Clarifying Some Common Misconceptions

BY CRAIG MULLINS

Part of my job is talking to IT professionals who use DB2, including developers, DBAs and managers. One of the things I've noticed recently, more than in the past, is that there are some aspects of DB2 that are commonly misunderstood—not by everyone, of course—but by enough of us that it warrants some investigation and clarification. There is no shame in this; DB2 has grown by leaps and bounds in the past decade making it larger and more complex than ever before. This article will look at several of the most common DB2 misunderstandings and try to shed some light on them.

The PIECESIZE Clause

The creation of non-partitioning indexes (NPIs) on tables in a partitioned table space is one of the most vexing problems facing DBAs. Partitioned table spaces tend to be large and by their very design will span multiple underlying data sets. The partitioning index that defines the partitioning key and key ranges also spans multiple data sets. But there can be only one partitioning index per partitioned table space. What happens when you need to define more than one index on a table in a partitioned table space?

Well, in the old days (pre-V5), the DBA could not control the creation of the underlying data set(s) used for NPIs. As of V5, the PIECESIZE clause of the CREATE INDEX statement can be used during index creation to break an NPI into several data sets (or "pieces"). More accurately, the PIECESIZE clause specifies the largest data set size for a non-partitioned index. PIECESIZE can be specified in kilobytes, megabytes, or gigabytes. For example, the following statement will limit the size of individual data sets for the XACT2 index to 256 megabytes:

```
 CREATE TYPE 2 UNIQUE INDEX
DSN8710.XACT2
  ON DSN8710.ACT (ACTKWD ASC)
  USING STOGROUP DSN8G710
       PRIQTY 65536K
       SECQTY 8192K
       ERASE NO
  BUFFERPOOL BP0
  CLOSE NO
  PIECESIZE 256M;
```

Basically, PIECESIZE is used to enable NPIs to be created on very large partitioned table spaces. It breaks apart the NPI into separate pieces that can be somewhat managed individually. Without PIECESIZE, NPIs would be quite difficult to manage and administer. Keep in mind, though, that PIECESIZE does not magically partition an NPI based on the partitioning scheme of the tablespace. This is a common misperception of the PIECESIZE clause. So, if you have a partitioned table space with four partitions and then create an NPI with four pieces, the data in the NPI pieces will not match up with the data in the four partitions.

When using PIECESIZE, more data sets will be created and, therefore, you can obtain greater control over data set placement. Placing the pieces on separate disk devices can help reduce I/O contention for SQL operations that access NPIs during read or update processing. The elapsed time improvement may be even greater when multiple tasks are accessing the NPI.

Separating the NPI into pieces allows for better performance of INSERT, UPDATE and DELETE processes by eliminating bottlenecks that can be caused by using only one data set for the index. The use of pieces also improves concurrency and performance of heavy INSERT, UPDATE and DELETE processing against any size partitioned table space with NPIs.

Keep in mind that PIECESIZE is only a specification of the maximum amount of data that a piece (that is, a data set) can hold and not the actual allocation of storage, so PIECESIZE has no effect on primary and secondary space allocation. Each data set will max out at the PIECESIZE value, so specifying PRIQTY greater than PIECESIZE will waste space. Avoid setting the PIECESIZE too small. A new data set will be allocated each time the PIECESIZE threshold is reached. DB2 will increment the A001 component of the data set name each time. This makes the physical limit 999 data sets (A001 through A999). If PIECESIZE is set too small, the data set name can limit the overall size of the table space. Ideally, the value of your primary quantity and secondary quantities should be evenly divisible into PIECESIZE to avoid wasting space.

To choose a PIECESIZE value, divide the overall size of the entire NPI by the number of data sets that you wish to have. For example, for an NPI that is 8 megabytes, you can arrive at 4 data sets for the NPI by specifying PIECESIZE 2M. Of course, if your NPI grows over 8 megabytes in total you will get additional data sets. Keep in mind that 32 pieces is the limit if the underlying tablespace is not defined with DSSIZE 4G or greater. The

limit is 254 pieces if the tablespace is defined as DSSIZE 4G or greater.

IDENTITY Columns

There also is a lot of confusion surrounding DB2's implementation of IDENTITY columns. Some of the common questions I hear are: "What do they do, anyway?"; "How do they work?" and "Why should I bother to learn about them at all with all those limitations?" So let's try to answer these questions.

A common requirement of relational applications and databases is the need to store a counter that identifies rows in tables. Until V7, DB2 provided no inherent, or built-in, support for such functionality. Now DB2 V7 adds support for IDENTITY columns. An IDENTITY column can be defined to a DB2 table such that DB2 will automatically generate a unique, sequential value for that column when a row is added to the table. DB2's implementation of IDENTITY columns avoids some of the concurrency and performance problems that can occur when application programs are used to populate sequential values for a "counter" column.

When inserting data into a table that uses an IDENTITY column, the developer or user does not provide a value to be inserted for the IDENTITY column. Instead, DB2 will calculate the appropriate value to be inserted.

Only one IDENTITY column can be defined per DB2 table. Additionally, the data type of the column must be SMALLINT, INTEGER, BIGINT or DECIMAL with a zero scale; that is, DECIMAL(x,0). The data type also can be a user-defined DISTINCT type based on one of these numeric data types. The designer has control over the starting point for the generated sequential values, and the number by which the count is incremented.

An example creating a table with an IDENTITY column is shown below:

```
CREATE TABLE EXAMPLE
(ID_COL INTEGER NOT NULL
GENERATED ALWAYS AS IDENTITY
START WITH 100
INCREMENT BY 10
...);
```

In this example, the IDENTITY column is named ID_COL. The first value stored in the column will be 100 and subsequent INSERTs will add 10 to the last value. So the identity column values generated will be 100, 110, 120, 130, and so on.

The GENERATED clause provides two options to control how DB2 will generate identity column values. GENERATED ALWAYS indicates that DB2 will always generate a value for the column when a row is inserted into the table. GENERATED BY DEFAULT indicates that DB2 will generate a value for the column when a row is inserted into the table unless a value is specified. The IBM manuals recommend using ALWAYS unless you are using data propagation.

This all sounds wonderful, but there are problems with identity columns. Some of these problems include:

There are problems loading data into a table with an identity column defined as GENERATED BY DEFAULT. (The next identity value stored by DB2 to be assigned may not be the correct value that should be generated.)

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

What about environments that require regular loading and reloading (REPLACE) for testing? The identity column will not necessarily hold the same values for the same rows from test to test.

When you decide to use GENERATED BY DEFAULT, you cannot change back to GENERATED ALWAYS.

The IDENTITY_VAL_LOCAL function returns the value used for the last insert to the identity column. But it only works after a singleton INSERT. This means you cannot use INSERT INTO SELECT FROM, or LOAD, if you need to rely on this function.

If the CYCLE clause is specified, when the maximum value is reached for the identity column, DB2 will cycle back to the beginning to begin reassigning values - which can be problematic. The alternative is NO CYCLE, which means that DB2 just stops generating values – which also might be a problem.

DB2 will not verify that an IDENTITY column value is unique unless a unique, single-column index is defined on the identity column.

If the CACHE option is specified DB2 will pre-allocate values in memory. But in the event of a system failure all cached values that were not assigned will be lost, and thus never used.

So, as with most things, the answer to "should I use them" is, of course, "it depends!" Identity columns can be useful, depending on your specific needs, even in their current implementation. If you can live with these caveats, then identity columns might be useful to your applications. However, in general, these "problems" make identity columns very much a niche solution. And IBM has hinted (at the May 2002 IDUG conference) that some of the problems with IDENTITY columns will be addressed over time in upcoming versions of DB2.

Synopsis

These are just a few of the many misunderstandings that exist in DB2-land. In future issues we'll look at some of the others. But until then, let's be careful "out there!"