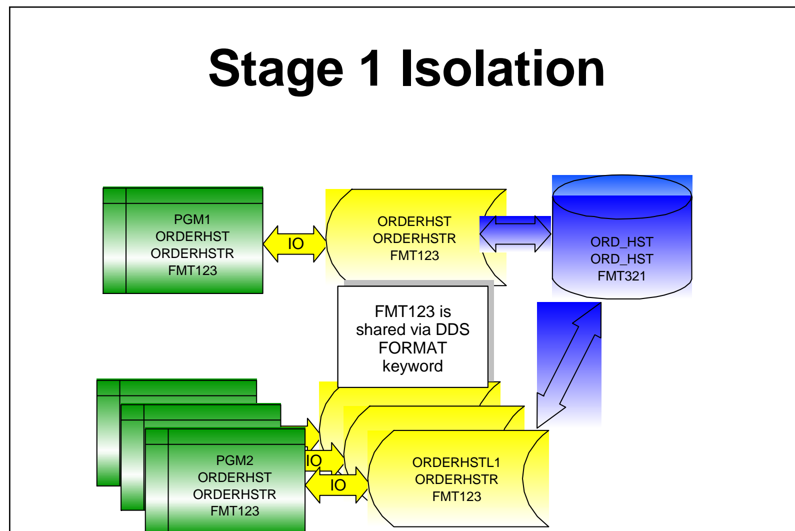
Understanding Access Plans and Open Data Paths

In my previous article I discussed architectural differences between SQL DDL and DDS created database objects. I also provided a strategy for reengineering existing DDS databases to allow existing High Level Language (HLL) programs to take advantage of the SQL enhancements. I refer to this strategy as Stage 1 of my database reengineering and application program transformation methodology.

In this article I begin to look at what I refer to as Stage 2 or Database Isolation. Actually, Stage 1 is the first step towards database isolation. In essence, all DDS PF source members and objects were converted to DDS LF source members and objects. Existing DDS LF source was modified to share SQL 64K index access paths and, if necessary, share the format of the transformed original physical file. This is shown in the figure on the right.



Stage 2 environments are characterized by the separation of the database IO function from the application business and presentation processes. This has gone by several names (n-tier application development, Model-View-Controller, etc). I believe that for best results when utilizing SQL that you begin with an SQL created database (Stage 1) and that all database access be done via SQL Data Manipulation Language (DML) statements embedded within HLL IO modules (Stage 2).

In this article I am going to discuss some other significant architectural changes in how a Record Level Access (RLA) HLL program and a HLL program containing static embedded SQL are activated and subsequently executed. In particular, I will compare the following a) RLA record format checking versus SQL access plan validation and b) Open Data Path (ODP) reusability.

In writing this article I will follow the structure of the Performance Mystery series. First I will discuss an actual customer situation comparing before and after results. Next, I will go into details about the solution to the customer problem. I will then close with some considerations and recommendations.

It is my belief that once an application developer understands the basic mechanics of program initiation, execution and termination then the application developer will be able to write better, more efficient code the first time, thus avoiding an expensive visit by yours truly. Sometimes these visits are not very pretty.

Background

ABC Corporation has continually benefited financially from steady growth. This has resulted in a need to acquire more personnel, inside and outside the IT organization. Unable to attract skilled individuals, due to the less than modern look of the existing green screen oriented applications and the lack of RPG trained college graduates, ABC Corporation took on an application modernization project. This consisted mainly of replacing the front-end green screens with Web browser based interfaces and separating the RPG IO access methods into separate IO modules.

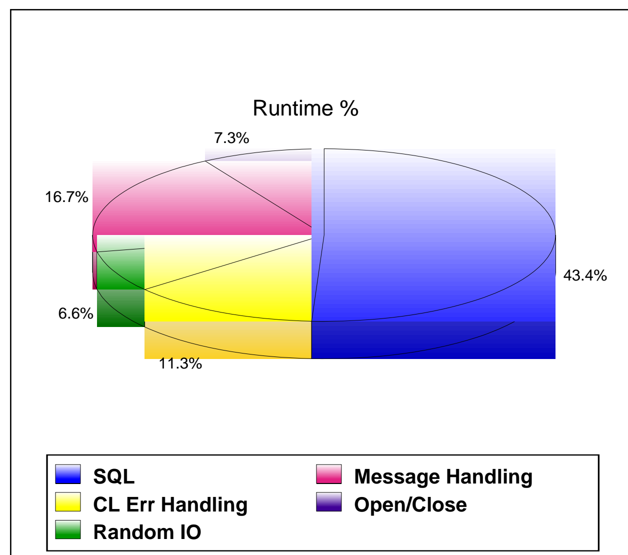
The initial project was successful and based upon favorable feedback from the end user community a decision was made to go forward and convert all applications. The application developers were given permission to choose their access method of choice when creating the IO modules. As a result, older legacy based programmers chose RPG RLA access methods and new hires chose SQL.

Interactive response times were tolerable, mainly sub-second irregardless of the data access method; however the overnight batch process began to suffer. As the workload increased over time, the window for completing the overnight batch continued to shrink. Eventually, the batch work began to intrude on daytime productivity causing deep concern amongst corporate executives.

Initial investigation via the iDoctor Job Watcher tool revealed a substantial amount of time being spent in the SQL IO modules versus the RPG IO modules. This caused a division between the programming groups and before a civil war erupted, the CIO of ABC Corporation contacted IBM for help. A short time later yours truly showed up at the customer location.

The first step in the performance investigation was to create a test environment where the batch jobs could be analyzed in more detail. The Performance Explorer (PEX) data collector was used during testing to identify the most costly programs within the jobs and the major application performance related issues.

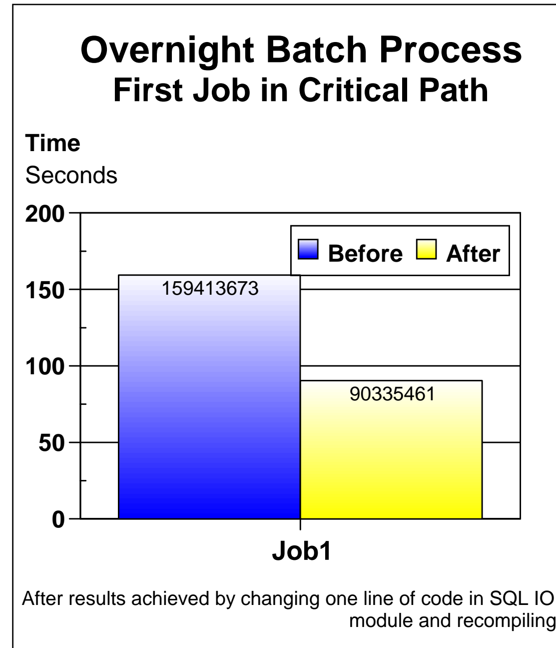
The chart on the right represents a summary of the top i5/OS system modules accounting for the most elapsed clock time during the execution of the test run. The high cost of the modules associated with SQL (optimization and execution) validated the iDoctor Job Watcher



findings; in essence a majority of time was being spent within the SQL functions.

The next step was to identify the program, or programs, which were executing the most SQL functions. As it turned out, in this step of the nightly process there was only 1 program with embedded SQL. Review of the program revealed a less than optimal pre-compiler option being used. The option was changed, the program was recompiled and another test conducted.

The graph on the right shows the before and after cumulative run time results of the test runs performed during the onsite visit. The huge reduction in run time was the result of simply recompiling the RPG SQL program with the proper compiler options. In fact, the RPG programs using RLA are now at the top of the list accounting for the most direct clock time.



Before I tell you what the proper compiler option was, I think it is important to understand some basic concepts that apply to both RLA programs and programs containing static embedded SQL. These concepts involve the use of Access Plans and Open Data Paths (ODPs). Although the concepts are the same the implementation is quite different.

The Access Plan

To understand the differences in the implementation of RLA and SQL programs I am going to require the assistance of two RPG programs. Although extremely simple these programs are similar to the IO modules being developed by ABC Corporation as part of their application modernization strategy.

The RLA Plan

Before creating the IO module the programmer analyst needed to gather information regarding the functional use of this IO module. This was done via a question and answer session that included the following types of questions:

1. What is the requirement? Answer: Retrieve the employee last name from the employee table.
2. Where will this be done? Answer: mainly interactive environments
3. How often will this be done? Answer: as often as needed.

Based on the answers to the above questions the programmer analyst came up with the following plan to implement the new IO module:

1. Find an existing DDS logical file keyed on employee number that contains the employee last name as part of the format
2. If an existing keyed logical file does not exist then create a new one.
3. Code and compile
4. Test the program (multiple executions).

The following is the DDS of the EMPLOYEE keyed logical file that the programmer analyst decided to use for this IO module:

```
A*************************************************
A* Entity: EMPLOYEE
A* Purpose: "Surrogate Logical file for format sharing. This was
A*           the original physical file prior to conversion.
A*************************************************
A          R EMPLOYEER                    PFILE(EMP_TABLE)
A*
A            EMPNO         6
A            FIRSTNME     12
A            MIDINIT       1
A            LASTNAME     15
A            WORKDEPT      3
A            PHONENO       4
A            HIREDATE      8S 0
A            JOB           8
A            EDLEVEL       4S 0
A            SEX           1A
A            BIRTHDATE     8P 0
A            SALARY        9P 2
A            BONUS         9P 2
A            COMM          9P 2
A*
A          K EMPNO
```

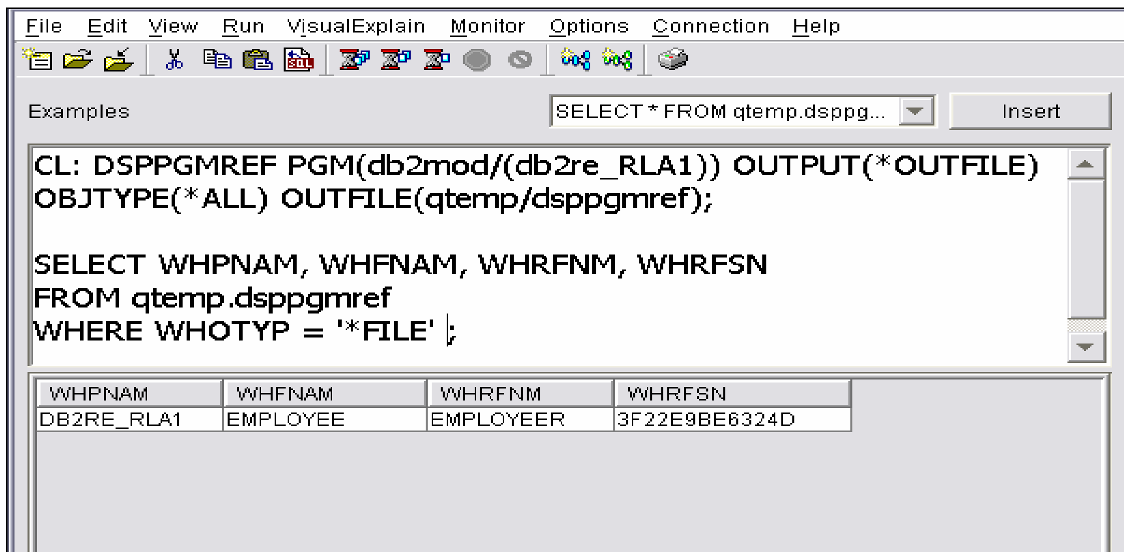
Notice that the above keyed logical is a "surrogate" file that was converted as part of the DDS to DDL conversion methodology discussed in my previous article. I refer to this as a Stage 1 DDS LF.

The following is a code snippet of the source program coded to access the keyed logical file and return the employee last name.

```
DB2RE_RLA1 RPG RLA Example code snippet

     FEMPLOYEE  IF   E           K DISK     Use this keyed logical file

     DEmployee_Nbr     S                   LIKE(EMPNO)
     DEmp_Last_Name    S                   LIKE(LASTNAME)

     C      *entry        Plist
     C                    Parm                    Employee_Nbr
     C                    Parm                    Emp_Last_Name

      /FREE
       Chain Employee_Nbr EMPLOYEE;   Random keyed read occurs here
       IF  %FOUND;
         // Record found code follows here\
       ELSE;
         // Record not found code follows here
       ENDIF;
       *INLR = *ON;
       Return;
```

During the compilation of the above program the Record Format Level Identifier of the EMPLOYEE keyed logical file is recorded as part of the program object. During program activation the format ID within the program is compared to the format ID of the database object. If they are different the program will fail with a "level check" error. The program must then be recompiled or, heaven forbid, an override must be issued to ignore level check errors.

You can view the Format Level Identifier within the program by using the DSPPGMREF command. The following is an example of doing this via the iSeries Navigator Run SQL Script window (note: the DSPPGMREF command is included in the list of examples. Simply click on the down arrow, scroll through the examples until you find the DSPPGMREF command, highlight it and then insert it into the script window):



Because the DDS LF is at Stage 1 the associated base table EMP_TABLE can be altered, either by adding new columns or changing existing columns, without the EMPLOYEE record format ID changing. This means that all programs using this DDS LF file do not need to be recompiled. They will execute without a level check error.

The programmer analyst tested the program by calling it 3 times with different employee numbers each time. The program executed successfully and each execution took approximately 1.8 milliseconds (based on the configuration of the development iSeries system). This seemed very acceptable and the program was put into production.

The SQL Plan

On the other side of the house, another team of programmers was developing new applications and they also had a requirement to retrieve the employee name from the employee table. However this group was developing IO modules that would be used as external stored procedures (i.e. a HLL program that is called via an SQL CALL statement and returns data in the form of result sets or parameters).

The SQL database programmer also created an action plan similar to the one created by the RLA programmer analyst with some exceptions. The plan looked something like this:

1. Code and compile the HLL program
2. ***Review the SQL access plan***
3. ***Create an index if advised by DB2***
4. Test the program (multiple executions) reviewing the access plan after each execution.

Notice that in the above plan the SQL database programmer is no longer concerned about locating an existing index (keyed logical file), this is the responsibility of the DB2 optimizer. However, the DB2 database programmer does need to review the choices the optimizer makes and react to any suggestions DB2 may provide (steps 2 and 3).

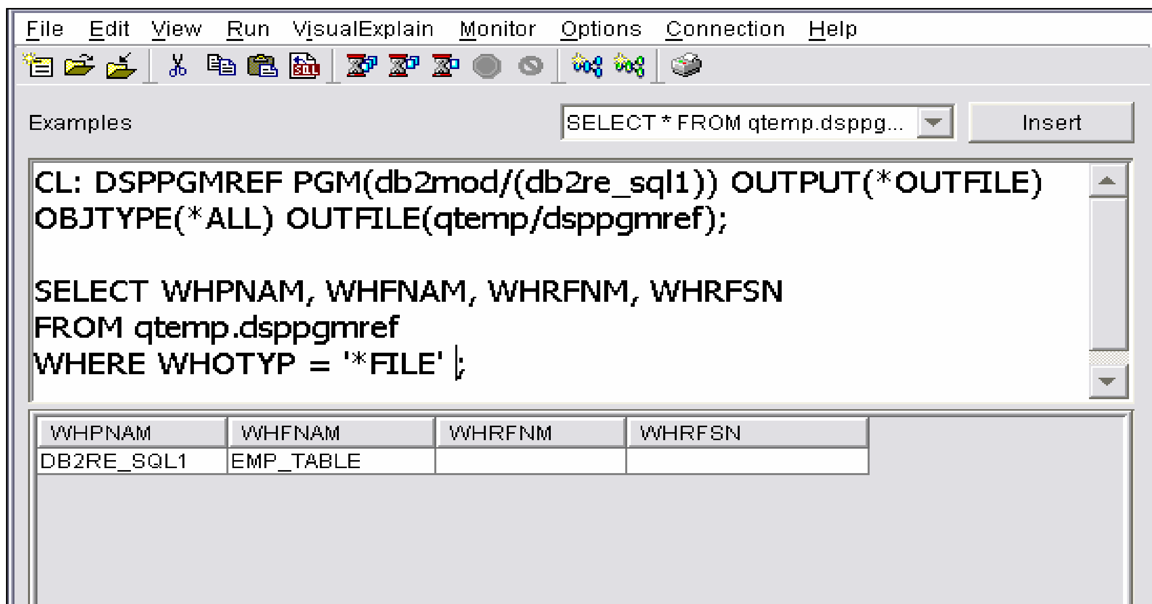
The following is a code snippet for the RPG program using embedded SQL:

```
RPG SQL Example code snippet

     DEMPLOYEER      E DS                     EXTNAME(EMPLOYEE)

     DEmployee_Nbr     S                      LIKE(EMPNO)
     DEmp_Last_Name    S                      LIKE(LASTNAME)

     C     *entry        Plist
     C                   Parm                       Employee_Nbr

     C/EXEC SQL
     C+ SELECT LASTNAME INTO :EMP_LAST_NAME
     C+ FROM EMP_TABLE
     C+ WHERE EMPNO = :Employee_Nbr
     C/END-EXEC
```

```
/FREE
 IF SQLCOD = 100 ;
     // Record found code follows here
 ELSE;
   // Record not found code follows here
 ENDIF;
 *INLR = *ON;
 Return;
```

Programs compiled using embedded SQL do not contain record format IDs. During run time the plan associated with the embedded SQL statement is validated against the database. If the plan is still valid (even though the database may have been altered) the program will execute. In essence, no level check error occurs.

A DSPPGMREF of a program object which contains embedded SQL will show the database object referenced in the SQL statement however there is no associated format ID as shown in the following figure:



When a program containing static embedded SQL (as opposed to dynamic embedded SQL) is compiled, an incomplete plan is generated and stored with the program object. The plan is not fully optimized until the first execution of the program. One reason for doing this is that the environment in which a program is compiled may not be the same as the environment in which the program will be executed (e.g. interactive versus batch).

The iSeries Navigator Explain SQL function (or the Print SQL Information (PRTSQLINF) i5/OS command) can be used to display the SQL access plan generated for a program containing embedded SQL statements.

The following is the SQL access plan generated for the RPG SQL program immediately after compilation:

Explain SQL for DB2MOD.DB...

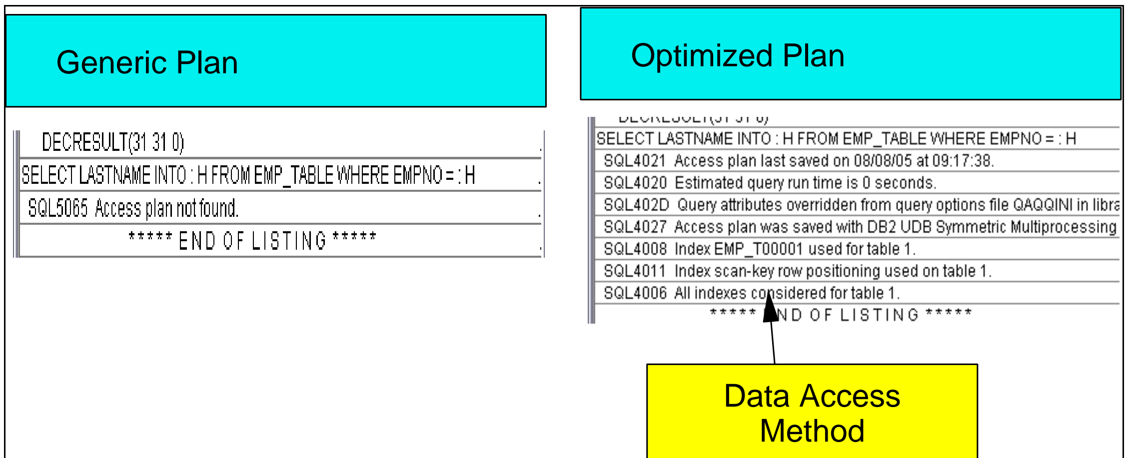
```
C1
5722SS1 V5R3M0 040528     Print SQL information          Program DB2MOD/DB2R.
Object name..............DB2MOD/DB2RE_SQL1
Object type..............*PGM
  CRTSQLRPGI
     OBJ(QTEMP/DB2RE_SQL1)
     SRCFILE(DB2MOD/QRPGLESRC)
     SRCMBR(DB2RE_SQL1)
     COMMIT(*CHG)
     OPTION(*SYS *PERIOD)
     TGTRLS(V5R3M0)
     ALWCPYDTA(*OPTIMIZE)
     CLOSQLCSR(*ENDACTGRP)
     RDB(*LOCAL)
     DATFMT(*MDY)
     DATSEP('/')
     TIMFMT(*HMS)
     TIMSEP(':')
     DFTRDBCOL(*NONE)
     DYNDFTCOL(*NO)
     SQLPKG(DB2MOD/DB2RE_SQL1)
     ALWBLK(*ALLREAD)
     DLYPRP(*NO)
     DYNUSRPRF(*USER)
     SRTSEQ(*HEX)
     LANGID(ENU)
     RDBCNNMTH(*DUW)
     STATEMENT TEXT CCSID(37)
     SQLPATH(*LIBL)
     DECRESULT(31 31 0)
SELECT LASTNAME INTO : H FROM EMP_TABLE WHERE EMPNO = : H
  SQL5065  Access plan not found.
                    ***** E N D  O F  L I S T I N G *****
```

Compiler Options

SQL Statement and associated plan

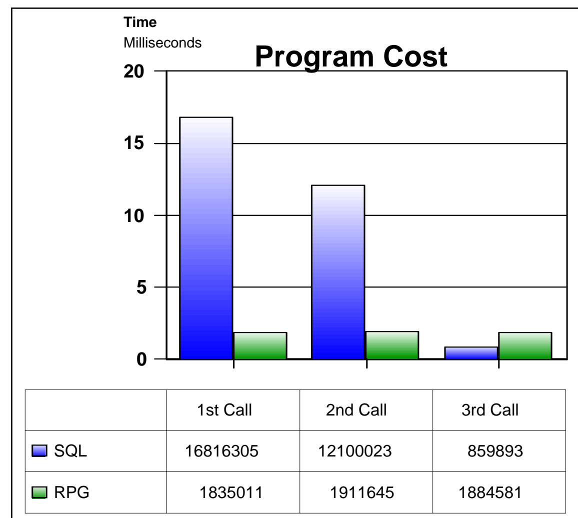
The Explain panel contains the compiler settings used during the pre-compilation stage of the RPG program. There are several settings which have a direct impact on performance and should be reviewed as part of the coding review process. These include ALWCPYDTA, CLOSQLCSR and ALWBLK. We will discuss the CLOSQLCSR option in more detail later in this article.

Each SQL statement within the program is then listed along with its associated access plan. In the above figure a plan does not exist for the SQL statement. The plan will not be generated until the first execution of the RPG program. The following figure compares the plans after the first execution of the RPG program:



The right side of the above figure now contains the optimized plan for the SQL statement embedded within the RPG program. Information about the plan is stored as text messages which explain the environment the program was executed in (e.g. DB2 SMP is active, QAQQINI file is being used, etc.) and the data access methods used to produce the result set.

Based on the optimized explain we can see that DB2 reviewed all appropriate indexes for physical file EMP_TABLE (message SQL4006), chose an index with a system name of EMP_T00001 (message SQL4008) and will use that index to perform key row positioning (a.k.a index probe) to access the database.



	1st Call	2nd Call	3rd Call
SQL	16816305	12100023	859893
RPG	1835011	1911645	1884581

The database programmer tested the SQL RPG program by executing it 3 times and noticed that the third execution was faster than the first 2 executions. The SQL programmer called is RPG buddy and asked to compare results of both their tests. This is displayed in the chart on the right. In essence the RPG RLA program was consistently executing under 2 ms. The first execution of the RPG SQL program was nearly 10 times the cost of the RPG RLA. However, the third execution of the RPG SQL program was less than half the time of the RPG RLA program (.8 ms versus 1.8 ms). Both programs are using the Last Record (*INLR) indicator. What is going on? The answer is ***reusable ODP***.

The Open Data Path (ODP)

The Open Data Path or ODP is a temporary object that contains point in time information about the file being accessed on behalf of your HLL or SQL request. This information

changes as the ODP is being used. Every job (or session) has its own set of ODPs. The major expense is incurred when the ODP is created, not when it is updated.

Contrary to popular belief, there is run time overhead whenever a program is called. Every program must be activated and storage allocated, existing access methods must be validated against the database (this is done for both RLA and SQL) and finally the ODP must be created. If a program is non-reentrant (e.g. *INLR is turned on prior to returning from an RPG program) then it is deactivated when it returns to the calling program. Program deactivation results in the deletion of any ODP created for that program and the subsequent release of internal storage. The next time the program is called this process is repeated. This is an extremely inefficient use of system resources and can result in longer run times and reductions in overall system throughput.

You may have noticed in the above program examples that the (*INLR) is being turned on prior to returning from this program. The programming staff at ABC Corporation has made a conscious decision to use *INLR even though they have been told about the performance overhead. In essence, these programmers have found that if they execute an Override Database File (OVRDBF) command to switch from one library or database member to another, they need to ensure that the file is closed, otherwise the ODP remains connected to the previous file or member. The use of the *INLR indictor ensures the file will be closed. In addition, the programmer doesn't have to worry about repositioning file cursors or re-initializing program variables.

The normal behavior for SQL (whether it is embedded static, embedded dynamic, dynamic or extended dynamic) is to attempt to reuse an existing ODP after the second execution of the same SQL statement within the same job. Once an ODP becomes reusable the plan associated with the SQL statement is no longer validated as part of program execution. This is desirable within most high volume transaction processing environments.

The box on the right contains a joblog snippet taken during the testing of the RPG SQL program running in debug mode. Informational message SQL7912 is sent when a "full open" occurs, that is, when an ODP is created from scratch. Informational message SQL7913 is sent when the ODP is deleted, a.k.a "hard close". Information message SQL7914 is sent when the ODP is not deleted. The next execution of the program results in ODP reuse (message SQL7911).

```
▪ First Execution
  SQL7912   Message . . . . :   ODP created.
  SQL7913   Message . . . . :   ODP deleted.
▪ Second Execution
  SQL7912   Message . . . . :   ODP created.
  SQL7914   Message . . . . :   ODP not deleted.
▪ Third Execution
  SQL7911   Message . . . . :   ODP reused.
  SQL7914   Message . . . . :   ODP not deleted.
```

This behavior can be circumvented. This is what happened at ABC Corporation. The SQL database programmer compiled the SQL IO module with the Close SQL Cursor (CLOSQLCSR) pre-compiler setting set to *ENDMOD. This caused the ODP to be deleted when control was returned to the calling program. The programmer was

concerned about the ODP not being recreated when a library list was changed or an OVRDBF command was issued and the ODP was already attached to a file and member (similar to the problem the RLA programmers had to contend with).

SQL does recognize these types of environmental changes after going into ODP reuse mode. The joblog snippet on the right shows the ODP being deleted on the next execution after a CHGCURLIB i5/OS command was issued. In this case, the program has been running and prior to the nth execution the library list was changed. DB2 recognizes that and deletes the existing ODP (message SQL7918). The second level text of the message contains text describing the reason for the deletion (code 5).

```
■ First Execution
   SQL7912    Message . . . . :   ODP created.
   SQL7913    Message . . . . :   ODP deleted.
■ Second Execution
   SQL7912    Message . . . . :   ODP created.
   SQL7914    Message . . . . :   ODP not deleted.
■ Third Execution
   SQL7911    Message . . . . :   ODP reused.
   SQL7914    Message . . . . :   ODP not deleted.

   CPC2198    Message . . . . :   Current library changed to DB2MOD.
■ Nth execution
   SQL7918    Message . . . . :   Reusable ODP deleted.  Reason code 5.
   SQL7912    Message . . . . :   ODP created.
```

In this case DB2 found multiple instances of EMP_TABLE in multiple libraries (schemas) thus forcing the ODP deletion. This on demand implementation of ODP deletion is far less expensive then deleting the ODP after each use, especially if the library list never changes, or an OVRDBF is never issued, within every job using this program.

Once the programming staff understood these differences between SQL and RLA they simply recompiled the RPG SQL program, taking the recommended default of *ENDACTGRP for the CLOSQLCSR option. The result was a huge performance benefit measured in terms of decreased batch run times (as shown in the prior graphs).

Considerations

Although the reuse of ODPs may result in a need for additional memory, the non-reuse of ODPs is characterized by a high number of non-database faults. Many customers have thrown memory at this problem; however the fault rates never seem to diminish. Once the applications are changed the amount of existing memory tends to be sufficient.

Unlike RLA, there is no sharing of ODPs between SQL statements. Should program A and program B both contain the same SQL statement and both programs are called within the same job then each program will have a separate ODP. This form of duplicate ODPs can be resolved by isolating SQL statements into IO modules called, or shared, by both programs.

Since SQL does not share ODPs then there is no benefit to pre-opening files with the Open Database File (OPNDBF) command.

The normal behavior of SQL to reuse ODPs after second execution can be altered to first execution by creating a data area named QSQPSCLS1 in a library of your choice. DB2 checks for this data area within a job's library list at the beginning of each job. If found then all ODPs that are candidates for reuse will not be deleted after first execution. The data area contents, type, and length do not matter. The first transaction processed will pay the full open price, subsequent transactions will benefit by using the existing ODP.

Overriding members results in access plan revalidation and, possibly, the deletion and recreation of ODPs. There is no performance benefit by physically sub setting data on the iSeries/i5 platform via member support. In fact, you cannot create an SQL index or view over a member other than the first member within a physical file. Now is the time to consider redesigning databases with multiple members to a single database table.

Changing library lists results in access plan revalidation and, possibly, the deletion and recreation of ODPs for unqualified SQL statements running under system naming (library/file) versus SQL naming (schema.table). Revisit the business requirement for using this type of technique for isolating databases. Many organizations have separate development systems for compiling and testing, eliminating the need to change library lists. If data needs to be separated for security reasons (i.e. Company A data versus Company B data) then consider the use of Independent ASP (IASP) support. With this feature, multiple databases can exist on a single iSeries/i5 system with identical schema (library) names.

Separating IO into independent modules and deploying these modules across all databases can minimize the impact of access plan validation especially if qualification is used (specifying the schema and table). Today, each qualified SQL statement that is optimized by the new SQE engine can have up to 3 plans stored in the SQE plan cache.

Summary

For over 20 years the impact on system performance of repeated full opens and hard closes has always appeared as one of the top application related performance issues, sharing the spotlight with excessive random keyed access and inefficient blocking techniques. In many cases, the cost of repairing a poor performing program is more expensive (in terms of programming dollars) then simply upgrading the system.

SQL, out of the box, addresses these most common application related performance issues. In my previous article we learned that the more efficient SQL created access paths can be shared by DDS keyed logical files, thus reducing keyed access costs. In this article we see that SQL automatically attempts to reuse Open Data Paths thus reducing the high cost of file open and close operations.

We have yet to discuss the implicit blocking used by SQL, thus eliminating the need for programmers to manually calculate the n value of the SEQONLY(*YES n) parameter used in the OVRDBF command. Nor have we touched on SQL only performance enhancements such as Index Only Access, extended fetch (FETCH FOR n ROWS), blocked insert (INSERT INTO Table n ROWS), fast DELETE support, sharing result sets

across jobs, advanced join capabilities, Look ahead Predicate Generation (LPG), Materialized Query Tables (MQTs), etc. and the list goes on.

In summary consider the following:

1. Cost of optimizing an access plan on first execution: 17ms versus 1.8ms in a non-SQL program.
2. Cost of reusing an ODP: Less than 1ms versus 1.8ms in a non-reentrant RLA program.
3. Cost of not having to pay IBM for CPU on demand everytime the workload approaches 100% due to excessive file open and close activity or not having to come in on Sundays to recompile all my programs because of format ID changes: Priceless