITY_VAL_LOCAL function value is affected and not affected by SQL statements.

Figure 2
Statements' effects on the IDENTITY_VAL_Local function value



Statements	IDENTITY_VAL_LOCAL() function value
INSERT INTO NoIdentityTbl VALUES(0)	NULL
INSERT INTO employee VALUES(default,'Joe','ABC')	1
COMMIT	1
INSERT INTO employee VALUES(default,'Greg','DEF')	2
ROLLBACK	2
INSERT INTO employee VALUES(default,'Jim','ABC')	3

If you use triggers to perform business processing, you find the generated identity value within the trigger by accessing an SQL trigger transition variable or the external trigger buffer. You can use the IDENTITY_VAL_LOCAL function inside a trigger, but you should consult the *DB2 for i5/OS SQL Reference* before coding this logic. The IDENTITY_VAL_LOCAL function will always return null if it's invoked within an Insert Before trigger or the trigger condition (ie, WHEN clause) of a Before Insert trigger. If the function is invoked by an Insert After trigger, it will also return null unless the After trigger logic includes an INSERT statement that has already been executed.

If the IDENTITY_VAL_LOCAL function is invoked after an INSERT statement that caused triggers to be executed, the result is the value actually assigned to the identity column (that is, the value that would be returned on a subsequent SELECT statement). This value isn't necessarily the value provided in the INSERT statement or a value generated by the database manager. The assigned value for an identity column could be a value that was changed by a Before trigger. SQL Before triggers can change data for any column with a SET statement and external triggers accomplish this data change by modifying the trigger buffer.

No functions are available to non-SQL interfaces for retrieving the most recently assigned value. Thus, applications using non-SQL interfaces will have to read the last record inserted to access the generated identity value.

MAINTAINING YOUR IDENTITY

When you start using something new, there are definitely times when you'd like to be able to start over or tweak something. In V5R2, the ALTER TABLE statement has been enhanced to provide that capability for identity columns. To starting a sequence over at the beginning, you specify the RESTART clause on the ALTER TABLE statement:

```
ALTER TABLE employee ALTER COLUMN empid RESTART;
```

You can use the SET clause to change one of the identity column options such as MAXVALUE.

```
ALTER TABLE employee ALTER COLUMN empid SET MAXVALUE 1000;
```

If you need an identity column to go back to being a normal numeric column, you can use the DROP IDENTITY clause to change an identity column to a non-identity column.

```
ALTER TABLE employee ALTER COLUMN empid DROP IDENTITY;
```

The ALTER TABLE statement is the only way to restart the sequence for an identity column value. The CLRPFM (Clear Physical File Member) and RGZPFM (Reorganize Physical File Member) commands have no impact on the identity column values. CL commands such as INZPFM (Initialize Physical File Member) and CPYF (Copy File) can be used to add new rows to a table, are recognized by DB2 UDB as non-SQL interfaces, so a new identity column value is always generated.

Give It a Try!

Hopefully, you now have an idea of how to use identity columns to simplify application development. For more information regarding the different information for controlling identity key values, consult the *DB2 for i5/OS SQL Reference* guide.