

Modernizing Database Access The Madness Behind the Methods By Dan Cruikshank

Overview

The evolution of the iSeries/i5 has resulted in a mix of old and new technologies. Most notably are the methods used to create, populate and manipulate databases. Many iSeries/i5 application developers have embraced modern tools for creating front end masterpieces to their existing data; however the underlying foundation, the database itself, has been woefully neglected. Basically the existing databases were created using a tool known as DDS (Data Description Specification). All other Relational Database Management Systems (RDMS) use Structured Query Language, or SQL, to define the database.

For many iSeries/i5 developers initial attempts in the use of SQL have resulted in unacceptable performance causing system administrators and/or Chief Information Officers to discourage its use. For others, the ease of use of SQL has made it the preferred tool for data creation and access. When these two groups come together the result is like a bad beer commercial. Shouts of "It's great!" are hurled in response to yells of "Less fulfilling!"

As IBM continues to enhance DB2 UDB for iSeries (e.g. the new and improved SQL Query Engine, more efficient data access methods, new database primitives, etc) the decision to use SQL over traditional methods will no longer be if, but when. In fact, many iSeries/i5 shops are developing all new applications using SQL. In addition, the use of SQL defined databases may result in improved throughput as a result in changes to the underlying architecture (more on this later).

So all new SQL development will benefit from new database technology but what about the existing DDS defined databases that are still serving thousands of applications using Record Level Access (RLA) methods via High Level Language (HLL) read and write operations? Is there a way that these applications can take advantage of SQL database enhancements without a total rewrite?

The answer is a resounding Yes! This is the basic premise of this article. In essence this article provides a high level overview of the madness behind the methods known as the Stage 1 DDS to DDL reengineering strategy detailed in the soon to be released Redbook "*Modernizing iSeries Application Data Access - a Roadmap Cornerstone*".

Database Reengineering Strategy

The strategy described in this document assumes that you have an existing database that is fully described using DDS specifications. This strategy will work with partially described DDS databases (i.e. the DDS contains the key definitions and great big fields which are program described) however your first step would be to manually describe the physical files using SQL DDL.

The strategy is divided into 4 stages. The stages are:

1. DDS to SQL DDL Reverse Engineering
2. Isolating the database
3. Modifying the database
4. Enhancing the database

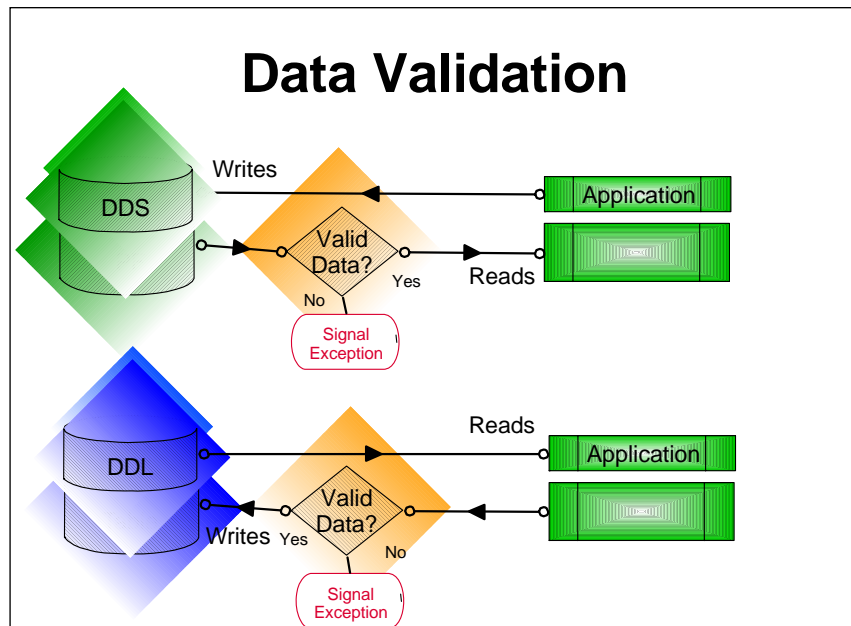
I have seen many application developers attempt to mix SQL and DDS with less than optimal results. I believe that by following this staged approach to database modernization many of these pitfalls will be avoided. In this article I will a) discuss the importance of Stage 1 as it relates to DB2 UDB for iSeries architectural differences when creating and accessing database objects and b) describe the process used to reengineer a DDS database.

Before discussing the process it is important to talk about some of the architectural differences between SQL and DDS which have not received much attention. Two very important differences are a) data validation and b) access path size.

SQL and DDS Data Validation Differences

The major difference between these two types of physical database objects is the process that determines when the data is validated. For DDS, the data is validated as data is read through the open cursor. For SQL, data is validated as it is written through the open cursor. This is shown in the picture on the right.

This subtle change has two significant impacts on database IO operations. First, it ensures the integrity of numeric (packed or zoned) data entered into the database. Those of us who grew up on this platform are painfully aware of this. We learned the hard way after receiving numerous “decimal data error” exception messages in our application programs. For those of you who are not aware of this issue simply perform the following test.



Start by creating a physical file object that is not externally described as follows:

```
CRTPF FILE(TESTDATA) RCDLEN(80)
```

Once created enter some alphanumeric data into this file, for example 'dan123ouch'.

Next create a DDS externally described physical file object name DDS_FILE using the following definition and command:

```
A          R DDS_FILE
A          FIELD1          9P 0
```

```
CRTPF FILE(DDS_FILE)
```

Perform the following copy file command to copy the alphanumeric data into the physical file object:

```
CPYF FROMFILE(*CURLIB/TESTDATA) TOFILE(*CURLIB/DDS_FILEPF)
MBROPT(*ADD) FMTOPT(*NOCHK)
```

Review of the joblog should show the following messages:

```
Data from file TESTDATA truncated to 5 characters.
1 records copied from member TESTDATA.
```

The joblog indicates that the data was inserted into the database. Using the DSPPFM command reveals that the invalid data does indeed exist within the physical file object.

```
Display Physical File Member
File . . . . . : DDS_FILEPF          Library . . . . . :
DCRANK
Member . . . . . : DDS_FILEPF          Record . . . . . : 1
Control . . . . .          Column . . . . . : 1
Find . . . . .
*...+
dan12
***** END OF DATA *****
```

Now create an SQL table identical to the DDS physical file created earlier as follows:

```
CREATE TABLE DDL_TABLE (FIELD1 DECIMAL(9,0) NOT NULL DEFAULT 0 );
```

Perform the same copy file operation however change the TOFILE parameter to use the new SQL table as follows:

```
CPYF FROMFILE(*CURLIB/TESTDATA) TOFILE(*CURLIB/DDL_TABLE)
MBROPT(*ADD) FMTOPT(*NOCHK)
```

Review of the joblog should reveal the following messages:

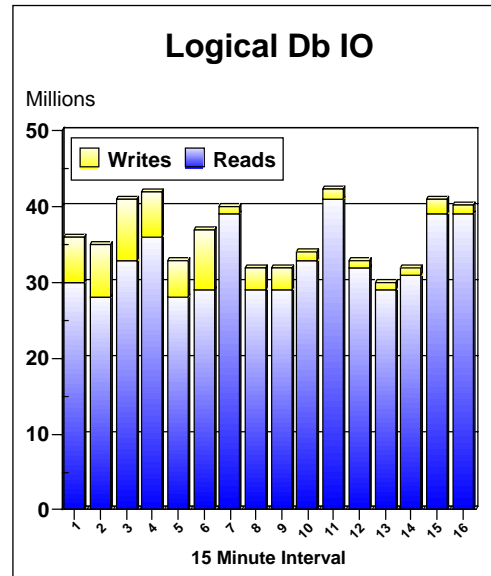
```

Data from file TESTDATA truncated to 5 characters.
Data mapping error on member DDL_TABLE.
Data mapping error on member DDL_TABLE.
C
Cancel reply received for message CPF5029.
Error writing to member DDL_TABLE in file DDL_TABLE.
0 records copied to DDL_TABLE.

```

Besides better data integrity the data validation change provides an additional improvement for probably 99% of all iSeries/i5 customers. Most applications running on the iSeries/i5 platform perform substantially more read operations to write operations. On average I have found this to be approximately 25 to 1.

The figure on the right represents a composite of many customer performance data collections during peak time periods. The graph clearly shows a huge disparity between reads and writes (Logical Database IO operations). These operations are performed on behalf of your HLL program read and write operations.



To determine the benefit of using DDL created physical files versus DDS created physical files I wrote a simple RPG program to perform random keyed operations against a table containing approximately 600000 rows.

The following is a code snippet for the program which I named DB2RE_RPG2:

```

DB2RE_RPG2
  FOrderHstL2IF      E                K DISK

  DSearchKey          S                LIKE(PARTKEY)

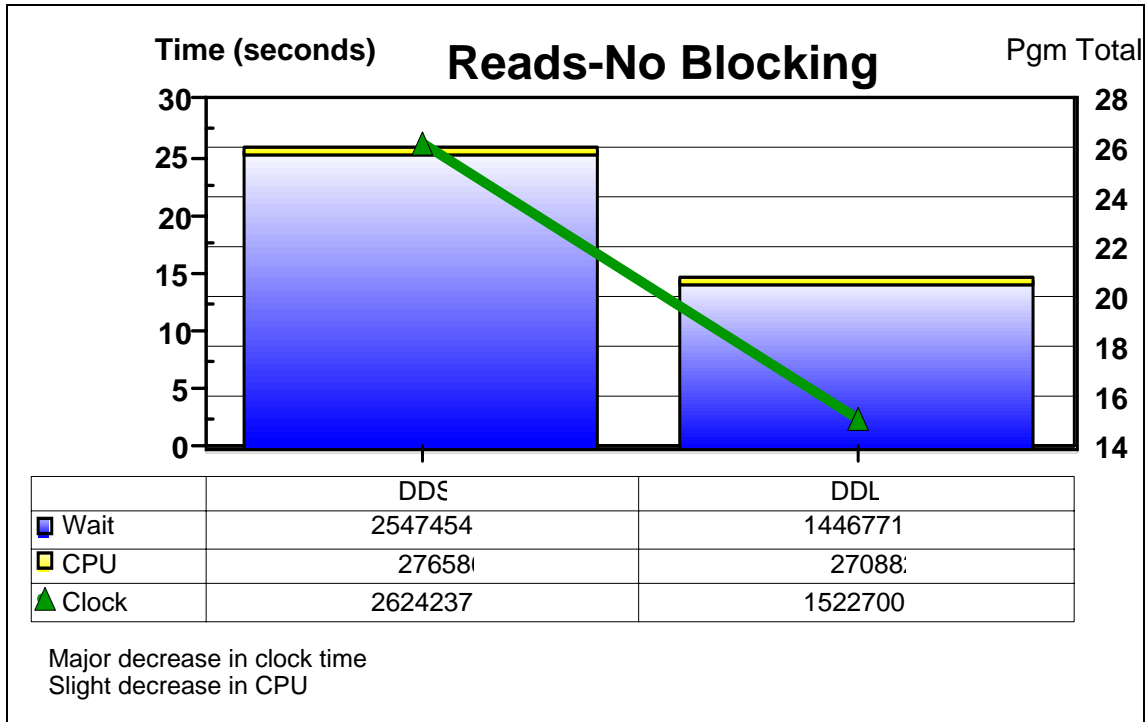
  C      *entry          Plist
  C                                Parm          SearchKey
  /FREE
  Setll SearchKey OrderHstR2;
  For I = 1 To 10;
    READE SearchKey OrderHstR2;
    If %EOF;
      Leave;
    Endif;
  EndFor;

```

The program is meant to simulate an application which positions to a specific key entry and then reads the first 10 matching entries, perhaps to fill a subfile page. I then called this program 802 times using random part numbers. I ran the test several times, purging the database objects from memory before each run. I then repeated the process after reengineering the OrderHst DDS PF to an SQL Table named Ord_Hst. After loading the table I created logical file OrderHstL2 over the

new SQL table. Note that my SQL table has a different name than my DDS PF. More about this as we discuss reengineering.

The following graph displays the results of the tests.



The blue bar represents the time the Set Cursor MI instruction spent waiting for a work request to complete. This is known as a Type B wait. To learn more about wait time watch for the new Redbook “*IBM iDoctor for iSeries Job Watcher: Advanced Performance Tool*”. The number of synchronous IO operations (i.e. bringing data from auxiliary storage to memory) are nearly identical between the DDS and DDL runs (7027 versus 7018 respectively). The data coming from the DDL tables took approximately 11 seconds less time to arrive. This represents about a 43% decrease in clock time.

I made some slight modifications to the RPG program in order to do some updating. The following code snippet represents the new program DB2RE_RPG3:

```

DB2RE_RPG3
  FOrderHstL2UF    E          K DISK
  DSearchKey      S          LIKE(PARTKEY)
  C      *entry    Plist
  C                          Parm          SearchKey
  /FREE
  Setll SearchKey OrderHstR2;
  If %Found;
  DOU %EOF;
    READE SearchKey OrderHstR2;
  If %EOF;
    Leave;
  Endif;

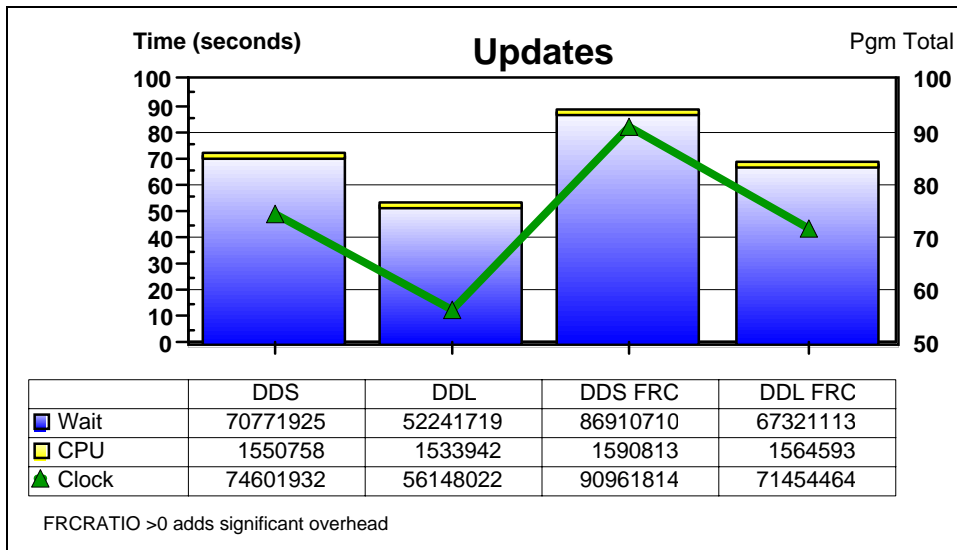
```

```

UPDATE OrderHstR2 %Fields(QUANTITY);
EndDo;
EndIf;

```

I then tested the new program the same way, randomly accessing the keyed logical file. In addition, I ran a second set of tests forcing all changes to disk (FRCRATIO = 1). The following graph shows the results of these tests:



As you can see, forcing the changes to disk adds significant amounts of overhead. This is due to a) the increased number of synchronous disk IO operations (both DDS and DDL) and b) the additional validation required for the DDL tables. This use of FRCRATIO is something that I run into quite often with legacy databases. I do not consider the use of FRCRATIO a modern technique for protecting data. For more information about modern techniques for protecting your data consider the following Redbook: *“Striving for Optimal Journal Performance on DB2 Universal Database for iSeries.”*

I was surprised by the reduction in clock time in the second DDL test. I actually spent time with the SLIC developers to find the reason for this reduction (which was actually a reduction in synchronous IO). As it turns out there is another performance enhancement known as concurrent write support.

Concurrent write support is available to those physical files that are created with the REUSEDLT parameter setting set to *YES. This is the default generated by an SQL CREATE TABLE statement. The default for the CRTPF command is *NO. Concurrent write support is the normal behavior for V5R3 (i5/OS). I was testing on a V5R2 system where this support had been turned on via the QDBENCWT API. To find out more about this support refer to the journal Redbook mentioned earlier.

I created another version of my RPG program to test the inserts against DDS or DDL created physical files. I created a DDL and DDS version of an object named OrderWrk. This is a non-keyed physical file.

The following is the new version of the program:

```

DB2RE RPG4

```

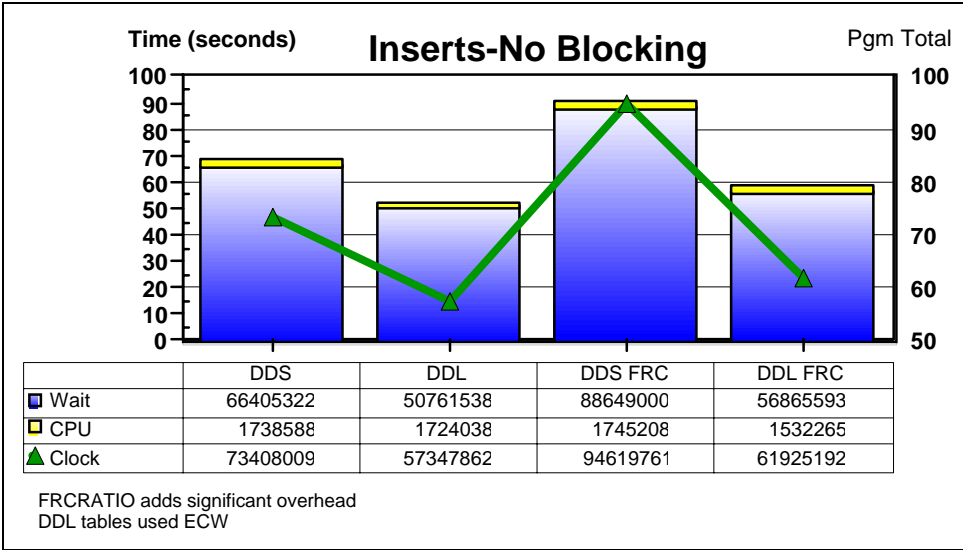
```

FOrderHstL2UF      E          K DISK
FOrderWrk  O      E          DISK
DSearchKey          S          LIKE(PARTKEY)
C      *entry      Plist
C          Parm          SearchKey
/FREE
Setll SearchKey OrderHstr2;
If %Found;
DOU %EOF;
  READE SearchKey OrderHstr2;
  If %EOF;
    Leave;
  Endif;

WRITE OrderWrkR;
EndDo;
EndIf;

```

I tested this program using the same methodology for the update program, with and without the FRCRATIO setting. The next chart contains the results of those tests:



I used the SEQONLY(*NO) parameter of the OVRDBF command to turn off blocking. Although blocking is a performance enhancement many organizations I work with are not utilizing blocking to its full potential, intentionally or unintentionally. Again we see the benefits of using a DDL database with concurrent write support enabled. The added overhead of data validation on inserts is not a factor.

Conclusions

The process of bringing data from auxiliary storage to main memory is significantly improved when the physical container has been created using SQL DDL. This is the default when creating database objects via iSeries Navigator.

Applications using physical files created via the CRTPF command with a FRCRATIO value greater than 0 may see significant improvements after converting to SQL tables. The default for FRCRATIO is *NONE when creating a physical file object via the CREATE TABLE DDL statement. If you want to continue to use a FRCRATIO setting greater than 1 you will need to either use the CHGPF command to permanently change the physical file or use the OVRDBF command to temporarily change the physical file attributes.

Again the use of FRCRATIO is not a modern method and its use is strongly discouraged. Many organizations that I have worked with were not aware that this method was being used or were using it inappropriately. If you discover that it is being used be sure to find out why and then explore more modern alternatives.

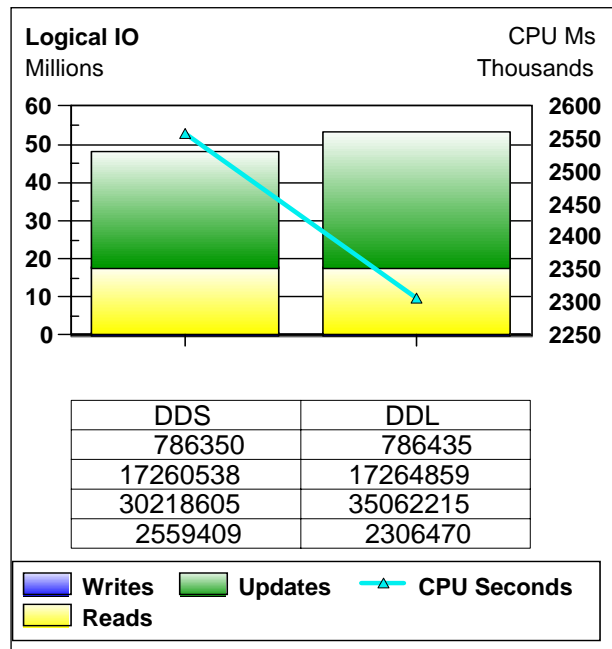
For many organizations, significant improvements will be achieved via concurrent write support in combination with the reuse of deleted records. Many databases created via DDS and CRTPF are using the default value of *NO for the REUSEDLT parameter because it is the default. For these organizations changing to REUSEDLT(*YES) may improve performance. However, I suggest going all the way and creating a modern SQL table.

If you are aware of some application dependency that requires the non-reuse of deleted records (e.g. RRN processing or End of File Delay) then I would strongly recommend that you put this application at the top of your list for rewrite in Stage 2. Again, I do not consider these types of applications to be modern.

If you are still on the fence about switching to DDL then let me throw out some actual customer results after migrating to an SQL created database. The results are shown in the graph on the right which is taken from system performance data. It represents a 27 minute period when approximately 60 jobs were processing transactions via data queues. All database access is done via HLL reads and writes.

The data shows a near 10% reduction in the cost of CPU (2559.4 seconds versus 2306.4 seconds) while processing a half million more update operations (30218605 versus 35062215).

This application could represent any of the following types of business functions where the iSeries is the platform of choice: Web based catalog orders, slot machine transactions, EDI activity, travel reservations, stock trades, ATM transactions, etc.



Here's some food for thought, this performance data may have come from your closest competitor.

SQL and DDS Access Path Size Differences

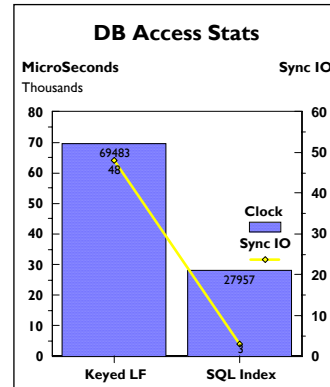
Assume you are driving to work taking the same highway (or access path) that you always take. And you have been doing this for a number of years. But lately you've noticed that it seems to take longer and longer. In fact there seems to be more and more commuters like yourself, alone in your HLL sedan, stuck in traffic. There are several sedans just like yours in front of you, and several behind you, all on the same access path. It also seems like the others to the right of you, on a different access path, never seem to move at all.

Suddenly you see a blur to your left and before you can get a good look at whatever it was, it's already gone. Just like that there's another one and then another. Finally you determine that these are those new SQL SUV's using the HOV lane because they have more than 1 occupant. You shake your head and say to yourself. "Man, I need to get me into some of that."

I used the commuter analogy to describe a major difference between SQL created indexes and DDS keyed logical files. The difference is the size of the access path associated with an index object. The access path represents the order in which data will be retrieved from a data space (physical file). Since V4R2 an access path created via SQL, whether it be an index or constraint, has a logical page size of 64K. A DDS keyed logical file will create, on average, an 8K access path up to a maximum size of 32K.

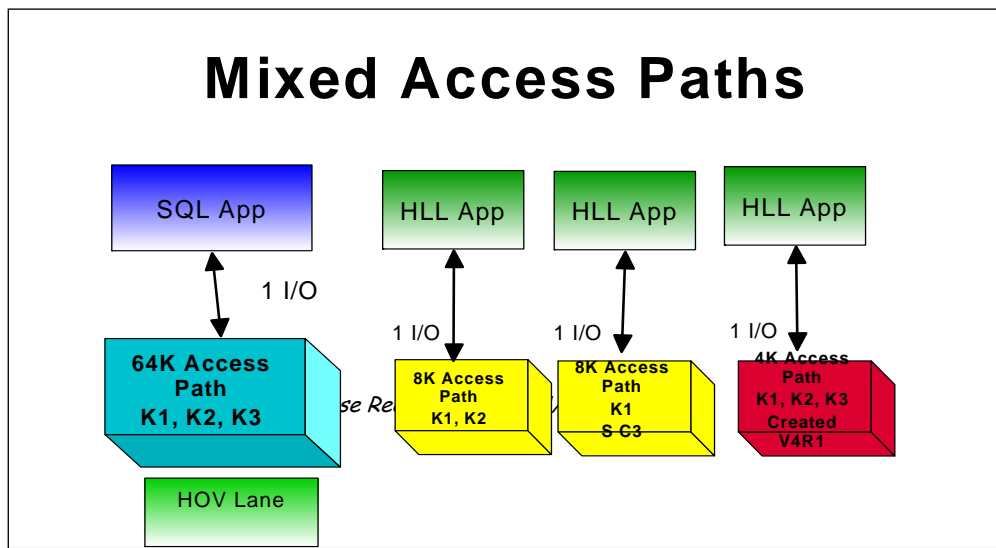
This change is due in part to the aggressive index usage associated with SQL operations. For example, an SQL implementation of a single SQL statement could result in an index probe (like a SETLL in RPG) followed by several index scan operations (similar to an RPG READE).

The figure on the right shows the difference between this SQL statement `SELECT COUNT(*) FROM PART_DIM WHERE PART LIKE 'B%' OR PART LIKE 'C%'` using a keyed logical file versus an SQL index. In both tests index only access was used for implementation of the SQL statement.



So you may be saying to yourself, "If the 64K access path helps SQL to do index probes and scans wouldn't that same access path benefit my HLL keyed operations?" The answer is yes and it comes in the form of a shared access path kind of like "Park and ride" still using the commuter analogy.

The following picture depicts an iSeries environment that contains a mix of SQL and DDS applications:



Each HLL application uses a different access path due to various reasons. For example the red access path is very old and has not required recreation in some time. The right most yellow access path is using select/omit logic and the left access path was created before the SQL index.

Basically, the SQL application is allowed to use the HOV lane because it may contain 2 or more occupants (sets of data). The HLL applications use the slower lanes because they contain 1 occupant (1 record). However, in the real world the most critical HLL application may be performing tens if not hundreds of record accesses to complete a single transaction. In fact, it is generally these high volume transactions that represent 80% or more of an organizations computer resource requirement (CPU and IO). In addition, out of the hundreds or thousands of application programs that exist in an organizations software portfolio, there may be only a few that account for a majority of CPU and IO time.

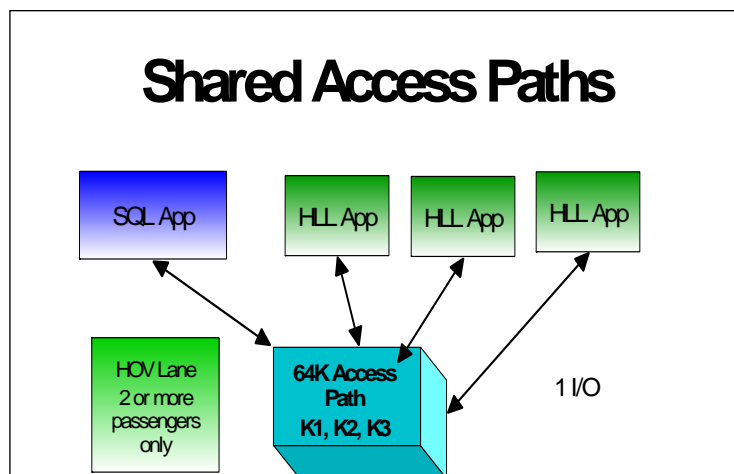
In order for the HLL applications to use the 64K access path we need to take advantage of the built in access path sharing support of OS/400 (i5/OS). This support is not new. On the System/38 you needed to specify the REFACCPATH file level keyword within your DDS source for the keyed logical file you were creating. Those of us who were aware of this keyword used it to minimize access path maintenance.

Today access path sharing is now performed implicitly by the operating system. However there are some rules that you need to be aware of. These rules are:

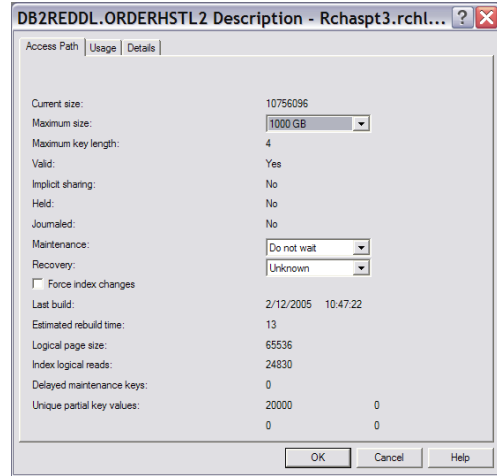
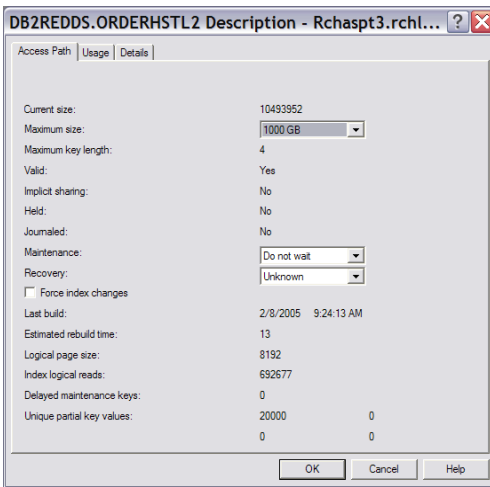
1. An SQL index will not share a DDS keyed logical file's access path
2. A DDS keyed logical file will share an access path created via SQL
3. A DDS keyed logical file will share an access path if the keys are a subset of the key fields in the existing access path and in the same order. For example LF1 with keys K1 and K2 will share an access path that is keyed on K1, K2 and K3.
4. An SQL index will only share another SQL created access path if the key columns are an exact match in the same order.
5. A DDS multi-formatted logical file will not share existing access paths.
6. A DDS join logical file can share existing access paths
7. The DYNLSLT file level keyword can be used to cause DDS logical files containing select/omit criteria to share an existing access path

So based on the above rules, if we added the DYNLSLT keyword to the DDS source representing the second logical file, then recreated our logical files in order by most keyed columns to last keyed columns we should end up with one access path shared as shown in the picture on the right:

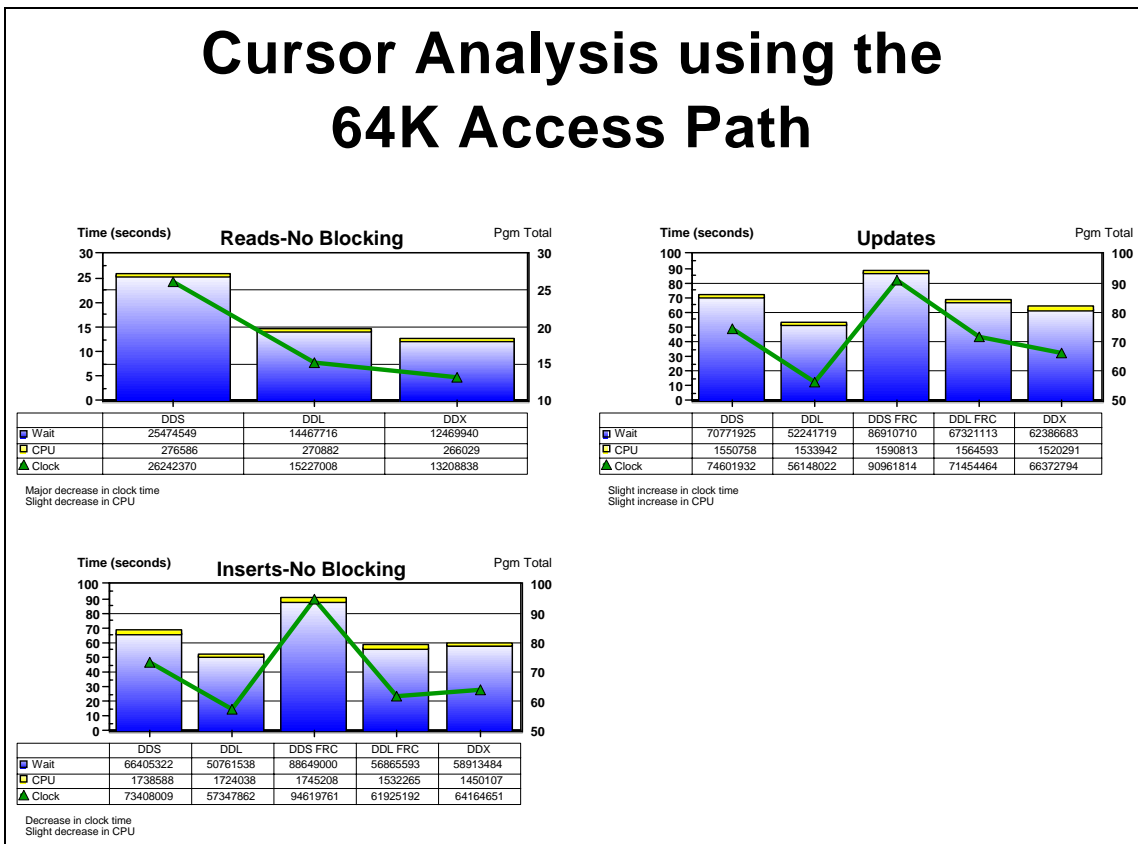
I built an SQL index over the ORD_HST table with the same key column as keyed logical file OrderHstL2 (the file used in my RPG programs). I then recreated OrderHstL2 over the ORD_HST table.



The following are screen prints from iSeries Navigator showing the two logical files. Notice the difference in logical page size, 8192 versus 65536. This means that the keyed logical file on the right is sharing an SQL created 64K access path.



I then ran the same tests from the previous section now using the logical file with the 64K access path. The following graphs contain the results:



Based on the data the use of the 64K access path further reduced overall clock time during the Read and Update tests. The Insert test took more time then the previous test using an 8K access

path over a DDL table but less time than using the same size access path over a DDS created physical file.

Conclusions

Those organizations that are currently running in environments that contain a mix of SQL created indexes and DDS keyed logical files may see improved performance by sharing the SQL access paths where appropriate. Sharing the SQL access paths will result in less access path maintenance.

Those organizations that have a large number of keyed logical files may see improved performance as a result of recreating the logical files after comparable SQL indexes have been created. This is due to the possibility that some logical files may have been created out of order (e.g. access path K1, K2 being created after access path K1 was created). Creating the access paths in order by most key columns first may result in fewer access paths. In addition, the first application to read in an index page will benefit other applications that need to reference the same index page.

Those organizations that have many older keyed logical files that are still defined using an access path size of *MAX4GB may see better performance in the form of fewer access path seize conflicts during index maintenance. This is due to the concurrent maintenance capability that can be employed when access paths are created using the newer *MAX1TB access path size.

On the down side, those organizations that are currently memory constrained may see an increase in fault rates due to the larger memory requirement of the 64k access path. These organizations will need to proceed in a more cautious manner, perhaps gathering performance data before and after each logical file change.

Those applications that are using select/omit logical files may experience some degradation when adding the DYNSLT keyword as there may now be some delay before records are returned to the application program. Simply removing the DYNSLT keyword and recreating the logical will cause it to revert back to an 8K access path and eliminate the delay time.

Keep in mind as you move down the path of database modernization if your intent is to eventually replace your legacy record level access methods with SQL then non-modern techniques such as select/omit logical files will need to be replaced by more efficient keyed access methods. The applications that use these files would be at the top of my list for modernization.

Reengineering DDS to SQL DDL (Stage 1)

So we're finally here. Hopefully I have given you some more incentive to move your legacy DDS database to an SQL DDL defined database. Before we begin I want to state my number one objective when modernizing a database. From this day forward we need to minimize the impact of change on the business. This is my mission statement.

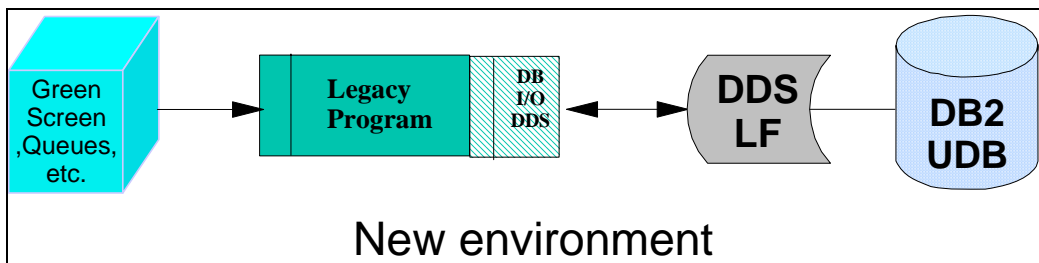
The previous two sections contained the results of running RPG programs against the original DDS database, then against the new database where the physical file object names were different. Then we built SQL indexes, recreated the logical files and reran the same tests. In all cases I never recompiled the programs. They ran without fail, no level check ID errors. This is what I mean by minimizing the impact of change.

Many organizations are now running near 24x7 operations. The window for application and database modification is shrinking. The last thing you want is to use up that valuable window of time waiting for every program to recompile because you added a new column to a single table. To accomplish this feat not only requires modernization of your database but also modernization of your process, and, most importantly, modern tools.

The following figure below depicts a typical iSeries legacy application that uses RLA methods to read and write data to and from a DDS created database.



The figure below depicts the same application after the DDS database has been converted to an SQL DB2 database. Note the main difference is the DDS LF that now comes between the application program and the actual data.



In essence, done correctly, the underlying DDS database can be converted to a SQL DDL created database without changing a single line of code in the legacy programs. This is the strategy I refer to as Stage 1.

Approach to Stage 1

We learned that reads from DDL created tables are faster. In addition, we now know that keyed reads through the 64K access path are also better performers. Now we need to identify the tables that will best benefit from faster reads. To do this we need to have profile data of our tables. In essence which tables are read the most?

Every physical file object contains statistical data in the form of opens, closes, reads, writes, updates, etc. This information is accumulated from scratch after each IPL. To collect profile information on those tables that have the most activity during peak periods I use the DSPFD command to create an outfile containing my profile data. The following is an example of the DSFPD command that I use:

```
DSPFD FILE(DB2REDDS/*ALL) TYPE(*MBR) OUTPUT(*OUTFILE)
FILEATR(*PF) OUTFILE(DB2REDDS/DSPFD_MBR)
```

I run the command once before the start of the peak period to create a baseline. I then run the command a second time after the peak period to capture changes in the table statistics. I modify the command to add the new data to the existing outfile created from the first execution of the command. This is shown below:

```
DSPFD FILE(DB2REDDS/*ALL) TYPE(*MBR) OUTPUT(*OUTFILE)
FILEATR(*PF) OUTFILE(DB2REDDS/DSPFD_MBR) OUTMBR(*FIRST *ADD)
```

I then run the following query, substituting the &Schema and &Xref_File variables with the name of the library and outfile used in the DSPFD command. (Note: the DELTA function is a User Defined Function (UDF) and can be found at the following Web site: <http://www-1.ibm.com/servers/eserver/series/db2/db2code.html> The DELTA function is found under the RPG UDF example).

```
WITH DELTA AS(
  SELECT A.MBFILE, MBRTIM,
         QGPL.DELTA(A.MBWROP , MBFILE) AS LOGICALWRITES,
         QGPL.DELTA(A.MBUPOP , MBFILE) AS LOGICALUPDATES,
         QGPL.DELTA(A.MBDLOP, MBFILE) AS LOGICALDELETES,
         QGPL.DELTA(A.MBLRDS, MBFILE) AS LOGICALREADS,
         QGPL.DELTA(A.MBPRDS, MBFILE) AS PHYSICALREADS
  FROM &Schema.&Xref_File A
  ORDER BY 1 ASC, 2)

SELECT MBFILE, MAX(PHYSICALREADS) AS PHYREADS,
       MAX(LOGICALREADS) AS LGLREADS,
       MAX(LOGICALUPDATES) as UPDATES,
       MAX(LOGICALDELETES) AS DELETES,
       MAX(LOGICALWRITES) AS WRITES

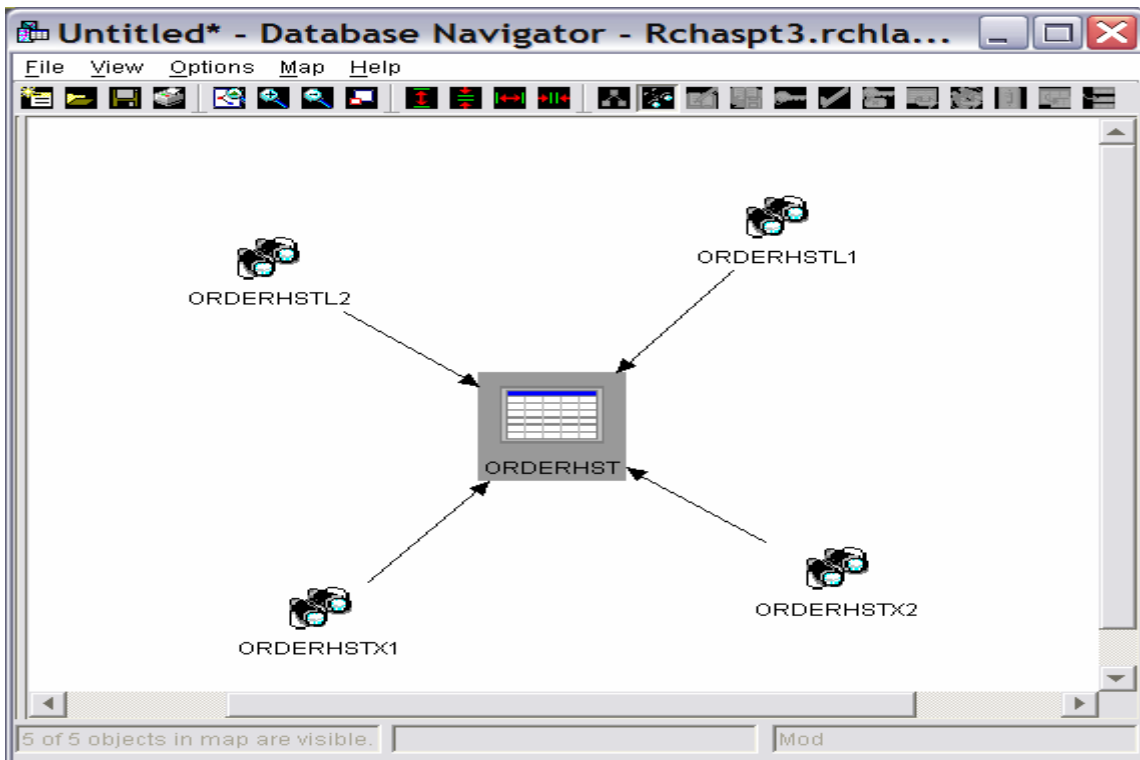
FROM DELTA
GROUP BY MBFILE
HAVING MAX(PHYSICALREADS) > 0
ORDER BY MBFILE
OPTIMIZE FOR 1 ROW
```

I use QMF for Windows to run my SQL statements. For more information about QMF refer to the following Redbook “*A DB2 Enterprise Query Environment - Build It with QMF for Windows !*” I fill a spreadsheet with the results from QMF and sort it by physical reads in descending order.

Here are the results:

File	PHYREADS	LGLREADS	UPDATES	DELETES	WRITES
ORDERHST	91932	856299	216267	0	58424
PF01	58842	73354	73597	0	39
PF02	18826	664611	71366	0	0
PF03	2749	150606	8913	0	76975
PF04	1703	89295	0	86974	265164
PF05	419	343701	52190	1627	18211
PF06	125	5131	20182	0	95
PF07	51	59581	0	0	0
PF10	12	26336290	0	0	0
PF12	4	93389	0	0	0

I like OrderHst due to the high number of physical reads (disk to memory) plus a decent logical read to update ratio (approximately 4 to 1). After identifying the physical file candidate I identify the logical files based on this physical file using the iSeries Navigator Database Navigator function. Below is a map of the OrderHst physical file created from iSeries Navigator:



By right clicking on the ORDERHST icon I get a pop-up menu which allows me to generate a CREATE TABLE SQL statement from the existing DDS created physical file. The following is a snippet from the generated statement:

```
CREATE TABLE DB2REDDS.ORDERHST (  
-- SQL150B 10 REUSEDLT(*NO) in table ORDERHST in DB2REDDS  
ignored.
```

```

-- SQL1509  10  Format name ORDERHSTR for ORDERHST in DB2REDDS
ignored.
      ORDERKEY DECIMAL(16, 0) NOT NULL DEFAULT 0 ,
      |
PRIMARY KEY( ORDERKEY , PARTKEY , SUPPKY , LINENUMBER ) ) ;

```

You need to review the generated script for informational messages that identify differences between SQL table definition and DDS definitions. Notice in the above script that the reuse of deleted records setting *NO will not be honored. Also the format name from the original physical file will be ignored. In essence the SQL physical file object format name will be the same as the physical file object name. At this point this is no big deal. We will discuss format names shortly.

Before creating the SQL table I modify the SQL script to a) change the schema (library) where the SQL table will be created, b) change the name of the table and c) comment out the PRIMARY KEY constraint (my original DDS physical file is simply using a UNIQUE key). The following is the modified script:

```

CREATE TABLE DB2REDDL.ORD_HST (
      ORDERKEY DECIMAL(16, 0) NOT NULL DEFAULT 0 ,
      |
--PRIMARY KEY( ORDERKEY , PARTKEY , SUPPKY , LINENUMBER )
) ;

```

I now run the script and create the new table. The physical file object name and format name is ORD_HST. Again, no big deal. Bear with me until we reach the format name discussion.

The existing DDS created logical files appear as views in iSeries Navigator. The Generate SQL option will generate a CREATE VIEW statement for these files however it will not generate a CREATE INDEX statement. In Stage 1 we need to create SQL indexes to take advantage of the 64K access path. Since iSeries Navigator doesn't do it for us we need to find another way. To become more familiar with iSeries Navigator I recommend the "Get started with iSeries™ Navigator" topic under Database overview in the iSeries Information web site. Use the following URL <http://publib.boulder.ibm.com/infocenter/iseres/v5r3/ic2924/index.htm>.

To help identify indexes that need to be created I use the SQL stored procedure Index Listing Utility from the same web site where I obtained the DELTA UDF. The following is the end result of running the Index Listing Utility from QMF and formatting the results in a spreadsheet:

Ordinal Position										
	SYSTEM INDEX	SYSTEM INDEX	Total Key	Column 1 Key	Column 2 Key	Column 3 Key	Column 4 Key			
IS UNIQUE	NAME	SCHEMA	Columns	Column	Ordering	Column	Ordering	Column	Ordering	Column
U	ORDERHST	DB2REDDS	4	ORDERKEY ASC	PARTKEY ASC	SUPPKY ASC	LINENUMBER ASC			
D	ORDERHSTL1	DB2REDDS	1	ORDERDATE ASC						
D	ORDERHSTX1	DB2REDDS	1	ORDERDATE ASC						
D	ORDERHSTL2	DB2REDDS	1	PARTKEY ASC						
D	ORDERHSTX2	DB2REDDS	1	PARTKEY ASC						

QMF Tip: Remember the System 38 Query had the ability to create horizontal results? QMF has that same feature. Its called Across. I specified Across on the Ordinal Position column in the QMF form and my key columns are listed horizontally. From here it is pretty easy to copy and paste the columns into an SQL script that contains the CREATE INDEX statement as shown below:

```

CREATE UNIQUE INDEX DB2REDDL.OrderHistory_IX1
  ON DB2REDDL.ORD_HST (ORDERKEY ASC, PARTKEY ASC, SUPPKEY
ASC,LINENUMBER ASC);

CREATE INDEX DB2REDDL.OrderHistory_IX2
  ON DB2REDDL.ORD_HST (ORDERDATE ASC);

CREATE INDEX DB2REDDL.OrderHistory_IX3
  ON DB2REDDL.ORD_HST (PARTKEY ASC);

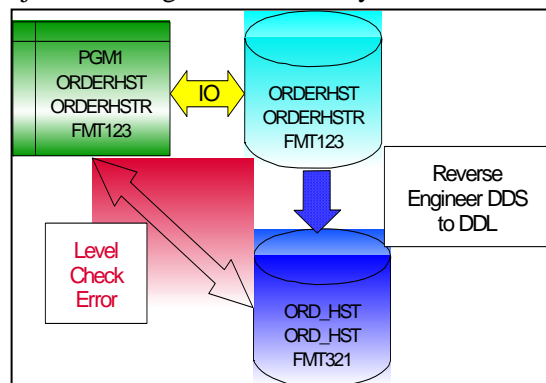
```

There are 5 keyed logical files however I only need to create 3 indexes as ORDERDATE and PARTKEY appear twice. Also, since I will never ever, reference these indexes in a HLL program or on an SQL statement I can name them however I please. In this case I use a nice meaningful long name OrderHistory_ followed by some kind of suffix for uniqueness. Note that the first index is the UNIQUE index that originally was part of the ORDERHST physical file.

Before we continue, now is the time to discuss the concept of format sharing. Every file object has an object name and a format name. For SQL created tables, the object and format names are identical. RPG programs require that the format name be different than the file name. If they are the same then the program would not compile unless the format was renamed within the RPG program itself. To avoid compilation errors and/or program format renaming most organizations used DDS to create physical file objects with format names different from the object name.

The program compilation problem is made worse because of format sharing. The most common form of format sharing is simply specifying the format name of a physical file within the DDS of a logical file. This is another very common technique used by most organizations. In essence, multiple keyed logical files were created to share the format of a physical file but describe different ways of ordering and selecting data.

Furthermore, every format has an associated format ID. This ID is duplicated into every program object that references a file object. Should the file object be changed in such a way that the format ID changes then the system will signal an exception when a program object references the changed file object. When you reverse engineer a DDS physical file object to an SQL physical file object the format ID will change. An RPG program that references the original physical file object (or any logical files that share the format) will receive a “level check” exception message. This is shown in the picture on the right. The only solution would be to a) recompile the program or b) ignore the exception (which I strongly do not recommend).



Remember I said I did my testing without recompiling programs? To avoid having to recompile programs after converting DDS to DDL I use another, less common form of format sharing. In DDS there is a FORMAT keyword. The FORMAT keyword is used to share the format of an existing physical or logical file object by the physical or logical file being created. To take advantage of this keyword I convert my original DDS physical file source to a logical file as shown below:

Original PF ORDERHST			
	A		UNIQUE
Org	A	R ORDERHSTR	
	A*		
	A	K ORDERKEY	
	A	K PARTKEY	
	A	K SUPPKEY	
	A	K LINENUMBER	
Modified LF ORDERHST			
	A		UNIQUE
Chg	A	R ORDERHSTR	PFILE(ORD_HST)
	A*		
	A	K ORDERKEY	
	A	K PARTKEY	
	A	K SUPPKEY	
	A	K LINENUMBER	

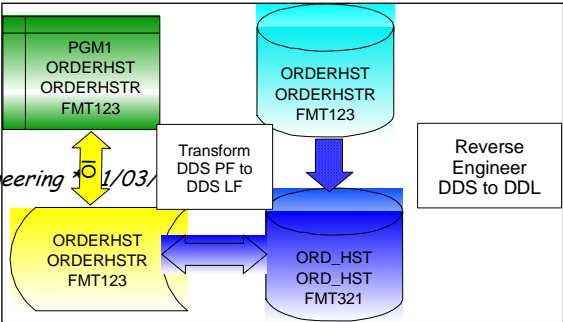
The only change I make to the source is to add the PFILE keyword which points to the new SQL table. Everything else remains the same. If I was using a Field Reference File (FRF) then my change would look like this:

Original PF ORDERHST			
Org	A		REF(FIELDREF)
	A		UNIQUE
Org	A	R ORDERHSTR	
	A	FLD1 R	
Org	A	FLD2 R	REFFLD(FLD3)
	A*		
	A	K ORDERKEY	
	A	K PARTKEY	
	A	K SUPPKEY	
	A	K LINENUMBER	
Modified LF ORDERHST			
Chg	A*		REF(FIELDREF)
	A		UNIQUE
Chg	A	R ORDERHSTR	PFILE(ORD_HST)
	A	FLD1 R	
Chg	A*	FLD2 R	REFFLD(FLD3)
New	A	FLD2	
	A*		
	A	K ORDERKEY	
	A	K PARTKEY	
	A	K SUPPKEY	
	A	K LINENUMBER	

I add the PFILE and comment out the REF(FIELDREF) line. For REFFLD I comment out the original and drop the REFFLD keyword from the new specification. More about field reference files in my concluding remarks.

I recreate the modified PF to LF file and all programs that reference the original file now

iSeries Database Reengineering 1/03/

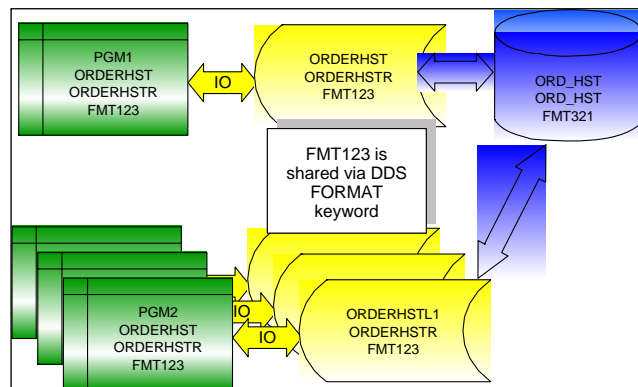


run without a level check error. In essence the format has not changed even though it is now a LF pointing to an SQL table. This is shown in the figure on the right. This eliminates the level check issue for those programs which reference the PF file object directly.

We now have to modify and recreate the logical files that referenced the original physical file. The following are code snippets from that process:

Logical File ORDERHSTL1		
Org	A	R ORDERHSTR1 PFILE(ORDERHST)
Chg	A	R ORDERHSTR1 PFILE(ORD_HST)
Logical File ORDERHSTL2		
Org	A	R ORDERHSTR2 PFILE(ORDERHST)
Chg	A	R ORDERHSTR2 PFILE(ORD_HST)
Logical File ORDERHSTX1		
Org	A	R ORDERHSTR PFILE(ORDERHST)
Org	A	K ORDERDATE
Org	A	S LINESTATUS COMP(EQ 'S')
New	A	DYNSLT
Chg	A	R ORDERHSTR PFILE(ORD_HST)
New	A	FORMAT(ORDERHST)
	A	K ORDERDATE
	A	S LINESTATUS COMP(EQ 'S')
Logical File ORDERHSTX2		
Org	A	R ORDERHSTR PFILE(ORDERHST)
Org	A	K PARTKEY
Org	A	S DISCOUNT COMP(GT 0)
New	A	DYNSLT
Chg	A	R ORDERHSTR PFILE(ORD_HST)
New	A	FORMAT(ORDERHST)
	A	K PARTKEY
	A	S DISCOUNT COMP(GT 0)

Logical files ORDERHSTL1 and ORDERHSTL2 were not sharing a format name, thus the only change required was to the PFILE keyword. Logical files ORDERHSTX1 and ORDERHSTX2 were sharing the format of file ORDERHST. In addition they were using select/omit criteria. In both cases I added the DYNSLT keyword and the FORMAT keyword. The DYNSLT keyword allows the access path to be shared and the FORMAT keyword allows the format of now logical file ORDERHST to be shared (and its format ID). In essence this eliminates level check errors on programs that reference these logical files. This is shown in the figure on the right.



After creating the logical files review the file descriptions and verify that the access paths are being shared and format IDs have not changed. At this point you are now ready to migrate the

data and test the programs that use the DDS files referencing the new SQL database. Once the new SQL table has been implemented you are now ready for Stage 2.

Conclusions

Some organizations have discovered “dirty data” during the migration process. You may have to create some data cleansing programs as part of this stage.

Some organizations have found new exceptions due to data validation occurring at insert time. In the past we may have simply ignored decimal data errors. Now you may have to change the programs that execute write operations to handle new exceptions.

Removing the REFFLD keyword during the PF to LF conversion will result in the inability to do where used searches based on field reference information. I suggest that you reverse engineer your existing FRFs to SQL tables. Be aware that you will lose all presentation related keywords, so do not delete the original FRF. Use the new SQL FRF table to add new columns, then use iSeries Navigator to reference those columns when creating or changing SQL tables.

All display files and print files will continue to reference the original FRF. If these types of files reference real physical files then they should be changed to reference an FRF. See the Field Reference sidebar for more information on this matter.

I strongly recommend that from this point forward all new physical files be created from SQL CREATE TABLE DDL statements. In addition, all HLL access to the DDL created physical files will be done through DDS logical files. Also, now that you know about access path, sharing, be sure to create an SQL index with the same keys as the logical that you are creating. Furthermore, make it a habit to only specify the fields needed by the program in the logical file.

To offset the loss of FRF support I recommend using a data modeling tool for all database object creation and alteration. There are several worthy products on the market. Also, by implementing a good logical file strategy you will be able to use SQL catalog support in addition to DSPPGMREF to perform column to program where used queries.

The hardest thing to do at this time is avoiding the urge to change the database. We all want to use identity column support, Referential Integrity (RI), rename or resize columns, change attributes, etc. Any change to the database may result in format changes to the DDS logical files, resulting in the need to recompile.

Be patient. In some cases it took over 20 years to create these databases. Let’s not try to change it all overnight.

Summary

Over time the iSeries/i5 platform has evolved. Many of us have attempted to evolve with it, however we have discovered that we can no longer be experts in all areas of this platform. The tools that we used to develop applications in the past just don't cut it when we attempt to develop applications based on today's requirements.

There was and still is a need for the functions that FRCRATIO and Field Reference Files provided. These were a means to an end. However, the means have changed but the requirement has not. There are now better, more modern ways and means to accomplish the same end.

Ten years ago I considered myself to be an RPG expert. Today, I have to refer to the manual more than I care to admit. In fact, the only reason my RPG examples are in free format is because the WDS Sc LPEX editor provides a function that does it for me. Even this old dog can learn new tricks.

Reengineering older DDS created databases is not simply pushing a button, nor is it an overnight process. If it was easy we all would've done it years ago. It requires an understanding of the changes in database architecture and the benefits or pitfalls in using the new architecture.

As you begin this journey I strongly recommend that you create a position (if you haven't already) for a Database Architect. There are going to be lots of decisions that have to be made regarding legacy issues such as multi-format and multi-member files, select/omit logical files, field reference files, etc. In addition there will be other decisions regarding the use of identity columns, Referential Integrity, column level security, large object support, XML, etc.

These decisions cannot be left up to application programmers or system administrators. The Database Architect provides a bridge between these two entities. Indeed this person may be able to provide both "great" and "fulfilling" database solutions utilizing SQL.

Field Reference File Sidebar

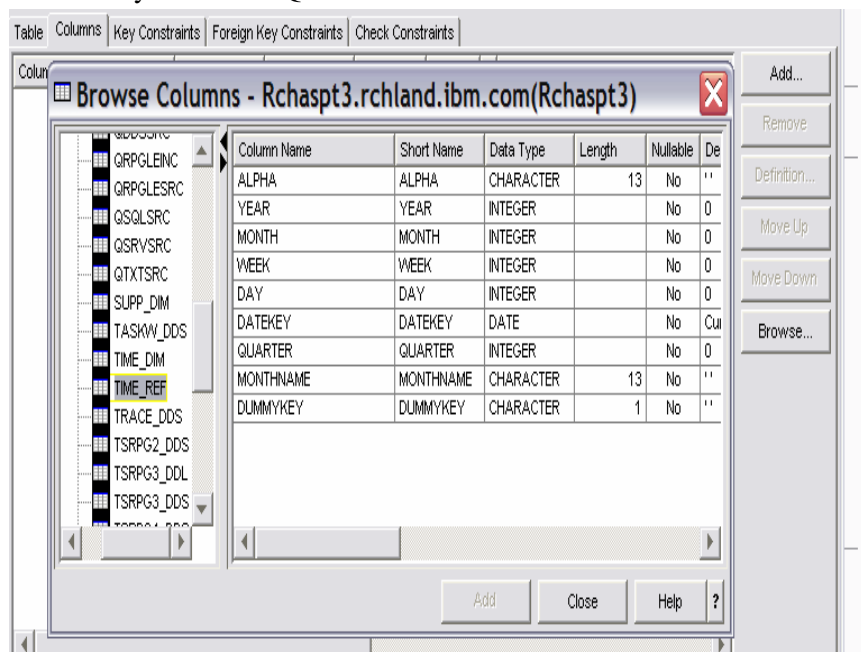
It never fails that when I discuss the subject of reengineering DDS to DDL someone will bring up the use of Field Reference Files (FRF). The concept of an FRF is simple. When defining a file (database, display or printer) via DDS rather than having to remember the attributes of a field simply specify an R in the reference column of the DDS specification. The REF file level keyword identified a database object (any externally described file) which contains the field definition. During the creation of the object the field definitions from the FRF were used.

The tool used by most organizations to enter DDS source was SEU. Specifying the R was fewer keystrokes then pressing F15, searching for another source member that contained the field definition and then copying that definition into the new source.

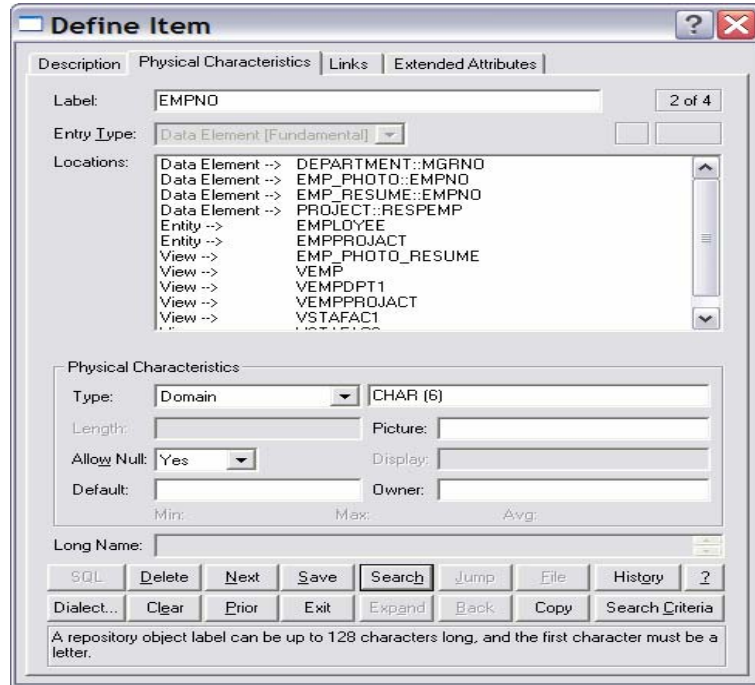
The system metadata captured this reference information and allowed developers to create cross reference databases that could be searched to find all the files that contained a referenced field. This allowed developers to do impact analysis when business requirements dictated the need for a change. Probably the most commonly known scenario would be Y2K conversions. There are several products available that exploit this cross reference data. In addition, these products provide plug-ins to modern tools like iSeries Navigator and WebSphere Development Studio client.

The Interactive Data Dictionary Utility (IDDU) was a carry over from the System/36 platform. In essence IDDU provided interactive field reference file support. What I liked about IDDU was the ability to define fields without having to create a physical file. In essence it provided a means of defining the data at the field level and then replicating those definitions to record formats. IDDU definitions could also be accessed by SQL. In fact, an IDDU data dictionary is created when the WITH DATA DICTIONARY clause is specified on a CREATE SCHEMA statement. Unfortunately, like DDS, IDDU has not kept up with the modern RDMS support.

iSeries Navigator provides the ability to create SQL tables that reference columns from other tables. The table definition Columns tab contains a browse button which allows you to select a physical file that contains the column definitions you wish to use when creating a new table or adding columns to an existing table. This is shown on the right. iSeries Navigator does not have a column where used capability.



Other modern tools exist for designing and altering databases. Products like Visible Analyst contain the ability to define fields as domains and then reference those domains when building entities. The picture on the right is from the Visible Analyst tool. I can use this tool to identify where a field is used (e.g. EMPNO), modify that field and be assured that all instances of the field have been changed. I can then rebuild the logical data model. This model can then be compared to the physical database on my iSeries and ALTER TABLE statements can be generated where the differences occur.



Whatever method or tool you use is only as good as the level of adherence to standards imposed on a development shop by its managers. Today there is still no built in *domain integrity* support (for lack of a better name) on the iSeries that forces compliance to development standards. In essence, if all programmers have the ability to create database objects then all it takes is one to not use the tool and you no longer have the cross reference integrity that you may be relying on.

I have had many opportunities to review several legacy databases. I never fail to find inconsistencies in the use of field reference files. I simply do not rely on them as a good data dictionary. I believe the best way to avoid this problem in your organization is to designate a single individual or team as the keeper of the database. This person or team is called the Database Architect/Team/Group. They are the only personnel allowed to create or alter database objects.