Moving from OPNQRYF to SQL

Transforming OPNQRYF programs to an SQL access model

.

Gene Cobb
ISV Business Strategy and Enablement
March 2008

Table of contents

Abstract

This white paper discusses the major advantages that SQL has in comparison to the IBM i5/OS Open Query File (OPNQRYF) command. It also provides a methodology for converting OPNQRYF applications to an SQL-based model, as well as some key points to consider during the conversion process.

The OPNQRYF command was introduced in 1987, near the end of life for the IBM System/38 platform. This command received rave reviews from the programming community because it provided an easy interface to dynamically access records in a database file. Consequently, thousands of programs that use OPNQRYF were developed and are still in production today. A few years after OPNQRYF was made available, IBM introduced SQL to the IBM AS/400 family of systems, giving programmers an alternative database-access tool set — one that also provides great flexibility, but one that actually surpasses OPNQRYF in many important areas.

Introduction

Before SQL was available on the IBM® AS/400® platform (now the IBM System i™ platform), using the IBM OS/400® operating-system Open Query File (OPNQRYF) command (now an IBM i5/OS® operating-system command) was a very popular method of accessing information that was stored in a database file. As SQL does on System i today, OPNQRYF provided midrange application developers with a great deal of flexibility and ease of use when specifying selection, joining, ordering and grouping. Rather than hardcoding various access methods within the tiers of multiple-case or if-then-else statements, programmers simply constructed character strings dynamically with these criteria and passed those strings as OPNQRYF parameters. The database engine processed that request accordingly to return a result set. The dynamic nature of this command eliminated many lines of code and made applications easier to understand and maintain.

OPNQRYF is still widely used on the System i platform today. Thousands of applications take advantage of its ability to dynamically access information that is stored in an IBM DB2® for i5/OS® database. There are many ways that you can use OPNQRYF in application programs. Some of the most popular OPNQRYF uses are listed here:

- Dynamic record selection
- Dynamic ordering without using data description specification (DDS)
- Grouping – summarizing data
- Specifying key fields from different files
- Dynamic joining without using DDS
- Unique-key processing
- Defining fields that are derived from existing field definitions
- Final total-only processing
- Random access of a result set
- Mapping virtual fields

Although OPNQRYF does an excellent job at performing these types of tasks and IBM still fully supports it, you can use a pure SQL-based implementation to enjoy additional benefits while maintaining the application flexibility provided by OPNQRYF. In fact, there are many reasons to consider converting your applications to use pure SQL access. This white paper examines some of these reasons in detail; it also discusses some approaches to performing this data-access conversion. There are many examples and other considerations to help you understand and carry out this conversion process successfully.

Reasons to consider conversion

Although OPNQRYF is quite adept at handling the tasks just listed, there are many compelling reasons to consider converting your programs to a pure SQL-statement model, including the following reasons:

- SQL is the industry standard.
- SQL is the strategic interface for DB2 for i5/OS.
- The SQL query engine (SQE) provides superior performance.
- The SQL interfaces offer advanced functions.
- DB2 performance tools are tailored for SQL interfaces.
- SQL offers simpler application development and maintenance.
- Using SQL reduces the number of programs and lines of code.

Each of these advantages is examined in detail in the following sections of this white paper.

SQL is the industry standard

Several years after OPNQRYF was made available on IBM System/38™ severs, IBM shipped SQL as an application-development tool for creating and accessing database objects on AS/400. Since that time, SQL has become the widely adopted industry standard for all relational-database platforms. Although functional and powerful, OPNQRYF is a proprietary, non-SQL interface and is supported only on IBM midrange systems. In this evolving world of system openness, this is important: the days of relying on your green-screen application to provide the sole access to your data are nearing the end. Almost every non-5250 client application that accesses information from a database on the System i platform will use an SQL interface.

SQL is strategic

The integrated database engine that is provided with i5/OS processes both OPNQRYF and SQL requests. Consequently, both methods rely on the engine to optimize the request (that is, to find the optimal access plan) and to process the result. Even though OPNQRYF and SQL requests are processed similarly, there are many key differences in the two techniques. First, OPNQRYF is not a strategic implementation for IBM. It uses the older classic query engine (CQE) to optimize query requests. Because it is not strategic, IBM has announced no plans to enhance the new SQE to support OPNQRYF requests or to make any future investment in enhancing the capabilities of non-SQL interfaces. These non-SQL interfaces include OPNQRYF, the IBM Query/400 product and the IBM QQQQry application programming interface (API).

IBM introduced SQE in i5/OS V5R2 and offers algorithms and features that give it distinct advantages as compared to its predecessor, CQE. When a request is submitted to the database engine, the Query Dispatcher component of the engine first analyzes that request. The Query Dispatcher determines which engine will optimize and process the query. Only pure SQL queries are considered for SQE. This means that requests that are initiated from any of the aforementioned non-SQL interfaces are sent down to the CQE and, therefore, cannot take advantage of all the new features that SQE offers.

SQE features

There are multiple advantages when the SQE processes a query. Some of the major features that are only available with SQE are as follows:

- **SQE plan cache:** This internal matrix-like repository holds access plans for queries that the SQE optimizes and allows plans for identical and similar statements to be shared across jobs. Reusing existing access plans means that the optimizer does not have to create new plans, saving time and resources.
- **Automated collection of column statistics:** These statistics provide a valuable source of information to the optimizer when evaluating the cost of each available access plan. A better understanding of the data and more accurate estimates of the number of rows that are to be processed results in the selection of a better access plan and a more efficient query.
- **Autonomic Indexing:** This is the ability of the optimizer to create temporarily maintained indexes that both the current query and future queries can use, system-wide.

Superior performance of SQE

SQE also introduced new data-access primitives to process the data. The improved results are most evident for complex queries that require grouping, ordering and joining. As a result, SQL statements that use SQE generally perform better than similar requests that CQE processes. Because CQE processes all OPNQRYF requests, most SQL-access requests to the database should outperform an equivalent OPNQRYF request.

Quantifying how much better an application performs after making modifications is a tricky venture. Many factors influence performance; thus, it is better to conduct your own benchmarks: perform the conversion on a sample application and record the resulting performance metrics.

However, it is important to have an idea of the degree of improvement that is possible with this conversion. Some of the more widely used OPNQRYF applications are listed in the Introduction section. Part of the exercise of writing this white paper included converting examples of some of these types of applications from OPNQRYF to SQL and then running both versions to measure the performance differences. The conversion examples and performance results are documented in Appendix C: Conversion examples and performance measurements.

But again, it must be stressed that each environment is different; thus, the standard disclaimer must be made, “Your performance can vary.”

To learn more about SQE, refer to *Preparing for and Tuning the SQL Query Engine on DB2 for i5/OS*, which you can find at the IBM Redbooks® Web site: www.redbooks.ibm.com/abstracts/sg246598.html?Open.

Advanced functions of the SQL interface

SQL on DB2 for i5/OS is rich with features that are simply not available with OPNQRYF. The following list highlights some of the more prominent features that can only be used with pure SQL data-access methods: Here are some of the more prominent features to consider:

- **User-defined functions (UDFs):** OPNQRYF provides a base set of built-in functions to use in query selection and grouping and for defining a derived field. However, OPNQRYF doe not always have the required functions for you to implement complex business calculations or processes, such as computing shipping costs or a customer's credit risk. If you need to extend OPNQRYF by writing custom functions, there is no way to do so. With SQL, it is easy to create UDFs and to use them just as you use the SQL built-in functions.
- **User-defined table functions (UDTFs):** Another valuable SQL feature is the ability to search for and retrieve data in nonrelational system objects (such as data areas and data queues, and even information that is stored in the integrated file system [IFS]), and then return that information to any SQL interface. You can do this by creating UDTFs.
- **Materialized query tables (MQTs):** This is a summary table that contains the results of a previously run query, along with the query's definition. It provides a way to improve the response time of complex SQL queries. What sets an MQT apart from a regular temporary summary table is that the SQE optimizer is aware of its relationship to the query and base tables that are used to create and populate it. This means that the optimizer considers using the MQT in the access plan of subsequent similar queries. Because the MQT is already created and populated, this can improve performance significantly for complex queries. For details on MQTs, see the white paper *Creating and using materialized query tables (MQT) in IBM DB2 for i5/OS* (**ibm.com**/servers/enable/site/education/abstracts/438a_abs.html).
- **SQL views:** An SQL view is a *virtual table*. It provides another way to view data in one or more tables and is created based on an SQL SELECT statement. With views, you can transform the data and move business logic from the application layer down to the database. You can define selection, grouping and complex logic through CASE statements. This means that, in an SQL view, you can define the selection, joining and ordering specifications for an OPNQRYF command. Because those specifications exist in the view definition, you need not specify them again in SQL. Instead, the SQL refers to the view. In fact, a technique that many System i programmers employ is to specify an SQL view in the OPNQRYF FILE parameter.
- **Subqueries:** By definition, a subquery is a query that is embedded within the WHERE or HAVING clause of another SQL (parent) statement. The parent statement then uses the rows that are returned to further restrict the retrieved rows. A subquery can include selection criteria of its own, and these criteria can also include other subqueries. This means that an SQL statement can contain a hierarchy of subqueries. This is a powerful SQL searching mechanism that cannot be accomplished with OPNQRYF.
- **Common table expressions (CTEs) and recursive SQL:** CTEs can be thought of as temporary views that only exist when running an SQL statement. After the CTE is defined, you can reference it multiple times in the same query. You can use this to reduce the complexity of the query, making it easier to comprehend and maintain.

Among the i5/OS V5R4 enhancements for DB2 for i5/OS, is the ability for a CTE to reference itself. This feature provides the mechanism for recursive SQL, which is especially useful when querying data that is hierarchical in nature (such as a bill of materials, organizational charts and airline flight schedules). (**Note:** For details on recursive CTEs, see the article *i5/OS V5R4 SQL Packs a Punch*, (**ibm.com**/servers/eserver/iseries/db2/pdf/rcte_olap.pdf).

- **Complex joining:** To help solve your complex business requirements, SQL provides various ways of joining data in tables together by using INNER, OUTER and EXCEPTION joins.
- **Fullselect:** An SQL *fullselect* is the term for generating an SQL result set by combining multiple SELECT statements that use the UNION, INTERSECT and EXCEPT operators. This is yet another feature that helps you solve more complex business requirements and is beyond the capabilities of OPNQRYF.
- **Encryption:** The importance of data security in today's business environment cannot be overstated. Hackers, phishers and others with malicious intentions constantly and incessantly attempt to access data to which they have no rights. If you store sensitive information in your database, it is your obligation (and often a lawful requirement) to protect that information from these threats. Object security, firewalls and other security measures are all valuable tools to thwart unauthorized access and to secure the data. Data encryption provides another line of defense. A hacker who is somehow able to penetrate your security implementations, will only find encrypted information.

  Note: For more information on database encryption, refer to the white paper *Protecting i5/OS data with encryption* (**ibm.com**/servers/enable/site/education/ibo/record.html?efbe).

DB2 performance tools tailored for SQL interfaces

The primary method of gathering feedback from the database is through the use of the SQL performance monitor. Collected monitor data contains every SQL request that was submitted during the collection period, and includes valuable information, such as: what statement was requested, which access plan was used, how much time the database spent optimizing that request, and how much time it took to complete the request. Although the SQL performance monitors do collect information that relates to non-SQL requests, such as OPNQRYF, some vital information is not collected from those interfaces.

- **No 1000 record – cannot see actual statement:** Although the database performance monitor captures feedback for OPNQRYF requests (such as the implemented access method and index advisories), the actual statement (contained in the 1000 record) is not contained. This can make tuning challenging, especially if the OPNQRYF request runs long and causes system problems. Isolating the specific statement can be difficult if you do not know what the statement is.
- **No Visual Explain support:** Visual Explain is one of the most powerful features that you can implement by using collected monitor data. This feature renders a graphical representation of the query, and shows all of the database objects and access methods that the optimizer chooses. This provides the ability to drill down on identified problem queries to quickly diagnose and address problem areas. However, the information needed to support Visual Explain is not collected for non-SQL interfaces, such as OPNQRYF.

Application readability and understanding

SQL is taught as a core-curriculum subject in most colleges and universities today. Upon graduation, most programmers have mastered the necessary SQL basics and are ready to be productive. Although OPNQRYF does provide many of the same functions as SQL, it can be rather cryptic and hard to understand, especially for a programmer who is new to the System i platform. If you use all the apostrophes required by OPNQRYF, the various parameters, the cryptic functions and shared-open considerations, you can end up with an application that is more difficult to comprehend than an equivalent one that uses SQL. Consider the following example:

```
OPNQRYF FILE(CUST_MAST) QRYSLT('%XLATE(LASTNAME QSYSTRNTBL) *CT "FRED"
   *AND STATE *EQ %VALUES("CA" "NY" "TX") ')
```

In this fairly simple example, the query-selection parameter specifies a case-insensitive search of the column LASTNAME and STATE in table CUST_MAST. Only those rows that contain the value FRED in the LASTNAME column who live in the states of CA, NY or TX are added to the result set. Now, compare that syntax with this equivalent SQL example:

```
SELECT * FROM CUST_MAST WHERE UPPER(LASTNAME) LIKE '%FRED%'
   AND STATE IN ('CA', "NY', 'TX')
```

Which method is easier to understand? Of course this is a subjective question. A seasoned System i developer might prefer OPNQRYF because that has been the chosen implementation over the years and has served its purpose well. However, those who are less experienced with the platform, the control-language (CL) language or the OPNQRYF command itself will most likely find the SQL statement easier to comprehend. Its syntax is more English-like and, as mentioned, most programmers have had some exposure to SQL.

Reduction in number of programs and lines of code

Proper implementation of the SQL programming model usually means a more condensed application. The OVRDBF command is no longer required to share the ODP, and the OPNQRYF command can be removed, as can the other CL commands (DLTOVR and CLOF) that are used to support the OPNQRYF method. In many cases, you can remove a CL program entirely, letting the SQL in the RPG program do all the work. It also provides overall performance improvements because the additional overhead of those CL commands and programs are eliminated.

Furthermore, using SQL promotes more of a data-centric programming model, which means moving as much of the business logic down to the database level as possible. You accomplish this by using database features such as triggers, stored procedures, functions and referential integrity (primary and foreign-key constraints). A data-centric model is compelling for multiple reasons:

- The logic is stored in one place; thus, eliminating duplicate code. This frees you from the unnecessary maintenance that is associated with multiple versions of the same business logic.
- The business logic is enforced for all database interfaces and, thus, you cannot circumvent it. Database integrity is ensured, regardless of the client who accesses the data.
- It reduces the lines of code. This is important from a programming-management point of view, because, in many organizations, lines of code are the unit of measure for determining the cost of

program maintenance. Consequently, fewer lines of code to maintain can mean fewer programmers are needed to maintain the code. This frees up more people for new development.

- It extends the life of an existing application. Many green-screen applications lack data integrity. Fixing these issues at a database level can extend an application's usefulness (especially green-screen applications) when there are no plans to change the presentation layer.

Anatomy of OPNQRYF

To understand how to successfully carry out this transformation, you first must understand the composition of both techniques. In this section, the focus is on the typical anatomy of an application that uses OPNQRYF. Although there are multiple ways to implement the OPNQRYF technique, in its simplest form, it requires three objects (see Figure 1):

- A database file that contains the data to be processed; this can be either a physical or logical file
- A CL program to set up the environment and issue the OPNQRYF command
- A high-level language (HLL) program to read and process the rows in the query file (result set)

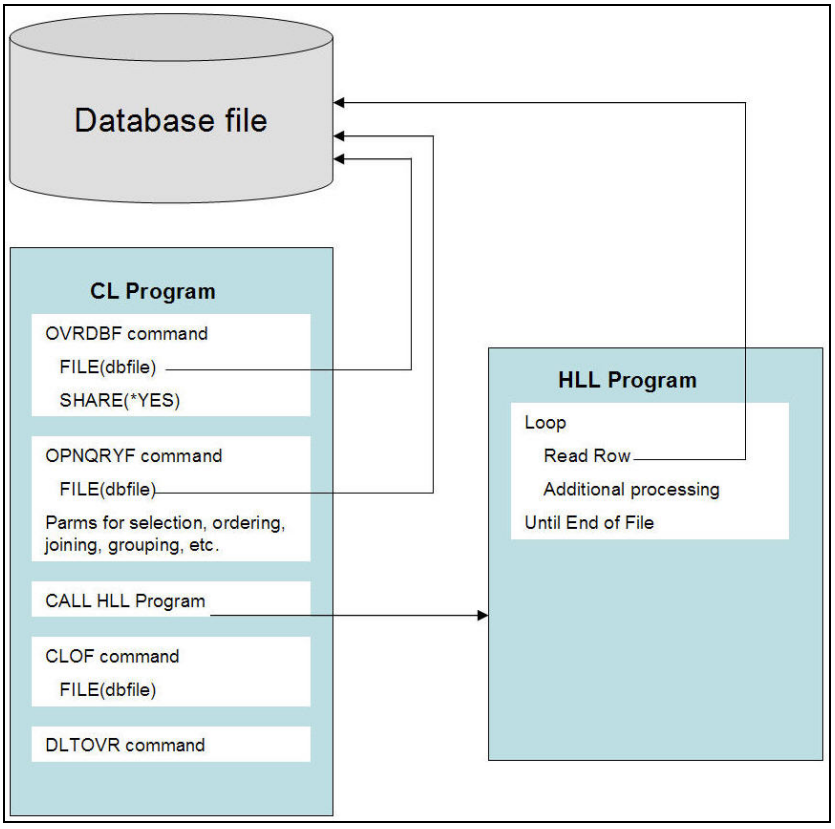

Figure 1. The OPNQRYF technique requires three objects

CL program

The CL program typically contains five commands to perform the necessary processing. These commands and the tasks that they perform are explained as follows:

- **OVRDBF (Override database file):** This is responsible for the following tasks:
  - Specifies the file to override
  - Specifies that the open data path (ODP) of the file (that is opened by OPNQRYF command) is to be shared with other programs in the same routing step
  - Specifies the scope of the override (job or activation group)
- **OPNQRYF (Open query file):** This contains the specification of the selection, joining, sorting and grouping of the data to be processed. When run, the command opens a file to a set of database records that satisfies this specification.
- **CALL:** This invokes the HLL program that reads those records and performs further processing.
- **CLOF (Close file):** This closes the file created by the OPNQRYF command.
- **DLTOVR (Delete override):** This deletes the override specified in the OVRDBF command.

Putting it all together, Figure 2 shows a simple example of an OPNQRYF CL program:

```
OVRDBF     FILE(ORD_DTL) SHARE(*YES) OVRSCOPE(*JOB)
OPNQRYF    FILE((ORD_DTL)) QRYSLT('YEAR *EQ 1996')OPTION(*INP)
CALL       PGM(PROC_ROWS)
CLOF       OPNID(ORD_DTL)
DLTOVR     FILE(ORD_DTL)LVL(*JOB)
```

Figure 2. Example of an OPNQRYF CL program

This example program creates a shared ODP of all rows in the ORD_DTL table that have the value of 1996 for the column YEAR. The called program PROC_ROWS can use that shared ODP.

HLL program

The HLL program is typically written in RPG or COBOL and is responsible for the following tasks:

- IT opens the same file (through the shared ODP) as the one previously processed by the OPNQRYF command.
- It reads the rows in this file. Because SQL was not used to generate the result set, you perform the access by using record-level access (RLA) methods (also known as *native I/O methods*).
- It performs additional processing that is based on the data in those rows (for example, using the information in a row to add a record to a subfile or to add a line to a print file).

A snippet of a simple HLL program example is shown in Figure 3. It uses a looping construct to read each row in the result set and calls a subroutine to load the contents of each row into a subfile. At the end of the loop, the subfile is presented on the screen. Figure 3 shows a typical interactive usage of OPNQRYF.

```
Ford_dtl     if   e             disk
d rowsFetched     s              9b 0

c                   eval      rowsFetched = 0
c                   exsr      inzSubFile

c     *start        setll     ord_dtl
c                   dow       not %eof
c                   read      ord_dtl
c                   if        %eof
c                   leave
c                   endif

c                   eval      rowsFetched = rowsFetched + 1
c                   exsr      loadSubFile
c                   enddo
c                   exsr      dspSubFile

c                   eval      *inlr=*on
c                   return
```

Figure 3. Example of HLL program that processes results of OPNQRYF

Anatomy of an SQL-based model

As with OPNQRYF, there are various ways to implement the SQL-based model, but in its simplest form, it requires two objects:

- A database file to hold data to be processed (table [physical file] or view [nonkeyed logical file])
- A HLL program that creates the ODP to the result set and, subsequently, reads and processes the rows in the result set

The difference between the SQL model and OPNQRYF is that no CL program is needed to issue file overrides (to share the ODP) and the OPNQRYF command (see Figure 4). Thus, no cleanup CL commands are needed. The HLL program performs all work through embedded SQL. In fact, there is no need to share the ODP because it is created in the same routing step (program) that reads the table's rows.

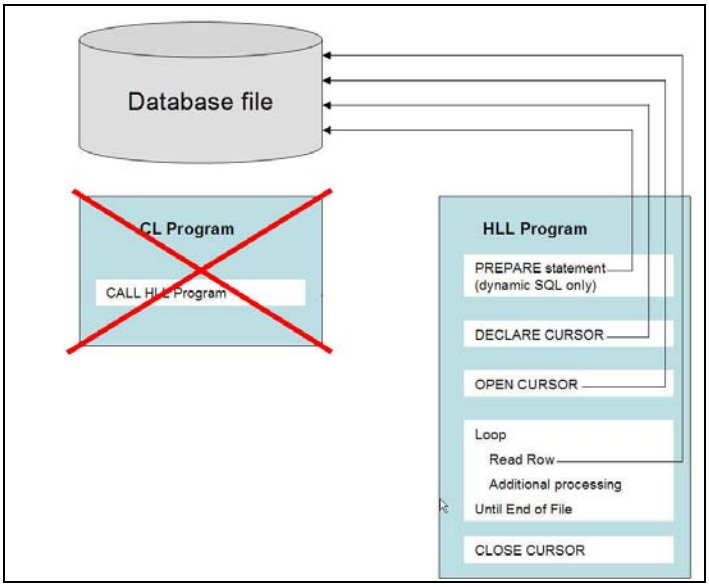

Figure 4. The SQL-based model requires two objects

HLL program

As is the case with the OPNQRYF-based model, the HLL program is typically written in RPG or COBOL and is generally responsible for the following tasks:

- It issues the SQL DECLARE statement to define the cursor. This statement contains the actual SQL that generates the result set. This step also validates the user's authorization to the table.
- It issues the SQL OPEN statement to open the cursor that is defined in the previous step.
- Within a looping construct, it issues the SQL FETCH statement to read the rows in the result set.
- It performs additional processing, based on the data in those rows.

The RPG program example seen earlier in this white paper has been revised in Figure 5 (see the code lines shown in bold) to use embedded static SQL to perform the same function. The differences between static and dynamic embedded SQL are discussed in detail in a later section.

```
F*ord_dtl     if   e    ** notice this line has been commented out **
d ord_dtl_Data  e ds                  extname(ord_dtl)
d rowsFetched     s              9b 0


c                   eval      rowsFetched = 0
c                   exsr      inzSubFile

**** Declare the cursor
C/EXEC SQL
C+   DECLARE c1 CURSOR FOR
c+   SELECT *
c+   FROM ord_dtl
c+   WHERE year = 1996
C/END-EXEC

**** Open the cursor
C/EXEC SQL
C+   OPEN c1
C/END-EXEC

c                   dow       SQLSTAT <> '02000'
c                   eval      rowsFetched = rowsFetched + 1
 **** Fetch matching record
C/EXEC SQL
C+   FETCH C1 INTO :ord_dtl_Data
C/END-EXEC
c                   if       SQLSTAT = '02000'                            0
c                   leave
c                   endif

c                   exsr      loadSubFile
c                   enddo
c                   exsr      dspSubFile


c                   eval      *inlr=*on
c                   return
```

Figure 5. RPG program example, modified to use embedded static SQL

Conversion methodology

The example shown in Figure 5 is a fairly simple one. Most production applications are more complex than this and require a significant amount of effort. Unfortunately, the System i platform does not provide a comprehensive end-to-end tool to perform such a conversion. As a result, you must perform the majority of the conversion effort manually. This section describes how to plan for, and approach, this conversion process. The conversion methodology contains the following tasks:

1. Convert the CL commands to equivalent SQL statements
2. Create SQL views with the converted SQL statements
3. Modify the HLL language to include embedded SQL statements
4. Modify or eliminate the CL program

For this section, the OPNQRYF implementation shown in Figure 6 is converted to SQL (see Figure 7):

```
OVRDBF     FILE(ITEM_JOIN3) TOFILE(ITEM_FACT) +
             OVRSCOPE(*JOB) SHARE(*YES)
OPNQRYF    FILE((ITEM_FACT) (CUST_DIM) (TIME_DIM)) +
           FORMAT(ITEM_JOIN3)
           QRYSLT('TIME_DIM/YEAR *EQ 1997') +
           JFLD((ITEM_FACT/CUSTKEY CUST_DIM/CUSTKEY) +
                (ITEM_FACT/SHIPDATE TIME_DIM/DATEKEY)) +
           GRPFLD(CUST_DIM/CUSTOMER) +
           MAPFLD((TOTITEMS '%SUM(QUANTITY)') +
           (TOTREV '%SUM(REVENUE)') +
           (TRANSCOUNT '%COUNT')) +
           OPNSCOPE(*JOB)
CALL       PGM(PROC_ROWS)
CLOF       OPNID(ITEM_FACT)
DLTOVR     FILE(ITEM_JOIN3) LVL(*JOB)
```

Figure 6. An OPNQRYF implementation

```
Fitem_join3if   e           k disk
d rowsFetched     s              9b 0

 /FREE
  rowsFetched = 0;
  exsr inzSubFile;

  dow not %eof;
    read item_join3;
    if %eof;
      leave;
    endif;
    rowsFetched = rowsFetched + 1;
    exsr loadSubFile;
  enddo;
  exsr dspSubFile;

  *inlr=*on;
  return;
```

Figure 7. The same OPNQRYF implementation converted to SQL

Step 1: Convert the CL commands to equivalent SQL statements

As mentioned, the OPNQRYF implementation typically uses five CL commands. You can convert each of these to an SQL equivalent statement in the HLL program, or you can eliminate then entirely. Table 1. summarizes each of these commands and lists its corresponding SQL replacement statement.

CL command	SQL replacement
OVRDBF	No longer needed because embedded SQL performs the work
OPNQRYF	PREPARE statement (if dynamic SQL) in the HLL program DECLARE CURSOR statement in the HLL program OPEN CURSOR statement in the HLL program
CALL	No change in CL program is necessary
CLOF	CLOSE CURSOR statement in the HLL program
DLTOVR	No longer needed because no override is in effect

Table 1. Summary of commands to replace between OPNQRYF and SQL

OVRDBF command

This command sets up a shared ODP that the subsequently called HLL program uses to read and process the rows in the file. With an SQL implementation, there is no need to specify a shared ODP environment because the ODP is actually created in the same program as the one that reads and processes the rows. In addition, there is no concept of a shared ODP in SQL. For these reasons, no equivalent SQL statement is required and this CL command does not need to be converted.

OPNQRYF command

This command carries out most of the work of the OPNQRYF implementation. It is responsible for specifying the selection, joining, sorting and grouping of the data to be processed, as well as opening the file that meets the specified criteria. As such, perhaps the most challenging part of the conversion exercise is to transform this command into an SQL statement that generates identical results while maintaining or enhancing performance. To do this successfully, you must understand a couple of key SQL concepts so that you can select the optimal implementation for your application environment.

Convert OPNQRYF parameters to corresponding SQL clauses

Most of the OPNQRY parameters have an equivalent SQL clause or can be implemented using other techniques. To help guide you in this process, refer to “Appendix A: OPNQRYF command parameters and SQL equivalents.” This appendix lists each OPNQRYF parameter and explains how you might implement it in your application when moving to an SQL model.

Build SQL statement

After you convert the OPNQRYF commands to SQL equivalents by using the information in Appendix A, you can build the appropriate SQL statements. These statements are eventually used to create the SQL view and the view that is embedded into the RPG program. Table 2. shows the OPNQRYF statement for this example, as well as the converted SQL statement. The dashed lines separate the segments of code that have been implemented to show the mapping of OPNQRYF parameters to SQL clauses.

OPNQRYF command	Equivalent SQL statement
OPNQRYF **FORMAT(ITEM_JOIN3)** **MAPFLD((CUSTOMER 'CUST_DIM/CUSTOMER')** **(TOTITEMS '%SUM(QUANTITY)')** **(TOTREV '%SUM(REVENUE)')** **(TRANSCOUNT '%COUNT'))** - - - - - - - - - - - - - - - - - - **FILE((ITEM_FACT)** **(CUST_DIM)** **(TIME_DIM))** - - - - - - - - - - - - - - - - - - **QRYSLT('TIME_DIM/YEAR *EQ 1997')** - - - - - - - - - - - - - - - - - - **JFLD((ITEM_FACT/CUSTKEY CUST_DIM/CUSTKEY)** **(ITEM_FACT/SHIPDATE TIME_DIM/DATEKEY))** - - - - - - - - - - - - - - - - - - **GRPFLD(CUST_DIM/CUSTOMER)** - - - - - - - - - - - - - - - - - - **KEYFLD((CUST_DIM/CUSTOMER *ASCEND))**	SELECT C.CUSTOMER, SUM(I.QUANTITY) AS TOTITEMS, SUM(I.REVENUE_WO_TAX) AS TOTREV, COUNT(*) AS TRANSCOUNT - - - - - - - - - - - - - - - - - - FROM ITEM_FACT I, CUST_DIM C, TIME_DIM T - - - - - - - - - - - - - - - - - - WHERE T.YEAR = 1997 - - - - - - - - - - - - - - - - - - AND I.SHIPDATE = T.DATEKEY AND I.CUSTKEY = C.CUSTKEY - - - - - - - - - - - - - - - - - - GROUP BY C.CUSTOMER - - - - - - - - - - - - - - - - - - ORDER BY C.CUSTOMER ASC

Table 2. The OPNQRYF example statement and the equivalent converted SQL statement

CALL command

You do not typically need to change the command to call the HLL program. This is the case unless, for some reason, you decide to alter the HLL program's parameter list. However, because it is possible to convert all of the necessary OPNQRYF-related CL commands into SQL statements in the HLL program, you might consider analyzing whether the conversion completely eliminates the need for the CL program. Eliminating a dynamic program call will result in less overhead, better application performance and a reduction in the application's complexity.

CLOF command

This command is issued to close the file (result set) created by the OPNQRYF command. If an SQL cursor has been opened in the converted HLL program, the SQL equivalent is the CLOSE CURSOR statement in the same HLL program.

DLTOVR command

Because no override needs to be in effect to share an open data path, this command is no longer needed; you can eliminate it.

Step 2: Create SQL views

As mentioned, SQL views are virtual tables that are based on an SQL Select statement. They exist as permanent objects on the System i platform and are implemented as nonkeyed logical files. As such, they have no access-path maintenance and, consequently, none of its associated overhead. They are very useful and (in the opinion of the author) an often underused tool on the System i platform.

Having said this, implementing SQL views in this conversion process is not a required step. But there are advantages when you choose to use them. Here are some benefits that you should consider:

- You can move complex business logic (for example, join syntax, grouping, CTEs and CASE statements) from the HLL program down to the SQL-view definition. When there, the business logic is enforced for all database interfaces that access the view (including other client applications, such as ODBC, JDBC and the new DB2 Web Query tool).
- Moving this business logic down to the database effectively masks the complexity of the underlying database and makes your applications easier to understand and maintain.
- It is possible to implement row- and column-level security by using views. By restricting access to the underlying tables and forcing all users to access data through SQL views, you can lock down your database, yet still provide a flexible solution that is easy to maintain.
- Ability to test the view without having to write a single line of program code.

You can use an SQL view to externally describe a data structure in a HLL program. As you will see in Step 3, you can then use this data structure as the host variable that holds the contents of an SQL FETCH or SELECT INTO statement.

Using the converted SQL statement that was formulated in Step 1, you can use the SQL statement shown in Figure 8 to create the view that is necessary for the conversion:

```
CREATE VIEW CUSTSUM97 (
  CUSTOMER ,
  TOTITEMS ,
  TOTREV ,
  TRANSCOUNT )
AS (
  SELECT
    C.CUSTOMER,
    SUM(I.QUANTITY) AS TOTITEMS,
    SUM(I.REVENUE_WO_TAX) AS TOTREV,
    COUNT(*) AS TRANSCOUNT
  FROM
    ITEM_FACT I,
    CUST_DIM C,
    TIME_DIM T
  WHERE
    I.SHIPDATE = T.DATEKEY
      AND I.CUSTKEY = C.CUSTKEY
      AND T.YEAR = 1997
GROUP BY C.CUSTOMER)
```

Figure 8. SQL statement that creates the view that is required for conversion

Step 3: Modify HLL program to include embedded SQL

Now, you must modify the HLL program, by embedding SQL statements and views that were formulated in previous steps and other tasks that support the embedded SQL statements.

Step 3.1: Remove the File Specification

Because SQL access methods will be used to read the database, you no longer need to declare the accessed file in the RPG program's File Specifications (F Specs) — unless the file is accessed by RLA methods. Therefore, the task in this HLL modification step is to comment out or remove this line from the RPG program. This is shown in bold in the HLL code example in Figure 9.

```
F* item_join3if   e           k disk
d rowsFetched     s              9b 0

 /FREE
  rowsFetched = 0;
  exsr inzSubFile;

  dow not %eof;
    read item_join3;
    if %eof;
      leave;
    endif;
    rowsFetched = rowsFetched + 1;
    exsr loadSubFile;
  enddo;
  exsr dspSubFile;

  *inlr=*on;
  return;
```

Figure 9. Example of commenting out the file's F Spec

Step 3.2: Add an externally described data structure

To retrieve the results of an SQL Select statement, you need to be able to access the result set. In a HLL program, the only way to get the contents of an SQL result set is to receive them into host variables. There are two ways to declare these variables: by using either externally described data structures or program described variables and data structures. Tables and views have formats to describe a result set's column or field attributes. Consequently, you can use them for externally described data structures. The ILE RPG compiler uses the external name to locate and extract the specified table's (or view's) format. Then, you can use this structure as the host variable to store the result set's contents by using a FETCH or SELECT INTO statement.

Alternatively, you can define program-described data structures, where each result-set column is explicitly defined in the RPG program's Data Specifications (D Specs). Two primary disadvantages with the program-described data-structure method involve program maintenance:

1. You must manually type in each data-structure field, particularly if the result set has many columns.
2. When the format of the table or view changes (for example, a new column is added), you must manually modify and recompile the program to pick up those changes. With externally described structures (in the example shown in bold in Figure 10, they are based in the SQL view from Step 2), you only need to recompile the program.

```
F* item_join3if   e           k disk
d cust_data     e ds                  extname(custSum97)
d rowsFetched     s              9b 0

 /FREE
  rowsFetched = 0;
  exsr inzSubFile;

  dow not %eof;
    read item_join3;
    if %eof;
      leave;
    endif;
    rowsFetched = rowsFetched + 1;
    exsr loadSubFile;
  enddo;
  exsr dspSubFile;

  *inlr=*on;
  return;
```

Figure 10. Example of adding new externally described data structure

Step 3.3: Add embedded SQL statements to generate a result set

Now, all required structures are in place. The next HLL program change is to embed SQL statements to create the result set, including the DECLARE CURSOR and OPEN statements (bold text in Figure 11).

```
F* item_join3if   e           k disk
d cust_data     e ds                  extname(custSum97)
d rowsFetched     s              9b 0

 /FREE
  rowsFetched = 0;
  exsr inzSubFile;

  // Declare the cursor
  EXEC SQL
    DECLARE c1 CURSOR FOR
    SELECT *
    FROM custSum97
    ORDER BY customer;


  // Open the cursor
  EXEC SQL
    OPEN c1;

  dow not %eof;
    read item_join3;
    if %eof;
      leave;
    endif;
    rowsFetched = rowsFetched + 1;
    exsr loadSubFile;
  enddo;
  exsr dspSubFile;

  *inlr=*on;
  return;
```

Figure 11. Example of adding embedded SQL statement to declare and open the cursor

Two things should be pointed out at this stage:

- Because most of the logic (selection, joining, grouping) is defined in the SQL view, the SQL DECLARE CURSOR statement is greatly simplified.
- The ORDER BY clause was specified in the SQL SELECT statement — not in the SQL view. Recall that no access paths are associated with SQL. Consequently, you cannot define ordering within the view. You specify it in the SELECT statement that refers to the view.

Step 3.4: Replace native-access methods with SQL access methods

The final changes to make to the HLL program involve replacing all the RLA-access methods (SETLL, READ, CHAIN and others) with SQL-access methods. This is shown in Figure 12:

```
F* item_join3if   e           k disk
d cust_data     e ds                  extname(custSum97)
d rowsFetched     s              9b 0

 /FREE
  rowsFetched = 0;
  exsr inzSubFile;

  // Declare the cursor
  EXEC SQL
    DECLARE c1 CURSOR FOR
    SELECT *
    FROM custSum97
    ORDER BY customer;

  // Open the cursor
  EXEC SQL
    OPEN c1;

   //dow not %eof; **this line can be deleted **
    dow SQLSTAT <> '02000';

    //read item_join3; **this line can be deleted **
    EXEC SQL
      FETCH C1 INTO :cust_data
    //if %eof; **this line can be deleted **
    if SQLSTAT = '02000';
      leave;
    endif;
    rowsFetched = rowsFetched + 1;
    exsr loadSubFile;
  enddo;
  exsr dspSubFile;

  *inlr=*on;
  return;
```

Figure 12. Example of replacing RLA operations with embedded SQL statements (to retrieve rows in result set)

In the example shown in Figure 12, the following has been done:

- Commented out the lines of code that contain RLA methods (READ) and supporting statements (As indicated, you can delete the lines for improved readability.)
- Inserted the SQL FETCH statement and supporting operations

Step 6: Modify or eliminate CL program

After making all required conversion changes, remove the unneeded commands from the CL program. In the example in Figure 13, you can delete all commands shown in bold — they are no longer needed.

```
OVRDBF     FILE(ITEM_JOIN3) TOFILE(ITEM_FACT) +
             OVRSCOPE(*JOB) SHARE(*YES)
OPNQRYF    FILE((ITEM_FACT) (CUST_DIM) (TIME_DIM)) +
           FORMAT(ITEM_JOIN3)
           QRYSLT('TIME_DIM/YEAR *EQ 1997') +
           JFLD((ITEM_FACT/CUSTKEY CUST_DIM/CUSTKEY) +
                (ITEM_FACT/SHIPDATE TIME_DIM/DATEKEY)) +
           GRPFLD(CUST_DIM/CUSTOMER) +
           MAPFLD((TOTITEMS '%SUM(QUANTITY)') +
           (TOTREV '%SUM(REVENUE)') +
           (TRANSCOUNT '%COUNT')) +
           OPNSCOPE(*JOB)
CALL       PGM(PROC_ROWS)
CLOF       OPNID(ITEM_FACT)
DLTOVR     FILE(ITEM_JOIN3) LVL(*JOB)
```

Figure 13. The commands shown in bold font can be deleted

If you remove all the commands shown in bold, the only command left is the call to the HLL program PROC_ROWS. You can leave it this way and the application can run. However, it is recommended that you perform further analysis to see whether you can remove the entire CL program from the application. If you can change the process or program that invokes the CL program so that it calls PROC_ROWS directly, this removes an entry in the program stack and eliminates some overhead. This additional program-stack entry might not seem like much, but the savings can be significant in high-transaction environments. CL program overhead can be a performance bottleneck for heavily used transactions.

Other conversion considerations

Additional conversion considerations are worthy of discussion.

Set-at-a-time or row-at-a-time processing of result set

With OPNQRYF, you must read and process file rows one at a time, by using RLA methods in a HLL program (called *row-at-a-time* processing). SQL's *set-at-a-time* option processes multiple rows (a set of rows) with a single SQL statement. Thus, with SQL access methods, you can create a cursor and fetch (read) rows one at a time, or you can use set-at–a-time. With set-at-a-time, the specific implementation in your HLL varies, depending on your intention with the data. Here are the possible HLL implementations:

- **READ and FETCH:** To read a set of rows, the HLL program can define an array and fetch blocks of rows into the array. Depending on the size of array and number of rows you ultimately want to fetch, you might need to issue multiple FETCH statements to read all the desired rows. For example, if you want to read 100 000 rows from a table and your array is defined to contain 1000 elements, you must issue the FETCH statement 100 times to retrieve all the rows. Or, if you are only interested in the first 1000 rows, you need to issue the FETCH once.
- **UPDATE:** To update all rows that meet specific criteria, you can use a single SQL UPDATE statement.
- **DELETE:** To delete all rows that meet specific criteria, you can use a single SQL DELETE statement.
- **INSERT:** Similar to fetching, you can insert sets of rows through arrays and blocked inserts. The HLL defines an array, populates it and issues a blocked INSERT statement to add the rows to the table.

Example 1: Comparing row-at-a-time with set-at-a-time when deleting database rows

For further clarification on set-at-a-time processing, consider the following OPNQRYF implementation example that deletes all rows in table ORD_DTL whose YEAR column equals 1996. The CL program looks similar to the example shown in Figure 14:

```
OVRDBF     FILE(ORD_DTL) SHARE(*YES) OVRSCOPE(*JOB)
OPNQRYF    FILE((ORD_DTL)) QRYSLT('YEAR *EQ 1996')OPTION(*ALL)
CALL       PGM(DLT_ROWS)
CLOF       OPNID(ORD_DTL)
DLTOVR     FILE(ORD_DTL)LVL(*JOB)
```

Figure 14. CL program using OPNQRYF implementation to delete rows

Figure 15 shows the called HLL program DLT_ROWS.

```
     ford_dtl   uf   e             disk

     /FREE
      setll *start ord_dtl;
      dow not %eof;
        read ord_dtl;
        if %eof;
          leave;
        endif;
        delete ord_rcd;
      enddo;

      return;
      *inlr = *on;
     /END-FREE
```

Figure 15. RPG program that uses the OPNQRYF implementation to delete rows

In this example, OPNQRYF is used to open a file that contains only those rows with the YEAR column equal to 1996. The RPG DLT_ROWS program is then called to read and delete those rows, by employing row-at-a-time processing. After converting this application to SQL by using row-at-a-time processing, the CL program is no longer needed and the HLL program is modified (see Figure 16).

```
    d ord_row       e ds                  extname(ord_dtl)

     /FREE
      EXEC SQL
         DECLARE c1 CURSOR FOR
          SELECT *
           FROM ord_dtl
           WHERE year = 1996;
      saXEC SQL
         OPEN c1;
      dow SQLSTAT <> '02000';
        EXEC SQL
          FETCH c1 into :itemRow

      if SQLSTAT = '02000';
        leave;
      endif;

      EXEC SQL
        DELETE FROM item_fact
        WHERE CURRENT OF c1
      enddo;
      return;
      *inlr = *on;
     /END-FREE
```

Figure 16. RPG program that uses SQL the row-at-a-time implementation to delete rows

The row-at-a-time conversion works fine, but it is more complex than it needs to be. By implementing SQL set-at-a-time instead, the program looks similar to the example shown in Figure 17:

```
/FREE
 EXEC SQL
   DELETE FROM ord_dtl
   WHERE year = 1996;

 return;
 *inlr = *on;
/END-FREE
```

Figure 17. RPG program using SQL set-at-a-time implementation to delete rows

With SQL, you can use one statement to delete all the rows that meet the criteria — there is no need to read these rows one at a time and then issue the DELETE statement. Not only is this version of the application easier to understand and maintain, it clearly outperforms both the OPNQRYF and SQL row-at-a-time versions. Several iterations of tests were performed to measure the performance differences. Results of each of the methods are shown in Table 3.

Method	Rows deleted	Average elapsed time (seconds)
OPNQRYF row-at-a-time	1 930 560	53.349
SQL row-at-a-time	1 930 560	111.388
SQL set-at-a-time	1 930 560	30.071

Table 3. Results of using different methods to delete rows

Based on the performance results of this example conversion, it is clear that the SQL set-at-a-time implementation is the optimal choice. This version is easier to understand, easier to maintain and easily outperforms the OPNQRYF version. It also shows that there can be a significant difference in performance in SQL row-at-a-time implementations as compared to set-at-a-time. This is a key point to consider because many RPG programmers are accustomed to programming in the row-at-a-time paradigm. This is not meant to be overly critical — it is, after all, the only way to write code then using native-access methods. Set-at-a-time is a different programming mindset and is often overlooked by programmers who move to SQL access methods. Therefore, it is strongly recommended that you design your applications to use SQL set-at-a-time implementations. It might mean the difference between an application that performs well and one that does not.

Example 2: SQL Set-at-a-time to read rows from a database

As stated earlier, an SQL set-at-a-time implementation can also be used when reading the rows of a file. The example in Figure 18 shows how to use an array (defined with 30 elements) as the host-variable structure so that a single SQL FETCH statement can retrieve and return multiple rows in the result set with one trip down to the database. One fetch execution populates all 30 elements of the array, as long as there are that many rows to satisfy the local selection. The RPG program processes the array until it has reached the last element of the array (at which time, it issues another fetch to get the next 30 rows in the result set).

```
d ord_dtl_Array e ds                  extname(ord_dtl)
d                                     dim(100)
d rowsFetched     s              9b 0
      /FREE
       EXSR printHeader;

       //** Declare the cursor
       EXEC SQL
          DECLARE c1 CURSOR FOR
          SELECT *
          FROM ord_dtl
          WHERE year = 1996;
       //** Open the cursor
       EXEC SQL
          OPEN c1;

       DOW SQLSTAT <> '02000';
         //*** Fetch matching records
         EXEC SQL
           FETCH NEXT FROM C1 FOR 100 ROWS INTO :ord_dtl_Array;
         IF SQLSTAT = '02000';
           LEAVE;
         ENDIF;
         for z = 1 to sqler3;
           EXSR printDetails;
           rowsFetched = rowsFetched + 1;
         endfor;
       enddo;


       EXSR printFooter;
       *inlr=*on;
       return;
      /END-FREE
```

Figure 18. RPG program using SQL set-at-a-time implementation to read rows and load a subfile

Static or dynamic SQL

Embedded SQL offers two implementation choices: Static or dynamic SQL. For the conversion process, the one chosen depends on how you built your OPNQRYF parameter values. If the values for selection, ordering, joining and grouping are passed into the CL program as parameters and the command is built dynamically, you probably should use dynamic SQL. Yet, if these values are hardcoded in the program (or if the only thing that changes is the value of the variables that are used in the QRYSLT parameter), then you can use static SQL, instead. This section describes each implementation type.

Static SQL

With static SQL, the basic structure of the embedded SQL statement does not change. Referenced tables and columns are the same each time the statement runs. The only things that can vary from one statement execution to the next are the host-variable values that are used for local selection (in the WHERE clause). Although not as flexible as dynamic SQL, this technique provides a performance advantage because the statement can be partially optimized when the HLL compiles. Because the columns that are specified for selection, joining, ordering and grouping are defined at compile time, the database engine can use this information to create a preliminary access plan during program compilation. The plan is stored in the program object and is ready to use when the program is called.

If static SQL is the chosen method for the conversion, you can replace the OPNQRYF command with two embedded SQL statements: DECLARE CURSOR and OPEN CURSOR.

Dynamic SQL

With dynamic SQL, the contents of the SQL statements are unknown until the program actually runs. Thus, the HLL program must first resolve the statement by issuing the SQL PREPARE command. This converts it from a character string to an executable SQL statement. After the statement is prepared, it can be run. With this method, the entire statement can be flexible (for example, passed into the program as an input parameter) and need not be known when compiling the HLL program. Although this flexibility is certainly nice, there is a bit of a performance price to pay. Because the statement is not known during program creation, it cannot be optimized at this time. Optimization must be deferred until the program's run time, and occurs when the PREPARE statement runs. If you decide to embed dynamic statements into your HLL program and fetch the rows for further processing, you can replace the OPNQRYF command with the following embedded SQL statements: PREPARE, DECLARE CURSOR or OPEN CURSOR.

Dynamic SQL consideration

If the PREPARE, DECLARE or OPEN statements run repeatedly, then you must use the DLYPRP compiler option to limit the validation to the OPEN statement. If the SQL statement does not change from one program execution to the next (only the host-variable values change), then the PREPARE and DECLARE statements only need to run one time.

```
PGM         PARM(&SELECTION &ORDERBY)

DCL         VAR(&SELECTION) TYPE(*CHAR) LEN(80)
DCL         VAR(&ORDERBY) TYPE(*CHAR) LEN(10)

OVRDBF      FILE(ORD_DTL) SHARE(*YES) OVRSCOPE(*JOB)
OPNQRYF     FILE((CUST_DIM)) QRYSLT(&SELECTION)
              KEYFLD((&ORDERBY *ASCEND))
CALL        PGM(PROC_ROWS)
CLOF        OPNID(ORD_DTL)
DLTOVR      FILE(ORD_DTL)LVL(*JOB)

ENDPGM
```

Figure 19. Example 2 — OPNQRYF to Dynamic SQL

In this example, the actual table columns to be used for selection and ordering are passed in as input parameters and can vary each time that the program is called. Consequently, this is not a candidate for static SQL conversion. You must implement it as a dynamic SQL statement where the statement can be dynamically built and prepared. After conversion, the CL program is eliminated and the HLL program named PROC_ROWS contains SQL statements to prepare the statement, as well as to declare and open the cursor. It looks similar to the statements shown in Figure 20.:

```
      /free
          sqlStatement = 'SELECT *            +
                          FROM ord_dtl        +
                          WHERE ' + selection +
                         'ORDER BY ' + orderBy
          exec sql PREPARE sqlStm from :sqlStatement;
          exec sql DECLARE c1 CURSOR FOR slqStm;
          exec sql OPEN C1 using :sqlStatement;
```

Figure 20. SQL statements that prepare the dynamic SQL statement

Note: Keep in mind that the input *selection* parameter, in its original form, is used for OPNQRYF parameter QRYSLT. Thus, you need to modify it to adhere to the proper SQL WHERE clause syntax.

When deciding which method to use, you must consider the tradeoff between application flexibility, reusability and performance. Each application is obviously different; it is up to you to measure the performance degradation with dynamic SQL and to determine whether this method's flexibility and reusability is offset by that loss in performance.

SELECT INTO when returning one row

Most of this white paper has focused on using OPNQRYF to process multiple rows in database tables. However, in some cases, you might use OPNQRYF to return a single row. A good example of this is returning a count of the rows that satisfy a specified WHERE clause condition. In such cases, some of the conversion techniques discussed previously (such as declaring a cursor, opening a cursor and fetching rows in the application) might not be required. In fact, you can probably eliminate these unnecessary statements and the resulting overhead by simply using the SQL statement SELECT INTO, as shown in Figure 21. This example retrieves the contents of the matching row into the host variable ord_dtl_var. In this case, it was not necessary to declare and open a cursor.

```
d ord_dtl_var   e ds                  extname(ord_dtl)
D custParm        s                   like(custKey)
c     *entry        plist
c                   parm                    custParm
      /FREE
         EXEC SQL
           SELECT *
          INTO :ord_dtl_var
          FROM cust_dim
          WHERE custkey = :custParm;
         if SQLSTAT = '02000';
           // row not found
         else
           // row found
         endif;
       *inlr=*on;
       return;
      /END-FREE
```

Figure 21. RPG program using SQL SELECT INTO to retrieve a single row

CPYFRMQRYF

A popular technique when using OPNQRYF is to send the output directly to another file. This is accomplished by using the Copy from Query File (CPYFRMQRYF) command. The purpose of this command is to copy the result set from an OPNQRYF request to another file. In effect, this immediately materializes the result set, allowing you to do things you previously could not, such as issue DSPPFM or RUNQRY commands to see the contents of the OPNQRYF result set. The target file of the CPYFRMQRYF command is often (but not always) created in the QTEMP library (for easy cleanup) and is typically used as the input for a subsequent HLL program call or even for another OPNQRYF command. An example of this type of CPYFRMQRYF implementation is shown in Figure 22:

```
PGM (&PARTNUMBER)

OVRDBF     FILE(ITEM_JOIN2) TOFILE(ITEM_FACT) +
             OVRSCOPE(*JOB) SHARE(*YES)
OPNQRYF    FILE((ITEM_FACT) (CUST_DIM)) +
             FORMAT(ITEM_JOIN2) KEYFLD((TERRITORY +
             *ASCEND) (CUSTOMER *ASCEND)) +
             JFLD((ITEM_FACT/CUSTKEY +
             CUST_DIM/CUSTKEY)) JDFTVAL(*NO) +
             OPNID(JOINFILEID) OPNSCOPE(*JOB)
CPYFRMQRYF FROMOPNID(JOINFILEID) TOFILE(QTEMP/WRKFILE) +
             MBROPT(*REPLACE) CRTFILE(*YES)
CALL       PROC_ROWS
CLOF       OPNID(JOINFILEID)
DLTOVR     FILE(ITEM_JOIN2) LVL(*JOB)

ENDPGM
```

Figure 22. CPYFRMQRY example

In this example, the result set of the OPNQRYF command is materialized into the WRKFILE file object in library QTEMP. The PROC_ROWS HLL program can then access WRKFILE just as it does any other table or view.

For the conversion process, if there is a requirement to materialize the result set, consider using CREATE TABLE AS (…) WITH DATA. You can convert the example shown in Figure 22 by using the SQL statement shown in Figure 23:

```
CREATE TABLE QTEMP/WRKFILE AS (
  SELECT TERRITORY, SALESREP, CUSTOMER, REVENUE, QUANTITY
  FROM ITEM_FACT A
  INNER JOIN CUST_DIM B ON A.CUSTKEY = B.CUSTKEY
  ORDER BY TERRITORY, CUSTOMER
) WITH DATA
```

Figure 23. CREATE TABLE AS example

Running this statement results in the creation of the temporary WRKFILE file in library QTEMP. The temp file is also populated with the result set of the SELECT statement.

The problem with these implementations (including the converted SQL version) is that they both involve the creation of a temporary object on the system. If possible, it is more efficient to simply eliminate the temporary file. Fortunately, even in more complex OPNQRFY implementations, there is a way to do this: using CTEs. For some examples of using CTEs to eliminate temporary objects in this type of conversion process, see the article in the System i Network entitled “Accessing Data through SQL views” at www.systeminetwork.com/artarchive/21029/Accessing_Data_Using_SQL_Views.html.

Summary

Even though you might agree with the merits of performing this conversion, you might have concluded that it is not a trivial exercise. If you are completely content with the performance and functions of your applications that use OPNQRYF, there is no need to convert them simply for the sake of change (unless your objective is to claim to have modernized the data-access methods of your application). However, if some of the advantages listed in this white paper (such as improved performance and SQL-only functions) seem valuable to you, or new requirements dictate that you need to make enhancements to an application that uses the OPNQRYF method, this conversion process is something you should seriously consider.

Appendix A: OPNQRYF command parameters and SQL equivalents

It is important to understand the OPNQRYF command parameters and how to control the behavior that they influence, by using equivalent SQL settings. To simplify the comparisons and examples, this appendix focuses on converting the OPNQRYF command to SQL and ignores peripheral aspects that you must also convert. Other sections in this white paper explain how to convert the peripheral environment.

File specifications (FILE)

This parameter specifies the physical or logical file processed by the OPNQRYF command (see Table 4).

OPNQRYF parameter	SQL equivalent
FILE	FROM schema table
Member name	No equivalent for member (consider using SQL ALIAS)
Record format	No equivalent for record format (consider using UNIONs)

Table 4. Using the FILE parameter

The equivalent for the FILE parameter is the FROM clause in the SQL statement. Both OPNQRYF and SQL support qualified library schemas. But, SQL does not support members or record formats (see Table 5). To support members, create an SQL alias that points to a specific member and use the alias in the SQL (see Table 6). To support a specific record format in a multiformat logical file, reference the underlying physical file directly (see Figure 24). To retrieve data from record format FORMAT3, specify the parameter information shown in Table 7.

OPNQRYF	OPNQRYF FILE((ORD_DTL)) FORMAT(ORD_DTL)
SQL	SELECT * FROM ord_dtl

Table 5. Simple physical-file reference

OPNQRYF	OPNQRYF FILE((ORD_DTL JANUARY))
SQL	CREATE ALIAS OrdersJanuary FOR ord_dtl (january); SELECT * FROM OrdersJanuary

Table 6. Physical file-member reference

```
A           R FORMAT1                   PFILE(CUST_MAST)
A             CUSNUM
A             CUSNAM
A             CUSLOC
A           K CUSNUM
A           K CUSNAM
A           S CUSNUM1                   COMP(NE '        ')

A           R FORMAT2                   PFILE(CUST_MAST)
A             CUSNUM2
A             CUSNAM2
A             CUSLOC2
A           K CUSNUM2
A           K CUSNAM
A           S CUSNUM2                   COMP(NE '        ')

A           R FORMAT3                   PFILE(CUST_CON)
A             CONNUM
A             CONNAM
A             CONRGN
A           K CONNUM
A           K CONNAM
```

Figure 24. Multiformat logical-file reference with specific record format (logical file definition)

OPNQRYF	OPNQRYF FILE((CUST_LF *FIRST FORMAT3))
SQL	SELECT * FROM CUST_CON

Table 7. Retrieving data from a record format

Logical file alert

Do not specify DDS logical files as the referenced database object in an SQL statement — this causes the CQE to process the query. Always specify either the physical file (table) or an SQL view in your SQL statements.

One restriction of OPNQRYF is its inability to retrieve the data in more than one record format. If the OPNQRYF command references a multiformat logical file, you must specify a specific record format within that logical file. This means that you cannot query more than one format (and its underlying physical file) in a single command (unless you take additional steps to join them together). Although SQL does not directly support the concept of record formats, you can overcome this single-format restriction in SQL by using UNIONS.

For example, suppose that customer information (customer number, name and location) is spread across multiple fields and files and you want to centralize this information so that your programs can access a single interface to retrieve this data. Your files look something like that shown in Table 8 and Table 9:

CUSNUM	CUSNAM	CUSLOC	CUSNUM2	CUSNAM2	CUSLOC2
12345	Jones, Ed	SE			
223344	Berry, Frank	NW			
			908343	Anderson, Bob	SE
			887733	Ellison, Brit	MW

Table 8. CUST_MAST

CONNUM	CONNAM	CONRGN
89732	Garber, Matt	NE
223344	Larson, Ed	NW
987777	Stephens, Jackie	MW

Table 9. CUST_CON

Notice that the customer numbers, names and locations are scattered throughout various fields and files. If the goal is to consolidate these into one logical file, you can use a multiformat logical, such as the one defined in Table 8 and Table 9. Such a logical file allows native-access methods, such as RPG's READ, READE or CHAIN, to retrieve all records in the logical file, regardless of the underlying record format. This is a very typical way of using multiformat logical files. However, because OPNQRYF requires a specific record format, you cannot easily use this command to access all the records throughout each of the three record formats in the logical file. To overcome this restriction in SQL, first create a view (through the use of UNION) that combines the underlying physical files, as shown in Table 10:

```
CREATE VIEW ALL_CUSTOMERS (CUSNUM, CUSNAM, CUSLOC) AS
SELECT CUSNUM, CUSNAM, CUSLOC FROM CUST_MAST WHERE CUSNUM1 <> ''
  UNION
SELECT CUSNUM2, CUSNAM2, CUSLOC2 FROM CUST_MAST WHERE CUSNUM2 <> ''
  UNION
SELECT CONNUM, CONNAM, CONRGN FROM CUST_CON
```

Table 10. Creating a view that combines underlying physical files

Now, through the view, you can access all the rows (that satisfy the selection criteria) from all three physical files. As such, it becomes the single interface for accessing this customer information. You can use the SQL statement shown in Figure 25 to generate the result set. The results of the SQL statement look similar to that shown in Table 11.

```
SELECT * FROM all_customers
ORDER BY cusnum
```

Figure 25. SQL statement that generates the result set

CUSNUM	CUSNAME	CUSLOC
12345	Jones, Ed	SE
223344	Berry, Frank	NW
908343	Anderson, Bob	SE
887733	Ellison, Brit	MW
89732	Garber, Matt	NE
223344	Larson, Ed	NW
987777	Stephens, Jackie	MW

Table 11. Results of the SQL statement shown in Figure 25

Open options (OPTIONS)

This parameter specifies the open option to use for the query file. The options you choose on the first full open of a file do not change on subsequent shared opens. You can specify *ALL or a value that combines *INP, *OUT,*UPD and *DLT in a list of up to four values in any order (see Table 12 and Table 13).

OPNQRYF parameter	SQL equivalent
OPTIONS(*INP)	DECLARE CURSOR cursorName FOR READ ONLY
OPTIONS(*OUT)	** Not required for SQL INSERT statement
OPTIONS(*UPD)	DECLARE CURSOR cursorName FOR UPDATE
OPTIONS(*DLT)	DECLARE CURSOR cursorName FOR UPDATE

Table 12. Defining up to four open options

OPNQRYF	OPNQRYF FILE((ORD_DTL)) FORMAT(ORD_DTL) OPTIONS(*INP)
SQL	DECLARE CURSOR cursorName FOR READ ONLY

Table 13. Specifying the open options

Format specifications (FORMAT)

This parameter specifies the format for records that are available through the open-query file (see Table 14).

OPNQRYF parameter	SQL equivalent
FORMAT File name Member name Record format	Specify columns in the SQL statement or create an SQL view that contains only the desired columns; then reference that view in your SQL statement

Table 14. Specifying the record format for records that are available through the open-query file

Instead of specifying the FORMAT that contains the columns to be returned, you specify the columns (that you want returned in the result set) in the SQL statement. As an alternative, if the file name that is specified in the FORMAT parameter is an SQL view, you can reference that view in your SQL statement and specify * (all columns) for projection (see Table 15). For example, SELECT * FROM view_name.

OPNQRYF	OPNQRYF FILE((ORD_DTL)) FORMAT(ORD_DTL)
SQL	SELECT LINENUMBER, QUANTITY, EXTENDEDPRICE FROM ORD_DTL

Table 15. specifying columns in result set

OPNQRYF	OPNQRYF FILE((ORD_DTL)) FORMAT(ORD_DTL)
SQL	CREATE VIEW ORD_DTL_VIEW (LINENUMBER , QUANTITY , EXTENDEDPRICE) AS (SELECT SELECT LINENUMBER, QUANTITY, EXTENDEDPRICE FROM ORD_DTL); SELECT * FROM ORD_DTL_VIEW

Table 16. Creating a view with columns, specify view in FROM clause

Query selection expression (QRYSLT)

This parameter specifies the selection values that you use (before grouping) to determine the available records through the open-query file (see Table 17).

OPNQRYF parameter	SQL equivalent
QRYSLT	WHERE clause of SQL SELECT statement

Table 17. Specifying the selection values for determining available records

The QRYSLT parameter maps to the WHERE clause of the SQL statement. However, it is not a direct mapping. The apostrophes that OPNQRYF requires are no longer needed with SQL. In addition, you have to convert the specified QRYSLT functions (such at *EQ) to the SQL equivalent (see Table 18). For more information, see Appendix B: OPNQRYF functions and SQL equivalents.

OPNQRYF	OPNQRYF FILE((ORD_DTL)) QRYSLT('YEAR *EQ 1997')
SQL	SELECT * FROM ORD_DTL WHERE YEAR = 1997

Table 18. Using the QRYSLT parameter

Key-field specifications and ordering (KFLD)

This parameter specifies the name of one or more key fields that you use to arrange the query records, You can also use it to specify that the access-path sequence of the first, or only file, member and record format (that is specified for the File Specifications (FILE) parameter) is used to arrange the query records (see Table 19).

OPNQRYF parameter	SQL equivalent
KFLD Key field File or element Key field order Order by absolute value	 ORDER BY clause of SQL SELECT statement DESC or ASC ABSVAL function in ORDER BY clause

Table 19. Specifying the key fields for arranging query records

The KLFD parameter maps fairly directly to the ORDER BY clause in the SQL statement (see Table 20, Table 21 and Table 22).

OPNQRYF	OPNQRYF FILE((ORD_DTL)) KFLD(PARTKEY)
SQL	SELECT * FROM ORD_DTL ORDER BY PARTKEY

Table 20. Returning all rows from ORD_DTL file, ordered by column PARTKEY

OPNQRYF	OPNQRYF FILE((ORD_DTL)) KFLD(PARTKEY *DESCEND)
SQL	SELECT * FROM ORD_DTL ORDER BY PARTKEY DESC

Table 21. Returning all rows from ORD_DTL file, sorted in descending order by column PARTKEY.

OPNQRYF	OPNQRYF FILE((ORD_DTL)) KFLD(DAYS_COMMIT_TO_RECEIPT *ASCEND *ABSVAL)
SQL	SELECT * FROM ORD_DTL ORDER BY ABSVAL(DAYS_COMMIT_TO_RECEIPT)

Table 22. Returning all rows from ORD_DTL file, ordered by the absolute value of column DAYS_COMMIT_TO_RECEIPT

Unique key fields (UNIQUEKEY)

This parameter specifies whether you want to restrict the query to records with unique key values, and specifies how many of the key fields must be unique (see Table 23).

OPNQRYF parameter	SQL equivalent
UNIQUEKEY	SQL SELECT statement using a derived table and the ROW_NUMBER function:

Table 23. Restricting the query to unique key values

For example, you can read records that use only some key fields. Assume you are processing a file with the sequence: SALESPERSON, COUNTRY and REGION, but need only one record per SALESPERSON and COUNTRY. Essentially, you only want to return the first record of the group (see Table 24).

OPNQRYF	OPNQRYF FILE((ORD_DTL)) KEYFLD((SALESREP) (COUNTRY) (REGION)) UNIQUEKEY(2)
SQL	SELECT * FROM (SELECT salesperson, country, region, customer, ROW_NUMBER() OVER (PARTITION BY salesperson, country ORDER BY salesperson, country, region) AS rowNum FROM ord_dtl) AS o WHERE rowNum = 1

Table 24. Reading records using only some of the key fields

Join field specifications (JFLD)

This parameter specifies whether the query joins records from multiple file members and how to join field values from the files, members and record formats (specified for the FILE parameter) to construct query records (Table 25).

OPNQRYF parameter	SQL equivalent
JFLD From field To field Join operator	Join syntax of SQL SELECT statement.

Table 25. Specifying whether the query joins records from multiple file members

OPNQRYF	OPNQRYF FILE((ITEM_FACT) (CUST_DIM)) FORMAT(ITEM_JOIN2) JFLD((ITEM_FACT/CUSTKEY CUST_DIM/CUSTKEY))
SQL	SELECT TERRITORY, SALESREP, CUSTOMER, REVENUE, QUANTITY FROM ITEM_FACT A INNER JOIN CUST_DIM B ON A.CUSTKEY = B.CUSTKEY

Table 26. Example of joining records from multiple file members

Join file order (JORDER)

For a join query, this parameter specifies whether the join order must match the order that is specified for the File Specifications (FILE) parameter (see Table 27 and Table 28).

OPNQRYF parameter	SQL equivalent
JORDER	QAQQINI setting FORCE_JOIN_ORDER

Table 27. Using the JORDER parameter

OPNQRYF	OPNQRYF FILE((ITEM_FACT) (CUST_DIM)) FORMAT(ITEM_JOIN2) JFLD((ITEM_FACT/CUSTKEY CUST_DIM/CUSTKEY)) JORDER(*FILE)
SQL	QAQQINI setting FORCE_JOIN_ORDER = *YES

Table 28. Example of specifying the join order

Join file order (JDFTVAL)

This parameter specifies whether the query file should include join records that use default values for fields from a join secondary file — when the secondary file does not contain a record with correct field values that satisfy the join connections specified on the Join Field Specifications (JFLD) parameter. It describes what the system should do if a record is missing from the secondary file (see Table 29).

OPNQRYF parameter	SQL equivalent
JDFTVAL	Use SQL LEFT OUTER JOIN or EXCEPTION JOIN

Table 29. Using the JDFTVAL parameter

The OPNQRYF JDFTVAL parameter value of *YES maps to an SQL LEFT OUTER JOIN (see Table 30).

OPNQRYF	OPNQRYF FILE((ITEM_FACT) (CUST_DIM)) FORMAT(ITEM_JOIN2) JFLD((ITEM_FACT/CUSTKEY CUST_DIM/CUSTKEY)) JDFTVAL(*YES)
SQL	SELECT TERRITORY, SALESREP, CUSTOMER, REVENUE, QUANTITY FROM ITEM_FACT A LEFT OUTER JOIN CUST_DIM B ON A.CUSTKEY = B.CUSTKEY

*Table 30. Using the JDFTVAL parameter with a value of *YES*

The value of *ONLYDFT maps to an SQL EXCEPTION JOIN (see Table 31).

OPNQRYF	OPNQRYF FILE((ITEM_FACT) (CUST_DIM)) FORMAT(ITEM_JOIN2) JFLD((ITEM_FACT/CUSTKEY CUST_DIM/CUSTKEY)) JDFTVAL(*ONLYDFT)
SQL	SELECT TERRITORY, SALESREP, CUSTOMER, REVENUE, QUANTITY FROM ITEM_FACT A EXCEPTION JOIN CUST_DIM B ON A.CUSTKEY = B.CUSTKEY

*Table 31. Using the JDFTVAL parameter with a value of *ONLYDFT*

Grouping field names (GRPFLD)

This parameter specifies the field names that are used to group query results (see Table 32 and Table 33).

OPNQRYF parameter	SQL equivalent
GRPFLD	GROUP BY clause of SQL SELECT statement

Table 32. Using the GRPFLD parameter

OPNQRYF	OPNQRYF FILE((ITEM_FACT)) FORMAT(ITMFMT) GRPFLD(CUSTKEY) MAPFLD((COUNT '%COUNT'))
SQL	SELECT CUSTKEY, COUNT(*) FROM ITEM_FACT GROUP BY CUSTKEY

Table 33. Example of grouping field names for query results

Group-selection expression (GRPSLT)

This parameter specifies the selection values that are used after grouping to determine which records are available through the open-query file (see Table 34).

OPNQRYF parameter	SQL equivalent
GRPSLT	HAVING clause of SQL SELECT statement

Table 34. Using the OPNQRYF parameter

For example, you can group the data by customer key and then analyze the revenue-sum field. In this case, you select only the summary records in which the revenue sum is less than 1 000 000 (see Table 35).

OPNQRYF	OPNQRYF FILE((ITEM_FACT)) FORMAT(ITMFMT) KEYFLD(CUSTKEY) GRPFLD(CUSTKEY) MAPFLD((COUNT '%COUNT') (REVSUM '%SUM(REVENUE)') (REVAVG '%AVG(REVENUE)') (REVMAX '%MAX(REVENUE)')) GRPSLT('REVSUM *LT 1000000')
SQL	SELECT CUSTKEY, COUNT(*), SUM(REVENUE),AVG(REVENUE),MAX(REVENUE) FROM ITEM_FACT GROUP BY CUSTKEY HAVING SUM(REVENUE) < 1000000 ORDER BY CUSTKEY

Table 35. Example of grouping data by customer key to analyze revenue

Mapped-field specifications (MAPFLD)

This parameter defines the query fields that are mapped or derived from other fields (see Table 36).

OPNQRYF parameter	SQL equivalent
MAPFLD Mapped field Field definition expression	Use system-supplied SQL functions and UDFs Implementation of Common table expressions and views

Table 36. Using the MAPFLD parameter

What makes the conversion of this parameter somewhat tricky is that OPNQRYF always maps the field definitions before the QRYSLT parameter is evaluated. This means that, unless the mapped field specifies the use of an aggregate function such as %AVG, %MIN or %MAX, the mapped field can be used in the QRYSLT parameter. (You can use the GRPLST list to perform selection on mapped fields that are aggregated). SQL does not work this way. To obtain this behavior with SQL, you must either create an SQL view with the derived column, or specify a Common Table Expression (CTE) in the SQL statements. If you are not using the derived column in the selection, there is no need to implement a CTE. One example of using a mapped field that is not used in the query-selection criteria is shown in Table 37:

OPNQRYF	OPNQRYF FILE((ORD_DTL)) FORMAT(ORDDETAIL) MAPFLD((CHAR6 '%DIGITS(ORDDATE)') (ORDYEAR '%SST(CHAR6 5 2)' *CHAR 2))
SQL	SELECT SUBSTRING(DIGITS(orddate) , 3 , 2) as ordyear, orderpriority, linestatus, shipmode, revenue FROM ORD_DTL

Table 37. Example of using the mapped-field parameter

Table 38 shows another example of using a mapped field in query-selection criteria. The SQL version has two statements: one to create an SQL view and one to reference that SQL view in a SELECT statement.

OPNQRYF	OPNQRYF FILE((ORD_DTL)) FORMAT(ORDDETAIL) QRYSLT('ORDYEAR *EQ "05" ') MAPFLD((CHAR6 '%DIGITS(ORDDATE)') (ORDYEAR '%SST(CHAR6 5 2)' *CHAR 2))
SQL	CREATE VIEW ord_dtl_view AS (SELECT SUBSTRING(DIGITS(orddate) , 3 , 2) as ordyear, orderpriority, linestatus, shipmode, revenue FROM ord_dtl); SELECT * FROM ord_dtl_view WHERE ordyear = '05'

Table 38. Another example of using the mapped-field parameter

Yet another example (see Table 40) involves a mapped field that you might use in the query-selection criteria. The SQL version uses a CTE:

OPNQRYF	OPNQRYF FILE((ORD_DTL)) FORMAT(ORDDETAIL) QRYSLT('ORDYEAR *EQ "05" ') MAPFLD((CHAR6 '%DIGITS(ORDDATE)') (ORDYEAR '%SST(CHAR6 5 2)' *CHAR 2))
SQL	WITH cte1 AS (SELECT SUBSTRING(DIGITS(orddate) , 3 , 2) as ordyear, orderpriority, linestatus, shipmode, revenue FROM ORD_DTL) SELECT * FROM cte1 WHERE ordyear = '05'

Table 39. Yet another esample of using the mapped-field parameter

Ignore decimal-data errors (IGNDECERR)

This parameter specifies whether the system ignores decimal data errors during query processing. If you specify *YES, the system ignores decimal data errors. When errors in decimal data are encountered, the not-valid sign or digits are automatically changed to valid values (see Table 40).

OPNQRYF parameter	SQL equivalent
IGNDECERR	Use the High level language (HLL) compiler parameter FIXNBR

Table 40. Using the IGNDECERR parameter

The FIXNBR parameter is not available with the HLL precompiler. To specify this parameter, you must issue the CRTSQLRGPI command with the OPTION(*NOGEN) parameter to instruct the precompiler not to call the RPG compiler. As a result, no module, program or service program is created. However, the precompiler still generates a temporary source-file member by the same name. By default, this member is in the QTEMP library, in the QSQLTEMP1source file. In this member, embedded SQL statements are converted to comments and calls to the SQL run time. After you locate the temporary member, you can issue the CRTBNDRPG command (see Table 41).

OPNQRYF	OPNQRYF FILE((ORD_DTL))
SQL	CRTSQLRPGI OBJ(TESTLIB/TESTPGM1) SRCFILE(TESTLIB/QRPGLESRC) OPTION(*NOGEN) CRTBNDRPG PGM(TESTLIB/TESTPGM1) SRCFILE(QTEMP/QSQLTEMP1) SRCMBR(*PGM) FIXNBR(*ZONED)

Table 41. Example of using the IGNDECERR parameter

Note: The recommended way of handling decimal data errors is to cleanse the offending data. In other words, if zoned or packed-decimal fields contain blanks or other nonnumeric values, you need to update these values to contain zeroes or numeric values. To start this kind of analysis, issue the STRSQL command and use an SQL statement, such as the following, on all your numeric columns in a table:

```
SELECT COUNT(DISTNICT numeric_column1), COUNT(DISTNICT numeric_column2),
       COUNT(DISTNICT numeric_column3), COUNT(DISTNICT numeric_column4)
FROM file_name
```

This points out numeric columns that contain invalid data. In the results (shown in the following example), you can see that the first numeric column of the SELECT statement (numeric_column1) contains invalid data (denoted by the '+++++++++++++++' value in the first column).

```
        COUNT           COUNT           COUNT           COUNT
++++++++++++++            9384            5111            6383
```

Use the STRSQL command to see the invalid output, as just shown. When you use the iSeries Navigator *Run SQL Scripts* interface, the count columns show valid values. Determining which rows contain invalid data is a trickier. The article *Ending those Decimal Data Error Blues* provides example RPG code to help you perform this analysis (www.ibmsystemsmag.com/i5/july03/enewsletterexclusive/13470p1.aspx).

Open-file identifier (OPNID)

This parameter specifies the identifier that you use to name the open-query file — so that it is referred to on the Close File (CLOF) or Position Database File (POSDBF) command when it is closed (see Table 42).

OPNQRYF parameter	SQL equivalent
OPNID	There is no SQL equivalent for this parameter. However, because no file needs to be closed by the CLOF command, this parameter can be eliminated.

Table 42. Using the OPNID parameter

Limit to sequential only (SEQONLY)

This parameter specifies the use of sequential-only file processing and specifies the number of records to process as a group when performing read or write operations to the open-query file (see Table 43).

OPNQRYF parameter	SQL equivalent
SEQONLY Sequential only . . . . . . Number of records	OVRDBF CL command Sequential only . . . . . . Number of records . . . . . .

Table 43. Using the SEQONLY parameter

To obtain the same blocking behavior, issue the OVRDBF command prior to issuing the SQL SELECT statement (see Table 44).

OPNQRYF	OPNQRYF FILE((ORD_DTL)) SEQONLY(*YES 2000)
CL command (prior to SQL statement)	OVRDBF FILE(ORD_DTL) SEQONLY(*YES 2000)

Table 44. Example of using the SEQONLY parameter

Commitment control active (COMMIT)

This parameter specifies whether this file is placed under commitment control. You use this setting in conjunction with the Start Commitment Control (STRCMTCTL) command to establish either a job level or activation-group level commitment definition (see Table 45).

OPNQRYF parameter	SQL equivalent
COMMIT	Use the HLL precompiler parameter COMMIT

Table 45. Using the COMMIT parameter

Because you specify the isolation level in the HLL precompiler setting, using the STRCMTCTL command is no longer required. You should remove this command from the CL program (see Table 46).

OPNQRYF	OPNQRYF FILE((ORD_DTL)) COMMIT(*YES) STRCMTCTL LCKLVL(*CS)
HLL Pre-compiler	CRTSQLRPGI OBJ(TESTLIB/TESTPGM1) SRCFILE(TESTLIB/QRPGLESRC) COMMIT(*CS)

Table 46. Example of enabling commitment control with the isolation level of Cursor Stability

Open scope (OPNSCOPE)

This parameter specifies the extent of influence (scope) of the open operation (see Table 47).

OPNQRYF parameter	SQL equivalent
OPNSCOPE	No longer required because OPNQRYF command is not used

Table 47. Using the OPNSCOPE parameter

Duplicate key check (DUPKEYCHK)

This parameter (*YES or *NO) specifies whether duplicate-key checking and feedback is provided on input and output commands. Use the default (*NO) if the programs are not written in COBOL or ILE C and C++, or if your program does not use the returned duplicate-key feedback information (see Table 48).

OPNQRYF parameter	SQL equivalent
DUPKEYCHK	Separate SQL SELECT statement – see below.

Table 48. Using the DUPKEYCHK parameter

No direct equivalent exists for this parameter. However, you can embed a separate SQL statement to perform the duplicate-key check, and then examine the result set. For example, to check for duplicate keys in the customer file for customer Customer#000000209, the following SQL statement returns a result set with the customer and the number of occurrences. If no duplicates exist, the result set is empty.

```
SELECT customer, COUNT(customer) AS custcount FROM cust_dim
WHERE customer = 'Customer#000000209'
GROUP BY customer HAVING ( COUNT(customer) > 1 )
```

Allow copy of data (ALWCPYDTA)

This parameter lets the system copy data from the files, members and record formats that are specified for the FILE parameter. If this parameter is used, the system opens the query file to the copy (see Table 49 and Table 50).

OPNQRYF parameter	SQL equivalent
ALWCPYDTA	Use the HLL precompiler parameter ALWCPYDTA

Table 49. Using the ALWCPYDTA parameter

OPNQRYF	OPNQRYF FILE((ORD_DTL)) ALWCPYDTA(*NO)
HLL precompiler command for RPG	CRTSQLRPGI OBJ(TESTLIB/TESTPGM1) SRCFILE(TESTLIB/QRPGLESRC) ALWCPYDTA(*NO)

Table 50. Example of using the ALWCPYDTA parameter

Performance optimization (OPTIMIZE)

This parameter specifies the optimization goal that the system uses to decide how to perform the selection and join processing that satisfies other specifications on this command (see Table 51 and Table 52).

OPNQRYF parameter	SQL equivalent
OPTIMIZE Performance optimization Number of records .	OPTIMIZE FOR n ROWS

Table 51. Using the OPTIMIZE parameter

OPNQRYF	OPNQRYF FILE((ORDERS)) OPTIMIZE(*FIRSTIO 10)
SQL	SELECT * FROM ORD_DTL OPTIMIZE FOR 10 ROWS

Table 52. Example of using the OPTIMIZE parameter

Optimize all access paths (OPTALLAP)

This parameter tells the query optimizer to consider all access paths that exist over the query files when determining how to do the query (see Table 53). OPTALLAP has no equivalent if the query goes to SQE. Therefore, if you want SQE process the SQL, this parameter is irrelevant.

OPNQRYF parameter	SQL equivalent
OPTALLAP	QAQQINI setting OPTIMIZE_STATISTIC_LIMITATION (CQE only)

Table 53. Using the OPTALLAP parameter

Sort sequence (SRTSEQ)

This parameter specifies the sort sequence for sorting and grouping selections that are specified for the QRYSLT or GRPSLT parameters, as well as joins that are specified for the JFLD parameter, the ordering that is specified for the KEYFLD parameter, the grouping that is specified for the GRPFLD parameter, and the %MIN or %MAX built in functions, or unique key values that are specified for the UNIQUEKEY parameter (see Table 54 and Table 55).

OPNQRYF parameter	SQL equivalent
SRTSEQ	Use the HLL precompiler parameter SRTSEQ

Table 54. Using the SRTSEQ parameter

OPNQRYF	OPNQRYF FILE((ORD_DTL)) KEYFLD(JOB) SRTSEQ(*LANGIDUNQ)
HLL Pre-compiler command for RPG	CRTSQLRPGI OBJ(TESTLIB/TESTPGM1) SRCFILE(TESTLIB/QRPGLESRC) SRTSEQ(*LANGIDUNQ)

Table 55. Example of using the SRTSEQ parameter

Language ID (LANID)

This parameter specifies the language identifier to be used when SRTSEQ(*LANGIDUNQ) or SRTSEQ(*LANGIDSHR) is specified (see Table 56 and Table 57).

OPNQRYF parameter	SQL equivalent
LANGID	Use the HLL precompiler parameter SRTSEQ

Table 56. Using the LANGID parameter

OPNQRYF	OPNQRYF FILE((ORD_DTL)) KEYFLD(JOB) SRTSEQ(*LANGIDUNQ) LANGID(ITA)
HLL precompiler command for RPG	CRTSQLRPGI OBJ(TESTLIB/TESTPGM1) SRCFILE(TESTLIB/QRPGLESRC) SRTSEQ(*LANGIDUNQ) LANGID(ITA)

Table 57. Example of using the LANGID parameter

Final-output CCSID (CCSID)

This parameter specifies the coded character set identifier (CCSID) in which data from character, DBCS-open, DBCS-either and graphic fields are returned (see Table 58 and Table 59).

OPNQRYF parameter	SQL equivalent
CCSID	Use the Control specifications (H Specs) of RPG compiler

Table 58. Using the CCSID parameter

OPNQRYF	OPNQRYF FILE((ORD_DTL)) CCSID(13488)
Control Spec in HLL program	H CCSID(*GRAPH :13488)

Table 59. Example of using the CCSID parameter

Type of open (TYPE)

This parameter specifies the level at which the Reclaim Resources (RCLRSC) command closes the file.

OPNQRYF parameter	SQL equivalent
TYPE	No equivalent — not required because the CLOF command does not need to close any file

Table 60. Using the TYPE parameter

Appendix B: OPNQRYF functions and SQL equivalents

Listed in this section are the OPNQRYF operators and built-in functions — along with the equivalent SQL predicate. Table 61 and Table 62 provide quick summaries of the differences.

Description	OPNQRYF operator	SQL predicate
Equal	*EQ	=
Greater than	*GT	>
Greater than or equal to	*GE	>=
Less than	*LT	<
Less than or equal to	*LE	<=
Contains	*CT	LIKE
OR operator	%OR	OR
AND operator	%AND	AND
NOT	%NOT	NOT
Range of values	%RANGE	BETWEEN
Set of values	%VALUES	IN
Wildcard	%WLDCRD	LIKE

Table 61. Differences between the OPNQRYF operator and the SQL predicate

Description	OPNQRYF built-in function	SQL built-in function
Absolute value	%ABSVAL *(numeric-argument)*	ABSVAL (expression)
Arc cosine	%ARCCOS *(numeric-argument)*	ARCCOS (expression)
Antilogarithm (base 10)	%ANTILOG*(numeric-argument)*	ANTILOG (expression)
Arc sine	%ASIN (numeric-argument)	ASIN (expression)
Arc tangent	%ATAN *(numeric-argument)*	ATAN (expression)
Hyperbolic arc tangent	%ATANH *(numeric-argument)*)	ATANH (expression)
Average	%AVG *(numeric-argument)*	AVG (expression)
Character representation of the date time	%CHAR (date/time-argument date/time-format)	CHAR (expression)
Cosine	%COS *(numeric-argument)*	COS (expression)
Hyperbolic cosine	%COSH *(numeric-argument)*	COSH (expression)
Cotangent	%COT *(numeric-argument)*	COT (expression)
Number of records in group	%COUNT	COUNT(*)
Current date	%CURDATE	CURDATE
Current server name	%CURSERVER	DATABASE
Current time	%CURTIME	CURTIME or CURRENT TIME special register
Current timestamp	%CURTIMESTP	CURRENT TIMESTAMP special register
Current time zone	%CURTIMEZONE	CURRENT TIMEZONE special register
The date part of the argument	%DATE *(date/time-argument)*	DATE (expression)
Day part of the argument	%DAY *(date/time-argument)*	DAY (expression)
Integer representation of date	%DAYS *(date/time-argument)*	DAYS (expression)
Character representation of numeric value	%DIGITS *(numeric-argument)*	DIGITS (expression)
Labeled duration of days	%DURDAY *(integer-argument)*	TIMESTAMPDIFF (*numeric-expression* , *string-expression*)
Labeled duration of hours	%DURHOUR *(integer-argument)*	TIMESTAMPDIFF (*numeric-expression* , *string-expression*)
Labeled duration of microseconds	%DURMICSEC *(integer-argument)*	TIMESTAMPDIFF (*numeric-expression* , *string-expression*)
Labeled duration of minutes	%DURMINUTE (integer-argument)	TIMESTAMPDIFF (*numeric-expression* , *string-expression*)
Labeled duration of months	%DURMONTH *(integer-argument)*	TIMESTAMPDIFF (*numeric-expression* , *string-expression*)
Labeled duration of seconds	%DURSEC *(integer-argument)*	TIMESTAMPDIFF (*numeric-expression* , *string-expression*)

Labeled duration of years	%DURYEAR *(integer-argument)*	TIMESTAMPDIFF (*numeric-expression* , *string-expression*)
Base of the natural logarithm (e) raised to a power specified by the argument	%EXP *(numeric-argument)*	EXP (*expression)*
Partition number of a set of values	%HASH *(expression-argument)*	HASH (*expression)*
Hexadecimal equivalent of the argument's value	%HEX *(argument)*	HEX (*expression)*
Hour part of the argument	%HOUR *(date/time-argument)*	HOUR (*expression)*
Length of a value.	%LEN (length-argument)	LENGTH (*expression)*
Natural logarithm of argument	%LN (numeric-argument)	LN (*expression)*
Common logarithm (base 10) of the argument	%LOG *(numeric-argument)*	LOG (*expression)*
Maximum value in a set of values	%MAX *(numeric-or-string-or-date/time-argument ...))*	MAX *(expression, expression)*
Microsecond	%MICSEC *(date/time-argument)*	MICROSECOND *(expression)*
Minimum value in a set of values	%MIN *(numeric-or-string-or-date/time-argument ...)*	MIN *(expression, expression)*
Minute part of the argument	%MINUTE (date/time-argument)	MINUTE *(expression)*
Month part of the argument	%MONTH *(date/time-argument)*	MONTH *(expression)*
Relational database name for the record retrieved	%NODENAME *(integer-argument)*	DBPARTITIONNAME (*table-designator*) or NODENAME (*table-designator*)
Node number	%NODENUMBER	DBPARTITIONNUM (*table-designator*) or NODENUMBER (*table-designator*)
Value of the first non-null expression in the argument	%NONNULL *(argument ...)*	COALESCE (*expression*, *expression*) or IFNULL (*expression*, *expression*)
Partition map index number of a row	%PARTITION *(integer-argument)*	HASHED_VALUE (*table-designator*) or PARTITION(*table-designator*)
Second part of the argument	%SECOND (date/time-argument)	SECOND *(expression)*
Sine of the argument	%SIN (numeric-argument)	SIN (*expression)*
Hyperbolic sine of the argument	%SINH *(numeric-argument*	SINH (*expression)*
Square root of the argument.	%SQRT *(numeric-argument)*	SQRT (*expression*)
Substring of a string	%SST *(string-argument start-position-expression <length-expression>)*	SUBSTR (*expression*, *start, <length>*) or SUBSTRING (*expression*, *start, <length>*)
Standard deviation of argument for the group of records	%STDDEV *(numeric-argument)*	STDEV (*expression*) or STDEV_POP (*expression*)
Result string with strip character removed from string argument as specified by the strip function	%STRIP	STRIP (*expression*)
Substring of a string	%SUBSTRING (*string-field-name start-position length)*	SUBSTR (*expression*, *start, <length>*) or SUBSTRING (*expression*, *start, <length>*)
Sum of all values for the argument in the group of records	%SUM *(numeric-argument)*	SUM *(numeric-expression)*
Tangent of the argument	%TAN *(numeric-argument*)	TAN (*expression*)
Hyperbolic tangent of argument	%TANH *(numeric-argument)*	TANH (*expression*)
Time part of the argument	%TIME *(date/time-argument)*	TIME *(expression)*
Returns a timestamp from its argument or arguments.	%TIMESTP (*date/time-argument date/time-argument)*	TIMESTAMP (*expression-1, expression-2*)
User-profile name of the job in which the query is running	%USER	USER special register or SESSION USER special register
Variance of the argument for the group of records	%VAR *(numeric-argument)*	VARIANCE_SAMP (*expression*) or VAR_SAMP (*expression*)
Returns a translated string	%XLATE *(string-argument qualified-table)*	TRANSLATE (*expression, to-string, from-string*)
Bitwise 'XOR' (logical exclusive or) of the arguments	%XOR *(string-argument...)*	XOR (*expression*)
Year part of the argument	%YEAR *(date/time-argument)*	YEAR *(expression)*

Table 62. OPNQRYF built-in functions and the equivalent SQL predicate

Appendix C: Conversion examples and performance measurements

Benchmark tests had the following characteristics:

- Each test ran five times.
- All referenced database objects were purged from memory prior to each execution.
- Tests were performed against the same files in the same libraries.

Figure 26, Figure 27 and Figure 28 show the template programs used in the performance tests. Notice that the OPNQRYF command and SQL SELECT statement in the templates are followed by '???????????'. This means that these are substituted with the specified OPNQRYF command and the SQL SELECT statement that is specified in each of the testcase sections that follow in this section of the white paper.

```
            PGM

            SETOBJACC  OBJ(FILE_NAME) OBJTYPE(*FILE) POOL(*PURGE)

            OVRDBF     FILE(FILE_NAME) OVRSCOPE(*JOB) SHARE(*YES)
            OPNQRYF    ??????????????  OPNSCOPE(*JOB)
            CALL       PGM(PROC_ROWS)
            CLOF       OPNID(ITEM_FACT)
            DLTOVR     FILE(ITEM_FACT) LVL(*JOB)

            ENDPGM
```

Figure 26. CL example: OPNQRYF template

```
     Ffile_name if   e             disk

     d rowsFetched     s               9b 0
      /FREE
       rowsFetched = 0;
       setll *start file_name;
       dow not %eof;
         read file_name;
         if %eof;
           leave;
         endif;
         EXSR printDetails;
         rowsFetched = rowsFetched + 1;
       enddo;

       *inlr=*on;
       return;
      /END-FREE
```

Figure 27. Code snippet of RPG program PROC_ROWS: Processes rows created by OPNQRY template program

```
d varArray      e ds                  extname(fileName)
d                                     dim(100)
d rowsFetched     s              9b 0
      /FREE

       //** Declare the cursor
       EXEC SQL
          DECLARE c1 CURSOR FOR
          ??????????????;
       //** Open the cursor
       EXEC SQL
          OPEN c1;

       DOW SQLSTAT <> '02000';
         //*** Fetch matching records
         EXEC SQL
           FETCH NEXT FROM C1 FOR 100 ROWS INTO :varArray;
         IF SQLSTAT = '02000';
           LEAVE;
         ENDIF;
         for z = 1 to sqler3;
           EXSR printDetails;
           rowsFetched = rowsFetched + 1;
         endfor;
       enddo;

      *inlr=*on;
       return;
      /END-FREE
```

Figure 28. Code snippet of RPG program PROC_ROWS after program has been converted to use SQL access: Uses blocked fetching techniques when retrieving the rows from the table.

Dynamic record selection

Table 63 and Table 64 show the testcase command statement and results for the dynamic record selection:

OPNQRYF	OPNQRYF FILE((CUST_DIM)) QRYSLT('SALESREP *EQ "SalesPerson#00007"')
SQL	SELECT * FROM CUST_DIM WHERE SALESREP = 'SalesPerson#00007'

Table 63. Testcase command statement for the dynamic record selection

		Average elapsed time (seconds)		
Rows in table	Rows selected	OPNQRYF	SQL	Commentary on results
1 500 000	148 700	6756	3039	SQL was more than 2.2 times faster

Table 64. Testcase results for the dynamic record selection

Dynamic ordering

Table 65 and Table 66 show the testcase command statement and the results for dynamic ordering:

OPNQRYF	OPNQRYF FILE((ITEM_FACT)) KEYFLD((DAYS_00001 *ASCEND))
SQL	SELECT * FROM item_fact ORDER BY days_00001

Table 65. Testcase command statement for dynamic ordering

		Average elapsed time (seconds)		
Rows in table	Rows ordered	OPNQRYF	SQL	Commentary on results
6 001 215	6 001 215	94 944	30 960	SQL was more than three times faster

Table 66. Testcase results for dynamic ordering

Grouping

Table 67 and Table 68 show the testcase command statement and the results for grouping:

OPNQRYF	OPNQRYF FILE((ITEM_FACT)) FORMAT(REVTOTAL) KEYFLD((REVTTL *DESCEND)) GRPFLD(YEAR MONTH) MAPFLD((REVTTL '%SUM(REVENUE)'))
SQL	SELECT * FROM item_fact ORDER BY days_00001

Table 67. Testcase command statement for grouping

			Average elapsed time (seconds)		
Rows in table	Rows processed	# of distinct groups	OPNQRYF	SQL	Commentary on results
6 001 215	6 001 215	36	101 572	2266	Testcase with the most dramatic improvement (SQL version was 44 times faster)

Table 68. Testcase results for grouping

Dynamic joining

Table 69 and Table 70 show the testcase command statement and the results for dynamic joining:

OPNQRYF	OPNQRYF FILE((ITEM_FACT) (CUST_DIM)) FORMAT(ITEM_JOIN2) KEYFLD((TERRITORY *ASCEND) (CUSTOMER *ASCEND)) JFLD((ITEM_FACT/CUSTKEY CUST_DIM/CUSTKEY)) JDFTVAL(*NO)
SQL	SELECT TERRITORY, SALESREP, CUSTOMER, REVENUE, QUANTITY FROM ITEM_FACT A INNER JOIN CUST_DIM B ON A.CUSTKEY = B.CUSTKEY ORDER BY TERRITORY, CUSTOMER

Table 69. Testcase command statement for dynamic joining

		Average elapsed time (seconds)		
Rows in table	Rows ordered	OPNQRYF	SQL	Commentary on results
6 001 215	6 001 215	122 687	32 818	SQL was more than 3.73 times faster.

Table 70. Testcase results for dynamic joining

Unique-key processing

Table 71 and Table 72 show the testcase command statement and results for unique-key processing:

OPNQRYF	OPNQRYF FILE((CUST_DIM)) KEYFLD((SALESREP) (COUNTRY) (REGION)) UNIQUEKEY(2)
SQL	SELECT * FROM (SELECT salesperson, country, region, customer, ROW_NUMBER() OVER (PARTITION BY salesperson, country ORDER BY salesrep, country, region) AS rowNum FROM cust_dim) AS o WHERE rowNum = 1

Table 71. Testcase command statement for unique-key processing

			Average elapsed time (seconds)		
Rows in table	Rows processed	Number of distinct groups	OPNQRYF	SQL	Commentary on results
150 000	1 500 000	250	6566	4789	SQL version was 27% faster.

Table 72. Testcase results for unique-key processing

Final total-only processing

Table 73 and Table 74 show the testcase command statement and results or unique-key processing:

OPNQRYF	OPNQRYF FILE((ITEM_FACT)) FORMAT(FINTOT) MAPFLD((ROWCNT '%COUNT') (TOTREV'%SUM(REVENUE)') MAXREV '%MAX(REVENUE)'))
SQL	SELECT COUNT(*), SUM(revenue)AS total_revenue, MAX(revenue)AS max_revenue FROM item_fact

Table 73. Testcase command statement for final total-only processing

		Average elapsed time (seconds)		
Rows in table	Rows ordered	OPNQRYF	SQL	Commentary on results
6 001 215	6 001 215	24 974	3711	SQL was more than 6.7 times faster.

Table 74. Testcase results for final total-only processing

Random access of result set

A commonly used implementation lets the OPNQRYF command generate a result set that the HLL program can then access randomly (rather than reading it sequentially). This amounts to a two-step filtering process:

1. OPNQRYF uses the QRYSLT parameter to select a subset of the data and the FORMAT and KEYFIELD parameters to specify that the result set be in a keyed sequence.
2. This gives the called HLL program the ability to perform further selection filtering against the result set by using random-access operations (such as SETLL, READE and CHAIN).

Consider the example (shown in Figure 29 and Figure 30) of this implementation:

```
            PGM

            OVRDBF     FILE(ITEM_JOIN2) TOFILE(ITEM_FACT) +
                         OVRSCOPE(*JOB) SHARE(*YES)
            OPNQRYF    FILE((ITEM_FACT) (CUST_DIM)) +
                         FORMAT(ITEM_JOIN) +
                         QRYSLT('SALESREP *EQ "SalesPerson#00009”') +
                         KEYFLD((TERRITORY *ASCEND) (CUSTOMER +
                         *ASCEND)) JFLD((ITEM_FACT/CUSTKEY +
                         CUST_DIM/CUSTKEY)) JDFTVAL(*NO) +
                         OPNSCOPE(*JOB) OPTIMIZE(*ALLIO)
            CALL       PGM(PROC_ROWS)
            CLOF       OPNID(ITEM_FACT)
            DLTOVR     FILE(ITEM_JOIN) LVL(*JOB)

            ENDPGM
```

Figure 29. CL example: Using OPNQRYF to allow random access of result set

```
     fitem_fact if   e           k disk    rename(item_fact : item_rcd)
     fitemdspf  cf   E             workstn sfile(SFL:SFLRN)

     d rowsFetched     s              9b 0
     d data            s            132a
     d keyTerritory    s             25a

      /FREE

       runcount = 0;
       dow *in12 = *off;
         exfmt getData;
         if *in12=*on;
          leave;
         endif;
         exsr InzSubFile;

         rowsFetched = 0;
         setll keyTerritory item_join;
         dow not %eof;
           read item_join;
           if %eof
                 or keyTerritory <> territory;
             leave;
           endif;
           rowsFetched = rowsFetched + 1;
           exsr loadSubFile;
         enddo;
       enddo;


       *inlr=*on;
       return;
      /END-FREE
```

Figure 30. Code snippet of RPG program PROC_ROWS: Performs random access of result set

In this example, OPNQRYF joins two tables for rows that have a SALESREP value of *SalesPerson#00009* and calls the HLL program PROC_ROWS for further processing. PROC_ROWS presents a screen that allows the user to specify a territory to further filter the rows in the result set before loading them into the subfile. The key point is that the random access(in step two) only works against a subset of the table – those rows in the result set selected in step 1 (not the entire table).

Although SQL does not have the direct ability to randomly access the result set, this can be simulated by adding the matching rows in step 1 to a temporary table, thus, creating an index against that temporary

table, and allowing further searching against the temp table in step 2. Of course, this implementation falls apart if you want to update the rows in the permanent tables or need a sensitive cursor.

There are more efficient and robust ways to accomplish the same thing in SQL. Rather than implement a two-step process, you can use one SQL statement. In the case of the example shown in Figure 30, one SQL statement performs local selection for both SALESREP and TERRITORY (as well, it handles all joining and ordering). This SQL implementation is shown in Figure 31 (again, the CL program is no longer needed because the embedded SQL handles the ODP processing).

```
     d join_data     e ds                  extname(item_join)
     d rowsFetched     s              9b 0
     d printData       s             80a
     d keyTerritory    s             25a
     d keySalesRep     s             25a
     d runCount        s              9b 0

      /FREE
       dow *in12 = *off;
         exfmt getData;
         if *in12=*on;
           leave;
         endif;

         EXEC SQL
            DECLARE c1 CURSOR FOR
            SELECT territory, salesrep, customer, revenue, quantity
            FROM item_fact A
            INNER JOIN cust_dim B
            ON A.custkey = B.custkey
            WHERE salesrep = :keySalesRep
              AND territory = :keyTerritory;
         rowsFetched = 0;
         EXEC SQL
            OPEN c1;
         dow SQLSTAT <> '02000';
           //*** Fetch matching record
            EXEC SQL
                FETCH C1 INTO :join_data;
            if SQLSTAT = '02000';
             leave;
            endif;
           rowsFetched = rowsFetched + 1;
         enddo;
         EXEC SQL
            CLOSE c1;
       enddo;
       *inlr=*on;
       return;
      /END-FREE
```

Figure 31..Code snippet of RPG program PROC_ROWS (one SQL statement for selection, joining and ordering)

From a performance perspective, you might be skeptical about how this implementation compares with the OPNQRYF method. After all, it does perform local selection of TERRITORY against the entire table, instead of against a subset of the table (those with SALESREP = '00009'). As such, the results shown in Table 75 might be somewhat surprising:

		Average elapsed time (seconds)		
Rows in table	Rows accessed randomly	OPNQRYF	SQL	Commentary on results
6 001 215	720 484	84 883	9800	SQL was more than 8.6 times faster.

Table 75. Testcase results for random access

Appendix D: Resources

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

- IBM System i Information Center
  http://publib.boulder.ibm.com/iseries/
- i5/OS on IBM PartnerWorld®
  ibm.com/partnerworld/i5os
- IBM Publications Center
  www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi?CTY=US
- DB2 for i5/OS online manuals
  ibm.com/iseries/db2/books.html
- IBM Publications Center
  www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi?CTY=US
- IBM Redbooks
  ibm.com/redbooks
    - *Preparing for and Tuning the SQL Query Engine on DB2 for i5/OS*
      www.redbooks.ibm.com/abstracts/sg246598.html?Open
- DB2 Information Center
  http://publib.boulder.ibm.com/infocenter/db2help/
- Online educational course: SQL Performance Basics
  ibm.com/servers/enable/site/education/ibo/record.html?4fa6
- The new SQL query engine (SQE)
  ibm.com/iseries/db2/sqe.html
- White paper: *Creating and using materialized query tables (MQT) in IBM DB2 for i5/OS*
  ibm.com/servers/enable/site/education/abstracts/438a_abs.html
- Article:*i5/OS V5R4 SQL Packs a Punch* (details on recursive CTEs)
  ibm.com/servers/eserver/iseries/db2/pdf/rcte_olap.pdf
- Article: *Accessing Data through SQL views*
  www.systeminetwork.com/artarchive/21029/Accessing_Data_Using_SQL_Views.html
- Article *Ending those Decimal Data Error Blues*
  www.ibmsystemsmag.com/i5/july03/enewsletterexclusive/13470p1.aspx

Appendix E: About the author

Gene Cobb is a DB2 for i5/OS technology specialist within the IBM ISV Enablement for System i organization. He has worked on IBM midrange systems since 1988, with 10 years in the IBM System i Lab Services group – formerly the Client Technology Center (CTC) in Rochester, Minnesota. When he was with the CTC, he assisted IBM clients with application design and development using RPG, DB2 for i5/OS, IBM CallPath/400 and IBM Lotus® Domino®. His current responsibilities include providing consulting services to System i developers, with special emphasis in application and database modernization.

Acknowledgements

Thanks to the following people who reviewed and contributed to this paper:

- Dan Cruikshank
- Kent Milligan
- Michael Cain

Trademarks and special notices

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

AS/400, DB2, Domino, i5/OS, IBM, the IBM logo, Lotus, OS/400, PartnerWorld, Redbooks, System/38 and System i are trademarks of International Business Machines Corporation in the United States, other countries, or both.

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

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

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.