Out in Left Field
strategic tools for strategic teams
Volume Two, Issue Two

DBMON – SQE vs CQE

DBMON, of course, stands for Database Monitor, the best System i tool for monitoring and tuning SQL performance. Centerfield Technology's insure/INDEX and insure/ANALYSIS are built on top of DBMON. At first glance, the raw database monitor output file seems "not for mere mortals". (A spell check of dbmon comes up with "demons".)

However, you will need to resort to this data when you encounter difficult SQL performance issues, and once filtered and interpreted via the insure/INDEX & ANALYSIS tools, there is no better data to help diagnose system wide or job related issues. DBMON has more data points than I could cover in a year of writing, so I'm going to focus this article specifically on using DBMON to help you enforce usage of the new SQL Query Engine (SQE) versus the old Classic Query Engine (CQE).

Recently I talked to a Centerfield Technology customer regarding some SQL jobs causing high resource consumption on his iSeries. With the help of our tools he was able to zero in on a single SQL statement and called me up to help him analyze it. We were using our Visual Explain tool within insure/INDEX&ANALYSIS and I noticed a message to the effect that a Select/Omit Logical File (S/O LF) was considered and rejected for query implementation. This raised a red flag.

This customer is running V5R3 and not leveraging the new SQE because his database contains S/O LFs over his physical files. Being fully aware of just how much work has gone into the new SQE engine and how much better (overall) it is from a performance standpoint, I knew he was missing some easily obtainable benefits. Since our Visual Explain allows for experimentation with different QAQQINI settings, I added another one: IGNORE_DERIVED_INDEX set to *YES. This only affected the job we were in and immediately the Visual Explain diagram changed dramatically, as did the associated statement run time. The statement was now using existing indexes where it previously would not. As it turns out, the mere existence of S/O LFs causes the SQE to redirect queries for CQE implementation. This 'throwback' action creates a 10-15% overhead in the query optimization process. In short, it's bad news.

Based on the benefit to that one SQL statement, our customer insisted we implement the fix system wide, so we duplicated the QAQQINI file from QSYS to QUSRSYS using CRTDUPOBJ and inserted a new row, namely IGNORE_DERIVED_INDEX set to *YES. He monitored the performance for a couple of days and hasn't called me since. ☺

Now you might be asking yourself: "Are the queries on my system actually going down the new SQE path? Am I taking advantage of all the good work those smart people in Rochester are doing?"

The best way to determine that on V5R3 is to run a system wide DBMON collection (on V5R4 you could simply dump SQL Plan Cache instead). Be aware that on an SQL intensive system, DBMON will collect a lot of data very quickly, so monitor the size of the output file carefully. Once you're satisfied that most of your key queries have run, end DBMON. Now let's look at that data with singular purpose in mind: SQE vs CQE.

I pulled this query from the IBM site:

```
SELECT   qqc16, COUNT(*)
FROM     qaqqdbmn
WHERE    qqrid=3014
GROUP BY qqc16
```

This seemingly simple query is actually quite powerful. It gives us the distribution

(Continued on page 6)

INSIDE THIS ISSUE:

Out with the old…

In with the new.

CTO MEMO

As I rejoin Centerfield Technology after spending several years working with the DB2 development team at IBM, I am very excited by the opportunity to help our customers in my new role as Chief Technology Officer. As CTO, my job is to help our development team create tools for System i customers to get the most out of their databases. I am particularly motivated to help those customers improve the performance of their applications by optimizing their database and SQL statements.

The DB2 development team in Rochester has dramatically improved the functionality and performance of SQL since it was introduced in the early 1990's. Over that same time, the applications that use DB2 have become more intricate and varied. Application architectures have grown in complexity. The database, however, continues to be a key (some might argue THE key) component of the application stack. In terms of overall system resources, many systems now devote a large percentage of those resources to serving database requests. In essence, some iSeries systems have become ***database machines*** – where DB2 IS the application. These facts have made the job of properly tuning SQL and the database more challenging yet more important than ever.

Our vision at Centerfield Technology is to help iSeries customers leverage their considerable investments. To achieve that vision, our strategy is to help System i customers easily tune their databases without being forced to learn the minutiae of the database engine or confronted by the hundreds of choices necessary to fully optimize their applications.

Our job is to deliver expertise and automation with our software – to help each and every one of our customers get the highest value from their applications, systems, and people.

Mark L. Holm

oH, DB2 UDB for iSeries, wHy arE yOU So caSe-senSITIVe?

I use SQL all the time for great number of tasks. Sometimes it's just to look up field types in interactive SQL (STRSQL command). I simply prompt on the SELECT * FROM mylib/myfile and hit F4 on the '*' or WHERE clause line. If I want to see field description text I just hit F11 on this display. Other times I use it for a simple lookup in our data or control files. It's easy to pick the fields, run column scalar functions on it, substring only part of the field, use the LIKE keyword to do wildcard type searches etc. I know I can use DSPFD or DSPFFD but hey, I like SQL and always have one session open.

Every once in a while, if I search on a large text (CHAR) field, I'd like to perform a case insensitive search. But no, I can't do it since our shop is not set up that way. I usually resort to just using the UPPER function to force everything to uppercase. This works, but depending on the size of the file, it may take a while as it can't take advantage of my existing indexes. If I'm searching a very long text field using a LIKE '%something%' type of search, I can't use UPPER since the word I'm searching may be in the middle of the field.

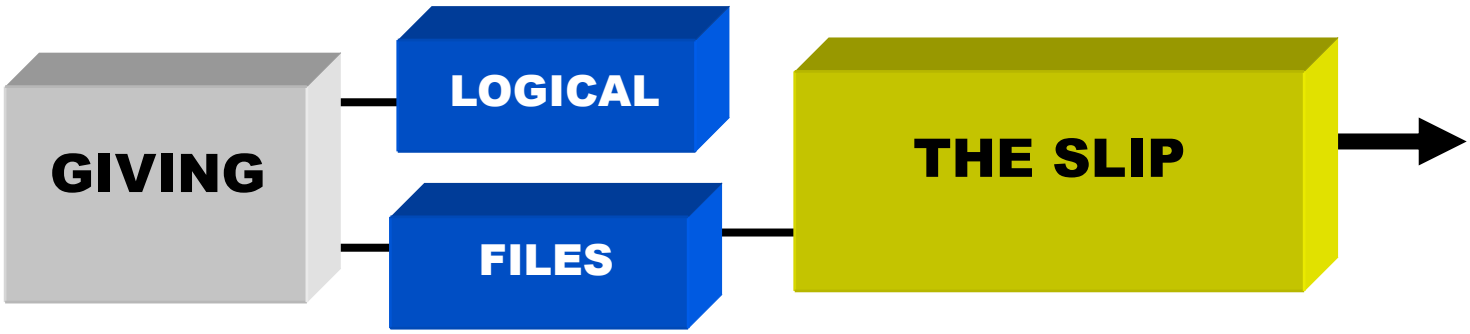
I know we've all wanted case insensitive searches at one point in time or the other, so let's find a way to make our favorite computer do what computers do well: searches, sorts and comparisons regardless of the case.

The iSeries answer for this technical challenge is the *sort sequence table*. Let's take a closer look at this solution.

Most SBCS (single byte character systems) jobs run with *HEX sort sequence. We will be changing this setting so let me first explain what *HEX is and why we want to change it. Hexadecimal sort sequence means that every letter in the alphabet has its own collating sequence or weight. For example, x'C1' is the weight for the letter 'A' and x'81' is the weight for the letter 'a'. For comparison purposes x'C1' is not considered equal to x'81'. For sort purposes x'81' would sort before x'C1' with other weights in between.

So, why is the default for most jobs *HEX? If you look at the QSRTSEQ system value, you'll notice that it's probably set to *HEX (the default shipped value). Furthermore, if you look at most user profiles on your system, the default sort sequence is set to *SYSVAL (whatever's in system value). Most SQL interfaces (i.e. STRSQL or CRTSQLxxx commands) default to *JOB. If you follow this dependencies thread, you will realize

(Continued on page 5)



Centerfield often gets questions from iSeries professionals about the differences between files created with DDS and objects created with SQL. We also get questions about how to transition from the RPG/DDS world into one that uses SQL to access data.

Most shops cannot convert from physical files to SQL tables and from logical files to SQL indexes over night. RPG programs expect logical files to exist and it does not make sense to rewrite software that is working well. Furthermore, traditional RPG programmers often do not see the advantages of using SQL objects.

The purpose of this article is to provide some rational for the use of SQL indexes in place of logical files. The trick is to introduce indexes without forcing the RPG programs to be rewritten.

Why would you even consider the use of an SQL index? Don't logical files work just as well? Indeed, it often makes a lot of sense to replace a logical file with an index in order to speed up queries and even RPG applications using native database access because SQL indexes have different performance characteristics than logical files.

When a keyed logical file is created a database index is created. Similarly, when an SQL index is created a database index is built. Even though a logical file and an SQL index are the "same", the data structures created by each command are different under the covers.

The first difference is that old logical files, and optionally new ones, can be created with a maximum size of only four-gigabytes (*4GB). SQL indexes, however, are created with a one-terabyte maximum size. Besides the obvious difference that the SQL object can grow much larger than the logical file, the internal structure of the SQL index is such that more jobs can actively update the index structure at the same time. This means that on systems that have many jobs updating the same file concurrently, the SQL index will perform better because DB2 uses a superior method to seize the index, which makes it less likely that different jobs will block each other out (i.e. reduce seize contention).

A second difference is the size of an SQL index's "logical page." This logical page is the storage used to hold keys. With a larger logical page size, an index can hold more keys on a single page. When DB2 navigates through an index, logical pages are brought into memory from disk as they are needed. Since more keys are stored on a single "page" within the index, sequential processing (e.g. keys that are read from A to Z) will result in fewer disk I/O's. SQL queries and RPG batch programs are two obvious situations where performance could be improved with the use of larger logical pages. To see the logical page size on your existing files, you can use iSeries Navigator. Simply list the files, right click, and then choose 'Description'.

So how do you get the benefit of better concurrency and fewer disk I/Os while still keeping your RPG programs running without change? The trick is to "slip" an SQL index under the logical file used by the RPG program. By using this technique, the RPG program does not have to change since it is still using the logical file as it was before, but the logical file itself is using a better structure to do its work. To get a logical file to use an SQL index do the following:

- Determine the key fields used by the logical file.
- Create an SQL index with those keys
- Delete the logical file and recreate it. Be sure and specify ACCPTHSIZ(*MAX1TB) on the CRTLF command so it matches the SQL index. When this is done, DB2 should notice the existence of the SQL index, match the keys required in the logical file and share the existing index.

The nice part of this procedure is that there is very little time in which the logical file is not available so downtime is minimized.

Once this procedure is done you've both taken a step toward the use of SQL and potentially helped out your existing RPG applications.


Da Vinci
Codes
for
DB2
7 HABITS
OF HIGHLY
EFFECTIVE
SQL TUNI
Call Centerfield
For performance under the covers
www.centerfieldtechnology.com

(Oh, DB2 UDB for iSeries, why are you so case-sensitive?
Continued from page 2)

why the default is usually *HEX as well as all the potential spots where you can change this setting.

Getting back to our objective, we want to perform case insensitive searches so we don't want to use *HEX. What we want instead is to use a *LANGIDSHR setting (shared weight sort sequence). This sort sequence assigns the same weight to letters 'A' and 'a' making our objective possible.

Let me first illustrate how to do that in my favorite environment, interactive SQL:

STRSQL
F13 – Services
1 – Change session attributes
page down and set 'Sort Sequence' to *LANGIDSHR
press Enter twice

Now when I run any of the four statements below, I'll get a hit on my 'regular' search term:

```
SELECT * FROM myLib/myFile WHERE myField = 'Regular'
SELECT * FROM myLib/myFile WHERE myField = 'regular'
SELECT * FROM myLib/myFile WHERE myField = 'REGULAR'
SELECT * FROM myLib/myFile WHERE myField = 'ReGuLaR'
```

That was simple enough. However, this interactive SQL solution is not going to suit all of your needs. You are a lot more likely to get this sort of request from your end users or CTOs, and they're definitely not using interactive SQL.
To that end, let's explore where else you can change this setting.

First, let's look at your programs with embedded SQL and SQL Stored Procedures (same rules apply). The default setting for CRTSQL*** compile commands is *JOB which defaults to the setting of the job where the compile command is running. I advise against using this setting since you always have to make sure the compile job is running with whatever setting you desire (i.e. *LANGIDSHR). An alternative is to use *LANGIDSHR on the compile command or RPG H spec, i.e. H SrtSeq (*LangIDShr) AltSeq(*Ext). This option is desirable if you always want case insensitive searches, sorts and comparisons, but for this scenario I recommend a 3rd option, using the *JOBRUN setting. This gives a system administrator and/or a programmer that calls your program or stored procedure the ability to switch the setting dynamically, simply by using the CHGJOB SRTSEQ (….) command.

Since our customer base is running modernized applications that use Web and SQL across an Ethernet wire (i.e. ODBC/JDBC), let me address the options available to them.

In essence, the same rules apply as the ones for programs with embedded SQL or SQL Stored Procedures. You can use CHGJOB on the QZDASOINIT jobs as well, as long as the programs and stored procedures were compiled with the *JOBRUN setting. Alternatively on your PC, you can change the ODBC data source Language setting to use "Sort based on language ID".
Another useful SQL tool IBM offers is "Run Sql Scripts" that comes with iSeries Navigator. To change the setting for this interface, click on Connection->JDBC Setup->Language->pick 'Language ID'->Shared. Java applications running JDBC can use connection string parameters to set this setting as well.

Let's get real

So far I've been focused solely on a functional way to affect case sensitivity, sorts and comparisons. However, our shop deals in performance and we all know that sooner or later we all have to deal with it whether we want to or not.

The crux of all performance issues relating to shared weight sort sequence is that English language SBCS shops (i.e. USA) usually don't build SQL indexes with matching sort sequence. The DB2 UDB for iSeries query optimizer cannot use incompatible indexes for query implementation and I can only guess it would be hard pressed to even use it for statistics (i.e. query optimization).

You've already guessed it, I'm going to suggest you create some compatible indexes, and for our need they are indexes with a shared weight sort sequence attribute.

To create SQL indexes with shared weight sort sequence, just make sure your job is running with *LANGIDSHR when you execute your CREATE INDEX statement. If you're still using keyed LFs, CRTLF has a SRTSEQ keyword you can use.
Use DSPFD to view the current setting of your indexes (keyed LFs).

Using UPPER SQL scalar function

Even if you ignored the *LANGIDSHR solution and went with SQL UPPER scalar function instead, you'd run into a similar performance obstacle on large files. To enable these statements to take advantage of SQL indexes on the iSeries, the index must be created with a sort sequence table that maps lower-case values to upper-case values — for English, the table is Q037 and is found in the QUSRSYS library.

"Another index!?", you may say. Yes, why not? Storage is getting cheaper by the day and the enhanced SQE query optimizer can now consider all available indexes for implementation without worrying about timeouts. Furthermore, studies have shown that in most shops 'reads' (i.e. SELECT) far surpass 'updates' (i.e. UPDATE,DELETE,INSERT) by at least 25 to 1 margin.

(DBMON — SQE vs — CQE
Continued from page 1)

of queries going to SQE represented by value 'Y' in QQC16 column. This will help you decide if any further analysis is even needed (i.e. no queries took the CQE path). If on the other hand a large proportion of queries took the CQE path on V5R3 or V5R4, further analysis may be beneficial. To that end, I have written a query that I believe will prove useful to shops running a significant amount of SQL:

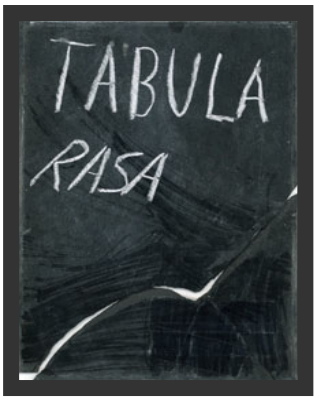
```
WITH sqe AS (
  SELECT qqjfld, qqucnt, qqc16, qvc43
  FROM   dbmon2
  WHERE  qqrid = 3014)
SELECT DISTINCT
      CASE WHEN b.qqc16 = 'Y' THEN 'Sqe'
           WHEN b.qqc16 = 'N' THEN 'Cqe'
           ELSE 'Unknown' END AS "SQL dispatch path",
      CASE WHEN b.qqc16 = 'Y' THEN 'N/A'
           WHEN b.qvc43 = 'XL' THEN 'Translation used in query'
           WHEN b.qvc43 = 'XU' THEN 'Translation for UTF used in query'
           WHEN b.qvc43 = 'UF' THEN 'User Defined Table Function used in query'
           WHEN b.qvc43 = 'LF' THEN 'DDS logical file specified in query definition'
           WHEN b.qvc43 = 'LC' THEN 'Lateral correlation'
           WHEN b.qvc43 = 'DK' THEN 'Select/Omit keyed LF exists over a queried table'
           WHEN b.qvc43 = 'NF' THEN 'Too many tables in query'
           WHEN b.qvc43 = 'NS' THEN 'Non-SQL query interface'
           WHEN b.qvc43 = 'DF' THEN 'Distributed table in query'
           WHEN b.qvc43 = 'RT' THEN 'Read Trigger defined on queried table'
           WHEN b.qvc43 = 'PD' THEN 'Program described file in query'
           WHEN b.qvc43 = 'WC' THEN 'WHERE CURRENT OF a partition table'
           WHEN b.qvc43 = 'IO' THEN 'Simple INSERT query'
           WHEN b.qvc43 = 'CV' THEN 'Create view statement'
-- Following reason codes are gone with V5R4 (probably not a complete list):
           WHEN b.qvc43 = 'LK' THEN 'LIKE clause'
           WHEN b.qvc43 = 'LO' THEN 'LOB columns'
           ELSE b.qvc43 END AS "CQE dispatch reason code",
           qq1000
FROM   dbmon2 a INNER JOIN sqe b ON
         a.qqjfld = b.qqjfld and a.qqucnt = b.qqucnt
WHERE  qqrid = 1000 AND SUBSTR(qq1000,1,4) <> 'HARD' AND
               SUBSTR(QQ1000,1,5) <> 'CLOSE' AND QQ1000 <> ' ' AND
                QQC21 <> 'MT'
ORDER BY 1
```

NOTES:

- I've written this statement with V5R3 in mind. For example, the SQL statement text shown is limited to the first 1000 characters (QQ1000 field without leveraging of MT continuation lines). In V5R4 you could use the new DBMON CLOB field QQ1000L instead; this will show statements up to 2MB in length (that's some statement!).
- On V5R3, PTF SI18184 is required to see the reason codes.

This query gives you which path the statement took (SQE or CQE), the reason code for CQE path and finally the actual SQL statement text. Now you can proceed with any further actions with information in hand rather than just a ballpark estimate. If a large portion of your CQE routed queries are taking that path due to S/O LFs, then perhaps using the QAQQINI setting to ignore those indexes is a valid option to consider. Or if you see a lot of CQE routing due to Read-Only triggers, perhaps it's worth considering dropping some of those not absolutely required by your auditors or management.

Obviously this statement can be further expanded to show a lot more information. I hope I have given you a good base for further customization as well as a glimpse into the tremendous power available at your fingertips via database monitor collections.



Our title's Latin translation is "scraped tablet" or "blank slate". I find it fitting since I intend to talk about the 'best' way to fully erase your system using native iSeries support.
This question arises frequently when Disaster Recovery testing is performed at an offsite location or if a company has upgraded their iSeries and is relocating the old box. Most businesses require the system be totally scrubbed before hand-off.

Over the years I have seen many methods used to clear user data off the iSeries. One of the simplest methods I've seen is:

* DLTLIB on all user libraries
* RCLSTG

But then you have to worry about folders, user profiles, spooled files, output queues, job queues, network attributes, authorization lists.... some folks pursue this list in perpetuity, but I don't feel this is the right approach.

IBM does offer tips on how to wipe data off your disk and the one I like the best involves using LIC install and DST to initialize all the drives with zeros. Here are detailed steps on how to go about that:

1) do a D Manual IPL using a SAVSYS, full system save or LIC install CD
2) select option 1 to "Install LIC"
3) select option 5 to "Install Licensed Internal Code and Initialize System"
4) after the install of LIC is complete, the system will IPL to the DST primary menu
5) Add all the disk drives to system ASP (ASP 1). This writes zeroes on the disk drives so only LIC is loaded
   * take option 3 to "Use DST"
   * Work with Disk Units
   * Work with Disk Configuration
   * Work with ASP Configuration
   * Add units to ASPs
   * type 1 in front of all the available units (if using LIC install CD (I_Base) default password for QSECOFR is QSECOFR
6) Power off the system by pressing the power button two times.
The system is now clean and ready to go!

For partitioned systems, there is more work following step 4. Basically, all the partition drives have to be re-added back to the system ASP.
Here are step 4-b instructions for partitioned systems. You will be presented with these options while in DST:
   * Work with system partitions
   * Recover configuration data
   * Clear non-configured disk unit configuration data
   * Exit back to DST primary menu
   * Follow steps 5 and 6 from this point on.

This type of clean up is sufficient for most iSeries shops.

If however your system contains state secrets, note that this type of disk cleanup does not follow the DOD (Department of Defense) 5220.22-M standard. No 3rd party application can have access to a non-configured disk unit on the iSeries, so there is currently nothing available for the iSeries. IBM recommends that disks be physically destroyed if a DOD standard has to be enforced.
I, on the other hand, believe that one could try configuring the iSeries disk drive inside a PC and use a vendor tool that complies with DOD 5220.22-M (i.e. KillDisk) to wipe all the data off of it.

Reader Input

February 2006 — Top 5 Reasons to Create an Index:
You list a reason to create an index is to support RI Constraints - however, if a constraint is created and no index exists, then DB2 will automatically create an index.

Kent Milligan, DB2 UDB for iSeries Technology Team

Yes, Kent is correct that when you define a referential integrity constraint between two tables, DB2 will automatically create a foreign key index. The database administrator is responsible for explicitly creating a primary key for the table that the child table references.

February 2006 — Fun Dates: I just read your new newsletter. Just a minor comment. The calculation of the last Day of month is not correct! Your formula was:

CURRENT_DATE + 1 MONTH - DAY
(CURRENT_DATE) DAY AS LAST_CUR_MONTH

Assuming the Current_Date is 01/30/2006.
Adding 1 Month will result in 02/28/2006.
Day(Current_Date) will return 30.
If you substract 30 days from 02/28/2006, you won't get 01/31/2006.

The correct formula is:
Current_Date + 1 Month - Day(Current_Date + 1 Month) DAY AS LAST_CUR_MONTH

Viele Grüße / Best regards
Birgitta Hauser—Database Administrator
Thanks Birgitta!

AUTO-INCREMENT AND UNIQUE IDENTIFIER

As long as people have needed to quickly go back and find stored and organized objects, there has been a requirement for a way to uniquely represent each item. The Dewey Decimal System is a good example. Even though today's business applications work best when there is a mechanism that *automatically* generates a unique value for any desired unique identifier, in the past this unique identifier requirement has been fulfilled using various system objects, including data areas, files, user indexes and data queues. Solutions like these have typically worked reliably for decades and should not be changed at a whim ("…if it isn't broken don't fix it.")

Given that, why even bother writing about a feature like auto-incremented database generated unique keys?

1. **We have a number of new developers coming into the System i arena.** These folks are likely to be familiar with SQL as an exclusive language for database definition and manipulation. They're either porting an existing SQL application from SQL Server or Oracle or are fresh college graduates that think of RPG only in terms of Rocket Propelled Grenade. They're familiar with auto-increment features on other platforms and expect DB2 to support them in a similar or identical fashion.

   The System i almost exclusively uses *natural keys*, which are keys built out of one or multiple columns in the table until a unique portion of the row is isolated. On other database platforms it is a lot more common to create *surrogate keys* instead.

   For example, when a social security number cannot be used as a unique identifier due to privacy protection requirements, database architects on non-System i platforms often create artificial keys that uniquely identify an employee or a customer. Since the resulting field's only purpose in life is to serve as a unique identifier, it is designated as a primary key or restricted with a unique constraint. This is also a perfect example of where auto-increment values come in handy for key generation needs. It takes another job off the application developer's plate.

2. **Pushing function down to the database level yields better performance and cheaper maintenance.** You all love and use file triggers. Auto-incremented fields are not much different. Since DB2 for System i version V5R4 is compliant with 2003 SQL core standard, why not take advantage of standardized database generated key functions like GENERATE_UNIQUE?

3. **Auto-incrementation may perform better than your current implementation.** Too often I see implementations like NEXT_ID = SELECT MAX(ID)+1 FROM IDTABLE. Although that may be an extreme example, even atomic RPG reads followed by writes may perform poorer than database generated keys.

My hope is that as you develop new application modules, architect new databases or simply experiment with SQL to build up your resume, some of these auto-increment features will come in handy.

I am going to focus on four pertinent SQL features and will also touch on an alternative non-SQL method that may help address similar business requirements. Here's a list that'll help you decide if it's even worth reading on or not:

- Identity Column (V5R2)
- ROWID (V5R2)
- Sequence (V5R3)
- GENERATE_UNIQUE (V5R4)
- GENUUID MI instruction (V3R6)

As I was going through the intricacies of all these options I realized I could write a book on the subject. Since I have neither the time nor the inclination to do that, I decided to reformat this article instead.

First I'll offer a review of my sample SQL script, touching on each SQL auto-increment option. This review will be based on what I personally found easy and user friendly. Let's call this section EASY READ.

Then I'll dig into more detail on each option, outlining as much technical detail as I find feasible. This part will include a lot of caveats and "gotchas" and will be tough reading. But I do think it's necessary for any real world implementation. Let's call that section TOUGH READ.

With that outline in mind, let's start with the sample script. I need you to paste the following SQL demo script into System i Navigator's "Run Sql Scripts" window and then click Run->All.

NOTE: shortcut to Run Sql Scripts executable is to click on Start->Run->cwbundbs->OK

/* SQL script demonstrating autoincrement and unique identifier features */

```
CREATE TABLE t1 (identity SMALLINT NOT NULL       -- 2 byte binary (15 bits + sign bit)
                    GENERATED ALWAYS AS IDENTITY  -- system always generates value
                    (START WITH 1                 -- initial id value is 1
                     INCREMENT BY 1               -- ids are in increments of 1
                     MINVALUE -32768              -- SMALLINT lowest is -32768 (-(2^15))
                     NO MAXVALUE                  -- for SMALLINT is 32767 (2^15 - 1)
                     CYCLE                        -- does not halt when max reached
                     CACHE 20                     -- allocates in blocks of 20 ids
                     NO ORDER),                   -- ids could have gaps (i.e. 1,2,20,21)
                 rowid ROWID NOT NULL             -- BIT varlen alloc(26) len(40)
                     GENERATED ALWAYS,            -- system always generates unique value
                 sequence DECIMAL(63,0),          -- sequence number
                 data_field SMALLINT NOT NULL,    -- finally, little piece of data :)
                 PRIMARY KEY (identity),          -- disallows duplicate ids
                 UNIQUE (sequence));              -- disallows duplicate sequences

CREATE SEQUENCE s1                                -- sequence name (*DTAARA)
   AS DECIMAL (63,0)                              -- 63 digits (pushing the limit here)
   START WITH 1                                   -- initial value is 1
   INCREMENT BY 1                                 -- sequence increments by 1
   MINVALUE -999999999999999999999999999999999999999999999999999999999999999999 -- why not ☺
   NO MAXVALUE                                    -- in this case, 63 digits (999...)
   CYCLE                                          -- does not halt when max reached
   CACHE 20                                       -- allocates sequences in blocks of 20
   NO ORDER;                                      -- sequences may have gaps

CREATE TRIGGER  sequence_update                   -- BEFORE INSERT trigger name
    NO CASCADE                                    -- only for compatibility for other dbs
    BEFORE INSERT ON t1                           -- trigger fires before actual INSERT
    REFERENCING NEW AS new_ins                    -- alias new row as new_ins
    FOR EACH ROW                                  -- trigger fires for each inserted row
    MODE DB2ROW                                   -- this mode allows BEFORE SQL triggers
    SET new_ins.SEQUENCE = NEXT VALUE FOR s1;     -- assign new key to new row's sequence

INSERT INTO t1 (data_field) VALUES(0);
INSERT INTO t1 (data_field) VALUES(SELECT MAX(data_field)+1 FROM t1);
INSERT INTO t1 (data_field) VALUES(SELECT MAX(data_field)+1 FROM t1);
DELETE FROM t1 WHERE data_field = 1;
```

```
INSERT INTO t1 (data_field) VALUES(SELECT MAX(data_field)+1 FROM t1);
INSERT INTO t1 (data_field) VALUES(SELECT MAX(data_field)+1 FROM t1);

SELECT identity, sequence, HEX(rowid) as "HEX(rowid)",
       SUBSTR(CHAR(rowid),9,2) || '-' || SUBSTR(CHAR(rowid),12,5) AS "Serial Number",
       rrn(t1) as rrn, data_field
FROM t1;
```

/* end SQL script */

An output similar to one in the following window will pop-up:

IDENTITY	SEQUENCE	HEX(rowid)	"Serial Number"	RRN	DATA_FIELD
1	1	8B43B0CC96850000F1F010C1F3 F4F5C240400000000000000001	10-A345B	1	0
4	4	8B43B42C9EAE8000F1F010C1F3 F4F5C240400000000000000001	10-A345B	2	3
3	3	8B43B37C7C928000F1F010C1F3 F4F5C240400000000000000001	10-A345B	3	2
5	5	8B43B43217268000F1F010C1F3 F4F5C240400000000000000001	10-A345B	4	4

EASY READ

Let's dissect our sample script.
Table T1 has 3 auto-incremented columns guaranteed to have unique values and one arbitrary data field. In real life you'll never need more than one unique identifier, but it helps me illustrate similarities and differences in one easy sample.

- Identity column attribute is my personal favorite:

It offers lots of flexibility and power to the database designer. I have created an IDENTITY field, explicitly specifying all available parameters for the identity attribute attached to the IDENTITY column. Furthermore, I have put a PRIMARY KEY on the IDENTITY column, ensuring that duplicates will never enter this field. I could have avoided a need for the primary key had I specified NO CYCLE, but I prefer this solution since it'll thwart even an ALTER TABLE attempt to RESTART the identity sequence with a particular value that has been used in the past.
Values will be generated in ascending order. Each job gets its own CACHE of values and if all values in a cache are not used by a particular job, they'll be lost, resulting in sequence gaps (no biggie; could have been avoided with NO CACHE or ORDER keywords, but performance would suffer).
Since I used SMALLINT data type and specified primary key designator, this table is limited to hold at most 65536 rows (2^16). It'll start at 1, reach the max of 32767, cycle to -32768 and ending row will have an ID of 0. So what happens when all identities have been exhausted?

It's a bit tricky. Primary key will not allow the same IDs in the table so even if you deleted rows and used ALTER TABLE to RESTART the sequence with a particular value you're bound to hit a duplicate key error eventually. Depending on what exactly your application requirement is of this field, this may be a problem. I personally would institute a policy where we clear these values periodically if we hit the limit. Even easier, if you know you're going to hit the limit with the number of insertions in the table, just use the data type that allows for greater range (DECIMAL will go to 31 digits, BIGINT to 19).
Use IDENTITY_VAL_LOCAL() function to retrieve the last value generated from the insert at your level (i.e. job). Programmatic example:

```
VALUES IDENTITY_VAL_LOCAL() INTO :MYVAR
```

Or while experimenting with identity columns use (this is what I used):

```
SELECT IDENTITY_VAL_LOCAL() FROM SYSIBM/SYSDUMMY1
```

- ROWID data type is not something I personally find a use for:

It has its own implicit unique constraint, again guaranteeing unique value in the table. Since it uses the system serial number as part of the auto-generated value, rowid should be unique across systems as well. IBM states that it generates "highly unique value" as well as that values in a table are guaranteed to be unique.
The nice thing about ROWID is that there's no cycle so you don't have to worry about insertions, deletions, reorgs etc. Also, the ROWID() function can be used to get ROWID values from other platforms, like mainframe created tables.

The biggest drawback comes when you need to retrieve the last generated value. You'd have to read the record you've just inserted, which is a lot of overhead for such a simple task. Another drawback is that its unique datatype (variable length 40 byte bit oriented value with allocated length of 26) make it unsuitable for human consumption and understanding. It can't even be compared against another ROWID value. You have to cast it to character if you want to compare it (some characters are not 'typeable', so you'd have to resort to hex representation).

If you need to use it in RPG, here's how you'd declare the data type:

```
D MY_ROWID          S           SQLTYPE(ROWID)
```

- Sequence object is identity column attribute's big brother:

Why "big brother" if it was introduced one release later? Well, it is not table specific so it could generate keys that are unique across the database. It allows for a really large value range (63 digits for DECIMAL data type). It is simply a data area (*DTAARA) object. It is frequently used on other platforms.
You'll notice that parameters are identical to those of the identity column attribute. I have used them in a similar fashion. Also notice that I have specified CYCLE. This means that I could get a duplicate value generated once I've exhausted the range. To avoid possibility of a non-unique key in my table, I specified a UNIQUE constraint on the column itself.

One nice thing about sequence is that you can generate an alpha-numeric key with it. For example, maybe I want to prepend the word ORDER# or company identifier (i.e. IBM, CTI) before the actual number. Here are a couple of examples of this approach:

```
SELECT 'ORDER#' || CAST(NEXT VALUE FOR s1 AS CHAR(5)) FROM SYSIBM/SYSDUMMY1
SELECT 'ORDER#' || SUBSTR(DIGITS(NEXT VALUE FOR s1),59) FROM SYSIBM/SYSDUMMY1
```

As seen in my sample, use the NEXT VALUE FOR function to generate and obtain the next increment of the sequence. Use the PREVIOUS VALUE FOR function to see what the last generated sequence value was, i.e.:

SELECT PREVIOUS VALUE FOR s1 FROM SYSIBM/SYSDUMMY1

If you look at the sample you'll notice that I use a BEFORE INSERT trigger to ensure the new sequence is generated automatically. If I didn't do that I would have to specify NEXT VALUE FOR explicitly in my insert statement. I really like letting the database engine do work for me whenever possible so I took advantage of the trigger functionality. Still, that's a kludge and I don't like it. Triggers sometimes fail or are removed by DBAs for massive data loads. Even worse, they have a negative impact on performance. That said, I can definitely find use for sequences for other needs (i.e. the alphanumeric order number I illustrated earlier).

Back to the SQL script.

In the CREATE statements I have added a number of comments I thought might prove helpful in understanding what each parameter does.

After table t1 and sequence object s1 were created I inserted 3 rows into t1 specifying only DATA_FIELD as the target, letting the database engine generate values for IDENTITY,ROWID and SEQUENCE columns. I could do away with explicit specifications of which columns were being inserted by specifying keyword DEFAULT for auto-generated values, but I figured it isn't as legible as the example I gave.

For data_field, I can specify any arbitrary value but I decided to have some fun with it (I start with 0 and then MAX+1 increments each subsequent insert by 1). In real life you'll probably have 100 real data fields (just kidding).
Since I just created t1 and s1, I know that values generated for both IDENTITY and SEQUENCE columns at this point will be 1,2,3. ROWID holds some arbitrary but unique value, not inherently in ascending or descending order (order's undefined).
Relative record numbers (RRN) are 1,2,3 and data field holds 0,1,2 values.
At this point I do a delete of 2nd row (data_field = 1). The database engine marks this row as deleted. Next, we insert two more rows. Since our table was created via SQL, we are automatically reusing deleted records (REUSEDLT = *YES). Row 2 is reused, so now we see IDENTITY & SEQUENCE are set to 4, while RRN remains at 2. This illustrates the fact that there are no ties between RRN and any of the auto-increment generators. The same goes for ROWID.

Finally, let's look at the SQL select statement. I display all fields, changing the column order a bit for readability. As an add-on, I demonstrate how to pull the serial number out of the ROWID value. It could come in handy when a DBA is trying to figure out what system a particular file came from and the file happens to have a ROWID column.

"Best practice" findings for identity column, ROWID and sequence:

1) Use GENERATED ALWAYS for identity columns and ROWIDs instead of BY DEFAULT. The whole idea is to let the database handle key generation, not the programmer.
2) There could be gaps in actual table rows with identity column values and sequence values regardless of the fact that ORDER may have been specified (ORDER disables CACHE). ORDER will guarantee that a unique key is generated in consecutive order across different jobs inserting into a table, but can't guarantee that the order in the table itself is without gaps. With NO ORDER, if jobs CACHE values and don't use them, there could be gaps in the actual table if not all values are used by jobs. Some things that can effect the gap are CACHEd values between different jobs, deletes and reuse of delete records, rollback, ALTER TABLE, RMVJRNCHG, APYJRNCHG.
3) Uniqueness is not guaranteed for CYCLE defined identity or sequence column. Even with NO CYCLE, someone might potentially use ALTER TABLE to RESTART the sequence and you may end up with duplicates inserted in the table. For this reason, I strongly recommend primary key or unique constraint are specified on the column

itself. This will enforce uniqueness regardless of what values are generated by the identity or sequence generator.
4) To avoid unpleasant surprises, spell out all of the parameters available for the SQL construct. This ensures all possibilities are considered and appropriate combinations of values are selected. Default values on some of these have somewhat complex interdependencies and I find it's safest if I explicitly specify them at creation time.
5) CACHE definitely helps performance and I strongly recommend its use whenever possible. This could introduce ID gaps though (i.e. 1,2,20,21,3,4,22).
6) I prefer identity column to ROWID for the unique identifier in the table. It's just easier to control and data types lend themselves for normal application usage (assignment, comparison etc.) while ROWID's unique data type does not.
7) Sequences are useful for cross-table uniqueness as they're independent objects (data area). Also they allow for handy alphanumeric key generation.
8) Relative record number has no ties to any of the auto-generated values.
9) ROWID row has to be read to see the generated value while the identity generated value can be looked up using the IDENTITY_LOCAL_VAL() function. Sequence generated values can be looked up using PREVIOUS VALUE FOR function (beware of concurrency though).

- GENERATE_UNIQUE is one neat SQL function:

I haven't demonstrated use of the GENERATE_UNIQUE() function in the sample script since I know most people don't have V5R4 loaded yet. If I included GENERATE_UNIQUE in the sample, most folks could not run it successfully. I do have a version of the script with it included for anyone that wants it (email info@centerfieldtechnology.com with request). In the 2nd sample I add another field to t1, generate_unique CHAR(13) FOR BIT DATA. I then add another line to the BEFORE INSERT trigger, SET new_ins.generate_unique = GENERATE_UNIQUE(). So, it works on the same principle as the sequence sample.
One difference is that I do not have to specify a UNIQUE constraint since each invocation of this function is guaranteed to return a unique value (I could still specify UNIQUE on the column itself). Among other things, this function uses UTC timestamp and serial number as part of the generated key. Timestamp ensures key is in ascending order and serial number should ensure uniqueness across different systems.
Database generator ensures no two values are the same even if timestamps for 2 or more rows are.
Guess what, you can retrieve that timestamp out of the field as well! For example:
SELECT TIMESTAMP(generate_unique) FROM t1

"Best practice" findings for GENERATE_UNIQUE

1) Use BEFORE INSERT trigger to enforce auto-increment unique key generation
2) Don't use in commercial products that need to run on systems with OS version lower than V5R4
3) UNIQUE column constraint is not necessary but can't hurt (belt and suspenders ☺)

- GENUUID MI instruction makes me want to go native:

GENUUID stands for Generate Universal Unique Identifier. It sounds impressive and it is. It generates IDs unique across space and time! C type include for the procedure and input data structure declaration are located in the QSYSINC/MIH/GENUUID header file.
It was originally intended to be used in networking but I don't see a reason why it can't be used as a transaction identifier. Engineering specs state that up to 10 million unique IDs per second could be generated, but I haven't seen any IBM literature confirming or denying this velocity.
If I was to include GENUUID in my SQL sample script, I'd create an SQL User Defined Function (UDF) that would map to a routine in my service program. Output variable should probably be declared as CHAR(16) FOR BIT DATA as GENUUID returns the unique id as a 128 bit value.

The only drawback for the UDF would be that since it is mapped to an external function it would not be portable to other platforms. Not a big concern for me as everything we do at Centerfield is System i specific.

"Best practice" findings for GENUUID

1) Use it if you can! It's a good solution.
2) There was bug pertaining to systems with multiple processors stressing the function call. Corrective PTFs are MF33189 for V5R1, MF33191 for V5R2, and MF33190 for V5R3. Apply them, just in case.
3) Create an external UDF mapped to a service program routine. This makes it integrate with SQL seamlessly.
4) To use it as a unique key generator, use the same principle I demonstrated for sequence object (BEFORE INSERT trigger).
5) You can define UNIQUE constraint for the table column but don't really need to.
6) Check out next section (Tough Read) for more details and sample for GENUUID.

TOUGH READ

Identity Column revisited

Identity column is a special attribute attached to one and only one column in a given table. This attribute instructs the database engine to generate sequential values for every newly inserted row.

- An identity column can be specified at CREATE TABLE time or later with ALTER TABLE command. If using ALTER TABLE note that existing columns cannot be altered to have identity attribute. Only newly added columns can be added and carry identity attribute. Furthermore, DDS created physical files cannot be altered to carry columns with identity attribute.
- An identity column is implicitly NOT NULL.
- An identity column is not allowed in a partitioned or distributed table.
- SMALLINT, INTEGER, BIGINT, DECIMAL or NUMERIC data types can be specified as identity columns. For the last two, precision cannot exceed 31 digits and scale must be 0. DECIMAL data type is more portable to other databases than NUMERIC so if you must choose between the two, opt for DECIMAL. That said, keep in mind that BIGINT's max is (2^63)-1, nineteen digit value, and should be more than sufficient for most applications.
- START WITH parameter defines starting value for the column. If not specified, MINVALUE is the starting value for the ascending sequence and MAXVALUE for descending sequence. If MINVALUE and MAXVALUE were not specified either, starting value and MINVALUE for the ascending sequence is 1.
  Conversely, if MINVALUE is not specified and START WITH is, starting value is used as effective MINVALUE (or MAXVALUE if descending sequence – i.e. negative increment value).
- INCREMENT BY specifies the interval between two consecutive values in the sequence. Positive values make the sequence ascending and negative make it descending. Zero will result in the identity value never changing. Default is 1 and I found it most useful.
- MAXVALUE must be greater than minimum value. If not explicitly specified, it's the maximum positive value for the data type or the START WITH value if descending sequence was specified (negative INCREMENT value). It's -1 if not specified and descending sequence and no START WITH was specified.
- MINVALUE must be smaller than maximum value. If not explicitly specified and descending sequence was chosen (negative INCREMENT value), it's the smallest possible value for the data type. If not specified and ascending sequence was chosen, it's either the START WITH value or 1 if START WITH was not specified.

- CACHE + integer value specifies how many values to pre-allocate in memory. This is a better performing alternative but can result in gaps in the target column. Minimum integer value is 2, default is 20 (if no integer is specified).
- NO CACHE means each value is generated in a serial fashion. This means a call has to be made to the database generator each time and will perform poorer than the CACHE option.
- CYCLE means that values will continue to be generated once maximum or minimum value is reached. When CYCLE is in effect, duplicate values can be generated by the database manager for an identity column. If a unique constraint or unique index exists on the identity column, and a non-unique value is generated for it, an error occurs.
- NO CYCLE means that database generator stops generating values once maximum or minimum value is reached. This is the default.
- ORDER turns off cache and guarantees that values are generated in the order of request.
- NO ORDER specifies that values do not need to be generated in order of request. This is the default and performs better than ORDER.

You can use SQL syntax ALTER TABLE <table name> ALTER COLUMN <column name> to:

- Change attributes of the existing identity column (i.e. RESTART the sequence with user specified value). For example:
  ALTER TABLE t1 ALTER COLUMN identity RESTART WITH 1
- Add a new column with the identity definition (if one does not exist yet)
- Drop an existing identity attribute definition without dropping the column itself. For example:
  ALTER TABLE t1 ALTER COLUMN identity DROP IDENTITY
  The column identity remains as a regular SMALLINT field, but the identity attribute is dropped. The system will no longer generate values for this column.
- You cannot designate an existing column as identity column, only new columns

You can use a non-deterministic IDENTITY_VAL_LOCAL() function to retrieve the value generated by the database generator. Here's a programmatic example:

VALUES IDENTITY_VAL_LOCAL() INTO :myVariable

Some attributes of this function are:

- This function is not affected by the following statements: UPDATE, COMMIT, ROLLBACK, and Non-Identity INSERT (i.e. INSERT into a table without the identity column).
- Output result is DECIMAL(31,0) data type
- The value returned is the value that was assigned to the identity column of the table identified in the most recent INSERT statement issued at the same level for a table containing an identity column. A new level is initiated when a trigger, function, or stored procedure is invoked so take care when using in that situation.
- I recommend GENERATED ALWAYS be used, but if you use GENERATED BY DEFAULT then a user can supply his own values and override the database generator.
- The result can be null. The result is null if an INSERT statement has not been issued for a table containing an identity column at the current processing level. This includes invoking the function in a before or after insert trigger.
- InfoCenter has some samples on what's returned in different circumstances (i.e. with triggers, without triggers etc.).

You can update the value stored by identity column by issuing the following statement:

UPDATE t1 SET (identity, data_field)= (DEFAULT, 47) WHERE data_field = 55

In the example above we just put the next generated sequence value into IDENTITY column. You can also specify your own literal value by using OVERRIDING SYSTEM VALUE clause:

UPDATE t1 OVERRIDING SYSTEM VALUE
SET (identity, data_field) = (786, 47) WHERE data_field = 55

Avoid using CPYF to duplicate tables with identity columns in them. Check out V5R2 APAR SE13418 for details on permanent restriction for this scenario. Use CRTDUPOBJ instead.

Sequence object revisited

Sequence object is a special data area (*DTAARA) object that allows for generation of unique keys independent of any database table. In principle, they are very similar to the identity column attribute (i.e. share all the same keywords) so I'm not going to repeat all of them, but rather focus on relevant differences.

- Sequences can be shared across many tables
- Sequences are not tied to a column and are accessed separately
- They are not treated as any part of a transaction's unit of work
- You can use NEXT VALUE FOR to generate and retrieve the next value in the sequence
- You can use PREVIOUS VALUE FOR to retrieve the previously generated value. At least one NEXT VALUE FOR must run prior to PREVIOUS VALUE FOR, else an error is generated.
- NEXTVAL and PREVVAL are syntax alternatives for NEXT VALUE and PREVIOUS VALUE
- NEXT and PREVIOUS can be used within any SELECT statement that does not contain:
    - DISTINCT keyword
    - GROUP BY clause
    - ORDER BY clause
    - UNION keyword
    - INTERSECT keyword
    - EXCEPT keyword

  Additionally, VALUE functions cannot be used in:
    - select-clause of the subselect of an expression in the SET clause of the UPDATE statement
    - within a materialized query table definition in a CREATE TABLE or ALTER TABLE statement
    - within a CHECK constraint
    - within a view definition
    - CASE expression
    - Parameter list of a column function
    - Sub-query in a context other than those explicitly allowed
    - SELECT statement for which the outer SELECT contains a DISTINCT operator or a GROUP BY clause
    - SELECT statement for which the outer SELECT is combined with another SELECT statement using the UNION, INTERSECT, or EXCEPT operator
    - Join condition of a join
    - Nested table expression
    - Parameter list of a table function
    - WHERE clause of the outermost SELECT statement or a DELETE, or UPDATE statement
    - ORDER BY clause of the outermost SELECT statement
    - IF, WHILE, DO . . . UNTIL, or CASE statements in an SQL routine

  This will work:
  UPDATE t1 SET sequence = NEXT VALUE FOR s1
  This will not:
  UPDATE t1 SET sequence = (SELECT NEXT VALUE FOR s1 FROM t1)
- If there are multiple instances of a NEXT VALUE expression specifying the same sequence name within a query, the sequence value is incremented only once for each row of the result, and all instances of NEXT VALUE return the same value for a row of the result.
- NEXT VALUE FOR cannot return NULL.
- NEXT VALUE will generate a value even if the statement fails or is rolled back.
- Sequence can be altered using ALTER SEQUENCE
- Sequence can be used to generate alpha-numeric values with the help of SQL's CONCAT function
- If a sequence is referenced in a statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the sequence identified in the statement,
  - The USAGE privilege on the sequence AND
  - The system authority *EXECUTE on the library containing the sequence
- Administrative authority

Normally, for SELECT NEXT VALUE FOR s1 FROM t1 type of statement, a unique value will be generated for each row retrieved from table t1. That is not always the case in some environments that use SQL cursors, so I'm going to list some special considerations for cursor usage:

- If blocking is done at a client in a DRDA environment, sequence values may get generated at the DB2 server before the processing of an application's FETCH statement. If the client application does not explicitly FETCH all the rows that have been retrieved from the database, the application will never see all those generated values of the sequence (as many as the rows that were not FETCHed). These values may constitute a gap in the sequence.
- A reference to the PREVIOUS VALUE expression in a SELECT statement of a cursor is evaluated at OPEN time. In other words, a reference to the PREVIOUS VALUE expression in the SELECT statement of a cursor refers to the last value generated by this application process for the specified sequence PRIOR to the opening of the cursor and, once evaluated at OPEN time, the value returned by PREVIOUS VALUE within the body of the cursor will not change from FETCH to FETCH, even if NEXT VALUE is invoked within the body of the cursor.
- If PREVIOUS VALUE is used in the SELECT statement of a cursor while the cursor is open, the PREVIOUS VALUE value would be the last NEXT VALUE for the sequence generated before the cursor was opened. After the cursor is closed, the PREVIOUS VALUE value would be the last NEXT VALUE generated by the application process.

ROWID revisited

Using ROWID is another way to have the system assign a unique value to a column in a table. ROWID is similar to identity columns, but rather than being an attribute of a numeric column, it is a separate and unique data type.

- ROWID is a unique SQL data type similar in nature to VARCHAR (40) ALLOCATE(26)
- Column declared as ROWID has an implicit UNIQUE constraint built into the file
- Only one column in a table is allowed to have a ROWID data type
- New column of type ROWID can be added to table via ALTER TABLE but existing column cannot be used.
- The ROWID data type is not subject to CCSID because it is treated by DB2 UDB as bit-oriented data.
- ROWID generation algorithm produces highly unique value with System i serial number being part of the generated value.
- ROWID operand can only be assigned to another ROWID operand. Assigning a specific (non-generated) value to ROWID field will only work for column defined as GENERATED BY DEFAULT or if OVERRIDING SYSTEM VALUE clause is specified.
  ROWID(<input value>) SQL function casts a character string to row ID so use that to assign a ROWID value. Although the string can contain any value, it is recommended that it contain a ROWID value that was previously generated by DB2 UDB for OS/390 and z/OS or DB2 UDB for iSeries to ensure a valid ROWID value is returned. For example, the function can be used to convert a ROWID value that was cast to a CHAR value back to a ROWID value.
- ROWID cannot be compared to any data type. To compare the bit representation of a ROWID, first cast the ROWID to a character string, i.e. CHAR(ROWID).
- ROWID values are maintained even across table reorganizations.
- To retrieve the last written ROWID value, you would need to re-read the row that was just inserted.

- Since ROWID values are unique, it is perfect for direct access, i.e. employee lookup. Here's an example of direct access:
  SELECT EMPNO FROM EMPLOYEE WHERE
  EMP_ROWID = ROWID(X'F0DFD230E3C0D80D81C201AA0A280100000000000203')
- ILE RPG for iSeries does not have a variable that corresponds to the SQL data type ROWID. To create host variables that can be used with this data type, use the SQLTYPE keyword. The SQL pre-compiler replaces this declaration with an ILE RPG for iSeries language declaration in the output source member. ROWID declarations can be either standalone or within a data structure.
  The following declaration:
  ```
  D MY_ROWID         S          SQLTYPE(ROWID)
  ```
  Results in the following precompiler generation:
  ```
  D MYROWID          S          40A  VARYING
  ```
  Some caveats that apply to use of ROWID in RPG:
  - SQLTYPE, ROWID can be in mixed case.
  - ROWID host variables are allowed to be declared in host structures.
  - SQLTYPE must be between positions 44 and 80.
  - When a ROWID is declared as a standalone host variable, position 24 must contain the character 'S' and position 25 must be blank.
  - The standalone field indicator 'S' in position 24 should be omitted when a ROWID is declared in a host structure.
  - ROWID host variables cannot be initialized.

GENERATE_UNIQUE revisited

The GENERATE_UNIQUE function returns a bit data character string 13 bytes long (CHAR(13) FOR BIT DATA) that is unique compared to any other execution of the same function. The function is defined as not-deterministic.

- The result of the function is a unique value that includes the internal form of the Universal Time, Coordinated (UTC) and the iSeries system serial number. The result cannot be null.
- Each successive value is greater than the previous value, resulting in ascending order of unique values.
- This function differs from using the special register CURRENT TIMESTAMP in that a unique value is generated for each row of a multiple row insert statement or an insert statement with a fullselect.
- The timestamp value that is part of the result of this function can be determined using the TIMESTAMP function with the result of GENERATE_UNIQUE as an argument. For example:
  SELECT TIMESTAMP(GENERATE_UNIQUE()) FROM SYSIBM/SYSDUMMY1
  Therefore, the table does not need a timestamp column to record when a row is inserted.

GENUUID revisited

This MI instruction generates a universal unique identifier and returns it in the template provided as input/output argument.

- The UUID is unique as an identifier across all time and space and is consistent with the Open Systems Foundation (OSF) Distributed Computing Environments (DCE) version 1 UUID specification described in the DCE's "Architecture Environment Specification/Distributed Computing: for Remote Procedure Calls", Appendix A.
- UUID is 128 bits long, and if generated according to one of the standards, it is guaranteed to be different from all other UUIDs/GUIDs generated until 3400 A.D.

- The template identified by operand 1 must be 16 byte aligned. The 16 byte Universal Unique Identifier (UUID) is returned in the UUID return template. See header file QSYSINC/MIH/GENUUID for template details.
- There is a corrective PTF for multi-processor systems. PTFs are MF33189 for V5R1, MF33191 for V5R2, and MF33190 for V5R3.
- Look up MI instruction details at: http://publib.boulder.ibm.com/iseries/v5r1/ic2924/tstudio/tech_ref/mi/GENUUID.htm
- To learn more about UUID standard, read the Internet Engineering Task Force (IETF) article at: http://www.ics.uci.edu/~ejw/authoring/uuid-guid/draft-leach-uuids-guids-01.txt
- Here's an RPG sample demonstrating the call to the MI built-in:

```
H option(*NoSrcStmt) DftActGrp(*No)

D UUIDTemplate  ds
D   bytesProv                   10u 0 Inz(%Size(UUIDTemplate))
D   bytesAvail                  10u 0
D   reserved                     8a   Inz(*Allx'00')
D   uuid                        16a

D GenUUID         pr                  ExtProc('_GENUUID')
D  UUIDTemplate                   *   Value

C                   callp     GenUUID( %Addr(UUIDTemplate))
C                   eval      *inlr = *on
C                   return
```

Variable uuid will contain a unique value.
This can and should be extended to a service program routine over which a UDF is created for ease of use in SQL.