

Extended Dynamic SQL for the masses

A C++ interface and interactive tool for high frequency / low latency database access

Eric Kass
Robert Waury

April 15, 2015

Extended Dynamic SQL is a unique feature of IBM® DB2® for i that can facilitate well performing SQL applications, but the associated QSQPRCED programming interface is of low level and is complex to use. This article introduces a set of C++ classes that can significantly aid in the development of applications designed for high-frequency and low-latency SQL processing. This article also presents a lightweight SQL query tool using the Extended Dynamic SQL interface.

Introduction

Extended Dynamic SQL is a unique feature of IBM DB2 for i that offers applications the ability to perform dynamic SQL while rivaling the performance typically attributed to embedded SQL. The performance advantage of embedded SQL originates from DB2 being able to persist and reuse *access plans*; namely, instructions the engine uses to satisfy SQL queries; or in other words, the *paths* to the data (see [Figure 1](#)). Constructing *access plans* is a time-consuming process for the database engine; but once available, simple database queries can often be performed in tens of microseconds. Applications using Extended Dynamic SQL prepare SQL statements into IBM i SQL package objects. Applications invoke SQL queries indirectly by identifying SQL statements already prepared in a package. Access plans of dynamically prepared statements stored in SQL packages persist even after restarting the IBM i system using *initial program load* (IPL). Although persistent, access plans stored in SQL packages might be automatically updated by DB2 Query Optimizer if related tables or indexes change considerably.

The goal of this article is twofold. The article introduces a set of lightweight object oriented C++ classes for the IBM Extended Dynamic SQL application programming interface (API), QSQPRCED. The C++ classes are conceptually similar to those offered by a *call level interface* (CLI) such as ODBC or JDBC. The second goal of the article is to present *QueryTool*, a small but useful command line based interactive SQL tool that is based on PRCEDpp. The source for both PRCEDpp and *QueryTool* are provided under an MIT open source license on *SourceForge*. *QueryTool* can be compiled to use the native QSQPRCED interface for local database access, or

compiled to use IBM XDA driver for both local and remote database access. The IBM XDA driver is available for IBM i, Microsoft® Windows®, and Linux®.

Extended Dynamic SQL with PRCEDpp

The procedure for issuing SQL queries and running SQL statements with Extended Dynamic is not significantly different than with industry-standard interfaces such as ODBC and JDBC. In each case, the method for reading data from the database is: prepare an SQL statement, open a cursor for the statement, fetch data row-by-row or in blocks, close the cursor. The difference with Extended Dynamic SQL is the low-level control offered by the QSQPRCED API such as prepared statement storage management, cursor lifetime determination, and specific optimization options.

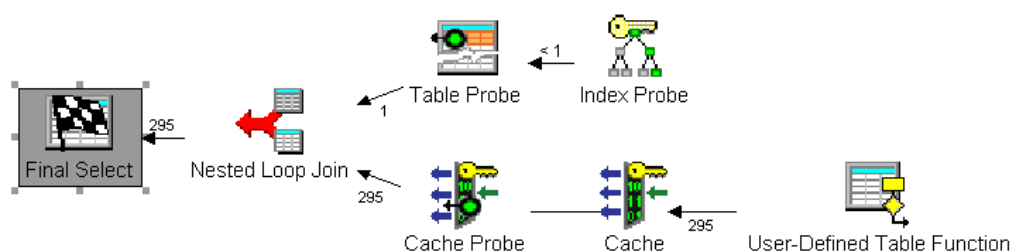
Notwithstanding the features and control offered by the QSQPRCED API, programming towards the API is confusing, time consuming, and error prone. PRCEDpp is a set of classes that handle the complexity of the QSQPRCED API though retain the control features and performance benefits of the API itself. The PRCEDpp API is described in detail in the subsequent topics.

High-frequency / Low-latency processing

Two features of QSQPRCED make it apt for high frequency / low latency SQL processing; the first is manually managed and prepared statements; the second is low-level cursor control. PRCEDpp exposes interfaces to use these benefits. Optimal performance is achieved when both prepared SQL statements and cursors can be reused.

Before an SQL statement is run by the database, the SQL statement must be prepared by the application. The *optimizer* component of the database engine constructs an access plan, a *program* of sorts, which the database engine *runs* to satisfy the query. An example for a simple query is shown in [Figure 1](#).

Figure 1: Access plan using Operations-Navigator Visual Explain



```
SELECT "TABLE_PARTITION" , "DATA_SIZE" / 1024 , "NUMBER_ROWS" ,
"NUMBER_DELETED_ROWS" , "SYSTEM_TABLE_SCHEMA" ,
"SYSTEM_TABLE_NAME" , "SYSTEM_TABLE_MEMBER" FROM
QSYS2 / SYSPARTITIONSTAT WHERE "TABLE_SCHEMA" = ? AND "TABLE_NAME" = ?
```

The optimizer considers time and resource requirements of various possible plans that would satisfy the query and stores the most likely optimal plan in an SQL package (the operating system command, PRTSQLINF, prints the contents of a package). Building an optimal access plan for large queries is a computationally costly process; therefore, the overall goal for well-performing applications is to reuse access plans. Each SQL statement prepared in a package can be uniquely

referenced by the name of the package and the statement name; PRCEDpp refers to this as a *Statement-In-Package-ID* (SIPIID). QSQPRCED can find an existing statement in a given package, but performance of the interface is best when the application knows the precise, fully qualified, identifying SIPIID of the statement.

A cursor is an internal database entity that is referenced by QSQPRCED to read data; row-by-row or block-by-block. An application first opens a cursor towards a specific SQL statement and then fetches data using the cursor as a reference. An application can be designed to reuse cursors in order to re-issue queries towards the same SQL statement either with different input predicate values or at a different point in time. For optimal cursor reuse, the aim is to identify cursors by name and perform *pseudo close cursor* instead of *hard close cursor* when the queries are complete. Yet even greater performance can be achieved when cursors are not closed at all between commit cycles; we term this *deferred close cursor*; meaning, the application will not be performing a subsequent fetch on the cursor but the database sees the cursor as open. This later method is primarily practical for uncommitted and committed isolation levels; but is not practical for higher isolation levels in which open cursors retain record locks that with this method might cause contention. When issuing subsequent queries against a *deferred close* cursor, an application must specify the *reopen* flag together with the *open* action to avoid a *cursor already opened* (-502) error. Cursors not opened with the `WITH HOLD` option are *pseudo* closed implicitly on commit or rollback. *QueryTool* only performs *pseudo closes* or *deferred closes* (no close) depending on the *reopen* option and whether *QueryTool* expects to be able to reuse a cursor.

For organizational simplicity, it is feasible to name a cursor the same as a particular SQL statement, if only one cursor is to be opened against that statement at a time.

QueryTool

QueryTool is a simple command line based interactive SQL utility that is internally based on PRCEDpp classes. The utility itself is about 1000 lines of code; yet it demonstrates the optimal use of the QSQPRCED API. *QueryTool* can issue queries using Data Query Language (DQL), run insert and update statements using Data Manipulation Language (DML), and issue database structure modification SQL statements using Data Definition Language (DDL).

QueryTool accepts SQL statements as input interactively or as scripts. *QueryTool* by way of PRCEDpp uses the features of QSQPRCED in a fashion that is tuned for high frequency / low latency database access. In addition to being valuable itself as a stand alone utility, *QueryTool* is beneficial as a template for other programs. An interactive *QueryTool* session-in-action example is shown in [Figure 2](#).

Originally designed to test and exercise the QSQPRCED interface, *QueryTool* uses many of the interfaces' features and supports a wealth of data types. Some examples of exercised features conclude this paragraph. The tool fetches blocks of data directly into c-language buffers either with type conversion or directly-mapped; wherein the database copies the memory region of the table into the c-buffer regardless of the field type and boundaries. Large binary objects are fetched piece-by-piece, which is characterized as streaming. When command line options specify; data inserted (using the `INSERT` command) or merged (using the `MERGE` command) into the database

is also written in blocks in some cases. For the later to occur; *QueryTool* rewrites SQL statements converting them to normalized SQL statements containing parameter markers in place of literals, and then runs these prepared normalized statements passing in blocks of input data.

QueryTool performs the optimal cursor procedure of *optimistic open*; whereby an *open-cursor* using an SQL statement name is attempted first, unknowing whether the statement had been previously *prepared*. An *unknown statement* failure at *open-cursor* is an indication to *QueryTool* to *prepare* the statement and attempt the *open* again. Application developers are welcome to use *QueryTool* as a template for this method.

QueryTool also uses the database's `describe` feature to retrieve the output variable format definitions of a particular query statement. This feature is well suited to programs that issue ad-hoc queries.

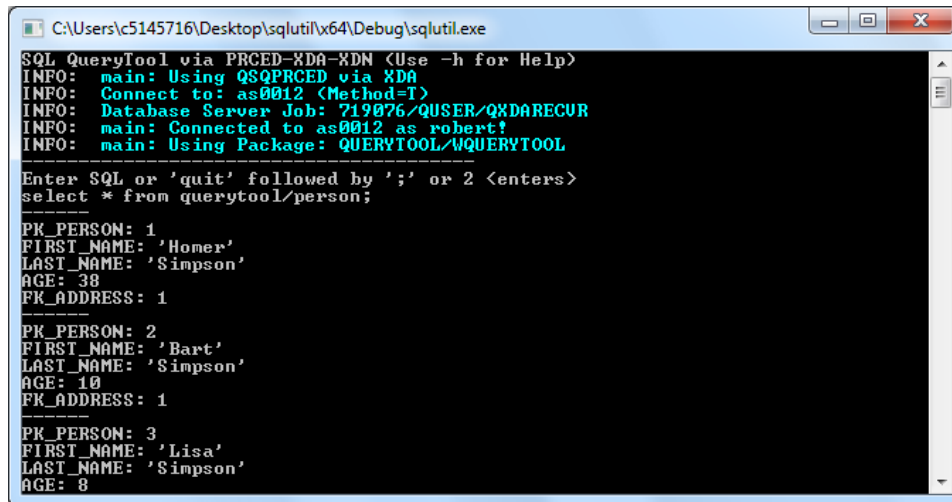
QueryTool operation

The *QueryTool* accepts SQL statements to be interactively issued towards a DB2 for i database using the command line or a script file. If the tool runs on an IBM i host with a local database, it uses the QSQPRCED interface directly. If the destination database resides on a remote system, *QueryTool* can be compiled to use the XDA or XDN database drivers. Because the tool uses Extended Dynamic SQL in an optimal manner, it will either prepare a statement into an SQL package before running or just issue the statement from the SQL package if it had already been prepared.

QueryTool reads console input until the SQL statement is terminated either with a semicolon or two successive carriage-returns. Based on the type of SQL statement, *QueryTool* performs the associated actions in the following order: Open, prepare, fetch, and close or prepare and execute. SQL statements are sent to the database unaltered with the exception of INSERT and MERGE statements when the option to normalize statements is enabled.

Figure 2 shows an example of *QueryTool* operating interactively. In this instance, *QueryTool* is running as a windows program and connecting to a remote IBM i database. On the ninth line of this example, the user has entered an SQL statement, `SELECT * FROM QUERYTOOL/PERSON`. Following the statement, *QueryTool* has printed the contents of the PERSON database preceded by the name of each column. *QueryTool* had acquired the column names by issuing a `describe` action before fetching the data from the database.

Figure 2: QueryTool running on Microsoft Windows through XDA



```

C:\Users\c5145716\Desktop\sqlutil\x64\Debug\sqlutil.exe
SQL QueryTool via PRCED-XDA-XDM <Use -h for Help>
INFO: main: Using OSQPRCED via XDA
INFO: Connect to: as0012 <Method=T>
INFO: Database Server Job: 719076/QUSER/QXDARECUR
INFO: main: Connected to as0012 as robert?
INFO: main: Using Package: QUERYTOOL/WQUERYTOOL
-----
Enter SQL or 'quit' followed by ';' or 2 <enters>
select * from querytool/person;
-----
PK_PERSON: 1
FIRST_NAME: 'Homer'
LAST_NAME: 'Simpson'
AGE: 38
FK_ADDRESS: 1
-----
PK_PERSON: 2
FIRST_NAME: 'Bart'
LAST_NAME: 'Simpson'
AGE: 10
FK_ADDRESS: 1
-----
PK_PERSON: 3
FIRST_NAME: 'Lisa'
LAST_NAME: 'Simpson'
AGE: 8

```

QueryTool displays the results of the SELECT queries for a maximum number of lines. For DML and DDL statements, *QueryTool* returns the status of these operations.

The error handling of *QueryTool* is simple and straight-forward but production applications would typically be more sophisticated. A database error results in the associated SQLCODE and exception data being displayed. *QueryTool* does not abort processing after an error; it continues to accept input or reads from a file for the next SQL statement or command.

Parameter marker replacement

A feature of *QueryTool* designed to optimize statement reuse for `INSERT` and `MERGE` statements is parameter marker replacement. The principle is to normalize SQL statements by removing literals and replacing these with variables: Two SQL statements which would otherwise be identical except for their embedded literals have the same normalized form. *QueryTool* prepares, runs, and reuses the normalized form of the SQL statement; passing literal values in host variables stored in application buffers.

Multi-row blocked operations using INSERT and MERGE

Single SQL `INSERT` and `MERGE` statements can affect multiple rows in a block operation, reducing the number of database round trips otherwise required for multiple singleton operations. To perform a block operation, a `VALUE` clause with a syntax shown in Example 1 is used to specify arrays of literals.

Example 1: Blocked INSERT syntax

```

INSERT INTO PERSON (pk_person,
                    first_name,
                    last_name,
                    age)
VALUES (1, 'Homer', 'Simpson', 38),
       (2, 'Marge', 'Simpson', 34),
       (3, 'Lisa', 'Simpson', 8)

```

The sequence of `VALUE` rows are replaced by the one containing the parameter markers in an equivalent normalized form of the statement (see Example 2) *QueryTool* initiates blocked execution for these statements passing values in a data array (host variables) and setting the blocked row count to a value greater than zero.

Example 2: Normalized rewritten INSERT statement

```
INSERT INTO PERSON (pk_person,
                    first_name,
                    last_name,
                    age)
  ? ROWS VALUES (?, ?, ?, ?)
```

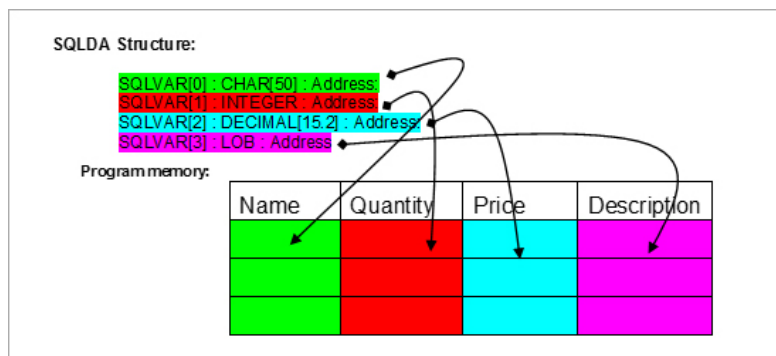
Data descriptor SQLDA

Data is passed between the database and application memory during open, fetch, and execute operations. Information about the application memory location and the application's expected data format resides in a data structure, SQLDA (see Figure 3). The SQLDA data structure has one entry for each variable; output field in the `SELECT` list of fetch, input host variable of open and execute, and of both input and output of `VALUES` statements.

QueryTool uses PRCEDpp to build SQLDAs automatically from the results of `Describe` operations on SQL query statements. A `Describe` operation on an SQL query statement returns metadata in the form of an SQLDA data structure describing the column name, type, and length of the results to be expected from the query.

For normalized `INSERT` and `MERGE` statements, *QueryTool* builds a temporarily associated query statement referencing the same fields as the `INSERT` or `MERGE` statement; and then issues a `Describe` operation on the associated query statement to build an SQLDA data structure. The `SelectBuilder` class in the code performs this transformation. The literals contained in the original `INSERT` and `MERGE` statements are placed in application buffers and are referenced in SQLDA. For multi-row blocked operations, literals are stored in a memory array and each field descriptor in the SQLDA points to the address of its respective first value in the column of the array (see Figure 3).

Figure 3: SQLDA – Host variable descriptor and buffer



SQL package handling

Prepared SQL statements and their associated access plans are stored in SQL package objects on the database host. *QueryTool* creates SQL packages in the QUERYTOOL library or in a

library specified on the command line. All statements are prepared in either of two packages; QUERYTOOL or WQUERYTOOL depending on the character encoding with which *QueryTool* is compiled; the default is dependent on the underlying interface. When running as a Microsoft Windows application, the default character encoding is ASCII and the package is WQUERYTOOL; when running as an IBM i application, the default character encoding is EBCDIC and the package is QUERYTOOL.

QueryTool may also create an additional SELECT package used to retrieve type information for creating SQLDA column description data structures (see Figure 3). Each statement in a package is identified by a statement name. *QueryTool* generates statement names starting with the characters, **ST** followed by a 16 character hexadecimal number derived from the current timestamp. Before a statement is prepared, *QueryTool* first checks whether a statement with the same SQL text had previously been prepared in the package. In the affirmative case, *QueryTool* runs the already prepared statement.

Usage

QueryTool accepts a number of parameters on the command line. The essential parameters specify the database host and logon credentials thereof. The primary configuration options control whether SQL statements are reused, whether data is fetched a row or block at a time, how cursors should be closed and reopened, and whether to replace literals with parameter markers. Secondary options include: indicating a database parameter file, QAAQINI, specifying an SQL package location, setting driver options, and controlling output and trace levels. *QueryTool* can also be called with a '-h' or '/?' parameter for the program to display detailed help.

As *QueryTool* accepts SQL statements on standard input, statements can either be entered interactively or in the form of a script directed as input. The script file is a sequence of SQL statements, each followed by a semicolon and a carriage return or two subsequent carriage returns; this formal syntax is shown in Example 3.

Example 3: SQL script file syntax

```
<script> := <comment> | <statement> | <script>
<comment> := #<string><enter>
<statement> := <string>;<enter>
<statement> := <string><enter><enter>
```

A simple script example is shown in Example 4. The first statement returns the current database time (the SQL syntax for the `SELECT` statement requires a table name to be specified; any existing table name is valid, but the particular one chosen is not relevant).

Example 4: SQL script file example

```
# Example SQL script file for QueryTool.
#
# 1st example comment
select current time from systables;
# 2nd example comment
INSERT INTO Addresses (Name, Address) VALUES ('Kermit', '123 Sesame')
```

The second statement inserts a row into a table named `Addresses`. Depending on the setting of *QueryTool*, the literals `kermit` and `123Sesame` will be converted to parameter markers in the SQL text and passed to the database as host variables, or just left as literals.

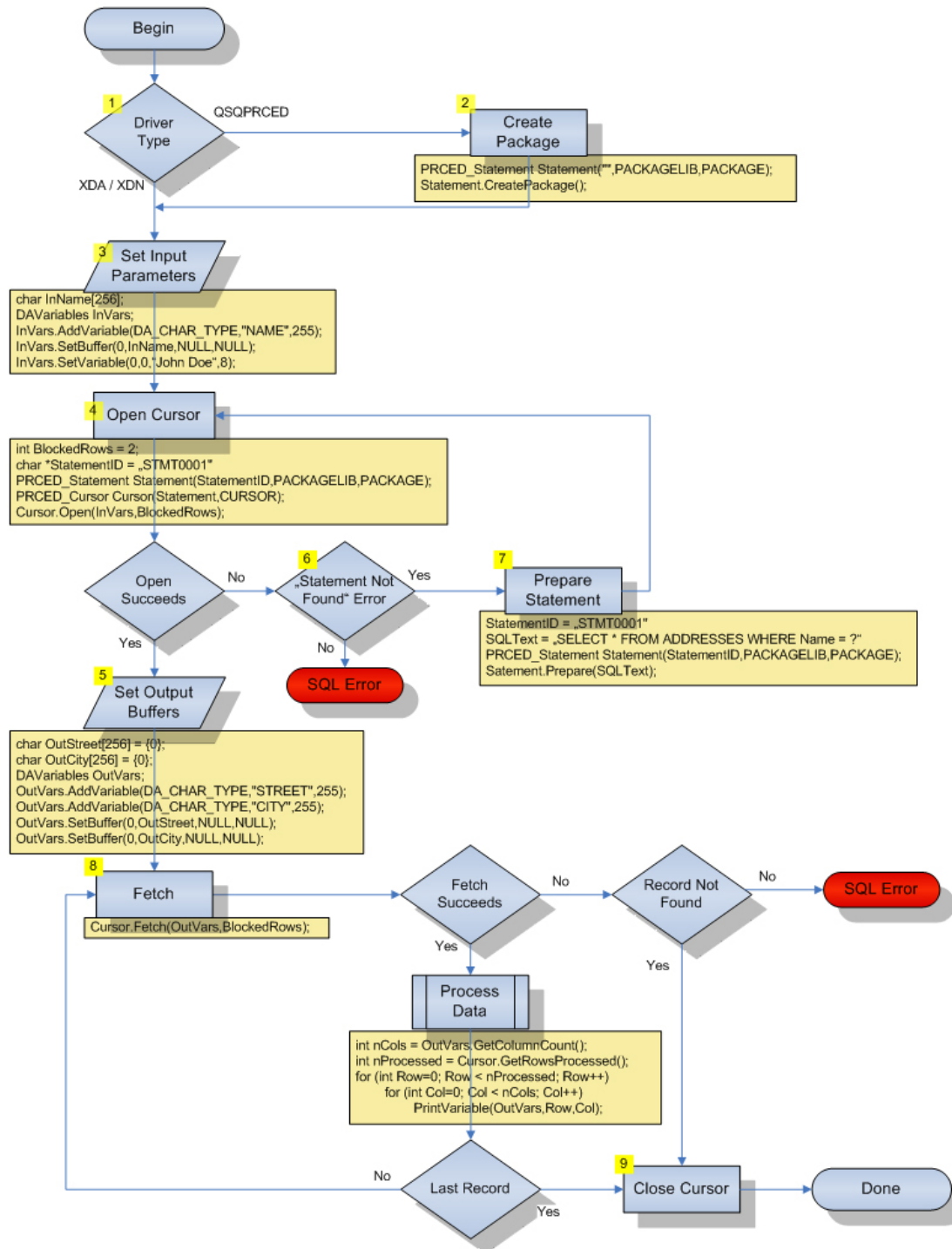
Results and error messages of *QueryTool* are written to standard output and standard error respectively; either or both can be directed to an output file.

PRCEDpp details

The PRCEDpp C++ classes that are used by *QueryTool* can be used as a basis for other Extended Dynamic SQL applications. PRCEDpp can use one of the two underlying interfaces: the native QSQPRCED API for direct access to database schemata residing on the local host, or the remote access capable XDA database driver to access database schemata through QSQPRCED over a network.

PRCEDpp query flowchart

Reading and writing to the database with PRCEDpp should be rather straight forward to those familiar with database API programming, albeit a number of steps beyond the typical are required to perform database access. The steps for reading using PRCEDpp are outlined in the flowchart in Figure 4.

Figure 4: PRCEDpp query flowchart**PRCEDpp QUERY Flow Chart**

1. SQL package – SQL statement and access plan storage:

- When PRCEDpp is using the native QSQPRCED (see step 1 in Figure 4) underlying interface, create a PRCED_Statement instance with only a package name and invoke the `CreatePackage()` method (see step 2 in Figure 4).
 - When PRCEDpp is using XDA, the package would be created automatically.
2. SQL statement - PRCEDpp object identifying a statement in a package:
 - Create a PRCED_Statement class instance with a statement ID and / or SQL text.
 - Using PRCED_Statement, prepare the SQL statement into a package.
 3. Input variables – Description and pointers to data.
 - If the SQL statement contains parameter markers for input variables (for example, `WHERE Field = ?`); create a `DAVariables` instance assigning a `DAVariable` for each parameter marker (see step 3 in Figure 4).
 4. Open cursor – Handle to retrieve data:
 - Prepare – Open
A statement ID referencing an SQL statement in a package is required to open a cursor. If the statement ID is not known; the `Prepare()` method of PRCED_Statement will find an existing statement ID when using XDA. With native QSQPRCED, invoke the `FindStatement()` method first and subsequently a `Prepare()` method, if required.
 - Open – Prepare – Open:
If the statement ID is already known (a constant value assigned by the application, for example), an optimistic approach of attempting to open a cursor (see step 4 in Figure 4) first, and only then prepare [see step 7] in Figure 4) the statement if it is not found corresponding to the conditional shown in the flow chart (see step 6 in Figure 4).
 5. Output variables – Description and pointers to buffers.
 - A `DAVariables` class instance describes the format and memory location of the data to be retrieved. A `DAVariables` class instance can be constructed manually or by using the PRCED_Statement classes' `Describe()` method.
 - For each output field (for example, `SELECT FieldA, FieldB`), assign a memory buffer (see step 5 in Figure 4) to the `DAVariables` class instance. When selecting multiple rows, the memory buffer must be continuous and equal in size to the *number of rows* multiplied the sum of the *lengths of all the fields*.
 6. Fetch data through the cursor – Fill program buffers with data:
 - The `Fetch()` method on the PRCED_Cursor object requests the database to fill the output buffers identified by `DAVariables` with query results.
 7. Close cursor – Release DB resources and release record locks:
 - As described in the [QueryTool](#) section above, a cursor can be *differed*, *pseudo*, or *hard* closed (see step [9] in Figure 4) depending on the performance, management, and locking requirements of the application.

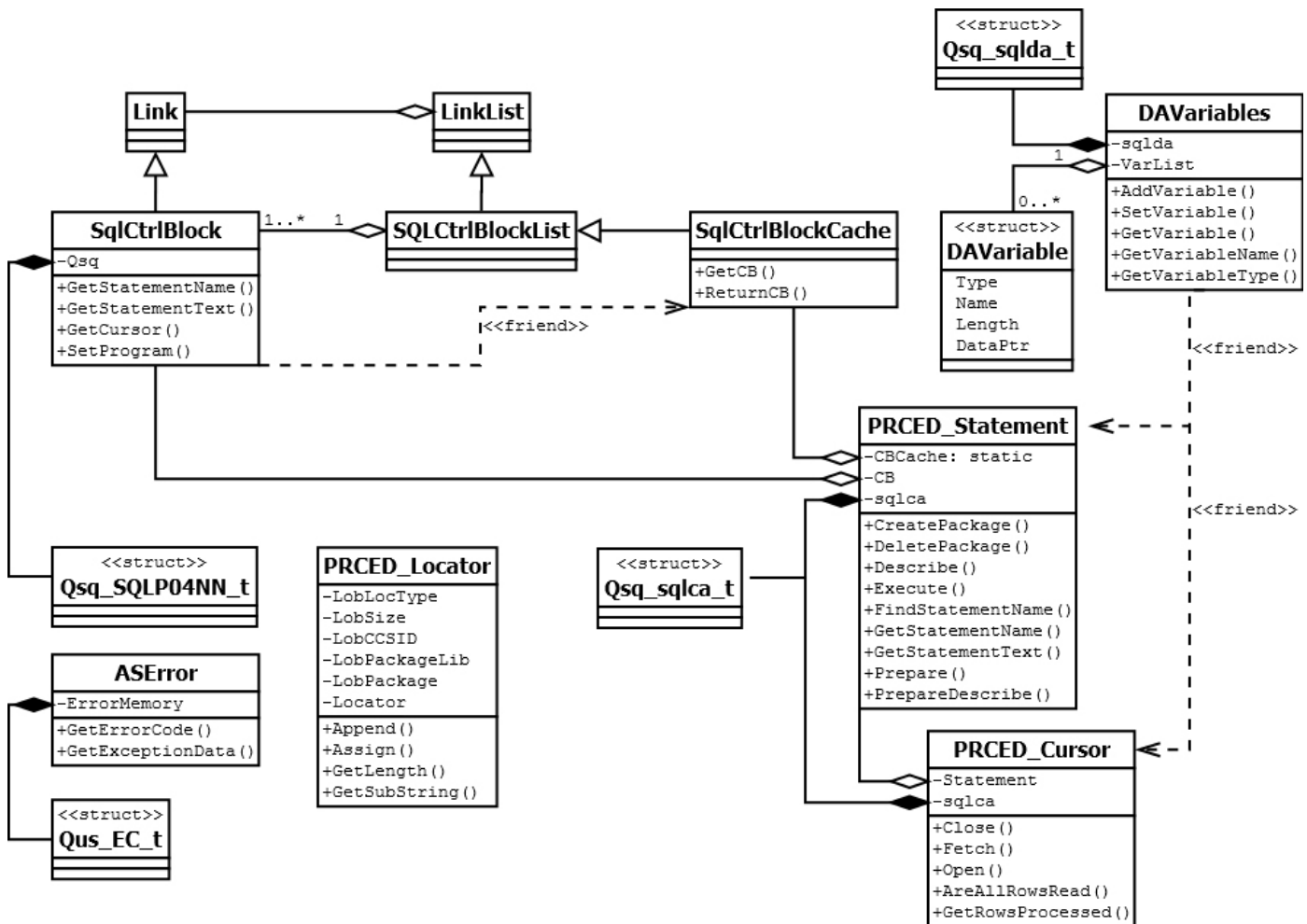
Writing to the database follows the same sequence for statement preparation and assigning input variables. Running the DML (for example INSERT, UPDATE) statements by using the `Execute()` method on PRCED_Statement is without reference to a cursor.

Classes

PRCEDpp publicizes classes to represent SQL statements, cursors for read / write access, large object (LOB) locators for efficient data processing, and I/O variables. Internally, PRCEDpp uses

other classes to represent internal data structures of the QSQPRCED API; namely the SQLP0400/SQLP0410 function template and communication and error structures. This abstraction layer affords the tool simplified management of statements, SQL packages, parameters, and error messages.

Figure 5: PRCEDpp class diagram



The class diagram in Figure 5 depicts QSQPRCED data structures encapsulated by PRCEDpp. The most important classes for application development are the statement, cursor, locator, and variables, namely: PRCED Statement, PRCED Cursor, PRCED Locator, and DAVariables.

The classes would be used by a typical application to perform a query in the following sequence similar in flow to. The application would begin by preparing an SQL statement for a query by invoking the `Prepare()` method on a statement object's instance. To begin to read data, an application would firstly instantiate a cursor based on the statement object, then would invoke the `open()` method on the cursor, optionally passing in a set of input variables. The result of the query would be materialized when the application would repeatedly invoke the `Fetch()` method of the cursor, typically passing in a set of output variables. In the case that a locator to a large object (LOB) would have been fetched instead of the data itself, a locator can be used to iterate over the LOB contents.

Internally, PRCEDpp manages the other classes that encapsulate structures used to interact natively with the database engine. The C++ `SqlCtrlBlock` class encapsulates the database request data structure `Qsq_SQLP04NN_t` (see [Table 1](#)) and provides `set` methods for the most significant fields. Internal memory management of the `SqlCtrlBlock` class that encapsulates the particular **SQLP0410** variant of the data structure is notably complex because the data structure contains a number of varying length fields and maintaining this involves tedious calculation of offsets and lengths. Therefore, `SqlCtrlBlock` encapsulation is particularly beneficial for this variant.

Table 1: Major fields of an SQLP400 function template

SQLP0410	
Function	
Package Name	
Statement Name	
Cursor Name	
Number of Rows	
... Options ...	
SQL Statement Text...	
... Client Information ...	

Results of the QSQPRCED requests are returned in a communication data structure, `Qsq_sqlca_t`. The most important fields are `SQLCODE` and `sqlerrd`. The `SQLCODE` has different meanings depending on the operation being performed. Usually, an `SQLCODE` equal to zero is an indication of success; but, for example, an `SQLCODE` of *100* as a result of a fetch operation indicates that no further record was found. Positive codes represent notifications and negative codes represent errors. The other most important field, `sqlerrd`, is used for indicating that *the last record is included* in the result set (`sqlerrd[4]`) and *the number of records affected* by a modify operation (`sqlerrd[2]`).

The `DAVariables` encapsulate the `Qsq_sqllda_t` structure that define input and output variable formats and memory locations. Critical errors of the underlying database engine that are not semantically SQL errors are returned in `qus_EC_t` that PRCEDpp deciphers and throws as an `ASError` object instance.

Programming examples

The following two code sequences compare performing an `INSERT` operation in the first sequence with the native QSQPRCED interface and in the second sequence with PRCEDpp. The main goal of the comparison is to reveal the programming ease and code simplification achieved using PRCEDpp.

[Listing 1](#) illustrates a minimum code sequence that is sufficient to prepare and run an SQL statement with QSQPRCED. The code inserts data into a database table by running an `INSERT` statement with literal values. An application would likely use input host variables as opposed to

literals as the former are more versatile. The code example assumes that an SQL package had already been created, though package creation code itself looks very similar. The code does not evaluate a return code nor does it handle any errors. [Download the code in Listing 1.](#)

Listing 1. QSQPRCED example

```

1. void PrepareExecute()
2. {
3.     Qsq_sqlca_t sqlca = {0};
4.     Qsq_sqlda_t sqlda = {0};
5.     sqlda->sqldabc = sizeof(sqlda);
6.     char ErrorMemory[ERROR_MEM_SIZE];
7.     Qus_EC_t* err = (Qus_EC_t*) ErrorMemory;
8.     err->Bytes_Provided = sizeof(ErrorMemory);
9.     err->Bytes_Available = 0;
10.    err->Reserved = 0;
11.    Qsq_SQLP0400_t qsq = {0};
12.    char qsqPlusStmt[1000];
13.    qsq.Function = '2'; /* Prepare */
14.    memcpy(qsq.SQL_Package_Name, "OS400PKG ", 10);
15.    memcpy(qsq.Library_Name, "QUERYTOOL ", 10);
16.    memcpy(qsq.Main_Pgm, "QCMD ", 10);
17.    memcpy(qsq.Main_Lib, "QSYS ", 10);
18.    memcpy(qsq.Statement_Name, "ST0000000000001234", 18);
19.    qsq.Open_Options = 0x90;
20.    qsq.Reuse_DA = 'N';
21.    qsq.Name_Check = 'Y';
22.    const char *SQL = "INSERT INTO QUERYTOOL/PERSON "
23.                      "(pk_person, first_name, last_name, age) "
24.                      "VALUES (4, 'Maggie', 'Simpson', 1)";
25.    qsq.Statement_Length = strlen(SQL);
26.    memcpy(qsqPlusStmt, (char *)&qsq, sizeof(Qsq_SQLP0400_t));
27.    char *stmttxt = qsqPlusStmt +
28.                  (offsetof(Qsq_SQLP0400_t, Statement_Length) *
29.                   sizeof(qsq.Statement_Length));
30.    memcpy(stmttxt, SQL, strlen(SQL));
31.    QSQPRCED(&sqlca, &sqlda, "SQLP0400", (void*)qsqPlusStmt, err);
32.    qsq.Function = '3'; /* Execute */
33.    QSQPRCED(&sqlca, sqlda, "SQLP0400", (void*)qsqPlusStmt, err);
34. }

```

The preamble (lines 11 through 21) fills in the detailed SQLP0400 data structure containing the package name, SQL statement, SQL statement name, various options, and the name of the program on the runtime call stack (QSYS/QCMD) that the commitment control definition will be bound. The SQL statement itself is copied into the SQLP0410 structure in code lines 22 through 26.

After the data structures are initialized, the actual call to QSQPRCED is trivial (line 29).

[Listing 2](#) shows how the same `INSERT` statement can be implemented in `PRCEDpp`. This application code is significantly shorter because the `PRCEDpp` classes manage all the underlying data structures required by `QSQPRCED`. [Download the code in Listing 2.](#)

Listing 2. PRCEDpp example

```
1. void PrepareExecute()
2. {
3.     PRCED_Statement Statement("ST0000000000001234",
                                "QUERYTOOL", "OS400PKG");
4.     char* StatementText = "INSERT INTO QUERYTOOL/PERSON "
                            "(pk_person, first_name, last_name, age) "
                            "VALUES (4, 'Maggie', 'Simpson', 1)";
5.     Statement.Prepare(StatementText);
6.     DAVariables NullDA;
7.     Statement.Execute(NullDA, 0);
8. }
```

As the data being inserted are encoded as literals directly in the statement text, there is no need to assign host variables. Even otherwise, this can be accomplished on the `NullDA` object in line 6 of the example; pragmatically also renaming the DA to something other than `NULL`.

As in the `QSQRCEd` example, the `PRCEDpp` example also neither evaluates nor handles errors. The return value of the `Execute()` command in line 7 is the associated SQL code resulting from the underlying database call. A runtime failure of the `QSQRCEd` would result in an `AS4Error` object instance (as shown in Figure 5) being thrown.

The `QueryTool.cpp` source file at *SourceForge* is a significantly more complete example and demonstrates a query using host variables.

Summary

`PRCEDpp` is an object oriented database programming interface that is a viable alternative for applications aiming to perform high-frequency and low-latency SQL queries. The advantage over the underlying low-level `QSQRCEd` interface is the code simplification achieved by using `PRCEDpp` instead. `PRCEDpp` retains the performance and low-level control that `QSQRCEd` provides.

The *QueryTool* application was written primarily as a program to exercise the `QSQRCEd` API within both IBM and SAP, but the code also well demonstrates `PRCEDpp`. *QueryTool* is also practical in itself either as a stand-alone database interactive tool or a template for other dynamic SQL applications.

Both `PRCEDpp` and *QueryTool* are available on *SourceForge* and can be used according to the minor restrictions dictated by the MIT license under which the source code is published.

Resources

- For more information about Extended Dynamic SQL, refer to the IBM [QSQRCEd documentation](#).
- Refer to the [DB2 for i SQL reference](#) topic.
- The XDA client is part of IBM i Access for Windows and Linux, but the latest drivers can be obtained for [Windows](#).

Refer to the source code hosted on [Source Forge](#).

© Copyright IBM Corporation 2015

(www.ibm.com/legal/copytrade.shtml)

[Trademarks](#)

(www.ibm.com/developerworks/ibm/trademarks/)