

Conservative Use of DB2 Identity Columns

by Daniel L. Luksetich, Software Alternatives Incorporated

Origin of Identity Columns

DB2 identity columns were introduced with version 6 (refresh) of DB2 for OS/390 via APAR PQ30652. This new feature allows users to create a column in a table whose numeric value increments for each new row added to the table. The database engine is responsible for the maintenance of this "next" value, and can guarantee its uniqueness. This type of column is perfectly suited for any column designated as a system-generated, or "dummy" key. This was a much-anticipated addition to the functionality of the database engine, since DB2's major competitors have had the same feature for some time. It is also quite desirable to relieve an application process of the obligation to maintain these sorts of columns within the application, and puts the functionality automatically within the database engine, where it belongs.

How Identity Columns Work

An identity column can be a regular, or user defined, numeric data type column. The allowed regular data types are SMALLINT, INTEGER, and DECIMAL with a scale of zero. Any user defined data type, based upon one of these, can also be used as an identity column. DB2 limits the use of identity columns to one per table. A column can be created as an identity column when a table is initially defined, or when a column is added to the table via an ALTER statement. In the later situation, if the table in which an identity column has been added contains data, then the associated tablespace is placed in the reorg pending status and a reorg must be performed in order to initialize the identity values.

An example of a CREATE TABLE statement, including an identity column would be:

```
CREATE TABLE TEST.DAN_TABLE
  (ACCT_ID      INTEGER NOT NULL
   GENERATED BY DEFAULT AS IDENTITY
   (START WITH 1, INCREMENT BY 1 CACHE 20)
  ,DATA_COL    CHAR(20) NOT NULL WITH DEFAULT)
IN DB1.TS1;
```

Here, an integer column called ACCT_ID has been designated as an identity column for table TEST.DAN_TABLE. In addition, the column is GENERATED BY DEFAULT. What this means is that DB2 will supply a value for the ACCT_ID column of a new row in the table, if the value for that column is not already supplied. The other side of this clause is GENERATED ALWAYS, which means that DB2 will always supply the value for this column, and no other values are allowed. There are distinct advantages to either choice, which are discussed later in this article. A starting value,

and an increment value tell DB2 what the first value assigned to the identity column should be, and what value to add for each successive value assigned. DB2 can also be directed to cache identity values in memory for performance. By default, 20 values are cached, allowing DB2 to quickly retrieve the next identity value. Since this caching can result in gaps in the sequence of values assigned (e.g. if there is a crash), the feature can be turned off by specifying "NO CACHE" if the sequence of assigned values is important to the application. Hopefully it's not, because the caching can afford a wonderful performance advantage to more traditional ways of retrieving system-generated values using traditional techniques.

One common way for an application to assign system-generated keys in the past has been to maintain a table that contained a single row with a single column that contained the next key value to assign. The application, when it needed the next key value, would read from this table, update the column with the next key value, and then commit. In a high performance application, these "next key" tables could become a performance bottleneck, when multiple processes are attempting to assign new key values concurrently. Caching these next key values by utilizing identity columns seriously reduces this risk!

Identity columns are useful for generating unique keys for various DB2 tables. In situations where a system-generated unique key is required, identity columns can fit the bill nicely. Because each new value is assigned incrementally, identity columns can be used with very large tables, in which inserts are directed to the end of the table. Clustering by an identity column in these situations can guarantee that the inserts are at the end of the table.

Information about identity columns is stored in the DB2 system catalog. The SYSIBM.SYSSEQUENCES table contains one row for each identity column in the subsystem. This table contains all of the information that DB2 needs to manage the identity column value, including the increment value, number of values to cache, and the last possible assigned value, which is called MAXASSIGNEDVAL. It is important to note that this value may not always represent the highest value stored in a table for an identity column, since cached values that may never be used are also reflected here. The SYSIBM.SYSSEQUENCEDEP table relates identity columns to the tables in which they are used.

DB2 generates identity values during LOAD, REORG (for identity columns added via an ALTER), and INSERT processing. A DEFAULT clause is provided on an INSERT statement to easily specify when default values for a column should be used, and for an identity column this means the next identity value. There is a DB2 function called IDENTITY_VAL_LOCAL() which allows an application to retrieve the last inserted identity value. This is useful when an application is inserting data into a parent table, which is using an identity column as the primary key, and then inserting data into one or more child tables, using the retrieved identity value in the foreign keys. This function can be used in a SET command, right after INSERT, to retrieve the value:

```
INSERT INTO TEST.DAN_TABLE VALUES (DEFAULT, :DANS-DATA) ;  
SET :ACCT-ID = IDENTITY_VAL_LOCAL ( ) ;
```

In the example above, a new row is inserted into the table, and the ACCT_ID that was generated was placed in a host variable called ACCT-ID.

To default or not to default?

The GENERATED ALWAYS clause prevents any process, other than the database engine itself, from creating an identity value. This is extremely valuable in that uniqueness of the generated values can be assumed, and the need for an enforced unique rule in the database is not necessary. This can eliminate the need for additional indexes as well. For example, using the table above, lets assume that the ACCT_ID is the primary key for the table TEST.DAN_TABLE (but not database enforced), and has been defined as an identity column using the GENERATED ALWAYS clause. Traditionally, a unique index on ACCT_ID would be established in order to enforce the unique rule. However, since DB2 is solely responsible for generating new ACCT_ID values, and guarantees they are unique, then making the index unique is not required. If we find that a large percentage of queries against TEST.DAN_TABLE always reference the DATA_COL and ACCT_ID columns together, and that putting the DATA_COL column in an index would improve performance, we could then create a non-unique index on ACCT_ID and DATA_COL which would improve query performance, but not negatively impact our unique rule. The effect is the same as a feature in non-mainframe DB2 UDB called "INCLUDE", where columns can be added to an index, but don't contribute to a uniqueness rule.

There are situations, however, where providing values for the identity columns cannot be avoided. One example cited in the DB2 manuals is tables that are the target of data replication. In this situation, the target table needs to use the identity values of the source table. Also, in some replication situations, data in a target table will need to be refreshed. In those situations, data is typically loaded from the source to the target, using the DB2 LOAD utility. In addition to the replication scenario, if you are not growing your data (that is, not starting you tables from scratch), then taking advantage of GENERATED ALWAYS is more difficult. Data for a lot of new DB2 applications come from legacy applications, which are being reengineered, or incorporated into new multi-functional systems. In these cases, there is typically a data conversion process, and in most situations the data is far too voluminous to convert using a DB2-based programmatic insert process. IBM advocates using identity columns as primary keys, which implies that there would be foreign keys referencing these primary keys. The application conversion process has to use the DB2 load facility to populate the tables, but then also has to accommodate the propagation of primary keys to various foreign keys. The application conversion process would then generate its own key values, which would be used in all primary and foreign keys, before loading the data into the various DB2 tables. In these situations, the choice is to use GENERATED BY DEFAULT, and once this option is chosen you can't go back!

When using GENERATED BY DEFAULT a unique rule must be enforced by the database engine via a unique index on the identity column. This prevents both the application, which can provide identity values, and the database engine, which provides the values when the application does not, from stepping on each other's toes. In addition, since there is no legitimate way to "advance" the MAXASSIGNEDVAL value for an identity column in the DB2 catalog (I am contemplating an "illegitimate" way), values assigned by the application, or by the database engine need to somehow be coordinated. This works fine for a data conversion situation, where the initial values in the database for the identity columns can be set to a value greater than the highest value that the conversion application will generate. However, this can become quite difficult in an application where some inserted rows are assigned identity values by the database and others by the application.

Availability Issues

The use of DB2 as a data-store for massive amounts of data is ever increasing. Likewise, the availability needs of these systems is typically 100%! So database administrators and application developers, are facing the challenge of having to manage enormously-sized DB2 tables, while maintaining full availability to these tables. Therefore, each feature of the database engine that is utilized has to be carefully scrutinized for any potential impact to availability. Assuming it takes longer to run DB2 utilities against a large DB2 table than a smaller one, a desirable situation for these large tables is one in which utilities are run only against a portion of those large tables, or not at all.

In topic 4.6.5 of the DB2 V6 Administration Guide it is stated that if a tablespace that contains a table with an identity column is recovered to a prior point-in-time, that the RECOVER utility sets the tablespace status to REORP. If a partition of a partitioned tablespace is recovered to a prior point-in-time, the entire tablespace is placed in REORP. The REORG utility is run in order to reset the REORP status. This description is somewhat vague, but implies that you can create an outage situation of an entire partitioned tablespace that contains an identity column, by recovering one partition of that tablespace to a prior point-in-time. While this sounds very serious, no other DB2 manual mentions it, and I have not been able to create this situation in a test environment. Nonetheless, it is worth taking into consideration in that there must have been a reason for including it in the DB2 Administration Guide. In a production environment in which there is a very large tablespace (perhaps billions of rows of data) with very high availability requirements, the additional risk of outage due to the use of identity columns might not be worth putting the identity columns into tables that contain data.

PUBLISHER'S NOTE: We have recently learned the following from IBM:

Documentation error:

- If the table was created with an identity column, then there is no REORP consideration for PIT recovery
- If the identity column was alter-added, then if the TS is recovered to a PIT prior to when REORG cleared the REORP status (i.e. prior to when REORG materialized the identity column values) then RECOVER enters the TS back into REORP status.
- Bottom line: it should be very rare that a TS is entered into REORP due to PIT recovery involving an identity column, so you probably do not want to add the complexity/overhead of defining a separate table to generate the ident. col. value for the reason of avoiding REORP for PIT recovery.

The DB2 Administration Guide will be corrected, but this one slipped through without making it into the V7 books as well. So, we'll be making the changes in the online V6 and V7 books.

Also, in a situation in which data may be added to a table with an identity column via a LOAD utility, and the identity column has been defined with GENERATED BY DEFAULT, the next identity value stored in the DB2 system catalog as MAXASSIGNEDVAL may not reflect the next value that needs to be generated. In this situation, the table could be dropped and recreated to advance the identity value by assigning the a value higher than any previously assigned value as initial value. This could also introduce an unwanted outage situation.

PUBLISHER'S: A late note from the paper's author - another potential problem:

Yet another reason to avoid identity in your table that contains data. This one from the utility guide:

"LOAD INTO PART x is not allowed if an identity column is part of the partitioning index"

Maintaining Availability and High Performance

A simple way to avoid these high availability issues is to not use identity columns in any table that contains data, and to create special tables to hold identity values. Here we are combining the new feature of DB2 into the old method of maintaining a "next key" table. In this situation, your primary entities do not have a direct dependency on the identity values generated by the database, and the application can retrieve identity values from the identity table, and then use them on insert into the appropriate table. This affords the following advantages:

- No direct dependency on DB2 catalog for identity values in primary entity tables
- No chance for REORP status on table with identity column
- Ability to easily "reset" max and increment identity values by dropping and recreating the table that contains just the identity column

Your high volume, high availability table will have no strings attached in the way of identity column maintenance, while you can still take advantage of the identity value generation, with additional control over that generation.

Applications would retrieve identity values from the separate "identity" table by inserting a row into that table, using the IDENTITY_VAL_LOCAL() function to retrieve the value, and then deleting the row. The identity table would never officially contain any data, but would simply be used to get the identity values. For a high performance application, this certainly sounds more expensive than just inserting data into the desired table. However, when compared to the traditional method of maintaining a "max key" table via SELECTs and UPDATEs, it is significantly more efficient. In addition, any locking problems that existed when using the traditional "max key" table do not exist in the "identity" table, because UPDATEs aren't being used. INSERTs and DELETEs guarantee that each process will not interfere with any other. Row level locking on the "identity" table could be employed to further guarantee high concurrency, and to reduce the size of the tablespace during high utilization. If even higher performance is desired, the application could avoid doing DELETEs on the "identity" table during high processing times, and defer the DELETEs to some overnight maintenance process.

Example

So the table definition above would change, removing the identity column, and adding another table that would hold the identity column:

```
CREATE TABLE TEST.DAN_TABLE
  (ACCT_ID    INTEGER    NOT NULL
  ,DATA_COL  CHAR(20)   NOT NULL WITH DEFAULT)
IN DB1.TS1;

CREATE TABLE TEST.DAN_IDENTITY
  (ACCT_ID    INTEGER NOT NULL
   GENERATED ALWAYS AS IDENTITY
   (START WITH 1, INCREMENT BY 1 CACHE 20))
IN DB1.TS2;
```

Now I can load TEST.DAN_TABLE with data containing ACCT_ID's that I created myself in a conversion program, or by some other means. I can use the TEST.DAN_IDENTITY table to generate new ACCT_ID's by inserting a row into the table, retrieving the identity value, then deleting the row. Here is an example of a SQL/PSM procedure that does just that (and tries up to 10 times to get a value):

```

CREATE PROCEDURE TEST.DAN_PROC (OUT P_ACCT_ID int)
  LANGUAGE SQL
  MODIFIES SQL DATA
  COLLID TEST
  ASUTIME NO LIMIT
  WLM ENVIRONMENT DB2TEST
  RUN OPTIONS 'NOTEST(ALL,*, ,VADTCPIP&10.1.9.34:*)'

P1: BEGIN
DECLARE I INTEGER DEFAULT 0;
DECLARE SQLCODE INT DEFAULT 0;
DECLARE NEW_ACCT_ID INT DEFAULT 0;
set I = 1;

insert1: LOOP
  INSERT INTO DAN_IDENTITY VALUES (DEFAULT);
  IF SQLCODE = 0
    THEN set NEW_ACCT_ID = IDENTITY_VAL_LOCAL();
    LEAVE insert1;
  ELSE set I = I +1;
  END IF;
  IF I > 10
    THEN LEAVE insert1;
  END IF;
END LOOP insert1;
IF NEW_ACCT_ID <> 0
  THEN DELETE FROM DAN_IDENTITY WHERE ACCT_ID = NEW_ACCT_ID;
END IF;
set P_ACCT_ID = NEW_ACCT_ID;
END P1

```

When the application needs to insert a new row into TEST.DAN_TABLE, it calls this stored procedure to return the next identity value, and then performs the insert:

```

CALL TEST.DAN_PROC (:ACCT-ID);
INSERT INTO TEST.DANTABLE VALUES (:ACCT-ID, :DANS-DATA);

```

Cheers!