SQL Tricks & Treats

```
UPDATE QGPL/TEST a SET
C1 = rrn(a),
C2 = rand(47)* 10000,
C3 = substring
('alkowlkaajosuiyireoqxskfjyt',
        integer(mod((rand(47)*
        100+20),  20)),
        integer(mod((rand(34)
        *100+26),6 ))),
C4 = rand()*1345
```

Have you ever wanted to do something with SQL but weren't sure how to do it? We have all been in that situation. This column will explore some problems we've either been asked about or tried to solve ourselves and will give you some ideas on how to get started.

Generating test data

Let's say you want to create a test table and want to populate it with random data to do some testing but need to make it large enough so you can see how well your query will perform. Here is an example SQL statement that you can use to generate different patterns. First of all lets assume we have an SQL table that was created with:

```
CREATE TABLE QGPL/TEST
(C1 INT NOT NULL WITH DEFAULT,
 C2 DEC ( 7,  2) NOT NULL WITH
DEFAULT,
C3 CHARACTER ( 10) NOT NULL WITH
DEFAULT,
C4 NUMERIC (5 ) NOT NULL WITH
DEFAULT)
```

Now let's initialize the file with 1,000,000 rows using:

```
INZPFM FILE(QGPL/TEST) RECORDS
(*DFT) TOTRCDS(1000000)
```

This will put zeros in the numeric columns and blanks in the character column. Now let's update all of the rows putting some predictable data in some columns and random data in others.

This statement uses the RRN (relative record number) and RAND (random) functions to generate data. The RRN function, in this case, simply generates a number from one to a million and updates the first column. The RAND function generates a number between 0 and 1 and is then multiplied by another number to get the appropriate range to fit into the numeric type. The RAND function is also used to generate various strings based on an input string by taking a random substring (both start position and length) and updating the character field.

Varchar or not to Varchar

If you suspect you have character columns that are wasting a lot of space, you may want to consider making them variable length character columns. Let's say for example you have a text description column that is 100 characters long, but you suspect that it either isn't used or only a small number of characters are ever used. If that is the case, then you could save a lot of disk space and memory if the field was turned into a variable length character field so that only the used portion of the column was actually stored (plus a bit of overhead for the size and information on how to locate the extra data since it won't be stored with the fixed length columns). So how do we figure that out?

Here is the answer…use the LENGTH built in function to find out the make-up of the column:

```
SELECT AVG(LENGTH(TRIM(MEMO)))
FROM PROD/CUSTMEMO
```

This statement takes the character column, MEMO and first uses the TRIM function to take off any trailing blanks. The next step is to use the LENGTH built-in function to determine the actual number of non-blank characters in the column. The last step is to average all of the lengths for all of the rows in the table. This will tell you what the average size of the usable data. If you want to be more scientific and look at other statistics like standard deviation you can also use those functions to determine what the variance is in the length of the data.

If it looks like there is a lot of wasted space in the column, the next step is to determine how to create the variable length column. When a VARCHAR column is defined, it can have an ALLOCATE attribute. For example, here is a create table statement that defines a variable length character string.

```
CREATE TABLE QGPL/VC
(MEMO VARCHAR (100 ) ALLOCATE(20)
NOT NULL WITH DEFAULT)
```

This tells the database manager to create a character field but if the data is of length 20 or less, then put the character string with the rest of the fixed length data. When the row is read by a query, this data will be in memory and will provide fast performance. If the data is longer than 20 characters, the data will be stored in a separate area and require two disk I/O's to read.

The bottom line is that if you want the best possible read performance and the column information is always needed then use a larger ALLOCATE value. If very little of the column's space is actually used or the column is rarely needed then make the ALLOCATE value small.

Mike Cain
IBM Rochester's Senior Technical Staff Member within the IBM Systems and Technology Group and team leader of the DB2 UDB for iSeries Center of Competency.

IRON CHEF CHICAGO

DB2 UDB iSeries

Mark Holm
CTO and founder of Centerfield Technology. Works at IBM on the new DB2 query optimizer. Has held various management & technical positions in his 20-year career in the computer industry with IBM, Showcase Corporation (now SPSS) and CTI.

Thursday October 20, 2005

schedule

2:00 - 3:00PM
Database Tuning — a Holistic Approach
Mark Holm
The iSeries has become more and more of a "database machine" in the last ten years. With the growing popularity of SQL, ODBC and JDBC, journaling, and advanced database features like triggers, it has become necessary to understand how to tune the database. This presentation provides a balanced view of database optimization in the context of complex application architectures and complicated queries. You'll leave with many valuable suggestions you can take back to your shop and apply immediately.

3:00 – 4:00PM
insure/INDEX & ANALYSIS Demonstration

4:00 – 5:00PM
To DBA or not to DBA
Mike Cain
While the AS/400 and iSeries servers have always had an integrated relational database management system, the need for a DBA has traditionally been avoided, or very limited. In this presentation, we will explore the definition of a DBA, and whether or not a DBA is now advantageous or even required, in an iSeries environment. The traditional tasks of a DBA will review from the DB2 UDB for iSeries and SQL perspective, with insights on how to build DBA skills.

5:00 – 6:00PM
Expert Roundtable
Mark Holm, Mike Cain and Elvis Budimlic
This hour is an open forum where you can ask database experts challenging questions about your particular environment or application issues.

Registration not required, however our space is limited. Please register by email: chicago2005@centerfieldtechnology.com **or online. If you have questions about the event, please visit:** http://www.centerfieldtechnology.com/chicago2005.asp

A Beautiful View

In our last newsletter we talked about the use of SQL views to simplify a complex database by hiding unnecessary columns from end users and by giving the remaining columns understandable names. This issue will discuss the use of views to hide file relationships and to turn data into readable values.

View into a Relationship

Good database design means that facts (data elements) are stored only once. For example, a database may have one table that stores addresses. These addresses may be for customers, businesses, or employees. The data associated with customers or employees would be stored in other tables. Therefore, to get a name and address for a customer, two or more tables must be joined together. In complex production databases these file relationships may involve five to ten tables to extract information about a single event like a product sale. Often times, the file relationships are not documented well or at all.

As a result, it is often difficult to assemble information with SQL queries – even for technical personnel who work with the files often. End-users who attempt to write their own reports typically need a lot of support to understand these relationships. This leads to a drain on already overburdened IT staff.

Using SQL views, the file joins can be done once and reused. This is a much better approach than joining the files in queries for three reasons.

The first is that the file relationships are documented within the view. Since the relationships between tables may not be written down, the view at least provides one way to document the database. Furthermore, the view name can provide a clue as to the resultant data to help users find the information more quickly.

The second is that no one needs to remember how the tables are joined once the view is in place. Essentially, the view provides a level of abstraction so the end-user does not need to know the details of the relationships and can focus on their goal instead of unnecessary details.

The third reason is that the view can insulate queries and reports from table changes. If, for example, it is decided that a file needs to be split into two separate files the view can potentially be changed to include the new file without forcing reports to change.

Cryptic to usable

When programmers build applications, they often try to minimize the amount of space taken by the file. One of the most common techniques to do this is to encode data. For example, rather than have a long character field to describe a status, a number is used instead. If the application has a common routine to convert the number into the text description for application reports, this may not pose any problem. If users query the database directly using an SQL-based tool however, this conversion is not available. As a result, end-users must come up with another way to convert the coded value into a readable form.

The good news, is that SQL views can come to the rescue! Depending on how many unique values there are, two techniques can be used to solve this problem.

The most simple approach is to use built-in SQL support to convert the encoded value into a text string within the statement itself. Lets say, for example, the database has a state column that contains a number between one and fifty. In most reports, the state needs to be in text form. To accomplish this the view would be created in the following way:

```
CREATE VIEW  state  (column1, column2, stateName,
column4) AS
SELECT
    SomeColumn1,
    SomeColumn2,
    CASE  statenumber
      WHEN 1 THEN 'Alabama'
      WHEN 2 THEN 'Alaska'
      WHEN 3 THEN 'Arizona'
       o
       o
       o
      WHEN 48 THEN 'West Virginia'
      WHEN 49 THEN 'Wisconsin'
      WHEN 50 THEN 'Wyoming'
    END  stateName,
    SomeColumn4
FROM
    CustomerInfo
```

When data was selected from the 'state' view, the text names would be returned instead of the numeric values.

Page Three

What if data is in the table that does not match the criteria specified in the CASE clause? The answer is that a null value would be returned. If that happened you might see returned values that looked like:

STATENAME
Alabama
Arizona
_____-_____ ⬅Null value; actual value in physical file was 0.
Wyoming

If too many values exist in the table to be conveniently coded in the SELECT part of the view definition, then it is necessary to use a join to accomplish the same goal. In this case, the coded value and the text value are stored together in a separate physical table and joined into the production file. The view might look something like this:

```
CREATE VIEW  state  (column1,
column2, stateName, column4) AS
SELECT
     A.SomeColumn1,
     A.SomeColumn2,
     B.StateName
     A.SomeColumn4
FROM
     CustomerInfo A,
     StateCodes B
WHERE
     A.StateCode = B.StateCode
```

In this case, the state code from the production file is joined to the same column in another table that contains the descriptive data and the text column is returned from the second table.

The major advantage of this approach is that as the data values grow or change, the secondary table simply needs to be updated and the view does not have to be modified. The disadvantage is the performance penalty that gets introduced when tables are joined together.

In our next issue we'll talk about using views to further provide report formatting and to help provide an additional layer of security to production data.

How do you know that your application is and will continue to perform as well as it possibly can?

Case Study, *Anylarge, Inc.*

Upon initial deployment, Anylarge's new homegrown XYZ application was doing fine. Users loved it, IT was a hero for once, ROI was on the high positive side. Anylarge IT brought up a few more sites with similar positive results…response time was a little slow, but a planned hardware upgrade took care of that.

Then, about a year after the initial go-live, after numerous enhancements and more than 1000 end users in production, Anylarge IT started getting reports of slow response time, especially in query and reporting functions. A sudden downturn in the business climate made an out-of-budget hardware upgrade impossible.

The original application had been written with the help of several consultants and some of the logic behind the SQL and indexing strategy was either forgotten or had become indecipherable to Anylarge's now-smaller IT staff. A few unscheduled executive meetings took place, with the sting of "Take care of this or else…" e-mail following the meetings.

Fortunately, one of Anylarge's iSeries administrators had learned of a unique service available from Centerfield Technology. Called a database/

ASSESSMENT, the service allowed Anylarge IT, with Centerfield's assistance, to design a collection strategy that turned on iSeries dbMON during periods of the worst slowdowns. The resulting dataset was then sent to Centerfield for analysis. Two weeks later, an 85 page document was sent to Anylarge, and a 2 hour conference call with Centerfield predicted that building several indexes and making several modifications to SQL statements would substantially improve performance without adding hardware resources.

With nearly immediate performance improvements in hand, Anylarge IT was also able to justify purchase of the Centerfield Technology tools and training which proved valuable to both development and operations staff as they incorporated intelligent database tuning and diagnostics into their process.

While the names and characters in this story are fictional, the scenario is very common in the large group of iSeries shops that end each day on a happier note because they use Centerfield Technology iSeries performance tuning and diagnostic tools.

iSeries database/ASSESSMENT
IBM ServerProven®

Learn More:
http://www.developer.ibm.com/gsdod/solutiondetails.do?solutionId=16389&lc=en

EASY Button for Executing

iSeries Commands From SQL

Every once in a while I find myself in need of calling an iSeries command or program from SQL. For many years, obvious solution has been to use OS built-in command interpreter QCMDEXC and call it as a stored procedure.

To be clear, QCMDEXC is not really a stored procedure (i.e. no CREATE PROCEDURE has been done for it) but iSeries allows for dynamic CALL to programs on the iSeries with the following caveats:

- ♦ All arguments are treated as IN type parameters.
- ♦ The CALL type is GENERAL (no indicator argument is passed).
- ♦ The program to call is determined based on the procedure name specified on the CALL and the naming convention.
- ♦ The language of the program to call is determined based on information retrieved from the system about the program.

But I digress, so let me get back to QCMDEXC.

I love the power it offers, but I absolutely abhor the fact that I have to count up the number of characters in the command to pass them as DECIMAL(15,5) as expected for the 2nd parameter.
Pain is somewhat alleviated by ability of languages like Java to return string length and cast it to appropriately formatted Decimal object.

However, I kept thinking, why can't I just pass the command string and let system figure out the length. That's exactly what I've done by creating EXCCMD stored procedure.

First run this on your system to create your own command executor procedure. This is a one time configuration step.

```
CREATE PROCEDURE QGPL.
EXCCMD
(IN command VARCHAR(500))
LANGUAGE SQL
BEGIN
DECLARE commandLength DECIMAL
(15,5);
SET commandLength = DECIMAL
(LENGTH(command),15,5);
CALL QSYS.QCMDEXC(command,
commandLength);
END
```

Then in your SQL scripts and even in interactive SQL (i.e. STRSQL or RunSqlScripts) you can use it as:

```
CALL EXCCMD('RMVM FILE(QGPL/
TABLE1) MBR(JANUARY)')
```

Elvis

Elvis Budimlic
Development Director

Ka-Ching For Your Product Ideas

expert@centerfieldtechnology.com

Couple of weeks ago we received a question about simulating Outer, Inner, Left & Right joins using DDS on our Ask-the-Expert website. The answer to that deserves to be shared with our readers.

My personal preference in creating non-keyed LFs is to use SQL views, as they allow me to use any of the SQL join keywords available on the DB2 UDB for iSeries and read it via SQL or RPG as non-keyed logical file (i.e. arrival sequence). However, if you want keyed LF or your business rules dictate using DDS, then DDS it will be!

In this article I am going to illustrate SQL join keywords (INNER, LEFT OUTER, RIGHT OUTER) and equate them to their powerful predecessor DDS.

Let me first list 2 physical file sample DDS sources that will be joined by subsequent DDS and SQL files:

```
PF1 dds source
     A          R REC1
     A            NBR          10
     A            NAME         20
     A          K NBR

PF2 dds source
     A          R REC2
     A            NBR          10
     A            SALARY        7
2
     A          K NBR
```

These are very straightforward and minimal samples of two physical files that user may want to join to see both name and salary for a specific employee.

INNER JOIN

Inner or equi-join is a default join type for join logical files that do not specify the JDFTVAL keyword. It gives all the records from the primary file PF1 along with matching records from the secondary file PF2.

This join is illustrated in the JLF1 dds source sample. Its SQL equivalent would be:

```
CREATE VIEW JLF1 AS
SELECT A.NBR, A.NAME, B.SALARY FROM PF1 A
INNER JOIN PF2 B ON A.NBR = B.NBR

JLF1 dds source
     A          R JOINREC    JFILE(PF1 PF2)
     A          J            JOIN(PF1 PF2)
     A                       JFLD(NBR NBR)
     A            NBR        JREF(PF1)
     A            NAME
     A            SALARY
```

LEFT OUTER JOIN

This join will return all matched records from primary and secondary files as well as non-matched records from the primary file. For non-matched records in the primary file, default values of the secondary file's field are returned.

This join requires use of JDFTVAL keyword in the DDS for join logical file.

This join is illustrated in the JLF2 dds source sample. Its SQL equivalent would be:

```
CREATE VIEW JLF2 AS
SELECT A.NBR, A.NAME, IFNULL(B.SALARY,0.00)
AS SALARY FROM PF1 A LEFT OUTER JOIN PF2 B
ON A.NBR = B.NBR

JLF2 dds source
     A                       JDFTVAL
     A          R JOINREC    JFILE(PF1 PF2)
     A          J            JOIN(PF1 PF2)
     A                       JFLD(NBR NBR)
     A            NBR        JREF(PF1)
     A            NAME
     A            SALARY
```

RIGHT OUTER JOIN

This type of join is the same as a left outer join with the tables specified in the opposite order. In fact, that is exactly how it's implemented on the iSeries. To get this type of join in a LF, you simply need to reverse the order files are joined in. This join is illustrated in the JLF3 dds source sample. Its SQL equivalent would be:

```
CREATE VIEW JLF3 AS
SELECT B.NBR, IFNULL(A.NAME,' ') AS NAME, B.
SALARY FROM PF1 A RIGHT OUTER JOIN PF2 B ON
A.NBR = B.NBR

JLF3 dds source
     A                       JDFTVAL
     A          R JOINREC    JFILE(PF2 PF1)
     A          J            JOIN(PF2 PF1)
     A                       JFLD(NBR NBR)
     A            NBR        JREF(PF2)
     A            NAME
     A            SALARY
```

There are other type of joins in SQL that are not as readily available in the DDS, like EXCEPTION JOIN and like. To do some of these joins in DDS we would have to resort to select/omit criteria (keyword S or O).

Most of these views will perform just as well as join LFs as long as indexes on the join fields exist either as primary keys on the physical files themselves or SQL indexes are built for that specific purpose.

IBM's development direction is SQL, and even though DDS is very powerful and offers some unique capabilities, I recommend using SQL when circumstances allow it.

Elvis

Centerfield TECHNOLOGY

Encoded Vector Indexes

To use or not to use -- that is the question!

When IBM introduced the Encoded Vector Index (EVI) in 1998, it was introducing one of the most revolutionary database technologies seen in DB2 for quite some time. Here we are, over six years later and Centerfield still gets many questions about these mysterious indexes. In general, a vast majority of the iSeries shops do not use them and some of those companies are missing an opportunity to give their query and reporting work a good performance boost.

This article won't cover the inner workings of an EVI. Instead, we'll give you a high level overview and give you some guidelines to help you decide if they make sense in your database. If you want to understand the data structures that an EVI is built on there are some web resources at the end of the article.

As a general rule, traditional indexes created as a result of a CRTLF command or using the CREATE INDEX SQL syntax are very efficient at finding and retrieving small amounts of data. On the other hand, if a query needs to process a large percentage of a file or table then it is much more efficient to read the records in arrival sequence. This is because using an index to fetch data will almost always result in small, random disk I/Os on the rows associated with the index keys. Because the rows associated with the keys are generally not sorted the way the keys are placed in the index, the row will normally not be in memory unless the data happens to be added that way or someone has reorganized the file using that logical file to reorder the data.

Because of these factors, traditional indexes work well when fetching 20% or less of the rows in a table. Table scans work well when more than 70% of the rows are processed. What is the most efficient method between these two extremes? The answer is that table scans will generally win if the query optimizer thinks that more than 20% of the data will be processed. Table scans over very large tables, however, can be very

time consuming and also use a lot of resources filtering through rows that aren't selected or are already deleted.

Enter the Encoded Vector Index. EVIs are specifically designed to fill the gap between traditional indexes and table scans. So why would this 'index' be any better than a traditional one?

The answer lies in how the EVI is used to find and process the rows associated with its keys. The unique data structure inside of an EVI allows DB2 to use the best aspects of indexes and the best aspects of table scans.

Traditional indexes work well because it is extremely fast to locate any given key even in a very large table. The secret behind this performance is that the structure allows the value to be found using a binary search. A binary search, for those of you that aren't familiar with the algorithm, cuts the search list in half every time it looks at the full list. If there are a million rows in a table, it will take less than 20 comparisons on average to find the value.

A table scan is efficient because it does not do small random disk I/Os and instead does fewer and larger requests to disk.

EVIs leverage the speed of the binary search and the advantages of a table scan. When DB2 uses an EVI, it first finds the first and last rows in the table [where?] it will find selected rows. Next it does intelligent disk I/O to both leverage large I/Os and also to skip doing any I/O where there are not going to be selected rows.

Guidelines

So when does it make sense to start experimenting with EVIs? As with all performance recommendations, the answer is "it depends." There are, however, guidelines that you can use to at least know where you shouldn't use them and also situations where they are most likely to be of benefit.

Where NOT to use EVIs:
♦ On tables with relatively low query activity
♦ On fairly small tables
♦ On columns that are highly unique. An example might be a date/time column that will always have a new value being inserted
♦ On columns that are highly volatile. This is a 'softer' guideline but if the column has a large number of distinct values then this rule should be observed

Where EVI's are most likely to provide benefit:
♦ On tables that are large and used as the source of reports or ad hoc queries. Of particular interest are tables that have queries that need to process between 20 and 60% of the data. These tables can be identified by Centerfield's insure/ANALYSIS product
♦ On columns with a reasonable number of distinct values (typically less than approximately 64,000)
♦ On columns that have data skew – especially where the data skew is physically clustered
♦ On columns that are commonly ANDed or ORed together and the combination either results in 20-60% of the rows being selected or if there is correlation between the columns such that two or more indexes can narrow the selected rows down to a small number.

Further Reading

If you'd like to know more about encoded vector indexes, see the following web pages:

http://www-03.ibm.com/servers/enable/site/bi/teraplex/evi.html

http://www.intelligententerprise.com/db_area/archives/1999/992601/scalable.jhtml;jsessionid=PZKUK0OCKDMEUQSNDBECKHSCJUMEKJVN