



Modernizing the FLGHT400 database:

Modernizing a DB2 UDB for iSeries application

• • • • • • • • •

Gene Cobb
IBM ISV Strategy and Enablement
December 2005

Table of contents

Change history	Error! Bookmark not defined.
Abstract.....	1
Introduction	1
Modernization Goals.....	1
Modularization Plan	2
Tools used	2
Description of libraries and schemas.....	3
Stage 1: Reverse engineering DDS to SQL DDS.....	3
Step 1: Classifying the existing environment.....	4
Step 2: Establishing a list of all DDS files to be converted.....	5
Identifying tables.....	5
Identifying Indexes.....	6
Step 3: Creating SQL DDL scripts.....	9
Creating script to build the tables.....	9
Creating script to build the indexes.....	11
Reviewing the SQL DDL scripts	12
Changing the target schema	12
Adding columns.....	12
Other considerations	13
Record Format Name.....	13
Managing SQL DDL.....	13
Creating the new SQL objects	13
Creating the new DB2 Schema (collection) on the iSeries server	13
Creating the tables	14
Creating the indexes.....	15
Copying data to new schema	15
Stage 2: Creating I/O modules to access data	17
Step 1: Identifying all programs and modules that are to be converted.....	18
Step 2: Identifying all instances of RLA in the programs identified in step 1	19
Step 3: Documenting the business rule	20
Step 4: Creating the SQL view to access data.....	24
Step 5: Creating an I/O procedure for each unique RLA instance or group	25
Step 6: Replacing each RLA instance or group with call to new I/O procedure	31
Step 7: Creating the I/O module.....	32
Step 8: Creating the I/O service program.....	32
Step 9: Recompiling existing modules, service programs, and programs	33
Other considerations.....	34
Null values.....	34
SQL error handling.....	34
Stage 3: Moving business rules to the database	36
Implementing referential integrity constraints	37
Implementing check constraints	38
Implementing automatic key generation and unique identifiers	39
Implementing trigger programs.....	41
Stage 4: Externalizing data access	41
Summary.....	45
Appendix A: SQL scripts.....	46
SQL procedure GenIndexList	46
SQL procedure GenIndexList2	47
SQL DDL script to create the tables	50
SQL DDL script to create the indexes	53
SQL DDL script to create the views.....	54

SQL DDL script to create the constraints	55
Appendix B: Source code after conversion.....	56
NFSSQL (binder source)	56
NFSSQLPR (prototypes)	56
NFSSQL (SQL I/O procedures).....	60
Programs and modules converted from RLA to SQL I/O procedures	79
NFS001	79
NFS402	85
NFS404	86
NFS405	88
Appendix C: Resources.....	90
Appendix D: About the author	91
Trademarks and special notices.....	92

Abstract

This paper enables the reader to understand the steps involved in updating the database definitions and data access methods of an IBM® iSeries™ RPG 5250 application. This is done by bringing the reader through a sample application, FLGHT400. Readers can use this example to better understand how to implement improved database techniques in their own applications.

Introduction

The IBM iSeries Developer Roadmap is a key initiative that helps iSeries developers enhance their existing applications and toolsets. The SQL programming language and IBM DB2 Universal Database™ (DB2® UDB) are two modern tools that are available to help iSeries programmers move along the iSeries Developer Roadmap. Using the FLGHT400 application as an example, this white paper explains the process from an SQL and DB2 UDB point of view.

Embracing newer database techniques provides multiple benefits for your application:

- Portability of code and skills
- Strategic database interface for the application development industry and the IBM i5/OS® operating system
- Better positioning of the iSeries family of systems as a database server
- Reduction of the total lines of SQL code
- Enablement of the use of DB2 UDB symmetric multiprocessing (SMP) and parallel database processing

Establishing database enhancement goals

In the white paper “Modernizing FLGHT400,” the author focused on one aspect of iSeries application modernization. That paper discussed and demonstrated various modularization techniques and methodologies using the FLGHT400 application. Its primary objective was to explain and demonstrate the process of separating the application into callable modules. When modularized, you can access the functions of the application through either a Web interface or 5250 terminal interface. Because both interfaces used the same business logic, dual maintenance issues were avoided and the architecture for reusable components was established. Refer to the **Resources** section for information regarding the “Modernizing FLGHT400” white paper.

In this paper, the evolution of the FLGHT400 application continues and focuses on implementing newer database techniques, and includes step-by-step examples of the implementation.

As a guide to this process, you will follow the stages as documented in the IBM Redbook abstract “Modernizing IBM eServer iSeries Application Data Access - A Roadmap Cornerstone.” This Redbook provides a thorough discussion on why database enhancement is important and explains its benefits. Refer to the **Resources** section for the Web site associated with this Redbook.

Knowing the modernization plan

The methodology for the database modernization involves three stages:

- **Stage 1:** Reverse engineering data description specifications (DDS) to SQL data definition language (DDL)
- **Stage 2:** Creating I/O modules to access the database
- **Stage 3:** Moving business rules into the database

Using FLGHT400 as an example, this paper show the activities involved in each stage.

Using the tools

This section describes the tools used during the database modernization process of the FLGHT400 application.

IBM WebSphere® Development Studio Client Advanced Edition for iSeries is an integrated development environment (IDE) and tool set for developing Java™, Web, Web-service, client/server, and iSeries system applications. The Remote Systems Explorer (RSE) component of this development tool is used to access and edit the FLGHT400 application. To access this WebSphere tool set:

1. Open WebSphere Development Studio Client from your workstation.
2. Specify your workspace.
3. At the toolbar menu, click **Window** > **Open Perspective** > **Remote Systems Explorer** (Figure 1).
4. Expand the iSeries connection.

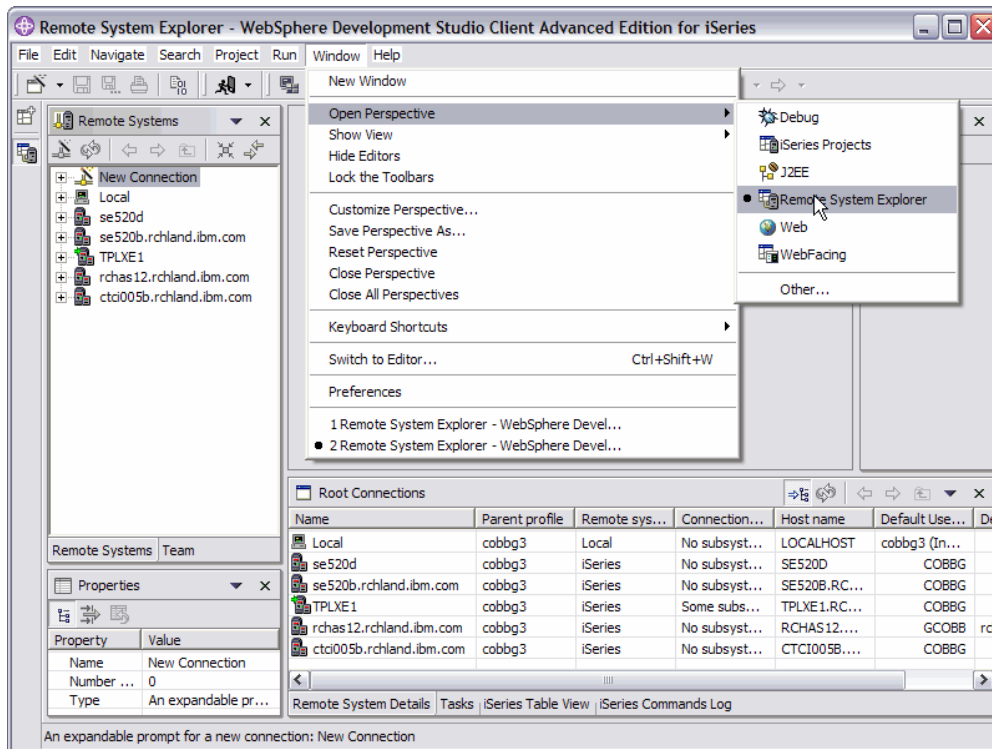


Figure 1: Open RSE perspective

iSeries Navigator is the main interface used to work with database objects and is accessed as follows:

1. Open iSeries Navigator from your PC.
2. Expand an iSeries connection and click **Databases > Database (your system name) > Schemas**.
3. Select the schema **FLGHT400**.

Describing libraries and schemas

As a result of the evolution of the FLGHT400 application, multiple libraries have been created to store the various versions.

- **FLGHT400:** Contains the original programming model (OPM) programs, physical files, logical files, source code files, and other objects of the original version of the FLGHT400 application.
- **FLGHT400M:** Contains the Integrated Language Environment (ILE) program and service programs. The application modernization process stimulated the development of this library.
- **FLGHT400M2:** Contains the ILE and service programs, as well as the SQL objects (tables, views, indexes, and so forth). The database modernization process stimulated the development of this library.

Note: The last two libraries listed above hold only the incremental changes made to the application. Thus, to run the application you must restore all three libraries and set up the library list in the following order:

1. FLGHT400M2
2. FLGHT400M
3. FLGHT400

Stage 1: Reverse engineering DDS to SQL DDS

This stage, also known as improving data definitions, is not a required or prerequisite for the other stages but is important for several reasons. The conversion of DDS-created files to SQL DDL-created database objects results in the benefits listed below.

- **Faster reads as data validation is performed at write instead of read:** This can yield performance improvements. In most applications, data is read more often than it is written. In those cases, the data no longer requires validation during every read operation, thus realizing better performance.
- **64-kilobyte access paths on SQL created indexes:** This means more keys can be loaded into memory. Single-key lookups using an index might not be as efficient, because it takes longer to read a 64-kilobyte page than an 8-kilobyte page. Regardless, you will see performance improvements in queries that load many key values into an index (as more of the keys are brought into memory).
- **More data types:** The expanded choices for supported data types include datalinks, binary large objects (BLOBs), and character large objects (CLOBs).
- **Constraint definitions can be included in object source:** DDS has no support for constraints.
- **Longer, more descriptive table and column names:** DDS column names are restricted to 10 characters. In the release following i5/OS Version 5 Release 3, SQL will support up to 128-character column names.

In the remainder of the section, you will learn how the FLGHT400 database converts from DDS to SQL DDL. This stage is comprised of six steps:

1. Classify the existing environment.
2. Establish a list of all DDS files to be converted.
3. Define naming conventions for SQL objects.
4. Convert the DDS to SQL DDL.
5. Review the generated SQL DDL.
6. Create the new DB2 UDB schemas and SQL objects on the iSeries system.
7. Copy data to new schemas.

Step 1: Classifying the existing environment

The first step in this stage is to classify and understand the existing environment that we will convert to SQL. The primary objective of this step is to provide an idea of the amount of effort that lies ahead. In general, the lower the classification, the more work is involved to complete the enhancement. Here is description of each classification:

- **Class 0:** These are program-described files. In this environment, the DDS contains some key fields and one large field. The large field is specified in RPG in an I spec and in COBOL in the FD section.
- **Class 1:** This is a mix of program-described and externally-described files. This environment usually has no normalization in place.
- **Class 2:** These files are externally-described with no referential integrity. In this environment, some type of normalization exists. Journaling is usually not used, and unique keys are defined for the physical or logical files.
- **Class 3:** These files are also externally-described with some referential integrity constraints. In this environment, a greater degree of normalization is in place. Transaction files usually are in second normal form (2NF) and master files in third normal form (3NF). Primary and foreign key constraints are defined, and journaling is used; although commitment control is probably not used.
- **Class 4:** These files are externally-described with referential integrity and some business logic has moved to the database. In this environment, the database is highly normalized. Some of the business logic has moved to the database using Referential Integrity (RI), triggers, stored procedures, user-defined types (UDTs), user-defined functions (UDFs), journaling, and commitment control.

Using the information about each of the classifications, it is possible to determine that the FLGHT400 application has the following characteristics:

- Contains all externally described files
- Has no referential integrity
- No primary or foreign key constraints are defined
- Uses neither journaling nor commitment control

These facts indicate that class 2 is the appropriate classification.

Step 2: Establishing a list of all DDS files to be converted

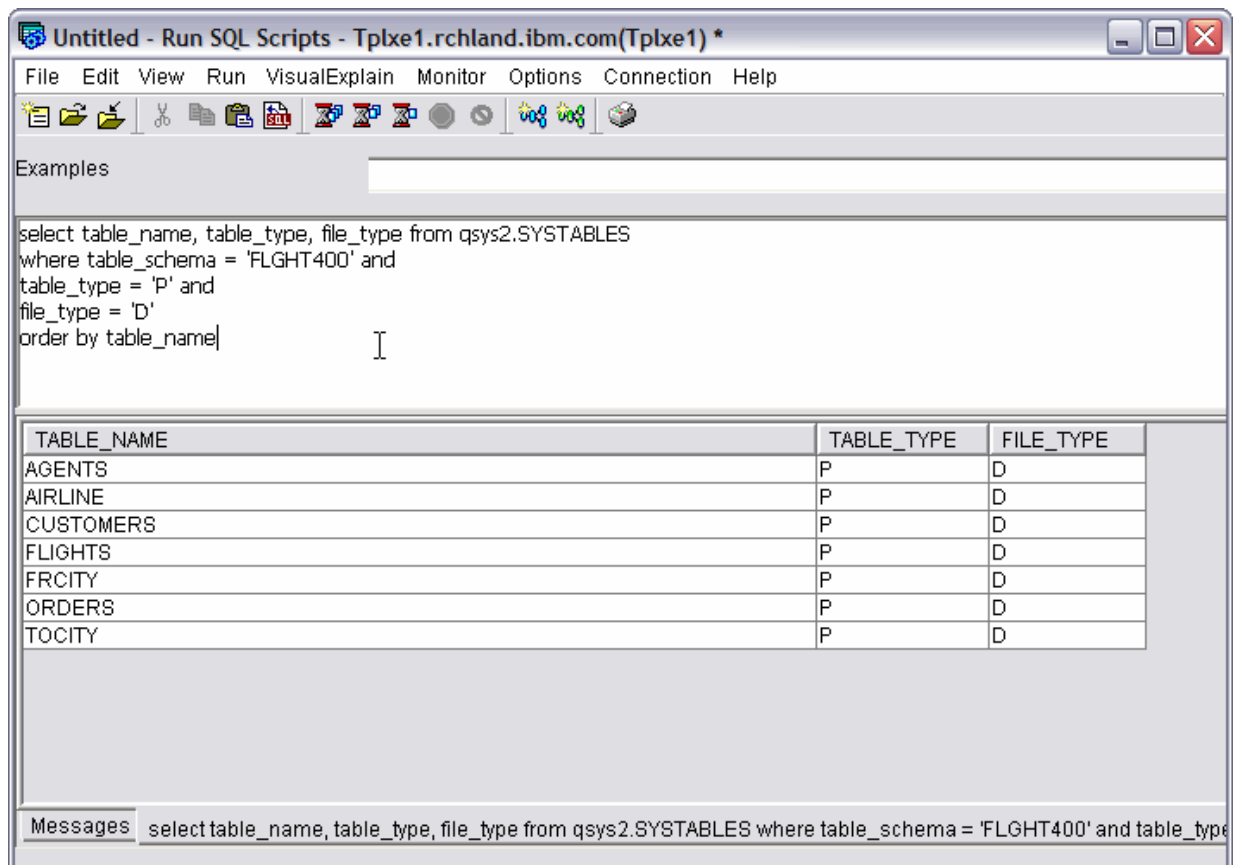
The purpose of this section is to identify all of the DDS-created files that are to be converted to SQL DDL-created tables and indexes.

Identifying tables

The first step is to determine which DDS physical files are to convert to SQL tables. In the case of FLGHT400, all application files are within the same library. Because of this, the technique is straightforward; find all the physical files (table type = 'P') in the library with an object attribute of **Data** (file type = 'D'). The following query produces a list of tables in schema FLGHT400 to reverse engineer:

```
SELECT table_name, table_type, file_type
FROM qsys2/SYSTABLES
WHERE table_schema = 'FLGHT400' and
      table_type = 'P' and file_type = 'D'
ORDER BY table_name
```

Open a Run SQL Script window from iSeries Navigator and execute this query statement. Figure 2 shows the results.



The screenshot shows a window titled "Untitled - Run SQL Scripts - Tplxe1.rchland.ibm.com(Tplxe1) *". The window contains a menu bar (File, Edit, View, Run, VisualExplain, Monitor, Options, Connection, Help) and a toolbar with various icons. Below the toolbar is an "Examples" section with a text input field. The main area contains the following SQL query:

```
select table_name, table_type, file_type from qsys2.SYSTABLES
where table_schema = 'FLGHT400' and
table_type = 'P' and
file_type = 'D'
order by table_name|
```

The results of the query are displayed in a table with the following data:

TABLE_NAME	TABLE_TYPE	FILE_TYPE
AGENTS	P	D
AIRLINE	P	D
CUSTOMERS	P	D
FLIGHTS	P	D
FRCITY	P	D
ORDERS	P	D
TOCITY	P	D

At the bottom of the window, a "Messages" pane shows the executed query: "select table_name, table_type, file_type from qsys2.SYSTABLES where table_schema = 'FLGHT400' and table_type

Figure 2: List of DDS files to convert

Identifying Indexes

The next step is to identify the indexes to create (based on the existing keyed logical files). It is important to note that the indexes are not required for SQL database access. During execution of an SQL statement, the SQL optimizer will decide which access method to use. This means that if no indexes exist, the optimizer might perform a full table scan to carry out the query. Though the query will still execute successfully without any indexes, its performance can be less than optimal. Creating the right indexes will not only improve performance, it will also give the optimizer more information to accurately gauge the resource cost for each available access method.

At this stage, it is difficult to determine which indexes to create. The best option is to build indexes based on the existing keyed logical files.

In the “SQL scripts” section of Appendix A, you will find the code for two utility procedures, GENINDEXLIST and GENINDEXLIST2. When called, these procedures will return the list of indexes to consider creating, based on the existing keyed logical and physical files. The first procedure, GenIndexList, accepts the schema name as an input parameter. It then searches SYSTABLES and finds all physical data files in the specified schema. For each physical data file found, procedure GenIndexList2 is called and accepts the physical file name and schema name as input parameters. It finds any object that can be recreated as an SQL index (keyed logical file, constraint, and keyed physical file) against that physical file and adds a row to the result table. The contents of the result table can then be used to create the script with the necessary Create Index statements.

To create and run the procedures, take the following actions:

1. From a Run SQL Scripts window, enter and execute the CREATE PROCEDURE statements (located in Appendix A: SQL scripts) to build the utility procedures.
2. From a Run SQL Scripts window, enter and execute the statement to call the utility procedures and generate a list of indexes to consider creating:

```
CALL QGPL.GenIndexList('FLGHT400');
```

Figure 3 shows the results of these actions.

TABLE_NAME	TABLE_SCHEMA	IDX_NAME	IDX_SCHEMA	IS_UNIQUE	KEY_COLUMN	ORDINAL_POS	ORDERING	SYS_IDX_NAME	SYS_IDX_SCHEMA
AGENTS	FLGHT400	AGENTS	FLGHT400	U	AGENT_NO	1A	1A	AGENTS	FLGHT400
AGENTS	FLGHT400	AGENTS_L	FLGHT400	D	AGENT_NAME	1A	1A	AGENTS_L	FLGHT400
AGENTS	FLGHT400	AGENTS_N	FLGHT400	D	AGENT_NO	1A	1A	AGENTS_N	FLGHT400
AGENTS	FLGHT400	AGENTS_Z	FLGHT400	D	AGENT_NO	1A	1A	AGENTS_Z	FLGHT400
AIRLINE	FLGHT400	AIRLINE	FLGHT400	U	AIRLNM	1A	1A	AIRLINE	FLGHT400
AIRLINE	FLGHT400	AIRLINE_L	FLGHT400	U	AIRLIN	1A	1A	AIRLINE_L	FLGHT400
CUSTOMERS	FLGHT400			D	CUSTO00001	2A	2A	CUSTORD	FLGHT400
CUSTOMERS	FLGHT400			D	ORDER00001	2A	2A	ORDDATE	FLGHT400
CUSTOMERS	FLGHT400	CUSTNAME	FLGHT400	D	CUSTO00002	1A	1A	CUSTNAME	FLGHT400
CUSTOMERS	FLGHT400	CUSTOMER	FLGHT400	D	CUSTO00001	1D	1D	CUSTOMER	FLGHT400
CUSTOMERS	FLGHT400	CUSTOMRZ	FLGHT400	D	CUSTO00001	1A	1A	CUSTOMRZ	FLGHT400
CUSTOMERS	FLGHT400	CUSTORD	FLGHT400	D	CUSTO00002	1A	1A	CUSTORD	FLGHT400
CUSTOMERS	FLGHT400	ORDDATE	FLGHT400	D	DEPAR00001	1A	1A	ORDDATE	FLGHT400
CUSTOMERS	FLGHT400	ORDNAME	FLGHT400	D	ORDER00001	1A	1A	ORDNAME	FLGHT400
CUSTOMERS	FLGHT400	Q_FLGHT400_CUSTOMERS_CUSTO00000	FLGHT400	U	CUSTOMER_NO	1A	1A	Q_FLGHT400_CUSTOMERS_CUSTO00001_00001	FLGHT400
FLIGHTS	FLGHT400			D	ARRIVAL	2A	2A	FLIGHTS_L	FLGHT400
FLIGHTS	FLGHT400			D	DAY_000001	3A	3A	FLIGHTS_L	FLGHT400
FLIGHTS	FLGHT400			D	FLIGH00001	4A	4A	FLIGHTS_L	FLGHT400
FLIGHTS	FLGHT400	FLIGHTS	FLGHT400	U	FLIGH00001	1A	1A	FLIGHTS	FLGHT400
FLIGHTS	FLGHT400	FLIGHTS_L	FLGHT400	D	DEPARTURE	1A	1A	FLIGHTS_L	FLGHT400
FLIGHTS	FLGHT400	FLIGHTS_Z	FLGHT400	D	FLIGH00001	1A	1A	FLIGHTS_Z	FLGHT400
FRCITY	FLGHT400	FRCITY	FLGHT400	U	FRCNAM	1A	1A	FRCITY	FLGHT400
FRCITY	FLGHT400	FRCITY_L	FLGHT400	U	FRCINT	1A	1A	FRCITY_L	FLGHT400

Figure 3: Results of Generate Index Listing Utility

If you are reverse engineering a single table, you might consider using the Index Evaluator feature that is available with the i5/OS V5R3 version of iSeries Navigator. This feature will show you the list of keyed logical files and indexes for a selected table. (See Figure 4.) To see the logical files using this method, do the following:

1. From iSeries Navigator, expand **Schemas**.
2. Select and expand the relevant schema.
3. Select **Tables** as shown in Figure 4.

SQL Name	Partitioned	Owner	Last Changed	Short Name	Text
AGENTS	No	COBBG	11/29/05 5:55:20 PM	AGENTS	Airline Agents
AIRLINE	No	COBBG	11/29/05 5:55:38 PM	AIRLINE	Airline Table for validation
CUSTOMERS	No	COBBG	11/29/05 5:56:14 PM	CUSTOMERS	Airline Customers
DDSSRCUNSP	No	QSECOFR	9/12/05 7:26:04 AM	DDSSRCUNSP	
FLIGHTS	No	COBBG	11/29/05 5:55:48 PM	FLIGHTS	Flight schedule
FNAME	No	COBBG	11/29/05 5:55:50 PM	FNAME	First Name Table for building Customer file
FNEXTO	No	COBBG	11/29/05 5:55:52 PM	FNEXTO	Next Order Number
FRFCITY	No	COBBG	11/29/05 5:55:53 PM	FRFCITY	City table for building Flights (From City)
LNAME	No	COBBG	11/29/05 5:55:54 PM	LNAME	Last Name Table for building Customer file
ORDERS	No	COBBG	11/29/05 5:55:56 PM	ORDERS	Airline Orders
QCLPSRC	No	QDFTOWN	9/12/05 7:26:05 AM	QCLPSRC	Flight Application/Build Specifications
QCMSDRC	No	QDFTOWN	9/12/05 7:26:05 AM	QCMSDRC	
QDSSRCD	No	QDFTOWN	9/12/05 7:26:05 AM	QDSSRCD	Flight Display File Specifications
QDSSRCENH	No	QSECOFR	9/12/05 7:26:05 AM	QDSSRCENH	Flight Display File Specifications
QDSSRCF	No	QDFTOWN	9/12/05 7:26:05 AM	QDSSRCF	Flight Application File Specifications
QMNUSRC	No	QDFTOWN	9/12/05 7:26:05 AM	QMNUSRC	Flight Application Menu Specifications
QMNUSRCENH	No	QSECOFR	9/12/05 7:26:05 AM	QMNUSRCENH	Flight Application Menu Specifications
QRPGLSRC	No	QSECOFR	9/12/05 7:26:05 AM	QRPGLSRC	The source for RPGLE
QRPGSRC	No	QDFTOWN	9/12/05 7:26:05 AM	QRPGSRC	Flight Application/Build Specifications
QSQLSRC	No	QSECOFR	9/12/05 7:26:05 AM	QSQLSRC	
RPGSRC	No	QDFTOWN	9/12/05 7:26:05 AM	RPGSRC	
RPGSRCUNSP	No	QSECOFR	9/12/05 7:26:05 AM	RPGSRCUNSP	
TOCITY	No	COBBG	11/29/05 5:55:57 PM	TOCITY	City table for building Flights (To City)

Figure 4: List of tables

4. Select the desired table from the list displayed.

5. From the right-click menu, click **Indexes** as shown in Figure 5.

SQL Name	Partitioned	Owner	Last Changed	Short Name	Text
AGENTS	No	COBBG	11/29/05 5:55:20 PM	AGENTS	Airline Agents
AIRLINE	No	COBBG	11/29/05 5:55:38 PM	AIRLINE	Airline Table for validation
CUSTOMI	No	COBBG	11/29/05 5:56:14 PM	CUSTOMERS	Airline Customers
DDSSRCL	No	QSECOFR	9/12/05 7:26:04 AM	DDSSRCUNSP	
FLIGHTS	No	COBBG	11/29/05 5:55:48 PM	FLIGHTS	Flight schedule
FNAME	No	COBBG	11/29/05 5:55:50 PM	FNAME	First Name Table for building Customer file
FNEXTO	No	COBBG	11/29/05 5:55:52 PM	FNEXTO	Next Order Number
FRCITY	No	COBBG	11/29/05 5:55:53 PM	FRCITY	City table for building Flights (From City)
LNAME	No	COBBG	11/29/05 5:55:54 PM	LNAME	Last Name Table for building Customer file
ORDERS	No	COBBG	11/29/05 5:55:56 PM	ORDERS	Airline Orders
QCLPSRC	No	QDFTOWN	9/12/05 7:26:05 AM	QCLPSRC	Flight Application/Build Specifications
QCMDSRC	No	QDFTOWN	9/12/05 7:26:05 AM	QCMDSRC	
QDDSSRC	No	QDFTOWN	9/12/05 7:26:05 AM	QDDSSRC	Flight Display File Specifications
QDDSSRC	No	QSECOFR	9/12/05 7:26:05 AM	QDDSSRCENH	Flight Display File Specifications
QDDSSRC	No	QDFTOWN	9/12/05 7:26:05 AM	QDDSSRCF	Flight Application File Specifications
QMNUSR	No	QDFTOWN	9/12/05 7:26:05 AM	QMNUSRC	Flight Application Menu Specifications
QMNUSR	No	QSECOFR	9/12/05 7:26:05 AM	QMNUSRCENH	Flight Application Menu Specifications
QRPGLS	No	QSECOFR	9/12/05 7:26:05 AM	QRPGLSRC	The source for RPGL
QRPGSRC	No	QDFTOWN	9/12/05 7:26:05 AM	QRPGSRC	Flight Application/Build Specifications
QSQLSRC	No	QSECOFR	9/12/05 7:26:05 AM	QSQLSRC	
RPGSRC	No	QDFTOWN	9/12/05 7:26:05 AM	RPGSRC	
RPGSRCUNSP	No	QSECOFR	9/12/05 7:26:05 AM	RPGSRCUNSP	
TOCITY	No	COBBG	11/29/05 5:55:57 PM	TOCITY	City table for building Flights (To City)

Figure 5: Select Indexes

Figure 6 presents the list of available keyed logical files (indexes) for that table. You can scroll to the right to see more information about the indexes.

SQL Name	Type	Schema	Short Name	Text	INDEX PARTITION	VALID	CREATION DATE	LAST BUILD	LAST QU
AGENTS	PHYSICAL FILE	FLGHT400	AGENTS	Airline Agents	AGENTS	Yes	2005-11-29 17:5...	2005-11-29 17:5...	
AGENTSZ	LOGICAL FILE	FLGHT400	AGENTSZ	Airline Agents Binary converted to Zoned	AGENTSZ	Yes	2005-11-30 10:0...	2005-11-30 10:0...	
AGENTSZ	LOGICAL FILE	FLGHT400	AGENTSZ	Airline Agents by Name	AGENTSZ	Yes	2005-11-30 10:0...	2005-11-30 10:0...	

Figure 6: List of indexes for selected table

Note: This method will show you the list of indexes that the SQL optimizer evaluates when SQL accesses the table.

Step 3: Building SQL DDL scripts

In this section, the SQL DDL scripts will be built to perform the conversion.

Building scripts to build tables

To reverse engineer the physical files or tables from DDS to SQL DDL, use the **Generate SQL** feature in iSeries Navigator:

6. In iSeries Navigator, select the schema **FLGHT400** and then choose **Tables**. The main window presents a list of the tables in the FLGHT400 schema.
7. The “Identifying tables” section defines a query that produces a list of seven physical files to reverse engineer. Select the seven files, right-click, and choose **Generate SQL**, as shown in Figure 7. (**Note:** Use the **Ctrl** key to choose multiple tables.)

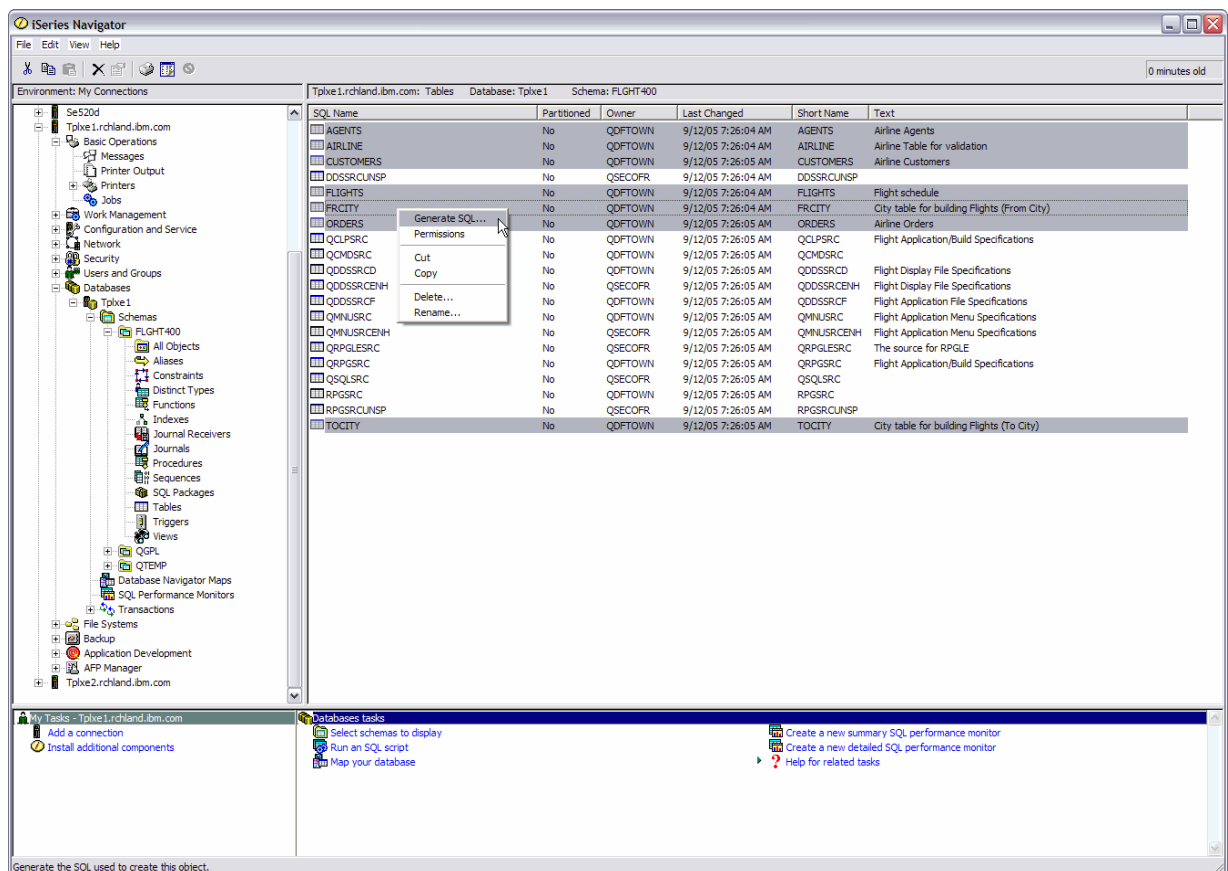


Figure 7: Select tables to reverse engineer

8. The Generate SQL window appears (Figure 8).

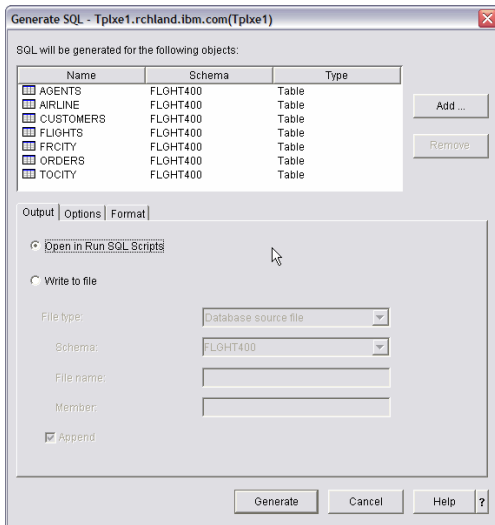


Figure 8: Generate SQL window

9. Select **Open** in Run SQL Scripts and click **Generate**.

10. The RUN SQL window appears with the generated SQL statements (Figure 9).

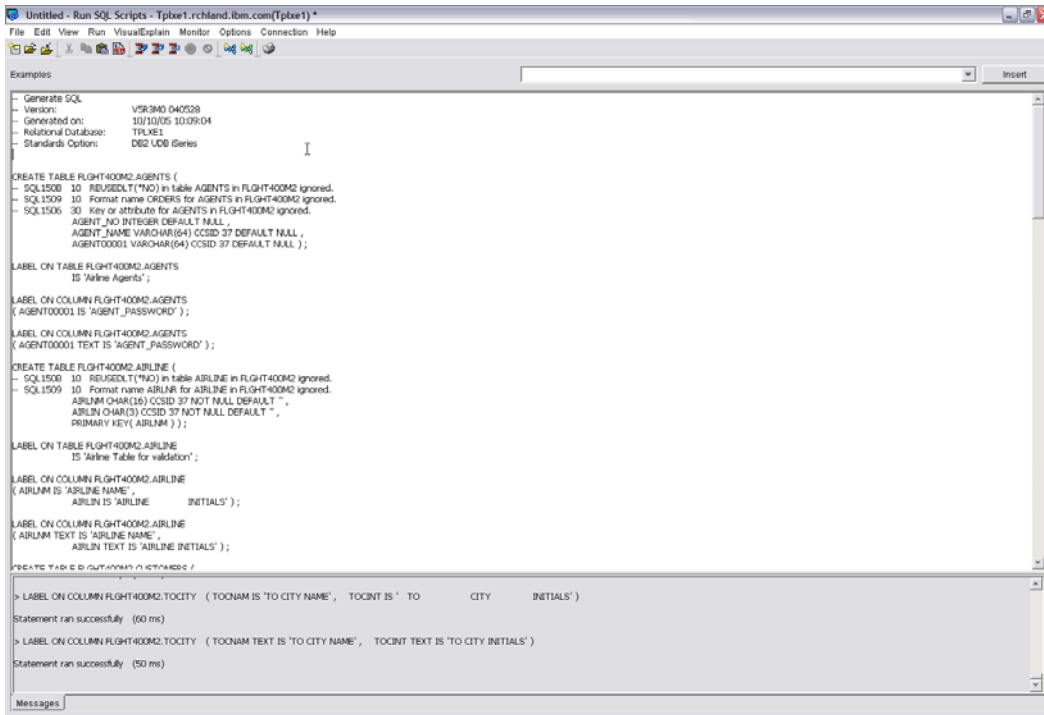


Figure 9: Generated SQL statements

11. From the toolbar menu, click **File > Save**.

12. Specify file name **Create Tables.SQL** and press **Save**.

Note: The Generate SQL feature uses the Generate DDL (QSQGNDDL) API to retrieve the SQL source from System Catalogs.

Building the script to create the indexes

This section shows you how to build the SQL script to create the indexes into the new schema.

Note: It is possible to utilize the iSeries Navigator Generate SQL feature to produce the DDL that creates the tables. This feature is not used against the existing logical files to generate the DDL for the indexes because this technique will not yield the desired results. For logical files, iSeries Navigator generates DDL to create an SQL view instead of an index. However, one of the parameters of the QSQGNDDL API is **database object type**. This parameter specifies the type of the database object or object attribute for which DDL is generated. If a program is written to call QSQGNDDL, specifying the logical file and "INDEX" for the database object-type parameter, then the DDL to create the Index will be generated. For complete information on the QSQGNDDL API and its parameters, refer to the iSeries Information Center Web site, found in the **Resources** section.

In the previous section, "Identifying Indexes," SQL procedures were built and run to produce a list of the indexes to be created. These indexes were based on the keyed logical and physical files that already existed in the FLGHT400 application. After examining this list, it was evident that several indexes could be eliminated because they were duplicates. Given the result, the DDL script was manually generated to create the necessary indexes. Refer to Appendix A, "SQL DDL script to create the indexes" for an illustration of this script.

To build the Create Indexes script, take the following actions:

1. Open a new Run SQL Scripts window.
2. Type the following statements:

```
SET CURRENT SCHEMA flght400m2;
CREATE INDEX agents_ix1
  ON agents ( agent_no ASC ) ;
CREATE INDEX agents_ix2
  ON agents ( agent_name ASC ) ;
CREATE INDEX airline_ix1
  ON airline ( airlin ASC ) ;
CREATE INDEX customers_ix1
  ON customers ( cust_name ASC ) ;
CREATE INDEX customers_ix2
  ON customers ( cust_no DESC ) ;
CREATE INDEX FLIGHTS_IX1
  ON flights ( departure ASC, arrival ASC, day_week ASC, flight_no ASC ) ;
CREATE INDEX flights_ix2
  ON flights ( flight_no ASC ) ;
CREATE INDEX frcity_ix1
  ON frcity ( frcint ASC ) ;
CREATE INDEX orders_ix1
  ON orders ( order_no ASC ) ;
CREATE INDEX tocity_ix1
  ON tocity ( tocint ASC ) ;
```

3. From the toolbar menu, click **File >Save**.
4. Specify file name **Create Indexes.SQL** and press **Save**.
5. Close the Run SQL Scripts window.

Reviewing the SQL DDL scripts

Prior to executing the SQL DDL scripts, review the code and make any necessary modifications.

Changing the target schema

FLGHT400M2 will be selected as the new target schema name. This schema will contain the reverse engineered SQL objects as well as any programs, service programs, and all other objects created or updated as a result of this database modernization effort. As such, it is necessary to modify the SQL DDL scripts so that the new objects are created in this new target schema.

Rather than hard coding the target schema into each CREATE statement, the current schema can be set. Using this technique, all unqualified objects will be created in the specified current schema.

1. Open a new Run SQL Scripts window.
2. From the toolbar menu, click **File > Open**.
3. Select the **CREATE TABLES.SQL** file.
4. Add the following line to the very top of the script (the first statement of the script):

```
SET CURRENT SCHEMA FLGHT400M2;
```
5. From the toolbar menu, click **File > Save**.
6. Repeat steps one through five for file **CREATE INDEXES.SQL**.
7. Close the Run SQL Scripts window.

Adding columns

The reverse engineering process might be an opportune time to add new columns to the tables. To the CUSTOMERS table, new columns were added for Address, City, State, Zip Code, and Telephone number by performing the following steps:

1. Open a new Run SQL Scripts window.
2. From the toolbar menu, click **File > Open**.
3. Select the **CREATE TABLES.SQL** file.
4. Change the CREATE TABLE CUSTOMERS statement as shown:

```
CREATE TABLE CUSTOMERS (  
  CUSTOMER_NO FOR COLUMN CUST_NO INTEGER DEFAULT NULL ,  
  CUSTOMER_NAME FOR COLUMN CUST_NAME VARCHAR(64) CCSID 37 DEFAULT NULL ,  
  ADDRESS VARCHAR(150) CCSID 37 NOT NULL DEFAULT ' ' ,  
  CITY CHAR(50) CCSID 37 DEFAULT NULL ,  
  STATE CHAR(2) CCSID 37 DEFAULT NULL ,  
  ZIPCODE CHAR(9) CCSID 37 DEFAULT NULL ,  
  TELEPHONE CHAR(20) CCSID 37 DEFAULT NULL ,  
  CREDIT_CARD FOR COLUMN CRED_CARD CHAR(30) CCSID 37 DEFAULT NULL ,  
  CC_NUMBER CHAR(20) CCSID 37 DEFAULT NULL ,  
  EXP_DATE CHAR(20) CCSID 37 DEFAULT NULL ,  
  PREF_AIRLINE_ID FOR COLUMN PREF_AIRLN CHAR(10) CCSID 37 DEFAULT NULL ,  
  FF_NUMBER CHAR(20) CCSID 37 DEFAULT NULL ) ;
```

5. From the toolbar menu, click **File > Save**.
6. Close the Run SQL Scripts window.

Other considerations

Additionally, you will need to determine whether the following issues are relevant.

Record format name

When a table is created using SQL, the table name and the record format name will be the same. This can be problematic because it will usually result in errors when you attempt to recompile high-level language (HLL) programs using record-level access. However, if your HLL programs use embedded SQL, this is not an issue. Because the data access is being converted to embedded SQL in the next stage, this paper does not address this topic. If you are interested in the options available to work around this issue, refer to the Redbook, "Modernizing Data Access."

Managing SQL DDL

Traditionally, the source for database objects (physical and logical files) are stored in source file members, and the objects are created by issuing the control language (CL) command CRTPF and CRTLF against those members. Managing SQL DDL scripts is done similarly. The scripts can also be stored in source file members, but to create the objects, issue the CL command RUNSQLSTM (instead of CRTPF/CRTLF) against the source file members. Some other things to consider regarding SQL DDL management are:

- If you are using change management tools that are not specific to the iSeries platform, store the SQL DDL scripts in PC or integrated file system (IFS) files.
- If the SQL DDL source is misplaced or deleted, you can use the Generate Data Definition Language (QSQGNDDL) API to retrieve the SQL source from System Catalogs (SYSIBM and QSYS2).

Creating the new SQL objects

In this section, the new target schema will be created. The generated SQL scripts will also be run to build the new SQL tables and indexes.

Creating the new DB2 UDB schema (collection) on the iSeries system

At this point, the SQL scripts for reverse engineering the FLGHT400 database objects are ready. But before executing them, you must determine where the new objects will be created. The new SQL objects will be built in a new schema. On DB2 UDB for iSeries, a schema is used to group related database objects. When a CREATE SCHEMA statement executed, the following objects are created:

- OS/400 library
- OS/400 journal and journal receiver
- DB2 UDB views that contain a subset of the information in system-wide catalog views

As mentioned earlier, as the container for this new version of FLGHT400, the new schema name will be FLGHT400M2. To create the schema, take the following steps:

1. Open a new Run SQL Scripts window.
2. Type the following statement: `CREATE SCHEMA FLGHT400M2;`
3. From the toolbar menu, click **Run > All**.

4. Verify that the script runs successfully by checking the results in **Run History**. (The bottom pane of the Run SQL scripts window will show the “Statement ran successfully” message.)
5. Close the Run SQL Scripts window.

The CREATE SCHEMA statement will automatically enable journaling for all tables created in the specified schema. If journaling of the tables is not desired, you can turn this feature off by deleting the journal QSQJRN in the schema. It is important to do this before any database objects are created (and subsequently automatically journaled) in the schema. If database objects have been created and are being journaled, you must end journaling for those objects before the journal object can be deleted.

To delete the QSQJRN journal, take the following actions :

1. From iSeries Navigator select Schema FLGHT400M2
2. Select Journals
3. Select QSQJRN
4. From the right-click menu, select Delete (Figure 10: Delete Journal).

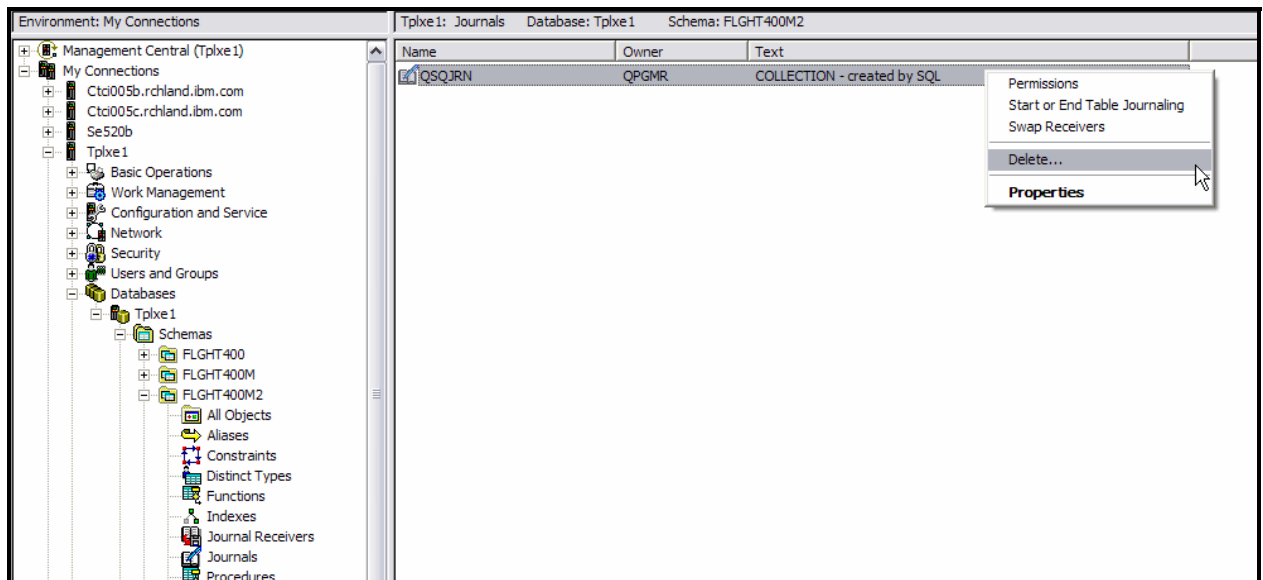


Figure 10: Delete Journal

Creating the tables

Now that you have built the new schema and updated the scripts, it is time to execute them and create the database objects. First, create the new tables by taking the following actions:

1. Open a new Run SQL Scripts window.
2. From the toolbar menu, click **File > Open**.
3. Select the **CREATE TABLES.SQL** file.
4. From the toolbar menu, click **Run > All**.
5. Verify that the script ran successfully by checking the results in **Run History**. (The bottom pane of the Run SQL scripts window will show the “Statement ran successfully” message.)
6. Close the Run SQL Scripts window.

Creating the indexes

Once the tables have been created, the indexes can be built over them.

Take the following actions to create the new indexes:

1. Open a new Run SQL Scripts window.
2. From the toolbar menu, click **File > Open**.
3. Select the **CREATE INDEXES.SQL** file.
4. From the toolbar menu, click **Run > All**.
5. Verify that the script ran successfully by checking the results in **Run History**. (The bottom pane of the Run SQL scripts window will show the “Statement ran successfully” message.)
6. Close the Run SQL Scripts window.

Copying data to new schema

Once the new tables have been created in the new schema, they will obviously be empty. The next step is to copy the production data to the new tables. The following SQL statements will copy rows from tables in the FLGHT400 schema to the reverse engineered tables in FLGHT400M2:

1. Open a new Run SQL Scripts window.
2. Type the following statements:

```
SET TRANSACTION ISOLATION LEVEL NO COMMIT;
INSERT INTO FLGHT400M2.AGENTS
  SELECT * FROM FLGHT400.AGENTS;
INSERT INTO FLGHT400M2.ORDERS
  SELECT * FROM FLGHT400.ORDERS;
INSERT INTO FLGHT400M2.FLIGHTS
  SELECT * FROM FLGHT400.FLIGHTS;
INSERT INTO FLGHT400M2.FRCITY
  SELECT * FROM FLGHT400.FRCITY;
INSERT INTO FLGHT400M2.TOCITY
  SELECT * FROM FLGHT400.TOCITY;
INSERT INTO FLGHT400M2.AIRLINE
  SELECT * FROM FLGHT400.AIRLINE;
INSERT INTO FLGHT400M2.CUSTOMERS
  (CUSTOMER_NO,
  CUSTOMER_NAME,
  CREDIT_CARD,
  CC_NUMBER,
  EXP_DATE,
  PREF_AIRLINE_ID,
  FF_NUMBER)
  SELECT CUSTOMER_NO,
  CUSTOMER_NAME,
  CREDIT_CARD,
  CC_NUMBER, EXP_DATE,
  PREF_AIRLINE_ID,
  FF_NUMBER
  FROM FLGHT400.CUSTOMERS;
```

3. From the toolbar menu, click **Run > All**.
4. Verify that the script ran successfully.
5. Close the Run SQL Scripts window.

Note: Recall the addition of new columns to the CUSTOMERS table in the new schema. Because of this, the SQL statement that copies rows from table CUSTOMERS to the new schema looks a little different.

Stage 2: Creating I/O modules to access data

This section describes the process of designing, creating, and using prototyped procedures to provide an encapsulated database interface to the data. This form of data encapsulation provides multiple benefits.

- Maintenance is simplified. If changes are made to a table, only one service program needs to be updated or recreated. Some instances do not need to be found in the application that is reading, writing, or updating a particular table and performing the necessary maintenance.
- There is more consistency with the object-oriented (OO) approach to data access and manipulation.
- Procedures can be wrapped as stored procedures and called from client programs.

Think of this approach as creating an API layer to your database. If you want to force all database access to be performed through this interface, you can do so by restricting access to your database and giving users access to these procedures.

In addition, the data access method that is used in the I/O procedures is embedded SQL. The advantages of using SQL over record-level access (RLA) include the following:

- SQL is the iSeries platform's strategic direction for database development and data access.
- Programs that utilize embedded SQL do not have format-level check issues.
- The SQL optimizer determines the access plans, which frees you from this responsibility.

The following is a list of steps involved in this process:

1. Identify all programs and modules that are to be converted.
2. Identify all instances of RLA in the programs identified in step 1.
3. Document the business rule of the RLA.
4. Create a view to access the data.
5. Create an I/O procedure for each unique RLA instance or group.
6. Replace each RLA instance or group with a call to new I/O procedure.
7. Create the I/O module.
8. Create the I/O service program.
9. Recompile existing modules, programs, and service programs.

Note: In this database modernization exercise, stages 1 and 2 were implemented in a relatively short period of time. This means that after files are reverse engineered, programs that use the files to incorporate SQL access were immediately converted, and the transition period between the two stages did not present a concern. Obviously, this simplifies the process significantly as the effort of making sure the native RLA interfaces still work with the new SQL database object is eliminated.

In a real life application, it is rather impractical to expect to carry out stages 1 and 2 in this manner. Typically, after the files are reverse engineered, existing application programs must be able to continue using the re-engineered database objects without requiring immediate changes to the application. Stages 2 and beyond can then be carried out at the desired pace. The IBM Redbook "Modernizing IBM eServer iSeries Application Data Access - A Roadmap Cornerstone" provides details on the methods and techniques of managing this transition period. (Appendix C has a listing of the IBM Redbooks Web site.)

Step 1: Identifying programs and modules to convert

The goal is to replace all RLA operations; therefore, the first step entails finding all programs and modules that contain those I/O operations. This can be accomplished using a variety of techniques.

- Manually check each source file member for F specs (file declarations).
- Use Programming Development Manager (PDM) to scan the source file members for each table.
- Develop your own analysis utility.
- Use a third-party analysis tool (such as: Databorough X-Analysis or Hawkeye Pathfinder™) to perform impact analyses and to identify objects for conversion.

Note: See Appendix C for a Web listing for the IBM Redpaper entitled “Modernizing and improving the maintainability of RPG applications using x-Analysis Version 5.6.”

To obtain the list of programs and modules with RLA access that need to be changed, an analysis utility was created to perform the following:

1. Using the DSPPGMREF command the utility creates a work file with all programs and service programs in the FLGHT400M library and their file references. Notice that it is the FLGHT400M library and not the original library, FLGHT400. The reason for this is that, during the application modularization process, all RLA operations were moved to the new library.
2. Using the DSPOBJD command, the utility creates another work file of all files in the FLGHT400 library (the library that contains the physical and logical files for FLGHT400 Version 2).
3. The utility executes an SQL statement to display only programs/service programs that reference either physical or logical files.
4. The Run SQL Script window was opened and the following statements were executed:

```
CL: DSPPGMREF PGM(FLGHT400M/*ALL) OUTPUT(*OUTFILE) OBJTYPE(*ALL)
OUTFILE(QTEMP/PGMREFFILE);
CL: DSPOBJD OBJ(FLGHT400/*ALL) OBJTYPE(*FILE) OUTPUT(*OUTFILE)
OUTFILE(QTEMP/ALLFILES);
SELECT WHPNAM,WHFNAM, WHFUSG, WHRFNM FROM
qtemp.pgmreffile WHERE whotyp = '*FILE' and whfnam in (SELECT odobnm FROM
qtemp.allfiles WHERE odobat = 'PF' OR odobat = 'LF')
```

5. The query yielded the following list of programs and modules to convert (Table 1):

Program Name	File Name	Record Usage	Format	SRVPGM=V, PGM=P, MODULE=M
NFS001	ORDERSZ	7	ORDNMZ	V
NFS001	CUSTORD	1	CUSNMR	V
NFS001	ORDDATE	1	ORDDTR	V
NFS400	FRCITY	1	FRCTYR	V
NFS400	TOCITY	1	TOCTYR	V
NFS400	FRCITYL	1	FRCTYL	V
NFS400	TOCITYL	1	TOCTYL	V
NFS400	FLIGHTSL	1	FLGHTR	V
NFS400	FLIGHTSZ	1	FLGHTR	V
NFS400	CUSTNAME	1	CUSTR	V
NFS400	CUSTOMER	3	CUSTR	V
NFS400	CUSTOMRZ	1	CUSTZ	V
NFS400	CUSTOMERS	1		V

Table 1: Query of program and modules to convert

In an ILE setting, keep in mind that the underlying module objects (used to build the programs and service programs) might have been compiled into QTEMP or deleted from the library (as they are no longer needed after the programs and service programs have been created). In these cases, you must interrogate the programs and service programs further to determine the modules that need converting.

Step 2: Identifying instances of RLA

Next, locate all instances of RLA in the programs and modules identified in step 1. The following list contains all the RPG operations that perform record-level access:

- SETLL
- SETGT
- READ
- READP
- READE
- READEP
- CHAIN
- UPDAT and UPDATE
- DELET and DELETE
- WRITE

In addition, you can check for the following operations and built-in functions that, when used, often accompany RLA operations:

- OPEN
- CLOSE
- %EOF
- %OPEN
- %FOUND
- %EQUAL

For each of the objects identified in step 1, scan the corresponding source member for all of the listed RLA operations. Another technique of locating all RLA operations is to scan the source for the

filename or record format name. For locating the desired search string, you have a couple of options depending on your editor of choice:

- Find String feature (F14) of Source Entry Utility (SEU) or Find String (option 25) of the PDM.
- Find feature (Ctrl-F) of WDSCs LPEX editor.

Step 3: Documenting the business rule

After the RLA operations have been identified, the next step is to determine if each operation acts independently or if multiple RLA operations can be grouped together to form a single business rule. Keep in mind that the ultimate goal is to replace the identified RLA operation or group of operations with a call to a single I/O procedure that uses embedded SQL for the data access. To accomplish this, you must fully understand the business rule behind the RLA operations as well as have a good comprehension of SQL data-access methods.

RLA operations are not necessarily mapped one-to-one with an I/O procedure. For example, a single SQL SELECT statement can replace a SETLL operation followed by a READ operation. As another example, a single SELECT statement (using join syntax) can replace a series of consecutive CHAIN operations (drilling down through a set of normalized files). This is why it is important to understand and document the business rule behind each RLA operation.

To emphasize this point, some examples follow. (**Note:** These examples are not part of the FLGHT400 database modernization exercise):

Example 1:

Calculate the number of records/rows with a departure date of 05/18/2004 (Table 2).

RLA version	SQL version
<pre> /free countRet = 0; findDate = %date ('051804' : *mdy0); setll 1 orders; dou %error; read(e) orders; deparDate = %date (depar_date); if %eof; leave; endif; if deparDate = findDate; countRet = countRet + 1; endif; enddo; /end-free </pre>	<pre> /free countRet = 0; findDate = %date ('051804' : *mdy0); /end-free C/EXEC SQL c+ SELECT count(*) INTO :countRet c+ FROM orders c+ WHERE DATE(departure_date) = :findDate C/END-EXEC /free /end-free </pre>

Table 2: Departure date examples in RLA and SQL

In the RLA version, the program logic loops through and reads each record the ORDERS file. For each record that has a departure date of 05/18/2004, the count variable is incremented.

In the SQL version, notice that there is no looping in the program logic. Instead, a single embedded SQL statement calculates the number of orders whose departure date is 05/18/2004.

Example 2:

Change the status field to **CANCELED** for all records with a departure date of 05/18/2004 (Table 3).

RLA version	SQL version
<pre>/free findDate = %date ('051804' : *mdy0); setll 1 orders; dou %error; read(e) orders; deparDate = %date (depar date); if %eof; leave; endif; if deparDate = findDate; status = "Canceled"; update orderRec; endif; enddo; /end-free</pre>	<pre>/free countRet = 0; findDate = %date ('051804' : *mdy0); /end-free C/EXEC SQL c+ UPDATE c+ FROM orders c+ SET status = 'Canceled' c+ WHERE DATE(departure_date) :findDate C/END-EXEC /free /end-free</pre>

Table 3: RLA and SQL versions of **CANCELED** in status field

Again, in the RLA version, the program logic loops through and reads each record in the ORDERS file. For each record that has a departure date of 05/18/2004, the value of the STATUS field is changed to **CANCELED** and the record is updated.

In the SQL version, one statement finds all of the rows with a departure date of 05/18/2004 and sets the STATUS field to **CANCELED**.

Example 3:

Search through five tables to return data for the specified time period of the sixth month of 1998. For the SQL version, the following SQL view is created with a join query, as shown in Table 4:

```
CREATE VIEW joinView
  (year, month, orderdate, country,
   customer, part, supplier, quantity, revenue)
AS SELECT
  t.year,t.month,i.orderdate,c.country,c.customer,
  p.part,s.supplier,i.quantity,i.revenue_wo_tax
FROM item_fact i
INNER JOIN part_dim p ON (i.partkey =p.partkey)
INNER JOIN time_dim t ON (i.orderdate=t.datekey)
INNER JOIN cust_dim c ON (i.custkey=c.custkey)
INNER JOIN supp_dim s ON (i.suppkey=s.suppkey);
```


RLA version	SQL version
<pre> SearchKey KList Kfld SearchYear Kfld SearchMonth Times Occur Result_Set /free SearchYear = 1998; SearchMonth = 6; Setll SearchKey TIME_DIML1; if %found; DOU RowsReq= RowsRd; READ TIME_DIML1; If %EOF; Leave; Endif; Setll DATEKEY ITEMFACTL1; If %FOUND; DOU RowsReq = RowsRd; READE DATEKEY ITEMFACTL1; If %EOF; Leave; Endif; CHAIN PARTKEY PART_DIML1; If Not %FOUND; Iter; Endif; CHAIN CUSTKEY CUST_DIML1; If Not %FOUND; Iter; Endif; CHAIN SUPPKEY SUPP_DIML1; If Not %FOUND; Iter; Endif; RowsRd = RowsRd + 1 Enddo; Endif; Enddo; Endif; /end-free </pre>	<pre> Times Occur result_set C/EXEC SQL C+ DECLARE sql_jn CURSOR FOR C+ SELECT * FROM JoinView C+ WHERE year=1998 AND month=6 C/END-EXEC C/EXEC SQL C+ OPEN sql_jn C/END-EXEC C/EXEC SQL C+ FETCH NEXT FROM sql_jn FOR C+ :RowsReq ROWS INTO :result_set C/END-EXEC * SQLER3 contains the number of rows fetched. /free If SQLCOD = 0 and SQLER5 = 100 and SQLER3 > 0; RowsRd = SQLER3; Endif; /end-free </pre>

Table 4: SQL view with a join query

Notice how SQL performs the search in one request. In contrast, the RLA program performs READ operations against each table individually and must specify a logical file to use. Though you can write SQL code to access each table separately, this is not recommended to mirror the RLA code (for performance reasons). The real value of SQL is processing a set of data on a single request and not subdividing a single request into multiple parts. Additionally, notice how the SQL version is performing a blocked fetch into a multi-occurrence data structure to process multiple rows in one statement.

Keep in mind that you can simplify the RLA version by creating a join logical file. However, SQL views have an advantage over join logical files in that the join order is not fixed and more complex join types are available. The query optimizer has an opportunity to analyze the join request and modify the join order coded in the SQL view if it determines that a different join order will perform better. With join

logical files, the burden is on the programmer to code the join order correctly and, over time, continually check to ensure that the coded join order is still optimal.

Getting back to the FLGHT400 exercise, consider the GetCustNumber procedure:

```

/*****
p GetCustNumber  b          export
/*****
d GetCustNumber  pi
d  Name          64      const
d  Number        9B 0
d  Generate      1      const options(*nopass)
//
d NameV          s          64      varying
/free
NameV = %trim(Name);
chain(e) NameV CUSTR;
if not %found;
  if %parms > 2 and Generate = 'Y';
    dou not %error;
    setll *hival CUSTOMERR;
    read(e) CUSTOMERR;
    CNUMBR = CNUMBR + 1;
    CUSTNM = NameV;
    write(e) CUSTOMERR;
  enddo;
  CUSTNO = CNUMBR;
else;
  CUSTNO = -1;
endif;
endif;
Number = CUSTNO;
/end-free
p GetCustNumber  e

```

The purpose of the GetCustNumber procedure is to search the CUSTOMER file for a record that matches the specified customer name. If the record is found, the customer number is returned and the procedure ends. If the record is not found, a new one is inserted into the table. Before the insertion, the procedure must first generate a new key (the customer number field) for the new row. To do this, the procedure searches the CUSTOMERS file again, looking for the record with the highest current value of the **customer_number** field. That value is incremented by one and is used as the customer number field of the new row to be inserted. The value of the new **customer_number** field is then returned.

Now that you understand what the procedure does, document the business rule for each RLA operation or group of operations (Table 5):

RLA operation	Table accessed	Business rule
chain	CUSTOMERS	Search the CUSTOMERS table for a record that matches the specified customer name.
setll read	CUSTOMERS	Find the record with the highest value for the customer number field.
write	CUSTOMERS	Insert a new record/row into the CUSTOMERS table

Table 5: RLA operations business rules

Step 4: Creating the SQL view to access data

In this next step, the necessary SQL views used to access the data in the FLGHT400 tables are created. For this modernization exercise, the decision was made to allow only data access through views. None of the programs access the physical data model (the tables) directly. This is the ideal approach for database administrators (DBAs) to consider implementing. In this type of implementation, DBAs must determine what columns are eligible to be queried from both users and applications. After that analysis is completed, the DBA can then create views that project only those columns that users and applications need to see.

The following list contains some advantages of using SQL views:

- More flexibility in selecting and processing data:
 - CASE expression
 - Date/time functions
 - Grouping
 - Join processing
- Views can be opened by native programs as nonkeyed logical files
- Views can be directly accessed via ODBC/JDBC
- Enables database programmers to mask complexity of the database to users
- Provides a way to restrict access to the data in the table
- Views can be utilized to define an externally defined data structure in programs
- View structure can be used as the parameter to pass to or from the I/O procedures

With these advantages in mind, the new SQL I/O procedures will use views (instead of tables) to access the data. The following list in Table 6, taken from the previous step, is updated to include the chosen view to access the data:

RLA operation	Table	Business rule	New view
chain	CUSTOMERS	Search the CUSTOMERS table for a record that matches the specified customer name.	ALLCUSTS
setll read	CUSTOMERS	Find the record/row in table CUSTOMERS with the highest value for the customer number field.	None – replaced by Sequence object (Note)
write	CUSTOMERS	Insert a new record/row into the CUSTOMERS table	ALLCUSTS

Table 6: New RLA operation view

Note: The file access has been replaced with a sequence object. This is discussed in detail in the section “Implementing automatic key generation and unique identifiers.”

The remaining two documented business rules access the same table and thus can use the same SQL view. ALLCUSTS is the name of the new view. To create the new view, open a Run SQL Scripts window and run the following statement:

```
CREATE VIEW allCusts
(CUSTOMER_NO,
CUSTOMER_NAME,
ADDRESS,
CITY,
STATE,
ZIPCODE,
TELEPHONE,
CREDIT_CARD,
CC_NUMBER,
EXP_DATE,
PREF_AIRLINE_ID,
FF_NUMBER)
AS select * From customers;
```

Repeat this step for all documented RLA business rules.

Appendix A includes a script program, "SQL DDL script to create all of the views."

Step 5: Creating an I/O procedure for each unique RLA instance or group

The RLA operations that need to be replaced have been identified, and the views that the SQL I/O procedure will use to access the data have been created. The next step is to construct the I/O procedures.

Before you create the I/O procedures, complete the following activities to set the proper environment:

1. Create source file **QRPGLESRC** in schema FLGHT400M2
2. Add two new members to FLGHT400M2/QRPGLESRC:
 - NFSSQL**: Contains all of the I/O procedures
 - NFSSQLPR**: Contains all of the prototypes of the I/O procedures
3. Add one new member to FLGHT400M2/QSRVSRC (binder source):
 - NFSSQL**: Contains all EXPORT statements that identify the procedures to export from the NFSSQL service program

To set up the environment for the new I/O procedures, take the following actions:

4. Open the RSE perspective in a WebSphere Development Studio Client window.
5. From the **Toolbar** menu, click **File > New > iSeries Source Physical file** (Figure 11).

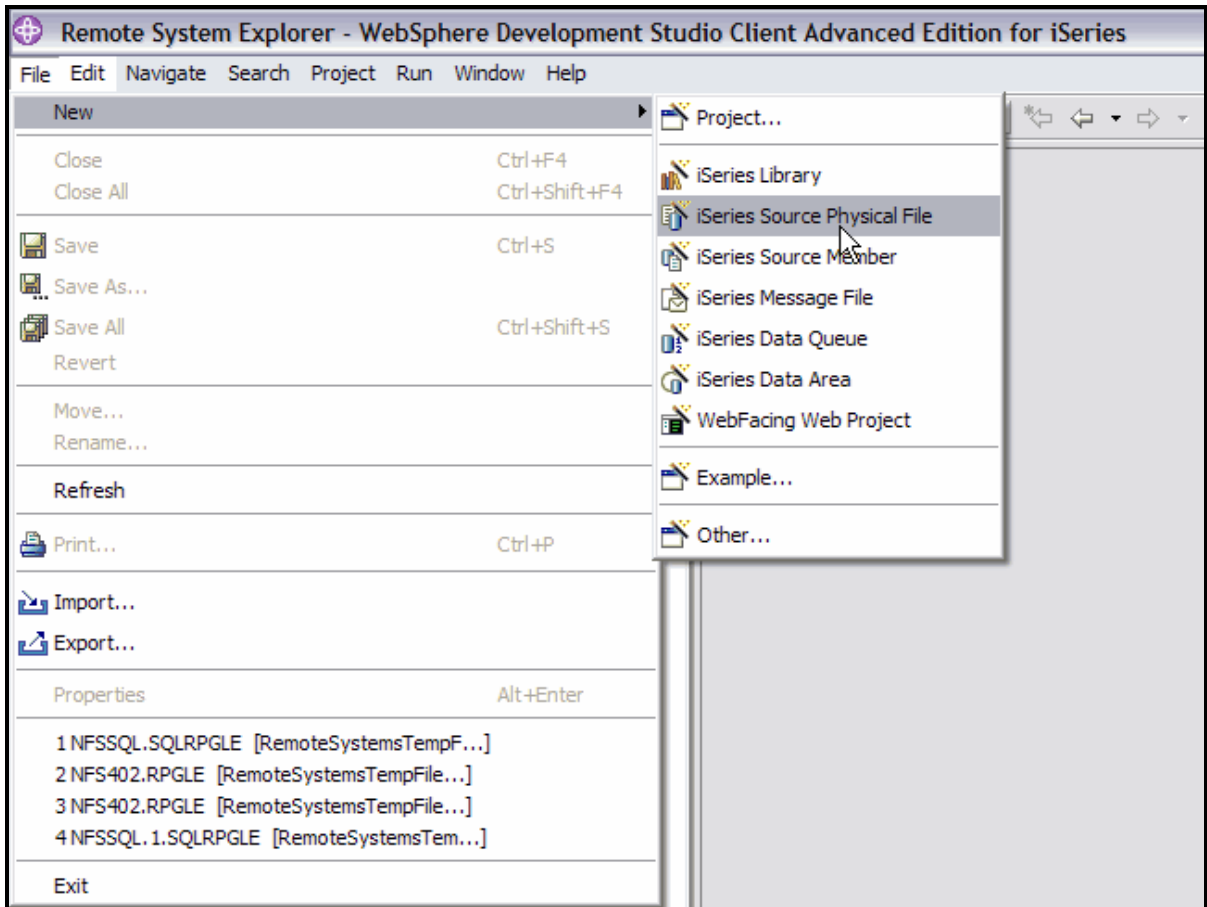


Figure 11: Create new source physical file

6. Specify **FLGHT400M2** for the library and **QRPGLESRC** for the file (shown in Figure 12).

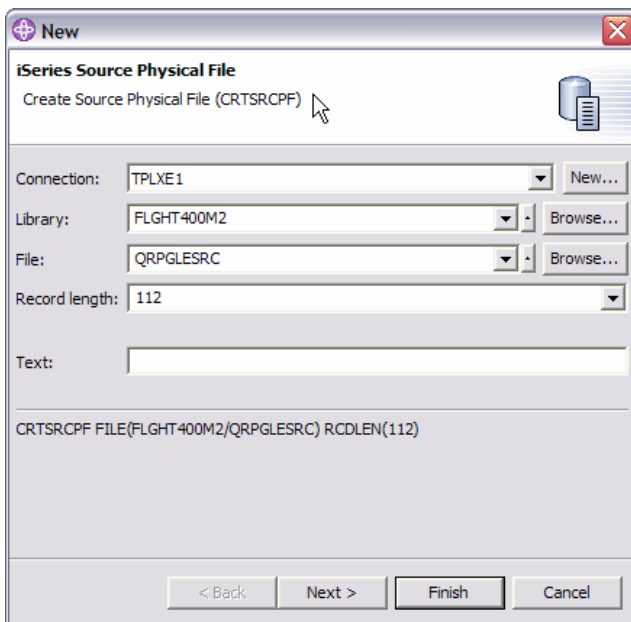


Figure 12: Specify source physical file attributes

7. Click **Finish**.
8. From the **Toolbar** menu, click **File > New > iSeries Source Member** (see Figure 13).

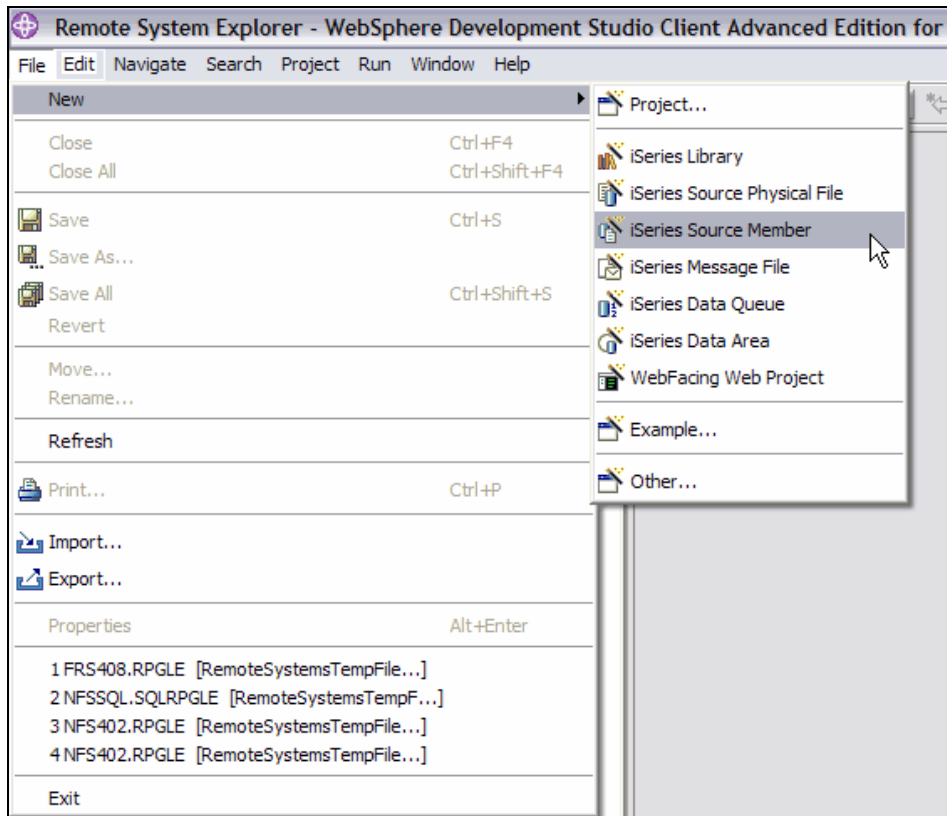


Figure 13: Create iSeries Source Member

- Specify **FLGHT400M2** for the library, **QRPGLESRC** for the file, **NFSSQLPR** for the member, **RPGLE** for the member type, and **I/O Procedure prototypes** for the text (Figure 14).

The screenshot shows a 'New' dialog box titled 'iSeries Source Member' with the subtitle 'Add Physical File Member (ADDPFM)'. It contains several input fields: 'Connection' (TPLXE1), 'Library' (FLGHT400M2), 'File' (QRPGLESRC), 'Member' (NFSSQLPR), and 'Member type' (RPGLE). Below these is a list of 'Types to choose from' including ICFF, LF, MENU, MINU, MINUCMD, MINUDDS, PF, PNLGRP, PRTF, REXX, RPG, RPGLR (highlighted), and RPT. The 'Text' field contains 'I/O Procedure prototypes'. At the bottom, a summary line reads: 'ADDPFM FILE(FLGHT400M2/QRPGLESRC) MBR(NFSSQLPR) SRCTYPE(RPGLR) TEXT('I/O Procedure prototypes')'. 'Finish' and 'Cancel' buttons are at the bottom right.

Figure 14: Create member for I/O procedure prototypes

- Click **Finish**.
- From the **Toolbar** menu, click **File > New > iSeries Source Member**.
- Specify **FLGHT400M2** for the **Library**, **QRPGLESRC** for the **File**, **NFSSQL** for the member, **SQLRPGLR** for the member type, and **I/O Procedures** for the text.
- Click **Finish**.
- From the **Toolbar** menu, click **File > New > iSeries Source Member**.
- Specify **FLGHT400M2** for the library, **QSRVSR** for the file, **NFSSQL** for the member, **BND** for the member type, and **NFSSQL Procedures to export** for the text.
- Click **Finish**.

For each identified RLA operation or group of operations, an equivalent I/O procedure will be created that uses embedded SQL to access the data. This involves the following steps:

- Add externally described data structure definition based on the SQL views to member **NFSSQLPR** (if one does not already exist for the view). These will be used as parameters for the data access subprocedures.

2. Add procedure prototype to source file member **NFSSQLPR** in file **QRPGLESRC**.
3. Add new procedure to source file member **NFSSQL** in file **QRPGLESRC**.
4. Add the procedure name to the list of procedures to export in source file member **NFSSQL** in file **QSRVSRRC**. (**Note:** For FLIGHT400, a naming convention was implemented for the I/O procedures. To clearly distinguish them from other procedures, all database access procedures begin with the prefix "dbx-".)

Table 7 summarizes the I/O procedure methodology behind each type of table access:

Type of access	I/O procedure methodology
READ	Pass in key as input parameter. This key is used to find desired row(s). It returns the row(s), SQL state return code (indicating success or failure), and the SQL error message.
UPDATE	Pass in key and updated row as input parameters. This key is used to find the row and row is updated. It returns the SQL state return code and SQL error message.
INSERT	Pass in the new row as input parameter. The row is inserted. It returns the SQL state return code and SQL error message.
DELETE	Pass in the key as input parameter. This key is used to find and delete the row. It returns the SQL state return code and SQL error message.

Table 7: Table access I/O procedure methodology

To continue examining the example procedure GetCustNumber, the list in Table 8 will be updated to include the new I/O procedure implemented to access the data:

RLA operation	Table	Business rule	view	New procedure
chain	CUSTOMERS	Search CUSTOMERS table for a record that matches the specified customer name.	ALLCUSTS	dbxGetCusByNam
setll read	CUSTOMERS	Find the record/row in CUSTOMERS table with the highest value for the customer number field.	None; replaced by sequence object	dbxInsCus
Write	CUSTOMERS	Insert a new record/row into the CUSTOMERS table	ALLCUSTS	dbxInsCus

Table 8: New RLA operations business rule procedure

Next, the two new I/O modules, dbxGetCusByNam and dbxInsCus are added:

7. To incorporate the externally-described data structure definition based on the SQL view ALLCUSTS, add the following line to the member NFSSQLPR:

```
d customerRow    e ds                extname(allCusts)
```

8. To incorporate the prototypes for the two procedures, add the following lines to the to the source file member NFSSQLPR:

```
*-----*
d dbxGetCusByNam  pr                likeds(customerRow)
*-----*
d  inCusName      64a  const
d  outSqlState    5a
d  outSqlMsg      256a

*-----*
d dbxInsCus       pr
*-----*
d  inCusRow       likeds(customerRow)
d  outSqlState    5a
d  outSqlMsg      256a
```

9. To add the two procedures, insert the code on the next page into the source member NFSSQL:


```

*****
p dbxGetCusByNam b export
*****
* Returns one row from CUSTOMERS table that matches the
* specified customer name

d dbxGetCusByNam pi likeds(CustomerRow)
d inCusName 64a const
d outSqlState 5a
d outSqlMsg 256a

d outCustRow ds likeds(CustomerRow)

C/EXEC SQL
c+ select * into :outCustRow :customerNIArr
c+ from allCusts
c+ where customer_Name = :inCusName
C/END-EXEC

/free
outSqlState = sqlState;
outSqlMsg = *blanks;
if %subst(sqlState:1:2) <> '00';
getSQLDiagMsg(outSqlMsg);
clear outCustRow;
endif;
return outCustRow;
/end-free

*****
p dbxInsCus b export
*****
* Inserts a new row into CUSTOMERS table

d dbxInsCus pi likeds(CustomerRow)
d inCusRow 5a
d outSqlState 256a
d outSqlMsg 9b 0

d NextCustNum s

* Use SQL sequence to generate next customer number
C/EXEC SQL
c+ VALUES NEXT VALUE FOR customer_number
c+ INTO :nextCustNum
C/END-EXEC

/free
outSqlState = sqlState;
outSqlMsg = *blanks;
if %subst(sqlState:1:2) <> '00';
getSQLDiagMsg(outSqlMsg);
inCusRow.cust_no = -1;
return;
else;
inCusRow.cust_no = nextCustNum;
endif;
/end-free

C/EXEC SQL
c+ insert into allCusts
c+ values (:inCusRow)
C/END-EXEC

/free
outSqlState = sqlState;
outSqlMsg = *blanks;
if %subst(sqlState:1:2) <> '00';
getSQLDiagMsg(outSqlMsg);
endif;
/end-free

p dbxInsCus e

```

10. Add the new procedure names to the binder source member NFSSQL in physical file member FLGHT400M2/QSRVSRC. This contains all EXPORT statements identifying the procedures to export from NFSSQL. Once added, this service program will look as shown below. (**Note:** Appendix B lists the complete version of the binder source member.)

```
STRPGMEXP PGMLVL(*CURRENT)
EXPORT    SYMBOL(DBXGETCUSBYNAM)
EXPORT    SYMBOL(DBXINSCUS)
ENDPGMEXP
```

Two I/O modules have been created to replace the RLA operations in procedure GetCustNumber.

Step 6: Replacing each RLA instance or group with call to new I/O procedure

Once the I/O procedures have been written, the RLA operations can be replaced. For each RLA operation or group of operations identified in step 2, replace it with a call to the new I/O procedures.

Below are the activities involved with this step (and illustrated in Table 9):

1. Copy the source file member to the new schema.
2. Edit the new source file member.
3. Remove/comment out the file declarations (F specs) for each physical and logical files.
4. Remove/comment out all explicit file OPEN and CLOSE operations and all RLA operations.
5. Insert calls to I/O procedures where RLA operations previously existed.
6. Save the member.

FLGHT400 (Original) version	FLGHT400M2 (Modernized) version
<pre> //***** p GetCustNumber b export //***** d GetCustNumber pi d Name 64 const d Number 9B 0 d Generate 1 const options(*nopass) // d NameV s 64 varying /free NameV = %trim(Name); chain(e) NameV CUSTR; if not %found; if %parms > 2 and Generate = 'Y'; dou not %error; setll *hival CUSTOMERR; read(e) CUSTOMERR; CNUMBR = CNUMBR + 1; CUSTNM = NameV; write(e) CUSTOMERR; enddo; CUSTNO = CNUMBR; else; CUSTNO = -1; endif; endif; Number = CUSTNO; /end-free p GetCustNumber e </pre>	<pre> //***** p GetCustNumber b export //***** d GetCustNumber pi d Name 64 const d Number 9B 0 d Generate 1 const options(*nopass) // d NameV s 64 varying d outSqlState s 5a d outSqlMsg s 256a d newCustRow ds likes(CustomerRow) d custNo s like(customerRow.cust_no) /free NameV = %trim(Name); customerRow = dbxGetCusByNam (Name:outSqlState:outSqlMsg); [1] if %subst(outSqlState:1:2) <> '00'; [2] if %parms > 2 and Generate = 'Y'; [3] newCustRow.cust_name = Name; [3] dbxInsCus(newCustRow : outSqlState : outSqlMsg); [4] custNo = newCustRow.cust no; else; custNo = -1; endif; else; custNo = customerRow.cust_no; [5] endif; Number = CUSTNO; /end-free p GetCustNumber e </pre>

Table 9: Replacing RLA operations

The following changes are made to the new version (refer to the numbers in red):

1. A function call to `dbxGetCusByNam` replaced the RPG CHAIN operation. This function returns a data structure containing the row that matches the specified customer name.
2. Instead of checking the value returned by the built-in function `%FOUND`, the value of the SQL State return code (that was returned from `dbxGetCusByName`) is checked.
3. The data structure `newCustRow` is primed with the passed in Customer Name.
4. The RPG SETLL and READ operations, as well as the WRITE operations are replaced by a function call to `dbxInsCus`. This generates the new order number, inserts the new row into the table, and returns the new row in the `newCustRow` data structure.
5. The new customer number is extracted from the `newCustRow` data structure.

Everything else remains relatively unchanged.

Step 7: Creating the I/O module

After all of the I/O procedures have been added to NFSSQL and the prototypes added to NFSSQLPR, it is time to create the module object. Issue the following command to build the module:

```

CRTSQLRPGI OBJ(QTEMP/NFSSQL) SRCFILE(FLGHT400M2/QRPGLESRC) +
  COMMIT(*NONE) OBJTYPE(*MODULE) DLYPRP(*YES) DBGVIEW(*SOURCE)

```

Step 8: Creating the I/O service program

After the prototypes have been defined and the module created, you can create the service program.

1. Issue the following command to build the service program:

```
CRTSRVPGM SRVPGM(FLGHT400M2/NFSSQL) MODULE(QTEMP/NFSSQL) +
TEXT('Flight 400 SQL I/O Procedures')
```

Step 9: Recompiling existing modules, service programs, and programs

You now have a service program with the I/O procedures, and the RLA operations with calls to the new I/O procedure were replaced. The next step is the recompile all existing objects.

1. Recreate the module and service program NFS001:

```
CRTRPGMOD MODULE(QTEMP/NFS001) +
SRCFILE(FLGHT400M2/QRPGLESRC) DBGVIEW(*ALL)
CRTSRVPGM SRVPGM(FLGHT400M2/NFS001) MODULE(QTEMP/NFS001) +
TEXT('Flight 400 Information Procedures')
```

2. Recreate the modules and service program NFS400:

```
CRTRPGMOD MODULE(QTEMP/NFS402) +
SRCFILE(FLGHT400M2/QRPGLESRC) DBGVIEW(*ALL)
CRTRPGMOD MODULE(QTEMP/NFS404) +
SRCFILE(FLGHT400M2/QRPGLESRC) DBGVIEW(*ALL)
CRTRPGMOD MODULE(QTEMP/NFS405) +
SRCFILE(FLGHT400M2/QRPGLESRC) DBGVIEW(*ALL)
CRTSRVPGM SRVPGM(FLGHT400M2/NFS400) MODULE(QTEMP/NFS402 +
QTEMP/NFS404 QTEMP/NFS405) TEXT('Flight +
400 Information Procedures')
```

3. Create a new binding directory: **CRTBNDDIR BNDDIR(FLGHT400M2/FLGHT400M)**

4. Add the following binding directory entries to the new binding directory:

NFS001	*SRVPGM	*LIBL
NFSUTIL	*SRVPGM	*LIBL
NFS400	*SRVPGM	*LIBL
NFSSQL	*SRVPGM	*LIBL

5. The NFS001 and NFS400 service programs were updated; therefore, the existing programs that bind them must be recompiled. Here is a list of those programs:

- FRS000
- FRS000X
- FRS000Y
- FRS001
- FRS002
- FRS003
- FRS004
- FRS009
- FRS402
- FRS403
- FRS404
- FRS405
- FRS407
- FRS408
- FRS410

For each program listed, follow these steps:

6. Copy source file member from FLGHT400M to FLGHT400M2
7. Issue command to create the bound RPG program:

```
CRTBNDRPG PGM(FLGHT400M2/FRSxxx) SRCFILE(FLGHT400M2/QRPGLESRC)
```

Other considerations

Null values

A NULL is an attribute that a column can have to indicate a missing or unknown value. All data types can be assigned a NULL value, but they can be problematic in embedded SQL programs when selecting or fetching into host variables. The error SQL0305 occurs when attempting to fetch columns with null values into program variables.

In the original version of FLGHT400, the keyword ALWNULL was specified for several of the physical file fields. We need to be able to handle null values because of the use of embedded SQL and fetching into host variables.

When using embedded SQL, you must handle values by choosing indicator variables.

Indicator variables detect NULL values in host variables.

If you choose a host structure for the retrieval values (as is the case with all of the FLGHT400 examples), define a 2-byte binary (integer) array the same number of elements as the number of data structure subfields. Specify this indicator immediately after the host structure.

The following example shows the use of a host structure with an indicator array.

```
* Define the host structure
d customerRow   e ds                extname(xcustomers) prefix(c_)

* Null indicator arrays
D customerNIArr  s                5i 0 dim(12)

C/EXEC SQL
C+   FETCH cursorCusLstCN into :customerRow :customerNIArr
C/END-EXEC
```

SQL error handling

When you process an SQL statement in your program, SQL places a return code in both the SQLCODE and SQLSTATE fields. Originally, iSeries developers used the SQLCODE field to detect errors and warning conditions. However, SQLCODE was never a standard, and problems arose when different database managers developed their own error code structures. This made it difficult to build portable code to manage error conditions.

The addition of a new field, SQLSTATE, complies with SQL-92 standards. This field contains a standardized error code consistent across other IBM database products and other SQL-92 conformant database managers.

Both return codes indicate the success or failure of the running of your statement. If SQL encounters an error while processing the statement, the SQLCODE is a negative number, and the first two digits of the SQLSTATE field are not **00**, **01**, or **02**. When processing the SQL statement, if you encounter a warning (which is a valid condition), the SQLCODE is a positive number, and the first two digits of SQLSTATE are **01** or **02**. If you process an SQL statement without encountering an error or warning condition, the SQLCODE is zero and the SQLSTATE is **00000**.

As a rule, the return code field to use in your applications is SQLSTATE when there is any concern about portability. This is because SQLSTATE provides a platform-independent error code structure

and is common across many database managers. For FLGHT400, the SQLSTATE field was chosen to detect and process error conditions.

In i5/OS V5R3, additional support is available for the GET DIAGNOSTICS statement in HLL programs using embedded SQL. You can use the GET DIAGNOSTICS statement to return diagnostic information about the last executed SQL statement. This simplifies error handling and is consistent with the errors that are typically handled in the SQL procedural language.

Below is an example of how FLGHT400 uses these SQL error-handling features. A procedure getSQLDiagMsg was created to capture an SQL error message, extract the message text and constraint name (if the error was a constraint violation), and return the message text to the caller.

Below is the code for this procedure:

```

* _____
*
*   Get SQL Diagnostic message
* _____
p getSQLDiagMsg   b

d getSQLDiagMsg   pi
d   SQLmsg                256A
*
d   cst_name           s           128A
*
/free
   cst_name = *blanks;
   sqlMsg   = *blanks;
/end-free
*
C/EXEC SQL
C+   GET DIAGNOSTICS CONDITION 1
C+       :cst_name = CONSTRAINT_NAME,
C+       :sqlMsg = MESSAGE_TEXT
C/END-EXEC
*
p getSQLDiagMsg   e

```

You might have noticed that this procedure only processes a single error condition. The following modified example demonstrates how you can process and display multiple errors that will possibly result from one SQL statement:

```

C/EXEC SQL
C+   GET DIAGNOSTICS
C+       :cond_count = NUMBER
C/END-EXEC

/free
   for i = 1 to cond_Count;
/end-free

C/EXEC SQL
C+   GET DIAGNOSTICS CONDITION :i
C+       :cst_name = CONSTRAINT_NAME,
C+       :sqlMsg = MESSAGE_TEXT
C/END-EXEC

/free
   dsply sqlMsg; //display the error message to the screen
   endfor;
/end-free

```

Below is an example of how this procedure is called:

```
*****
p dbxGetFrCty      b                                export
*****
* Returns one row from FRCITY table that matches the
* specified departure city initials

d dbxGetFrCty      pi                                likeds(frCityRow)
d inCityInt        3a                                const
d outSqlState      5a
d outSqlMsg        256a

d outFrCityRow    ds                                likeds(frCityRow)

C/EXEC SQL
c+  select * into :outFrCityRow [1]
c+  from fromCities
c+  where frcint = :inCityInt
C/END-EXEC

/free
outSqlState = sqlState; [2]
outSqlMsg = *blanks;
if %subst(sqlState:1:2) <> '00'; [3]
  getSQLDiagMsg(outSqlMsg);
  clear outFrCityRow;
endif;
return outFrCityRow;
/end-free

p dbxGetFrCty      e
```

Code sample notes (refer to the numbers in red):

1. Execute the SQL statement.
2. Assign the SQLState field value to the output parameter, return code field.
3. Check the value of SQLSTATE. If an error occurred, call the getSQLDiagMsg procedure to extract and return the error messages.

The SQLState field and error message fields can be bubbled up to calling programs so that the appropriate feedback can be provided to the user or requester (display the error message to the screen, send the error message back to the client, and so forth).

Stage 3: Moving business rules to the database

One of the problems with coding business rules at the application program level is that if another interface is used to access the data, the business rules can be circumvented. For example, if the application has an edit in the RPG program to reject the creation of a new order for a customer with a bad credit rating, the business rule is only enforced if the order is placed through the application. A user with authority to the database can still add a row to the order table and violate this rule by using an SQL interface or even the Data File Utility (DFU) on a 5250 emulation session. When you enforce business rules this way, you expose your database to major data integrity issues. By moving business rules to the database level, you can enforce the defined rules, regardless of the interface. In stage 3, there is an examination of this process and examples are provided regarding how this was carried out for the FLGHT400 database.

Implementing referential integrity constraints

Referential integrity is a set of database rules that enforce the following: The value of each foreign key in a table must match the value of a primary key in another table. The table with the foreign key is referred to as the dependent table. The table with the primary key is called the parent table.

In this section, referential integrity (RI) constraints are added to the tables to enforce business rules at the database level. This will reduce application-level changes and eliminate the possibility of an introduction of RI violations by other interfaces to the tables (Data File Utility, SQL statements, and so forth).

Neither the original version of FLGHT400 nor the modernized version contained any referential integrity. Thus, an order can be created with an agent number that did not exist in the agent table, or with a customer number that did not exist in the Customer table. To prevent such violations of data integrity, either you need to add code to the application to enforce data integrity, or you can let the database do it for you. If you add code to your application, entered or updated data through other interfaces (for example, STRDFU or SQL statements) is a possibility, and the exposure will still exist. However, if RI constraints are defined, integrity is enforced at the database level with no possibility of circumventing it.

In addition, if the constraints are coded in SQL, they are more portable, both from a platform perspective and a skills perspective. Consider the following example of adding RI constraint. First, a primary key constraint must exist for the parent table. In this example, a primary key constraint is added to table FLIGHTS:

```
ALTER TABLE flights
    ADD CONSTRAINT flights_pk_flight_number
    PRIMARY KEY( flight_number ;
```

Note: You can also define a primary key constraint when creating the table with the CREATE TABLE statement. Below, you can see how the same constraint is defined for the FLIGHTS table using the CREATE TABLE statement:

```
CREATE TABLE flights (
    flight_number FOR COLUMN flight_no INTEGER DEFAULT NULL ,
    departure_initials FOR COLUMN depar_int VARCHAR(16) CCSID 37 DEFAULT NULL,
    departure varchar(16) CCSID 37 DEFAULT NULL ,
    day_of_week FOR COLUMN day_week VARCHAR(16) CCSID 37 DEFAULT NULL ,
    arrival_initials FOR COLUMN arriv_int VARCHAR(16) CCSID 37 DEFAULT NULL ,
    arrival varchar(16) CCSID 37 DEFAULT NULL ,
    departure_time FOR COLUMN depar_time VARCHAR(32) CCSID 37 DEFAULT NULL ,
    arrival_time FOR COLUMN arriv_time VARCHAR(32) CCSID 37 DEFAULT NULL ,
    airlines VARCHAR(32) CCSID 37 DEFAULT NULL ,
    seats_available FOR COLUMN seats_avl INTEGER DEFAULT NULL ,
    ticket_price FOR COLUMN ticket_prc VARCHAR(22) CCSID 37 DEFAULT NULL ,
    mileage INTEGER DEFAULT NULL,
    CONSTRAINT flights_pk_flightnumber PRIMARY KEY( flight_number ) );
```

Now, a foreign key constraint can be added to the dependant table. Below, you can see that the constraint has been added to the ORDERS table:

```
ALTER TABLE orders
    ADD CONSTRAINT orders_fk_flight_number
    ADD FOREIGN KEY (flight_number)
    REFERENCES flights (flight_number)
    ON DELETE NO ACTION ON UPDATE NO ACTION;
```


No matter what interface is used to update the table, a row cannot be added to ORDERS unless the specified value of the flight number column has a row with that flight number in table FLIGHTS.

```
INSERT INTO FLGHT400M2/ORDERS VALUES(999555599, 1, 5555, 5555, NULL, 1, '3', '3333')
```

If such an operation as the one above is attempted (and there is no row in table FLIGHTS with flight number 999555599), the error SQL0530 will be returned.

```
Message ID . . . . . : SQL0530      Severity . . . . . : 30
Message type . . . . . : Diagnostic

Message . . . . . : Operation not allowed by referential constraint
                   FLIGHTS_PK_FLIGH_TNUMBER in FLGHT400M2.
Cause . . . . . : If this is an INSERT or UPDATE statement, the value is not
                  valid for the foreign key because it does not have a matching value in the
                  parent key. If this is a DELETE statement affected by a SET DEFAULT delete
                  rule, the default value is not valid for the same reason. If this is an
                  ALTER TABLE statement, the result of the operation would violate the
                  constraint FLIGHTS_PK_FLIGHT_NUMBER. Constraint FLIGHTS_PK_FLIGHT_NUMBER in
                  FLGHT400M2 for table ORDERS in FLGHT400M2 requires that any non-null value
                  of the foreign key have a matching value in the parent key.
Recovery . . . . . : To conform to the constraint rule, you must either:
                   -- change the INSERT or UPDATE value to match a value in the parent key,
                   -- insert a row in the parent file that matches the foreign key values
                   being inserted or updated.
                   -- insert a row in the parent file that matches the foreign key default
                   values of the dependent rows. Otherwise, you must drop the referential constraint.
```

Note: The SQL DDL script to create all of the RI constraints can be found in Appendix A, “SQL DDL script to create the constraints.” For more information on how your application can detect and handle RI constraint violations, refer to the “SQL error handling” section (in Stage 2 above).

Implementing check constraints

Check constraints are used to ensure that column data does not violate rules defined for the column or only a certain set of values is allowed into a column.

As mentioned, you can code edits into the application, but at the risk of circumventions through other interfaces. A check constraint defined at the database level ensures the constraint is always enforced.

In the following example, a check constraint is created to ensure that only the values of **1**, **2**, or **3** can be specified for the column CLASS in the table ORDERS:

```
ALTER TABLE orders ADD CONSTRAINT flght400m2/orders_ck_class
CHECK (class='1' OR class='2' OR class='3')
```

In the following example, there is an attempt to violate this rule by adding a row with a value other than 1, 2, or 3. The result is the insert fails with error message SQL0545.

```

INSERT INTO FLGHT400M2/ORDERS VALUES(999555599, 1, 5555, 1100171, NULL, 1, 'Z', '3333')

Message ID . . . . . :   SQL0545           Severity . . . . . :   30
Message type . . . . . :   Diagnostic

Message . . . . . :   INSERT or UPDATE not allowed by CHECK constraint.
Cause . . . . . :   The value being inserted or updated does not meet the
                    criteria of CHECK constraint ORDERS_CK_CLASS. The operation is not allowed.
Recovery . . . . . :   Change the values being inserted or updated so that the
                    CHECK constraint is met. Otherwise, drop the CHECK constraint
                    ORDERS_CK_CLASS.

```

For more information on how your application can detect and handle check constraint violations, refer to the “SQL error handling” section (in Stage 2 above).

Implementing automatic key generation and unique identifiers

Two of the tables (CUSTOMERS AND ORDERS) contain primary, application generated key fields. Both tables generate the next key by finding the current maximum key value in the table and incrementing it by one. This is not an optimal solution when many concurrent users are trying to generate the next key value; it can result in locking and serialization issues. In these cases, a much better solution is to utilize database features such as identity columns or sequence objects to automatically generate unique values. Let the database manager handle the key generation as well as locking and serialization of that value, so the programmer can concentrate on real business logic.

FLGHT400 involved the implementation of automatically generated keys using Sequence objects rather than identity columns, because the tables already contained many rows and existing key values. With a Sequence object implementation, keep existing keys as is. You merely specify the starting value of the next key value for the Sequence object. This is not possible with identity columns. Removing and regenerating key values for the tables when implementing identity columns requires more sophisticated analysis. Tool development will also be required to prevent compromise to database integrity. To implement automatic key generation using a sequence object, take the following steps:

1. Identify the tables and keys to implement automatic key generation:

Table	Key
CUSTOMERS	CUSTOMER_NO
ORDERS	ORDER_NUMBER

2. Determine the current maximum value for the identified tables and keys. From Run SQL script window, execute the following statements:

```

SELECT MAX(CUSTOMER_NO) FROM FLGHT400M2.CUSTOMERS;
SELECT MAX(ORDER_NUMBER) FROM FLGHT400M2.ORDERS;

```

3. Increment the result of each query by 1 and document that value as the next key value.

Table	Key	Next key value
CUSTOMERS	CUSTOMER_NO	10023
ORDERS	ORDER_NUMBER	5671308

4. Using the calculated next key values, execute the following SQL statements to create the Sequence objects:

```

CREATE SEQUENCE flght400m2.customer_number
AS INTEGER
START WITH 10023
INCREMENT BY 1
MINVALUE 10023
MAXVALUE 2147483647
NO CYCLE CACHE 20 NO ORDER ;

CREATE SEQUENCE flght400m2.order_number
AS INTEGER
START WITH 5671308
INCREMENT BY 1
MINVALUE 5671308
MAXVALUE 2147483647
NO CYCLE CACHE 20 NO ORDER ;

```

- For the tables and keys identified, locate the procedures that are responsible for generating the next key value and inserting the new row. Modify these procedures to use the Sequence object instead (as shown in Table 9).

FLGHT400 version (original)	FLGHT400M2 version (modernized)
<pre> chain(e) NameV CUSTR; if not %found; if %parms > 2 and Generate = 'Y'; dou not %error; setll *hival CUSTOMERR; read(e) CUSTOMERR; CNUMBR = CNUMBR + 1; CUSTNM = NameV; write(e) CUSTOMERR; enddo; CUSTNO = CNUMBR; else; CUSTNO = -1; return; endif; endif; </pre>	<pre> * Get next customer number from sequence object c/EXEC SQL c+ VALUES NEXT VALUE FOR customer_number c+ INTO :nextCustNum c/END-EXEC /free if sqlCode = 0; inCusRow.cust_no = nextCustNum; else; inCusRow.cust_no = -1; return; endif; /end-free C/EXEC SQL c+ insert into customers c+ values (:inCusRow) C/END-EXEC </pre>

Table 9: Generating next key value in FLGHT400 (original) and FLGHT400M2 (enhanced)

In the original version of FLGHT400, the CUSTOMERS file is searched for a record that matches the specified customer name. If that name is not found, the program (using the SETGT and READ operations) interrogates the same file for the current maximum customer number value. This value is incremented by one to yield the new customer number. A new record, with that customer number, is added to the CUSTOMERS file. This method is prone to error as multiple versions of the program running simultaneously in different jobs can retrieve the same high value for customer number. These simultaneous runs generate duplicate keys.

In the enhanced version, the program retrieves the next customer number value from the SQL Sequence object (a data area). Using that customer number, a new row is then inserted in the CUSTOMERS table. The issue of duplicate key generation is avoided as each job caches 20 unique sequence values (based on the **CACHE 20** clause in the **CREATE SEQUENCE** statement). When those 20 sequence values are consumed, the database manager will access and lock the Sequence object for the next 20 values. **Note:** You can specify **NO CACHE** on the **CREATE SEQUENCE** statement, but performance is affected because the database manager must access and lock the Sequence object for each value generated.

Implementing trigger programs

Defining constraints is a sound way of enforcing simple table and column-level business rules; however, DB2 UDB triggers can carry out more complex rules. A trigger is a user-written program or SQL routine that is associated with a database table. A trigger is automatically activated (triggered) by the database manager when a change occurs in the table, regardless of the interface that initiated the change.

For FLGHT400, a trigger tracks changes made to the **Credit Card Number** field (CC_NUMBER) in the table CUSTOMERS. To create this trigger and supporting database object, do the following:

1. Create a table to track the credit card number changes. From a new Run SQL Scripts window, execute the following statement:

```
CREATE TABLE flght400m2.cc_number_track (  
customer_no FOR COLUMN cust_no INTEGER NOT NULL DEFAULT 0,  
old_cc_number FOR COLUMN old_cc_no CHAR(20) CCSID 37 NOT NULL DEFAULT '' ,  
new_cc_number FOR COLUMN new_cc_no CHAR(20) CCSID 37 NOT NULL DEFAULT '' ,  
update_user FOR COLUMN upd_user CHAR(20) CCSID 37 NOT NULL DEFAULT '' ,  
update_timestamp FOR COLUMN upd_ts TIMESTAMP DEFAULT NULL ) ;
```

2. Create a trigger on the CUSTOMERS table that will insert a new row into the table CC_NUMBER_TRACK when column CC_NUMBER is changed. Execute the following statement:

```
CREATE TRIGGER flght400m2.cc_number_track  
AFTER UPDATE OF cc_number ON flght400m2.customers  
REFERENCING OLD AS orow  
NEW AS nrow  
FOR EACH ROW  
MODE DB2SQL  
WHEN ( nrow.cc_number <> orow.cc_number )  
BEGIN INSERT INTO flght400m2.cc_number_track  
    (customer_no ,  
     old_cc_number ,  
     new_cc_number ,  
     update_user ,  
     update_timestamp )  
VALUES  
    (nrow.customer_no ,  
     orow.cc_number ,  
     nrow.cc_number ,  
     USER ,  
     CURRENT_TIMESTAMP ) ;  
END;
```

Stage 4: Externalizing data access

So far, in the database enhancement process we have performed the following processes:

- Reverse engineered the database so that SQL defines the database objects
- Created I/O modules to access the data using SQL access methods
- Moved some of the business rules to the database

One final implementation worth mentioning is that stored procedures can be used to enhance the FLGHT400 application. Stored procedures provide a standard way to call an external procedure from within an application by using an SQL statement. Some advantages of stored procedures include the following:

- Improved modularity by allowing the same code to be used for both an existing 5250 application and new Web-based solutions
- Better partitioning of logic (for example, separation of presentation and database logic)
- Using an industry standard interface for remote invocations of host programs (including interfaces for JDBC and ODBC)
- Allowing you to take advantage of iSeries security features. The stored procedure's underlying program or service program can adopt authority, giving the stored procedure the ability to access data that is otherwise restricted.

The two types of stored procedures are:

- **SQL stored procedures:** These procedures are based on procedural extensions to SQL and enable better portability of logic (and programming skills) to and from other platforms.
- **External stored procedures:** These procedures are coded in one of the high-level languages available on the iSeries system. As a general rule, use external stored procedures when reusing code. External stored procedures can contain SQL statements, but they are not required.

For this example, an external stored procedure will be created that uses an I/O procedure in the service program created in the FLGHT400 database modernization process.

Note: The ability to create external stored procedures that invoke iSeries procedures in service programs was a feature added in i5/OS V5R3. Before that, external procedures only invoked program objects. The following list describes the process of creating an external stored:

1. **Create a user-written service program:** Use the I/O procedure dbxinscus2 in service program NFSSQL. This is one of the procedures that was created during stage 2 of the modernization process. The procedure accepts 11 input parameters (containing new customer information) and inserts a new row in the table CUSTOMERS. Here is the source code for that procedure:

```

*****
p dbxInsCus2      b                export
*****

d dbxInsCus2      pi
d customer_name   like(CustomerRow.cust_name)
d address         like(CustomerRow.address)
d city            like(CustomerRow.city)
d state           like(CustomerRow.state)
d zipCode         like(CustomerRow.zipcode)
d telephone      like(CustomerRow.telephone)
d creditCard      like(CustomerRow.cred_card)
d CC_Number       like(CustomerRow.cc_number)
d exp_Date        like(CustomerRow.exp_date)
d pref_Airline    like(CustomerRow.pref_airln)
d ff_Number       like(CustomerRow.ff_Number)
d outSqlState     5a
d outSqlMsg       256a

d Apostrophe      C                ''''

d SQLStatement    s                512
d nextCustNum2    s                like(CustomerRow.cust_no)
d customer_no     s                like(CustomerRow.cust_no)

* Useq SQL sequence to generate next customer number
C/EXEC SQL
c+  VALUES NEXT VALUE FOR customer_number
c+  INTO :nextCustNum2
C/END-EXEC

/free
outSqlState = sqlState;
outSqlMsg = *blanks;
if %subst(sqlState:1:2) <> '00';
  getSQLDiagMsg(outSqlMsg);
  customer_no = -1;
  return;
else;
  customer_no = nextCustNum2;
endif;
/end-free

* Insert the new row
C/EXEC SQL
c+  insert into allCusts
c+  values (:customer_no, :customer_name, :address, :city,
c+         :state, :zipcode, :telephone, :creditCard,
c+         :cc_number, :exp_date, :pref_airline, :ff_Number)
C/END-EXEC

/free
outSqlState = sqlState;
outSqlMsg = *blanks;
if %subst(sqlState:1:2) <> '00';
  getSQLDiagMsg(outSqlMsg);
endif;
/end-free

p dbxInsCus2      e

```

2. Register the program or procedure as a stored procedure using any SQL interface:

To do this, open a Run SQL Script window and execute the following statement:

```
CREATE PROCEDURE FLGHT400M2.ADDCUSTOMER ([a]
    IN customer_name VARCHAR(64) ,
    IN customer_address VARCHAR(150) ,
    IN customer_city CHAR(50) ,
    IN customer_state CHAR(2) ,
    IN customer_zip CHAR(9) ,
    IN customer_telephone CHAR(20) ,
    IN customer_credit_card CHAR(30) ,
    IN customer_cc_number CHAR(20) ,
    IN customer_cc_exp_date CHAR(20) ,
    IN customer_pref_airline CHAR(10) ,
    IN customer_ff_number CHAR(20) ,
    OUT sql_state char(5)
    OUT sql_msg char(256) )
LANGUAGE RPGLE
NOT DETERMINISTIC
MODIFIES SQL DATA
CALLED ON NULL INPUT
EXTERNAL NAME 'FLGHT400M2/NFSSQL(DBXINSCUS2 )' [b]
PARAMETER STYLE SQL ;
```

Code sample notes (see annotations in red):

- a. The SQL long name for the stored procedure
 - b. The name of the library (FLGHT400M2), service program (NFSSQL), and procedure (DBXINSCUS2) to invoke when calling the stored procedure.
3. With the stored procedure created and registered, you can call it from any SQL-based interface that supports the SQL CALL statement. For example, from a Run SQL Scripts window, execute the following statement to call the stored procedure and insert a new customer record:

```
Call FLGHT400M2.ADDCUSTOMER (
    'Steve Carfino' ,
    '123 Mockingbird Lane',
    'Rochester',
    'MN',
    '55901' ,
    '5074445555',
    'VISA',
    '1111111111111111',
    '10/2007',
    'NWA',
    '33333333333',
    ?,
    ?);
```

The following Java code snippet demonstrates how this stored procedure can be called from a Java client:

```
conn = ds.getConnection();
stmt = conn.prepareCall("CALL ADDCUSTOMER (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)");
stmt.setString(1, "Steve Carfino");
stmt.setString(2, "123 Mockingbird Lane");
stmt.setString(3, "Rochester");
stmt.setString(4, "MN");
stmt.setString(5, "55901");
stmt.setString(6, "50744455555");
stmt.setString(7, "VISA");
stmt.setString(8, "111111111111111");
stmt.setString(9, "10/2007");
stmt.setString(10, "NWA");
stmt.setString(11, "33333333333");

// call the stored procedure
stmt.execute();
```

Again, it is important to note that the same procedure (dbxInsCus2 in service program NFSSQL) can be called directly from an RPG program; or it can be called as a stored procedure from any SQL-based interface. This ability to reuse existing code is a compelling reason to consider implementing external stored procedures.

Summary

A vital, but often overlooked, component in the overall iSeries Developer Roadmap process is the modernization of the database. Most of the attention is focused on program modularization, separating business logic from screen logic, and reengineering or rewriting the user interface. However, the database is the foundation of most applications, and as such, needs to be given equal consideration in this enhancement process. Without database modernization, applications will continue to use native I/O access methods and will not be able to take advantage of many new database features and enhancements that are only available through SQL interfaces. While it is acknowledged that the database modernization effort is not a trivial one, the objective of this white paper is to provide enough information (through discussion and examples) to enhance your awareness on this subject. Hopefully, this white paper has convinced you that it is a vital step and that many advantages can be realized if it is included in the modernization process.

Appendix A: SQL scripts

Six SQL scripts are provided in this appendix:

- SQL procedure GenIndexList
- SQL procedure GenIndexList2
- SQL DDL script to create the tables
- SQL DDL script to create the indexes
- SQL DDL script to create the views
- SQL DDL script to create the constraints

SQL procedure GenIndexList

```
CREATE PROCEDURE QGPL.GenIndexList(SchemaName VARCHAR(10) )
LANGUAGE SQL
```

```
Level_1 :
BEGIN
```

```
    DECLARE ixTable VARCHAR(128);
    DECLARE ixTableSchema VARCHAR(128);
    DECLARE at_end INT DEFAULT 0;
```

```
    DECLARE c1 CURSOR FOR
        SELECT table_name, table_schema
        FROM qsys2.systables
        WHERE (table_schema=UPPER(SchemaName) OR
              system_table_schema=UPPER(SchemaName)) AND
              table_type = 'P' and
              file_type = 'D'
              order by table_name;
```

```
    DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET at_end = 1;
```

```
    DECLARE CONTINUE HANDLER FOR SQLSTATE '42704'
        SET at_end = 0;
```

```
    DROP TABLE QTEMP.IndexList;
    CREATE TABLE QTEMP.IndexList (
        TABLE_NAME VARCHAR(128) DEFAULT NULL,
        TABLE_SCHEMA VARCHAR(128) DEFAULT NULL,
        IDX_NAME VARCHAR(128) DEFAULT NULL,
        IDX_SCHEMA VARCHAR(128) DEFAULT NULL,
        IS_UNIQUE CHAR(1) DEFAULT NULL,
        KEY_COLUMN VARCHAR(128) DEFAULT NULL,
        ORDINAL_POS INTEGER DEFAULT NULL ,
        ORDERING CHAR(1) DEFAULT NULL,
        SYS_IDX_NAME VARCHAR(128) DEFAULT NULL,
        SYS_IDX_SCHEMA VARCHAR(128) DEFAULT NULL);
```

```
    OPEN c1;
    WHILE at_end = 0 DO
        FETCH FROM c1 INTO ixTable, ixTableSchema;
        call QGPL.GenIndexList2(ixTable, ixTableSchema);
    END WHILE;
    CLOSE c1;
End Level_1;
```

SQL procedure GenIndexList2

```
CREATE PROCEDURE QGPL.GenIndexList2(Table_name VARCHAR(128), SchemaName VARCHAR(128) )
LANGUAGE SQL

BEGIN

DECLARE ixTable VARCHAR(128);
DECLARE ixTableSchema VARCHAR(128);

SELECT table_name, table_schema INTO ixTable, ixTableSchema
FROM qsys2.systables
WHERE (table_name=UPPER(Table_name) and table_schema=UPPER(SchemaName)) OR
( system_table_name=UPPER(Table_name)
and system_table_schema=UPPER(SchemaName));

INSERT INTO QTEMP.IndexList
WITH SQL_Indices

(table_name, table_schema, index_name, index_schema, is_unique, system_index_name,
system_index_schema)
AS ( SELECT table_name, table_schema, index_name, index_schema, is_unique,
system_index_name,
system_index_schema
FROM qsys2.sysindexes
WHERE table_name = ixTable AND table_schema = ixTableSchema ),

Const_Indices (table_name, table_schema, index_name, index_schema, is_unique)
AS ( SELECT table_name, table_schema, constraint_name, constraint_schema,
CASE WHEN constraint_type = 'FOREIGN KEY' THEN 'D' ELSE 'U' END
FROM qsys2.syscst
WHERE table_name = ixTable AND table_schema = ixTableSchema
and constraint_type<>'CHECK'),

KeyedLF_Indices (table_name, table_schema, index_name, index_schema, is_unique)
AS ( SELECT DBFFIL, DBFLIB, dbxfile, dbxlib, dbxunq
FROM qsys.qadbxref, qsys.qadbfdep
WHERE dbxatr='LF' AND dbxnkf>0 AND dbffdp=dbxfile AND dbfldp=dbxlib
AND dbffil=ixTable
AND dbflib=ixTableSchema),

KeyedPF_Index (table_name, table_schema, index_name, index_schema, is_unique,dbxfile)
AS (SELECT dbxlib,dbxlib,dbxlib,dbxlib,dbxunq,dbxfile
FROM qsys.qadbxref
WHERE dbxatr='PF' AND dbxnkf>0 AND dbxlib=ixTableSchema AND dbxlib=ixTable
AND NOT EXISTS ( SELECT 1 FROM qsys.qadbfdep
WHERE dbxlib=dbccfl AND dbxfile=dbccff AND dbccty='PRIMARY KEY') )

/* -----*/
/* SQL Indices */
/* -----*/
SELECT
SUBSTRING(SQL_Indices.table_name, 1, 30),
SUBSTRING(SQL_Indices.table_schema, 1, 10),
CASE
WHEN ordinal_position = 1
THEN SUBSTRING(SQL_Indices.index_name, 1, 30)
ELSE ' '
END as INDEX_NAME,
CASE
WHEN ordinal_position = 1
THEN SUBSTRING(SQL_Indices.index_schema, 1, 10)
```

```

ELSE ' '
    END as INDEX_SCHEMA,
    is_unique,
    SUBSTRING(column_name, 1, 30) as KEY_COLUMN,
    ordinal_position,
    ordering ,
    SQL_Indices.system_index_name,
    SQL_Indices.system_index_schema
FROM SQL_Indices, qsys2.syskeys B
WHERE SQL_Indices.system_index_name = B.system_index_name
    AND SQL_Indices.system_index_schema = B.system_index_schema

UNION

/* -----*/
/* Constraint Indices */
/*-----*/
SELECT
SUBSTRING(Const_Indices.table_name, 1, 30),
SUBSTRING(Const_Indices.table_schema, 1, 10),
CASE
    WHEN ordinal_position = 1
        THEN SUBSTRING(Const_Indices.index_name, 1, 30)
    WHEN ordinal_position > 1 THEN ' '
    END,
CASE
    WHEN ordinal_position = 1
        THEN SUBSTRING(Const_Indices.index_schema, 1, 10)
    WHEN ordinal_position > 1 THEN ' '
    END,
is_unique,
SUBSTRING(column_name, 1, 30),
ordinal_position,
'A' ,
index_name,
index_schema
FROM Const_Indices, qsys2.syskeycst B
WHERE Const_Indices.index_name = B.constraint_name
    AND Const_Indices.index_schema = B.constraint_schema

UNION

/* -----*/
/* Keyed LF Indices */
/*-----*/
SELECT
SUBSTRING(KeyedLF_Indices.table_name, 1, 30),
SUBSTRING(KeyedLF_Indices.table_schema, 1, 10),
CASE
    WHEN dbkpos = 1
        THEN SUBSTRING(KeyedLF_Indices.index_name, 1, 10)
    ELSE ' '
    END,
CASE
    WHEN dbkpos = 1
        THEN SUBSTRING(KeyedLF_Indices.index_schema, 1, 10)
    ELSE ' '
    END ,
is_unique,
dbiint,
dbkpos,
dbkord,

```

```

        index_name,
        index_schema
FROM KeyedLF_Indices,   qsys.qadbkfld B       , qsys.qadbifld C
WHERE KeyedLF_Indices.index_name = dbkfil
      AND KeyedLF_Indices.index_schema = dbklib
      AND KeyedLF_Indices.index_name = dbifil
      AND KeyedLF_Indices.index_schema = dbilib
      AND dbkfld = dbifld

UNION

/* -----*/
/* Keyed PF Indices */
/*-----*/
SELECT
  SUBSTRING(KeyedPF_Index.table_name, 1, 30),
  SUBSTRING(KeyedPF_Index.table_schema, 1, 10),
  CASE
    WHEN dbkpos = 1
      THEN SUBSTRING(KeyedPF_Index.index_name, 1, 10)
    ELSE ' '
  END ,
  CASE
    WHEN dbkpos = 1
      THEN SUBSTRING(KeyedPF_Index.index_schema, 1, 10)
    ELSE ' '
  END ,
  is_unique,
  dbiint,
  dbkpos,
  dbkord,
  index_name,
  index_schema
FROM KeyedPF_Index,   qsys.qadbkfld B, qsys.qadbifld C
WHERE KeyedPF_Index.index_name = dbkfil
      AND KeyedPF_Index.index_schema = dbklib
      AND KeyedPF_Index.index_name = dbifil
      AND KeyedPF_Index.index_schema = dbilib
      AND dbkfld = dbifld
ORDER BY 1,2,9,10,7 ;

END;

```

SQL DDL script to create the tables

```
SET CURRENT SCHEMA FLGHT400M2;

CREATE TABLE agents (
    agent_no INTEGER DEFAULT NULL ,
    agent_name VARCHAR(64) CCSID 37 DEFAULT NULL ,
    agent_pwd VARCHAR(64) CCSID 37 DEFAULT NULL ) ;

LABEL ON TABLE agents
    IS 'Airline Agents' ;

LABEL ON COLUMN agents
    ( agent_pwd IS 'AGENT_PASSWORD' ) ;

LABEL ON COLUMN agents
    ( agent_pwd TEXT IS 'AGENT_PASSWORD' ) ;

CREATE TABLE AIRLINE (
    airlnm CHAR(16) CCSID 37 NOT NULL DEFAULT ' ' ,
    airlin CHAR(3) CCSID 37 NOT NULL DEFAULT ' ' ,
    PRIMARY KEY( airlnm ) ) ;

LABEL ON TABLE airline
    IS 'Airline Table for validation' ;

LABEL ON COLUMN airline
    ( airlnm IS 'AIRLINE NAME' ,
    airlin IS 'AIRLINE INITIALS' ) ;

LABEL ON COLUMN airline
    ( airlnm TEXT IS 'AIRLINE NAME' ,
    airlin TEXT IS 'AIRLINE INITIALS' ) ;

CREATE TABLE customers (
    customer_no FOR COLUMN cust_no INTEGER DEFAULT NULL ,
    customer_name FOR COLUMN cust_name VARCHAR(64) CCSID 37 DEFAULT NULL ,
    address VARCHAR(150) CCSID 37 NOT NULL DEFAULT ' ' ,
    city CHAR(50) CCSID 37 DEFAULT NULL ,
    state CHAR(2) CCSID 37 DEFAULT NULL ,
    zipcode CHAR(9) CCSID 37 DEFAULT NULL ,
    telephone CHAR(20) CCSID 37 DEFAULT NULL ,
    credit_card FOR COLUMN cred_card CHAR(30) CCSID 37 DEFAULT NULL ,
    cc_number CHAR(20) CCSID 37 DEFAULT NULL ,
    exp_date CHAR(20) CCSID 37 DEFAULT NULL ,
    pref_airline_id FOR COLUMN pref_airln CHAR(10) CCSID 37 DEFAULT NULL ,
    ff_number CHAR(20) CCSID 37 DEFAULT NULL ) ;

LABEL ON TABLE customers
    IS 'Airline Customers' ;

LABEL ON COLUMN customers
    ( customer_no TEXT IS 'CUSTOMER_NO' ,
    customer_name TEXT IS 'CUSTOMER_NAME' ,
    credit_card TEXT IS 'CREDIT_CARD' ,
    cc_number TEXT IS 'CREDIT_CARD' ,
    exp_date TEXT IS 'CREDIT_CARD' ,
    pref_airline_id TEXT IS 'CREDIT_CARD' ,
    ff_number TEXT IS 'CREDIT_CARD' ) ;

CREATE TABLE FLIGHTS (
```

```

flight_number FOR COLUMN flight_no  INTEGER DEFAULT NULL ,
departure_initials FOR COLUMN depar_int  CHAR(3) CCSID 37 DEFAULT NULL ,
departure VARCHAR(16) CCSID 37 DEFAULT NULL ,
day_of_week FOR COLUMN day_week  VARCHAR(16) CCSID 37 DEFAULT NULL ,
arrival_initials FOR COLUMN arriv_int  CHAR(3) CCSID 37 DEFAULT NULL ,
arrival VARCHAR(16) CCSID 37 DEFAULT NULL ,
departure_time FOR COLUMN depar_time VARCHAR(32) CCSID 37 DEFAULT NULL ,
arrival_time FOR COLUMN arriv_time VARCHAR(32) CCSID 37 DEFAULT NULL ,
airlines CHAR(16) CCSID 37 DEFAULT NULL ,
seats_available FOR COLUMN seats_avl  INTEGER DEFAULT NULL ,
ticket_price FOR COLUMN ticket_prc VARCHAR(22) CCSID 37 DEFAULT NULL ,
mileage INTEGER DEFAULT NULL ) ;

LABEL ON TABLE flights
  IS 'Flight schedule' ;

LABEL ON COLUMN flights
  (flight_number TEXT IS 'FLIGHT_NUMBER' ,
  departure_initials TEXT IS 'DEPARTURE_INITIALS' ,
  departure TEXT IS 'DEPARTURE' ,
  day_of_week TEXT IS 'DAY_OF_WEEK' ,
  arrival_initials TEXT IS 'ARRIVAL_INITIALS' ,
  arrival TEXT IS 'ARRIVAL' ,
  departure_time TEXT IS 'DEPARTURE_TIME' ,
  arrival_time TEXT IS 'ARRIVAL_TIME' ,
  airlines TEXT IS 'AIRLINES' ,
  seats_available TEXT IS 'SEATS_AVAILABLE' ,
  ticket_price TEXT IS 'TICKET_PRICE' ,
  mileage TEXT IS 'MILEAGE' ) ;

CREATE TABLE FRCITY (
  frcnam CHAR(16) CCSID 37 NOT NULL DEFAULT ' ' ,
  frcint CHAR(3) CCSID 37 NOT NULL DEFAULT ' ' ,
  frcaln CHAR(3) CCSID 37 NOT NULL DEFAULT ' ' ,
  frcnbr NUMERIC(3, 0) NOT NULL DEFAULT 0 ,
  PRIMARY KEY( frcnam ) ) ;

LABEL ON TABLE frcity
  IS 'City table for building Flights (From City)' ;

LABEL ON COLUMN frcity
  (frcnam IS 'CITY NAME' ,
  frcint IS 'CITY INITIALS' ,
  frcaln IS 'CITY AIRLINE' ,
  frcnbr IS 'FROM CITY NUMBER' ) ;

LABEL ON COLUMN frcity
  (frcnam TEXT IS 'FROM CITY NAME' ,
  frcint TEXT IS 'FROM CITY INITIALS' ,
  frcaln TEXT IS 'FROM CITY AIRLINE' ,
  frcnbr TEXT IS 'FROM CITY NUMBER' ) ;

CREATE TABLE ORDERS (
  order_number FOR COLUMN order_no  INTEGER DEFAULT NULL ,
  agent_no INTEGER DEFAULT NULL ,
  customer_no FOR COLUMN cust_no  INTEGER DEFAULT NULL ,
  flight_number FOR COLUMN flight_no  INTEGER DEFAULT NULL ,
  departure_date FOR COLUMN depar_date  TIMESTAMP DEFAULT NULL ,
  tickets_ordered FOR COLUMN ticks_ord  INTEGER DEFAULT NULL ,
  class VARCHAR(1) CCSID 37 DEFAULT NULL ,
  send_signature_with_order FOR COLUMN send_sig VARCHAR(16) CCSID 37 DEFAULT
NULL);

```

```

LABEL ON TABLE orders
  IS 'Airline Orders' ;

LABEL ON COLUMN orders
  (send_signature_with_order IS ' SEND_SIGNATURE WITH_ORDER' ) ;

LABEL ON COLUMN orders
  (order_number TEXT IS 'ORDER_NUMBER' ,
   agent_no TEXT IS 'AGENT_NO' ,
   customer_no TEXT IS 'CUSTOMER_NO' ,
   flight_number TEXT IS 'FLIGHT_NUMBER' ,
   departure_date TEXT IS 'DEPARTURE_DATE' ,
   tickets_ordered TEXT IS 'TICKETS_ORDERED' ,
   class TEXT IS 'CLASS' ,
   send_signature_with_order TEXT IS ' SEND_SIGNATURE WITH_ORDER' ) ;

CREATE TABLE TOCITY (
  tocnam CHAR(16) CCSID 37 NOT NULL DEFAULT '' ,
  tocint CHAR(3) CCSID 37 NOT NULL DEFAULT '' ,
  PRIMARY KEY( tocnam ) ) ;

LABEL ON TABLE tocity
  IS 'City table for building Flights (To City)' ;

LABEL ON COLUMN tocity
  ( tocnam IS 'TO CITY NAME' ,
   tocint IS 'TO CITY INITIALS' ) ;

LABEL ON COLUMN tocity
  (tocnam TEXT IS 'TO CITY NAME' ,
   tocint TEXT IS 'TO CITY INITIALS' );

```

SQL DDL script to create the indexes

```
SET CURRENT SCHEMA flght400m2;
CREATE INDEX agents_ix1
    ON agents ( agent_no ASC ) ;
CREATE INDEX agents_ix2
    ON agents ( agent_name ASC ) ;
CREATE INDEX airline_ix1
    ON airline ( airlin ASC ) ;
CREATE INDEX customers_ix1
    ON customers ( cust_name ASC ) ;
CREATE INDEX customers_ix2
    ON customers ( cust_no DESC ) ;
CREATE INDEX FLIGHTS_IX1
    ON flights ( departure ASC, arrival ASC, day_week ASC, flight_no ASC ) ;
CREATE INDEX flights_ix2
    ON flights ( flight_no ASC ) ;
CREATE INDEX frcity_ix1
    ON frcity ( frcint ASC ) ;
CREATE INDEX orders_ix1
    ON orders ( order_no ASC ) ;
CREATE INDEX tocity_ix1
    ON tocity ( tocint ASC ) ;
```


SQL DDL script to create the views

```
SET CURRENT SCHEMA flght400m2;

CREATE VIEW AllOrdCust
  (order_number FOR COLUMN order_no,
   customer_name FOR COLUMN cust_name,
   departure_date FOR COLUMN depar_date)
AS SELECT o.order_Number, c.customer_Name,
o.departure_date FROM customers c INNER JOIN orders o on
c.customer_no = o.customer_no;

CREATE VIEW AllOrders
  (order_number  FOR COLUMN order_no,
   agent_no,
   customer_no FOR COLUMN cust_no,
   flight_number FOR COLUMN flight_no,
   departure_date FOR COLUMN depar_date,
   tickets_ordered FOR COLUMN ticks_ord,
   class,
   send_signature_with_order FOR COLUMN send_sig)
AS SELECT * FROM orders;

CREATE VIEW allCusts
  (customer_no  FOR COLUMN cust_no,
   customer_name FOR COLUMN cust_name,
   address,
   city,
   state,
   zipcode,
   telephone,
   credit_card FOR COLUMN cred_card,
   cc_number,
   exp_date,
   pref_airline_id  FOR COLUMN pref_airln,
   ff_number)
AS SELECT * FROM customers;

CREATE VIEW allFlights
  (flight_number  FOR COLUMN flight_no,
   departure_initials FOR COLUMN depar_int,
   departure,
   day_of_week FOR COLUMN day_week,
   arrival_initial FOR COLUMN arriv_int,
   arrival,
   departure_time FOR COLUMN depar_time,
   arrival_time FOR COLUMN arriv_time,
   airlines,
   seats_available FOR COLUMN seats_avl,
   ticket_price FOR COLUMN ticket_prc,
   mileage)
AS SELECT * FROM flights;

CREATE VIEW fromCities
  (frcnam,
   frcint,
   frcaln,
   frcnbr )
AS SELECT * FROM frCity;

CREATE VIEW toCities
  (tocnam,
   tocint)
AS SELECT * FROM toCity;
```

SQL DDL script to create the constraints

```
SET CURRENT SCHEMA flght400m2;

ALTER TABLE customers
  ADD CONSTRAINT customers_pk_customer_number
  PRIMARY KEY( customer_no );

ALTER TABLE agents
  ADD CONSTRAINT agents_pk_agent_number
  PRIMARY KEY( agent_no );

ALTER TABLE airline
  ADD CONSTRAINT airline_pk_airline_name
  PRIMARY KEY( airlnm );

ALTER TABLE tocity
  ADD CONSTRAINT tocity_pk_to_city_initials
  PRIMARY KEY( tocint );

ALTER TABLE frcity
  ADD CONSTRAINT frcity_pk_from_city_initials
  PRIMARY KEY( frcint );

ALTER TABLE flights
  ADD CONSTRAINT flights_pk_flight_number
  PRIMARY KEY( flight_number) ;

ALTER TABLE flights
  ADD CONSTRAINT flights_fk_departure_city
  FOREIGN KEY (departure_initials)
  REFERENCES frcity (frcint)
  ON DELETE NO ACTION ON UPDATE NO ACTION;

ALTER TABLE flights
  ADD CONSTRAINT flights_fk_arrival_city
  FOREIGN KEY (arrival_initials)
  REFERENCES tocity (tocint)
  ON DELETE NO ACTION ON UPDATE NO ACTION;

ALTER TABLE flights
  ADD CONSTRAINT flights_fk_airlines
  FOREIGN KEY (airlines)
  REFERENCES airline (airlnm)
  ON DELETE NO ACTION ON UPDATE NO ACTION;

ALTER TABLE ORDERS
  ADD CONSTRAINT orders_fk_customer_number
  FOREIGN KEY (customer_no) REFERENCES customers (customer_no)
  ON DELETE NO ACTION ON UPDATE NO ACTION;

ALTER TABLE orders
  ADD CONSTRAINT orders_fk_flight_number
  FOREIGN KEY (flight_number)
  REFERENCES flights (flight_number)
  ON DELETE NO ACTION ON UPDATE NO ACTION;

ALTER TABLE orders
  ADD CONSTRAINT orders_fk_agent_number
  FOREIGN KEY (agent_no)
  REFERENCES agents (agent_no)
  ON DELETE NO ACTION ON UPDATE NO ACTION;
```

Appendix B: Source code after conversion

Here are the binder source, prototypes, and various I/O procedures for fLGHT400.

NFSSQL (binder source)

```
STRPGMEXP  PGMLVL (*CURRENT)
EXPORT     SYMBOL (DBXGETORD)
EXPORT     SYMBOL (DBXUPDORD)
EXPORT     SYMBOL (DBXDELORD)
EXPORT     SYMBOL (DBXNXTORDNUM)
EXPORT     SYMBOL (DBXINSORD)
EXPORT     SYMBOL (DBXGETCUSLST)
EXPORT     SYMBOL (DBXGETCUSBYNUM)
EXPORT     SYMBOL (DBXGETCUSBYNAM)
EXPORT     SYMBOL (DBXUPDCUS)
EXPORT     SYMBOL (DBXUPDCUS2)
EXPORT     SYMBOL (DBXINSCUS)
EXPORT     SYMBOL (DBXINSCUS2)
EXPORT     SYMBOL (DBXGETALLCUS)
EXPORT     SYMBOL (DBXGETCUSWCNAM)
EXPORT     SYMBOL (DBXGETORDCUSCN)
EXPORT     SYMBOL (DBXGETORDCUSTS)
EXPORT     SYMBOL (DBXGETFLTLST)
EXPORT     SYMBOL (DBXGETFLT)
EXPORT     SYMBOL (DBXGETFRCTY)
EXPORT     SYMBOL (DBXGETTOCTY)
EXPORT     SYMBOL (DBXGETTOCTYLST)
EXPORT     SYMBOL (DBXGETFRCTYLST)
ENDPGMEXP
```

NFSSQLPR (prototypes)

```

d orderRow      e ds      extname(allOrders) qualified
d customerRow   e ds      extname(allCusts)   qualified
d frCityRow     e ds      extname(FromCities) qualified
d toCityRow     e ds      extname(ToCities)  qualified
d flightsRow    e ds      extname(allFlights) qualified
d ordCustRow    e ds      extname(allOrdCust) qualified

* Null indicator arrays
D customerNIArr s          5i 0 dim(12)

*-----
d dbxGetOrd      pr          likeds(orderRow)
*-----
d inOrderNumber 9b 0 const
d outSqlState   5a
d outSqlMsg     256a

*-----
d dbxUpdOrd      pr          5a
*-----
d inOrderNumber 9b 0 const
d inOrderRow    likeds(orderRow)
d outSqlMsg     256a

*-----
d dbxDelOrd      pr          5a
*-----
```

```

d inOrderNumber          9b 0 const
d outSqlMsg              256a

*-----*
d dbxNxtOrdNum          pr              like(orderRow.order_no)
*-----*
d outSqlState           5a
d outSqlMsg              256a

*-----*
d dbxInsOrd             pr
*-----*
d inOrdRow              likeds(OrderRow)
d outSqlState           5a
d outSqlMsg              256a

*-----*
d dbxGetCusByNum        pr              likeds(customerRow)
*-----*
d inCusNumber           like(customerRow.cust_no) const
d outSqlState           5a
d outSqlMsg              256a

*-----*
d dbxGetCusByNam        pr              likeds(customerRow)
*-----*
d inCusName             64a const
d outSqlState           5a
d outSqlMsg              256a

*-----*
d dbxGetCusLst          pr
*-----*
d Position              64 const
d ListType              1 const
d CountReq              10i 0 const
d CountRet              10i 0
d OutputType            1 const
d outCustomers          likeds(CustomerRow) dim(100)
d                       options(*varsize)

*-----*
d dbxUpdCus             pr
*-----*
d inCusNumber           like(customerRow.cust_no) const
d inCusRow              likeds(customerRow)
d outSqlState           5a
d outSqlMsg              256a

*-----*
d dbxUpdCus2           pr
*-----*
d customer_no           like(customerRow.cust_no)
d customer_name         like(customerRow.cust_name)
d address               like(customerRow.address)
d city                 like(customerRow.city)
d state                like(customerRow.state)
d zipCode              like(customerRow.zipcode)
d telephone            like(customerRow.telephone)
d creditCard           like(customerRow.cred_card)
d CC_Number            like(customerRow.cc_number)
d exp_Date             like(customerRow.exp_date)

```

```

d  pref_Airline                like(customerRow.pref_airln)
d  ff_Number                   like(customerRow.ff_Number)
d  outSqlState                 5a
d  outSqlMsg                   256a

*-----*
d  dbxInsCus                    pr
*-----*
d  inCusRow                     likeds(customerRow)
d  outSqlState                 5a
d  outSqlMsg                   256a

*-----*
d  dbxInsCus2                   pr
*-----*
d  customer_name               like(customerRow.cust_name)
d  address                     like(customerRow.address)
d  city                        like(customerRow.city)
d  state                       like(customerRow.state)
d  zipCode                     like(customerRow.zipcode)
d  telephone                   like(customerRow.telephone)
d  creditCard                  like(customerRow.cred_card)
d  CC_Number                   like(customerRow.cc_number)
d  exp_Date                    like(customerRow.exp_date)
d  pref_Airline                like(customerRow.pref_airln)
d  ff_Number                   like(customerRow.ff_Number)
d  outSqlState                 5a
d  outSqlMsg                   256a

*-----*
d  dbxGetAllCus                 pr
*-----*

*-----*
d  dbxGetCusWCNam              pr
*-----*
d  inLastName                  64

*-----*
d  dbxGetOrdCus                pr
*-----*
d  custPosition                64    const
d  tsPosition                  26    const
d  listType                    1     const
d  countReq                    10i 0 const
d  countRet                    10i 0
d  outputType                  1     const
d  ordCustList                 likeds(ordCustRow) dim(100)

*-----*
d  dbxGetOrdCusCN              pr
*-----*
d  custPosition                64    const
d  listType                    1     const
d  countReq                    10i 0 const
d  countRet                    10i 0
d  outputType                  1     const
d  ordCustList                 likeds(ordCustRow) dim(100)

*-----*
d  dbxGetOrdCusTS              pr
*-----*

```

```

d  tsPosition          26    const
d  listType            1     const
d  countReq           10i 0  const
d  countRet           10i 0
d  outputType         1     const
d  ordCustList        likeds(ordCustRow) dim(100)

*-----*
d  dbxGetFrCty        pr                likeds(frCityRow)
*-----*
d  inCityInt          3a    const
d  outSqlState        5a
d  outSqlMsg          256a

*-----*
d  dbxGetToCty        pr                likeds(toCityRow)
*-----*
d  inCityInt          3a    const
d  outSqlState        5a
d  outSqlMsg          256a

*-----*
d  dbxGetFrCtyLst    pr
*-----*
d  Position            16    const
d  ListType            1     const
d  CountReq           10i 0  const
d  CountRet           10i 0
d  CityList            likeds(frCityRow) dim(100)
d                    options(*varsize)

*-----*
d  dbxGetToCtyLst    pr
*-----*
d  Position            16    const
d  ListType            1     const
d  CountReq           10i 0  const
d  CountRet           10i 0
d  CityList            likeds(toCityRow) dim(100)
d                    options(*varsize)

*-----*
d  dbxGetFltLst      pr
*-----*
d  inFromCity         16    const
d  inToCity           16    const
d  inFlightDoW        16    const
d  outFlightCnt       10i 0
d  outFlights         likeds(FlightsRow) dim(50)

*-----*
d  dbxGetFlt          pr                likeds(flightsRow)
*-----*
d  inFlightNum        like(flightsRow.flight_no)
d  outSqlState        5a
d  outSqlMsg          256a

```

NFSSQL (SQL I/O procedures)

```
h nomain bnmdir('FLGHT400M')

/copy nfsSQLpr

*-----*
d closeCursor      pr
*-----*
d cursorID          10a  const
*-----*
d getSQLDiagMsg    pr
*-----*
d  outSqlMsg       256a

*-----*
*
* Order SQL I/O procedures
*-----*

*****
p dbxGetOrd        b          export
*****
* Returns one row from ORDERS table that matches the
* specified order number

d dbxGetOrd        pi          likeds(orderRow)
d inOrderNumber    9b 0  const
d outSqlState      5a
d outSqlMsg        256a

d outOrderRow      ds          likeds(orderRow)

C/EXEC SQL
c+  select * into :outOrderRow
c+  from allOrders
c+  where order_Number = :inOrderNumber
C/END-EXEC

/free
outSqlState = sqlState;
outSqlMsg = *blanks;
if %subst(sqlState:1:2) <> '00';
  getSQLDiagMsg(outSqlMsg);
  clear outOrderRow;
endif;
return outOrderRow;
/end-free

p dbxGetOrd        e

*****
p dbxUpdOrd        b          export
*****
* Updates the row with the specified order number

d dbxUpdOrd        pi          5a
d inOrderNumber    9B 0  const
```

```

d inOrderRow                                likeds(orderRow)
d outSqlMsg                                  256a

d outSqlState    s                            5a

C/EXEC SQL
c+  update allOrders
c+  set row = :inOrderRow
c+  where order_Number = :inOrderNumber
C/END-EXEC

/free
outSqlState = sqlState;
outSqlMsg = *blanks;
if %subst(sqlState:1:2) <> '00';
    getSQLDiagMsg(outSqlMsg);
endif;
return outSqlState;
/end-free

p dbxUpdOrd    e

*****
p dbxDelOrd    b                            export
*****
* Deletes the row with the specified order number

d dbxDelOrd    pi                            5a
d inOrderNumber 9B 0 const
d outSqlMsg    256a

d outSqlState    s                            5a

C/EXEC SQL
c+  delete from allOrders
c+  where order_Number = :inOrderNumber
C/END-EXEC

/free
outSqlState = sqlState;
outSqlMsg = *blanks;
if %subst(sqlState:1:2) <> '00';
    getSQLDiagMsg(outSqlMsg);
endif;
return outSqlState;
/end-free

p dbxDelOrd    e

*****
p dbxNxtOrdNum    b                            export
*****
* Returns the next order number

d dbxNxtOrdNum    pi                            like(orderRow.order_no)
d outSqlState    5a
d outSqlMsg    256a

d nextOrdNum    s                            like(orderRow.order_no)

/free

```



```

    nextOrdNum = 0;
/end-free

* Use SQL sequence to generate next order number
C/EXEC SQL
c+   VALUES NEXT VALUE FOR order_number
c+   INTO :nextOrdNum
C/END-EXEC

/free
    outSqlState = sqlState;
    outSqlMsg = *blanks;
    if %subst(sqlState:1:2) <> '00';
        getSQLDiagMsg(outSqlMsg);
    endif;
    return nextOrdNum;
/end-free

p dbxNxtOrdNum      e

*****
p dbxInsOrd         b                               export
*****
* Inserts a new row into ORDER table

d dbxInsOrd         pi
d inOrdRow          5a                             likeds(orderRow)
d outSqlState       256a
d outSqlMsg

C/EXEC SQL
c+   insert into allOrders
c+   values (:inOrdRow)
C/END-EXEC

/free
    outSqlState = sqlState;
    outSqlMsg = *blanks;
    if %subst(sqlState:1:2) <> '00';
        getSQLDiagMsg(outSqlMsg);
    endif;
/end-free

p dbxInsOrd        e

*
*
* Customer SQL I/O procedures
*
*
*****
p dbxCusLst         b                               export
*****
* Returns multiple rows from CUSTOMERS table that are
* at or beyond the specified starting customer name

d dbxCusLst         pi
d custPosition      64                             const
d listType          1                             const

```

```

d countReq          10i 0 const
d countRet          10i 0
d outputType        1      const
d outCustomers      likeds(CustomerRow) dim(100)
d                  options(*varsize)

d SQLStatement      s          512a
d tableID           s          10a  inz('Cust')
d cursorID          s          4a
d i                 s          10i 0

* ListType options:
* S Start a new list beginning at the position specified
* N Start the list at the position just after the value specified
* M Return only those values that start with the position specified
* C Continue from the last position

* Output Type options:
* P Return results in output parameter (array)
* R Return results in a results set

* Construct the SQL Statement

/free
CountRet = 0;

if ListType <> 'C';
  closeCursor('CusLstCN ');
/end-free

C/EXEC SQL
C+ declare cursorCusLstCN CURSOR FOR
C+ select *
C+ from allCusts
C+ where Customer_Name >= :custPosition
C+ order by customer_name
C+ optimize for 10 rows
C/END-EXEC
C/EXEC SQL
C+ OPEN cursorCusLstCN
C/END-EXEC SQL

/free
endif;

select;
when outputType = 'P';
  for i = 1 to CountReq;
    if sqlcod <> 0;
      leave;
    endif;

/end-free

C/EXEC SQL
C+ FETCH cursorCusLstCN into :customerRow :customerNIArr
C/END-EXEC

/free

```

```

        if sqlcod <> 0;
            leave;
        endif;
        CountRet = CountRet + 1;
        outCustomers(CountRet) = customerRow;
    endfor;

    when outputType = 'R';
/end-free

C/EXEC SQL
C+   SET RESULT SETS CURSOR cursorCusLstCN
C/END-EXEC SQL
/free
endsl;
/end-free

p dbxGetCusLst     e

*****
p dbxGetCusByNum  b          export
*****
* Returns one row from CUSTOMERS table that matches the
* specified customer number
d dbxGetCusByNum  pi          likeds(CustomerRow)
d inCusNumber    5a          like(customerRow.cust_no) const
d outSqlState    5a
d outSqlMsg      256a

d outCusRow      ds          likeds(CustomerRow)

C/EXEC SQL
c+   select * into :outCusRow :customerNIarr
c+   from allCusts
c+   where customer_No = :inCusNumber
C/END-EXEC

/free
outSqlState    = sqlState;
outSqlMsg      = *blanks;
if %subst(sqlState:1:2) <> '00';
    getSQLDiagMsg(outSqlMsg);
    clear outCusRow;
endif;
return outCusRow;
/end-free

p dbxGetCusByNum  e

*****
p dbxGetCusByNam  b          export
*****
* Returns one row from CUSTOMERS table that matches the
* specified customer name
d dbxGetCusByNam  pi          likeds(CustomerRow)
d inCusName      64a          const
d outSqlState    5a
d outSqlMsg      256a

d outCustRow     ds          likeds(CustomerRow)

```

```

C/EXEC SQL
c+  select * into :outCustRow :customerNIArr
c+  from allCusts
c+  where customer_Name = :inCusName
C/END-EXEC

/free
outSqlState = sqlState;
outSqlMsg = *blanks;
if %subst(sqlState:1:2) <> '00';
  getSQLDiagMsg(outSqlMsg);
  clear outCustRow;
endif;
return outCustRow;
/end-free

p dbxGetCusByNam e

*****
P dbxUpdCus      b      export
*****

d dbxUpdCus      pi
d inCusNumber    like(customerRow.cust_no) const
d inCusRow       likeds(CustomerRow)
d outSqlState    5a
d outSqlMsg      256a

C/EXEC SQL
c+  update allCusts
c+  set row = :inCusRow
c+  where customer_no = :inCusNumber
C/END-EXEC

/free
outSqlState = sqlState;
outSqlMsg = *blanks;
if %subst(sqlState:1:2) <> '00';
  getSQLDiagMsg(outSqlMsg);
endif;
/end-free

p dbxUpdCus      e

*****
P dbxUpdCus2    b      export
*****

d dbxUpdCus2    pi
d customer_no   like(CustomerRow.cust_no)
d customer_name like(CustomerRow.cust_name)
d address       like(CustomerRow.address)
d city          like(CustomerRow.city)
d state         like(CustomerRow.state)
d zipCode       like(CustomerRow.zipcode)
d telephone    like(CustomerRow.telephone)
d creditCard    like(CustomerRow.cred_card)
d CC_Number     like(CustomerRow.cc_number)
d exp_Date      like(CustomerRow.exp_date)
d pref_Airline  like(CustomerRow.pref_airln)
d ff_Number     like(CustomerRow.ff_Number)

```

```

d  outSqlState          5a
d  outSqlMsg            256a

* Update the customer row
C/EXEC SQL
c+  update allCusts
c+  set customer_no = :customer_no,
c+  customer_name = :customer_name,
c+  address = :address,
c+  city = :city,
c+  state = :state,
c+  zipcode = :zipCode,
c+  telephone = :telephone,
c+  credit_card = :creditcard,
c+  cc_number = :cc_number,
c+  exp_date = :exp_date,
c+  pref_airline_id = :pref_airline,
c+  ff_Number = :ff_Number
c+  where customer_no = :customer_no
C/END-EXEC

/free
outSqlState = sqlState;
outSqlMsg = *blanks;
if %subst(sqlState:1:2) <> '00';
  getSQLDiagMsg(outSqlMsg);
endif;
/end-free

*
p dbxUpdCus2          e
*****
p dbxInsCus          b          export
*****
* Inserts a new row into CUSTOMERS table

d dbxInsCus          pi
d inCusRow                                likeds(CustomerRow)
d outSqlState          5a
d outSqlMsg            256a

d NextCustNum        s          9b 0

* Use SQL sequence to generate next customer number
C/EXEC SQL
c+  VALUES NEXT VALUE FOR customer_number
c+  INTO :nextCustNum
C/END-EXEC

/free
outSqlState = sqlState;
outSqlMsg = *blanks;
if %subst(sqlState:1:2) <> '00';
  getSQLDiagMsg(outSqlMsg);
  inCusRow.cust_no = -1;
  return;
else;
  inCusRow.cust_no = nextCustNum;
endif;
/end-free

C/EXEC SQL

```

```

c+ insert into allCusts
c+ values (:inCusRow)
C/END-EXEC

/free
outSqlState = sqlState;
outSqlMsg = *blanks;
if %subst(sqlState:1:2) <> '00';
    getSQLDiagMsg(outSqlMsg);
endif;
/end-free

p dbxInsCus e

*****
p dbxInsCus2 b export
*****

d dbxInsCus2 pi
d customer_name like(CustomerRow.cust_name)
d address like(CustomerRow.address)
d city like(CustomerRow.city)
d state like(CustomerRow.state)
d zipCode like(CustomerRow.zipcode)
d telephone like(CustomerRow.telephone)
d creditCard like(CustomerRow.cred_card)
d CC_Number like(CustomerRow.cc_number)
d exp_Date like(CustomerRow.exp_date)
d pref_Airline like(CustomerRow.pref_airln)
d ff_Number like(CustomerRow.ff_Number)
d outSqlState 5a
d outSqlMsg 256a

d Apostrophe C ' ' ' '

d SQLStatement s 512
d nextCustNum2 s like(CustomerRow.cust_no)
d customer_no s like(CustomerRow.cust_no)

* Useq SQL sequence to generate next customer number
C/EXEC SQL
c+ VALUES NEXT VALUE FOR customer_number
c+ INTO :nextCustNum2
C/END-EXEC

/free
outSqlState = sqlState;
outSqlMsg = *blanks;
if %subst(sqlState:1:2) <> '00';
    getSQLDiagMsg(outSqlMsg);
    customer_no = -1;
    return;
else;
    customer_no = nextCustNum2;
endif;
/end-free

* Insert the new row
C/EXEC SQL
c+ insert into allCusts
c+ values (:customer_no, :customer_name, :address, :city,
c+ :state, :zipcode, :telephone, :creditCard,

```

```

c+          :cc_number, :exp_date, :pref_airline, :ff_Number)
C/END-EXEC

/free
  outSqlState  = sqlState;
  outSqlMsg = *blanks;
  if %subst(sqlState:1:2) <> '00';
    getSQLDiagMsg(outSqlMsg);
  endif;
/end-free

p dbxInsCus2      e

//*****
p dbxGetAllCus    b          export
//*****
* Returns all rows from CUSTOMERS table to result set

d dbxGetAllCus    pi

C/EXEC SQL
C+  DECLARE allCusCursor CURSOR FOR
C+  SELECT customer_name, customer_no, credit_card
C+  FROM allCusts
C+  ORDER BY customer_name
C/END-EXEC SQL

C/EXEC SQL
C+  OPEN allCusCursor
C/END-EXEC SQL

C/EXEC SQL
C+  SET RESULT SETS CURSOR allCusCursor
C/END-EXEC SQL

p dbxGetAllCus    e

*****
p dbxGetCusWCNam  b          export
*****
* Finds all rows from CUSTOMERS table
* that match a wildcard search by name.
* Results returned to a results set

d dbxGetCusWCNam  pi
d  inLastName      64

d Apostrophe      C          ' ' ' '
d LastNameKey     s          64
d FetchName       s          64
d SQLStatement    s          512

c          eval          LastNameKey = %trim(inLastName) + '%'
* Construct the SQL Statement
C          Eval          SQLStatement = 'SELECT customer_name, +
c          customer_no, credit_card +
c          from flight400m2/allCusts      +
c          where UPPER(customer_name) +
c          like UPPER(' +
c          Apostrophe +

```

```

c                                     %trim(inLastName) + '%' +
c                                     Apostrophe +
c                                     ') Order By customer_name'

C/EXEC SQL
C+ PREPARE allCusStatement FROM :SQLStatement
C/END-EXEC
*
* Declare the SQL cursor to hold the data retrieved from the SELECT
C/EXEC SQL
C+ DECLARE CustLstNamCursor CURSOR FOR allCusStatement
C/END-EXEC
*
* Open the SQL cursor.
C/EXEC SQL
C+ OPEN CustLstNamCursor
C/END-EXEC SQL
*
c*                                     enddo

*
C/EXEC SQL
C+ SET RESULT SETS CURSOR CustLstNamCursor
C/END-EXEC SQL

p dbxGetCusWCNam e

*****
p dbxGetOrdCusCN b export
*****
* Returns multiple rows from Orders and Customers tables
* that are at or beyond the specified starting customer name

d dbxGetOrdCusCN pi
d custPosition          64   const
d listType              1   const
d countReq             10i 0 const
d countRet             10i 0
d outputType           1   const
d ordCustList          likeds(ordCustRow) dim(100)

d i                    s    10i 0
d cursorID             s    4a

* ListType options:
* S Start a new list beginning at the position specified
* N Start the list at the position just after the value specified
* M Return only those values that start with the position specified
* C Continue from the last position

* Output Type options:
* P Return results in output parameter (array)
* R Return results in a results set

* If listType is not C (Continue fetching rows in the cursor)
* declare and open the cursor

/free
CountRet = 0;
if listType <> 'C';
closeCursor('OrdCusCN ');

```



```

/end-free

C/EXEC SQL
C+  declare cursorOrdCusCN CURSOR FOR
C+  select *
C+  from allOrdCust
C+  where customer_name >= :custPosition
C+  order by customer_name
C+  optimize for 10 rows
C/END-EXEC
C/EXEC SQL
C+  OPEN cursorOrdCusCN
C/END-EXEC SQL

/free
endif;

select;
when outputType = 'P';
  for i = 1 to CountReq;
    if sqlcod <> 0;
      leave;
    endif;
  endfor;
/end-free

C/EXEC SQL
C+  FETCH cursorOrdCusCN into :ordCustRow
C/END-EXEC

/free
  if sqlcod <> 0;
    leave;
  endif;
  CountRet = CountRet + 1;
  ordCustList(CountRet) = ordCustRow;
endfor;

when outputType = 'R';
/end-free

C/EXEC SQL
C+  SET RESULT SETS CURSOR cursorOrdCusCN
C/END-EXEC SQL

/free
endsl;
return;

/end-free

p dbxGetOrdCusCN e

*****
p dbxGetOrdCusTS b export
*****
* Returns multiple rows from Orders and Customers tables
* that are at or beyond the specified starting timestamp

d dbxGetOrdCusTS pi

```

```

d  tsPosition          26      const
d  listType            1       const
d  countReq            10i 0   const
d  countRet            10i 0
d  outputType          1       const
d  ordCustList         likeds(ordCustRow) dim(100)

d  i                   s       10i 0

d  OrdCust             ds       likeds(ordCustRow) dim(100)

*  ListType options:
*  S  Start a new list beginning at the position specified
*  N  Start the list at the position just after the value specified
*  M  Return only those values that start with the position specified
*  C  Continue from the last position

*  Output Type options:
*  P  Return results in output parameter (array)
*  R  Return results in a results set

*  If listType is not C (Continue fetching rows in the cursor)
*  declare and open the cursor

/free
CountRet = 0;
if listType <> 'C';
    closeCursor('OrdCusTS ');
/end-free

C/EXEC SQL
C+  declare cursorOrdCusTS  CURSOR FOR
c+  select *
c+  from allOrdCust
c+  where departure_date >= :tsPosition
c+  order by departure_date
c+  optimize for 10 rows
C/END-EXEC
C/EXEC SQL
C+  OPEN cursorOrdCusTS
C/END-EXEC SQL

/free
endif;

select;
when outputType = 'P';
    for i = 1 to CountReq;
        if sqlcod <> 0;
            leave;
        endif;
    /end-free

C/EXEC SQL
C+  FETCH cursorOrdCusTS into :ordCustRow
C/END-EXEC

/free
if sqlcod <> 0;
    leave;

```

```

        endif;
        CountRet = CountRet + 1;
        ordCustList(CountRet) = ordCustRow;
    endfor;

    when outputType = 'R';
/end-free

C/EXEC SQL
C+ SET RESULT SETS CURSOR cursorOrdCusTS
C/END-EXEC SQL

/free
endsl;
return;
/end-free

p dbxGetOrdCusTS e

```

*

```

*
* Flights SQL I/O procedures

```

*

```

*****
p dbxGetFltLst      b                export
*****
* Returns an array of up to 50 rows from flights table that match the
* specified departure city, arrival city, and day of the week

d dbxGetFltLst      pi
d   inFromCity      16      const
d   inToCity        16      const
d   inFlightDOW     16      const
d   outFlightCnt    10i 0
d   outFlights      likeds(FlightsRow) dim(50)

C/EXEC SQL
C+ declare FlightsCursor cursor for
C+ select *
C+ from allFlights
C+ where departure = :inFromCity
C+    and arrival = :inToCity
C+    and day_of_week = :inFlightDOW
C+ order by flight_number
C/END-EXEC

C/EXEC SQL
C+ open flightsCursor
C/END-EXEC

/free
for outFlightCnt = 1 to 50;
/end-free

C/EXEC SQL

```

```

C+  fetch flightsCursor INTO :flightsRow
C/END-EXEC

/free
  if %subst(sqlState:1:2) <> '00';
    leave;
  endif;
  outFlights(outFlightCnt) = flightsRow;
endfor;
outFlightCnt = outFlightCnt - 1;
/end-free

C/EXEC SQL
C+  close flightsCursor
C/END-EXEC

p dbxGetFltLst      e

*****
p dbxGetFlt        b                export
*****
* Returns one row from flights table that matches the
* specified flight number

d dbxGetFlt        pi                likeds(flightsRow)
d inFlightNum      like(flightsRow.flight_no)
d outSqlState      5a
d outSqlMsg        256a

d outFlightRow    ds                likeds(flightsRow)

C/EXEC SQL
c+  select * into :outFlightRow
c+  from allFlights
c+  where flight_number = :inFlightNum
C/END-EXEC

/free
  outSqlState = sqlState;
  outSqlMsg = *blanks;
  if %subst(sqlState:1:2) <> '00';
    getSQLDiagMsg(outSqlMsg);
    clear outFlightRow;
  endif;
  return outFlightRow;
/end-free

p dbxGetFlt        e

```

```

*
*
* City SQL I/O procedures

```

```

*****
p dbxGetFrCty      b                export
*****
* Returns one row from FRCITY table that matches the
* specified departure city initials

```

```

d dbxGetFrCty      pi                likeds(frCityRow)
d inCityInt       3a                const
d outSqlState     5a
d outSqlMsg       256a

d outFrCityRow    ds                likeds(frCityRow)

C/EXEC SQL
c+ select * into :outFrCityRow
c+   from fromCities
c+   where frcint = :inCityInt
C/END-EXEC

/free
outSqlState = sqlState;
outSqlMsg = *blanks;
if %subst(sqlState:1:2) <> '00';
  getSQLDiagMsg(outSqlMsg);
  clear outFrCityRow;
endif;
return outFrCityRow;
/end-free

p dbxGetFrCty      e

*****
p dbxGetToCty      b                export
*****
* Returns one row from TOCITY table that matches the
* specified arrival city initials

d dbxGetToCty      pi                likeds(toCityRow)
d inCityInt       3a                const
d outSqlState     5a
d outSqlMsg       256a

d outToCityRow    ds                likeds(toCityRow)

C/EXEC SQL
c+ select * into :outToCityRow
c+   from toCities
c+   where tocint = :inCityInt
C/END-EXEC

/free
outSqlState = sqlState;
outSqlMsg = *blanks;
if %subst(sqlState:1:2) <> '00';
  getSQLDiagMsg(outSqlMsg);
  clear outToCityRow;
endif;
return outToCityRow;
/end-free

p dbxGetToCty      e

*****
p dbxGetFrCtyLst  b                export
*****
* Returns multiple rows from FRCITY table that are
* at or beyond the specified starting departure

```

```

* city name

d dbxGetFrCtyLst  pi
d  Position              16  const
d  ListType            1  const
d  CountReq           10i 0  const
d  CountRet           10i 0
d  frCityList          liked(frCityRow) dim(100)
d                      options(*varsize)

d i                      s          10i 0
d mlength               s          10i 0

/free
  select;
  when ListType = 'S' or ListType = 'M';
/end-free
C/EXEC SQL
C+  declare frCityCursor cursor for
c+  select *
c+  from fromCities
c+  where frcnam >= :Position
c+  order by frcnam
c+  optimize for 10 rows
C/END-EXEC
C/EXEC SQL
C+  open frCityCursor
C/END-EXEC
/free
  when ListType = 'N';
/end-free
C/EXEC SQL
C+  declare frCityCursor2 cursor for
c+  select *
c+  from fromCities
c+  where frcnam > :Position
c+  order by frcnam
c+  optimize for 10 rows
C/END-EXEC
C/EXEC SQL
C+  open frCityCursor2
C/END-EXEC
/free
  ends1;
/end-free

/free
  if ListType = 'M';
  mlength = %len(%trimr(Position));
  endif;
  CountRet = 0;
  for i = 1 to CountReq;
  select;
  when ListType = 'S' or ListType = 'M';
/end-free
C/EXEC SQL
C+  fetch frCityCursor INTO :frCityRow
C/END-EXEC
/free
  when ListType = 'N';
/end-free

```

```

C/EXEC SQL
C+  fetch frCityCursor2 INTO :frCityRow
C/END-EXEC
/free
  ends1;

  if %subst(sqlState:1:2) <> '00' or
    (ListType = 'M' and
     %subst(Position:1:mlength) <> %subst(frCityRow.frctnam:1:mlength));
    leave;
  endif;
  CountRet = CountRet + 1;
  frCityList(CountRet) = frCityRow;
endfor;

select;
  when ListType = 'S' or ListType = 'M';
/end-free
C/EXEC SQL
C+  close frCityCursor
C/END-EXEC

/free
  when ListType = 'N';
/end-free
C/EXEC SQL
C+  close frCityCursor2
C/END-EXEC
/free
  ends1;
/end-free

p dbxGetFrCtyLst  e
*****
p dbxGetToCtyLst  b          export
*****
* Returns multiple rows from TOCITY table that are
* at or beyond the specified starting arrival
* city name

d dbxGetToCtyLst  pi
d Position                16      const
d ListType              1        const
d CountReq              10i 0    const
d CountRet              10i 0
d ToCityList              likeds(toCityRow) dim(100)
d                          options(*varsize)

d i                        s          10i 0
d mlength                  s          10i 0

/free
  select;
  when ListType = 'S' or ListType = 'M';
/end-free
C/EXEC SQL
C+  declare ToCityCursor cursor for
C+  select *
C+  from toCities
C+  where tocnam >= :Position
C+  order by tocnam
C+  optimize for 10 rows

```

```

C/END-EXEC
C/EXEC SQL
C+ open toCityCursor
C/END-EXEC
/free
    when ListType = 'N';
/end-free
C/EXEC SQL
C+ declare toCityCursor2 cursor for
c+ select *
c+ from toCities
c+ where tocnam > :Position
c+ order by tocnam
c+ optimize for 10 rows
C/END-EXEC
C/EXEC SQL
C+ open toCityCursor2
C/END-EXEC
/free
    ends1;
/end-free

/free
    if ListType = 'M';
        mlength = %len(%trimr(Position));
    endif;
    CountRet = 0;
    for i = 1 to CountReq;
        select;
            when ListType = 'S' or ListType = 'M';
        /end-free
C/EXEC SQL
C+ fetch toCityCursor INTO :toCityRow
C/END-EXEC
/free
    when ListType = 'N';
/end-free
C/EXEC SQL
C+ fetch toCityCursor2 INTO :toCityRow
C/END-EXEC
/free
    ends1;
    if %subst(sqlState:1:2) <> '00' or
        (ListType = 'M' and
         %subst(Position:1:mlength) <> %subst(toCityRow.tocnam:1:mlength));
        leave;
    endif;
    CountRet = CountRet + 1;
    toCityList(CountRet) = toCityRow;
endfor;

select;
    when ListType = 'S' or ListType = 'M';
/end-free
C/EXEC SQL
C+ close toCityCursor
C/END-EXEC

/free
    when ListType = 'N';
/end-free

```



```

C/EXEC SQL
C+   close toCityCursor2
C/END-EXEC
/free
  ends1;
/end-free

p dbxGetToCtyLst  e

*****
p closeCursor    b
*****
* Closes the Cursor ordCustCursor

d closeCursor    pi
d  cursorID      10      const

/free
  select;
  when cursorID = 'OrdCusCN  ';
/end-free
* Close the SQL cursor (in case it is already open)
C/EXEC SQL
C+   close cursorOrdCusCN
C/END-EXEC

/free
  when cursorID = 'OrdCusTS  ';
/end-free
* Close the SQL cursor (in case it is already open)
C/EXEC SQL
C+   close cursorOrdCusTS
C/END-EXEC

/free
  when cursorID = 'CusLstCN  ';
/end-free
* Close the SQL cursor (in case it is already open)
C/EXEC SQL
C+   close cursorCusLstCN
C/END-EXEC

/free
  ends1;
/end-free

p closeCursor    e

*
* _____
*   Get SQL Diagnostic message
*
p getSQLDiagMsg  b

d getSQLDiagMsg  pi
d  SQLmsg        256A
*
d cst_name       s      128A
d i              s      10i 0
*
/free
  cst_name = *blanks;

```

```

    sqlMsg = *blanks;
/end-free
*
C/EXEC SQL
C+   GET DIAGNOSTICS
C+   :cond_count = NUMBER
C/END-EXEC

/free
for i = 1 to cond_Count;
/end-free

C/EXEC SQL
C+   GET DIAGNOSTICS CONDITION :i
C+   :cst_name = CONSTRAINT_NAME,
C+   :sqlMsg = MESSAGE_TEXT
C/END-EXEC

/free
  dsply sqlMsg;
endfor;
/end-free
p getSQLDiagMsg e

```

Programs and modules converted from RLA to SQL I/O procedures

NFS001

```

h nomain bnddir('FLGHT400M')

D RCVM0100      DS          qualified
D   BytesRtn           10I 0
D   BytesAvail         10I 0
D   MsgSev             10I 0
D   MsgID              7A
D   MsgType            2A
D   MsgKey             4A
D                   7A
D   CCSID_status       10I 0
D   CCSID              10I 0
D   MsgDtaLen          10I 0
D   MsgDtaAvail        10I 0
D   MsgDta             8000A

D ErrorCode       ds          qualified
D   BytesProv           10I 0 inz(0)
D   BytesAvail         10I 0 inz(0)

D Reply          s           100A

/copy nfs001pr
/copy nfs400pr
/copy nfsSQLpr

*-----
d rcvMsg          pr          extpgm('QMHRVCVPM')
*-----
D   MsgInfo           32767A options(*varsize)
D   MsgInfoLen        10I 0 const
D   Format             8A const

```

```

D StackEntry          10A  const
D StackCount         10I 0  const
D MsgType            10A  const
D MsgKey              4A  const
D WaitTime           10I 0  const
D MsgAction           10A  const
D ErrorCode           8000A options(*varsize)

*-----*
p GetTimeStamp      pr          z
*-----*
d DateChars         8  value
d TimeChars         8  value

*-----*
d CheckOrder        pr          10i 0
*-----*
d OrderInfo                    liked(ReserveInfo) const

*-----*
d ConvertRecord     pr                    liked(ReserveInfo)
*-----*

*-----*
d ConvertOrder      pr
*-----*
d OrderInfo                    liked(ReserveInfo) const

*-----*
d SendMessage       pr
*-----*
d peMsg             256A  const

*-----*
p GetTimeStamp      b
*-----*
d GetTimeStamp      pi          z
d DateChars         8  value
d TimeChars         8  value
/free
return %timestamp(%char(%date(DateChars:*mdy)) + '-' +
                 %char(%time(TimeChars:*usa)) + '.000000');
/end-free
p GetTimeStamp      e

*-----*
p CheckOrder        b
*-----*
d CheckOrder        pi          10i 0
d OrderInfo                    liked(ReserveInfo) const
e*
d Flight            ds                    liked(FlightInfo)
d DepartInfo        s          z
/free
GetFlightInfo(OrderInfo.FlightNumber:Flight);
if Flight.Flight = *blank;
return -1;
endif;
DepartInfo = GetTimeStamp(OrderInfo.DepartDate:OrderInfo.DepartTime);
if %time(Flight.DepartTime:*usa) <> %time(DepartInfo);
return -2;

```

```

endif;
if OrderInfo.Tickets < 1;
    return -3;
endif;
return 0;
/end-free
p CheckOrder      e

*-----
p ConvertRecord   b
*-----
d ConvertRecord   pi          likeds(ReserveInfo)
d*
d OrderInfo       ds          likeds(ReserveInfo)
/free
clear OrderInfo;

OrderInfo.AgentNumber = orderRow.AGENT_NO;          // dbmodc //
OrderInfo.CustNumber = orderRow.CUST_NO;           //dbmc //
OrderInfo.FlightNumber = %char(orderRow.FLIGHT_NO); //dbmc //
OrderInfo.Tickets = orderRow.TICKS_ORD;
select;
    when orderRow.CLASS = '1';
        OrderInfo.ServiceClass = 'F';
    when orderRow.CLASS = '2';
        OrderInfo.ServiceClass = 'B';
    when orderRow.CLASS = '3';
        OrderInfo.ServiceClass = 'C';
endsl;
OrderInfo.DepartDate = %char(%date(orderRow.DEPAR_DATE):*mdy);
OrderInfo.DepartTime = %char(%date(orderRow.DEPAR_DATE):*usa);

return OrderInfo;
/end-free
p ConvertRecord   e

*-----
p ConvertOrder    b
*-----
d ConvertOrder    pi          likeds(ReserveInfo) const
d OrderInfo       ds          likeds(ReserveInfo) const
/free
orderRow.agent_no = OrderInfo.AgentNumber;
orderRow.cust_no = OrderInfo.CustNumber;
orderRow.flight_no = %dec(OrderInfo.FlightNumber:7:0);
orderRow.ticks_ord = OrderInfo.Tickets;
select;
    when OrderInfo.ServiceClass = 'F';
        orderRow.class = '1';
    when OrderInfo.ServiceClass = 'B';
        orderRow.class = '2';
    other;
        orderRow.class = '3';
endsl;
orderRow.depar_date = GetTimeStamp(OrderInfo.DepartDate:
                                OrderInfo.DepartTime);

orderRow.send_sig = 'N';
/end-free
p ConvertOrder    e

*****
p ComputePrice    b          export

```

```

*****
d ComputePrice      pi
d   BasePrice          3      const
d   ServiceClass      1      const
d   Tickets            3 0    const
d   Price              7 2
d   Tax                5 2
d   TotalDue           7 2
/free
Price = %dec(BasePrice:7:2);
select;
  when ServiceClass = 'F';
    Price = Price * 3;
  when ServiceClass = 'B';
    Price = Price * 2;
endsl;
if Tickets > 1;
  Price = Price * Tickets;
endif;
Tax = Price * 0.04;
TotalDue = Price + Tax;
/end-free
p ComputePrice      e

*****
p ReserveFlight     b      export
*****
d ReserveFlight     pi
d   OrderInfo              likeds(ReserveInfo) const
d   OrderNumber           9B 0
d   outSqlState           5a
d   outSqlMsg             256a

d   newOrderRow           ds      likeds(orderRow)

/free
OrderNumber = CheckOrder(OrderInfo);
if OrderNumber <> 0;
  return;
endif;

OrderNumber = dbxNxtOrdNum(outSqlState:outSqlMsg);
ConvertOrder(OrderInfo);
orderRow.order_no = OrderNumber;
//orderRow.order_no = 5671303;
//orderRow.cust_no = 9897983;
newOrderRow = OrderRow;
dbxInsOrd(newOrderRow : outsqlState: outsqlMsg);
/end-free
p ReserveFlight     e

*****
p FindOrderCust     b      export
*****
d FindOrderCust     pi
d   custPosition     64      const
d   ListType         1      const
d   CountReq         10i 0  const
d   CountRet         10i 0
d   OrderList                likeds(OrderSummary) dim(100)

d   ordCustList           ds      likeds(ordCustRow) dim(100)

```

```

d i          s          10i 0

d outputType s          1a  inz('P')

/free
  dbxGetOrdCusCN(custPosition :
                ListType :
                CountReq :
                CountRet :
                outputType :
                ordCustList);
  for i = 1 to CountReq;
    OrderList(i).OrderNumber = ordCustList(i).order_no;
    OrderList(i).CustName = ordCustList(i).cust_name;
    OrderList(i).DepartDate =
      %char(%date(ordCustList(i).depar_date):*mdy);
  endfor;
/end-free
p FindOrderCust  e

*****
p FindOrderDate  b          export
*****
d FindOrderDate  pi
d Position          8      const
d ListType        1      const
d CountReq        10i 0  const
d CountRet        10i 0
d OrderList          likeds(OrderSummary) dim(100)
d*
d ordCustList      ds          likeds(ordCustRow) dim(100)
d i          s          10i 0
d timeStamp      s          z
d tsPosition     s          26a
d outputType     s          1a  inz('P')

/free
timeStamp = GetTimeStamp(Position:'12:01 AM');
tsPosition = %char(timestamp);
dbxGetOrdCusTS(tsPosition :
              ListType :
              CountReq :
              CountRet :
              outputType :
              ordCustList);
for i = 1 to CountReq;
  OrderList(i).OrderNumber = ordCustList(i).order_no;
  OrderList(i).CustName = ordCustList(i).cust_name;
  OrderList(i).DepartDate =
    %char(%date(ordCustList(i).depar_date):*mdy);
endfor;
/end-free
p FindOrderDate  e

*****
p GetOrderInfo   b          export
*****
d GetOrderInfo   pi
d OrderNumber    9B 0  const
d OrderInfo      likeds(ReserveInfo)

```

```

d outSqlState      s              5a
d outSqlMsg        s              256a

/free
clear OrderInfo;
orderRow = dbxGetOrd(OrderNumber : outSqlState : outSqlMsg);

if %subst(outSqlState:1:2) = '00';
  OrderInfo = ConvertRecord();
endif;
/end-free
p GetOrderInfo      e

*****
p UpdateOrder       b              export
*****
d UpdateOrder       pi
d OrderNumber       9B 0 const
d OldOrder          likeds(ReserveInfo) const
d NewOrder          likeds(ReserveInfo) const
d outSqlState       5a
d outSqlMsg         256a
/free
if CheckOrder(NewOrder) <> 0;
  return;
endif;

orderRow = dbxGetOrd(OrderNumber : outSqlState : outSqlMsg);
if %subst(outSqlState:1:2) = '00'
  and OldOrder = ConvertRecord();
  ConvertOrder(NewOrder);
  outSqlState = dbxUpdOrd(OrderNumber : orderRow : outSqlMsg);
endif;
/end-free
p UpdateOrder       e

*****
p DeleteOrder       b              export
*****
d DeleteOrder       pi
d OrderNumber       9B 0 const
d outSqlState       5a
d outSqlMsg         256a
/free
  outSqlState = dbxDelOrd(OrderNumber : outSqlMsg);
/end-free
p DeleteOrder       e
*
*
* _____
*   Send Message to joblog
*
* _____
*
p SendMessage       b
d SendMessage       pi
D   peMsg           256A  const

D SndPgmMsg         PR              ExtPgm('QMHSNDPM')
D MessageID         7A              Const
D QualMsgF          20A             Const
D MsgData           256A            Const
D MsgDtaLen         10I 0           Const

```

```

D   MsgType                10A   Const
D   CallStkEnt             10A   Const
D   CallStkCnt             10I 0  Const
D   MessageKey             4A
D   ErrorCode              32766A options(*varsize)
D dsEC                     DS
D dsECBytesP               1      4I 0 INZ(256)
D dsECBytesA               5      8I 0 INZ(0)
D dsECMsgID                9      15
D dsECReserv              16      16
D dsECMsgDta              17      256

D wwMsgLen                 S      10I 0
D wwTheKey                 S      4A

c                               eval      wwMsgLen = %len(%trimr(peMsg))
c                               if        wwMsgLen<1
c                               return
c                               endif

c                               callp     SndPgmMsg('CPF9897': 'QCPFMSG *LIBL':
c                               peMsg: wwMsgLen: '*INFO':
c                               '*PGMBDY': 1: wwTheKey: dsEC)
c                               return
p   SendMessage            e
*
```

NFS402

```

h   nomain   bnmdir('FLGHT400M')

/copy nfs402pr
/copy nfsSQLpr

*****
p   FindFromCities   b                               export
*****
d   FindFromCities   pi
d   Position         16      const
d   ListType         1      const
d   CountReq         10i 0  const
d   CountRet         10i 0
d   CityList         likeds(CityInfo) dim(100)
d                   options(*varsize)

d   frCityList       ds      likeds(frCityRow) dim(100)
d   i                 s      10i 0

/free
  dbxGetFrCtyLst(position: ListType: CountReq: CountRet: frCityList);
  for i = 1 to CountReq;
    CityList(i).name      = frCityList(i).frcnam;
    CityList(i).initials = frCityList(i).frcint;
    CityList(i).airline   = frCityList(i).frcaln;
  endfor;
/end-free

p   FindFromCities   e

*****
```



```

p FindToCities      b                                export
*****
d FindToCities      pi
d Position          16      const
d ListType          1      const
d CountReq          10i 0  const
d CountRet          10i 0
d CityList          likeds(CityInfo) dim(100)
d                  options(*varsize)

d toCityList        ds                                likeds(toCityRow) dim(100)
d i                 s                                10i 0

/free
  dbxGetToCtyLst(position: ListType: CountReq: CountRet: toCityList);
  for i = 1 to CountReq;
    CityList(i).name      = toCityList(i).tocnam;
    CityList(i).initials  = toCityList(i).tocint;
    CityList(i).airline   = *blanks;
  endfor;
/end-free
p FindToCities      e

*****
p GetCityName      b                                export
*****
d GetCityName      pi
d Initials         3      const
d FromTo           1      const
d Name             16

d outSqlState      s          5a
d outSqlMsg        s          256a

/free
Name = *blank;
if FromTo = 'F';
  frCityRow = dbxGetFrCty(Initials : outSqlState : outSqlMsg);
  if %subst(outSqlState:1:2) = '00';
    Name = frCityRow.FRCNAM;
  endif;
else;
  toCityRow = dbxGetToCty(Initials : outSqlState : outSqlMsg);
  if %subst(outSqlState:1:2) = '00';
    Name = toCityRow.TOCNAM;
  endif;
endif;
/end-free
p GetCityName      e

```

NFS404

```

  h nomain bnmdir('FLGHT400M')

/copy nfs404pr
/copy nfsutilpr
/copy nfsSQLpr

*****
p FindFlightsDoW  b                                export
*****

```

```

d FindFlightsDoW pi
d FromCity 16 const
d ToCity 16 const
d FlightDoW 16 const
d FlightCount 10i 0
d Flights likeds(FlightInfo) dim(50)
*
d flRows ds likeds(FlightsRow) dim(50)
*
d i s like(flightCount)
d FlightKey ds
d KFromCity 16
d KToCity 16
d KDoW 16
d Flight09 10i 0
/free
FlightCount = 0;
KFromCity = %trim(FromCity);
KToCity = %trim(ToCity);
KDoW = %trim(FlightDoW);

// call procedure to return all flights for the
// departure city, arrival city, and day of the week
// then load them into the return data structure
dbxGetFltLst(kFromCity : kToCity : kDOW: FlightCount : flRows);
for i = 1 to FlightCount;
    Flights(i).Airline = flRows(i).airlines;
    Flights(i).Flight = %char(flRows(i).flight_no);
    Flights(i).DoW = flRows(i).day_week;
    Flights(i).DepartCity = flRows(i).depar_int;
    Flights(i).ArriveCity = flRows(i).arriv_int;
    Flights(i).DepartTime = flRows(i).depar_time;
    Flights(i).ArriveTime = flRows(i).arriv_time;
    Flights(i).Price = flRows(i).ticket_prc;
endfor;
/end-free
p FindFlightsDoW e

*****
p FindFlights b export
*****
d FindFlights pi
d FromCity 16 const
d ToCity 16 const
d FlightDate 8 const
d FlightCount 10i 0
d Flights likeds(FlightInfo) dim(50)
/free
FindFlightsDoW(FromCity:ToCity:
    DayOfWeek(%dec(%subst(FlightDate:1:2):2:0):
        %dec(%subst(FlightDate:4:2):2:0):
        %dec(%subst(FlightDate:7:2):2:0) + 2000):
    FlightCount:Flights);
/end-free
p FindFlights e

*****
p GetFlightInfo b export
*****
d GetFlightInfo pi
d FlightNumber 7 const
d Flight likeds(FlightInfo)

```

```

d FlightKey      s                               like(flightsRow.flight_no)
d flRow          ds                             likeds(flightsRow)
d outSqlState    s                               5a
d outSqlMsg      s                               256a

/free
FlightKey = %dec(FlightNumber:7:0);
flRow = dbxGetFlt(FlightKey : outSqlState : outSqlMsg);
if %subst(outSqlState:1:2) = '00';
  Flight.Airline = flRow.airlines;
  Flight.Flight = %char(flRow.flight_no);
  Flight.DoW = flRow.day_week;
  Flight.DepartCity = flRow.depar_int;
  Flight.ArriveCity = flRow.arriv_int;
  Flight.DepartTime = flRow.depar_time;
  Flight.ArriveTime = flRow.arriv_time;
  Flight.Price = flRow.ticket_prc;
else;
  clear Flight;
endif;
/end-free
p GetFlightInfo e

```

NFS405

```

h nomain bnmdir('FLGHT400M')

/copy nfs405pr
/copy nfsSQLpr

//*****
p FindCustomers b export
//*****
d FindCustomers pi
d Position      64 const
d ListType      1 const
d CountReq      10i 0 const
d CountRet      10i 0
d CustList      likeds(CustInfo) dim(100)
d               options(*varsize)
//
d i             s 10i 0
d outputType    s 1 inz('P')
d custRows      ds likeds(CustomerRow) dim(100)

/free
dbxGetCusLst(Position : ListType : CountReq :
              CountRet : outputType : custRows);
for i = 1 to CountRet;
  CustList(i).Name = custRows(i).cust_name;
  CustList(i).Number = custRows(i).cust_no;
endfor;
/end-free
p FindCustomers e

//*****
p GetCustNumber b export
//*****
d GetCustNumber pi

```

```

d Name 64 const
d Number 9B 0
d Generate 1 const options(*nopass)
//
d NameV s 64 varying
d outSqlState s 5a
d outSqlMsg s 256a
d newCustRow ds likeds(CustomerRow)
d custNo s like(customerRow.cust_no)

/free
NameV = %trim(Name);
customerRow = dbxCusByNam(Name : outSqlState : outSqlMsg);
if %subst(outSqlState:1:2) <> '00';
  if %parms > 2 and Generate = 'Y';
    newCustRow.cust_name = Name;
    dbxInsCus(newCustRow : outSqlState : outSqlMsg);
    custNo = newCustRow.cust_no;
  else;
    custNo = -1;
  endif;
else;
  custNo = customerRow.cust_no;
endif;
Number = CUSTNO;
/end-free
p GetCustNumber e

//*****
p GetCustName b export
//*****
d GetCustName pi
d Number 9B 0 const
d Name 64

d outSqlState s 5a
d outSqlMsg s 256a

/free
Name = *blank;
customerRow = dbxCusByNum(Number :outSqlState:outSqlMsg);
if %subst(outSqlState:1:2) = '00';
  Name = customerRow.cust_name;
endif;
/end-free
p GetCustName e

```

Appendix C: Resources

These Web sites provide useful references to supplement the information contained in this document:

- DB2 UDB for iSeries home page
ibm.com/eserver/series/db2
- DB2 UDB Modernization Roadmaps
 - Modernizing DB2 UDB definitions and usage
www.developer.ibm.com/vic/hardware/myportal/develop/roadmap?roadMapId=appiniti
(**Note:** Use this roadmap to learn how to reverse engineer database objects and replace all DDS-created physical files and logical files with SQL-DDL created tables, views, and indexes.)
 - Modernizing data access with SQL
www.developer.ibm.com/vic/hardware/myportal/develop/roadmap?roadMapId=appinitj
(**Note:** Use this roadmap to learn how to update applications so that native I/O database access methods are replaced with SQL interfaces.)
 - Optimizing SQL performance
developer.ibm.com/vic/hardware/myportal/develop/roadmap?roadMapId=appinith
(**Note:** This roadmap teaches you how to optimize database and SQL statements so that query response time, network traffic, disk I/O, and CPU time are all minimized when executing your queries.)
- Online Publications for iSeries
 - ibm.com/eserver/series/infocenter
 - ibm.com/eserver/series/db2/books.html
- Education Resources (classroom and online)
 - ibm.com/eserver/series/db2/db2educ_m.htm
 - ibm.com/servers/enable/site/education/ibo/view.html?wp#db2
 - ibm.com/servers/enable/site/education/ibo/view.html?oc#db2
- Online Newsgroups and Forums
 - USENET: comp.sys.ibm.as400.misc, comp.databases.ibm-db2
 - AS/400 Network SQL & DB2 UDB Forum
iseriesnetwork.com/isnforums
- Recommended IBM Redbooks™
 - redbooks.ibm.com/
 - Modernizing IBM eServer iSeries Application Data Access - A Roadmap Cornerstone, (SG24-6393)
Stored Procedures, Triggers, and User-Defined Functions on DB2 UDB for iSeries (SG24-6503)
 - Advanced Database Functions and Administration on DB2 UDB for iSeries (SG24-4249-03)
 - DB2 UDB for AS/400 Object Relational Support (SG24-5409)
 - Modernizing and improving the maintainability of RPG applications using x-Analysis Version 5.6 (REDP-4046-00)

- Other Publications
 - White paper: Modernizing Flight 400
ibm.com/servers/enable/site/education/abstracts/40d2_abs.html
 - SQL/400 Developer's Guide by Paul Conte & Mike Cravitz
29th Street Press (ISBN: 1-882419-70-7)
 - iSeries & AS/400 SQL at Work, by Howard Arner
www.sqlthing.com/books.htm
 - IBM eServer i5 Information Center
publib.boulder.ibm.com/infocenter/iseriess/v5r3/ic2924/index.htm
 - IBM eServer p5 Information Center
publib.boulder.ibm.com/infocenter/pseries/index.jsp
 - IBM Publications Center
elink.ibm.com/public/applications/publications/cgibin/pbi.cgi?CTY=US

Appendix D: About the author

Gene Cobb is a DB2 UDB technology specialist in ISV Strategy and Enablement. He has worked in IBM midrange systems since 1988, with 10 years in the IBM Client Technology Center (CTC) in Rochester, Minnesota. While in the CTC, he assisted customers with application design and development using RPG, DB2 UDB for iSeries, CallPath/400, and IBM Lotus® Domino®. His current responsibilities include providing consulting services to iSeries developers, with special emphasis in application and database modernization.

Trademarks and special notices

© IBM Corporation 1994-2005. All rights reserved.

References in this document to IBM products or services do not imply that IBM intends to make them available in every country.

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM	eServer	i5/OS	DB2
ibm.com	iSeries	WebSphere	DB2 Universal Database
the IBM logo	pSeries	AIX	Virtualization Engine
Redbooks	Tivoli	AIX 5L	Enterprise Storage Server
PartnerWorld	HACMP	Lotus	Domino

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Intel Inside (logos), MMX, and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Red Hat, the Red Hat "Shadow Man" logo, and all Red Hat-based trademarks and logos are trademarks or registered trademarks of Red Hat, Inc., in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

SET and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product or service names may be trademarks or service marks of others.

Information is provided "AS IS" without warranty of any kind.

All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer.

Information concerning non-IBM products was obtained from a supplier of these products, published announcement material, or other publicly available sources and does not constitute an endorsement of such products by IBM. Sources for non-IBM list prices and performance numbers are taken from publicly available information, including vendor announcements and vendor worldwide homepages. IBM has not tested these products and cannot confirm the accuracy of performance, capability, or any other claims related to non-IBM products. Questions on the capability of non-IBM products should be addressed to the supplier of those products.

All statements regarding IBM future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. Contact your local IBM office or IBM authorized reseller for the full text of the specific Statement of Direction.

Some information addresses anticipated future capabilities. Such information is not intended as a definitive statement of a commitment to specific levels of performance, function or delivery schedules with respect to any future products. Such commitments are only made in IBM product announcements. The information is presented here to communicate IBM's current investment and development activities as a good faith effort to help with our customers' future planning.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Photographs shown are of engineering prototypes. Changes may be incorporated in production models.