DB2 for OS/390
Version 5

Call Level Interface Guide and Reference

SC26-8959-02

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page vii.

First Edition (June 1997)

This edition applies to Version 5 of IBM DATABASE 2 Server for OS/390 (DB2 for OS/390), 5655-DB2, and to any subsequent releases until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

This softcopy version is based on the printed version of the book, and includes the changes indicated in the printed version by vertical bars. Additional changes made to this softcopy version of the manual since the hardcopy manual was published are indicated by the hash (#) symbol in the left-hand margin.

Contents

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Licensees of this program who wish to have information about it for the purpose of enabling (1) the exchange of information between independently created programs and other programs (including this one) and (2) the mutual use of the information that has been exchanged, should contact:

IBM Corporation
IBM Corporation
J74/G4
555 Bailey Avenue
P.O. Box 49023
San Jose, CA 95161-9023

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Programming Interface Information

This book is intended to help the customer write applications that use DB2 Call Level Interface to access IBM DB2 for OS/390 servers. This book documents General-use Programming Interface and Associated Guidance Information provided by DATABASE 2 for OS/390 (DB2 for OS/390).

General-use programming interfaces allow the customer to write programs that obtain the services of DB2 for OS/390.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX
BookManager
C++/MVS
CICS
CICS/ESA
CICS/MVS
DATABASE 2
DB2
DB2/2
DB2/6000
DFSMS
DFSMShsm
Distributed Relational Database Architecture
DRDA
IBM
IMS
IMS/ESA
Language Environment
MVS
MVS/ESA
OS/2
OS/390
OS/400
Parallel Sysplex
QMF
RACF
SQL/DS
VTAM

Throughout the library, the DB2 licensed program and a particular DB2 subsystem are each referred to as “DB2.” In each case, the context makes the meaning clear. The term *MVS* is used to represent the MVS/Enterprise Systems Architecture (MVS/ESA). *CICS* is used to represent CICS/MVS and CICS/ESA; *IMS* is used to represent IMS/ESA; *C* and *C language* are used to represent the C/370 and C/C++ for MVS/ESA programming language.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Microsoft, Windows, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

Chapter 1. Introduction to this Book and the DB2 Library

About This Book

This book provides the information necessary to write applications using DB2 Call Level Interface to access IBM DATABASE 2 servers as well as any database that supports DRDA level 1 or DRDA level 2 protocols. This book should also be used as a supplement when writing portable ODBC applications that can be executed in a native DB2 for OS/390 environment using the DB2 Call Level Interface.

Who Should Use This Book

DB2 application programmers with a knowledge of SQL and the C programming language.

ODBC application programmers with a knowledge of SQL and the C programming language.

How this Book is Structured

This book is divided into the following chapters:

- “Chapter 1. Introduction to this Book and the DB2 Library,” identifies the purpose, the audience, and the use of this book.
- “Chapter 2. Introduction to CLI” on page 15, introduces DB2 CLI and discusses the background of the interface and its relation to embedded SQL and Microsoft ODBC.
- “Chapter 3. Writing a DB2 CLI Application” on page 23, provides an overview of a typical DB2 CLI application. This chapter discusses the basic tasks or steps within a simple DB2 CLI application. General concepts are introduced as well as the basic functions and the interaction between them.
- “Chapter 4. Configuring CLI and Running Sample Applications” on page 49, contains information for setting up the necessary environment to compile and run DB2 CLI applications. Sample applications are provided in order to verify your environment.
- “Chapter 5. Functions” on page 73, is a reference for the functions that make up DB2 CLI.
- “Chapter 6. Using Advanced Features” on page 341, provides an overview of more advanced tasks and the functions used to perform them.
- “Chapter 7. Problem Diagnosis” on page 381, explains how to work with traces and debug applications.

The appendixes contain the following information:

- Appendix A, “Programming Hints and Tips” on page 393, provides some common hints and tips for improving performance and/or portability of DB2 CLI applications.

- Appendix B, “DB2 CLI and ODBC” on page 399, discusses the differences between ODBC and DB2 CLI.
- Appendix C, “Extended Scalar Functions” on page 405, describes the scalar functions that can be accessed as DB2 functions, or using ODBC vendor escape clauses.
- Appendix D, “ Appendix D. SQLSTATE Cross Reference” on page 409, contains an SQLSTATE table that lists the functions that may generate each SQLSTATE. (Each function description in “Chapter 5. Functions” on page 73 lists the possible SQLSTATEs for each function.)
- Appendix E, “Data Conversion” on page 419, contains information about SQL and C data types, and conversion between them.
- Appendix F, “Example Code” on page 443, provides an extensive stored procedure example.

How to Read the Syntax Diagrams

The following rules apply to the syntax diagrams used in this book:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►── symbol indicates the beginning of a statement.

  The ──► symbol indicates that the statement syntax is continued on the next line.

  The ►── symbol indicates that a statement is continued from the previous line.

  The ──►◄ symbol indicates the end of a statement.

  Diagrams of syntactical units other than complete statements start with the ►── symbol and end with the ──► symbol.

- Required items appear on the horizontal line (the main path).

```
►►──required_item────────────────────────────────────────────►◄
```

- Optional items appear below the main path.

```
►►──required_item──┬───────────────┬─────────────────────────►◄
                   └─optional_item─┘
```

  If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

```
                   ┌─optional_item─┐
►►──required_item──┴───────────────┴─────────────────────────►◄
```

- If you can choose from two or more items, they appear vertically, in a stack.

  If you *must* choose one of the items, one item of the stack appears on the main path.

```
►►──required_item──┬─required_choice1─┬──────────────────────►◄
                   └─required_choice2─┘
```

  If choosing one of the items is optional, the entire stack appears below the main path.

```
►►──required_item──┬──────────────────┬──────────────────────────────►◄
                   ├─optional_choice1─┤
                   └─optional_choice2─┘
```

If one of the items is the default, it appears above the main path and the remaining choices are shown below.

```
                   ┌─default_choice──┐
►►──required_item──┼─────────────────┼───────────────────────────────►◄
                   ├─optional_choice─┤
                   └─optional_choice─┘
```

- An arrow returning to the left, above the main line, indicates an item that can be repeated.

```
                   ┌─────────────────┐
►►──required_item──▼─repeatable_item─┴───────────────────────────────►◄
```

If the repeat arrow contains a comma, you must separate repeated items with a comma.

```
                   ┌─,───────────────┐
►►──required_item──▼─repeatable_item─┴───────────────────────────────►◄
```

A repeat arrow above a stack indicates that you can repeat the items in the stack.

- Keywords appear in uppercase (for example, FROM). They must be spelled exactly as shown. Variables appear in all lowercase letters (for example, *column-name*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

How to Use the DB2 Library

Titles of books in the library begin with DB2 for OS/390 Version 5. However, references from one book in the library to another are shortened and do not include the product name, version, and release. Instead, they point directly to the section that holds the information. For a complete list of books in the library, and the sections in each book, see the bibliography at the back of this book.

Throughout the library, the DB2 for OS/390 licensed program and a particular DB2 for MVS/ESA subsystem are each referred to as “DB2.” In each case, the context makes the meaning clear.

The most rewarding task associated with a database management system is asking questions of it and getting answers, the task called *end use*. Other tasks are also necessary—defining the parameters of the system, putting the data in place, and so on. The tasks associated with DB2 are grouped into the following major categories (but supplemental information relating to all of the below tasks for new releases of DB2 can be found in *Release Guide*):

Installation: If you are involved with DB2 only to install the system, *Installation Guide* might be all you need.

If you will be using data sharing then you also need *Data Sharing: Planning and Administration*, which describes installation considerations for data sharing.

End use: End users issue SQL statements to retrieve data. They can also insert, update, or delete data, with SQL statements. They might need an introduction to SQL, detailed instructions for using SPUFI, and an alphabetized reference to the types of SQL statements. This information is found in *Application Programming and SQL Guide* and *SQL Reference*.

End users can also issue SQL statements through the Query Management Facility (QMF) or some other program, and the library for that program might provide all the instruction or reference material they need. For a list of some of the titles in the QMF library, see the bibliography at the end of this book.

Application Programming: Some users access DB2 without knowing it, using programs that contain SQL statements. DB2 application programmers write those programs. Because they write SQL statements, they need *Application Programming and SQL Guide*, *SQL Reference*, and *Call Level Interface Guide and Reference* just as end users do.

Application programmers also need instructions on many other topics:

- How to transfer data between DB2 and a host program—written in COBOL, C, or FORTRAN, for example
- How to prepare to compile a program that embeds SQL statements
- How to process data from two systems simultaneously, say DB2 and IMS or DB2 and CICS
- How to write distributed applications across platforms
- How to write applications that use DB2 Call Level Interface to access DB2 servers
- How to write applications that use Open Database Connectivity (ODBC) to access DB2 servers
- How to write applications in the Java programming language to access DB2 servers

The material needed for writing a host program containing SQL is in *Application Programming and SQL Guide* and *Application Programming Guide and Reference for Java*. The material needed for writing applications that use DB2 Call Level Interface or ODBC to access DB2 servers is in *Call Level Interface Guide and Reference*.

For handling errors, see *Messages and Codes*.

Information about writing applications across platforms can be found in *Distributed Relational Database Architecture: Application Programming Guide*.

System and Database Administration: *Administration* covers almost everything else. *Administration Guide* divides those tasks among the following sections:

- Section 2 (Volume 1) of *Administration Guide* discusses the decisions that must be made when designing a database and tells how to bring the design into being by creating DB2 objects, loading data, and adjusting to changes.

- Section 3 (Volume 1) of *Administration Guide* describes ways of controlling access to the DB2 system and to data within DB2, to audit aspects of DB2 usage, and to answer other security and auditing concerns.
- Section 4 (Volume 1) of *Administration Guide* describes the steps in normal day-to-day operation and discusses the steps one should take to prepare for recovery in the event of some failure.
- Section 5 (Volume 2) of *Administration Guide* explains how to monitor the performance of the DB2 system and its parts. It also lists things that can be done to make some parts run faster.

In addition, the appendixes in *Administration Guide* contain valuable information on DB2 sample tables, National Language Support (NLS), writing exit routines, interpreting DB2 trace output, and character conversion for distributed data.

If you are involved with DB2 only to design the database, or plan operational procedures, you need *Administration Guide*. If you also want to carry out your own plans by creating DB2 objects, granting privileges, running utility jobs, and so on, then you also need:

- *SQL Reference*, which describes the SQL statements you use to create, alter, and drop objects and grant and revoke privileges
- *Utility Guide and Reference*, which explains how to run utilities
- *Command Reference*, which explains how to run commands

If you will be using data sharing, then you need *Data Sharing: Planning and Administration*, which describes how to plan for and implement data sharing.

Additional information about system and database administration can be found in *Messages and Codes*, which lists messages and codes issued by DB2, with explanations and suggested responses.

Diagnosis: Diagnosticians detect and describe errors in the DB2 program. They might also recommend or apply a remedy. The documentation for this task is in *Diagnosis Guide and Reference* and *Messages and Codes.*

How to Obtain DB2 Information

DB2 on the Web

Stay current with the latest information about DB2. View the DB2 home page on the World Wide Web. News items keep you informed about the latest enhancements to the product. Product announcements, press releases, fact sheets, and technical articles help you plan your database management strategy. Technical professionals can access DB2 publications on the Web and follow links to other Web sites with more information about DB2 family and OS/390 solutions. Access DB2 on the Web with the following URL:

http://www.ibm.com/software/db2os390

DB2 Publications

The DB2 publications are available in both hardcopy and softcopy format. Using online books on CD-ROM, you can read, search across books, print portions of the text, and make notes in these BookManager books. With the appropriate BookManager READ product or IBM Library Readers, you can view these books on the MVS, VM, OS/2, DOS, AIX and Windows platforms.

When you order DB2 Version 5, you are entitled to one copy of the following CD-ROM, which contains the DB2 licensed book for no additional charge:

DB2 Server for OS/390 Version 5 Licensed Online Book, LK2T-9075.

You can order multiple copies for an additional charge by specifying feature code 8207.

When you order DB2 Version 5, you are entitled to one copy of the following CD-ROM, which contains the DB2 and DATABASE 2 Performance Monitor online books for no additional charge:

DB2 Server for OS/390 Version 5 Online Library, SK2T-9092

You can order multiple copies for an additional charge through IBM's publication ordering service.

Periodic updates will be provided on the following collection kit available to licensees of DB2 Version 5:

IBM Online Library Transaction Processing and Data Collection, SK2T-0730

SK2T-9092 will be superseded by SK2T-0730 when updates to the online library are available.

In some countries,including the United States and Canada, you receive one copy of the collection kit at no additional charge when you order DB2 Version 5. You will automatically receive one copy of the collection kit each time it is updated, for no additional charge. To order multiple copies of SK2T-0730 for an additional charge, see “How to Order the DB2 Library” on page 7. In other countries, updates will be available in displayable softcopy format in the IBM Online Book Library Offering (5636–PUB), SK2T-0730 IBM Online Library Transaction Processing and Data Collection at a later date.

See your IBM representative for assistance in ordering the collection.

DB2 Server for OS/390 books are also available for an additional charge on the following collection kits, which contain online books for many IBM products:

IBM Online Library MVS Collection, SK2T-0710, in English

Online Library Omnibus Edition OS/390 Collection, SK2T-6700, in English

IBM Online Library MVS Collection Kit, SK88-8002, in Japanese, for viewing on DOS and Windows platforms

How to Order the DB2 Library

You can order DB2 publications and CD-ROMs through your IBM representative or the IBM branch office serving your locality. If you are located within the United States or Canada, you can place your order by calling one of the toll-free numbers :

- In the U.S., call 1-800-879-2755.
- In Canada, call 1-800-565-1234.

To order additional copies of licensed publications, specify the SOFTWARE option. To order additional publications or CD-ROMs, specify the PUBLICATIONS & SLSS option. Be prepared to give your customer number, the product number, and the feature code(s) or order numbers you want.

Summary of Changes to DB2 for OS/390 Version 5

DB2 for OS/390 Version 5 delivers a database server solution for OS/390. Version 5 supports all functions available in DB2 for MVS/ESA Version 4 plus enhancements in the areas of performance, capacity, and availability, client/server and open systems, and user productivity.

If you are currently using DB2, **you can migrate only from a DB2 for MVS/ESA Version 4 subsystem**. This summary gives you an overview of the differences to be found between these versions.

Server Solution

OS/390 retains the classic strengths of the traditional MVS/ESA operating system, while offering a network-ready, integrated operational environment.

The following features work directly with DB2 for OS/390 applications to help you use the full potential of your DB2 subsystem:

- Net.Data for OS/390
- DB2 Installer
- DB2 Estimator for Windows
- DB2 Visual Explain
- Workstation-based Performance Analysis and Tuning
- DATABASE 2 Performance Monitor

Net.Data for OS/390

Net.Data provides support for Internet access to DB2 data through a Web server. Applications built with Net.Data make data stored in any DB2 server more accessible and useful. Net.Data Web applications provide continuous application availability, scalability, security, and high performance.

This no charge feature can be ordered with DB2 Version 5 or downloaded from Internet. The Net.Data URL is:

http://www.ibm.com/software/data/net.data/downloads.html

DB2 Installer

DB2 Installer offers the option to install DB2 on an OS/2 workstation. Now, you can use a friendly graphical interface to complete installation tasks easily with DB2 Installer.

This function is delivered on CD-ROM with DB2 Visual Explain.

DB2 Estimator for Windows

DB2 Estimator provides an easy-to-use capacity planning tool. You can estimate the sizes of tables and indexes, and the performance of SQL statements, groups of SQL statements (transactions), utility runs, and groups of transactions (capacity runs). From a simple table sizing to a detailed performance analysis of an entire DB2 application, DB2 Estimator saves time and lowers costs. You can investigate the impact of new or modified applications on your production system, *before* you implement them.

This no charge feature can be ordered with DB2 Version 5 or downloaded from the Internet. From the internet, use the IBM Software URL:

http://www.ibm.com/software/

From here, you can access information about DB2 Estimator using the download function.

DB2 Visual Explain

DB2 Visual Explain lets you tune DB2 SQL statements on an OS/2 workstation. You can see DB2 EXPLAIN output in a friendly graphical interface and easily access, modify, and analyze applications with DB2 Visual Explain.

Workstation-based Performance Analysis and Tuning

The new workstation-based Performance Analysis and Tuning function simplifies system administration. You can access statistical data to help you analyze and improve system performance. This function works with the optional DB2 PM feature to provide full analysis and tuning functionality.

DATABASE 2 Performance Monitor (DB2 PM)

DB2 PM lets you monitor, analyze, and optimize the performance of DB2 Version 5 and its applications. An online monitor, for both host and workstation environments, provides an immediate "snap-shot" view of DB2 activities and allows for exception processing while the system is operational. The workstation-based online monitor can connect directly to the Visual Explain function of the DB2 base product.

DB2 PM also offers a history facility, a wide variety of customizable reports for in-depth performance analysis, and an EXPLAIN function to analyze and optimize SQL statements. For more information, see *DB2 PM for OS/390 General Information* .

This feature can be ordered with DB2 Version 5.

Performance

Sysplex Query Parallelism

The increased power of Sysplex query parallelism in DB2 for OS/390 Version 5 allows DB2 to go far beyond DB2 for MVS/ESA Version 4 capabilities; from the ability to split and process a single query within a DB2 subsystem to processing that same query across many different DB2 subsystems in a data sharing group.

The advances this release offers in scalable query processing let you process queries quickly while accommodating the potential growth of data sharing groups and the increasing complexity of queries.

Prepared Statement Caching

DB2 reduces the cost of duplicate prepares for the same dynamic SQL statement by saving them in a cache. Now, different application processes can share prepared statements and they are preserved past the commit point. This performance improvement offers the most benefit for:

- Client/server applications that frequently use dynamic SQL for repeated execution of SQL statements
- Relatively short dynamic SQL statements for which PREPARE cost accounts for most of the CPU expended

Reoptimization

When host variables, parameter markers, or special registers were used in previous releases, DB2 could not always determine the best access path because the values for these variables were unknown. Now, you can tell DB2 to reevaluate the access path at run time, after these values are known. As a result, queries can be processed more efficiently, and response time is improved.

Faster Transactions and Batch

- Caching of package authorization improves performance at run time for remote packages and applications that use pattern-matching characters in a package list.
- You can define a table space to use ***selective partition locking***, which can reduce locking costs for applications that do partition-at-a-time processing. It also can reduce locking costs for certain data sharing applications that rely on an affinity between members and data partitions.
- A new standalone utility lets you preformat active logs.
- With LOAD and REORG, you can preformat data sets up to the high allocated RBA, which can make processing for sequential inserts more predictable.

Faster Utilities

- LOAD and REORG jobs run faster and more efficiently with enhanced index key sorting that reduces CPU and elapsed time, and an inline copy feature that lets you make an image copy without a separate copy step.
- New REORG options let you select rows to discard during a REORG and, optionally, write the discarded records to a file.
- When you run the REBUILD, RECOVER, REORG, or LOAD utility on DB2-managed indexes or table spaces, a new option lets you logically reset and reuse the DB2-managed objects.

- RECOVER INDEX and LOAD run faster on large numbers of rows per page.
- Sampling support for RUNSTATS reduces the processing required to collect nonindexed column statistics.
- BSAM striping improves the I/O capability of DB2 utilities.

Other Performance Enhancements

- There are several significant performance enhancements to data sharing, including selective partition locking, the MAXROWS option, and several optimizations to reduce data sharing overhead.
- DB2 installations that run in the OS/390 Version 2 Release 6 environment can now have as many as (approximately) 25 000 open DB2 data sets at one time. The maximum number of open data sets in earlier releases of OS/390 is 10000.
- You can easily alter the length of variable-length character columns using the new ALTER COLUMN clause of the ALTER TABLE statement.
- SQL CASE expressions let you eliminate queries with multiple UNIONs and improve performance by using only one table scan.
- You can collect a new statistic on concatenated index keys to improve the performance of queries with correlated columns. The statistic lets DB2 estimate the number of rows that qualify for the query more accurately, and select access paths more efficiently.
- DB2 scans partitions more efficiently and allows scans during parallel processing.
- Query enhancements include the ability to:
  - Use indexes for joins on string columns that have different lengths
  - Use an index to access predicates with noncorrelated IN subqueries
- Noncolumn expressions in simple predicates are evaluated at stage 1 and can be indexable.

Increased Capacity

DB2 for OS/390 Version 5 introduces the concept of a *large* partitioned table space. Defining your table space as large allows a substantial capacity increase: to approximately one terabyte of data and up to 254 partitions. In addition to accommodating growth potential, large partitioned table spaces make database design more flexible, and can improve availability.

Improved Availability

Online REORG

DB2 for OS/390 Version 5 adds a major improvement to availability with *Online REORG*. Now, you can avoid the severe availability problems that occurred while offline reorganization of table spaces restricted access to read only during the unload phase and no access during reload phase of the REORG utility. Online REORG gives you full read and write access to your data through most phases of the process with only very brief periods of read only or no access.

Data Sharing Enhancements

- Version 5 provides continuous availability with group buffer pool duplexing. Prior releases of DB2 rely on DASD and the merged recovery logs to recover group buffer pool (GBP) data that is lost if a coupling facility fails. With group buffer pool duplexing, DB2 writes changed pages to both a *primary GBP* and a *secondary GBP.* Overlapped writes to the GBPs provide good performance and eliminate the writes to DASD.
- Group buffer pool rebuild makes coupling facility maintenance easier and improves access to the group buffer pool during connectivity losses.
- Automatic group buffer pool recovery accelerates GBP recovery time, eliminates operator intervention, and makes data available faster when GBPs are lost because of coupling facility failures.
- Improved restart performance for members of a data sharing group reduces the impact of retained locks by making data available faster when a group member fails.
- Changes to traces and DISPLAY GROUPBUFFERPOOL output improve monitoring.

Tracker site for disaster recovery

You can set up a tracker site that shadows the activity of a primary site, and eliminate the need to constantly ship image copies.

Client/Server and Open Systems

Native TCP/IP Network Support

DB2's support of TCP/IP networks allows DRDA clients to connect directly to DDF and eliminate the gateway machine. In addition, customers can now use asynchronous transfer mode (ATM) as the underlying communication protocol for both SNA and TCP/IP connections to DB2.

Stored Procedures

- Return multiple SQL result sets to local and remote clients in a single network operation.
- Receive calls from applications that use standard interfaces, such as Open Database Connectivity** (ODBC) and X/Open** Call Level Interface, to access data in DB2 for OS/390.
- Run in an enhanced environment. DB2 supports multiple stored procedures address spaces managed by the MVS Workload Manager (WLM). The WLM environment offers efficient program management and allows WLM-managed stored procedures to run as subprograms and use RACF security.
- Use individual MVS dispatching priorities to improve stored procedure scheduling.
- Access data sources outside DB2 with two-phase commit coordination.
- Use an automatic COMMIT feature on return to the caller that reduces network traffic and the length of time locks are held.
- Have the ability to invoke utilities, which means you can now invoke utilities from an application that uses the SQL CALL statement.

- Support IMS Open Database Access (ODBA). Now a DB2 stored procedure can directly connect to IMS DBCTL and access IMS data.

Dynamic Query and Network Performance

Improvements for DRDA Applications

- Reduced processing costs for block fetch operations
- DRDA support for OPTIMIZE FOR n ROWS on SELECT
- Faster dynamic SQL queries and reduced processing costs for VTAM network operations
- Reduced message traffic for dynamic SQL SELECT statements

Improved Application Portability

- DB2 for OS/390 Version 5 introduces the DB2 Call Level Interface (CLI) to MVS/ESA. Unlike applications that use embedded SQL to access DB2 data, applications that choose CLI are not tied to a precompiler, packages, or a plan.

  Workstation and desktop applications use standard interfaces, such as Open Database Connectivity (ODBC), to access relational data. Standard interfaces need one version of an application to access many data sources. Now, you can port UNIX workstation and PC desktop applications to DB2 for OS/390 and exploit the CLI (ODBC) capabilities without modification. In addition, applications can issue ODBC or CLI calls from within a stored procedure.
- You can now access DB2 for OS/390 databases in your Java applications. You can use DB2 Connect Java Database Connectivity (JDBC) for your dynamic SQL applications, or SQLJ for your static SQL applications.
- DB2 adds DRDA support for the DESCRIBE INPUT statement to improve performance for many ODBC applications.
- Now, you can write multithreaded DB2 CLI applications, and restrictions on connection switching no longer exist.
- DB2 now provides ASCII table support for clients and servers across platforms. This support reduces the cost of translation between EBCDIC and ASCII encoding schemes. ASCII table support also offers an alternative to writing field procedures that provide the ASCII sort sequence, which improves performance.

Improved Security

- DB2 for OS/390 supports Distributed Computing Environment (DCE) for authenticating remote DRDA clients. DCE offers the following benefits:
  - Network security: By providing an encrypted DCE ticket for authentication, remote clients do not need to send an MVS password in readable text.
  - Simplified security administration: End users do not need to maintain a valid password on MVS to access DB2; instead, they maintain their DCE password only.
- New descriptive error codes help you determine the cause of network security errors.
- You can change end user MVS passwords from DRDA clients.

User Productivity

Improved SQL Compatibility

DB2 conforms to the ANSI/ISO SQL entry level standard of 1992. Application programmers can take advantage of a more complete set of standard SQL to use across the DB2 family to write portable applications. New SQL function includes:

- More check options for view definitions.
- Foreign keys that reference UNIQUE keys as well as PRIMARY keys.
- An extension to GRANT that lets the REFERENCES privilege apply to a list of columns.
- A new delete rule, NO ACTION, that you can use to define referential constraints for self-referencing tables.
- SQL CASE expressions provide the capability to create conditional logic wherever an expression is allowed.
- SQL temporary tables allow application programs to easily create and use temporary tables that store results of SQL transactions without logging or recovery.

New Access Choice

A new attachment facility, the Recoverable Resource Manager Services attachment facility, improves access in a client/server environment. It coordinates two-phase commit processing between DB2 and other participating resource managers in any MVS application environment. Other key features include the ability for multiple users to run in a single address space, thread reuse, and moving threads between MVS tasks.

Image Copy Enhancements

The COPY, LOAD, and REORG utilities provide:

- Features of the COPY utility that help you quickly determine what type of image copy to take, when to take it, and let DB2 automatically take it for you.
- Inline copy in LOAD and REORG that lets you create an image copy while improving data availability.

Improved Integration of C^{++} and IBM COBOL for MVS & VM Support

It is easier for application programmers to use object-oriented programming techniques in their DB2 applications. DB2 for OS/390 Version 5 adds COBOL and C^{++} languages as options on installation panels, DB2I panels, the DSNH command, and DCLGEN.

Other Usability Enhancements

- To prevent long running units of work and to help avoid unnecessary work during the recovery phase of restart, DB2 issues new warning messages at an interval of your choice.
- A new special register for decimal precision provides better granularity, so that applications that need different values for decimal precision can run in the same DB2 subsystem.

- Trace records for IFCID 0022 now include most information in the PLAN_TABLE.
- An increase from 127 to 255 rows on a page improves table space processing and eliminates the need for compression.
- Install SYSOPR can recover objects using the START DATABASE command.
- A filtering capability for DISPLAY BUFFERPOOL limits statistics information to a specified set of page sets.
- You can enter comments within the SYSIN input stream for DB2 utilities.

Summary of Changes to This Book

A new API, `SQLDescribeParam()`, is added to “Chapter 5. Functions” on page 73.

All application and service diagnostic and debugging information is in a new chapter, “Chapter 7. Problem Diagnosis” on page 381.

This book no longer contains information about DB2 for OS/390's JDBC support. See *Application Programming Guide and Reference for Java*.

Updates are marked with revision bars.

Chapter 2. Introduction to CLI

DB2 Call Level Interface (CLI) is IBM's callable SQL interface by the DB2 family of products. It is a 'C' and 'C^{++}' application programming interface for relational database access, and it uses function calls to pass dynamic SQL statements as function arguments. It is an alternative to embedded dynamic SQL, but unlike embedded SQL, it does not require a precompiler.

DB2 CLI is based on the Microsoft** Open Database Connectivity** (ODBC) specification, and the X/Open** Call Level Interface specification. These specifications were chosen as the basis for the DB2 Call Level Interface in an effort to follow industry standards and to provide a shorter learning curve for those application programmers already familiar with either of these *data source* interfaces. In addition, some DB2 specific extensions were added to help the DB2 application programmer specifically exploit DB2 features.

DB2 CLI Background Information

To understand DB2 CLI or any callable SQL interface, it is helpful to understand what it is based on, and to compare it with existing interfaces.

The X/Open Company and the SQL Access Group jointly developed a specification for a callable SQL interface referred to as the X/Open Call Level Interface. The goal of this interface is to increase the portability of applications by enabling them to become independent of any one database product vendor's programming interface. Most of the X/Open Call Level Interface specification was accepted as part of the ISO Call Level Interface Draft International Standard (ISO CLI DIS).

Microsoft developed a callable SQL interface called Open Database Connectivity (ODBC) for Microsoft operating systems based on a preliminary draft of X/Open CLI. The Call Level Interface specifications in ISO, X/Open, ODBC, and DB2 CLI continue to evolve in a cooperative manner to provide functions with additional capabilities.

The ODBC specification also includes an operating environment where data source specific ODBC drivers are dynamically loaded at run time by a driver manager based on the data source name provided on the connect request. The application is linked directly to a single driver manager library rather than to each DBMS's library. The driver manager mediates the application's function calls at run time and ensures they are directed to the appropriate DBMS specific ODBC driver.

The ODBC driver manager only knows about the ODBC-specific functions, that is, those functions supported by the DBMS for which no API is specified. Therefore, DBMS-specific functions cannot be directly accessed in an ODBC environment. However, DBMS-specific dynamic SQL statements are indirectly supported via a mechanism called the vendor escape clause. See “Using Vendor Escape Clauses” on page 376 for detailed information.

ODBC is not limited to Microsoft operating systems, other implementations are available, or are emerging on various platforms.

Differences Between DB2 CLI and ODBC Version 2.0.

While DB2 CLI is derived from the ISO Call Level Interface Draft International Standard (ISO CLI DIS) and ODBC Version 2.0., most existing products are written to ODBC specifications.

If you port existing ODBC applications to DB2 for OS/390 or write a new application according to the ODBC specifications, you must comply with the specifications defined in this publication. However, before you write to any API, validate that the API is supported by DB2 for OS/390 and that the syntax and semantics are identical. If there are any differences, you must code to the APIs documented in this publication.

On the DB2 for OS/390 platform, no ODBC driver manager exists. Consequently, DB2 CLI support is implemented as a CLI/ODBC driver/driver manager that is loaded at run time into the application address space. See “DB2 CLI Runtime Environment” on page 50 for details about the DB2 CLI runtime environment.

The DB2 for common server CLI executes on Windows and AIX as an ODBC driver, loaded by the Windows driver manager (Windows environment) or the Visi genic driver manager (UNIX platforms). In this context, DB2 CLI support is limited to the ODBC specifications. Alternatively, an application can directly invoke the CLI application programming interfaces (APIs) including those not supported by ODBC. In this context, the set of APIs supported by DB2 for common server is referred to as the "Call Level Interface". See *DATABASE 2 Call Level Interface Guide and Reference for common servers*.

The use of DB2 CLI in this publication refers to DB2 for OS/390 support of Call Level Interface unless otherwise noted.

ODBC Features Supported

DB2 CLI support should be viewed as consisting of most of ODBC Version 2.0 as well as IBM extensions. Where differences exist, applications should be written to the specifications defined in this publication.

DB2 CLI includes support of the following ODBC functions:

- All ODBC level 1 functions
- All ODBC level 2 functions with the following exceptions:

  SQLBrowseConnect()
  SQLSetPos()
  SQLSetScrollOptions()

- Some X/Open CLI functions
- Some DB2 specific functions

For a complete list of supported functions, see “DB2 CLI Function Summary” on page 74.

The following DB2 features are available to both ODBC and DB2 CLI applications:

- The double byte (graphic) data types
- Stored procedures
- Distributed unit of work (DUW) as defined by DRDA, two-phase commit

DB2 CLI contains extensions to access DB2 features that can not be accessed by ODBC applications:

- SQLCA access for detailed DB2 specific diagnostic information
- Control over null termination of output strings.

DB2 CLI does not support the following functions (a deviation from the Microsoft ODBC Version 2.0 Specification):

- Multiple connections to the same data source
- Asynchronous SQL
- A connection to a data source unless the connection state is on a transaction boundary (CONNECT (Type 1) only)
- Scrollable cursor support
- Interactive data source connection as specified via `SQLBrowseConnect()` and `SQLDriverConnect()`.

For more information on the relationship between DB2 CLI and ODBC, see Appendix B, "DB2 CLI and ODBC" on page 399.

Differences Between DB2 CLI and Embedded SQL

An application that uses an embedded SQL interface requires a precompiler to convert the SQL statements into code, which is then compiled, bound to the data source, and executed. In contrast, a DB2 CLI application does not have to be precompiled or bound, but instead uses a standard set of functions to execute SQL statements and related services at run time.

This difference is important because, traditionally, precompilers have been specific to each database product, which effectively ties your applications to that product. DB2 CLI enables you to write portable applications that are independent of any particular database product. This independence means DB2 CLI applications do not have to be recompiled or rebound to access different DB2 or DRDA data sources, but rather just connect to the appropriate data source at run time.

DB2 CLI and embedded SQL also differ in the following ways:

- DB2 CLI does not require the explicit declaration of cursors. They are generated by DB2 CLI as needed. The application can then use the generated cursor in the normal cursor fetch model for multiple row `SELECT` statements and positioned `UPDATE` and `DELETE` statements.
- The `OPEN` statement is not used in DB2 CLI. Instead, the execution of a `SELECT` automatically causes a cursor to be opened.
- Unlike embedded SQL, DB2 CLI allows the use of parameter markers on the equivalent of the `EXECUTE IMMEDIATE` statement (the `SQLExecDirect()` function).
- A `COMMIT` or `ROLLBACK` in DB2 CLI is issued via the `SQLTransact()` function call rather than by passing it as an SQL statement.
- DB2 CLI manages statement related information on behalf of the application, and provides a *statement handle* to refer to it as an abstract object. This handle eliminates the need for the application to use product specific data structures.

- Similar to the statement handle, the *environment handle* and *connection handle* provide a means to refer to all global variables and connection specific information.
- DB2 CLI uses the SQLSTATE values defined by the X/Open SQL CAE specification. Although the format and most of the values are consistent with values used by the IBM relational database products, there are differences. (There are also differences between ODBC SQLSTATES and the X/Open defined SQLSTATES). Refer to Table 143 on page 409 for a cross reference of all DB2 CLI SQLSTATEs.

Despite these differences, there is an important common concept between embedded SQL and DB2 CLI:

> DB2 CLI can execute any SQL statement that can be prepared dynamically in embedded SQL.

Table 1 lists each DB2 for OS/390 SQL statement, and indicates whether or not it can be executed using DB2 CLI.

Each DBMS might have additional statements that can be dynamically prepared, in which case DB2 CLI passes them to the DBMS. There is one exception: COMMIT and ROLLBACK can be dynamically prepared by some DBMSs but are not passed. The `SQLTransact()` function should be used instead to specify either COMMIT or ROLLBACK.

Table 1 (Page 1 of 2). SQL Statements

SQL Statement	Dynamic [a]	Call Level Interface [c] (CLI)
ALTER TABLE	X	X
ALTER DATABASE	X	X
ALTER INDEX	X	X
ALTER STOGROUP	X	X
ALTER TABLESPACE	X	X
BEGIN DECLARE SECTION [b]		
CALL		X [d]
CLOSE		`SQLFreeStmt()`
COMMENT ON	X	X
COMMIT	X	`SQLTransact()`
CONNECT (Type 1)		`SQLConnect()`, `SQLDriverConnect()`
CONNECT (Type 2)		`SQLConnect()`, `SQLDriverConnect()`
CREATE { ALIAS, DATABASE, INDEX, STOGROUP, SYNONYM, TABLE, TABLESPACE, VIEW }	X	X
DECLARE CURSOR [b]		`SQLAllocStmt()`
DECLARE STATEMENT		
DECLARE TABLE		
DELETE	X	X
DESCRIBE		`SQLDescribeCol()`, `SQLColAttributes()`

Table 1 (Page 2 of 2). SQL Statements

SQL Statement	Dynamic [a]	Call Level Interface [c] (CLI)
DROP	X	X
END DECLARE SECTION [b]		
EXECUTE		`SQLExecute()`
EXECUTE IMMEDIATE		`SQLExecDirect()`
EXPLAIN	X	X
FETCH		`SQLFetch()`
GRANT	X	X
INCLUDE [b]		
INSERT	X	X
LABEL ON	X	X
LOCK TABLE	X	X
OPEN		`SQLExecute()`, `SQLExecDirect()`
PREPARE		`SQLPrepare()`
RELEASE		
REVOKE	X	X
ROLLBACK	X	`SQLTransact()`
select-statement	X	X
SELECT INTO		
SET CONNECTION		`SQLSetConnection()`
SET host_variable		
SET CURRENT DEGREE	X	X
SET CURRENT PACKAGESET		
SET CURRENT SQLID	X	X
UPDATE	X	X
WHENEVER [b]		

Note:

[a] All statements in this list can be coded as static SQL, but only those marked with X can be coded as dynamic SQL.

[b] This statement is not executable.

[c] An X indicates that this statement can be executed using either `SQLExecDirect()` or `SQLPrepare()` and `SQLExecute()`. If there is an equivalent DB2 CLI function, the function name is listed.

[d] Although this statement is not dynamic, DB2 CLI allows the statement to be specified when calling either `SQLExecDirect()` or `SQLPrepare()` and `SQLExecute()`.

Advantages of Using DB2 CLI

DB2 CLI provides a number of key features that offer advantages in contrast to embedded SQL. DB2 CLI:

- Ideally suits the client-server environment in which the target data source is unknown when the application is built. It provides a consistent interface for executing SQL statements, regardless of which database server the application connects to.
- Lets you write portable applications that are independent of any particular database product. DB2 CLI applications do not have to be recompiled or rebound to access different DB2 or DRDA data sources. Instead they connect to the appropriate data source at run time.
- Reduces the amount of management required for an application while in general use. Individual DB2 CLI applications do not need to be bound to each data source. Bind files provided with DB2 CLI need to be bound only once for all DB2 Call Level Interface applications.
- Lets applications connect to multiple data sources from the same application.
- Allocates and controls data structures, and provides a handle for the application to refer to them. Application do not have to control complex global data areas such as the SQLDA and SQLCA.
- Provides enhanced parameter input and fetching capability. You can specify arrays of data on input to retrieve multiple rows of a result set directly into an array. You can execute statements that generate multiple result sets.
- Lets you retrieve multiple rows and result sets generated from a call to a stored procedure.
- Provides a consistent interface to query catalog information that is contained in various DBMS catalog tables. The result sets that are returned are consistent across DBMSs. Application programmers can avoid writing version-specific and server-specific catalog queries.
- Provides extended data conversion which requires less application code when converting information between various SQL and C data types.
- Aligns with the emerging ISO CLI standard in addition to using the accepted industry specifications of ODBC and X/Open CLI.
- Allows application developers to apply their knowledge of industry standards directly to DB2 Call Level Interface. The interface is intuitive for programmers who are familiar with function libraries but know little about product specific methods of embedding SQL statements into a host language.

Deciding Which Interface To Use

DB2 CLI is ideally suited for query-based applications that require portability. Use the following guidelines to help you decide which interface meets your needs.

Static and Dynamic SQL

Only embedded SQL applications can use static SQL. Both static and dynamic SQL have advantages. Consider these factors:

- Performance

  Dynamic SQL is prepared at run time. Static SQL is prepared at bind time. The preparation step for dynamic SQL requires more processing and might incur additional network traffic.

  However, static SQL does not always perform better than dynamic SQL. Dynamic SQL can make use of changes to the data source, such as new indexes, and can use current catalog statistics to choose the optimal access plan.

- Encapsulation and Security

  In static SQL, authorization to objects is associated with a package and validated at package bind time. Database administrators can grant execute authority on a particular package to a set of users rather than grant explicit access to each database object.

  In dynamic SQL, authorization is validated at run time on a per statement basis; therefore, users must be granted explicit access to each database object.

Use Both Interfaces

An application can take advantage of both static and dynamic interfaces. An application programmer can create a stored procedure that contains static SQL. The stored procedure is called from within a DB2 CLI application and executed on the server. After the stored procedure is created, any DB2 CLI or ODBC application can call it.

Write a Mixed Application

You can write a mixed application that uses both DB2 CLI and embedded SQL. In this scenario, DB2 CLI provides the base application, and you write key modules using static SQL for performance or security. Choose this option only if stored procedures do not meet your applications requirements.

Other Information Sources

Application developers should refer to the following publications as a supplement to this publication:

- *ODBC 2.0 Programmer's Reference and SDK Guide*
- *Inside ODBC*

When writing DB2 CLI applications, you also might need to reference information for the database servers that are being accessed, in order to understand any connectivity issues, environment issues, SQL language support issues, and other server-specific information. For DB2 for OS/390 versions, see *SQL Reference* and *Application Programming and SQL Guide*. If you are writing applications that access other DB2 server products, see *IBM SQL Reference* for information that is common to all products, including any differences.

Chapter 3. Writing a DB2 CLI Application

This section introduces a conceptual view of a typical DB2 CLI application.

A DB2 CLI application can be broken down into a set of tasks. Some of these tasks are organized into discrete steps, while others might apply throughout the application. Each task is carried out by calling one or more DB2 CLI functions.

Tasks described in this section are basic tasks that apply to all applications. More advanced tasks, such as using array insert, are described in “Chapter 6. Using Advanced Features” on page 341.

The functions are used in examples to illustrate their use in DB2 CLI applications. Refer to “Chapter 5. Functions” on page 73 for complete descriptions and usage information for each of the functions.

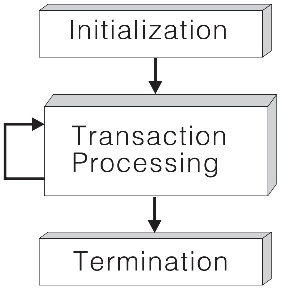

Figure 1. Conceptual View of a DB2 CLI Application

Every DB2 CLI application contains the three main tasks shown in Figure 1.

Initialization
This task allocates and initializes some resources in preparation for the main *transaction processing* task. Refer to “Initialization and Termination” on page 24 for details.

Transaction Processing
This is the main task of the application. SQL statements are passed to DB2 CLI to query and modify the data. Refer to “Transaction Processing” on page 29 for details.

Termination
This task frees allocated resources. The resources generally consist of data areas identified by unique handles. Refer to “Initialization and Termination” on page 24 for details.

In addition to the three tasks listed above, there are general tasks, such as handling diagnostic messages, which occur throughout an application.

Initialization and Termination

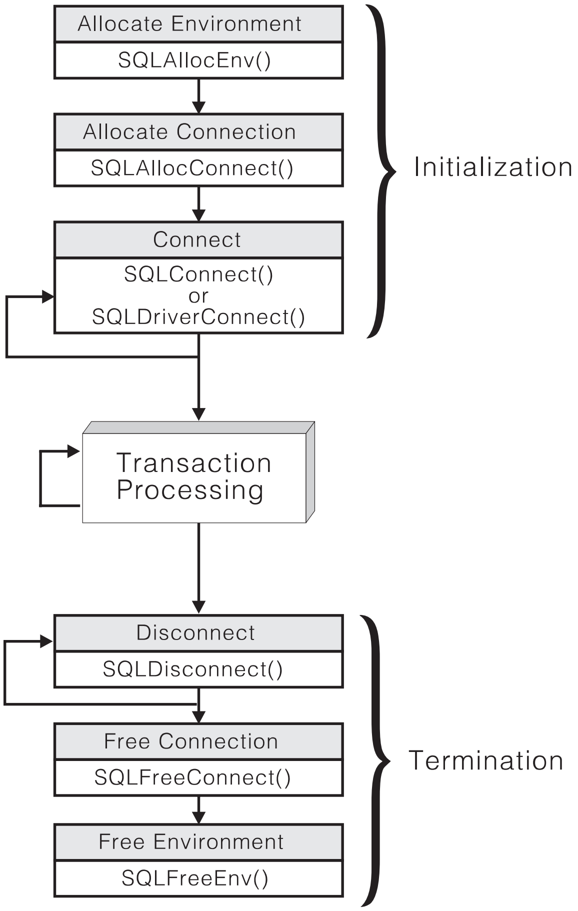

Figure 2. Conceptual View of Initialization and Termination Tasks

Figure 2 shows the function call sequences for both the initialization and termination tasks. The transaction processing task in the middle of the diagram is shown in Figure 3 on page 29.

Handles

The initialization task consists of the allocation and initialization of environment and connection handles (which are later freed in the termination task). An application then passes the appropriate handle when it calls other DB2 CLI functions. A handle is a variable that refers to a data object controlled by DB2 CLI. Using handles relieves the application from having to allocate and manage global variables or data structures, such as the SQLDA or SQLCA, used in IBM's embedded SQL interfaces.

There are three types of handles:

Environment Handle

The environment handle refers to the data object that contains information regarding the global state of the application, such as attributes and connections. This handle is allocated by calling `SQLAllocEnv()`, and freed by calling `SQLFreeEnv()`. An environment handle must be allocated before a connection handle can be allocated.

Connection Handle

A connection handle refers to a data object that contains information associated with a connection to a particular data source. This includes connection options, general status information, transaction status, and diagnostic information. Each connection handle is allocated by calling `SQLAllocConnect()` and freed by calling `SQLFreeConnect()`.

An application can be connected to several database servers at the same time. An application requires a connection handle for each concurrent connection to a database server. For information on multiple connections, refer to “Connecting to One or More Data Sources” on page 26.

Call `SQLGetInfo()` to determine if a user-imposed limit on the number of connection handles has been set.

Statement Handles

Statement handles are discussed in the next section, “Transaction Processing” on page 29.

ODBC Connection Model

The ODBC specifications support any number of concurrent connections, each of which is an independent transaction. That is, the application can issue SQLConnect to X, perform some work, issue SQLConnect to Y, perform some work, and then commit the work at X. ODBC supports multiple concurrent and independent transactions, one per connection.

DB2 CLI Restrictions on the ODBC Connection Model

If the application is not using the `MULTICONTEXT=1` initialization file setting, there are restrictions on DB2 CLI's support of the ODBC connection model. To obtain simulated support of the ODBC connection model, the application must specify a CONNECT type value of 1 (either by using the initialization file or the `SQLSetConnectOption()` API. See “Initialization Keywords” on page 64 and “Specifying the Connect Type” on page 26.)

The application can then logically connect to any number of data sources. However, the DB2 CLI driver maintains only a single physical connection, that of the last data source to which the application successfully connected or at which the last SQL statement was executed.

As a result, the application is affected as follows:

- When connected to one or more data sources so that the application has allocated some number of connect handles, any attempt to connect to a new data source COMMITs the work on the current data source and terminates that connection. Therefore, the application cannot have cursors concurrently open at two data sources (including cursors WITH HOLD).
- If the application is currently connected to X and has performed work at X that has not yet been committed or rolled back, then any execution of an API to perform work on a valid statement handle Y results in committing the transaction at X and reestablishing the connection to Y.

With multiple context support, DB2 CLI can fully support the ODBC connection model. See “DB2 CLI Support of Multiple Contexts” on page 368.

CONNECT Type 1 and Type 2

Every IBM RDBMS supports both type 1 and type 2 CONNECT semantics. In both cases, there is only one transaction active at any time.

CONNECT (Type 1) lets the application connect to only a single database at any time so that the single transaction is active on the current connection. This models the DRDA remote unit of work (RUW) processing.

Conversely, CONNECT (Type 2) connect lets the application connect concurrently to any number of database servers, all of which participate in the single transaction. This models the DRDA distributed unit of work (DUW) processing.

ODBC does not support multiple connections participating in a distributed transaction.

Specifying the Connect Type

The connect type **must** be established prior to issuing `SQLConnect`. You can establish the connect type using either of the following methods:

- Specify the CONNECTTYPE keyword in the common section of the initialization file with a value of 1 (CONNECT (Type 1)) or 2 (CONNECT (Type 2)). The initialization file is described in “DB2 CLI Initialization File” on page 62.
- Invoke `SQLSetConnectOption()`. Specify *fOption* = SQL_CONNECTTYPE with a value of SQL_CONCURRENT_TRANS (CONNECT (Type 1)) or a value of SQL_COORDINATED_TRANS (CONNECT (Type 2)).

Connecting to One or More Data Sources

DB2 CLI supports connections to remote data sources through DRDA.

If the application is executing with CONNECT (Type 1) and `MULTICONTEXT=0`, then DB2 CLI allows an application to *logically* connect to multiple data sources; however, all transactions other than the transaction associated with the current connection, must be complete (committed or rolled back). If the application is executing with CONNECT (Type 2), then the transaction is a distributed unit of work and all data sources participate in the disposition of the transaction (commit or rollback).

To connect concurrently to one or more data sources, an application calls `SQLAllocConnect()` once for each connection. The subsequent connection handle is used with `SQLConnect()` to request a data source connection and with `SQLAllocStmt()` to allocate statement handles for use within that connection. There is also an extended connect function, `SQLDriverConnect()`, which allows for additional connect options.

Unlike the distributed unit of work connections described in “Distributed Unit of Work (Coordinated Distributed Transactions)” on page 343, there is no coordination between the statements that are executed on different connections.

Initialization and Connection Example

```
/* ... */
/******************************************************
**    - demonstrate basic connection to two datasources.
**    - error handling mostly ignored for simplicity
**
**  Functions used:
**
**    SQLAllocConnect  SQLDisconnect
**    SQLAllocEnv      SQLFreeConnect
**    SQLConnect       SQLFreeEnv
**  Local Functions:
**    DBconnect
**
*******************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "sqlcli1.h"

int
DBconnect(SQLHENV henv,
          SQLHDBC * hdbc,
          char    * server);

#define MAX_UID_LENGTH    18
#define MAX_PWD_LENGTH    30
#define MAX_CONNECTIONS   2

int
main( )
{
    SQLHENV         henv;
    SQLHDBC         hdbc[MAX_CONNECTIONS];
    char *          svr[MAX_CONNECTIONS] =
                    {
                      "KARACHI"   ,
                      "DAMASCUS"
                    }

    /* allocate an environment handle   */
    SQLAllocEnv(&henv);

    /* Connect to first data source */
    DBconnect(henv, &hdbc[0],
              svr[0]);

    /* Connect to second data source */
    DBconnect(henv, &hdbc[1],
              svr[1]);
    /*********   Start Processing Step  *************************/
    /* allocate statement handle, execute statement, etc.       */
    /*********   End Processing Step  ***************************/

    /************************************************************/
    /* Commit work on connection 1.                             */
    /************************************************************/

    SQLTransact (henv,
                 hdbc[0],
                 SQL_COMMIT);


    /************************************************************/
    /* Commit work on connection 2. This has NO effect on the   */
    /* transaction active on connection 1.                      */
    /************************************************************/

    SQLTransact (henv,
                 hdbc[1],
                 SQL_COMMIT);

    printf("\nDisconnecting .....\n");

    SQLDisconnect(hdbc[0]);     /* disconnect first connection */

    SQLDisconnect(hdbc[1]);     /* disconnect second connection */
    SQLFreeConnect(hdbc[0]);    /* free first connection handle  */
    SQLFreeConnect(hdbc[1]);    /* free second connection handle */
    SQLFreeEnv(henv);           /* free environment handle       */

    return (SQL_SUCCESS);
}

/********************************************************************
**   Server is passed as a parameter. Note that USERID and PASSWORD**
**   are always NULL.                                              **
********************************************************************/

int
DBconnect(SQLHENV henv,
          SQLHDBC * hdbc,
          char    * server)
{
    SQLRETURN       rc;
    SQLCHAR         buffer[255];
    SQLSMALLINT     outlen;


    SQLAllocConnect(henv, hdbc);/* allocate a connection handle     */

    rc = SQLConnect(*hdbc, server, SQL_NTS, NULL, SQL_NTS, NULL, SQL_NTS);
    if (rc != SQL_SUCCESS) {
        printf(">--- Error while connecting to database: %s -------\n", server);
        return (SQL_ERROR);
    } else {
        printf(">Connected to %s\n", server);
        return (SQL_SUCCESS);
    }
}
/* ... */
```

Transaction Processing

The following figure shows the typical order of function calls in a DB2 CLI application. Not all functions or possible paths are shown.

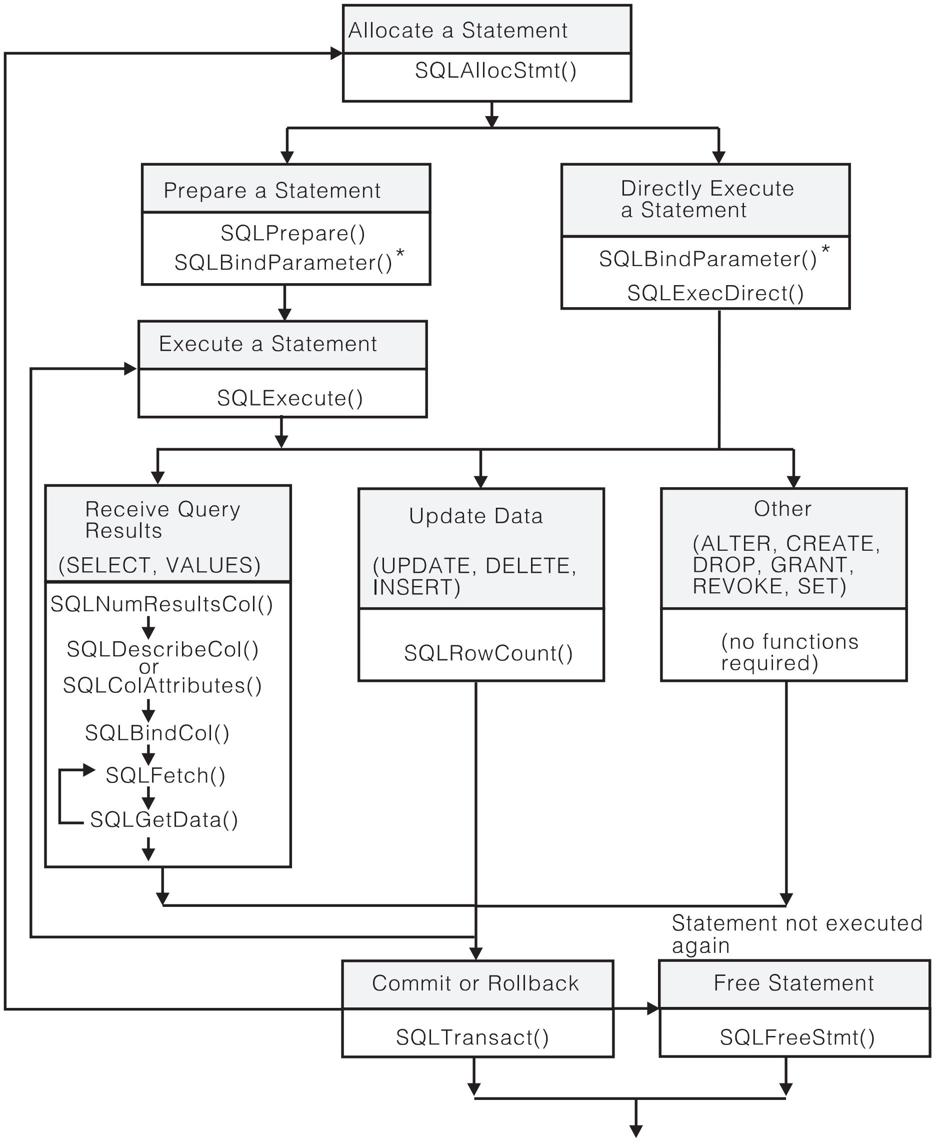

Figure 3. Transaction Processing

Figure 3 shows the steps and the DB2 CLI functions in the transaction processing task. This task contains five steps:

- Allocating statement handles
- Preparation and execution of SQL statements
- Processing results
- Commit or rollback
- Optionally, freeing statement handles if the statement is unlikely to be executed again.

Although the slightly simpler `SQLSetParam()` function can generally be used in place of `SQLBindParameter()`, the more current `SQLBindParameter()` function is recommended.

Allocating Statement Handles

SQLAllocStmt() allocates a statement handle. A statement handle refers to the data object that is used to track the execution of a single SQL statement. This includes information such as statement options, SQL statement text, dynamic parameters, cursor information, bindings for dynamic arguments and columns, result values and status information (these are discussed later). Each statement handle is associated with a unique connection handle.

A statement handle must be allocated before a statement can be executed. By default, the maximum number of statement handles that can be allocated at any one time is limited by the application heap size.

Preparation and Execution

After a statement handle is allocated, there are two methods of specifying and executing SQL statements:

1. Prepare then execute
   a. Call SQLPrepare() with an SQL statement as an argument.
   b. Call SQLBindParameter(), or SQLSetParam() if the SQL statement contains *parameter markers*.
   c. Call SQLExecute().
2. Execute direct
   a. Call SQLBindParameter() or SQLSetParam() if the SQL statement contains *parameter markers*.
   b. Call SQLExecDirect() with an SQL statement as an argument.

The first method splits the preparation of the statement from the execution. This method is used when:

- The statement is executed repeatedly (usually with different parameter values). This avoids having to prepare the same statement more than once. The subsequent executions make use of the access plans already generated by the prepare.
- The application requires information about the columns in the result set, prior to statement execution.

The second method combines the prepare step and the execute step into one. This method is used when:

- The statement is executed only once. This avoids having to call two functions to execute the statement.
- The application does not require information about the columns in the result set, before the statement is executed.

DB2 for OS/390 and DB2 for common server provide *dynamic statement caching* at the database server. In DB2 CLI terms this means that for a given statement handle, once a statement is prepared, it does not need to be prepared again (even after commits or rollbacks), as long as the statement handle is not freed. Applications that repeatedly execute the same SQL statement across multiple transactions, can save a significant amount of processing time and network traffic by:

1. Associating each such statement with its own statement handle, and

2. Preparing these statements once at the beginning of the application, then
3. Executing the statements as many times as is needed throughout the application.

Binding Parameters in SQL Statements

Both of the execution methods described above, allow the use of parameter markers in place of an *expression* (or host variable in embedded SQL) in an SQL statement.

Parameter markers are represented by the '?' character and indicate the position in the SQL statement where the contents of application variables are to be substituted when the statement is executed. The parameter markers are referenced sequentially, from left to right, starting at 1. `SQLNumParams()` can be used to determine the number of parameters in a statement.

When an application variable is associated with a parameter marker it is *bound* to the parameter marker. The application must bind an application variable to each parameter marker in the SQL statement before it executes that statement. Binding is carried out by calling the `SQLBindParameter()` function with a number of arguments to indicate, the numerical position of the parameter, the SQL type of the parameter, the data type of the variable, a pointer to the application variable, and length of the variable.

The bound application variable and its associated length are called *deferred* input arguments since only the pointers are passed when the parameter is bound; no data is read from the variable until the statement is executed. Deferred arguments allow the application to modify the contents of the bound parameter variables, and repeat the execution of the statement with the new values.

Information for each parameter remains in effect until overridden, or until the application unbinds the parameter or drops the statement handle. If the application executes the SQL statement repeatedly without changing the parameter binding, then DB2 CLI uses the same pointers to locate the data on each execution. The application can also change the parameter binding to a different set of deferred variables. The application must not de-allocate or discard variables used for deferred input fields between the time it binds the fields to parameter markers and the time DB2 CLI accesses them at execution time.

It is possible to bind the parameters to a variable of a different type from that required by the SQL statement. The application must indicate the C data type of the source, and the SQL type of the parameter marker, and DB2 CLI converts the contents of the variable to match the SQL data type specified. For example, the SQL statement might require an integer value, but your application has a string representation of an integer. The string can be bound to the parameter, and DB2 CLI converts the string to the corresponding integer value when you execute the statement.

`SQLDescribeParam()` can be used to determine the data type of a parameter marker. If the application indicates an incorrect type for the parameter marker, either an extra conversion by the DBMS, or an error can occur. See "Data Types and Data Conversion" on page 38 for more information about data conversion.

# For information on more advanced methods of binding application storage to
# parameter markers, see “Using Arrays to Input Parameter Values” on page 353
# and “Sending/Retrieving Long Data in Pieces” on page 351.

Processing Results

The next step after the statement has been executed depends on the type of SQL statement.

Processing Query (SELECT, VALUES) Statements

If the statement is a query statement, the following steps are generally needed in order to retrieve each row of the result set:

1. Establish (describe) the structure of the result set, number of columns, column types and lengths
2. (Optionally) bind application variables to columns in order to receive the data
3. Repeatedly fetch the next row of data, and receive it into the bound application variables
4. (Optionally) retrieve columns that were not previously bound, by calling `SQLGetData()` after each successful fetch

Each of the above steps requires some diagnostic checks. “Chapter 6. Using Advanced Features” on page 341 discusses advanced techniques of using `SQLExtendedFetch()` to fetch multiple rows at a time.

Step 1

Analyze the executed or prepared statement. If the SQL statement was generated by the application, then this step might not be necessary since the application might know the structure of the result set and the data types of each column. If the application does know the structure of the entire result set, and if there are a very large number of columns to retrieve, then the application might wish to supply DB2 CLI with the descriptor information. This can reduce network traffic since DB2 CLI does not have to retrieve the information from the server.

On the other hand, if the SQL statement was generated at runtime (for example, entered by a user), then the application has to query the number of columns, the type of each column, and perhaps the names of each column in the result set. This information can be obtained by calling `SQLNumResultCols()` and `SQLDescribeCol()` (or `SQLColAttributes()`) after preparing or after executing the statement.

Step 2

The application retrieves column data directly into an application variable on the next call to `SQLFetch()`. For each column to be retrieved, the application calls `SQLBindCol()` to bind an application variable to a column in the result set. The application can use the information obtained from Step 1 to determine the C data type of the application variable and to allocate the maximum storage the column value could occupy. Similar to variables bound to parameter markers using `SQLBindParameter()` and `SQLSetParam()`, columns are bound to deferred arguments. This time the variables are deferred output arguments, as data is written to these storage locations when `SQLFetch()` is called.

If the application does not bind any columns, as in the case when it needs to retrieve columns of long data in pieces, it can use `SQLGetData()`. Both the `SQLBindCol()` and `SQLGetData()` techniques can be combined if some columns are bound and some are unbound. The application must not deallocate or discard variables used for deferred output fields between the time it binds them to columns of the result set and the time DB2 CLI writes the data to these fields.

Step 3

Call SQLFetch() to fetch the first or next row of the result set. If any columns are bound, the application variable is updated. There is also a method that allows the application to fetch multiple rows of the result set into an array, refer to "Retrieving A Result Set Into An Array" on page 357 for more information.

If data conversion was indicated by the data types specified on the call to SQLBindCol(), the conversion occurs when SQLFetch() is called. Refer to "Data Types and Data Conversion" on page 38 for an explanation.

Step 4 (Optional)

Call SQLGetData() to retrieve any unbound columns. All columns can be retrieved this way, provided they were not bound. SQLGetData() can also be called repeatedly to retrieve large columns in smaller pieces, which cannot be done with bound columns.

Data conversion can also be indicated here, as in SQLBindCol(), by specifying the desired target C data type of the application variable. Refer to "Data Types and Data Conversion" on page 38 for more information.

To unbind a particular column of the result set, use SQLBindCol() with a null pointer for the application variable argument (*rgbValue*) To unbind all of the columns with one function call, use SQLFreeStmt().

Applications generally perform better if columns are bound rather than retrieved using SQLGetData(). However, an application can be constrained in the amount of long data that it can retrieve and handle at one time. If this is a concern, then SQLGetData() might be the better choice.

For information on more advanced methods for binding application storage to result set columns, refer to "Retrieving A Result Set Into An Array" on page 357 and "Sending/Retrieving Long Data in Pieces" on page 351.

Processing UPDATE, DELETE and INSERT Statements

If the statement is modifying data (UPDATE, DELETE or INSERT), no action is required, other than the normal check for diagnostic messages. In this case, SQLRowCount() can be used to obtain the number of rows affected by the SQL statement.

If the SQL statement is a positioned UPDATE or DELETE, it is necessary to use a *cursor*. A cursor is a moveable pointer to a row in the result table of an active query statement. (This query statement must contain the FOR UPDATE OF clause to ensure that the query is not opened as read-only.) In embedded SQL, cursors names are used to retrieve, update or delete rows. In DB2 CLI, a cursor name is needed only for positioned UPDATE or DELETE SQL statements as they reference the cursor by name.

To update a row that was fetched, the application uses two statement handles, one for the fetch and one for the update. The application calls SQLGetCursorName() to obtain the cursor name. The application generates the text of a positioned UPDATE or DELETE, including this cursor name, and executes that SQL statement using a second statement handle. The application cannot reuse the fetch statement handle to execute a positioned UPDATE or DELETE as it is still in use. You can also define your own cursor name using SQLSetCursorName(), but it is best to use the generated name, since all error messages reference the generated name, rather than the name defined by SQLSetCursorName().

Processing Other Statements

If the statement neither queries nor modifies the data, then there is no further action other than the normal check for diagnostic messages.

Commit or Rollback

A *transaction* is a recoverable unit of work, or a group of SQL statements that can be treated as one atomic operation. This means that all the operations within the group are guaranteed to be completed (committed) or undone (rolled back), as if they were a single operation. A transaction can also be referred to as a unit of work or a logical unit of work. When the transaction spans multiple connections, it is referred to as a distributed unit of work.

DB2 CLI supports two commit modes: *auto-commit* and *manual-commit*.

In auto-commit mode, every SQL statement is a complete transaction, which is automatically committed. For a non-query statement, the commit is issued at the end of statement execution. For a query statement, the commit is issued after the cursor is closed. Given a single statement handle, the application must not start a second query before the cursor of the first query is closed.

In manual-commit mode, transactions are started implicitly with the first access to the data source using SQLPrepare(), SQLExecDirect(), SQLGetTypeInfo(), or any function that returns a result set, such as those described in “Querying System Catalog Information” on page 348. At this point a transaction begins, even if the call failed. The transaction ends when you use SQLTransact() to either rollback or commit the transaction. This means that any statements executed (on the same connection) between these are treated as one transaction.

The default commit mode is auto-commit (except when participating in a coordinated transaction, see “Distributed Unit of Work (Coordinated Distributed Transactions)” on page 343). An application can switch between manual-commit and auto-commit modes by calling SQLSetConnectOption(). Typically, a query-only application might wish to stay in auto-commit mode. Applications that need to perform updates to the data source should turn off auto-commit as soon as the data source connection is established.

When multiple connections exist, each connection has its own transaction (unless CONNECT (Type 2) is specified). Special care must be taken to call SQLTransact() with the correct connection handle to ensure that only the intended connection and related transaction is affected. Unlike distributed unit of work connections (described in “Distributed Unit of Work (Coordinated Distributed Transactions)” on page 343), there is no coordination between the transactions on each connection.

When to Call SQLTransact()

If the application is in auto-commit mode, it never needs to call SQLTransact(), a commit is issued implicitly at the end of each statement execution.

In manual-commit mode, SQLTransact() must be called before calling SQLDisconnect(). If distributed unit of work is involved, additional rules can apply. Refer to “Distributed Unit of Work (Coordinated Distributed Transactions)” on page 343 for details.

It is recommended that an application that performs updates should not wait until the disconnect before committing or rolling back the transaction. The other extreme is to operate in auto-commit mode, which is also not recommended as this adds extra processing. The application can modify the auto-commit mode by invoking the SQLSetConnectOption() function. See “Environment, Connection, and Statement Options” on page 341 and the SQLSetConnectOption() function for information about switching between auto-commit and manual-commit.

Consider the following when deciding where in the application to end a transaction:

# - If using CONNECT (Type 1) with MULTICONTEXT=0, only the current connection
| can have an outstanding transaction. If using CONNECT (Type 2), all
| connections participate in a single transaction.
# - If using MULTICONTEXT=1, each connection can have an outstanding transaction.
- Various resources can be held while you have an outstanding transaction. Ending the transaction releases the resources for use by other users.
- When a transaction is successfully committed or rolled back, it is fully recoverable from the system logs. Open transactions are not recoverable.

Effects of Calling SQLTransact()

When a transaction ends:

- All locks on DBMS objects are released, except those that are associated with a held cursor.
- Prepared statements are preserved from one transaction to the next if the data source supports statement caching (DB2 for OS/390 Version 5 does). After a statement is prepared on a specific statement handle, it does not need to be prepared again even after a commit or rollback, provided the statement continues to be associated with the same statement handle.
- Cursor names, bound parameters, and column bindings are maintained from one transaction to the next.
- By default, cursors are preserved after a commit (but not a rollback). All cursors are defined using the WITH HOLD clause (except when connected to SQL/DS, which does not support the WITH HOLD clause). For information about changing the default behavior, refer to “SQLSetStmtOption - Set Statement Option” on page 315.

For more information and an example refer to “SQLTransact - Transaction Management” on page 338.

Freeing Statement Handles

Call SQLFreeStmt() to end processing for a particular statement handle. This function can be used to do one or more of the following:

- Unbind all columns of the result set
- Unbind all parameter markers
- Close any cursors and discard any pending results
- Drop the statement handle, and release all associated resources

The statement handle can be reused for other statements provided it is not dropped. If a statement handle is reused for another SQL statement string, any cached access plan for the original statement is discarded.

The columns and parameters should always be unbound before using the handle to process a statement with a different number or type of parameters or a different result set; otherwise application programming errors might occur.

Diagnostics

Diagnostics refers to dealing with warning or error conditions generated within an application. There are two levels of diagnostics when calling DB2 CLI functions :

- Return codes
- Detailed diagnostics (SQLSTATEs, messages, SQLCA)

Each CLI function returns the function return code as a basic diagnostic. The SQLError() function provides more detailed diagnostic information. The SQLGetSQLCA() function provides access to the SQLCA, if the diagnostic is reported by the data source. This arrangement lets applications handle the basic flow control, and the SQLSTATES allow determination of the specific causes of failure.

The SQLError() function returns three pieces of information:

- SQLSTATE
- Native error: if the diagnostic is detected by the data source, this is the SQLCODE; otherwise, this is set to -99999.
- Message text: this is the message text associated with the SQLSTATE.

For the detailed function information and example usage, refer to “SQLError - Retrieve Error Information” on page 143.

Function Return Codes

The following table lists all possible return codes for DB2 CLI functions.

Table 2. DB2 CLI Function Return Codes

Return Code	Explanation
SQL_SUCCESS	The function completed successfully, no additional SQLSTATE information is available.
SQL_SUCCESS_WITH_INFO	The function completed successfully, with a warning or other information. Call `SQLError()` to receive the SQLSTATE and any other informational messages or warnings. The SQLSTATE class is '01'. See Table 143 on page 409.
SQL_NO_DATA_FOUND	The function returned successfully, but no relevant data was found. When this is returned after the execution of an SQL statement, additional information might be available which can be obtained by calling `SQLError()`.
SQL_NEED_DATA	The application tried to execute an SQL statement but DB2 CLI lacks parameter data that the application had indicated would be passed at execute time. For more information, see “Sending/Retrieving Long Data in Pieces” on page 351.
SQL_ERROR	The function failed. Call `SQLError()` to receive the SQLSTATE and any other error information.
SQL_INVALID_HANDLE	The function failed due to an invalid input handle (environment, connection or statement handle). This is a programming error. No further information is available.

SQLSTATEs

Since different database servers often have different diagnostic message codes, DB2 CLI provides a standard set of *SQLSTATEs* that are defined by the X/Open SQL CAE specification. This allows consistent message handling across different database servers.

SQLSTATEs are alphanumeric strings of 5 characters (bytes) with a format of `ccsss`, where `cc` indicates class and `sss` indicates subclass. Any SQLSTATE that has a class of:

- '01', is a warning.
- 'S1', is generated by the DB2 CLI driver.

Note: X/Open has reserved class 'HY' for CLI implementations, which is currently equivalent to the 'S1' class. This might be a consideration if you intend to follow the X/Open and/or ISO CLI standard in the future.

For some error conditions, DB2 CLI returns SQLSTATES that differ from those states listed in the Microsoft *ODBC 2.0 Programmer' s Reference and SDK Guide*. This is a result of DB2 CLI following the X/Open SQL CAE and SQL92 specifications.

DB2 CLI SQLSTATEs include both additional IBM-defined SQLSTATEs that are returned by the database server, and DB2 CLI defined SQLSTATEs for conditions that are not defined in the X/Open specification. This allows for the maximum amount of diagnostic information to be returned.

Follow these guidelines for using SQLSTATEs within your application:

- Always check the function return code before calling `SQLError()` to determine if diagnostic information is available.
- Use the SQLSTATEs rather than the native error code.
- To increase your application's portability, only build dependencies on the subset of DB2 CLI SQLSTATEs that are defined by the X/Open specification, and return the additional ones as information only. (Dependencies refers to the application making logic flow decisions based on specific SQLSTATEs.)

  Note: It might be useful to build dependencies on the class (the first 2 characters) of the SQLSTATEs.
- For maximum diagnostic information, return the text message along with the SQLSTATE (if applicable, the text message also includes the IBM-defined SQLSTATE). It is also useful for the application to print out the name of the function that returned the error.

See Table 143 on page 409 for a listing and description of the SQLSTATEs explicitly returned by DB2 CLI.

SQLCA

Embedded applications rely on the SQLCA for all diagnostic information. Although DB2 CLI applications can retrieve much of the same information by using `SQLError()`, there still might be a need for the application to access the SQLCA related to the processing of a statement. (For example, after preparing a statement, the SQLCA contains the relative cost of executing the statement.) The SQLCA only contains meaningful information if there was an interaction with the data source on the previous request (for example: connect, prepare, execute, fetch, disconnect).

The `SQLGetSQLCA()` function is used to retrieve this structure. See “SQLGetSQLCA - Get SQLCA Data Structure” on page 229 for more information.

Data Types and Data Conversion

When writing a DB2 CLI application it is necessary to work with both SQL data types and C data types. This is unavoidable since the DBMS uses SQL data types, and the application must use C data types. This means the application must match C data types to SQL data types when transferring data between the DBMS and the application (when calling DB2 CLI functions).

To help address this, DB2 CLI provides symbolic names for the various data types, and manages the transfer of data between the DBMS and the application. It also performs data conversion (from a C character string to an SQL INTEGER type, for example) if required. To accomplish this, DB2 CLI needs to know both the source and target data type. This requires the application to identify both data types using symbolic names.

C and SQL Data Types

Table 3 on page 40 lists each of the SQL data types, with its corresponding symbolic name, and the default C symbolic name. These data types represent the combination of the ODBC V2.0 minimum, core, and extended data types. The ODBC extended data type SQL_BIGINT is not supported. In addition, DB2 CLI supports SQL_GRAPHIC, SQL_VARGRAPHIC and SQL_LONGVARGRAPHIC.

SQL Data Type

This column contains the SQL data types as they would appear in an SQL CREATE DDL statement. The SQL data types are dependent on the DBMS.

Symbolic SQL Data Type

This column contains SQL symbolic names that are defined (in `sqlcli1.h`) as an integer value. These values are used by various functions to identify the SQL data types listed in the first column. See "Example" on page 130 for an example using these values.

Default C Symbolic Data Type

This column contains C symbolic names, also defined as integer values. These values are used in various function arguments to identify the C data type as shown in Table 4 on page 41. The symbolic names are used by various functions, (such as `SQLBindParameter()`, `SQLGetData()`, `SQLBindCol()`) to indicate the C data types of the application variables. Instead of explicitly identifying the C data type when calling these functions, SQL_C_DEFAULT can be specified instead, and DB2 CLI assumes a default C data type based on the SQL data type of the parameter or column as shown by this table. For example, the default C data type of SQL_DECIMAL is SQL_C_CHAR.

Table 3. SQL Symbolic and Default Data Types

SQL Data Type	Symbolic SQL Data Type	Default Symbolic C Data Type
CHAR	SQL_CHAR	SQL_C_CHAR
CHAR FOR BIT DATA	SQL_BINARY	SQL_C_BINARY
DATE	SQL_DATE	SQL_C_DATE
DECIMAL	SQL_DECIMAL	SQL_C_CHAR
DOUBLE	SQL_DOUBLE	SQL_C_DOUBLE
FLOAT	SQL_FLOAT	SQL_C_DOUBLE
GRAPHIC	SQL_GRAPHIC	SQL_C_DBCHAR
INTEGER	SQL_INTEGER	SQL_C_LONG
LONG VARCHAR	SQL_LONGVARCHAR	SQL_C_CHAR
LONG VARCHAR FOR BIT DATA	SQL_LONGVARBINARY	SQL_C_BINARY
LONG VARGRAPHIC	SQL_LONGVARGRAPHIC	SQL_C_DBCHAR
NUMERIC [a]	SQL_NUMERIC [a]	SQL_C_CHAR
REAL [b]	SQL_REAL	SQL_C_FLOAT
SMALLINT	SQL_SMALLINT	SQL_C_SHORT
TIME	SQL_TIME	SQL_C_TIME
TIMESTAMP	SQL_TIMESTAMP	SQL_C_TIMESTAMP
VARCHAR	SQL_VARCHAR	SQL_C_CHAR
VARCHAR FOR BIT DATA	SQL_VARBINARY	SQL_C_BINARY
VARGRAPHIC	SQL_VARGRAPHIC	SQL_C_DBCHAR

Note:

a NUMERIC is a synonym for DECIMAL on DB2 for OS/390, DB2 for VSE and VM and DB2 for common server.
b REAL is not valid for DB2 for common server or DB2 for OS/390.

The data types, DATE, DECIMAL, NUMERIC, TIME, and TIMESTAMP cannot be transferred to their default C buffer types without a conversion.

Table 4 on page 41 shows the generic C type definitions for each symbolic C type.

C Symbolic Data Type
This column contains C symbolic names, defined as integer values. These values are used in various function arguments to identify the C data type shown in the last column. See “Example” on page 90 for an example using these values.

C Type
This column contains C defined types, defined in `sqlcli1.h` using a C `typedef` statement. The values in this column should be used to declare all DB2 CLI related variables and arguments, in order to make the application more portable. Refer to Table 6 on page 42 for a list of additional symbolic data types used for function arguments.

Base C Type
This column is shown for reference only. All variables and arguments should be defined using the symbolic types in the previous column. Some of the values are C structures that are described in Table 5 on page 41.

Table 4. C Data Types

C Symbolic Data Type	C Type	Base C type
SQL_C_CHAR	SQLCHAR	unsigned char
SQL_C_BIT	SQLCHAR	unsigned char or char (Value 1 or 0)
SQL_C_TINYINT	SQLSCHAR	signed char (Range -128 to 127)
SQL_C_SHORT	SQLSMALLINT	short int
SQL_C_LONG	SQLINTEGER	long int
SQL_C_DOUBLE	SQLDOUBLE	double
SQL_C_FLOAT	SQLREAL	float
SQL_C_DATE	DATE_STRUCT	see Table 5 on page 41
SQL_C_TIME	TIME_STRUCT	see Table 5 on page 41
SQL_C_TIMESTAMP	TIMESTAMP_STRUCT	see Table 5 on page 41
SQL_C_BINARY	SQLCHAR	unsigned char
SQL_C_DBCHAR	SQLDBCHAR	wchar_t

Table 5. C DATE, TIME, and TIMESTAMP Structures

<table>
<tr><th>C Type</th><th>Generic Structure</th></tr>
<tr><td>DATE_STRUCT</td><td><pre>
typedef struct DATE_STRUCT
  {
    SQLSMALLINT   year;cols='&cw2. &cw3. *'
    SQLUSMALLINT   month;
    SQLUSMALLINT   day;
  } DATE_STRUCT;
</pre></td></tr>
<tr><td>TIME_STRUCT</td><td><pre>
typedef struct TIME_STRUCT
  {
    SQLUSMALLINT   hour;
    SQLUSMALLINT   minute;
    SQLUSMALLINT   second;
  } TIME_STRUCT;
</pre></td></tr>
<tr><td>TIMESTAMP_STRUCT</td><td><pre>
typedef struct TIMESTAMP_STRUCT
  {
    SQLUSMALLINT   year;
    SQLUSMALLINT   month;
    SQLUSMALLINT   day;
    SQLUSMALLINT   hour;
    SQLUSMALLINT   minute;
    SQLUSMALLINT   second;
    SQLINTEGER   fraction;
  } TIMESTAMP_STRUCT;
</pre></td></tr>
</table>

Refer to Table 6 on page 42 for more information on the SQLUSMALLINT C data type.

Other C Data Types

In addition to the data types that map to SQL data types, there are also C symbolic types used for other function arguments, such as pointers and handles. Both the generic and ODBC data types are shown below.

Table 6. C Data Types and Base C Data Types

Defined C Type	Base C Type	Typical Usage
SQLPOINTER	void *	Pointers to storage for data and parameters.
SQLHENV	long int	Handle referencing environment information.
SQLHDBC	long int	Handle referencing data source connection information.
SQLHSTMT	long int	Handle referencing statement information.
SQLUSMALLINT	unsigned short int	Function input argument for unsigned short integer values.
SQLUINTEGER	unsigned long int	Function input argument for unsigned long integer values.
SQLRETURN	short int	Return code from DB2 CLI functions.

Data Conversion

As mentioned previously, DB2 CLI manages the transfer and any required conversion of data between the application and the DBMS. Before the data transfer actually takes place, the source, target or both data types are indicated when calling `SQLBindParameter()`, `SQLBindCol()` or `SQLGetData()`. These functions use the symbolic type names shown in Table 3 on page 40, to identify the data types involved.

For example, to bind a parameter marker that corresponds to an SQL data type of DECIMAL(5,3), to an application's C buffer type of double, the appropriate `SQLBindParameter()` call would look like:

```
SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_DOUBLE,
                  SQL_DECIMAL, 5, 3, double_ptr, NULL);
```

Table 3 shows only the default data conversions. The functions mentioned in the previous paragraph can be used to convert data to other types, but not all data conversions are supported or make sense. Table 7 on page 43 shows all the conversions supported by DB2 CLI.

The first column in Table 7 contains the data type of the SQL data type, the remaining columns represent the C data types. If the C data type columns contains:

D The conversion is supported and is the default conversion for the SQL data type.
X All IBM DBMSs support the conversion,
Blank No IBM DBMS supports the conversion.

For example, the table indicates that a CHAR (or a C character string as indicated in Table 7) can be converted into a SQL_C_LONG (a signed long). In contrast, a LONGVARCHAR cannot be converted to a SQL_C_LONG.

See Appendix E, “Data Conversion” on page 419 for information about required formats and the results of converting between data types.

Limits on precision, and scale, as well as truncation and rounding rules for type conversions follow rules specified in the *IBM SQL Reference* with the following exception; truncation of values to the right of the decimal point for numeric values returns a truncation warning, whereas truncation to the left of the decimal point returns an error. In cases of error, the application should call `SQLError()` to obtain the SQLSTATE and additional information on the failure. When moving and converting floating point data values between the application and DB2 CLI, no correspondence is guaranteed to be exact as the values can change in precision and scale.

Table 7. Supported Data Conversions

SQL Data Type	**SQL_C_CHAR**	**SQL_C_LONG**	**SQL_C_SHORT**	**SQL_C_TINYINT**	**SQL_C_FLOAT**	**SQL_C_DOUBLE**	**SQL_C_DATE**	**SQL_C_TIME**	**SQL_C_TIMESTAMP**	**SQL_C_BINARY**	**SQL_C_BIT**	**SQL_C_DBCHAR**
SQL_CHAR	X	X	X	X	X	X	X	X	X	X	X	
SQL_DATE	X						D		X			
SQL_DECIMAL	D	X	X	X	X	X					X	
SQL_DOUBLE	X	X	X	X	X	D					X	
SQL_FLOAT	X	X	X	X	X	D					X	
SQL_GRAPHIC	X											D
SQL_INTEGER	X	D	X	X	X	X					X	
SQL_LONGVARCHAR	D						X		X	X		
SQL_LONGVARGRAPHIC	X									X		D
SQL_NUMERIC	D	X	X	X	X	X					X	
SQL_REAL	X	X	X	X	D	X					X	
SQL_SMALLINT	X	X	D	X	X	X					X	
SQL_TIME	X							D	X			
SQL_TIMESTAMP	X						X	X	D			
SQL_VARCHAR	D	X	X	X	X	X	X	X	X	X	X	
SQL_VARGRAPHIC	X											D

Note:

REAL is not supported by DB2 for common server.

NUMERIC is a synonym for DECIMAL on DB2 for OS/390, DB2 for VSE and VM, and DB2 for common server.

Working With String Arguments

The following conventions deal with the various aspects of working with string arguments in Call Level Interface functions.

Length of String Arguments

Input string arguments have an associated length argument. This argument indicates to Call Level Interface, either the exact length of the argument (not including the null terminator), the special value SQL_NTS to indicate a null-terminated string, or SQL_NULL_DATA to pass a NULL value. If the length is set to SQL_NTS, Call Level Interface determines the length of the string by locating the null terminator.

Output string arguments have two associated length arguments, an input length argument to specify the length of the allocated output buffer, and an output length argument to return the actual length of the string returned by DB2 CLI. The returned length value is the total length of the string available for return, regardless of whether it fits in the buffer or not.

For SQL column data, if the output is a null value, SQL_NULL_DATA is returned in the length argument and the output buffer is untouched.

If a function is called with a null pointer for an output length argument, Call Level Interface does not return a length, and assumes that the data buffer is large enough to hold the data. When the output data is a NULL value, DB2 CLI can not indicate that the value is NULL. If it is possible that a column in a result set can contain a NULL value, a valid pointer to the output length argument must always be provided. It is highly recommended that a valid output length argument always be used.

If the length argument (*pcbValue*) and the output buffer (*rgbValue*) are contiguous in memory, DB2 CLI can return both values more efficiently, improving application performance. For example, if the following structure is defined and `&buffer.pcbValue` and `buffer.rgbValue` are passed to `SQLBindCol()`, DB2 CLI updates both values in one operation.

```
struct
{   SQLINTEGER pcbValue;
    SQLCHAR    rgbValue [BUFFER_SIZE];
} buffer;
```

Null-Termination of Strings

By default, every character string that Call Level Interface returns is terminated with a null terminator (hex 00), except for strings returned from the graphic data type into SQL_C_CHAR application variables. The graphic data type that is retrieved into SQL_C_DBCHAR application variables is null terminated with a double byte null terminator. This requires that all buffers allocate enough space for the maximum number of bytes expected, plus the null-terminator.

It is also possible to use `SQLSetEnvAttr()` and set an environment attribute to disable null termination of variable length output (character string) data. In this case, the application allocates a buffer exactly as long as the longest string it expects. The application must provide a valid pointer to storage for the output length argument so that DB2 CLI can indicate the actual length of data returned; otherwise, the application has no means to determine this. The DB2 CLI default is to always write the null terminator.

String Truncation

If an output string does not fit into a buffer, DB2 CLI truncates the string to the size of the buffer, and writes the null terminator. If truncation occurs, the function returns SQL_SUCCESS_WITH_INFO and an SQLSTATE of **01**004 indicating truncation. The application can then compare the buffer length to the output length to determine which string was truncated.

For example, if `SQLFetch()` returns SQL_SUCCESS_WITH_INFO, and an SQLSTATE of **01**004, at least one of the buffers bound to a column is too small to hold the data. For each buffer that is bound to a column, the application can compare the buffer length with the output length and determine which column was truncated.

ODBC specifies that string data can be truncated on input or output with the appropriate SQLSTATE. As the data source, an IBM relational database (DB2) does not truncate data on input, but might truncate data on output to maintain data integrity. On input, DB2 rejects string truncation with a negative SQLCODE (-302) and an SQLSTATE of **22**001. On output, DB2 truncates the data and issues SQL_SUCCESS_WITH_INFO and an SQLSTATE of **01**004.

Interpretation of Strings

Normally, DB2 CLI interprets string arguments in a case-sensitive manner and does not trim any spaces from the values. The one exception is the cursor name input argument on the `SQLSetCursorName()` function. In this case, if the cursor name is not delimited (enclosed by double quotes) the leading and trailing blanks are removed and case is preserved.

Querying Environment and Data Source Information

There are many situations when an application requires information about the characteristics and capabilities of the current DB2 CLI driver or the data source that it is connected to.

One of the most common situations involves displaying information for the user. Information such as the data source name and version, or the version of the DB2 CLI driver might be displayed at connect time, or as part of the error reporting process.

These functions are also useful to generic applications that are written to adapt and take advantage of facilities that might be available from some, but not all database servers. The following DB2 CLI functions provide data source specific information:

- “SQLDataSources - Get List of Data Sources” on page 125
- “SQLGetFunctions - Get Functions” on page 208
- “SQLGetInfo - Get General Information” on page 213
- “SQLGetTypeInfo - Get Data Type Information” on page 238

Querying Environment Information Example

```
/**********************************************************/
/*  Querying environment and data source information      */
/**********************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlcli1.h>

void main()
{
    SQLHENV          hEnv;            /* Environment handle               */
    SQLHDBC          hDbc;            /* Connection handle                */
    SQLRETURN        rc;              /* Return code for API calls        */
    SQLHSTMT         hStmt;           /* Statement handle                 */
    SQLCHAR          dsname[30];      /* Data source name                 */
    SQLCHAR          dsdescr[255];    /* Data source description          */
    SQLSMALLINT      dslen;           /* Length of data source            */
    SQLSMALLINT      desclen;         /* Length of dsdescr                */
    BOOL             found = FALSE;
    SQLSMALLINT      funcs[100];
    SQLINTEGER       rgbValue;

    /*
     *  Initialize environment - allocate environment handle.
     */
    rc = SQLAllocEnv( &hEnv );
    rc = SQLAllocConnect( hEnv, &hDbc );

    /*
     *  Use SQLDataSources to verify MVSDB2 does exist.
     */
    while( ( rc = SQLDataSources( hEnv,
                            SQL_FETCH_NEXT,
                            dsname,
                            SQL_MAX_DSN_LENGTH+1,
                            &dslen,
                            dsdescr,
                            &desclen ) ) != SQL_NO_DATA_FOUND )
    {
       if( !strcmp( dsname, "MVSDB2" ) )   /* data source exist          */
       {
           found = TRUE;
           break;
       }
    }

    if( !found )
    {
        fprintf(stdout, "Data source %s does not exist...\n", dsname );
        fprintf(stdout, "program aborted.\n");
        exit(1);
    }
    if( ( rc = SQLConnect( hDbc, dsname, SQL_NTS, "myid", SQL_NTS, "mypd", SQL_NTS ) )
        == SQL_SUCCESS )
    {
       fprintf( stdout, "Connect to %s\n", dsname );
    }

    SQLAllocStmt( hDbc, &hStmt );

    /*
     *   Use SQLGetFunctions to store all APIs status.
     */
    rc = SQLGetFunctions( hDbc, SQL_API_ALL_FUNCTIONS, funcs );

    /*
     *   Check whether SQLGetInfo is supported in this driver. If so,
     *   verify whether DATE is supported for this data source.
     */
    if( funcs[SQL_API_SQLGETINFO] == 1 )
    {
        SQLGetInfo( hDbc, SQL_CONVERT_FUNCTIONS, (SQLPOINTER)&rgbValue, 255, &desclen );
        if( rgbValue & SQL_CVT_DATE )
        {
           SQLGetTypeInfo( hStmt, SQL_DATE );

           /*  use SQLBindCol and SQLFetch to retrieve data ....*/
        }
    }

}
```

Chapter 4. Configuring CLI and Running Sample Applications

This section provides information about installing DB2 CLI, the DB2 CLI runtime environment, and the preparation steps needed to run a DB2 CLI application.

- “Installing DB2 CLI”
- “DB2 CLI Runtime Environment” on page 50
- “Setting up DB2 CLI Runtime Environment” on page 52
- “Preparing a DB2 CLI Application” on page 55
- “DB2 CLI Initialization File” on page 62
- “Chapter 7. Problem Diagnosis” on page 381

Installing DB2 CLI

The steps in this section describe the SMP/E jobs you must edit and run to install DB2 CLI. Customize these jobs to specify data set names for you DB2 installation and SMP/E data sets. Refer to the header notes in each job and to Section 2 of *Installation Guide*for details.

Step 1: Copy and Edit the SMP/E Jobs

Use this sample JCL to invoke the MVS utility IEBCOPY to copy the SMP/E jobs to DASD.

```
//* COMPID:  DB2,5740XYR00
//* DOC:     LOAD CLI SMP INSTALLATION JCL FROM TAPE FOR DB2
//LOAD    EXEC PGM=IEBCOPY
//SYSPRINT DD SYSOUT=*
//JCLTAPE  DD DSN=IBM.HDB5510.F2,VOL=(PRIVATE,,SER=DB5510),
//           UNIT=TAPE,LABEL=(3,SL),DISP=(OLD,PASS)
//*
//JCLDISK DD DSN=SYSADM.JCL.CNTL,VOL=SER=USER01,UNIT=SYSDA,
//           DISP=OLD
//SYSUT3   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSIN    DD *
  COPY I=JCLTAPE,O=JCLDISK
  SELECT MEMBER=(DSNTCJAE)
  SELECT MEMBER=(DSNTCJAC,DSNTCJAP,DSNTCJRC)
//*
```

Figure 4. Sample JCL to Copy SMP/E jobs to DASD

Step 2: Run the Allocate Job: DSNTCJAE

DSNTCJAE allocates a new header data set; creates DDDEFs in target and distribution libraries for SCEELKED and CSSLIB data sets, and provides the SYSLIB concatentation in SMP needed for CALLLIB support.

Step 3: Run the Receive Job: DSNTCJRC

DSNTCJRC invokes SMP/E to receive the FMIDs for DB2 CLI into the SMP/E control data sets.

Step 4: Run the Apply Job: DSNTCJAP

DSNTCJAP invokes SMP/E to apply the FMIDs for DB2 CLI to the DB2 target libraries.

The DSNTCJAP job can include objects for Language Environment (LE) or Callable Services (CSS) libraries. If maintenance is later applied to LE or CSS libraries, you might need to run the SMP REPORT CALLLIBS command to generate a summary report. See *System Modification Program Extended (SMP/E) Reference* for detailed information.

## Step 5: Run the Accept Job: DSNTCJAC

DSNTCJAC invokes SMP/E to accept the FMIDs for DB2 CLI into the DB2 distribution libraries.

DB2 CLI Runtime Environment

DB2 CLI does not support an ODBC driver manager. All API calls are routed through the single CLI/ODBC driver that is loaded at run time into the application address space. DB2 CLI support is implemented as an IBM C/C^{++} Dynamic Load Library (DLL). By providing DB2 CLI support via a DLL, DB2 CLI applications do not need to linkedit any DB2 CLI driver code with the application load module. Instead, the linkage to the DB2 CLI APIs is resolved dynamically at runtime by the IBM Language Environment (LE) runtime support.

The DB2 CLI driver can use either the call attachment facility (CAF) or the Recoverable Resource Manager Services attachment facility (RRSAF) to connect to the DB2 for OS/390 address space.

- If the DB2 CLI application is not running as a DB2 for OS/390 stored procedure, the `MVSATTACHTYPE` keyword in the DB2 CLI initialization file determines the attachment facility that DB2 CLI uses.
- If the DB2 CLI application is running as a DB2 for OS/390 stored procedure, then DB2 CLI uses the attachment facility that was specified for stored procedures.

When the DB2 CLI application invokes the first ODBC function, `SQLAllocEnv()`, the DB2 CLI driver DLL is loaded and the application is connected to the DB2 for OS/390 subsystem.

DB2 CLI supports access to the local DB2 for OS/390 subsystems and any remote data source that is accessible via DB2 for OS/390 Version 5. This includes:

- Remote DB2 subsystems via specification of an alias or three-part name
- Remote DRDA-1 and DRDA-2 servers via LU 6.2 or TCP/IP.

The relationship between the application, the DB2 for OS/390 V5 CLI driver and the DB2 for OS/390 subsystem are illustrated in Figure 5 on page 51.

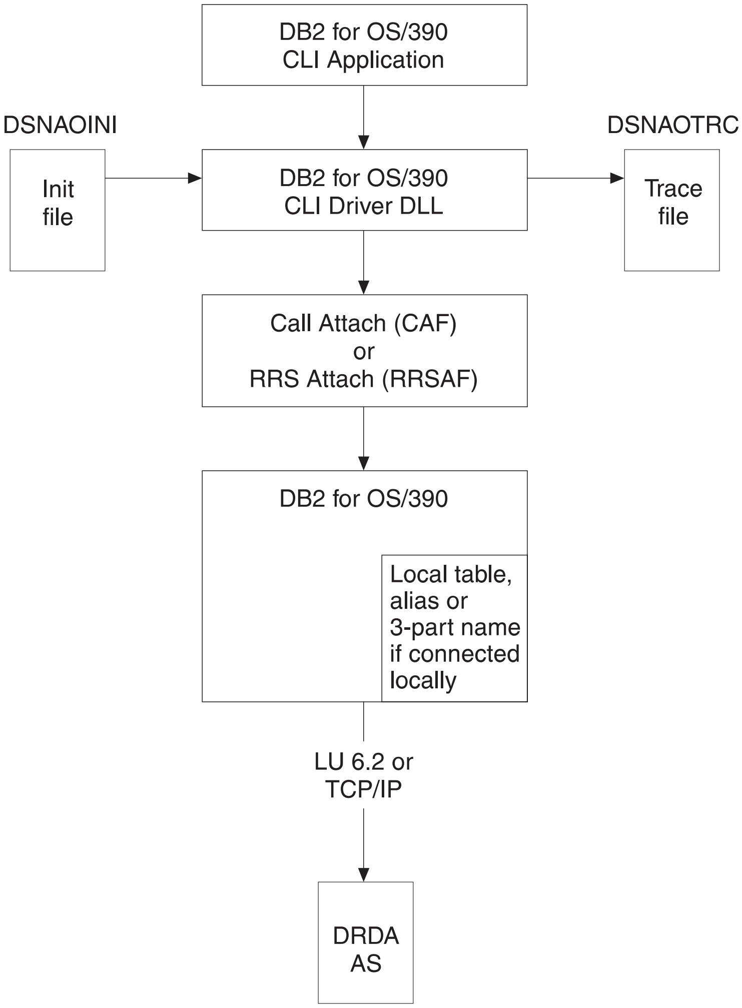

Figure 5. Relationship between DB2 for OS/390 V5 CLI components

Connectivity Requirements

DB2 for OS/390 V5 CLI has the following connectivity requirements:

- DB2 CLI applications must execute on a machine on which Version 5 of DB2 for OS/390 is installed.

# - If the application is executing with MULTICONTEXT=1, then there are multiple
# physical connections. Each connection corresponds to an independent
# transaction and DB2 thread.

| - If the application is executing CONNECT (Type 1) (described in "CONNECT
# Type 1 and Type 2" on page 26) and MULTICONTEXT=0, then there is only one current physical connection and one transaction on that connection. All transactions on logical connections (that is, with a valid connection handle) are rolled back by the application or committed by DB2 CLI. This is a deviation from the ODBC connection model.

Setting up DB2 CLI Runtime Environment

This section describes the general setup required to enable DB2 CLI applications. The steps in this section only need to be performed once, and are usually performed as part of the installation process for DB2 for OS/390.

The DB2 CLI bind files must be bound to the data source. The following two bind steps are required:

- Create packages at every data source
- Create at least one plan to name those packages.

These bind steps are described in the following sections:

- “Bind DBRMs to Packages”
- “Bind an Application Plan” on page 54

Special considerations for the OS/390 OpenEdition environment are described in the following section:

- “Setting up OS/390 OpenEdition Environment” on page 54

Bind DBRMs to Packages

For an application to access a data source using DB2 CLI, the following IBM supplied CLI DBRMs (shipped in DSN510.SDSNDBRM) must be bound to all data sources, including the local DB2 for OS/390 subsystem and all remote (DRDA) data sources.

- **DSNCLICS** bound with ISOLATION(CS)
- **DSNCLIRR** bound with ISOLATION(RR)
- **DSNCLIRS** bound with ISOLATION(RS)
- **DSNCLIUR** bound with ISOLATION(UR)
- **DSNCLINC** bound with ISOLATION(NC)
- **DSNCLIC1** bound with default options
- **DSNCLIC2** bound with default options
- **DSNCLIMS** bound with default options
- **DSNCLIVM** (only needed to access SQL/DS) bound with default options
- **DSNCLIAS** (only needed to access OS/400) bound with default options
- **DSNCLIV1** (only needed to access DB2 for common server Version 1) bound with default options
- **DSNCLIV2** (only needed to access DB2 for common server Version 2) bound with default options
- **DSNCLIQR** bound to any site that supports DRDA query result sets.
- **DSNCLIF4** bound with default options

Package Bind Options

For packages listed above that use the ISOLATION keyword, the impact of package bind options in conjunction with the DB2 CLI initialization file keywords is as follows:

- ISOLATION

  Packages must be bound with the isolation specified.

- DYNAMICRULES(BIND)

  Binding the packages with this option offers encapsulation and security similar to that of static SQL. The recommendations and consequences for using this option are as follows:

  1. Bind DB2 CLI packages or plan with DYNAMICRULES(BIND) from a `'driver'` authorization ID with table privileges.
  2. Issue GRANT EXECUTE on each collection.package or plan name to individual users. Plans are differentiated by plan name; packages are differentiated by collection.
  3. Select a plan or package by using the PLANNAME or COLLECTIONID keywords in the DB2 CLI initialization file.
  4. When dynamic SQL is issued, the statement is processed with the 'driver' authid. Users need execute privileges; table privileges are not required.
  5. The CURRENTSQLID keyword cannot be used in the DB2 CLI initialization file. Use of this keyword results in an error at `SQLConnect`.

- REOPT

  Do not use the REOPT keyword.

- SQLERROR(CONTINUE)

  Use this keyword to bind DB2 CLI to an earlier version of DB2 for OS/390 (prior to Version 5).

Attention: BIND issues a warning message if an attempt is made to use an unsupported function when binding to a DB2 for OS/390 release prior to Version 5.

Bind an Application Plan

A DB2 plan must be created using the `PKLIST` keyword to name all packages listed in “Bind DBRMs to Packages” on page 52. Any name can be selected for the plan; the default name is **DSNACLI**. If a name other than the default is selected, that name must be specified within the initialization file by using the `PLANNAME` keyword.

PLAN Bind Options

Use PLAN bind options as follows:

- DISCONNECT(EXPLICIT)

  All DB2 CLI plans are created using this option. DISCONNECT(EXPLICIT) is the default value; do not change it.

- CURRENTSERVER

  Do not specify this keyword when binding plans.

An online bind sample, DSNTIJCL, is available in DSN510.SDSNSAMP.

Binding Stored Procedures

A stored procedure running under DB2 CLI to a remote DB2 for OS/390, or another DBMS, does not need to be bound into the DB2 CLI plan; rather it can be bound as a package at the remote site.

For a stored procedure that resides on the local DB2 for OS/390, the stored procedure package must be bound in the DB2 CLI plan, using `PKLIST`. Stored procedures on remote servers only need to bind to that remote server as a package.

For example, DB2 CLI must always be bound in a plan to a DB2 for OS/390 subsystem to which DB2 CLI first establishes an affinity on the `SQLAllocEnv()` call. This is the local DB2. The scenario in this example is equivalent to specifying the `MVSDEFAULTSSID` keyword in the initialization file. If DB2 CLI calls a stored procedure that resides at this local DB2 for OS/390, that stored procedure package must be in the DB2 CLI plan, using `PKLIST`.

This process is unique to DB2 for OS/390 stored procedure support.

Setting up OS/390 OpenEdition Environment

To use DB2 CLI in the OS/390 OpenEdition environment, the DB2 CLI definition side-deck must be available to OpenEdition users.

The OpenEdition compiler determines the contents of an input file based on the file extension. In the case of a file residing in an MVS partitioned data set (PDS), the last qualifier in the PDS name is treated as the file extension.

The OpenEdition compiler recognizes the DB2 CLI definition side-deck by these criteria:

- It must reside in an MVS PDS
- The last qualifier in the PDS name must be `.EXP`

Therefore, to make the DB2 CLI definition side-deck available to OpenEdition users, you should define an MVS data set alias that uses `.EXP` as the last qualifier in the name. This alias should relate to the SDSNMACS data set which is where the DB2 CLI definition side-deck is installed.

For example, assume that DB2 is installed using DSN510 as the high level data set qualifier. You can define the alias using the following command:

```
DEFINE ALIAS(NAME('DSN510.SDSNC.EXP') RELATE('DSN510.SDSNMACS'))
```

This alias allows OpenEdition users to directly reference the DB2 CLI definition side-deck by specifying:

```
"//'DSN510.SDSNC.EXP(DSNAOCLI)'"
```

as one of the input files to the OpenEdition c89 command.

Preparing a DB2 CLI Application

This section provides an overview of the DB2 CLI components and explains the steps you follow to prepare a DB2 CLI application.

Figure 6 on page 56 shows the DB2 CLI components used to build the DB2 CLI DLL, and the process you follow to install and prepare a DB2 CLI application. The shaded areas identify the components that are shipped.

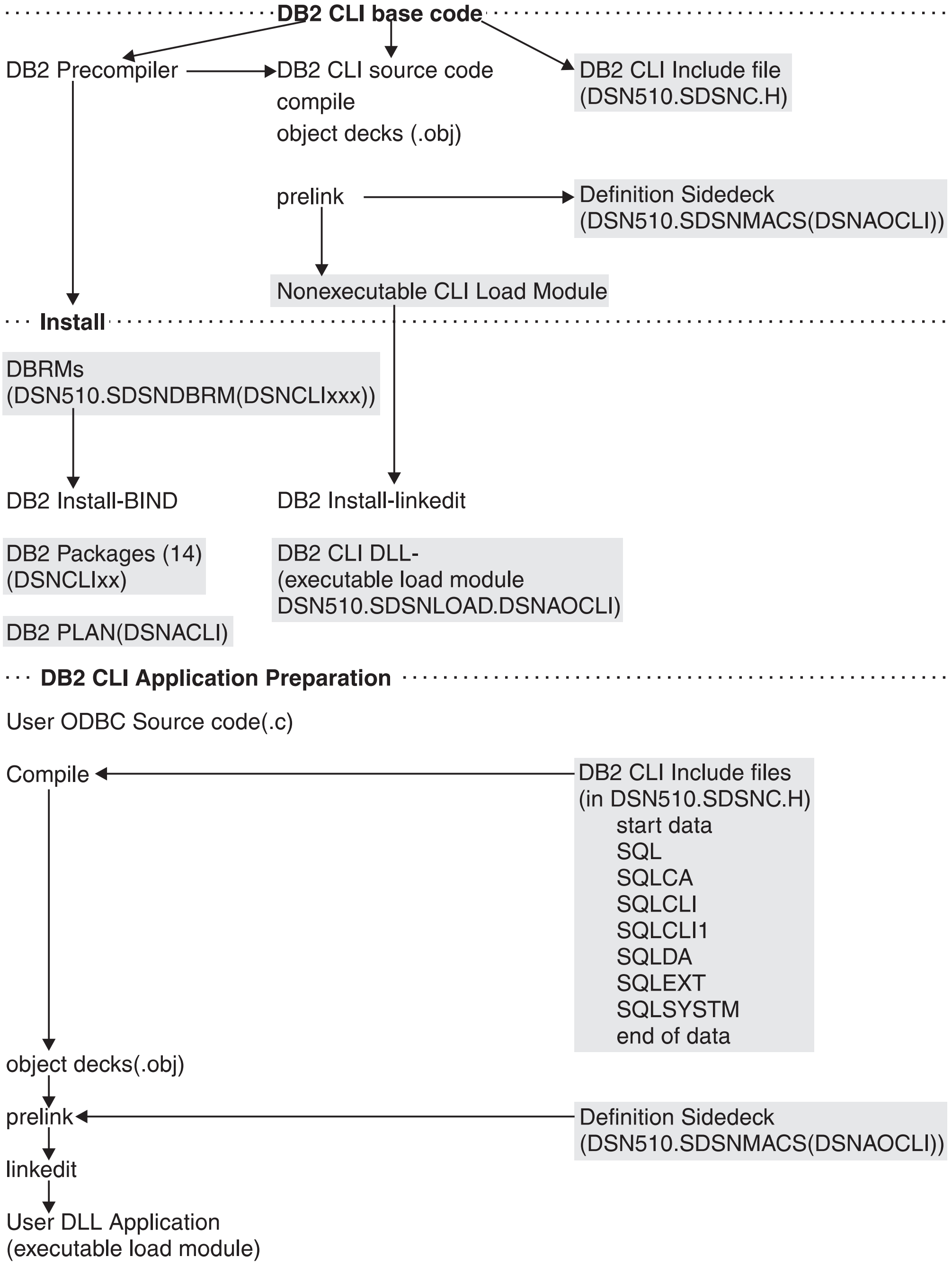

Figure 6. DB2 CLI Application Development and Execution

The following sections describe the requirements and steps that are necessary to run a DB2 CLI application.

- “DB2 CLI Application Requirements”
- “Application Execution”

DB2 CLI Application Requirements

To successfully build a DLL application, you must ensure that the correct compile, prelink, and linkedit options are used. In particular, your application must generate the appropriate DLL linkage for the exported DB2 CLI DLL functions.

The C^{++} compiler always generates DLL linkage. However, the C compiler only generates DLL linkage if the DLL compile option is used. Failure to generate the necessary DLL linkage can cause the prelinker and linkage editor to issue warning messages for unresolved references to DB2 CLI functions.

The minimum requirements for a DB2 CLI application are as follows:

- Compiler: IBM C/C^{++} for MVS/ESA Version 3, Release 1, or subsequent releases.

  If the C compiler is used, then the DLL compiler option must be specified.

- Language runtime support: IBM Language Environment Version 1, Release 5, or subsequent releases.
- The DB2 CLI application must be written and linkedited to execute with a 31-bit addressing mode, AMODE(31).

Special Considerations for OS/390 OpenEdition

A special consideration applies to DB2 CLI product data set access. If you build a DB2 CLI application in OS/390 OpenEdition, you can use the `c89` compile command to compile your application. Even though you compile your application under OpenEdition, you can directly reference the non-HFS DB2 CLI data sets in the `c89` command. There is no need to copy the DB2 CLI product files to HFS.

Application Execution

The following steps describe application preparation and execution:

- “Step 1. Compile the Application”
- “Step 2. Prelink and Linkedit the Application” on page 59
- “Step 3. Execute the Application” on page 61

Step 1. Compile the Application

Include the following statement in your DB2 CLI application:

```
#include <sqlcli1.h>
```

The `sqlcli1.h` file includes all information that is required for compiling your DB2 CLI application. All DB2 CLI header files, including `sqlcli1.h`, that define the function prototypes, constants, and data structures that are needed for a DB2 CLI application are shipped in the `DSN510.SDSNC.H` data set. Therefore, you must add this dataset to your `SYSPATH` concatenation when you compile your DB2 CLI application.

For example, your compile job might look like:

```
//*
//* Sample CLI application compile job
//*  This shows sample JCL to compile a CLI application program.
//*  In this sample, the CLI application consists of a single C source
//*  part (PDS member name CLIAPPL) and associated header files.
//*  This sample assumes that the user is using the following data set
//*  names:
//*
//*    USER01.MYPROG.SOURCE.C  - CLI application source code (PDS)
//*    USER01.MYPROG.INCLUDE.H - CLI application header files (PDS)
//*    USER01.MYPROG.OBJ       - CLI application OBJ (PDS)
//*    USER01.RUNLIB.LOAD      - User's load libary (PDS)
//*    CBC.V3R1M0              - High level qualifier for C/C++ V3.1
//*    CEE.V1R5M0              - High level qualifier for LE 1.5
//*    DSN510                  - High level qualifier for DB2 V5
//*
//COMPILE EXEC PGM=CBC310PP,REGION=32M,
//    PARM=('/OPTFILE(DD:CCOPT)')
//STEPLIB  DD  DSN=CEE.V1R5M0.SCEERUN,DISP=SHR
//         DD  DSN=CBC.V3R1M0.SCBC3CMP,DISP=SHR
//SYSMSGS  DD  DSN=CBC.V3R1M0.SCBC3MSG(EDCMSGE),DISP=SHR
//SYSXMSGS DD  DSN=CBC.V3R1M0.SCBC3MSG(CBCMSGE),DISP=SHR
//SYSIN    DD  DSN=USER01.MYPROG.SOURCE.C(CLIAPPL),DISP=SHR
//SYSLIN   DD  DSN=USER01.MYPROG.OBJ(CLIAPPL),DISP=SHR,
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSCPRT  DD  SYSOUT=*
//CCOPT    DD *
  USERPATH(/USER01/MYPROG/INCLUDE)
  SYSPATH(/CEE/V1R5M0/SCEEH,/CBC/V3R1M0/SCLB3H,/DSN510/SDSNC)
  LIST
  SOURCE
  LONG
//*
//SYSUT1   DD  UNIT=VIO,SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT4   DD  UNIT=VIO,SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT5   DD  UNIT=VIO,SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT6   DD  UNIT=VIO,SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT7   DD  UNIT=VIO,SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT8   DD  UNIT=VIO,SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT9   DD  UNIT=VIO,SPACE=(32000,(30,30)),
//             DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SYSUT10  DD  SYSOUT=*
//SYSUT14  DD  UNIT=VIO,SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT15  DD  SYSOUT=*
```

Compiling in OS/390 OpenEdition: If you build a DB2 CLI application in OS/390 OpenEdition, you can use the `c89` command to compile your application. For example, to compile a C application named `'myapp.c'` that resides in the current working directory, the `c89` compile command might look like:

```
c89 -c -W 'c,dll,long,source,list -
       -I"//'DSN510.SDSNC.H'" \
       myapp.c
```

Alternatively, if you write an application in C^{++}, the `cxx` command might look like:

```
cxx -c -W 'c,long,source,list' -
      -I"//'DSN510.SDSNC.H'" \
      myapp.C
```

If your source code is in C, rather than C^{++}, you must compile using the `'dll'` option to enable use of the DB2 CLI driver. This is a requirement even when using the `cxx` compile command to compile C parts.

Step 2. Prelink and Linkedit the Application

Before you can linkedit your DB2 CLI application, you must prelink your application with the DB2 CLI definition side-deck provided with Version 5 of DB2 for OS/390.

The definition side-deck defines all of the exported functions in the DB2 CLI dynamic load library, `DSNAOCLI`. It resides in the `DSN510.SDSNMACS` data set, as member `DSNAOCLI`. The definition side-deck should also be available under the alias data set name of `DSN510.SDSNC.EXP` as member `DSNAOCLI` (see “Setting up OS/390 OpenEdition Environment” on page 54 for details). You must include the `DSNAOCLI` member as input to the Prelinker by specifying it in the prelink `SYSIN` DD card concatenation. For example, your prelink and linkedit job might look like:

```
//*
//* Sample CLI application prelink and link-edit job
//*  This shows sample JCL to prelink and link-edit a CLI application
//*  program.  In this sample, the CLI application consists of
//*  a single C obj.  This sample assumes that the user is using the
//*  following data set names:
//*
//*    USER01.MYPROG.SOURCE.C  - CLI application source code (PDS)
//*    USER01.MYPROG.INCLUDE.H - CLI application header files (PDS)
//*    USER01.MYPROG.OBJ       - CLI application OBJ (PDS)
//*    USER01.RUNLIB.LOAD      - User's load libary (PDS)
//*    CBC.V3R1M0              - High level qualifier for C/C++ V3.1
//*    CEE.V1R5M0              - High level qualifier for LE 1.5
//*    DSN510                  - High level qualifier for DB2 V5
//*
//*--------------------------------------------------------------
//* PRE-LINKEDIT STEP:
//*--------------------------------------------------------------
//PLKED   EXEC PGM=EDCPRLK,REGION=2048K,
//        PARM='MAP'
//STEPLIB  DD  DSN=CEE.V1R5M0.SCEERUN,DISP=SHR
//SYSMSGS  DD  DSN=CEE.V1R5M0.SCEEMSGP(EDCPMSGE),DISP=SHR
//SYSLIB   DD  DSN=CEE.V1R5M0.SCEECPP,DISP=SHR
//SYSIN    DD  DSN=USER01.MYPROG.OBJ(CLIAPPL),DISP=SHR
//         DD  DSN=CBC.V3R1M0.SCLB3SID(IOSTREAM),DISP=SHR
//         DD  DSN=CBC.V3R1M0.SCLB3SID(COMPLEX),DISP=SHR
//         DD  DSN=CBC.V3R1M0.SCLB3SID(APPSUPP),DISP=SHR
//         DD  DSN=CBC.V3R1M0.SCLB3SID(COLLECT),DISP=SHR
//         DD  DSN=DSN510.SDSNMACS(DSNAOCLI),DISP=SHR
//SYSMOD   DD  DSN=&&PLKSET,UNIT=VIO,DISP=(NEW,PASS),
//             SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSDEFSD DD  DUMMY
//SYSOUT   DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//*
//*--------------------------------------------------------------------
//* LINKEDIT STEP:
//*--------------------------------------------------------------------
//LKED   EXEC PGM=HEWL,REGION=1024K,COND=(8,LE,PLKED),
//       PARM='AMODE=31,MAP'
//SYSLIB   DD  DSN=CEE.V1R5M0.SCEELKED,DISP=SHR
//SYSLIN   DD  DSN=*.PLKED.SYSMOD,DISP=(OLD,DELETE)
//SYSLMOD  DD  DSN=USER01.RUNLIB.LOAD(CLIAPPL),DISP=SHR
//SYSUT1   DD  UNIT=VIO,SPACE=(32000,(30,30))
//SYSPRINT DD  SYSOUT=*
```

For more information about DLL, refer to *IBM C/C++ for MVS/ESA Programming Guide*

Prelinking and Linkediting in OS/390 OpenEdition: If you build a DB2 CLI application in OS/390 OpenEdition, you can use the `c89` command to prelink and linkedit your application. You need to include the DB2 CLI definition side-deck as one of the input data sets to the `c89` command and specify `'dll'` as one of the linkedit options.

For example, assume that you have already compiled a C application named `'myapp.c'` to create a `'myapp.o'` file in the current working directory. The `c89` command to prelink and linkedit your application might look like:

```
c89  -W l,p,map,noer  -W l,dll,AMODE=31,map \
       -o myapp myapp.o //'DSN510.SDSNC.EXP(DSNAOCLI)'
```

Step 3. Execute the Application

DB2 CLI applications must access the DSN510.SDSNLOAD data set at execution time. The SDSNLOAD data set contains both the DB2 CLI dynamic load library and the attachment facility used to communicate with DB2.

In addition, the DB2 CLI driver accesses the DB2 for OS/390 load module DSNHDECP. DSNHDECP contains, among other things, the coded character set ID (CCSID) information that DB2 for OS/390 uses.

A default DSNHDECP is shipped with DB2 for OS/390 in the DSN510.SDSNLOAD data set. However, if the values provided in the default DSNHDECP are not appropriate for your site, a new DSNHDECP can be created during the installation of DB2 for OS/390. If a site specific DSNHDECP is created during installation, you should concatenate the data set containing the new DSNHDECP before the DSN510.SDSNLOAD data set in your STEPLIB or JOBLIB DD card.

For example, your execute job might look like:

```
//*
//* Sample CLI application execution job:
//*  This shows sample JCL to execute a CLI application program.
//*  In this sample, the CLI application does not have any input
//*  parameters and does not require any input from the user.
//*  This sample assumes that the user is using the following data set
//*  names:
//*
//*    USER01.MYPROG.SOURCE.C  - CLI application source code (PDS)
//*    USER01.MYPROG.INCLUDE.H - CLI application header files (PDS)
//*    USER01.MYPROG.OBJ       - CLI application OBJ (PDS)
//*    USER01.RUNLIB.LOAD      - User's load libary (PDS)
//*    USER01.MYINI            - User's CLI initialization file
//*    CBC.V3R1M0              - High level qualifier for C/C++ V3.1
//*    CEE.V1R5M0              - High level qualifier for LE 1.5
//*    DSN510                  - High level qualifier for DB2 V5
//*
//CLIAPPL  EXEC PGM=CLIAPPL
//STEPLIB  DD DSN=USER01.RUNLIB.LOAD,DISP=SHR
//         DD DSN=DSN510.SDSNEXIT,DISP=SHR
//         DD DSN=DSN510.SDSNLOAD,DISP=SHR
//         DD DSN=CEE.V1R5M0.SCEERUN,DISP=SHR
//         DD DSN=CBCV3R10.SCLB3DLL,DISP=SHR
//SYSPRINT DD SYSOUT=*
//CEEDUMP  DD SYSOUT=*
//DSNAOINI DD DSN=USER01.MYINI,DISP=SHR
//
```

Executing in OS/390 OpenEdition: To execute a DB2 CLI application in OS/390 OpenEdition, you need to include the DSN510.SDSNEXIT and DSN510.SDSNLOAD data sets in the data set concatenation of your STEPLIB environmental variable. The STEPLIB environmental variable can be set in your .profile with the statement:

```
export STEPLIB=DSN510.SDSNEXIT:DSN510.SDSNLOAD
```

Defining a Subsystem

There are two ways to define a DB2 subsystem to DB2 CLI. You can identify the DB2 subsystem by specifying the MVSDEFAULTSSID keyword in the common section of initialization file. If the MVSDEFAULTSSID keyword does not exist in the initialization file, DB2 CLI uses the default subsystem name specified in the DSNHDECP load module that was created when DB2 was installed. Therefore, you should ensure that DB2 CLI can find the intended DSNHDECP when your application issues the SQLAllocEnv call.

The DSNHDECP load module is usually linkedited into the DSN510.SDSNEXIT data set. In this case, your STEPLIB DD card includes:

```
//STEPLIB    DD  DSN=DSN510.SDSNEXIT,DISP=SHR
//           DD  DSN=DSN510.SDSNLOAD,DISP=SHR
...
```

DB2 CLI Initialization File

A set of optional keywords can be specified in a DB2 CLI *initialization file*, an EBCDIC file that stores default values for various DB2 CLI configuration options. Because the initialization file has EBCDIC text, it can be updated using a file editor, such as the TSO editor.

For most applications, use of the DB2 CLI initialization file is not necessary. However, to make better use of IBM RDBMS features, the keywords can be specified to:

- Help improve the performance or usability of an application.
- Provide support for applications written for a previous version of DB2 CLI.
- Provide specific work-arounds for existing ODBC applications.

The following sections describe how to create the initialization file and define the keywords:

- “Using the Initialization File”
- “Initialization Keywords” on page 64

Using the Initialization File

The DB2 CLI initialization file is read at application run-time. The file can be specified by either a DSNAOINI DD card or by defining a DSNAOINI OpenEdition environmental variable. The initialization file specified can be either a tradition MVS data set or an OpenEdition HFS file. For MVS data sets, the record format of the initialization file can be either fixed or variable length.

The following JCL examples use a DSNAOINI JCL DD card to specify the DB2 CLI initialization file types supported:

MVS sequential data set USER1.DB2CLI.CLIINI:

```
//DSNAOINI DD DSN=USER1.DB2CLI.CLIINI,DISP=SHR
```

MVS partitioned data set USER1.DB2CLI.DATA, member CLIINI:

```
//DSNAOINI DD DSN=USER1.DB2CLI.DATA(CLIINI),DISP=SHR
```

Inline JCL DSNAOINI DD specification:

```
//DSNAOINI DD *
  ﬄCOMMON“
  MVSDEFAULTSSID=V51A
/*
```

HFS file /u/user1/db2cli/cliini:

```
//DSNAOINI DD PATH='/u/user1/db2cli/cliini'
```

The following examples of OpenEdition export statements define the DB2 CLI DSNAOINI OpenEdition environmental variable for the DB2 CLI initialization file types supported:

HFS fully qualified file /u/user1/db2cli/cliini:

```
export DSNAOINI="/u/user1/db2cli/cliini"
```

HFS file ./db2cli/cliini, relative to the present working directory of the application:

```
export DSNAOINI="./db2cli/cliini"
```

MVS sequential data set USER1.CLIINI:

```
export DSNAOINI="USER1.CLIINI"
```

Redirecting to use a file specified by another DD card, MYDD, that is already allocated:

```
export DSNAOINI="//DD:MYDD"
```

MVS partitioned data set USER1.DB2CLI.DATA, member CLIINI:

```
export DSNAOINI="USER1.DB2CLI.DATA(CLIINI)"
```

When specifying an HFS file, the value of the DSNAOINI environmental variable must begin with either a single forward slash (/), or a period followed by a single forward slash (./). If a setting starts with any other characters, DB2 CLI assumes that an MVS data set name is specified.

Initialization File Structure

The initialization file consists of the following three sections, or stanzas:

Common section Contains parameters that are global to all applications using this initialization file.

Subsystem section Contains parameter values unique to that subsystem.

Data Source sections
Contain parameter values to be used only when connected to that data source. You can specify zero or more data source sections.

Each section is identified by a syntactic identifier enclosed in square brackets. Specific guidelines for coding square brackets are described in the list item below marked 'Common errors'.

The syntactic identifier is either the literal 'common', the subsystem ID or the data source (location name). For example:

```
[data-source-name]
```

This is the *section header*.

The parameters are set by specifying a keyword with its associated keyword value in the form:

```
KeywordName =keywordValue
```

- All the keywords and their associated values for each data source must be located below the data source section header.
- The keyword settings in each section apply only to the data source name in that section header.

- The keywords are **not** case sensitive; however, their values can be if the values are character based.
- For the syntax associated with each keyword, refer to “Initialization Keywords.”
- If a data source name is not found in the DB2 CLI initialization file, the default values for these keywords are in effect.
- Comment lines are introduced by having a semi-colon in the first position of a new line.
- Blank lines are also permitted. If duplicate entries for a keyword exist, the first entry is used (and no warning is given).
- **Common errors:** You can avoid common errors by ensuring that the following contents of the initialization file are accurate:
  - Square brackets: The square brackets in the initialization file must consist of the correct EBCDIC characters. The open square bracket must use the hexadecimal characters `X'AD'`. The close square bracket must use the hexadecimal characters `X'BD'`. DB2 CLI does not recognize brackets if coded differently.
  - Sequence numbers: The initialization file cannot accept sequence numbers. All sequence numbers must be removed.

The following is a sample DB2 CLI initialization file with a common stanza, a subsystem stanza, and two data source stanzas.

```
; This is a comment line...
; Example COMMON stanza
[COMMON]
MVSDEFAULTSSID=V51A

; Example SUBSYSTEM stanza for V51A subsystem
[V51A]
MVSATTACHTYPE=RRSAF
PLANNAME=DSNACLI

; Example DATA SOURCE stanza for STLEC1 data source
[STLEC1]
AUTOCOMMIT=0
CONNECTTYPE=2

; Example DATA SOURCE stanza for STLEC1B data source
[STLEC1B]
CONNECTTYPE=2
CURSORHOLD=0
```

Initialization Keywords

The initialization keywords are described in this section. The section (common, SUBSYTEM, or data source) in which each keyword must be defined is identified.

AUTOCOMMIT = 1 | 0

This keyword is placed in the data source section.

To be consistent with ODBC, DB2 CLI defaults with AUTOCOMMIT on, which means each statement is treated as a single, complete transaction. This keyword can provide an alternative default, but is only used if the application does not specify a value for AUTOCOMMIT as part of the program.

1 = on (default)

0 = off

Most ODBC applications assume the default of AUTOCOMMIT is on. Extreme care must be used when overriding this default during runtime as the application might depend on this default to operate properly.

This keyword also allows you to specify whether autocommit should be enabled in a distributed unit of work (DUW) environment. If a connection is part of a coordinated DUW, and AUTOCOMMIT is not set, the default does not apply; implicit commits arising from autocommit processing are suppressed. If AUTOCOMMIT is set to 1, and the connection is part of a coordinated DUW, the implicit commits are processed. This can result in severe performance degradations, and possibly other unexpected results elsewhere in the DUW system. However, some applications might not work at all unless this is enabled.

A thorough understanding of the transaction processing of an application is necessary, especially applications written by a third party, before applying it to a DUW environment.

BITDATA = 1 | 0

This keyword is placed in the data source section.

The `BITDATA` keyword allows you to specify whether ODBC binary data types, SQL_BINARY, SQL_VARBINARY, and SQL_LONGVARBINARY, are reported as binary type data. IBM DBMSs support columns with binary data types by defining CHAR, VARCHAR and LONG VARCHAR columns with the `FOR BIT DATA` attribute.

Only set BITDATA = 0 if you are sure that all columns defined as `FOR BIT DATA` contain only character data, and the application is incapable of displaying binary data columns.

1 = report `FOR BIT DATA` data types as binary data types (default).
0 = disabled.

CLITRACE = 0 | 1

This keyword is placed in the common section.

The `CLITRACE` keyword controls whether the DB2 CLI **application trace** is enabled. If enabled, every call to any DB2 CLI API from the application is traced, including input parameters. The trace is written to the file specified on the `TRACEFILENAME` keyword.

0 = disabled (default)
1 = enabled

For more information about using the `CLITRACE` keyword, see “Use of Trace Keywords” on page 397.

COLLECTIONID = *collection_id*

This keyword is placed in the data source section.

The `COLLECTIONID` keyword allows you to specify the collection identifier that is used to resolve the name of the package allocated at the server. This package supports the execution of subsequent SQL statements.

The value is a character string and must not exceed 18 characters. It can be overridden by executing the SET CURRENT PACKAGESET statement.

CONNECTTYPE = 1 | 2

This keyword is placed in the common section.

The `CONNECTTYPE` keyword allows you to specify the default connect type for all connections to data sources.

- 1 = Multiple concurrent connections, each with its own commit scope. If `MULTICONTEXT=0` is specified, a new connection might not be added unless the current transaction on the current connection is on a transaction boundary (either committed or rolled back).
- 2 = Coordinated connections where multiple data sources participate under the same distributed unit of work. `CONNECTTYPE=2` is ignored if `MULTICONTEXT=1`is specified.

CURRENTSQLID = *current_sqlid*

This keyword is placed in the data source section.

The `CURRENTSQLID` keyword is valid only for those DB2 DBMSs that support SET CURRENT SQLID (such as DB2 for OS/390). If this keyword is present, then a SET CURRENT SQLID statement is sent to the DBMS after a successful connect. This allows the end user and the application to name SQL objects without having to qualify by schema name.

Do not specify this keyword if you are binding the DB2 CLI packages with DYNAMICRULES(BIND).

CURSORHOLD = 1 | 0

This keyword is placed in the data source section.

The `CURSORHOLD` keyword controls the effect of a transaction completion on open cursors.

1 = Cursor hold (default). The cursors are not destroyed when the transaction is committed.

0 = Cursor no hold. The cursors are destroyed when the transaction is committed.

Cursors are always destroyed when transactions are rolled back.

This keyword can be used by an end user to improve performance. If the user is sure that the application:

1. Does not have behavior that is dependent on the SQL_CURSOR_COMMIT_BEHAVIOR or the SQL_CURSOR_ROLLBACK_BEHAVIOR information returned via `SQLGetInfo()`, and
2. Does not require cursors to be preserved from one transaction to the next,

then the value of this keyword can be set to **0**. The DBMS operates more efficiently as resources no longer need to be maintained after the end of a transaction.

DBNAME = *dbname*

This keyword is placed in the data source section.

The `DBNAME` keyword is only used when connecting to DB2 for OS/390, and only if (*base*) table catalog information is requested by the application.

If a large number of tables exist in the DB2 for OS/390 subsystem, a *dbname* can be specified to reduce the time it takes for the database to process the catalog query for table information, and reduce the number of tables returned to the application.

The value of the *dbname* keyword maps to the `DBNAME` column in the DB2 for OS/390 system catalog tables. If no value is specified, or if views, synonyms, system tables, or aliases are also specified via TABLETYPE, only table information is restricted; views, aliases, and synonyms are not restricted with DBNAME. This keyword can be used in conjunction with SCHEMALIST and TABLETYPE to further limit the number of tables for which information is returned.

GRAPHIC =0 | 1 | 2 | 3

This keyword is placed in the data source section.

The `GRAPHIC` keyword controls whether DB2 CLI reports IBM GRAPHIC (double byte character support) as one of the supported data types when `SQLGetTypeInfo()` is called. `SQLGetTypeInfo()` lists the data types supported by the data source for the current connection. These are not native ODBC types but have been added to expose these types to an application connected to a DB2 family product.

0 = disabled (default)

1 = enabled

2 = report the length of graphic columns returned by DESCRIBE in number of bytes rather than DBCS characters. This applies to all DB2 CLI and ODBC functions that return length or precision either on the output argument or as part of the result set.

3 = settings 1 and 2 combined; that is, **GRAPHIC=3** achieves the combined effect of 1 and 2.

The default is that GRAPHIC is not returned since many applications do not recognize this data type and cannot provide proper handling.

MAXCONN = 0 | positive number

This keyword is placed in the common section.

The `MAXCONN` keyword is used to specify the maximum number of connections allowed for each CLI application program. This can be used by an administrator as a governor for the maximum number of connections established by each application. A value of 0 can be used to represent *no limit*; that is, an application is allowed to open up as many connections as permitted by the system resources.

Note that this parameter limits the number of **SQLConnect** statements that the application can successfully issue. In addition, if the application is executing with CONNECT (Type 1) semantics, then this value specifies the number of logical connections. There is only one physical connection to either the local DB2/MVS subsystem or a remote DB2 subsystem or remote DRDA-1 or DRDA-2 server.

MULTICONTEXT = 0 | 1

This keyword is placed in the common section.

The `MULTICONTEXT` keyword controls whether each connection in an application can be treated as a separate unit of work with its own commit scope that is independent of other connections.

> 0 = The DB2 CLI code does not create an independent *context* for a data source connection. Connection switching among multiple data sources governed by the `CONNECTTYPE=1` rules is not allowed unless the current transaction on the current connection is on a transaction boundary (either committed or rolled back). This is the default.

> 1 = The DB2 CLI code creates an independent context for a data source connection at the connection handle level when `SQLAllocConnect()` is issued. Each connection to multiple data sources is governed by `CONNECTTYPE=1` rules and is associated with an independent DB2 thread. Connection switching among multiple data sources is not prevented due to the commit status of the transaction; an application can use multiple connection handles without having to perform a commit or rollback on a connection before switching to another connection handle. The use of `MULTICONTEXT=1` requires `MVSATTACHTYPE=RRSAF` and OS/390 Version 2 Release 5 or higher.

The application can use `SQLGetInfo()` with *finfoType*=`SQL_MULTIPLE_ACTIVE_TXN` to determine whether `MULTICONTEXT=1` is supported.

`MULTICONTEXT=1` is ignored if any of these conditions are true:

- The CLI application created a DB2 thread before invoking DB2 CLI. This is always the case for a stored procedure using DB2 CLI.
- The CLI application created and switched to a private context using OS/390 Context Services before invoking DB2 CLI.
- The CLI application started a unit of recovery with any RRS resource manager (for example, IMS) before invoking DB2 CLI.
- `MVSATTACHTYPE=CAF` is specified in the initialization file.
- The OS/390 operating system level does not support Unauthorized Context Services.

MVSATTACHTYPE = CAF | RRSAF

This keyword is placed in the subsystem section.

The `MVSATTACHTYPE` keyword is used to specify the DB2 for OS/390 attachment type that DB2 CLI uses to connect to the DB2 for OS/390 address space. This parameter is ignored if the DB2 CLI application is running as a DB2 for OS/390 stored procedure. In that case, DB2 CLI uses the attachment type that was defined for the stored procedure.

> CAF: DB2 CLI uses the DB2 for OS/390 call attachment facility (CAF).

> RRSAF: DB2 CLI uses the DB2 for OS/390 Recoverable Resource Manager Services attachment facility (RRSAF).

MVSDEFAULTSSID = *ssid*

This keyword is placed in the common section.

The `MVSDEFAULTSSID` keyword specifies the default DB2 subsystem to which the application is connected when invoking the `SQLAllocEnv` function. You must specify a four character name of an installed DB2 subsystem.

OPTIMIZEFORNROWS = *integer*

This keyword is placed in the data source section.

The `OPTIMIZEFORNROWS` keyword appends the "OPTIMIZE FOR n ROWS" clause to every select statement, where n is an integer larger than 0. The default action is not to append this clause.

For more information on the effect of the OPTIMIZE FOR n ROWS clause, refer to *SQL Reference*.

PLANNAME = *planname*

This keyword is placed in the subsystem section.

The `PLANNAME` keyword specifies the name of the DB2 for OS/390 PLAN that was created during installation. A PLAN name is required when initializing the application connection to the DB2 for OS/390 subsystem which occurs during the processing of the **SQLAllocEnv** call.

If no PLANNAME is specified, the default value **DSNACLI** is used.

SCHEMALIST = "'*schema1***', '***schema2***' ,..."**

This keyword is placed in the data source section.

The `SCHEMALIST` keyword specifies a list of schemas in the data source.

If there are a large number of tables defined in the database, a schema list can be specified to reduce the time it takes for the application to query table information, and reduce the number of tables listed by the application. Each schema name is case sensitive, must be delimited with single quotes and separated by commas. The entire string must also be enclosed in double quotes, for example:

```
SCHEMALIST="'USER1','USER2',USER3'"
```

For DB2 for OS/390, CURRENT SQLID can also be included in this list, but without the single quotes, for example:

```
SCHEMALIST="'USER1',CURRENT SQLID,'USER3'"
```

The maximum length of the keyword string is 256 characters.

This keyword can be used in conjunction with DBNAME and TABLETYPE to further limit the number of tables for which information is returned.

SCHEMALIST is used to provide a more restrictive default in the case of those applications that always give a list of every table in the DBMS. This improves performance of the table list retrieval in cases where the user is only interested in seeing the tables in a few schemas.

SYSSCHEMA = *sysschema*

This keyword is placed in the data source section.

The `SYSSCHEMA` keyword indicates an alternative schema to be searched in place of the SYSIBM (or SYSTEM, QSYS2) schemas when the DB2 CLI and ODBC catalog function calls are issued to obtain system catalog information.

Using this schema name, the system administrator can define a set of views consisting of a subset of the rows for each of the following system catalog tables:

DB2 for common server	DB2 for OS/390	DB2 for VSE and VM	OS/400	DB2 for OS/400
SYSTABLES	SYSTABLES	SYSCATALOG	SYSTABLES	SYSTABLES
SYSCOLUMNS	SYSCOLUMNS	SYSCOLUMNS	SYSCOLUMNS	SYSCOLUMNS
SYSINDEXES	SYSINDEXES	SYSINDEXES	SYSINDEXES	SYSINDEXES
SYSTABAUTH	SYSTABAUTH	SYSTABAUTH		SYSCST
SYSRELS	SYSRELS	SYSKEYCOLS		SYSKEYCST
SYSDATATYPES	SYSSYNONYMS	SYSSYNONYMS		SYSCSTCOL
	SYSKEYS	SYSKEYS		SYSKEYS
	SYSCOLAUTH	SYSCOLAUTH		SYSREFCST
	SYSFOREIGNKEYS			
	SYSPROCEDURES			
	SYSDATABASE			

For example, if the set of views for the system catalog tables are in the ACME schema, then the view for SYSIBM.SYSTABLES is ACME.SYSTABLES; and SYSSCHEMA should then be set to ACME.

Defining and using limited views of the system catalog tables reduces the number of tables listed by the application, which reduces the time it takes for the application to query table information.

If no value is specified, the default is:

- SYSIBM on DB2 for OS/390 and OS/400
- SYSTEM on DB2 for VSE and VM
- QSYS2 on DB2 for OS/400

This keyword can be used in conjunction with SCHEMALIST, TABLETYPE (and DBNAME on DB2 for OS/390) to further limit the number of tables for which information is returned.

TABLETYPE="'TABLE' | ,'ALIAS' | ,'VIEW' | , ' | , 'SYSTEM TABLE' | ,'SYNONYM'"

This keyword is placed in the data source section.

The TABLETYPE keyword specifies a list of one or more table types. If there are a large number of tables defined in the data source, a table type string can be specified to reduce the time it takes for the application to query table information, and reduce the number of tables listed by the application.

Any number of the values can be specified, but each type must be delimited with single quotes, separated by commas, and in upper case. The entire string must also be enclosed in double quotes, for example:

```
TABLETYPE="'TABLE','VIEW'"
```

This keyword can be used in conjunction with DBNAME and SCHEMALIST to further limit the number of tables for which information is returned.

TABLETYPE is used to provide a default for the DB2 CLI function that retrieves the list of tables, views, aliases, and synonyms in the data source. If the application does not specify a table type on the function call, and this keyword is not used, information about all table types is returned. If the application does supply a value for the *tabletype* on the function call, then that argument value overrides this keyword value.

If TABLETYPE includes any value other than TABLE, then the DBNAME keyword setting cannot be used to restrict information to a particular DB2 for OS/390 subsystem.

THREADSAFE= 1 | 0

This keyword is placed in the common section.

The `THREADSAFE` keyword controls whether DB2 CLI uses *POSIX mutexes* to make the DB2 CLI code *threadsafe* for multiple concurrent or parallel LE threads.

- 1 = The DB2 CLI code is threadsafe if the application is executing in a POSIX(ON) environment. Multiple LE threads in the process can use DB2 CLI. The threadsafe capability cannot be provided in a POSIX(OFF) environment. This is the default.
- 0 = The DB2 CLI code is not threadsafe. This reduces the overhead of serialization code in DB2 CLI for applications that are not *multithreaded*, but provides no protection for concurrent LE threads in applications that are multithreaded.

TRACE = 0 | 1

This keyword is placed in the common section.

The `TRACE` keyword allows you to enable the DB2 CLI trace. Use this keyword only if the trace is not already active.

0 = The DB2 CLI trace is not enabled. No diagnostic data is captured.

1 = The DB2 CLI trace is enabled. Diagnostic data is recorded within the application address space. If the user has included a **DSNAOTRC** DD card identifying a sequential data set, then the trace is externalized at normal program termination. It can then be formatted using the **DSNAOTRC** trace formatting program.

For more information about using the `TRACE` keyword, see “Use of Trace Keywords” on page 397.

TRACE_BUFFER_SIZE = *buffer size*

This keyword is placed in the common section.

The `TRACE_BUFFER_SIZE` keyword controls the size of the DB2 CLI trace buffer. This keyword is only used if a trace is started by using the `TRACE` keyword. *buffer size* is an integer value that represents the number of bytes to allocate for the trace buffer. The buffer size is rounded down to a multiple of 65536 (64K). If the value specified is less than 65536, then 65536 is used. The default value for the trace buffer size is 65536.

If a trace is already active, this keyword is ignored.

TRACEFILENAME = *dataset name*

This keyword is placed in the common section. `TRACEFILENAME` is only used if a trace is started by the `CLITRACE` keyword.

When `CLITRACE` is set to 1, use the `TRACEFILENAME` keyword to identify a sequential data set that records the DB2 CLI application trace. If the name has the form **"DD:DDNAME"**, then a data set must be currently allocated via the DDNAME. If the data set name is any other string, then the data set is dynamically allocated if it does not already exist. This keyword is only used if a trace is started by the `CLITRACE` keyword.

TRACE_NO_WRAP = 0 | 1

This keyword is placed in the common section.

The `TRACE_NO_WRAP` keyword controls the behaivor of `TRACE` when the DB2 CLI trace buffer fills up. This keyword is only used if a trace is started by the TRACE keyword.

0 = The trace table is a wrap-around trace. In this case, the trace remains active to capture the most current trace records. This is the default value.

1 = The trace stops capturing records when the trace buffer fills. The trace captures the initial trace records that were written.

If a trace is already active, this keyword is ignored.

TXNISOLATION = 1 | 2 | 4 | 8 | 32

This keyword is placed in the data source section.

The TXNISOLATION keyword sets the isolation level to:

1 = Read uncommitted (uncommitted read)
2 = Read committed (cursor stability) (default)
4 = Repeatable read (read stability)
8 = Serializable (repeatable read)
32 = (No commit, DB2 for OS/400 only)

The words in round brackets are the DB2 equivalents for SQL92 isolation levels. Note that *no commit* is not an SQL92 isolation level and is supported only on DATABASE 2 for OS/400. Refer to *Application Programming and SQL Guide*for more information on isolation levels.

UNDERSCORE = 1 | 0

This keyword is placed in the data source section.

The UNDERSCORE keyword specifies whether the underscore character "_" is to be used as a wildcard character (matching any one character, including no character), or to be used as itself. This parameter only affects catalog function calls that accept search pattern strings.

1 = "_" acts as a wildcard (default)

The underscore is treated as a wildcard matching any one character or none. For example, if two tables are defined as follows:

```
CREATE TABLE "OWNER"."KEY_WORDS" (COL1 INT)
CREATE TABLE "OWNER"."KEYWORDS" (COL1 INT)
```

The DB2 CLI catalog function call that returns table information (SQLTables()) returns both of these entries if "KEY_WORDS" is specified in the table name search pattern argument.

0 = "_" acts as itself

The underscore is treated as itself. If two tables are defined as shown in the example above, SQLTables() returns only the "KEY_WORDS" entry if "KEY_WORDS" is specified in the table name search pattern argument.

Setting this keyword to 0 can result in performance improvement in those cases where object names (owner, table, column) in the data source contain underscores.

Chapter 5. Functions

This section provides a description of each function. Each description has the following sections.

- Purpose
- Syntax
- Arguments
- Usage
- Return Codes
- Diagnostics
- Restrictions
- Example

Each section is described below.

Purpose

This section gives a brief overview of what the function does. It also indicates if any functions should be called before and after calling the function being described.

Each function also has a table, such as the one below that indicates which specification or standard the function conforms to. The first column indicates which version (1.0 or 2.0) of the ODBC specification the function was first provided. The second and third columns indicate if the function is included in the X/Open CLI CAE specification and the ISO CLI standard.

Table 8. Sample Function Specification Table

Specification:	**ODBC** 1.0	**X/OPEN CLI**	**ISO CLI**

This table indicates support of the function. Some functions use a set of options that do not apply to all specifications or standards. The restrictions section identifies any significant differences.

Syntax

This section contains the generic 'C' prototype. If the function is defined by ODBC V2.0, then the prototype should be identical to that specified in *ODBC 2.0 Programmer' s Reference and SDK Guide*.

All function arguments that are pointers are defined using the `FAR` macro. This macro is defined out (set to a blank). This is consistent with the ODBC specification.

Arguments

This section lists each function argument, along with its data type, a description and whether it is an input or output argument.

Only `SQLGetInfo()` and `SQLBindParameter()` have parameters that are both input and output.

Some functions contain input or output arguments which are known as *deferred* or *bound* arguments. These arguments are pointers to buffers allocated by the application, and are associated with (or bound to) either a parameter in an SQL statement, or a column in a result set. The data areas specified by the function are accessed by DB2 CLI at a later time. It is important that these deferred data areas are still valid at the time DB2 CLI accesses them.

Usage

This section provides information about how to use the function, and any special considerations. Possible error conditions are not discussed here, but are listed in the diagnostics section instead.

Return Codes

This section lists all the possible function return codes. When SQL_ERROR or SQL_SUCCESS_WITH_INFO is returned, error information can be obtained by calling `SQLError()`.

Refer to “Diagnostics” on page 36 for more information about return codes.

Diagnostics

This section contains a table that lists the SQLSTATEs explicitly returned by DB2 CLI (SQLSTATEs generated by the DBMS can also be returned) and indicates the cause of the error. These values are obtained by calling `SQLError()` after the function returns an SQL_ERROR or SQL_SUCCESS_WITH_INFO.

Refer to “Diagnostics” on page 36 for more information about diagnostics.

Restrictions

This section indicates any differences or limitations between DB2 CLI and ODBC that can affect an application.

Example

This section contains a code fragment that demonstrates the use of the function, using the generic data type definitions.

See “Chapter 4. Configuring CLI and Running Sample Applications” on page 49 for more information on setting up the DB2 CLI environment and accessing the sample applications.

References

This section lists related DB2 CLI functions.

DB2 CLI Function Summary

xproc=display. proc=display.

Table 9 (Page 1 of 4). Call Level Interface Function List by Category

Task Function Name	ODBC 2.0	X/OPEN	DB2 for OS/390 Support	Purpose
Connecting to a Data Source				
SQLAllocEnv	Core	Yes	Yes	Obtains an environment handle. One environment handle is used for one or more connections.
SQLAllocConnect	Core	Yes	Yes	Obtains a connection handle.
SQLConnect	Core	Yes	Yes	Connects to a specific data source by name.
SQLDriverConnect	Lvl 1	No	Yes	Connects to a specific driver by connection string or optionally requests that the Driver Manager and driver display connection dialogs for the user.
SQLSetConnection	No	No	Yes	Connects to a specific data source by connection string.

Table 9 (Page 2 of 4). Call Level Interface Function List by Category

Task	**ODBC**		**DB2 for OS/390**	
Function Name	**2.0**	**X/OPEN**	**Support**	**Purpose**
Obtaining Information about a Driver and Data Source				
SQLDataSources	Lvl 2	Yes	Yes	Returns the list of available data sources.
SQLGetInfo	Lvl 1	Yes	Yes	Returns information about a specific driver and data source.
SQLGetFunctions	Lvl 1	Yes	Yes	Returns supported driver functions.
SQLGetTypeInfo	Lvl 1	Yes	Yes	Returns information about supported data types.
Setting and Retrieving Driver Options				
SQLSetEnvAttr	No	Yes	Yes	Sets an environment option.
SQLGetEnvAttr	No	Yes	Yes	Returns the value of an environment option.
SQLSetConnectOption	Lvl 1	Yes	Yes	Sets a connection option.
SQLGetConnectOption	Lvl 1	Yes	Yes	Returns the value of a connection option.
SQLSetStmtOption	Lvl 1	Yes	Yes	Sets a statement option.
SQLGetStmtOption	Lvl 1	Yes	Yes	Returns the value of a statement option.
Preparing SQL Requests				
SQLAllocStmt	Core	Yes	Yes	Allocates a statement handle.
SQLPrepare	Core	Yes	Yes	Prepares an SQL statement for later execution.
SQLBindParameter	Lvl 1	No	Yes	Assigns storage for a parameter in an SQL statement (ODBC 2.0)
SQLSetParam	Core	Yes	Yes	Assigns storage for a parameter in an SQL statement (ODBC 1.0). **Note:** In ODBC 2.0 SQLBindParameter replaces this function.
SQLParamOptions	Lvl 2	No	Yes	Specifies the use of multiple values for parameters.
SQLGetCursorName	Core	Yes	Yes	Returns the cursor name associated with a statement handle.
SQLSetCursorName	Core	Yes	Yes	Specifies a cursor name.
Submitting Requests				
SQLExecute	Core	Yes	Yes	Executes a prepared statement.
SQLExecDirect	Core	Yes	Yes	Executes a statement.
SQLNativeSql	Lvl 2	No	Yes	Returns the text of an SQL statement as translated by the driver.
# SQLDescribeParam [a]	Lvl 2	Yes	Returns the description for a specific input parameter in a statement.	
SQLNumParams	Lvl 2	No	Yes	Returns the number of parameters in a statement.

Table 9 (Page 3 of 4). Call Level Interface Function List by Category

Task **Function Name**	**ODBC** **2.0**	**X/OPEN**	**DB2 for OS/390** **Support**	**Purpose**
SQLParamData	Lvl 1	Yes	Yes	Used in conjunction with `SQLPutData()` to supply parameter data at execution time. (Useful for long data values.)
SQLPutData	Lvl 1	Yes	Yes	Send part or all of a data value for a parameter. (Useful for long data values.)
Retrieving Results and Information about Results				
SQLRowCount	Core	Yes	Yes	Returns the number of rows affected by an insert, update, or delete request.
SQLNumResultCols	Core	Yes	Yes	Returns the number of columns in the result set.
SQLDescribeCol	Core	Yes	Yes	Describes a column in the result set.
SQLColAttributes	Core	Yes	Yes	Describes attributes of a column in the result set.
SQLSetColAttributes	No	No	Yes	Sets attributes of a column in the result set.
SQLBindCol	Core	Yes	Yes	Assigns storage for a result column and specifies the data type.
SQLFetch	Core	Yes	Yes	Returns a result row.
SQLExtendedFetch	Lvl 2	No	Yes	Returns multiple result rows.
SQLGetData	Lvl 1	Yes	Yes	Returns part or all of one column of one row of a result set. (Useful for long data values.)
SQLMoreResults	Lvl 2	No	Yes	Determines whether there are more result sets available and, if so, initializes processing for the next result set.
SQLError	Core	Yes	Yes	Returns additional error or status information.
SQLGetSQLCA	No	No	Yes	Returns the SQLCA associated with a statement handle.
Obtaining information about the data source's system tables (catalog functions)				
SQLColumnPrivileges	Lvl 2	Yes	Yes	Returns a list of columns and associated privileges for a specified table.
SQLColumns	Lvl 1	Yes	Yes	Returns the list of column names in specified tables.
SQLForeignKeys	Lvl 2	No	Yes	Returns a list of column names that comprise foreign keys, if they exist for a specified table.
SQLPrimaryKeys	Lvl 2	No	Yes	Returns the list of column names that comprise the primary key for a table.
SQLProcedureColumns	Lvl 2	No	Yes	Returns the list of input and output parameters for the specified procedures.
SQLProcedures	Lvl 2	No	Yes	Returns the list of procedure names stored in a specific data source.
SQLSpecialColumns	Lvl 1	Yes	Yes	Returns information about the optimal set of columns that uniquely identifies a row in a specified table.
SQLStatistics	Lvl 1	Yes	Yes	Returns statistics about a single table and the list of indexes associated with the table.

Table 9 (Page 4 of 4). Call Level Interface Function List by Category

Task **Function Name**	**ODBC** **2.0**	**X/OPEN**	**DB2 for OS/390** **Support**	**Purpose**
SQLTablePrivileges	Lvl 2	No	Yes	Returns a list of tables and the privileges associated with each table.
SQLTables	Lvl 1	Yes	Yes	Returns the list of table names stored in a specific data source.
Terminating a Statement				
SQLFreeStmt	Core	Yes	Yes	End statement processing and closes the associated cursor, discards pending results, and, optionally, frees all resources associated with the statement handle.
SQLCancel	Core	Yes	Yes	Cancels an SQL statement.
SQLTransact	Core	Yes	Yes	Commits or rolls back a transaction.
Terminating a Connection				
SQLDisconnect	Core	Yes	Yes	Closes the connection.
SQLFreeConnect	Core	Yes	Yes	Releases the connection handle.
SQLFreeEnv	Core	Yes	Yes	Releases the environment handle.

Note: The ODBC functions:

- SQLSetPos, SQLBrowseConnect, and SQLDescribeParam are not supported by Call Level Interface or DB2 CLI.
- SQLSetScrollOptions is not supported. It is superceded by the SQL_CURSOR_TYPE, SQL_CONCURRENCY, SQL_KEYSET_SIZE, and SQL_ROWSET_SIZE statement options.
- SQLDrivers is implemented by the ODBC driver manager and is not supported by DB2 CLI.

SQLAllocConnect - Allocate Connection Handle

Purpose

Specification:	**ODBC** 1.0	**X/OPEN CLI**	**ISO CLI**

`SQLAllocConnect()` allocates a connection handle and associated resources within the environment identified by the input environment handle. Call `SQLGetInfo()` with `fInfoType` set to SQL_ACTIVE_CONNECTIONS, to query the number of connections that can be allocated at any one time.

While this API is active, the DB2 CLI driver establishes an affinity with the DB2 subsystem. Processing includes allocating a DB2 for OS/390 plan as a resource.

`SQLAllocEnv()` must be called before calling this function.

This function must be called before calling `SQLConnect()` or `SQLDriverConnect()`.

Syntax

```
SQLRETURN   SQLAllocConnect  (SQLHENV          henv,
                              SQLHDBC     FAR   *phdbc);
```

Function Arguments

Table 10. SQLAllocConnect Arguments

Data Type	**Argument**	**Use**	**Description**
SQLHENV	*henv*	input	Environment handle
SQLHDBC *	*phdbc*	output	Pointer to connection handle

Usage

The output connection handle is used by DB2 CLI to reference all information related to the connection, including general status information, transaction state, and error information.

If the pointer to the connection handle (*phdbc*) already points to a valid connection handle previously allocated by `SQLAllocConnect()`, then the original value is overwritten as a result of this call. This is an application programming error which is not detected by DB2 CLI.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

If SQL_ERROR is returned, the *phdbc* argument is set to SQL_NULL_HDBC. The application should call `SQLError()` with the environment handle (*henv*) and with *hdbc* and *hstmt* arguments set to SQL_NULL_HDBC and SQL_NULL_HSTMT respectively.

Diagnostics

Table 11. SQLAllocConnect SQLSTATEs

CLI SQLSTATE	Description	Explanation
# **58**004	Unexpected system failure.	This could be a failure to establish the association with the DB2 for OS/390 subsystem or any other system related error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	*phdbc* was a null pointer.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.
S1014	No more handles.	Returned if the MAXCONN keyword or SQL_MAXCONN attribute is set to a positive integer and the number of connections has reached that value. If MAXCONN is set to zero, there is no limit. DB2 CLI is not able to allocate a handle due to internal resources.

Restrictions

None.

Example

The following example shows a basic connect, with minimal error handling.

```
/* ... */
/********************************************************
**   - demonstrate basic connection to two data sources.
**   - error handling mostly ignored for simplicity
**
**  Functions used:
**
**    SQLAllocConnect  SQLDisconnect
**    SQLAllocEnv      SQLFreeConnect
**    SQLConnect       SQLFreeEnv
**  Local Functions:
**    DBconnect
**
********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "sqlcli1.h"
int
DBconnect(SQLHENV henv,
          SQLHDBC * hdbc,
          char    * server);

#define MAX_UID_LENGTH    18
#define MAX_PWD_LENGTH    30
#define MAX_CONNECTIONS   2
```

```
int
main( )
{
    SQLHENV         henv;
    SQLHDBC         hdbc[MAX_CONNECTIONS];
    SERVER          svr[MAX_CONNECTIONS] =
                    {
                      "KARACHI"   ,
                      "DAMASCUSS"
                    }

    /* allocate an environment handle   */
    SQLAllocEnv(&henv);

    /* Connect to first data source */
    DBconnect(henv, &hdbc[0],
              svr[0]);

    /* Connect to second data source */
    DBconnect(henv, &hdbc[1],
              svr[1]);

    /*********   Start Processing Step  *************************/
    /* allocate statement handle, execute statement, etc.       */
    /*********   End Processing Step  ***************************/

    /************************************************************/
    /* Commit work on connection 1. This has NO effect on the   */
    /* transaction active on connection 2.                      */
    /************************************************************/

    SQLTransact (henv,
                 hdbc[0],
                 SQL_COMMIT);


    /************************************************************/
    /* Commit work on connection 2. This has NO effect on the   */
    /* transaction active on connection 1.                      */
    /************************************************************/

    SQLTransact (henv,
                 hdbc[1],
                 SQL_COMMIT);

    printf("\nDisconnecting .....\n");

    SQLDisconnect(hdbc[0]);     /* disconnect first connection */

    SQLDisconnect(hdbc[1]);     /* disconnect second connection */
    SQLFreeConnect(hdbc[0]);    /* free first connection handle  */
    SQLFreeConnect(hdbc[1]);    /* free second connection handle */
    SQLFreeEnv(henv);           /* free environment handle       */

    return (SQL_SUCCESS);
}
```

```
/*******************************************************************
**   Server is passed as a parameter. Note that USERID and PASSWORD**
**   are always NULL.                                              **
*******************************************************************/

int
DBconnect(SQLHENV henv,
          SQLHDBC * hdbc,
          char    * server)
{
    SQLRETURN       rc;
    SQLCHAR         buffer[255];
    SQLSMALLINT     outlen;


    SQLAllocConnect(henv, hdbc);/* allocate a connection handle     */

    rc = SQLConnect(*hdbc, server, SQL_NTS, NULL, SQL_NTS, NULL, SQL_NTS);
    if (rc != SQL_SUCCESS) {
        printf(">--- Error while connecting to database: %s -------\n", server);
        return (SQL_ERROR);
    } else {
        printf(">Connected to %s\n", server);
        return (SQL_SUCCESS);
    }
}
/* ... */
```

References

- “SQLAllocEnv - Allocate Environment Handle” on page 82
- “SQLConnect - Connect to a Data Source” on page 120
- “SQLDriverConnect - (Expanded) Connect to a Data Source” on page 137
- “SQLDisconnect - Disconnect from a Data Source” on page 135
- “SQLFreeConnect - Free Connection Handle” on page 178
- “SQLGetConnectOption - Returns Current Setting of A Connect Option” on page 185
- “SQLSetConnectOption - Set Connection Option” on page 298

SQLAllocEnv - Allocate Environment Handle

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

# `SQLAllocEnv()` allocates an environment handle and associated resources. An
# application can allocate more than one environment at a time.

An application must call this function prior to `SQLAllocConnect()` or any other DB2 CLI functions. The *henv* value is passed in all subsequent function calls that require an environment handle as input.

Syntax

```
SQLRETURN   SQLAllocEnv      (SQLHENV     FAR   *phenv);
```

Function Arguments

Table 12. SQLAllocEnv Arguments

Data Type	Argument	Use	Description
SQLHENV *	*phenv*	output	Pointer to environment handle

Usage

# There can be only one active environment at a time per application. Any
# subsequent calls to `SQLAllocEnv()` return the same handle as the first
# `SQLAllocEnv()` call.

`SQLFreeEnv()` must be called for each successful `SQLAllocEnv()` call before the
| resources associated with the handle are released. `SQLFreeEnv()` must also be called to free a restricted environment handle as described under 'Return Codes' below.

Return Codes

- SQL_SUCCESS
- SQL_ERROR

If SQL_ERROR is returned and *phenv* is equal to SQL_NULL_HENV, then `SQLError()` cannot be called because there is no handle with which to associate additional diagnostic information.

If the return code is SQL_ERROR and the pointer to the environment handle is not equal to SQL_NULL_HENV, then the handle is a *restricted handle*. This means the handle can only be used in a call to `SQLError()` to obtain more error information, or to `SQLFreeEnv()`.

Diagnostics

Table 13. SQLAllocEnv SQLSTATEs

SQLSTATE	Description	Explanation
# **S1**001	Memory allocation failure.	DB2 CLI is not able to allcoate memory required to support execution or completion of the function.

Restrictions

None.

Example

Refer to “Example” on page 79

References

- “SQLAllocConnect - Allocate Connection Handle” on page 78
- “SQLFreeEnv - Free Environment Handle” on page 180

SQLAllocStmt - Allocate a Statement Handle

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLAllocStmt() allocates a new statement handle and associates it with the connection specified by the connection handle. There is no defined limit on the number of statement handles that can be allocated at any one time.

SQLConnect() or SQLDriverConnect() must be called before calling this function.

This function must be called before SQLBindParameter(), SQLPrepare(), SQLExecute(), SQLExecDirect(), or any other function that has a statement handle as one of its input arguments.

Syntax

```
SQLRETURN   SQLAllocStmt     (SQLHDBC          hdbc,
                              SQLHSTMT    FAR   *phstmt);
```

Function Arguments

Table 14. SQLAllocStmt Arguments

Data Type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Connection handle
SQLHSTMT *	*phstmt*	output	Pointer to statement handle

Usage

DB2 CLI uses each statement handle to relate all the descriptors, attribute values, result values, cursor information, and status information to the SQL statement processed. Although each SQL statement must have a statement handle, you can reuse the handles for different statements.

A call to this function requires that *hdbc* references an active database connection.

To execute a positioned UPDATE or DELETE, the application must use different statement handles for the SELECT statement and the UPDATE or DELETE statement.

If the input pointer to the statement handle (*phstmt*) already points to a valid statement handle allocated by a previous call to SQLAllocStmt(), then the original value is overwritten as a result of this call. This is an application programming error that is not detected by DB2 CLI.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

If SQL_ERROR is returned, the *phstmt* argument is set to SQL_NULL_HSTMT. The application should call `SQLError()` with the same *hdbc* and with the *hstmt* argument set to SQL_NULL_HSTMT.

Diagnostics

Table 15. SQLAllocStmt SQLSTATEs

SQLSTATE	Description	Explanation
08003	Connection is closed.	The connection specified by the *hdbc* argument is not open. The connection must be established successfully (and the connection must be open) for the application to call `SQLAllocStmt()`.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	*phstmt* was a null pointer.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.
S1014	No more handles.	DB2 CLI is not able to allocate a handle due to internal resources.

Restrictions

None.

Example

Refer to "Example" on page 167.

References

- "SQLConnect - Connect to a Data Source" on page 120
- "SQLDriverConnect - (Expanded) Connect to a Data Source" on page 137
- "SQLFreeStmt - Free (or Reset) a Statement Handle" on page 182
- "SQLGetStmtOption - Returns Current Setting of A Statement Option" on page 236
- "SQLSetStmtOption - Set Statement Option" on page 315

SQLBindCol - Bind a Column to an Application Variable

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLBindCol() is used to associate (bind) columns in a result set to:

- Application variables or arrays of application variables (storage buffers), for all C data types. In this case, data is transferred from the DBMS to the application when SQLFetch() or SQLExtendedFetch() is called. Data conversion can occur as the data is transferred.

SQLBindCol() is called once for each column in the result set that the application needs to retrieve.

In general, SQLPrepare(), SQLExecDirect() or one of the schema functions is called before this function, and SQLFetch() or SQLExtendedFetch() is called after. Column attributes might also be needed before calling SQLBindCol(), and can be obtained using SQLDescribeCol() or SQLColAttributes().

Syntax

```
SQLRETURN   SQLBindCol       (SQLHSTMT          hstmt,
                              SQLUSMALLINT      icol,
                              SQLSMALLINT       fCType,
                              SQLPOINTER        rgbValue,
                              SQLINTEGER        cbValueMax,
                              SQLINTEGER  FAR   *pcbValue);
```

Function Arguments

Table 16 (Page 1 of 2). SQLBindCol Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle
SQLUSMALLINT	*icol*	input	Number identifying the column. Columns are numbered sequentially, from left to right, starting at 1.

Table 16 (Page 2 of 2). SQLBindCol Arguments

Data Type	Argument	Use	Description
SQLSMALLINT	*fCType*	input	The C data type for column number *icol* in the result set. The following types are supported: • SQL_C_BINARY • SQL_C_BIT • SQL_C_CHAR • SQL_C_DATE • SQL_C_DBCHAR • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG • SQL_C_SHORT • SQL_C_TIME • SQL_C_TIMESTAMP • SQL_C_TINYINT The supported data types are based on the data source to which you are connected. Specifying SQL_C_DEFAULT causes data to be transferred to its default C data type. Refer to Table 3 on page 40 for more information.
SQLPOINTER	*rgbValue*	output (deferred)	Pointer to buffer (or an array of buffers if using `SQLExtendedFetch()`) where DB2 CLI is to store the column data when the fetch occurs. If rgbValue is null, the column is unbound.
SQLINTEGER	*cbValueMax*	input	Size of *rgbValue* buffer in bytes available to store the column data. If *fCType* denotes a binary or character string (either single or double byte) or is SQL_C_DEFAULT, then cbValueMax must be > 0, or an error is returned. Otherwise, this argument is ignored.
SQLINTEGER *	*pcbValue*	output (deferred)	Pointer to value (or array of values) which indicates the number of bytes DB2 CLI has available to return in the *rgbValue* buffer. `SQLFetch()` returns SQL_NULL_DATA in this argument if the data value of the column is null. This pointer value must be unique for each bound column, or NULL. SQL_NO_LENGTH can also be returned. Refer to the 'Usage' section below for more information.

Note:

- For this function, pointers *rgbValue* and *pcbValue* are deferred outputs, meaning that the storage locations they point to do not get updated until a result set row is fetched. As a result, the locations referenced by these pointers must remain valid until `SQLFetch()` or `SQLExtendedFetch()` is called. For example, if `SQLBindCol()` is called within a local function, `SQLFetch()` must be called from within the same scope of the function or the *rgbValue* buffer must be allocated as static or global.

- DB2 CLI performs better for all variable length data types if *rgbValue* is placed consecutively in memory after *pcbValue*. See the 'Usage' section for more details.

Usage

The application calls SQLBindCol() once for each column in the result set for which it wishes to retrieve data. Result sets are generated either by calling SQLPrepare(), SQLExecDirect(), SQLGetTypeInfo(), or one of the catalog functions. When SQLFetch() is called, the data in each of these *bound* columns is placed into the assigned location (given by the pointers *rgbValue* and *cbValue*).

SQLExtendedFetch() can be used in place of SQLFetch() to retrieve multiple rows from the result set into an array. In this case, *rgbValue* references an array. For more information, refer to "Retrieving A Result Set Into An Array" on page 357 and "SQLExtendedFetch - Extended Fetch (Fetch Array of Rows)" on page 157. Use of SQLExtendedFetch() and SQLFetch() cannot be mixed for the same result set.

Columns are identified by a number, assigned sequentially from left to right, starting at 1. The number of columns in the result set can be determined by calling SQLNumResultCols() or by calling SQLColAttributes() with the *fdescType* argument set to SQL_COLUMN_COUNT.

The application can query the attributes (such as data type and length) of the column by first calling SQLDescribeCol() or SQLColAttributes(). (As an alternative, refer to Appendix A, "Programming Hints and Tips" on page 393 for information about using SQLSetColAttributes() when the application has prior knowledge of the format of the result set.) This information can then be used to allocate a storage location of the correct data type and length, to indicate data conversion to another data type. Refer to "Data Types and Data Conversion" on page 38 for more information on default types and supported conversions.

An application can choose not to bind every column, or even not to bind any columns. Data in any of the columns can also be retrieved using SQLGetData() after the bound columns have been fetched for the current row. Generally, SQLBindCol() is more efficient than SQLGetData(). For a discussion of when to use one function over the other, refer to Appendix A, "Programming Hints and Tips" on page 393.

In subsequent fetches, the application can change the binding of these columns or bind previously unbound columns by calling SQLBindCol(). The new binding does not apply to data already fetched, it is used on the next fetch. To unbind a single column, call SQLBindCol() with the *rgbValue* pointer set to NULL. To unbind all the columns, the application should call SQLFreeStmt() with the *fOption* input set to SQL_UNBIND.

The application must ensure enough storage is allocated for the data to be retrieved. If the buffer is to contain variable length data, the application must allocate as much storage as the maximum length of the bound column requires; otherwise, the data might be truncated. If the buffer is to contain fixed length data, DB2 CLI assumes the size of the buffer is the length of the C data type. If data conversion is specified, the required size might be affected, see "Data Types and Data Conversion" on page 38 for more information.

If string truncation does occur, SQL_SUCCESS_WITH_INFO is returned and *pcbValue* is set to the actual size of *rgbValue* available for return to the application.

Truncation is also affected by the SQL_MAX_LENGTH statement option (used to limit the amount of data returned to the application). The application can specify not to report truncation by calling `SQLSetStmtOption()` with SQL_MAX_LENGTH and a value for the maximum length to return for all variable length columns, and by allocating an *rgbValue* buffer of the same size (plus the null-terminator). If the column data is larger than the set maximum length, SQL_SUCCESS is returned when the value is fetched and the maximum length, not the actual length, is returned in *pcbValue*.

If the column to be bound is an SQL_GRAPHIC, SQL_VARGRAPHIC or SQL_LONGVARGRAPHIC type, then *fCType* can be set to SQL_C_DBCHAR or SQL_C_CHAR. If *fCType* is SQL_C_DBCHAR, the data fetched into the *rgbValue* buffer is null-terminated by a double byte null-terminator. If *fCType* is SQL_C_CHAR, then the data is not null-terminated. In both cases, the length of the *rgbValue* buffer (*cbValueMax*) is in units of bytes and should therefore be a multiple of 2.

When binding any variable length column, DB2 CLI can write *pcbValue* and *rgbValue* in one operation if they are allocated contiguously. For example:

```
struct {  SQLINTEGER  pcbValue;
          SQLCHAR     rgbValue[MAX_BUFFER];
       } column;
```

Note: SQL_NO_TOTAL is returned in *pcbValue* if:

- The SQL type is a variable length type, and
- *pcbValue* and *rgbValue* are contiguous, and
- The column type is NOT NULLABLE, and
- String truncation occurred.

The most recent bind column function call determines the type of binding that is in effect.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 17 (Page 1 of 2). SQLBindCol SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.

Table 17 (Page 2 of 2). SQLBindCol SQLSTATEs

SQLSTATE	Description	Explanation
S1002	Invalid column number.	The value specified for the argument *icol* was less than 1. The value specified for the argument *icol* exceeded the maximum number of columns supported by the data source.
S1003	Program type out of range.	*fCType* is not a valid data type or SQL_C_DEFAULT.
S1010	Function sequence error.	The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.
S1090	Invalid string or buffer length.	The value specified for the argument *cbValueMax* is less than 1 and the argument *fCType* is either SQL_C_CHAR, SQL_C_BINARY or SQL_C_DEFAULT.
S1C00	Driver not capable.	DB2 CLI recognizes, but does not support the data type specified in the argument *fCType*

Note: Additional diagnostic messages relating to the bound columns might be reported at fetch time.

Restrictions

None.

Example

Refer to “Example” on page 167.

References

- “SQLFetch - Fetch Next Row” on page 164
- “SQLExtendedFetch - Extended Fetch (Fetch Array of Rows)” on page 157

SQLBindParameter - Binds A Parameter Marker to a Buffer

Purpose

Specification:	**ODBC** 2.0		

SQLBindParameter() is used to associate (bind) parameter markers in an SQL statement to application variables or arrays of application variables (storage buffers), for all C data types. In this case, data is transferred from the application to the DBMS when SQLExecute() or SQLExecDirect() is called. Data conversion can occur as the data is transferred.

This function must also be used to bind application storage to a parameter of a stored procedure CALL statement where the parameter can be input, output or both. This function is essentially an extension of SQLSetParam().

Syntax

```
SQLRETURN  SQL_API SQLBindParameter(
                              SQLHSTMT          hstmt,
                              SQLUSMALLINT      ipar,
                              SQLSMALLINT       fParamType,
                              SQLSMALLINT       fCType,
                              SQLSMALLINT       fSqlType,
                              SQLUINTEGER       cbColDef,
                              SQLSMALLINT       ibScale,
                              SQLPOINTER        rgbValue,
                              SQLINTEGER        cbValueMax,
                              SQLINTEGER   FAR *pcbValue);
```

Function Arguments

Table 18 (Page 1 of 5). SQLBindParameter Arguments

Data Type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle
SQLUSMALLINT	ipar	input	Parameter marker number, ordered sequentially left to right, starting at 1.

Table 18 (Page 2 of 5). SQLBindParameter Arguments

Data Type	Argument	Use	Description
SQLSMALLINT	fParamType	input	The type of parameter. The supported types are: • SQL_PARAM_INPUT: The parameter marker is associated with an SQL statement that is not a stored procedure CALL; or, it marks an input parameter of the CALLed stored procedure. When the statement is executed, actual data value for the parameter is sent to the server: the *rgbValue* buffer must contain valid input data values; the *pcbValue* buffer must contain the corresponding length value or SQL_NTS, SQL_NULL_DATA, or (if the value should be sent via `SQLParamData()` and `SQLPutData()`) SQL_DATA_AT_EXEC. • SQL_PARAM_INPUT_OUTPUT: The parameter marker is associated with an input/output parameter of the CALLed stored procedure. When the statement is executed, actual data value for the parameter is sent to the server: the *rgbValue* buffer must contain valid input data values; the *pcbValue* buffer must contain the corresponding length value or SQL_NTS, SQL_NULL_DATA, or (if the value should be sent via `SQLParamData()` and `SQLPutData()`) SQL_DATA_AT_EXEC. • SQL_PARAM_OUTPUT: The parameter marker is associated with an output parameter of the CALLed stored procedure or the return value of the stored procedure. After the statement is executed, data for the output parameter is returned to the application buffer specified by *rgbValue* and *pcbValue*, unless both are NULL pointers, in which case the output data is discarded.
SQLSMALLINT	fCType	input	C data type of the parameter. The following types are supported: • SQL_C_BINARY • SQL_C_BIT • SQL_C_CHAR • SQL_C_DATE • SQL_C_DBCHAR • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG • SQL_C_SHORT • SQL_C_TIME • SQL_C_TIMESTAMP • SQL_C_TINYINT Specifying SQL_C_DEFAULT causes data to be transferred from its default C data type to the type indicated in *fSqlType*.

Table 18 (Page 3 of 5). SQLBindParameter Arguments

Data Type	Argument	Use	Description
SQLSMALLINT	fSqlType	input	SQL Data Type of the parameter. The supported types are: • SQL_BINARY • SQL_CHAR • SQL_DATE • SQL_DECIMAL • SQL_DOUBLE • SQL_FLOAT • SQL_GRAPHIC • SQL_INTEGER • SQL_LONGVARBINARY • SQL_LONGVARCHAR • SQL_LONGVARGRAPHIC • SQL_NUMERIC • SQL_REAL • SQL_SMALLINT • SQL_TIME • SQL_TIMESTAMP • SQL_VARBINARY • SQL_VARCHAR • SQL_VARGRAPHIC
# SQLUINTEGER	cbColDef	input	Precision of the corresponding parameter marker. If *fSqlType* denotes: • A binary or single byte character string (for example, SQL_CHAR, SQL_BINARY), this is the maximum length in bytes for this parameter marker. • A double byte character string (for example, SQL_GRAPHIC), this is the maximum length in double-byte characters for this parameter. • SQL_DECIMAL, SQL_NUMERIC, this is the maximum decimal precision. • Otherwise, this argument is ignored.
SQLSMALLINT	ibScale	input	Scale of the corresponding parameter if *fSqlType* is SQL_DECIMAL or SQL_NUMERIC. If *fSqlType* is SQL_TIMESTAMP, this is the number of digits to the right of the decimal point in the character representation of a timestamp (for example, the scale of yyyy-mm-dd hh:mm:ss.fff is 3). Other than for the *fSqlType* values mentioned here, *ibScale* is ignored.

Table 18 (Page 4 of 5). SQLBindParameter Arguments

Data Type	Argument	Use	Description
SQLPOINTER	rgbValue	input (deferred) and/or output (deferred)	• On input (*fParamType* set to SQL_PARAM_INPUT, or SQL_PARAM_INPUT_OUTPUT): At execution time, if *pcbValue* does not contain SQL_NULL_DATA or SQL_DATA_AT_EXEC, then *rgbValue* points to a buffer that contains the actual data for the parameter. If *pcbValue* contains SQL_DATA_AT_EXEC, then *rgbValue* is an application-defined 32-bit value that is associated with this parameter. This 32-bit value is returned to the application via a subsequent `SQLParamData()` call. If `SQLParamOptions()` is called to specify multiple values for the parameter, then *rgbValue* is a pointer to an input buffer array of *cbValueMax* bytes. • On output (*fParamType*) set to SQL_PARAM_OUTPUT, or SQL_PARAM_INPUT_OUTPUT): *rgbValue* points to the buffer where the output parameter value of the stored procedure is stored. If *fParamType* is set to SQL_PARAM_OUTPUT, and both *rgbValue* and *pcbValue* are NULL pointers, then the output parameter value or the return value from the stored procedure call is discarded.
SQLINTEGER	cbValueMax	input	For character and binary data, *cbValueMax* specifies the length of the *rgbValue* buffer (if treated as a single element) or the length of each element in the *rgbValue* array (if the application calls `SQLParamOptions()` to specify multiple values for each parameter). For non-character and non-binary data, this argument is ignored -- the length of the *rgbValue* buffer (if it is a single element) or the length of each element in the *rgbValue* array (if `SQLParamOptions()` is used to specify an array of values for each parameter) is assumed to be the length associated with the C data type. For output parameters, *cbValueMax* is used to determine whether to truncate character or binary output data in the following manner: • For character data, if the number of bytes available to return is greater than or equal to *cbValueMax*, the data in *rgbValue* is truncated to *cbValueMax-1* bytes and is null-terminated (unless null-termination has been turned off). • For binary data, if the number of bytes available to return is greater than *cbValueMax*, the data in *rgbValue* is truncated to *cbValueMax* bytes.

Table 18 (Page 5 of 5). SQLBindParameter Arguments

Data Type	Argument	Use	Description
SQLINTEGER *	pcbValue	input (deferred) and/or output (deferred)	- If this is an input or input/output parameter: This is the pointer to the location which contains (when the statement is executed) the length of the parameter marker value stored at *rgbValue*. To specify a null value for a parameter marker, this storage location must contain SQL_NULL_DATA. If *fCType* is SQL_C_CHAR, this storage location must contain either the exact length of the data stored at *rgbValue*, or SQL_NTS if the contents at *rgbValue* are null-terminated. If *fCType* indicates character data (explicitly, or implicitly using SQL_C_DEFAULT), and this pointer is set to NULL, it is assumed that the application always provides a null-terminated string in *rgbValue*. This also implies that this parameter marker never has a null value. If *fSqlType* denotes a graphic data type and the *fCType* is SQL_C_CHAR, the pointer to *pcbValue* can never be NULL and the contents of *pcbValue* can never hold SQL_NTS. In general for graphic data types, this length should be the number of octets that the double byte data occupies; therefore, the length should always be a multiple of 2. In fact, if the length is odd, then an error occurs when the statement is executed. When `SQLExecute()` or `SQLExecDirect()` is called, and *pcbValue* points to a value of SQL_DATA_AT_EXEC, the data for the parameter is sent with `SQLPutData()`. This parameter is referred to as a **data-at-execution** parameter. If `SQLParamOptions()` is used to specify multiple values for each parameter, *pcbValue* points to an array of SQLINTEGER values where each of the elements can be the number of bytes in the corresponding *rgbValue* element (excluding the null-terminator), or SQL_NULL_DATA. - If this is an output parameter (*fParamType* is set to SQL_PARAM_OUTPUT): This must be an output parameter or return value of a stored procedure CALL and points to one of the following, after the execution of the stored procedure: • number of bytes available to return in *rgbValue*, excluding the null-termination character. • SQL_NULL_DATA • SQL_NO_TOTAL if the number of bytes available to return cannot be determined.

Usage

A parameter marker is represented by a "?" character in an SQL statement and is used to indicate a position in the statement where an application supplied value is to be substituted when the statement is executed. This value can be obtained from an application variable. `SQLBindParameter()` (or `SQLSetParam()`) is used to bind the application storage area to the parameter marker.

The application must bind a variable to each parameter marker in the SQL statement before executing the SQL statement. For this function, *rgbValue* and *pcbValue* are deferred arguments, the storage locations must be valid and contain input data values when the statement is executed. This means either keeping the `SQLExecDirect()` or `SQLExecute()` call in the same procedure scope as the `SQLBindParameter()` calls, or, these storage locations must be dynamically allocated or declared statically or globally.

`SQLSetParam()` can be called before `SQLPrepare()` if the columns in the result set are known; otherwise, the attributes of the result set can be obtained after the statement is prepared.

Parameter markers are referenced by number (*icol*) and are numbered sequentially from left to right, starting at 1.

All parameters bound by this function remain in effect until `SQLFreeStmt()` is called with either the SQL_DROP or SQL_RESET_PARAMS option, or until `SQLBindParameter()` is called again for the same parameter *ipar* number.

After the SQL statement is executed, and the results processed, the application might wish to reuse the statement handle to execute a different SQL statement. If the parameter marker specifications are different (number of parameters, length or type), `SQLFreeStmt()` should be called with SQL_RESET_PARAMS to reset or clear the parameter bindings.

The C buffer data type given by *fCType* must be compatible with the SQL data type indicated by *fSqlType*, or an error occurs.

An application can pass the value for a parameter either in the *rgbValue* buffer or with one or more calls to `SQLPutData()`. In the latter case, these parameters are data-at-execution parameters. The application informs DB2 CLI of a data-at-execution parameter by placing the SQL_DATA_AT_EXEC value in the *pcbValue* buffer. It sets the *rgbValue* input argument to a 32-bit value which is returned on a subsequent `SQLParamData()` call and can be used to identify the parameter position.

Since the data in the variables referenced by *rgbValue* and *pcbValue* is not verified until the statement is executed, data content or format errors are not detected or reported until `SQLExecute()` or `SQLExecDirect()` is called.

`SQLBindParameter()` essentially extends the capability of the `SQLSetParam()` function by providing a method of:

- Specifying whether a parameter is input, input / output, or output, necessary for proper handling of parameters for stored procedures.
- Specifying an array of input parameter values when `SQLParamOptions()` is used in conjunction with `SQLBindParameter()`. `SQLSetParam()` can still be used to

bind single element application variables to parameter markers that are not part of a stored procedure CALL statement.

The *fParamType* argument specifies the type of the parameter. All parameters in the SQL statements that do not call procedures are input parameters. Parameters in stored procedure calls can be input, input/output, or output parameters. Even though the DB2 stored procedure argument convention typically implies that all procedure arguments are input/output, the application programmer can still choose to specify the nature of input or output more exactly on the `SQLBindParameter()` to follow a more rigorous coding style.

Note:

- If an application cannot determine the type of a parameter in a procedure call, set *fParamType* to SQL_PARAM_INPUT; if the data source returns a value for the parameter, DB2 CLI discards it.
- If an application has marked a parameter as SQL_PARAM_INPUT_OUTPUT or SQL_PARAM_OUTPUT and the data source does not return a value, DB2 CLI sets the *pcbValue* buffer to SQL_NULL_DATA.
- If an application marks a parameter as SQL_PARAM_OUTPUT, data for the parameter is returned to the application after the CALL statement is processed. If the *rgbValue* and *pcbValue* arguments are both null pointers, DB2 CLI discards the output value. If the data source does not return a value for an output parameter, DB2 CLI sets the *pcbValue* buffer to SQL_NULL_DATA.
- For this function, *rgbValue* and *pcbValue* are deferred arguments. In the case where *fParamType* is set to SQL_PARAM_INPUT or SQL_PARAM_INPUT_OUTPUT, the storage locations must be valid and contain input data values when the statement is executed. This means either keeping the `SQLExecDirect()` or `SQLExecute()` call in the same procedure scope as the `SQLBindParameter()` calls, or, these storage locations must be dynamically allocated or statically or globally declared.

  Similarly, if *fParamType* is set to SQL_PARAM_OUTPUT or SQL_PARAM_INPUT_OUTPUT, the *rgbValue* and *pcbValue* buffer locations must remain valid until the CALL statement is executed.

For character and binary C data, the *cbValueMax* argument specifies the length of the *rgbValue* buffer if it is a single element; or, if the application calls `SQLParamOptions()` to specify multiple values for each parameter, *cbValueMax* is the length of *each* element in the *rgbValue* array, INCLUDING the null-terminator. If the application specifies multiple values, *cbValueMax* is used to determine the location of values in the *rgbValue* array. For all other types of C data, the *cbValueMax* argument is ignored.

An application can pass the value for a parameter either in the *rgbValue* buffer or with one or more calls to `SQLPutData()`. In the latter case, these parameters are data-at-execution parameters. The application informs DB2 CLI of a data-at-execution parameter by placing the SQL_DATA_AT_EXEC value in the pcbValue buffer. It sets the *rgbValue* input argument to a 32-bit value which is returned on a subsequent `SQLParamData()` call and can be used to identify the parameter position.

When `SQLBindParameter()` is used to bind an application variable to an output parameter for a stored procedure, DB2 CLI can provide some performance enhancement if the *rgbValue* buffer is placed consecutively in memory after the *pcbValue* buffer. For example:

```
struct {  SQLINTEGER  pcbValue;
          SQLCHAR     rgbValue[MAX_BUFFER];
       } column;
```

*

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 19 (Page 1 of 2). SQLBindParameter SQLSTATEs

SQLSTATE	Description	Explanation
07006	Invalid conversion.	The conversion from the data value identified by the *fCType* argument to the data type identified by the *fSqlType* argument is not a meaningful conversion. (For example, conversion from SQL_C_DATE to SQL_DOUBLE.)
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1003	Program type out of range.	The value specified by the argument *fCType* not a valid data type or SQL_C_DEFAULT.
S1004	SQL data type out of range.	The value specified for the argument *fSqlType* is not a valid SQL data type.
S1009	Invalid argument value.	The argument *rgbValue* is a null pointer; the argument *pcbValue* is a null pointer; and *fParamType* is not SQL_PARAM_OUTPUT.
S1010	Function sequence error.	Function is called after `SQLExecute()` or `SQLExecDirect()` returned SQL_NEED_DATA, but data have not been sent for all *data-at-execution* parameters.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.
S1090	Invalid string or buffer length.	The value specified for the argument *cbValueMax* is less than 0.
S1093	Invalid parameter number.	The value specified for the argument *ipar* is less than 1 or greater than the maximum number of parameters supported by the server.
S1094	Invalid scale value.	The value specified for *fSqlType* is either SQL_DECIMAL or SQL_NUMERIC and the value specified for *ibScale* is less than 0 or greater than the value for the argument *cbParamDef* (precision). The value specified for *fSqlType* is SQL_C_TIMESTAMP; the value for *fSqlType* is either SQL_CHAR or SQL_VARCHAR; and the value for *ibScale* is less than 0 or greater than 6.

Table 19 (Page 2 of 2). SQLBindParameter SQLSTATEs

SQLSTATE	Description	Explanation
S1104	Invalid precision value.	The value specified for *fSqlType* was either SQL_DECIMAL or SQL_NUMERIC and the value specified for *cbParamDef* was less than 1.
S1105	Invalid parameter type.	*fParamType* is not one of SQL_PARAM_INPUT, SQL_PARAM_OUTPUT, or SQL_PARAM_INPUT_OUTPUT.
S1C00	Driver not capable.	DB2 CLI or data source does not support the conversion specified by the combination of the value specified for the argument *fCType* and the value specified for the argument *fSqlType*. The value specified for the argument *fSqlType* is not supported by either DB2 CLI or the data source.

Restrictions

In ODBC 2.0, this function has replaced `SQLSetParam()`.

A new value for *pcbValue*, SQL_DEFAULT_PARAM, was introduced in ODBC 2.0, to indicate that the procedure should use the default value of a parameter, rather than a value sent from the application. Since DB2 stored procedure arguments do not have the concept of default values, specification of this value for *pcbValue* argument results in an error when the CALL statement is executed since the SQL_DEFAULT_PARAM value is considered an invalid length.

ODBC 2.0 also introduced the SQL_LEN_DATA_AT_EXEC(*length*) macro to be used with the *pcbValue* argument. The macro is used to specify the sum total length of the entire data that is sent for character or binary C data via the subsequent `SQLPutData()` calls. Since the DB2 ODBC driver does not need this information, the macro is not needed. An ODBC application calls `SQLGetInfo()` with the SQL_NEED_LONG_DATA_LEN option to check if the driver needs this information. The DATABASE 2 ODBC driver returns 'N' to indicate that this information is not needed by `SQLPutData()`.

Example

The example shown below binds a variety of data types to a set of parameters. For an additional example refer to “Stored Procedure” on page 472.

```
/* ... */
    SQLCHAR         stmt[] =
    "INSERT INTO PRODUCT VALUES (?, ?, ?, ?, ?)";

    SQLINTEGER      Prod_Num[NUM_PRODS] = {
        100110, 100120, 100210, 100220, 100510, 100520, 200110,
        200120, 200210, 200220, 200510, 200610, 990110, 990120,
        500110, 500210, 300100
    };

    SQLCHAR         Description[NUM_PRODS][257] = {
        "Aquarium-Glass-25 litres", "Aquarium-Glass-50 litres",
        "Aquarium-Acrylic-25 litres", "Aquarium-Acrylic-50 litres",
        "Aquarium-Stand-Small", "Aquarium-Stand-Large",
        "Pump-Basic-25 litre", "Pump-Basic-50 litre",
        "Pump-Deluxe-25 litre", "Pump-Deluxe-50 litre",
        "Pump-Filter-(for Basic Pump)",
        "Pump-Filter-(for Deluxe Pump)",
        "Aquarium-Kit-Small", "Aquarium-Kit-Large",
        "Gravel-Colored", "Fish-Food-Deluxe-Bulk",
        "Plastic-Tubing"
    };
    SQLDOUBLE       UPrice[NUM_PRODS] = {
        110.00, 190.00, 100.00, 150.00, 60.00, 90.00, 30.00,
        45.00, 55.00, 75.00, 4.75, 5.25, 160.00, 240.00,
        2.50, 35.00, 5.50
    };
    SQLCHAR         Units[NUM_PRODS][3] = {
        "  ", "  ", "  ", "  ", "  ", "  ", "  ", "  ", "  ",
        "  ", "  ", "  ", "  ", "  ", "kg", "kg", "m"
    };
    SQLCHAR         Combo[NUM_PRODS][2] = {
        "N", "N", "N", "N", "N", "N", "N", "N", "N",
        "N", "N", "N", "Y", "Y", "N", "N", "N"
    };
    SQLUINTEGER     pirow = 0;
/* ... */
    /* Prepare the statement */
    rc = SQLPrepare(hstmt, stmt, SQL_NTS);

    rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER,
                          0, 0, Prod_Num, 0, NULL);

    rc = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_VARCHAR,
                     257, 0, Description, 257, NULL);

    rc = SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_DOUBLE, SQL_DECIMAL,
                          10, 2, UPrice, 0, NULL);

    rc = SQLBindParameter(hstmt, 4, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                          3, 0, Units, 3, NULL);

    rc = SQLBindParameter(hstmt, 5, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                          2, 0, Combo, 2, NULL);

    rc = SQLParamOptions(hstmt, NUM_PRODS, &pirow);

    rc = SQLExecute(hstmt);
    printf("Inserted %ld Rows\n", pirow);
/* ... */
```

References

- “SQLExecDirect - Execute a Statement Directly” on page 149
- “SQLExecute - Execute a Statement” on page 154
- “SQLParamData - Get Next Parameter For Which A Data Value Is Needed” on page 257
- “SQLParamOptions - Specify an Input Array for a Parameter” on page 259
- “SQLPutData - Passing Data Value for A Parameter” on page 287

SQLCancel - Cancel Statement

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLCancel() can be used to prematurely terminate the *data-at-execution* sequence described in “Sending/Retrieving Long Data in Pieces” on page 351.

Syntax

```
SQLRETURN   SQLCancel       (SQLHSTMT         hstmt);
```

Function Arguments

Table 20. SQLCancel Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle

Usage

After SQLExecDirect() or SQLExecute() returns SQL_NEED_DATA to solicit for values for data-at-execution parameters, SQLCancel() can be used to cancel the data-at-execution sequence described in “Sending/Retrieving Long Data in Pieces” on page 351. SQLCancel() can be called any time before the final SQLParamData() in the sequence. After the cancellation of this sequence, the application can call SQLExecute() or SQLExecDirect() to re-initiate the data-at-execution sequence.

If an application calls SQLCancel() on an hstmt not associated with a data-at-execution sequence, SQLCancel() has the same effect as SQLFreeStmt() with the SQL_CLOSE option. Applications should not call SQLCancel() to close a cursor; but rather SQLFreeStmt() should be used.

Return Codes

- SQL_SUCCESS
- SQL_INVALID_HANDLE
- SQL_ERROR

Diagnostics

Table 21. SQLCancel SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.

Restrictions

DB2 CLI does not support asynchronous statement execution.

Example

Refer to “Example” on page 289.

References

- “SQLPutData - Passing Data Value for A Parameter” on page 287
- “SQLParamData - Get Next Parameter For Which A Data Value Is Needed” on page 257

SQLColAttributes - Get Column Attributes

Purpose

Specification:	ODBC 1.0	X/OPEN CLI [1]	ISO CLI [1]

SQLColAttributes() is used to get an attribute for a column of the result set, and can also be used to determine the number of columns. SQLColAttributes() is a more extensible alternative to the SQLDescribeCol() function, but can only return one attribute per call.

Either SQLPrepare() or SQLExecDirect() must be called before calling this function.

If the application does not know the various attributes (such as, data type and length) of the column, this function (or SQLDescribeCol()) must be called before binding, via SQLBindCol(), to any columns.

Note: **1** - X/Open and ISO define this function with a singular name, SQLColAttribute().

Syntax

```
SQLRETURN   SQLColAttributes (SQLHSTMT          hstmt,
                              SQLUSMALLINT      icol,
                              SQLUSMALLINT      fDescType,
                              SQLPOINTER        rgbDesc,
                              SQLSMALLINT       cbDescMax,
                              SQLSMALLINT FAR   *pcbDesc,
                              SQLINTEGER  FAR   *pfDesc);
```

Function Arguments

Table 22. SQLColAttributes Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle
SQLUSMALLINT	*icol*	input	Column number in result set (must be between 1 and the number of columns in the results set inclusive). This argument is ignored when SQL_COLUMN_COUNT is specified.
SQLUSMALLINT	*fDescType*	input	The supported values are described in Table 23 on page 105.
SQLCHAR *	*rgbDesc*	output	Pointer to buffer for string column attributes
SQLSMALLINT	*cbDescMax*	input	Length of *rgbDesc* descriptor buffer.
SQLSMALLINT *	*pcbDesc*	output	Actual number of bytes returned in *rgbDesc* buffer. If this argument contains a value equal to or greater than the length specified in *cbDescMax*, truncation occurred. The column attribute value is then truncated to *cbDescMax* bytes minus the size of the null-terminator (or to *cbDescMax* bytes if null termination is off).
SQLINTEGER *	*pfDesc*	output	Pointer to integer which holds the value of numeric column attributes.

The following values can be specified for the *fDescType* argument:

Table 23 (Page 1 of 3). fDescType descriptor types

Descriptor	Description
SQL_COLUMN_AUTO_INCREMENT	Indicates if the column data type is an auto increment data type. FALSE is returned in *pfDesc* for all DB2 SQL data types.
SQL_COLUMN_CASE_SENSITIVE	Indicates if the column data type is a case sensitive data type. Either TRUE or FALSE is returned in *pfDesc* depending on the data type. Case sensitivity does not apply to graphic data types, FALSE is returned. FALSE is returned for non-character data types.
SQL_COLUMN_CATALOG_NAME (SQL_COLUMN_QUALIFIER_NAME)	The catalog of the table that contains the column is returned in *rgbDesc*. An empty string is returned since DB2 CLI only supports two-part naming for a table. SQL_COLUMN_QUALIFIER_NAME is defined for compatibility with ODBC. DB2 CLI applications should use SQL_COLUMN_CATALOG_NAME.
SQL_COLUMN_COUNT	The number of columns in the result set is returned in *pfDesc*.
SQL_COLUMN_DISPLAY_SIZE	The maximum number of bytes needed to display the data in character form is returned in *pfDesc*. Refer to Table 147 on page 423 for the display size of each of the column types.
SQL_COLUMN_DISTINCT_TYPE	The user defined distinct type name of the column is returned in *rgbDesc*. If the column is a built-in SQL type and not a user defined distinct type, an empty string is returned. **Note:** This is an IBM-defined extension to the list of descriptor attributes defined by ODBC.
SQL_COLUMN_LABEL	The column label is returned in *rgbDesc*. If the column does not have a label, the column name or the column expression is returned. If the column is unlabeled and unnamed, an empty string is returned.

Table 23 (Page 2 of 3). fDescType descriptor types

Descriptor	Description
SQL_COLUMN_LENGTH	The number of *bytes* of data associated with the column is returned in *pfDesc*. This is the length in bytes of data transferred on the fetch or `SQLGetData()` for this column if SQL_C_DEFAULT is specified as the C data type. Refer to Table 146 on page 422 for the length of each of the SQL data types. If the column identified in *icol* is a fixed length character or binary string, (for example, SQL_CHAR or SQL_BINARY) the actual length is returned. If the column identified in *icol* is a variable length character or binary string, (for example, SQL_VARCHAR) the maximum length is returned.
SQL_COLUMN_MONEY	Indicates if the column data type is a money data type. FALSE is returned in *pfDesc* for all DB2 SQL data types.
SQL_COLUMN_NAME	The name of the column *icol* is returned in *rgbDesc*. If the column is an expression, then the result returned is product specific.
SQL_COLUMN_NULLABLE	If the column identified by *icol* can contain nulls, then SQL_NULLABLE is returned in *pfDesc*. If the column is constrained not to accept nulls, then SQL_NO_NULLS is returned in *pfDesc*.
SQL_COLUMN_PRECISION	The precision in units of digits is returned in *pfDesc* if the column is SQL_DECIMAL, SQL_NUMERIC, SQL_DOUBLE, SQL_FLOAT, SQL_INTEGER, SQL_REAL or SQL_SMALLINT. If the column is a character SQL data type, then the precision returned in *pfDesc*, indicates the maximum number of *characters* the column can hold. If the column is a graphic SQL data type, then the precision returned in *pfDesc*, indicates the maximum number of double-byte *characters* the column can hold. Refer to Table 144 on page 420 for the precision of each of the SQL data types.
SQL_COLUMN_SCALE	The scale attribute of the column is returned. Refer to Table 145 on page 421 for the scale of each of the SQL data types.
SQL_COLUMN_SCHEMA_NAME (SQL_COLUMN_OWNER_NAME)	The schema of the table that contains the column is returned in *rgbDesc*. An empty string is returned as DB2 CLI is unable to determine this attribute. SQL_COLUMN_OWNER_NAME is defined for compatibility with ODBC. DB2 CLI applications should use SQL_COLUMN_SCHEMA_NAME.

Table 23 (Page 3 of 3). fDescType descriptor types

Descriptor	Description
SQL_COLUMN_SEARCHABLE	Indicates if the column data type is searchable: • SQL_UNSEARCHABLE if the column cannot be used in a WHERE clause. • SQL_LIKE_ONLY if the column can be used in a WHERE clause only with the LIKE predicate. • SQL_ALL_EXCEPT_LIKE if the column can be used in a WHERE clause with all comparison operators except LIKE. • SQL_SEARCHABLE if the column can be used in a WHERE clause with any comparison operator.
SQL_COLUMN_TABLE_NAME	The name of the table that contains the column is returned in *rgbDesc*. An empty string is returned as DB2 CLI cannot determine this attribute.
SQL_COLUMN_TYPE	The SQL data type of the column identified in *icol* is returned in *pfDesc*. The possible values for *pfSqlType* are listed in Table 3 on page 40.
SQL_COLUMN_TYPE_NAME	The type of the column (as entered in an SQL statement) is returned in *rgbDesc*. For information on each data type refer to the TYPE_NAME attribute found in “Data Types and Data Conversion” on page 38.
SQL_COLUMN_UNSIGNED	Indicates if the column data type is an unsigned type or not. TRUE is returned in *pfDesc* for all non-numeric data types, FALSE is returned for all numeric data types.
SQL_COLUMN_UPDATABLE	Indicates if the column data type is an updateable data type. SQL_ATTR_READWRITE_UNKNOWN is returned in *pfDesc* for all DB2 SQL data types. SQL_ATTR_READONLY is returned if the column is obtained from a catalog function call.

Usage

Instead of returning a specific set of attributes like `SQLDescribeCol()`, `SQLColAttributes()` allows you to specify which attribute you wish to receive for a specific column. If the desired information is a string, it is returned in *rgbDesc*. If the desired information is a number, it is returned in *pfDesc*.

`SQLColAttributes()` is an extensible alternative to `SQLDescribeCol()`, which is used to return a fixed set of commonly used column attribute information.

If an *fDescType* descriptor type does not apply to the database server, an empty string is returned in *rgbDesc* or zero is returned in *pfDesc*, depending on the expected result of the descriptor.

Columns are identified by a number (numbered sequentially from left to right starting with 1) and can be described in any order.

Calling `SQLColAttributes()` with *fDescType* set to SQL_COLUMN_COUNT is an alternative to calling `SQLNumResultCols()` to determine whether any columns can be returned.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 24. SQLColAttributes SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The character string returned in the argument *rgbDesc* is longer than the value specified in the argument *cbDescMax*. The argument *pcbDesc* contains the actual length of the string to be returned. (Function returns SQL_SUCCESS_WITH_INFO.)
07005	The statement did not return a result set.	The statement associated with the *hstmt* did not return a result set. There are no columns to describe. To prevent encountering this error, call `SQLNumResultCols()` before calling `SQLColAttributes()`.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1002	Invalid column number.	The value specified for the argument *icol* is less than 1. The value specified for the argument *icol* is greater than the number of columns in the result set. Not returned if SQL_COLUMN_COUNT is specified.
S1010	Function sequence error.	The function is called prior to calling `SQLPrepare()` or `SQLExecDirect()` for the *hstmt*. The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.
S1090	Invalid string or buffer length.	The length specified in the argument *cbDescMax* is less than 0 and *fDescType* requires a character string be returned in *rgbDesc*.
S1091	Descriptor type out of range.	The value specified for the argument *fDescType* was not equal to a value specified in Table 23 on page 105.
S1C00	Driver not capable.	The SQL data type returned by the database server for column *icol* is not recognized by DB2 CLI.

Restrictions

None.

Example

Refer to “Example” on page 130

References

- “SQLDescribeCol - Describe Column Attributes” on page 128
- “SQLExecDirect - Execute a Statement Directly” on page 149
- “SQLPrepare - Prepare a Statement” on page 261
- “SQLSetColAttributes - Set Column Attributes” on page 292

SQLColumnPrivileges - Get Privileges Associated With The Columns of A Table

Purpose

Specification:	**ODBC** 1.0		

SQLColumnPrivileges() returns a list of columns and associated privileges for the specified table. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated from a query.

Syntax

```
SQLRETURN SQLColumnPrivileges  (SQLHSTMT        hstmt,
                                SQLCHAR    FAR  *szCatalogName,
                                SQLSMALLINT     cbCatalogName,
                                SQLCHAR    FAR  *szSchemaName,
                                SQLSMALLINT     cbSchemaName,
                                SQLCHAR    FAR  *szTableName,
                                SQLSMALLINT     cbTableName,
                                SQLCHAR    FAR  *szColumnName,
                                SQLSMALLINT     cbColumnName);
```

Function Arguments

Table 25. SQLColumnPrivileges Arguments

Data Type	**Argument**	**Use**	**Description**
SQLHSTMT	hstmt	input	Statement handle.
SQLCHAR *	szCatalogName	input	Catalog qualifier of a 3-part table name. This must be a NULL pointer or a zero length string.
SQLSMALLINT	cbCatalogName	input	Length of *szCatalogName*. This must be set to 0.
SQLCHAR *	szSchemaName	input	Schema qualifier of table name.
SQLSMALLINT	cbSchemaName	input	Length of *szSchemaName*.
SQLCHAR *	szTableName	input	Table name.
SQLSMALLINT	cbTableName	input	Length of *szTableName*.
SQLCHAR *	szColumnName	input	Buffer that can contain a *pattern-value* to qualify the result set by column name.
SQLSMALLINT	cbColumnName	input	Length of *szColumnName*.

Usage

The results are returned as a standard result set containing the columns listed in Table 26 on page 111. The result set is ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME, COLUMN_NAME, and PRIVILEGE. If multiple privileges are associated with any given column, each privilege is returned as a separate row. A typical application might wish to call this function after a call to SQLColumns() to determine column privilege information. The application should use the character strings returned in the TABLE_SCHEM, TABLE_NAME, COLUMN_NAME columns of the `SQLColumns()` result set as input arguments to this function.

Since calls to `SQLColumnPrivileges()` in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating the calls.

The VARCHAR columns of the catalog functions result set are declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call `SQLGetInfo()` with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Note that the *szColumnName* argument accepts a search pattern. For more information about valid search patterns, refer to “Input Arguments on Catalog Functions” on page 349.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change.

Table 26 (Page 1 of 2). Columns Returned By SQLColumnPrivileges

Column Number/Name	Data Type	Description
1 TABLE_CAT	VARCHAR(128)	This is always NULL.
2 TABLE_SCHEM	VARCHAR(128)	The name of the schema containing TABLE_NAME.
3 TABLE_NAME	VARCHAR(128) not NULL	Name of the table or view.
4 COLUMN_NAME	VARCHAR(128) not NULL	Name of the column of the specified table or view.
5 GRANTOR	VARCHAR(128)	Authorization ID of the user who granted the privilege.
6 GRANTEE	VARCHAR(128)	Authorization ID of the user to whom the privilege is granted.

Table 26 (Page 2 of 2). Columns Returned By SQLColumnPrivileges

Column Number/Name	Data Type	Description
7 PRIVILEGE	VARCHAR(128)	The column privilege. This can be: • ALTER • CONTROL • DELETE • INDEX • INSERT • REFERENCES • SELECT • UPDATE Supported privileges are based on the data source to which you are connected. **Note:** Most IBM RDBMSs do not offer column level privileges at the column level. DB2 for OS/390 and DB2 for VSE and VM support the UPDATE column privilege; there is one row in this result set for each updateable column. For all other privileges for DB2 for OS/390 and DB2 for VSE and VM, and for all privileges for other IBM RDBMSs, if a privilege has been granted at the table level, a row is present in this result set.
8 IS_GRANTABLE	VARCHAR(3)	Indicates whether the grantee is permitted to grant the privilege to other users. Either "YES", "NO".

Note: The column names used by DB2 CLI follow the X/Open CLI CAE specification style. The column types, contents and order are identical to those defined for the `SQLColumnPrivileges()` result set in ODBC.

If there is more than one privilege associated with a column, then each privilege is returned as a separate row in the result set.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 27 (Page 1 of 2). SQLColumnPrivileges SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	*szTableName* is NULL.
S1014	No more handles.	DB2 CLI is not able to allocate a handle due to internal resources.
S1090	Invalid string or buffer length.	The value of one of the name length arguments is less than 0, but not equal to SQL_NTS.

Table 27 (Page 2 of 2). SQLColumnPrivileges SQLSTATEs

SQLSTATE	Description	Explanation
S1C00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for table name.

Restrictions

None.

Example

```
/* ... */
SQLRETURN
list_column_privileges(SQLHDBC hdbc, SQLCHAR *schema, SQLCHAR *tablename )
{
/* ... */

    rc = SQLColumnPrivileges(hstmt, NULL, 0, schema, SQL_NTS,
                             tablename, SQL_NTS, columnname.s, SQL_NTS);

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) columnname.s, 129,
                    &columnname.ind);

    rc = SQLBindCol(hstmt, 5, SQL_C_CHAR, (SQLPOINTER) grantor.s, 129,
                    &grantor.ind);

    rc = SQLBindCol(hstmt, 6, SQL_C_CHAR, (SQLPOINTER) grantee.s, 129,
                    &grantee.ind);

    rc = SQLBindCol(hstmt, 7, SQL_C_CHAR, (SQLPOINTER) privilege.s, 129,
                    &privilege.ind);

    rc = SQLBindCol(hstmt, 8, SQL_C_CHAR, (SQLPOINTER) is_grantable.s, 4,
                    &is_grantable.ind);


    printf("Column Privileges for %s.%s\n", schema, tablename);
    /* Fetch each row, and display */
    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
        sprintf(cur_name, " Column: %s\n",  columnname.s);
        if (strcmp(cur_name, pre_name) != 0) {
            printf("\n%s\n", cur_name);
            printf("   Grantor         Grantee         Privilege  Grantable\n");
            printf("   --------------- --------------- ---------- ---\n");
        }
        strcpy(pre_name, cur_name);
        printf("   %-15s", grantor.s);
        printf(" %-15s", grantee.s);
        printf(" %-10s", privilege.s);
        printf(" %-3s\n", is_grantable.s);
    }                           /* endwhile */
/* ... */
```

References

- “SQLColumns - Get Column Information for a Table” on page 115
- “SQLTables - Get Table Information” on page 334

SQLColumns - Get Column Information for a Table

Purpose

Specification:	**ODBC** 1.0	**X/OPEN CLI**	

SQLColumns() returns a list of columns in the specified tables. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to fetch a result set generated by a query.

Syntax

```
SQLRETURN   SQLColumns       (SQLHSTMT          hstmt,
                              SQLCHAR     FAR   *szCatalogName,
                              SQLSMALLINT       cbCatalogName,
                              SQLCHAR     FAR   *szSchemaName,
                              SQLSMALLINT       cbSchemaName,
                              SQLCHAR     FAR   *szTableName,
                              SQLSMALLINT       cbTableName,
                              SQLCHAR     FAR   *szColumnName,
                              SQLSMALLINT       cbColumnName);
```

Function Arguments

Table 28. SQLColumns Arguments

Data Type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle.
SQLCHAR *	szCatalogName	input	Buffer that can contain a *pattern-value* to qualify the result set. *Catalog* is the first part of a 3-part table name. This must be a NULL pointer or a zero length string.
SQLSMALLINT	cbCatalogName	input	Length of *szCatalogName*. This must be set to 0.
SQLCHAR *	szSchemaName	input	Buffer that can contain a *pattern-value* to qualify the result set by schema name.
SQLSMALLINT	cbSchemaName	input	Length of *szSchemaName*.
SQLCHAR *	szTableName	input	Buffer that can contain a *pattern-value* to qualify the result set by table name.
SQLSMALLINT	cbTableName	input	Length of *szTableName*.
SQLCHAR *	szColumnName	input	Buffer that can contain a *pattern-value* to qualify the result set by column name.
SQLSMALLINT	cbColumnName	input	Length of *szColumnName*.

Usage

This function is called to retrieve information about the columns of either a table or a set of tables. A typical application might wish to call this function after a call to SQLTables() to determine the columns of a table. The application should use the character strings returned in the TABLE_SCHEMA and TABLE_NAME columns of the SQLTables() result set as input to this function.

SQLColumns() returns a standard result set, ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and ORDINAL_POSITION. Table 29 on page 116 lists the columns in the result set.

The *szSchemaName, szTableName*, and *szColumnName* arguments accept search patterns. For more information about valid search patterns, see "Input Arguments on Catalog Functions" on page 349.

Since calls to SQLColumns() in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set are declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call SQLGetInfo() with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_OWNER_SCHEMA_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change.

Table 29 (Page 1 of 3). Columns Returned By SQLColumns

Column Number/Name	Data Type	Description
1 TABLE_CAT	VARCHAR(128)	This is always NULL.
2 TABLE_SCHEM	VARCHAR(128)	The name of the schema containing TABLE_NAME.
3 TABLE_NAME	VARCHAR(128) NOT NULL	Name of the table, view, alias, or synonym.
4 COLUMN_NAME	VARCHAR(128) NOT NULL	Column identifier. Name of the column of the specified table, view, alias, or synonym.
5 DATA_TYPE	SMALLINT NOT NULL	SQL data type of column identified by COLUMN_NAME. This is one of the values in the Symbolic SQL Data Type column in Table 3 on page 40.
6 TYPE_NAME	VARCHAR(128) NOT NULL	Character string representing the name of the data type corresponding to DATA_TYPE.
7 COLUMN_SIZE	INTEGER	If the DATA_TYPE column value denotes a character or binary string, then this column contains the maximum length in characters for the column. For date, time, timestamp data types, this is the total number of characters required to display the value when converted to character. For numeric data types, this is either the total number of digits, or the total number of bits allowed in the column, depending on the value in the NUM_PREC_RADIX column in the result set. See also, Table 144 on page 420.

Table 29 (Page 2 of 3). Columns Returned By SQLColumns

Column Number/Name	Data Type	Description
8 BUFFER_LENGTH	INTEGER	The maximum number of bytes for the associated C buffer to store data from this column if SQL_C_DEFAULT is specified on the SQLBindCol(), SQLGetData() and SQLBindParameter() calls. This length does not include any null-terminator. For exact numeric data types, the length accounts for the decimal and the sign. See also, Table 146 on page 422.
9 DECIMAL_DIGITS	SMALLINT	The scale of the column. NULL is returned for data types where scale is not applicable. See also, Table 145 on page 421.
10 NUM_PREC_RADIX	SMALLINT	Either 10 or 2 or NULL. If DATA_TYPE is an approximate numeric data type, this column contains the value 2, then the COLUMN_SIZE column contains the number of bits allowed in the column. If DATA_TYPE is an exact numeric data type, this column contains the value 10 and the COLUMN_SIZE contains the number of decimal digits allowed for the column. For numeric data types, the DBMS can return a NUM_PREC_RADIX of either 10 or 2. NULL is returned for data types where radix is not applicable.
11 NULLABLE	SMALLINT NOT NULL	SQL_NO_NULLS if the column does not accept NULL values. SQL_NULLABLE if the column accepts NULL values.
12 REMARKS	VARCHAR(254)	Might contain descriptive information about the column.
13 COLUMN_DEF	VARCHAR(254)	The column's default value. If the default value is a numeric literal, then this column contains the character representation of the numeric literal with no enclosing single quotes. If the default value is a character string, then this column is that string enclosed in single quotes. If the default value is a *pseudo-literal*, such as for DATE, TIME, and TIMESTAMP columns, then this column contains the keyword of the pseudo-literal (for example, CURRENT DATE) with no enclosing quotes. If NULL was specified as the default value, then this column returns the word NULL, not enclosed in quotes. If the default value cannot be represented without truncation, then this column contains TRUNCATED with no enclosing single quotes. If no default value was specified, then this column is NULL.
14 DATETIME_CODE	INTEGER	This column is currently NULL.
15 CHAR_OCTET_LENGTH	INTEGER	Contains the maximum length in octets for a character data type column. For Single Byte character sets, this is the same as COLUMN_SIZE. For all other data types it is NULL.
16 ORDINAL_POSITION	INTEGER NOT NULL	The ordinal position of the column in the table. The first column in the table is number 1.
17 IS_NULLABLE	VARCHAR(254)	Contains the string 'NO' if the column is known to be not nullable; and 'YES' otherwise.

Table 29 (Page 3 of 3). Columns Returned By SQLColumns

Column Number/Name	Data Type	Description

Note: This result set is identical to the X/Open CLI `Columns()` result set specification, which is an extended version of the `SQLColumns()` result set specified in ODBC V2. The ODBC `SQLColumns()` result set includes every column in the same position up to the REMARKS column.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 30. SQLColumns SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1014	No more handles.	DB2 CLI is not able to allocate a handle due to internal resources.
S1090	Invalid string or buffer length.	The value of one of the name length arguments is less than 0, but not equal SQL_NTS.
S1C00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for table name.

Restrictions

None.

Example

```
/* ... */
SQLRETURN
list_columns(SQLHDBC hdbc, SQLCHAR *schema, SQLCHAR *tablename )
{
/* ... */

    rc = SQLColumns(hstmt, NULL, 0, schema, SQL_NTS,
                    tablename, SQL_NTS, "%", SQL_NTS);

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) column_name.s, 129,
                    &column_name.ind);

    rc = SQLBindCol(hstmt, 6, SQL_C_CHAR, (SQLPOINTER) type_name.s, 129,
                    &type_name.ind);

    rc = SQLBindCol(hstmt, 7, SQL_C_LONG, (SQLPOINTER) &length,
                    sizeof(length), &length_ind);

    rc = SQLBindCol(hstmt, 9, SQL_C_SHORT, (SQLPOINTER) &scale,
                    sizeof(scale), &scale_ind);

    rc = SQLBindCol(hstmt, 12, SQL_C_CHAR, (SQLPOINTER) remarks.s, 129,
                    &remarks.ind);

    rc = SQLBindCol(hstmt, 11, SQL_C_SHORT, (SQLPOINTER) & nullable,
                    sizeof(nullable), &nullable_ind);

    printf("Schema: %s  Table Name: %s\n", schema, tablename);
    /* Fetch each row, and display */
    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
        printf("   %s", column_name.s);
        if (nullable == SQL_NULLABLE) {
            printf(", NULLABLE");
        } else {
            printf(", NOT NULLABLE");
        }
        printf(", %s", type_name.s);
        if (length_ind != SQL_NULL_DATA) {
            printf(" (%ld", length);
        } else {
            printf("(\n");
        }
        if (scale_ind != SQL_NULL_DATA) {
            printf(", %d)\n", scale);
        } else {
            printf(")\n");
        }
    }                           /* endwhile */
/* ... */
```

References

- “SQLTables - Get Table Information” on page 334
- “SQLColumnPrivileges - Get Privileges Associated With The Columns of A Table” on page 110
- “SQLSpecialColumns - Get Special (Row Identifier) Columns” on page 320

SQLConnect - Connect to a Data Source

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

`SQLConnect()` establishes a connection to the target database. The application must supply a target SQL database.

`SQLAllocConnect()` must be called before calling this function.

This function must be called before calling `SQLAllocStmt()`.

Syntax

```
SQLRETURN   SQLConnect       (SQLHDBC           hdbc,
                              SQLCHAR     FAR   *szDSN,
                              SQLSMALLINT       cbDSN,
                              SQLCHAR     FAR   *szUID,
                              SQLSMALLINT       cbUID,
                              SQLCHAR     FAR   *szAuthStr,
                              SQLSMALLINT       cbAuthStr);
```

Function Arguments

Table 31. SQLConnect Arguments

Data Type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Connection handle
SQLCHAR *	*szDSN*	input	Data Source: The name or alias-name of the database.
SQLSMALLINT	*cbDSN*	input	Length of contents of *szDSN* argument
SQLCHAR *	*szUID*	input	Authorization-name (user identifier). This parameter is validated but not used.
SQLSMALLINT	*cbUID*	input	Length of contents of *szUID* argument. This parameter is validated but not used.
SQLCHAR *	*szAuthStr*	input	Authentication-string (password). This parameter is validated but not used.
SQLSMALLINT	*cbAuthStr*	input	Length of contents of *szAuthStr* argument. This parameter is validated but not used.

Usage

The target database (also known as *data source*) for IBM RDBMSs is the location name as defined in SYSIBM.LOCATIONS. The application can obtain a list of databases available to connect to by calling `SQLDataSources()`.

The input length arguments to `SQLConnect()` (*cbDSN*, *cbUID*, *cbAuthStr*) can be set to the actual length of their associated data (not including any null-terminating character) or to SQL_NTS to indicate that the associated data is null-terminated.

The *szDSN* and *szUID* argument values must not contain any blanks. If these values are specified, they are ignored. The semantics of *szDSN* are as follows:

- If *szDSN* is not NULL and *cbDSN* is not 0, then DB2 CLI issues a CONNECT TO the data source.
- If *szDSN* is not NULL and *cbDSN* is 0, then DB2 CLI issues a CONNECT RESET, that is, a CONNECT to the local DB2 subsystem.
- If *szDSN* is NULL, then CONNECT.

  The latter usage is referred to as a `'NULL'` CONNECT and is required when the application is executing as a stored procedure. In this case it cannot connect to a data source but requires a valid connect handle for the DB2 subsystem.

  The CONNECT type (CONNECT (Type 1), CONNECT (Type 2)) is specified as described in "CONNECT Type 1 and Type 2" on page 26.

Use the more extensible `SQLDriverConnect()` function to connect when the application needs to override any or all of the keyword values specified for this data source in the initialization file.

Various connection characteristics (options) can be specified by the end user in the section of the initialization file associated with the `szDSN` data source argument or set by the application using `SQLSetConnectOption()`. The extended connect function, `SQLDriverConnect()`, can be called with additional connect options and can also perform a null connect.

Stored procedures written using DB2 CLI must make a *null* `SQLConnect()` call. A null `SQLConnect()` is where the *szDSN* argument pointer is set to NULL and the length argument is set to 0. A null `SQLConnect()` still requires `SQLAllocEnv()` and `SQLAllocConnect()` be called first, but does not require that `SQLTransact()` be called before `SQLDisconnect()`. For more information, refer to "Stored Procedure" on page 472.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 32 (Page 1 of 2). SQLConnect SQLSTATEs

SQLSTATE	Description	Explanation
08001	Unable to connect to data source.	DB2 CLI is not able to establish a connection with the data source (server). The connection request is rejected because an existing connection established via embedded SQL already exists.
08002	Connection in use.	The specified *hdbc* is already being used to establish a connection with a data source and the connection is still open.
08004	The application server rejected establishment of the connection.	The data source (server) rejected the establishment of the connection. The number of connections specified by the MAXCONN keyword has been reached.
58004	Unexpected system failure.	Unrecoverable system error.

Table 32 (Page 2 of 2). SQLConnect SQLSTATEs

SQLSTATE	Description	Explanation
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	A non-matching double quote (") is found in either the *szDSN*, *szUID*, or *szAuthStr* argument.
S1090	Invalid string or buffer length.	The value specified for argument *cbDSN* is less than 0, but not equal to SQL_NTS and the argument *szDSN* is not a null pointer. The value specified for argument *cbUID* is less than 0, but not equal to SQL_NTS and the argument *szUID* is not a null pointer. The value specified for argument *cbAuthStr* is less than 0, but not equal to SQL_NTS and the argument *szAuthStr* is not a null pointer.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.
S1501	Invalid data source name.	An invalid data source name is specified in argument *szDSN*.

Restrictions

The implicit connection (or default database) option for IBM RDBMSs is not supported. `SQLConnect()` must be called before any SQL statements can be executed.

Example

```
/* ... */
/* Global Variables for user id and password, defined in main module.
   To keep samples simple, not a recommended practice.
   The INIT_UID_PWD macro is used to initialize these variables.
*/
extern    SQLCHAR   server[SQL_MAX_DSN_LENGTH + 1];
/********************************************************************/
SQLRETURN
DBconnect(SQLHENV henv,
          SQLHDBC * hdbc)
{
    SQLRETURN       rc;
    SQLSMALLINT     outlen;

    /* allocate a connection handle     */
    if (SQLAllocConnect(henv, hdbc) != SQL_SUCCESS) {
        printf(">---ERROR while allocating a connection handle-----\n");
        return (SQL_ERROR);
    }
    /* Set AUTOCOMMIT OFF */
    rc = SQLSetConnectOption(*hdbc, SQL_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF);
    if (rc != SQL_SUCCESS) {
        printf(">---ERROR while setting AUTOCOMMIT OFF ------------\n");
        return (SQL_ERROR);
    }
    rc = SQLConnect(*hdbc, server, SQL_NTS, NULL, SQL_NTS, NULL, SQL_NTS);
    if (rc != SQL_SUCCESS) {
        printf(">--- Error while connecting to database: %s -------\n", server);
        SQLDisconnect(*hdbc);
        SQLFreeConnect(*hdbc);
        return (SQL_ERROR);
    } else {                    /* Print Connection Information */
        printf(">Connected to %s\n", server);
    }
    return (SQL_SUCCESS);
}

/*******************************************************************/
/* DBconnect2 - Connect with connect type                          */
/* Valid connect types SQL_CONCURRENT_TRANS, SQL_COORDINATED_TRANS  */
/*******************************************************************/
SQLRETURN DBconnect2(SQLHENV henv,
           SQLHDBC * hdbc, SQLINTEGER contype)
           SQLHDBC * hdbc, SQLINTEGER contype, SQLINTEGER conphase)
{
    SQLRETURN       rc;
    SQLSMALLINT     outlen;

    /* allocate a connection handle     */
    if (SQLAllocConnect(henv, hdbc) != SQL_SUCCESS) {
        printf(">---ERROR while allocating a connection handle-----\n");
        return (SQL_ERROR);
    }
    /* Set AUTOCOMMIT OFF */
    rc = SQLSetConnectOption(*hdbc, SQL_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF);
    if (rc != SQL_SUCCESS) {
        printf(">---ERROR while setting AUTOCOMMIT OFF ------------\n");
        return (SQL_ERROR);
    }
    rc = SQLSetConnectOption(hdbc[0], SQL_CONNECTTYPE, contype);
    if (rc != SQL_SUCCESS) {
        printf(">---ERROR while setting Connect Type -------------\n");
        return (SQL_ERROR);
    }
    if (contype == SQL_COORDINATED_TRANS ) {
        rc = SQLSetConnectOption(hdbc[0], SQL_SYNC_POINT, conphase);
        if (rc != SQL_SUCCESS) {
            printf(">---ERROR while setting Syncpoint Phase --------\n");
            return (SQL_ERROR);
        }
    }
    rc = SQLConnect(*hdbc, server, SQL_NTS, NULL, SQL_NTS, NULL, SQL_NTS);
    if (rc != SQL_SUCCESS) {
        printf(">--- Error while connecting to database: %s -------\n", server);
        SQLDisconnect(*hdbc);
        SQLFreeConnect(*hdbc);
        return (SQL_ERROR);
    } else {                    /* Print Connection Information */
        printf(">Connected to %s\n", server);
    }
    return (SQL_SUCCESS);

}
/* ... */
```

References

- “SQLAllocConnect - Allocate Connection Handle” on page 78
- “SQLDriverConnect - (Expanded) Connect to a Data Source” on page 137
- “SQLSetConnectOption - Set Connection Option” on page 298
- “SQLGetConnectOption - Returns Current Setting of A Connect Option” on page 185
- “SQLAllocStmt - Allocate a Statement Handle” on page 84
- “SQLDataSources - Get List of Data Sources” on page 125
- “SQLDisconnect - Disconnect from a Data Source” on page 135

SQLDataSources - Get List of Data Sources

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLDataSources() returns a list of target databases available, one at a time.

SQLDataSources() is usually called before a connection is made, to determine the databases that are available to connect to.

Syntax

```
SQLRETURN   SQLDataSources   (SQLHENV          henv,
                              SQLUSMALLINT     fDirection,
                              SQLCHAR     FAR  *szDSN,
                              SQLSMALLINT      cbDSNMax,
                              SQLSMALLINT FAR  *pcbDSN,
                              SQLCHAR     FAR  *szDescription,
                              SQLSMALLINT      cbDescriptionMax,
                              SQLSMALLINT FAR  *pcbDescription);
```

Function Arguments

Table 33. SQLDataSources Arguments

Data Type	Argument	Use	Description
SQLHENV	*henv*	input	Environment handle.
SQLUSMALLINT	*fDirection*	input	Used by application to request the first data source name in the list or the next one in the list. *fDirection* can take on only the following values: • SQL_FETCH_FIRST • SQL_FETCH_NEXT
SQLCHAR *	*szDSN*	output	Pointer to buffer to hold the data source name retrieved.
SQLSMALLINT	*cbDSNMax*	input	Maximum length of the buffer pointed to by *szDSN*. This should be less than or equal to SQL_MAX_DSN_LENGTH + 1.
SQLSMALLINT *	*pcbDSN*	output	Pointer to location where the maximum number of bytes available to return in the *szDSN* are stored.
SQLCHAR *	*szDescription*	output	Pointer to buffer where the description of the data source is returned. DB2 CLI returns the ***Comment*** field associated with the database cataloged to the DBMS.
Note: IBM RDBMSs always return blank padded to 30 bytes.			
SQLSMALLINT	*cbDescriptionMax*	input	Maximum length of the *szDescription* buffer. DB2 for OS/390 always returns NULL.
SQLSMALLINT *	*pcbDescription*	output	Pointer to location where this function returns the actual number of bytes available to return for the description of the data source. DB2 for OS/390 always returns zero.

Usage

The application can call this function any time with *fDirection* set to either SQL_FETCH_FIRST or SQL_FETCH_NEXT.

If SQL_FETCH_FIRST is specified, the first database in the list is always returned.

If SQL_FETCH_NEXT is specified:

- Directly following a SQL_FETCH_FIRST call, the second database in the list is returned
- Before any other `SQLDataSources()` call, the first database in the list is returned
- When there are no more databases in the list, SQL_NO_DATA_FOUND is returned. If the function is called again, the first database is returned.
- Any other time, the next database in the list is returned.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

Diagnostics

Table 34. SQLDataSources SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The data source name returned in the argument *szDSN* is longer than the value specified in the argument *cbDSNMax*. The argument *pcbDSN* contains the length of the full data source name. (Function returns SQL_SUCCESS_WITH_INFO.) The data source name returned in the argument *szDescription* is longer than the value specified in the argument *cbDescriptionMax*. The argument *pcbDescription* contains the length of the full data source description. (Function returns SQL_SUCCESS_WITH_INFO.)
58004	Unexpected system failure.	Unrecoverable system error.
S1000	General error.	An error occurred for which there is no specific SQLSTATE and for which no specific SQLSTATE is defined. The error message returned by SQLError in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.
S1090	Invalid string or buffer length.	The value specified for argument *cbDSNMax* is less than 0. The value specified for argument *cbDescriptionMax* is less than 0.
S1103	Direction option out of range.	The value specified for the argument *fDirection* is not equal to SQL_FETCH_FIRST or SQL_FETCH_NEXT.

Restrictions

None.

Example

```
/* ... */
/*******************************************************
**    - demonstrate SQLDataSource function
**    - list available servers
**      (error checking has been ignored for simplicity)
**
**  Functions used:
**
**    SQLAllocEnv      SQLFreeEnv
**    SQLDataSources
********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "sqlcli1.h"

int
main()
{
    SQLRETURN       rc;
    SQLHENV         henv;
    SQLCHAR         source[SQL_MAX_DSN_LENGTH + 1], description[255];
    SQLSMALLINT     buffl, desl;

    SQLAllocEnv(&henv);         /* allocate an environment handle   */

    /* list the available data sources (servers)         */
    printf("The following data sources are available:\n");
    printf("ALIAS NAME                      Comment(Description)\n");
    printf("-------------------------------------------------\n");

    while ((rc = SQLDataSources(henv, SQL_FETCH_NEXT, source,
                   SQL_MAX_DSN_LENGTH + 1, &buffl, description, 255, &desl))
           != SQL_NO_DATA_FOUND) {
        printf("%-30s  %s\n", source, description);
    }

    SQLFreeEnv(henv);

    return (SQL_SUCCESS);
}
/* ... */
```

References

None.

SQLDescribeCol - Describe Column Attributes

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLDescribeCol() returns a set of commonly used descriptor information (column name, type, precision, scale, nullability) for the indicated column in the result set generated by a query.

If the application needs only one attribute of the descriptor information, or needs an attribute not returned by SQLDescribeCol(), the SQLColAttributes() function can be used in place of SQLDescribeCol(). Refer to "SQLColAttributes - Get Column Attributes" on page 104 for more information.

Either SQLPrepare() or SQLExecDirect() must be called before calling this function.

This function (or SQLColAttributes()) is usually called before a bind column function SQLBindCol() to determine the attributes of a column before binding it to an application variable.

Syntax

```
SQLRETURN   SQLDescribeCol   (SQLHSTMT          hstmt,
                              SQLUSMALLINT      icol,
                              SQLCHAR     FAR   *szColName,
                              SQLSMALLINT       cbColNameMax,
                              SQLSMALLINT FAR   *pcbColName,
                              SQLSMALLINT FAR   *pfSqlType,
                              SQLUINTEGER FAR   *pcbColDef,
                              SQLSMALLINT FAR   *pibScale,
                              SQLSMALLINT FAR   *pfNullable);
```

Function Arguments

Table 35 (Page 1 of 2). SQLDescribeCol Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle.
SQLUSMALLINT	*icol*	input	Column number to be described. Columns are numbered sequentially from left to right, starting at 1.
SQLCHAR *	*szColName*	output	Pointer to column name buffer. Set this to NULL if column name is not needed.
SQLSMALLINT	*cbColNameMax*	input	Size of *szColName* buffer.
SQLSMALLINT *	*pcbColName*	output	Bytes available to return for *szColName* argument. Truncation of column name (*szColName*) to *cbColNameMax - 1* bytes occurs if *pcbColName* is greater than or equal to *cbColNameMax.*
SQLSMALLINT *	*pfSqlType*	output	Base SQL data type of column. Refer to the Symbolic SQL Data Type column of Table 3 on page 40 for the data types that are supported.
SQLUINTEGER *	*pcbColDef*	output	Precision of column as defined in the database.

Table 35 (Page 2 of 2). SQLDescribeCol Arguments

Data Type	Argument	Use	Description
SQLSMALLINT *	*pibScale*	output	Scale of column as defined in the database (only applies to SQL_DECIMAL, SQL_NUMERIC, SQL_TIMESTAMP). Refer to Table 145 on page 421 for the scale of each of the SQL data types.
SQLSMALLINT *	*pfNullable*	output	Indicates whether NULLS are allowed for this column • SQL_NO_NULLS • SQL_NULLABLE

Usage

Columns are identified by a number, are numbered sequentially from left to right starting with 1, and can be described in any order.

If a null pointer is specified for any of the pointer arguments, DB2 CLI assumes that the information is not needed by the application and nothing is returned.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

If `SQLDescribeCol()` returns either SQL_ERROR, or SQL_SUCCESS_WITH_INFO, one of the following SQLSTATEs can be obtained by calling the `SQLError()` function.

Table 36 (Page 1 of 2). SQLDescribeCol SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The column name returned in the argument *szColName* is longer than the value specified in the argument *cbColNameMax*. The argument *pcbColName* contains the length of the full column name. (Function returns SQL_SUCCESS_WITH_INFO.)
07005	The statement did not return a result set.	The statement associated with the *hstmt* did not return a result set. There are no columns to describe. (Call `SQLNumResultCols()` first to determine if there are any rows in the result set.)
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1002	Invalid column number.	The value specified for the argument *icol* is less than 1. The value specified for the argument *icol* is greater than the number of columns in the result set.
S1090	Invalid string or buffer length.	The length specified in argument *cbColNameMax* is less than 1.

Table 36 (Page 2 of 2). SQLDescribeCol SQLSTATEs

SQLSTATE	Description	Explanation
S1010	Function sequence error.	The function is called prior to calling `SQLPrepare()` or `SQLExecDirect()` for the *hstmt*. The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.
S1C00	Driver not capable.	The SQL data type of column *icol* is not recognized by DB2 CLI.

Restrictions

The ODBC defined data type SQL_BIGINT is not supported.

Example

```
/* ... */
/*******************************************************************
** process_stmt
** - allocates a statement handle
** - executes the statement
** - determines the type of statement
**   - if there are no result columns, therefore non-select statement
**      - if rowcount > 0, assume statement was UPDATE, INSERT, DELETE
**     else
**      - assume a DDL, or Grant/Revoke statement
**     else
**      - must be a select statement.
**      - display results
** - frees the statement handle
*******************************************************************/

int
process_stmt(SQLHENV henv,
             SQLHDBC hdbc,
             SQLCHAR * sqlstr)
{
    SQLHSTMT        hstmt;
    SQLSMALLINT     nresultcols;
    SQLINTEGER      rowcount;
    SQLRETURN       rc;

    SQLAllocStmt(hdbc, &hstmt); /* allocate a statement handle */

    /* execute the SQL statement in "sqlstr"    */

    rc = SQLExecDirect(hstmt, sqlstr, SQL_NTS);
    if (rc != SQL_SUCCESS)
        if (rc == SQL_NO_DATA_FOUND) {
            printf("\nStatement executed without error, however,\n");
            printf("no data was found or modified\n");
            return (SQL_SUCCESS);
        } else
            check_error(henv, hdbc, hstmt, rc, __LINE__, __FILE__);
    rc = SQLNumResultCols(hstmt, &nresultcols);

    /* determine statement type */
    if (nresultcols == 0) {     /* statement is not a select statement */
        rc = SQLRowCount(hstmt, &rowcount);
        if (rowcount > 0) {     /* assume statement is UPDATE, INSERT, DELETE */
            printf("Statement executed, %ld rows affected\n", rowcount);
        } else {                /* assume statement is GRANT, REVOKE or a DLL
                                 * statement */
            printf("Statement completed successful\n");
        }
    } else {                    /* display the result set */
        display_results(hstmt, nresultcols);
    }                           /* end determine statement type */

    rc = SQLFreeStmt(hstmt, SQL_DROP); /* free statement handle */

    return (0);
}                               /* end process_stmt */

/*******************************************************************
** display_results
**
**  - for each column
**      - get column name
**      - bind column
**  - display column headings
**  - fetch each row
**      - if value truncated, build error message
**      - if column null, set value to "NULL"
**      - display row
**      - print truncation message
**  - free local storage
*******************************************************************/
display_results(SQLHSTMT hstmt,
                SQLSMALLINT nresultcols)
{
    SQLCHAR         colname[32];
    SQLSMALLINT     coltype;
    SQLSMALLINT     colnamelen;
    SQLSMALLINT     nullable;
    SQLINTEGER      collen[MAXCOLS];
    SQLUINTEGER     precision;
    SQLSMALLINT     scale;
    SQLINTEGER      outlen[MAXCOLS];
    SQLCHAR        *data[MAXCOLS];
    SQLCHAR         errmsg[256];
    SQLRETURN       rc;
    SQLINTEGER      i;
    SQLINTEGER      x;
    SQLINTEGER      displaysize;

    for (i = 0; i < nresultcols; i++) {
        SQLDescribeCol(hstmt, i + 1, colname, sizeof(colname),
                       &colnamelen, &coltype, &precision, &scale, NULL);
        collen[i] = precision; /* Note, assignment of unsigned int to signed */

        /* get display length for column */
        SQLColAttributes(hstmt, i + 1, SQL_COLUMN_DISPLAY_SIZE, NULL, 0,
                         NULL, &displaysize);
        /*
         * set column length to max of display length, and column name
         * length.  Plus one byte for null terminator
         */
        collen[i] = max(displaysize, strlen((char *) colname)) + 1;

        printf("%-*.*s", collen[i], collen[i], colname);

        /* allocate memory to bind column                             */
        data[i] = (SQLCHAR *) malloc(collen[i]);

        /* bind columns to program vars, converting all types to CHAR */
        rc = SQLBindCol(hstmt, i + 1, SQL_C_CHAR, data[i], collen[i], &outlen[i]);
    }
    printf("\n");

    /* display result rows                                            */
    while ((rc = SQLFetch(hstmt)) != SQL_NO_DATA_FOUND) {
        errmsg[0] = '\0';
        for (i = 0; i < nresultcols; i++) {
            /* Build a truncation message for any columns truncated */
            if (outlen[i] >= collen[i]) {
                sprintf((char *) errmsg + strlen((char *) errmsg),
                        "%ld chars truncated, col %d\n",
                        outlen[i] - collen[i] + 1, i + 1);
                sprintf((char *) errmsg + strlen((char *) errmsg),
                        "Bytes to return = %ld sixe of buffer\n",
                        outlen[i], collen[i]);
            }
            if (outlen[i] == SQL_NULL_DATA)
                printf("%-*.*s", collen[i], collen[i], "NULL");
            else
                printf("%-*.*s", collen[i], collen[i], data[i]);
        }                       /* for all columns in this row  */

        printf("\n%s", errmsg); /* print any truncation messages    */
    }                           /* while rows to fetch */

    /* free data buffers                                              */
    for (i = 0; i < nresultcols; i++) {
        free(data[i]);
    }

}                               /* end display_results */
/* ... */
```

References

- “SQLColAttributes - Get Column Attributes” on page 104
- “SQLExecDirect - Execute a Statement Directly” on page 149
- “SQLNumResultCols - Get Number of Result Columns” on page 255
- “SQLPrepare - Prepare a Statement” on page 261

SQLDescribeParam - Describe parameter marker

Purpose

Specification:	**ODBC 1.0**	**X/OPEN CLI**	**ISO CLI**

SQLDescribeParam() retrieves the description of a parameter marker associated with a prepared statement.

Either SQLPrepare() or SQLExecDirect() must be called before calling this function.

Syntax

```
SQLRETURN  SQLDescribeParam  (SQLHSTMT          hstmt,
                              SQLUSMALLINT      ipar,
                              SQLSMALLINT FAR    *pfSqlType,
                              SQLUINTEGER FAR    *pcbColDef,
                              SQLSMALLINT FAR    *pibScale,
                              SQLSMALLINT FAR    *pfNullable);
```

Function arguments

Table 37. SQLDescribeParam arguments

Data type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle.
SQLUSMALLINT	ipar	input	Parameter marker number ordered sequentially left to right in prepared SQL statement, starting from 1.
SQLSMALLINT *	pfSqlType	output	Base SQL data type.
SQLUINTEGER *	pcbColDef	output	Precision of the parameter marker. See Appendix E, “Data Conversion” on page 419 for more details on precision, scale, length, and display size.
SQLSMALLINT *	pibScale	output	Scale of the parameter marker. See Appendix E, “Data Conversion” on page 419 for more details on precision, scale, length, and display size.
SQLSMALLINT *	pfNullable	output	Indicates whether the parameter allows NULL values. Returns one of the following values: • SQL_NO_NULLS: The parameter does not allow NULL values (this is the defualt). • SQL_NULLABLE: The parameter allows NULL values. • SQL_NULLABLE_UNKNOWN: The driver cannot determine if the parameter allows NULL values.

Usage

For distinct types, SQLDescribeParam() returns both base data types for the input parameter.

For information about a parameter marker associated with the SQL CALL statement, use the SQLProcedureColumns() function.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 38. SQLDescribeParam SQLSTATEs

SQLSTATE	Description	Explanation
01000	Warning.	Informational message indicating an internal commit was issued on behalf of the application as part of the processing to set the specified connection option.
S1000	General error.	An error occurred for which there is no specific SQLSTATE and for which no specific SQLSTATE is defined. The error message returned by `SQLError()` in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1093	Invalid parameter number.	The value specified for the argument *ipar* is less than 1 or greater than the maximum number of parameters supported by the server.
S1C00	Driver not capable.	The data source does not support the description of input parameters.

Restrictions

None.

References

- “SQLBindParameter - Binds A Parameter Marker to a Buffer” on page 91
- “SQLCancel - Cancel Statement” on page 102
- “SQLExecDirect - Execute a Statement Directly” on page 149
- “SQLExecute - Execute a Statement” on page 154
- “SQLPrepare - Prepare a Statement” on page 261

SQLDisconnect - Disconnect from a Data Source

Purpose

Specification:	**ODBC** 1.0	**X/OPEN CLI**	**ISO CLI**

`SQLDisconnect()` closes the connection associated with the database connection handle.

`SQLTransact()` must be called before calling `SQLDisconnect()` if an outstanding transaction exists on this connection.

After calling this function, either call `SQLConnect()` to connect to another database, or call `SQLFreeConnect()`.

Syntax

```
SQLRETURN   SQLDisconnect    (SQLHDBC          hdbc);
```

Function Arguments

Table 39. SQLDisconnect Arguments

Data Type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Connection handle

Usage

If an application calls `SQLDisconnect()` before it has freed all the statement handles associated with the connection, DB2 CLI frees them after it successfully disconnects from the database.

If SQL_SUCCESS_WITH_INFO is returned, it implies that even though the disconnect from the database is successful, additional error or implementation specific information is available. For example, a problem was encountered on the clean up subsequent to the disconnect, or if there is no current connection because of an event that occurred independently of the application (such as communication failure).

After a successful `SQLDisconnect()` call, the application can re-use *hdbc* to make another `SQLConnect()` or `SQLDriverConnect()` request.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 40. SQLDisconnect SQLSTATEs

SQLSTATE	Description	Explanation
01002	Disconnect error.	An error occurred during the disconnect. However, the disconnect succeeded. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection is closed.	The connection specified in the argument *hdbc* is not open.
25000 **25**501	Invalid transaction state.	A transaction is in process on the connection specified by the argument *hdbc*. The transaction remains active, and the connection cannot be disconnected. **Note:** This error does not apply to stored procedures written in DB2 CLI.
25501	Invalid transaction state.	A transaction is in process on the connection specified by the argument *hdbc*. The transaction remains active, and the connection cannot be disconnected.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.

Restrictions

None.

Example

Refer to “Example” on page 79

References

- “SQLAllocConnect - Allocate Connection Handle” on page 78
- “SQLConnect - Connect to a Data Source” on page 120
- “SQLDriverConnect - (Expanded) Connect to a Data Source” on page 137
- “SQLTransact - Transaction Management” on page 338

SQLDriverConnect - (Expanded) Connect to a Data Source

Purpose

Specification:	**ODBC** 1.0		

SQLDriverConnect() is an alternative to SQLConnect(). Both functions establish a connection to the target database, but SQLDriverConnect() supports additional connection parameters.

Use SQLDriverConnect() when you want to pass any or all keyword values defined in the DB2 CLI initialization file.

When a connection is established, the completed connection string is returned. Applications can store this string for future connection requests. This allows you to override any or all keyword values in the DB2 CLI initialization file.

Syntax

Generic

```
SQLRETURN SQLDriverConnect   (SQLHDBC           hdbc,
                              SQLHWND           hwnd,
                              SQLCHAR      FAR  *szConnStrIn,
                              SQLSMALLINT       cbConnStrIn,
                              SQLCHAR      FAR  *szConnStrOut,
                              SQLSMALLINT       cbConnStrOutMax,
                              SQLSMALLINT  FAR  *pcbConnStrOut,
                              SQLUSMALLINT      fDriverCompletion);
```

Function Arguments

Table 41 (Page 1 of 2). SQLDriverConnect Arguments

Data Type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Connection handle.
SQLHWND	*hwindow*	input	NULL value. Not used.
SQLCHAR *	*szConnStrIn*	input	A full, partial or empty (null pointer) connection string (see syntax and description below).
SQLSMALLINT	*cbConnStrIn*	input	Length of *szConnStrIn*.
SQLCHAR *	*szConnStrOut*	output	Pointer to buffer for the completed connection string. If the connection is established successfully, this buffer contains the completed connection string. Applications should allocate at least SQL_MAX_OPTION_STRING_LENGTH bytes for this buffer.
SQLSMALLINT	*cbConnStrOutMax*	input	Maximum size of the buffer pointed to by *szConnStrOut*.
SQLCHAR *	*pcbConnStrOut*	output	Pointer to the number of bytes available to return in the *szConnStrOut* buffer. If the value of *pcbConnStrOut* is greater than or equal to *cbConnStrOutMax*, the completed connection string in *szConnStrOut* is truncated to *cbConnStrOutMax* - 1 bytes.

Table 41 (Page 2 of 2). SQLDriverConnect Arguments

Data Type	Argument	Use	Description
SQLUSMALLINT	*fDriverCompletion*	input	Indicates when DB2 CLI should prompt the user for more information. Possible values: • SQL_DRIVER_PROMPT • SQL_DRIVER_COMPLETE • SQL_DRIVER_COMPLETE_REQUIRED • SQL_DRIVER_NOPROMPT However, DB2 for OS/390 supports SQL_DRIVER_NOPROMPT only.

Usage

The connection string is used to pass one or more values needed to complete a connection.

```
          ┌─,──────────────────────────────────────┐
          ↓                                        │
►►────────┬─DSN──────────────────────────┬──=──attribute─┴──────────────────►◄
          ├─UID──────────────────────────┤
          ├─PWD──────────────────────────┤
          └─DB2 CLI-defined-keyword──────┘
```

Each keyword above has an attribute that is equal to the following:

DSN Data source name. The name or alias-name of the database. Required if *fDriverCompletion* is equal to SQL_DRIVER_NOPROMPT.

UID Authorization-name (user identifier). This value is ignored.

PWD The password corresponding to the authorization name. If there is no password for the user ID, an empty string is specified (PWD=;). This value is ignored.

The list of DB2 CLI defined keywords and their associated attribute values are discussed in “Initialization Keywords” on page 64. Any one of the keywords in that section can be specified on the connection string. If any keywords are repeated in the connection string, the value associated with the first occurrence of the keyword is used.

If any keywords exist in the DB2 CLI initialization file, the keywords and their respective values are used to augment the information passed to DB2 CLI in the connection string. If the information in the CLI initialization file contradicts information in the connection string, the values in connection string take precedence.

The application receives an error on any value of *fDriverCompletion* as follows:

SQL_DRIVER_PROMPT:
DB2 CLI returns SQL_ERROR.

SQL_DRIVER_COMPLETE:
DB2 CLI returns SQL_ERROR.

SQL_DRIVER_COMPLETE_REQUIRED:
DB2 CLI returns SQL_ERROR.

SQL_DRIVER_NOPROMPT:
The user is not prompted for any information. A connection is attempted with the information contained in the connection string. If there is not enough information, SQL_ERROR is returned.

When a connection is established, the complete connection string is returned.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA_FOUND
- SQL_INVALID_HANDLE
- SQL_ERROR

Diagnostics

All of the diagnostics generated by "SQLConnect - Connect to a Data Source" on page 120 can be returned here as well. The following table shows the additional diagnostics that can be returned.

Table 42. SQLDriverConnect SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The buffer *szConnstrOut* is not large enough to hold the entire connection string. The argument *pcbConnStrOut* contains the actual length of the connection string available for return. (Function returns SQL_SUCCESS_WITH_INFO)
01S00	Invalid connection string attribute.	An invalid keyword or attribute value is specified in the input connection string, but the connection to the data source is successful because one of the following occurred: • The unrecognized keyword is ignored. • The invalid attribute value is ignored, the default value is used instead. (Function returns SQL_SUCCESS_WITH_INFO)
# # **01**S02	Option value changed	SQL_CONNECTTYPE changed to SQL_CONCURRENT_TRANS when MULTICONTEXT=1 in use.
S1090	Invalid string or buffer length.	The value specified for *cbConnStrIn* is less than 0, but not equal to SQL_NTS. The value specified for *cbConnStrOutMax* is less than 0.
S1110	Invalid driver completion.	The value specified for the argument *fCompletion* is not equal to one of the valid values.

Restrictions

See restrictions described above for *fDriverCompletion* and *SQLHWND* parameters.

Example

```
 /******************************************************************/
 /* DB2 for OS/390 Example:                                        */
 /*   Issues SQLDriverConnect to pass a string of initialization   */
 /*   parameters to compliment the connection to the data source.  */
 /******************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "sqlcli1.h"

  /******************************************************************/
  /* SQLDriverConnect -----------                                   */
  /******************************************************************/

int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLRETURN       rc      = SQL_SUCCESS;
   SQLINTEGER      RETCODE = 0;

   char            *ConnStrIn =
                   "dsn=STLEC1;connecttype=2;bitdata=2;optimizefornrows=30";

   char            ConnStrOut [200];
   SQLSMALLINT     cbConnStrOut;
   int             i;
   char            *token;

   (void) printf ("**** Entering CLIP10.\n\n");

  /*****************************************************************/
  /* CONNECT to DB2 for OS/390                                     */
  /*****************************************************************/

   rc = SQLAllocEnv(&hEnv);

   if( rc != SQL_SUCCESS )
     goto dberror;

  /*****************************************************************/
  /* Allocate Connection Handle to DSN                             */
  /*****************************************************************/

   RETCODE = SQLAllocConnect(hEnv,
                             &hDbc);

   if( RETCODE != SQL_SUCCESS )      // Could not get a Connect Handle
     goto dberror;

  /*****************************************************************/
  /* Invoke SQLDriverConnect -----------                           */
  /*****************************************************************/

   RETCODE = SQLDriverConnect (hDbc                 ,
                               NULL                 ,
                               (SQLCHAR *)ConnStrIn ,
                               strlen(ConnStrIn)    ,
                               (SQLCHAR *)ConnStrOut,
                             sizeof(ConnStrOut)   ,
                             &cbConnStrOut        ,
                             SQL_DRIVER_NOPROMPT);
   if( RETCODE != SQL_SUCCESS )      // Could not get a Connect Handle
   {
     (void) printf ("**** Driver Connect Failed. rc = %d.\n", RETCODE);
     goto dberror;
   }

  /*****************************************************************/
  /* Enumerate keywords and values returned from SQLDriverConnect  */
  /*****************************************************************/

   (void) printf ("**** ConnStrOut = %s.\n", ConnStrOut);

   for (i = 1, token = strtok (ConnStrOut, ";");
        (token != NULL);
        token = strtok (NULL, ";"), i++)
     (void) printf ("**** Keyword # %d is: %s.\n", i, token);

  /*****************************************************************/
  /* DISCONNECT from data source                                   */
  /*****************************************************************/

   RETCODE = SQLDisconnect(hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Deallocate Connection Handle                                  */
  /*****************************************************************/

   RETCODE = SQLFreeConnect (hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Disconnect from data sources in Connection Table              */
  /*****************************************************************/

   SQLFreeEnv(hEnv);           /* free the environment handle  */

   goto exit;

   dberror:
   RETCODE=12;

   exit:

   (void) printf ("**** Exiting  CLIP10.\n\n");

   return(RETCODE);
}
```

References

- “SQLAllocConnect - Allocate Connection Handle” on page 78
- “SQLConnect - Connect to a Data Source” on page 120

SQLError - Retrieve Error Information

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLError() returns the diagnostic information (both errors and warnings) associated with the most recently invoked DB2 CLI function for a particular statement, connection or environment handle.

The information consists of a standardized SQLSTATE and native error code. Refer to “Diagnostics” on page 36 for more information.

Call SQLError() after receiving a return code of SQL_ERROR or SQL_SUCCESS_WITH_INFO from another function call.

Note: Some database servers provide product-specific diagnostic information after returning SQL_NO_DATA_FOUND from the execution of a statement.

Syntax

```
SQLRETURN   SQLError        (SQLHENV          henv,
                             SQLHDBC          hdbc,
                             SQLHSTMT         hstmt,
                             SQLCHAR     FAR  *szSqlState,
                             SQLINTEGER  FAR  *pfNativeError,
                             SQLCHAR     FAR  *szErrorMsg,
                             SQLSMALLINT      cbErrorMsgMax,
                             SQLSMALLINT FAR  *pcbErrorMsg);
```

Function Arguments

Table 43 (Page 1 of 2). SQLError Arguments

Data Type	Argument	Use	Description
SQLHENV	*henv*	input	Environment handle. To obtain diagnostic information associated with an environment, pass a valid environment handle. Set *hdbc* and *hstmt* to SQL_NULL_HDBC and SQL_NULL_HSTMT respectively.
SQLHDBC	*hdbc*	input	Database connection handle. To obtain diagnostic information associated with a connection, pass a valid database connection handle, and set *hstmt* to SQL_NULL_HSTMT. The *henv* argument is ignored.
SQLHSTMT	*hstmt*	input	Statement handle. To obtain diagnostic information associated with a statement, pass a valid statement handle. The *henv* and *hdbc* arguments are ignored.
SQLCHAR *	*szSqlState*	output	SQLSTATE as a string of 5 characters terminated by a null character. The first 2 characters indicate error class; the next 3 indicate subclass. The values correspond directly to SQLSTATE values defined in the X/Open SQL CAE specification and the ODBC specification, augmented with IBM specific and product specific SQLSTATE values.

Table 43 (Page 2 of 2). SQLError Arguments

Data Type	Argument	Use	Description
SQLINTEGER *	*pfNativeError*	output	Native error code. In DB2 CLI, the *pfNativeError* argument contains the SQLCODE value returned by the DBMS. If the error is generated by DB2 CLI and not the DBMS, then this field is set to -99999.

SQLCHAR *	*szErrorMsg*	output	Pointer to buffer to contain the implementation defined message text. If the error is detected by DB2 CLI, then the error message is prefaced by:
			`[IBM][CLI Driver]`
			to indicate that it is DB2 CLI that detected the error and there is no database connection yet.
			If the error is detected while there is a database connection, then the error message returned from the DBMS is prefaced by:
			`[IBM][CLI Driver][`*`DBMS-name`*`]`
			where `DBMS-name` is the name returned by `SQLGetInfo()` with SQL_DBMS_NAME information type.
			For example,
			`DB2`
			`DB2/6000`
			`Vendor`
			`Vendor` indicates a non-IBM DRDA DBMS.
			If the error is generated by the DBMS, the IBM-defined SQLSTATE is appended to the text string.

SQLSMALLINT	*cbErrorMsgMax*	input	The maximum (that is, the allocated) length of the buffer *szErrorMsg*. The recommended length to allocate is SQL_MAX_MESSAGE_LENGTH + 1.
SQLSMALLINT *	*pcbErrorMsg*	output	Pointer to total number of bytes available to return to the *szErrorMsg* buffer. This does not include the null termination character.

Usage

The SQLSTATEs are those defined by the X/OPEN SQL CAE and the X/Open SQL CLI CAE, augmented with IBM specific and product specific SQLSTATE values.

To obtain diagnostic information associated with:

- An environment, pass a valid environment handle. Set *hdbc* and *hstmt* to SQL_NULL_HDBC and SQL_NULL_HSTMT respectively.
- A connection, pass a valid database connection handle, and set *hstmt* to SQL_NULL_HSTMT. The *henv* argument is ignored.
- A statement, pass a valid statement handle. The *henv* and *hdbc* arguments are ignored.

If diagnostic information generated by one DB2 CLI function is not retrieved before a function other than `SQLError()` is called with the same handle, the information for the previous function call is lost. This is true whether or not diagnostic information is generated for the second DB2 CLI function call.

Multiple diagnostic messages might be available after a given DB2 CLI function call. These messages can be retrieved one at a time by repeatedly calling `SQLError()`. For each message retrieved, `SQLError()` returns SQL_SUCCESS and removes it from the list of messages available. When there are no more messages to retrieve, SQL_NO_DATA_FOUND is returned, the SQLSTATE is set to "00000", *pfNativeError* is set to 0, and *pcbErrorMsg* and *szErrorMsg* are undefined.

Diagnostic information stored under a given handle is cleared when a call is made to `SQLError()` with that handle, or when another DB2 CLI function call is made with that handle. However, information associated with a given handle type is not cleared by a call to `SQLError()` with an associated but different handle type: for example, a call to `SQLError()` with a connection handle input does not clear errors associated does any statement handles under that connection.

SQL_SUCCESS is returned even if the buffer for the error message (*szErrorMsg*) is too short since the application is not able to retrieve the same error message by calling `SQLError()` again. The actual length of the message text is returned in the *pcbErrorMsg*.

To avoid truncation of the error message, declare a buffer length of SQL_MAX_MESSAGE_LENGTH + 1. The message text is never longer than this.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

SQL_NO_DATA_FOUND is returned if no diagnostic information is available for the input handle, or if all of the messages are retrieved by calls to `SQLError()`.

Diagnostics

SQLSTATEs are not defined, since `SQLError()` does not generate diagnostic information for itself.

Restrictions

Although ODBC also returns X/Open SQL CAE SQLSTATEs, only DB2 CLI (and the DB2 ODBC driver) returns the additional IBM-defined SQLSTATEs. These SQLSTATES are not defined by X/Open and are not returned by DB2 CLI. For more information on ODBC specific SQLSTATEs refer to *ODBC 2.0 Programmer' s Reference and SDK Guide*.

Because of this, you should only build dependencies on the standard SQLSTATEs. This means any branching logic in the application should only rely on the standard SQLSTATEs. The augmented SQLSTATEs are most useful for debugging purposes.

Note: It might be useful to build dependencies on the class (the first 2 characters) of the SQLSTATEs.

Example

This example shows several utility functions used by most of the other DB2 CLI examples.

```
/* ... */
/*******************************************************************
**  - print_error   - call SQLError(), display SQLSTATE and message
**                  - called by check_error, see below
*******************************************************************/

SQLRETURN
print_error(SQLHENV henv,
            SQLHDBC hdbc,
            SQLHSTMT hstmt,
            SQLRETURN frc,   /* Return code to be included with error msg  */
            SQLINTEGER line, /* Used for output message, indcate where     */
            SQLCHAR * file)  /* the error was reported from  */
{
    SQLCHAR         buffer[SQL_MAX_MESSAGE_LENGTH + 1];
    SQLCHAR         sqlstate[SQL_SQLSTATE_SIZE + 1];
    SQLINTEGER      sqlcode;
    SQLSMALLINT     length;


    printf(">--- ERROR -- RC= %ld Reported from %s, line %ld ------------\n",
           frc, file, line);
    while (SQLError(henv, hdbc, hstmt, sqlstate, &sqlcode, buffer,
                    SQL_MAX_MESSAGE_LENGTH + 1, &length) == SQL_SUCCESS) {
        printf("         SQLSTATE: %s\n", sqlstate);
        printf("Native Error Code: %ld\n", sqlcode);
        printf("%s \n", buffer);
    };
    printf(">--------------------------------------------------\n");
    return (SQL_ERROR);

}

/*******************************************************************
**  - check_error   - call print_error(), checks severity of return code
*******************************************************************/
SQLRETURN
check_error(SQLHENV henv,
            SQLHDBC hdbc,
            SQLHSTMT hstmt,
            SQLRETURN frc,
            SQLINTEGER line,
            SQLCHAR * file)
{
    SQLRETURN       rc;

    print_error(henv, hdbc, hstmt, frc, line, file);

    switch (frc) {
    case SQL_SUCCESS:
        break;
    case SQL_INVALID_HANDLE:
        printf("\n>------ ERROR Invalid Handle --------------------------\n");
    case SQL_ERROR:
        printf("\n>--- FATAL ERROR, Attempting to rollback transaction --\n");
        rc = SQLTransact(henv, hdbc, SQL_ROLLBACK);
        if (rc != SQL_SUCCESS)
            printf(">Rollback Failed, Exiting application\n");
        else
            printf(">Rollback Successful, Exiting application\n");
        exit(0);
        break;
    case SQL_SUCCESS_WITH_INFO:
        printf("\n> ----- Warning Message, application continuing --------\n");
        break;
    case SQL_NO_DATA_FOUND:
        printf("\n> ----- No Data Found, application continuing --------- \n");
        break;
    default:
        printf("\n> ----------- Invalid Return Code --------------------- \n");
        printf("> --------- Attempting to rollback transaction ---------- \n");
        SQLTransact(henv, hdbc, SQL_ROLLBACK);
        exit(0 );
        break;
    }
    return (SQL_SUCCESS);

}
/* ... */

}
/*********************************************************************
* The following macros use check_error
*
*   {check_error(henv, SQL_NULL_HDBC, SQL_NULL_HSTMT, RC, __LINE__, __FILE__);}
*
*  {check_error(SQL_NULL_HENV, hdbc, SQL_NULL_HSTMT, RC, __LINE__, __FILE__);}
*
*  {check_error(SQL_NULL_HENV, SQL_NULL_HDBC, hstmt, RC, __LINE__, __FILE__);}
*
************************************************************************/

/*******************************************************************
**  - check_error   - call print_error(), checks severity of return code
*******************************************************************/
SQLRETURN
check_error(SQLHENV henv,
            SQLHDBC hdbc,
            SQLHSTMT hstmt,
            SQLRETURN frc,
            SQLINTEGER line,
            SQLCHAR * file)
{
    SQLRETURN       rc;

    print_error(henv, hdbc, hstmt, frc, line, file);
    switch (frc) {
    case SQL_SUCCESS:
        break;
    case SQL_INVALID_HANDLE:
        printf("\n>------ ERROR Invalid Handle --------------------------\n");
    case SQL_ERROR:
        printf("\n>--- FATAL ERROR, Attempting to rollback transaction --\n");
        rc = SQLTransact(henv, hdbc, SQL_ROLLBACK);
        if (rc != SQL_SUCCESS)
            printf(">Rollback Failed, Exiting application\n");
        else
            printf(">Rollback Successful, Exiting application\n");
        exit(terminate(henv, frc));
        break;
    case SQL_SUCCESS_WITH_INFO:
        printf("\n> ----- Warning Message, application continuing --------\n");
        break;
    case SQL_NO_DATA_FOUND:
        printf("\n> ----- No Data Found, application continuing --------- \n");
        break;
    default:
        printf("\n> ----------- Invalid Return Code --------------------- \n");
        printf("> --------- Attempting to rollback transaction ---------- \n");
        SQLTransact(henv, hdbc, SQL_ROLLBACK);
        exit(terminate(henv, frc));
        break;
    }
    return (SQL_SUCCESS);

}                               /* end check_error */
/* ... */
```

References

- “SQLGetSQLCA - Get SQLCA Data Structure” on page 229

SQLExecDirect - Execute a Statement Directly

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLExecDirect() directly executes the specified SQL statement. The statement can only be executed once. Also, the connected database server must be able to dynamically prepare statement. (For more information about supported SQL statements refer to Table 1 on page 18.)

Syntax

```
SQLRETURN   SQLExecDirect    (SQLHSTMT         hstmt,
                              SQLCHAR     FAR  *szSqlStr,
                              SQLINTEGER       cbSqlStr);
```

Function Arguments

Table 44. SQLExecDirect Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle. There must not be an open cursor associated with *hstmt*, see “SQLFreeStmt - Free (or Reset) a Statement Handle” on page 182 for more information.
SQLCHAR *	*szSqlStr*	input	SQL statement string. The connected database server must be able to prepare the statement, see Table 1 on page 18 for more information.
SQLINTEGER	*cbSqlStr*	input	Length of contents of *szSqlStr* argument. The length must be set to either the exact length of the statement, or if the statement is null-terminated, set to SQL_NTS.

Usage

If the SQL statement text contains vendor escape clause sequences, DB2 CLI first modifies the SQL statement text to the appropriate DB2-specific format before submitting it for preparation and execution. If the application does not generate SQL statements that contain vendor escape clause sequences (“Using Vendor Escape Clauses” on page 376), then it should set the SQL_NO_SCAN statement option to SQL_NOSCAN_ON at the connection level so that each statement passed to DB2 CLI does not incur the performance impact of scanning for vendor escape clauses.

The SQL statement cannot be a COMMIT or ROLLBACK. Instead, SQLTransact() must be called to issue COMMIT or ROLLBACK. For more information about supported SQL statements refer to Table 1 on page 18.

The SQL statement string can contain parameter markers. A parameter marker is represented by a "?" character, and is used to indicate a position in the statement where an application supplied value is to be substituted when SQLExecDirect() is called. This value can be obtained from an application variable. SQLSetParam() or

SQLBindParameter() is used to bind the application storage area to the parameter marker.

All parameters must be bound before calling SQLExecDirect().

If the SQL statement is a query, SQLExecDirect() generates a cursor name, and open the cursor. If the application has used SQLSetCursorName() to associate a cursor name with the statement handle, DB2 CLI associates the application generated cursor name with the internally generated one.

If a result set is generated, SQLFetch() or SQLExtendedFetch() retrieves the next row (or rows) of data into bound variables. Data can also be retrieved by calling SQLGetData() for any column that was not bound.

If the SQL statement is a positioned DELETE or a positioned UPDATE, the cursor referenced by the statement must be positioned on a row and must be defined on a **separate** statement handle under the same connection handle.

There must not already be an open cursor on the statement handle.

If SQLParamOptions() is called to specify that an array of input parameter values is bound to each parameter marker, then the application needs to call SQLExecDirect() only once to process the entire array of input parameter values.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NEED_DATA
- SQL_NO_DATA_FOUND

SQL_NEED_DATA is returned when the application requests to input data-at-execution parameter values by calling SQLParamData() and SQLPutData().

# SQL_SUCCESS is returned if the SQL statement is a searched UPDATE or
# searched DELETE and no rows satisfy the search condition. SQLRowCount()
# should be used to determine the number of rows in a table that were affected by an
# UPDATE, INSERT, or DELETE statement executed against the table, or a view of
# the table.

Diagnostics

Table 45 (Page 1 of 3). SQLExecDirect SQLSTATEs

SQLSTATE	Description	Explanation
01504	The UPDATE or DELETE statement does not include a WHERE clause.	*szSqlStr* contains an UPDATE or DELETE statement but no WHERE clause. (Function returns SQL_SUCCESS_WITH_INFO or SQL_NO_DATA_FOUND if there are no rows in the table).
07001	Wrong number of parameters.	The number of parameters bound to application variables using SQLBindParameter() is less than the number of parameter markers in the SQL statement contained in the argument *szSqlStr*.

Table 45 (Page 2 of 3). SQLExecDirect SQLSTATEs

SQLSTATE	Description	Explanation
07006	Invalid conversion.	Transfer of data between DB2 CLI and the application variables would result in incompatible data conversion.
21S01	Insert value list does not match column list.	*szSqlStr* contains an INSERT statement and the number of values to be inserted did not match the degree of the derived table.
21S02	Degrees of derived table does not match column list.	*szSqlStr* contains a CREATE VIEW statement and the number of names specified is not the same degree as the derived table defined by the query specification.
22001	String data right truncation.	A character string assigned to a character type column exceeded the maximum length of the column.
22003	Numeric value out of range.	A numeric value assigned to a numeric type column caused truncation of the whole part of the number, either at the time of assignment or in computing an intermediate result. *szSqlStr* contains an SQL statement with an arithmetic expression which caused division by zero.
22005	Error in assignment.	*szSqlStr* contains an SQL statement with a parameter or literal and the value was incompatible with the data type of the associated table column. The length associated with a parameter value (the contents of the *pcbValue* buffer specified on `SQLBindParameter()`) is not valid. The argument *fSQLType* used in `SQLBindParameter()` or `SQLSetParam()`, denoted an SQL graphic data type, but the deferred length argument (*pcbValue*) contains an odd length value. The length value must be even for graphic data types.
22007	Invalid datetime format.	*szSqlStr* contains an SQL statement with an invalid datetime format; that is, an invalid string representation or value was specified, or the value was an invalid date.
22008	Datetime field overflow.	Datetime field overflow occurred; for example, an arithmetic operation on a date or timestamp has a result that is not within the valid range of dates, or a datetime value cannot be assigned to a bound variable because it is too small.
22012	Division by zero is invalid.	*szSqlStr* contains an SQL statement with an arithmetic expression that caused division by zero.
23000	Integrity constraint violation.	The execution of the SQL statement is not permitted because the execution would cause integrity constraint violation in the DBMS.
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
24504	The cursor identified in the UPDATE, DELETE, SET, or GET statement is not positioned on a row.	Results were pending on the *hstmt* from a previous query or a cursor associated with the *hsmt* had not been closed.
34000	Invalid cursor name.	*szSqlStr* contains a positioned DELETE or a positioned UPDATE and the cursor referenced by the statement being executed was not open.
37xxx [a]	Invalid SQL syntax.	*szSqlStr* contains one or more of the following: • a COMMIT • a ROLLBACK • an SQL statement that the connected database server could not prepare • a statement containing a syntax error

Table 45 (Page 3 of 3). SQLExecDirect SQLSTATEs

SQLSTATE	Description	Explanation
40001	Transaction rollback.	The transaction to which this SQL statement belongs is rolled back due to a deadlock or timeout.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
42xxx	Syntax error or access rule violation	**42**5xx indicates the authorization ID does not have permission to execute the SQL statement contained in *szSqlStr*. Other **42**xxx SQLSTATES indicate a variety of syntax or access problems with the statement.
44000	Integrity constraint violation.	*szSqlStr* contains an SQL statement with a parameter or literal. This parameter value is NULL for a column defined as NOT NULL in the associated table column, or a duplicate value is supplied for a column constrained to contain only unique values, or some other integrity constraint is violated.
58004	Unexpected system failure.	Unrecoverable system error.
S0001	Database object already exists.	*szSqlStr* contains a CREATE TABLE or CREATE VIEW statement and the table name or view name specified already exists.
S0002	Database object does not exist.	*szSqlStr* contains an SQL statement that references a table name or view name which does not exist.
S0011	Index already exists.	*szSqlStr* contains a CREATE INDEX statement and the specified index name already exists.
S0012	Index not found.	*szSqlStr* contains a DROP INDEX statement and the specified index name does not exist.
S0021	Column already exists.	*szSqlStr* contains an ALTER TABLE statement and the column specified in the ADD clause is not unique or identifies an existing column in the base table.
S0022	Column not found.	*szSqlStr* contains an SQL statement that references a column name that does not exist.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	*szSqlStr* is a null pointer.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.
S1014	No more handles.	DB2 CLI is not able to allocate a handle due to internal resources.
S1090	Invalid string or buffer length.	The argument *cbSqlStr* is less than 1 but not equal to SQL_NTS.

Note:

a xxx refers to any SQLSTATE with that class code. Example, **37**xxx refers to any SQLSTATE in the **37** class.

Restrictions

None.

Example

Refer to "Example" on page 167.

References

- "SQLBindParameter - Binds A Parameter Marker to a Buffer" on page 91
- "SQLExecute - Execute a Statement" on page 154
- "SQLExtendedFetch - Extended Fetch (Fetch Array of Rows)" on page 157
- "SQLFetch - Fetch Next Row" on page 164
- "SQLExtendedFetch - Extended Fetch (Fetch Array of Rows)" on page 157
- "SQLParamData - Get Next Parameter For Which A Data Value Is Needed" on page 257
- "SQLPutData - Passing Data Value for A Parameter" on page 287
- "SQLSetParam - Binds A Parameter Marker to a Buffer" on page 310

SQLExecute - Execute a Statement

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLExecute() executes a statement, that is successfully prepared using SQLPrepare(), once or multiple times. The statement is executed using the current value of any application variables that are bound to parameter markers by SQLBindParameter() or SQLSetParam() .

Syntax

```
SQLRETURN   SQLExecute      (SQLHSTMT         hstmt);
```

Function Arguments

Table 46. SQLExecute Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle. There must not be an open cursor associated with hstmt, see “SQLFreeStmt - Free (or Reset) a Statement Handle” on page 182 for more information.

Usage

The SQL statement string can contain parameter markers. A parameter marker is represented by a "?" character, and is used to indicate a position in the statement where an application supplied value is to be substituted when SQLExecute() is called. This value can be obtained from an application variable. SQLSetParam() or SQLBindParameter() is used to bind the application storage area to the parameter marker.

You must bind all parameters before calling SQLExecute().

After the application processes the results from the SQLExecute() call, it can execute the statement again with new (or the same) parameter values.

A statement executed by SQLExecDirect() cannot be re-executed by calling SQLExecute(); SQLPrepare() must be called first.

If the prepared SQL statement is a query, SQLExecute() generates a cursor name, and open the cursor. If the application uses SQLSetCursorName() to associate a cursor name with the statement handle, DB2 CLI associates the application generated cursor name with the internally generated one.

To execute a query more than once, the application must close the cursor by calling SQLFreeStmt() with the SQL_CLOSE option. There must not be an open cursor on the statement handle when calling SQLExecute().

If a result set is generated, SQLFetch() or SQLExtendedFetch() retrieves the next row (or rows) of data into bound variables. Data can also be retrieved by calling SQLGetData() for any column that was not bound.

If the SQL statement is a positioned DELETE or a positioned UPDATE, the cursor referenced by the statement must be positioned on a row at the time SQLExecute() is called, and must be defined on a separate statement handle under the same connection handle.

If SQLParamOptions() is called to specify that an array of input parameter values is bound to each parameter marker, then the application needs to call SQLExecDirect() only once to process the entire array of input parameter values. If the executed statement returns multiple result sets (one for each set of input parameters), then SQLMoreResults() should be used to advance to the next result set when processing on the current result set is complete. Refer to "SQLMoreResults - Determine If There Are More Result Sets" on page 246 for more information.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NEED_DATA
- SQL_NO_DATA_FOUND

SQL_NEED_DATA is returned when the application requests to input data-at-execution parameter values by calling SQLParamData() and SQLPutData().

# SQL_SUCCESS is returned if the SQL statement is a searched UPDATE or
# searched DELETE and no rows satisfy the search condition. SQLRowCount()
# should be used to determine the number of rows in a table that were affected by an
# UPDATE, INSERT, or DELETE statement executed against the table, or a view of
# the table.

Diagnostics

The SQLSTATEs for SQLExecute() include all those for SQLExecDirect() (refer to Table 45 on page 150) except for **S1**009, **S1**090 and with the addition of the SQLSTATE in the table below.

Table 47. SQLExecute SQLSTATEs

SQLSTATE	Description	Explanation
40001	Transaction rollback.	The transaction to which this SQL statement belongs is rolled back due to a deadlock or timeout.
S1010	Function sequence error.	The specified *hstmt* is not in prepared state. SQLExecute() is called without first calling SQLPrepare().

Restrictions

None.

Example

Refer to “Example” on page 264.

References

- “SQLExecDirect - Execute a Statement Directly” on page 149
- “SQLExecute - Execute a Statement” on page 154
- “SQLExtendedFetch - Extended Fetch (Fetch Array of Rows)” on page 157
- “SQLPrepare - Prepare a Statement” on page 261
- “SQLFetch - Fetch Next Row” on page 164
- “SQLSetParam - Binds A Parameter Marker to a Buffer” on page 310
- “SQLParamOptions - Specify an Input Array for a Parameter” on page 259
- “SQLBindParameter - Binds A Parameter Marker to a Buffer” on page 91

SQLExtendedFetch - Extended Fetch (Fetch Array of Rows)

Purpose

Specification:	**ODBC** 1.0		

SQLExtendedFetch() extends the function of SQLFetch() by returning a block of data containing multiple rows (called a *rowset*), in the form of an array, for each bound column. The size of the rowset is determined by the SQL_ROWSET_SIZE option on an SQLSetStmtOption() call.

To fetch one row of data at a time, an application should call SQLFetch().

For more description on block or array retrieval, refer to "Retrieving A Result Set Into An Array" on page 357.

Syntax

```
SQLRETURN SQLExtendedFetch  (SQLHSTMT          hstmt,
                             SQLUSMALLINT      fFetchType,
                             SQLINTEGER        irow,
                             SQLUINTEGER  FAR  *pcrow,
                             SQLUSMALLINT FAR  *rgfRowStatus);
```

Function Arguments

Table 48 (Page 1 of 2). SQLExtendedFetch Arguments

Data Type	**Argument**	**Use**	**Description**
SQLHSTMT	hstmt	Input	Statement handle.
SQLUSMALLINT	fFetchType	Input	Direction and type of fetch. DB2 CLI only supports the fetch direction SQL_FETCH_NEXT; that is, forward only cursor direction. The next array (rowset) of data is retrieved.
SQLINTEGER	irow	Input	Reserved for future use.
SQLUINTEGER *	pcrow	Output	Number of the rows actually fetched. If an error occurs during processing, *pcrow* points to the ordinal position of the row (in the rowset) that precedes the row where the error occurred. If an error occurs retrieving the first row *pcrow* points to the value 0.

Table 48 (Page 2 of 2). SQLExtendedFetch Arguments

Data Type	Argument	Use	Description
SQLUSMALLINT *	rgfRowStatus	Output	An array of status values. The number of elements must equal the number of rows in the rowset (as defined by the SQL_ROWSET_SIZE option). A status value for each row fetched is returned: • SQL_ROW_SUCCESS If the number of rows fetched is less than the number of elements in the status array (i.e. less than the rowset size), the remaining status elements are set to SQL_ROW_NOROW. DB2 CLI cannot detect whether a row has been updated or deleted since the start of the fetch. Therefore, the following ODBC-defined status values are not reported: • SQL_ROW_DELETED • SQL_ROW_UPDATED

Usage

`SQLExtendedFetch()` performs an array fetch of a set of rows. An application specifies the size of the array by calling `SQLSetStmtOption()` with the SQL_ROWSET_SIZE option.

Before `SQLExtendedFetch()` is called the first time, the cursor is positioned before the first row. After `SQLExtendedFetch()` is called, the cursor is positioned on the row in the result set corresponding to the last row element in the rowset just retrieved.

For any columns in the result set that are bound using the `SQLBindCol()` function, DB2 CLI converts the data for the bound columns as necessary and stores it in the locations bound to these columns. As mentioned in section “Retrieving A Result Set Into An Array” on page 357, the result set can be bound in a column-wise or row-wise fashion.

- For column-wise binding of application variables:

  To bind a result set in column-wise fashion, an application specifies SQL_BIND_BY_COLUMN for the SQL_BIND_TYPE statement option. (This is the default value.) Then the application calls the `SQLBindCol()` function.

  When the application calls `SQLExtendedFetch()`, data for the first row is stored at the start of the buffer. Each subsequent row of data is stored at an offset of *cbValueMax* bytes (argument on `SQLBindCol()` call) or, if the associated C buffer type is fixed width (such as SQL_C_LONG), at an offset corresponding to that fixed length from the data for the previous row.

  For each bound column, the number of bytes available to return for each element is stored in the *pcbValue* array buffer (deferred output argument on `SQLBindCol()`) buffer bound to the column. The number of bytes available to return for the first row of that column is stored at the start of the buffer, and the number of bytes available to return for each subsequent row is stored at an offset of *sizeof(SQLINTEGER)* bytes from the value for the previous row. If the data in the column is NULL for a particular row, the associated element in the pcbValue array is set to SQL_NULL_DATA.

- For row-wise binding of application variables:

  The application needs to first call SQLSetStmtOption() with the SQL_BIND_TYPE option, with the *vParam* argument set to the size of the structure capable of holding a single row of retrieved data and the associated data lengths for each column data value.

  For each bound column, the first row of data is stored at the address given by the *rgbValue* supplied on the SQLBindCol() call for the column and each subsequent row of data at an offset of *vParam* bytes (used on the SQLSetStmtOption() call) from the data for the previous row.

  For each bound column, the number of bytes available to return for the first row is stored at the address given by the *pcbValue* argument supplied on the SQLBindCol() call, and the number of bytes available to return for each subsequent row at an offset of *vParam* bytes from address containing the value for the previous row.

If SQLExtendedFetch() returns an error that applies to the entire rowset, the SQL_ERROR function return code is reported with the appropriate SQLSTATE. The contents of the rowset buffer are undefined and the cursor position is unchanged.

If an error occurs that applies to a single row:

- The corresponding element in the *rgfRowStatus* array for the row is set to SQL_ROW_ERROR
- An SQLSTATE of **01**S01 is added to the list of errors that can be obtained using *SQLError()*
- Zero or more additional SQLSTATEs, describing the error for the current row, are added to the list of errors that can be obtained using *SQLError()*

An SQL_ROW_ERROR in the *rgfRowStatus* array only indicates that there was an error with the corresponding element; it does not indicate how many SQLSTATEs were generated. Therefore, SQLSTATE **01**S01 is used as a separator between the resulting SQLSTATEs for each row. DB2 CLI continues to fetch the remaining rows in the rowset and returns SQL_SUCCESS_WITH_INFO as the function return code. After SQLExtendedFetch() returns, for each row encountering an error there is an SQLSTATE of **01**S01 and zero or more additional SQLSTATEs indicating the errors for the current row, retrievable via SQLError(). Individual errors that apply to specific rows do not affect the cursor which continues to advance.

The number of elements in the *rgfRowStatus* array output buffer must equal the number of rows in the rowset (as defined by the SQL_ROWSET_SIZE statement option). If the number of rows fetched is less than the number of elements in the status array, the remaining status elements are set to SQL_ROW_NOROW.

An application cannot mix SQLExtendedFetch() with SQLFetch() calls.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

Diagnostics

Table 49 (Page 1 of 2). SQLExtendedFetch SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The data returned for one *or more* columns is truncated. (Function returns SQL_SUCCESS_WITH_INFO.)
01S01	Error in row.	An error occurred while fetching one or more rows. (Function returns SQL_SUCCESS_WITH_INFO.)
07002	Too many columns.	A column number specified in the binding for one or more columns is greater than the number of columns in the result set. The application has used `SQLSetColAttributes()` to inform DB2 CLI of the descriptor information of the result set, but it did not provide this for every column in the result set.
07006	Invalid conversion.	The data value could not be converted in a meaningful manner to the data type specified by *fCType* in `SQLBindCol()`.
22002	Invalid output or indicator buffer specified.	The pointer value specified for the argument *pcbValue* in `SQLBindCol()` is a null pointer and the value of the corresponding column is null. There is no means to report SQL_NULL_DATA.
22003	Numeric value out of range.	Returning the numeric value (as numeric or string) for one or more columns causes the whole part of the number to be truncated either at the time of assignment or in computing an intermediate result. A value from an arithmetic expression is returned which results in division by zero.
22005	Error in assignment.	A returned value is incompatible with the data type of the bound column.
22007	Invalid datetime format.	Conversion from character a string to a datetime format is indicated, but an invalid string representation or value is specified, or the value is an invalid date. The value of a date, time, or timestamp does not conform to the syntax for the specified data type.
22008	Datetime field overflow.	Datetime field overflow occurred; for example, an arithmetic operation on a date or timestamp has a result that is not within the valid range of dates, or a datetime value cannot be assigned to a bound variable because it is too small.
22012	Division by zero is invalid.	A value from an arithmetic expression is returned which results in division by zero.
24000	Invalid cursor state.	The previous SQL statement executed on the statement handle is not a query.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.

Table 49 (Page 2 of 2). SQLExtendedFetch SQLSTATEs

SQLSTATE	Description	Explanation
S1010	Function sequence error.	SQLExtendedFetch() is called for an hstmt after SQLFetch() is called and before SQLFreeStmt() is called with the SQL_CLOSE option. The function is called prior to calling SQLPrepare() or SQLExecDirect() for the *hstmt*. The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.
S1106	Fetch type out of range.	The value specified for the argument *fFetchType* is not recognized.
S1C00	Driver not capable.	DB2 CLI or the data source does not support the conversion specified by the combination of the *fCType* in SQLBindCol() and the SQL data type of the corresponding column. A call to SQLBindCol() is made for a column data type which is not supported by DB2 CLI. The specified fetch type is recognized, but not supported.

Restrictions

None.

Example

```
/* ... */
    "SELECT deptnumb, deptname, id, name FROM staff, org \
                   WHERE dept=deptnumb AND job = 'Mgr'";

    /* Column-Wise */
    SQLINTEGER      deptnumb[ROWSET_SIZE];

    SQLCHAR         deptname[ROWSET_SIZE][15];
    SQLINTEGER      deptname_l[ROWSET_SIZE];

    SQLSMALLINT     id[ROWSET_SIZE];

    SQLCHAR         name[ROWSET_SIZE][10];
    SQLINTEGER      name_l[ROWSET_SIZE];
    /* Row-Wise (Includes buffer for both column data and length) */
    struct {
        SQLINTEGER      deptnumb_l; /* length */
        SQLINTEGER      deptnumb; /* value  */
        SQLINTEGER      deptname_l;
        SQLCHAR         deptname[15];
        SQLINTEGER      id_l;
        SQLSMALLINT     id;
        SQLINTEGER      name_l;
        SQLCHAR         name[10];
    }               R[ROWSET_SIZE];

    SQLUSMALLINT    Row_Stat[ROWSET_SIZE];
    SQLUINTEGER     pcrow;
    int             i;
/* ... */
    /*********************************************/
    /* Column-Wise Binding               */
    /*********************************************/
    rc = SQLAllocStmt(hdbc, &hstmt);

    rc = SQLSetStmtOption(hstmt, SQL_ROWSET_SIZE, ROWSET_SIZE);

    rc = SQLExecDirect(hstmt, stmt, SQL_NTS);

    rc = SQLBindCol(hstmt, 1, SQL_C_LONG, (SQLPOINTER) deptnumb, 0, NULL);

    rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) deptname, 15, deptname_l);

    rc = SQLBindCol(hstmt, 3, SQL_C_SSHORT, (SQLPOINTER) id, 0, NULL);

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) name, 10, name_l);

    /* Fetch ROWSET_SIZE rows ast a time, and display */
    printf("\nDEPTNUMB DEPTNAME         ID       NAME\n");
    printf("-------- -------------- -------- ---------\n");
    while ((rc = SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 0, &pcrow, Row_Stat))
           == SQL_SUCCESS) {
        for (i = 0; i < pcrow; i++) {
            printf("%8ld %-14s %8ld %-9s\n", deptnumb[i], deptname[i], id[i], name[i]);
        }
        if (pcrow < ROWSET_SIZE)
            break;
    }                           /* endwhile */

    if (rc != SQL_NO_DATA_FOUND && rc != SQL_SUCCESS)
        check_error(henv, hdbc, hstmt, rc, __LINE__, __FILE__);

    rc = SQLFreeStmt(hstmt, SQL_DROP);
    /***********************************************/
    /* Row-Wise Binding              */
    /***********************************************/
    rc = SQLAllocStmt(hdbc, &hstmt);
    if (rc != SQL_SUCCESS)
        check_error(henv, hdbc, SQL_NULL_HSTMT, rc, __LINE__, __FILE__);

    /* Set maximum number of rows to receive with each extended fetch */
    rc = SQLSetStmtOption(hstmt, SQL_ROWSET_SIZE, ROWSET_SIZE);
    if (rc != SQL_SUCCESS)
        check_error(henv, hdbc, hstmt, rc, __LINE__, __FILE__);

    /*
     * Set vparam to size of one row, used as offset for each bindcol
     * rgbValue
     */
    /* ie. &(R[0].deptnumb) + vparam = &(R[1].deptnum) */
    rc = SQLSetStmtOption(hstmt, SQL_BIND_TYPE, sizeof(R) / ROWSET_SIZE);

    rc = SQLExecDirect(hstmt, stmt, SQL_NTS);

    rc = SQLBindCol(hstmt, 1, SQL_C_LONG, (SQLPOINTER) & R[0].deptnumb, 0,
                    &R[0].deptnumb_l);

    rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) R[0].deptname, 15,
                    &R[0].deptname_l);

    rc = SQLBindCol(hstmt, 3, SQL_C_SSHORT, (SQLPOINTER) & R[0].id, 0,
                    &R[0].id_l);

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) R[0].name, 10, &R[0].name_l);

    /* Fetch ROWSET_SIZE rows at a time, and display */
    printf("\nDEPTNUMB DEPTNAME         ID       NAME\n");
    printf("-------- -------------- -------- ---------\n");
    while ((rc = SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 0, &pcrow, Row_Stat))
           == SQL_SUCCESS) {
        for (i = 0; i < pcrow; i++) {
            printf("%8ld %-14s %8ld %-9s\n", R[i].deptnumb, R[i].deptname,
                   R[i].id, R[i].name);
        }
        if (pcrow < ROWSET_SIZE)
            break;
    }                           /* endwhile */

    if (rc != SQL_NO_DATA_FOUND && rc != SQL_SUCCESS)
        check_error(henv, hdbc, hstmt, rc, __LINE__, __FILE__);
    /* Free handles, commit, exit */
/* ... */
```

References

- “SQLExecute - Execute a Statement” on page 154
- “SQLExecDirect - Execute a Statement Directly” on page 149
- “SQLFetch - Fetch Next Row” on page 164

SQLFetch - Fetch Next Row

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLFetch() advances the cursor to the next row of the result set, and retrieves any bound columns. When SQLFetch() is called, the appropriate data transfer is performed, along with any data conversion if conversion was indicated when the column was bound. The columns can also be received individually after the fetch, by calling SQLGetData().

SQLFetch() can only be called after a result set is generated (using the same statement handle) by either executing a query, calling SQLGetTypeInfo() or calling a catalog function.

To retrieve multiple rows at a time, use SQLExtendedFetch().

Syntax

```
SQLRETURN   SQLFetch        (SQLHSTMT         hstmt);
```

Function Arguments

Table 50. SQLFetch Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle

Usage

SQLFetch() can only be called after a result set is generated on the same statement handle. Before SQLFetch() is called the first time, the cursor is positioned before the start of the result set.

The number of application variables bound with SQLBindCol() must not exceed the number of columns in the result set or SQLFetch() fails.

If SQLBindCol() has not been called to bind any columns, then SQLFetch() does not return data to the application, but just advances the cursor. In this case SQLGetData() could be called to obtain all of the columns individually. Data in unbound columns is discarded when SQLFetch() advances the cursor to the next row. For fixed length data types, or small variable length data types, binding columns provides better performance than using SQLGetData().

Columns can be bound to application storage. SQLBindCol() is used to bind application storage to the column. Data is transferred from the server to the application at fetch time. Length of the available data to return is also set.

If any bound storage buffers are not large enough to hold the data returned by SQLFetch(), the data is truncated. If character data is truncated, SQL_SUCCESS_WITH_INFO is returned, and an SQLSTATE is generated indicating truncation. The SQLBindCol() deferred output argument *pcbValue* contains the actual length of the column data retrieved from the server. The application should compare the actual output length to the input buffer length (*pcbValue* and *cbValueMax* arguments from `SQLBindCol()`) to determine which character columns are truncated.

Truncation of numeric data types is reported as a warning if the truncation involves digits to the right of the decimal point. If truncation occurs to the left of the decimal point, an error is returned (refer to the diagnostics section).

Truncation of graphic data types is treated the same as character data types, except that the *rgbValue* buffer is filled to the nearest multiple of two bytes that is still less than or equal to the *cbValueMax* specified in `SQLBindCol()`. Graphic (DBCS) data transferred between DB2 CLI and the application is not null-terminated if the C buffer type is SQL_C_CHAR. If the buffer type is SQL_C_DBCHAR, then null-termination of graphic data does occur.

Truncation is also affected by the SQL_MAX_LENGTH statement option. The application can specify that Call Level Interface should not report truncation by calling `SQLSetStmtOption()` with SQL_MAX_LENGTH and a value for the maximum length to return for any one column, and by allocating an *rgbValue* buffer of the same size (plus the null-terminator). If the column data is larger than the set maximum length, SQL_SUCCESS is returned and the maximum length, not the actual length is returned in *pcbValue*.

When all the rows are retrieved from the result set, or the remaining rows are not needed, `SQLFreeStmt()` should be called to close the cursor and discard the remaining data and associated resources.

To retrieve multiple rows at a time, use `SQLExtendedFetch()`. An application cannot mix `SQLFetch()` with `SQLExtendedFetch()` calls on the same statement handle.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

SQL_NO_DATA_FOUND is returned if there are no rows in the result set, or previous `SQLFetch()` calls have fetched all the rows from the result set.

If all the rows were fetched, the cursor is positioned after the end of the result set.

Diagnostics

Table 51 (Page 1 of 3). SQLFetch SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The data returned for one or more columns is truncated. String values or numeric values are right truncated. (SQL_SUCCESS_WITH_INFO is returned if no error occurred.)

Table 51 (Page 2 of 3). SQLFetch SQLSTATEs

SQLSTATE	Description	Explanation
07002	Too many columns.	A column number specified in the binding for one or more columns is greater than the number of columns in the result set. The application used `SQLSetColAttributes()` to inform DB2 CLI of the descriptor information of the result set, but it did not provide this for every column in the result set.
07006	Invalid conversion.	The data value cannot be converted in a meaningful manner to the data type specified by *fCType* in `SQLBindCol()`
22002	Invalid output or indicator buffer specified.	The pointer value specified for the argument *pcbValue* in `SQLBindCol()` is a null pointer and the value of the corresponding column is null. There is no means to report SQL_NULL_DATA.
22003	Numeric value out of range.	Returning the numeric value (as numeric or string) for one or more columns causes the whole part of the number to be truncated either at the time of assignment or in computing an intermediate result. A value from an arithmetic expression is returned which results in division by zero. **Note:** The associated cursor is undefined if this error is detected by DB2 for OS/390. If the error is detected by DB2 for common server or by other IBM RDBMSs, the cursor remains open and continues to advance on subsequent fetch calls.
22005	Error in assignment.	A returned value is incompatible with the data type of binding.
22007	Invalid datetime format.	Conversion from character a string to a datetime format is indicated, but an invalid string representation or value is specified, or the value is an invalid date. The value of a date, time, or timestamp does not conform to the syntax for the specified data type.
22008	Datetime field overflow.	Datetime field overflow occurred; for example, an arithmetic operation on a date or timestamp has a result that is not within the valid range of dates, or a datetime value cannot be assigned to a bound variable because it is too small.
22012	Division by zero is invalid.	A value from an arithmetic expression is returned which results in division by zero.
24000	Invalid cursor state.	The previous SQL statement executed on the statement handle is not a query.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1002	Invalid column number.	The specified column is less than 0 or greater than the number of result columns. The specified column is 0, but DB2 CLI does not support ODBC bookmarks (*icol* = 0). `SQLExtendedFetch()` is called for this result set.

Table 51 (Page 3 of 3). SQLFetch SQLSTATEs

SQLSTATE	Description	Explanation
S1010	Function sequence error.	SQLFetch() is called for an hstmt after SQLExtendedFetch() is called and before SQLFreeStmt() had been called with the SQL_CLOSE option. The function is called prior to calling SQLPrepare() or SQLExecDirect() for the *hstmt*. The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.
S1C00	Driver not capable.	DB2 CLI or the data source does not support the conversion specified by the combination of the *fCType* in SQLBindCol() and the SQL data type of the corresponding column. A call to SQLBindCol() was made for a column data type which is not supported by DB2 CLI.

Restrictions

None.

Example

```
/* ... */
/*******************************************************************
** main
*******************************************************************/
int
main( int argc, char * argv[] )
{
    SQLHENV         henv;
    SQLHDBC         hdbc;
    SQLHSTMT        hstmt;
    SQLRETURN       rc;
    SQLCHAR         sqlstmt[] = "SELECT deptname, location from org where
                                        division = 'Eastern'";
    struct { SQLINTEGER ind;
             SQLCHAR  s[15];
           } deptname, location;

  /* macro to initalize server, uid and pwd */
    INIT_UID_PWD;

    rc = SQLAllocEnv(&henv);    /* allocate an environment handle    */
    if (rc != SQL_SUCCESS)
        return (terminate(henv, rc));

    rc = DBconnect(henv, &hdbc);/* allocate a connect handle, and connect */
    if (rc != SQL_SUCCESS)
        return (terminate(henv, rc));
    rc = SQLAllocStmt(hdbc, &hstmt);

    rc = SQLExecDirect(hstmt, sqlstmt, SQL_NTS);

    rc = SQLBindCol(hstmt, 1, SQL_C_CHAR, (SQLPOINTER) deptname.s, 15,
                    &deptname.ind);

    rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) location.s, 15,
                    &location.ind);

    printf("Departments in Eastern division:\n");
    printf("DEPTNAME       Location\n");
    printf("-------------- -------------\n");

    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
        printf("%-14.14s %-14.14s \n", deptname.s, location.s);
    }
    if (rc != SQL_NO_DATA_FOUND)
        check_error(henv, hdbc, hstmt, rc, __LINE__, __FILE__);

    rc = SQLFreeStmt(hstmt, SQL_DROP);

    rc = SQLTransact(henv, hdbc, SQL_COMMIT);

    printf("Disconnecting .....\n");
    rc = SQLDisconnect(hdbc);

    rc = SQLFreeConnect(hdbc);

    rc = SQLFreeEnv(henv);
    if (rc != SQL_SUCCESS)
        return (terminate(henv, rc));

}                               /* end main */
/* ... */
```

References

- “SQLExtendedFetch - Extended Fetch (Fetch Array of Rows)” on page 157
- “SQLExecute - Execute a Statement” on page 154
- “SQLExecDirect - Execute a Statement Directly” on page 149
- “SQLGetData - Get Data From a Column” on page 193

SQLForeignKeys - Get the List of Foreign Key Columns

Purpose

Specification:	**ODBC** 1.0		

SQLForeignKeys() returns information about foreign keys for the specified table. The information is returned in an SQL result set which can be processed using the same functions that are used to retrieve a result generated by a query.

Syntax

```
SQLRETURN   SQLForeignKeys  (SQLHSTMT          hstmt,
                             SQLCHAR      FAR  *szPkCatalogName,
                             SQLSMALLINT        cbPkCatalogName,
                             SQLCHAR      FAR  *szPkSchemaName,
                             SQLSMALLINT        cbPkSchemaName,
                             SQLCHAR      FAR  *szPkTableName,
                             SQLSMALLINT        cbPkTableName,
                             SQLCHAR      FAR  *szFkCatalogName,
                             SQLSMALLINT        cbFkCatalogName,
                             SQLCHAR      FAR  *szFkSchemaName,
                             SQLSMALLINT        cbFkSchemaName,
                             SQLCHAR      FAR  *szFkTableName,
                             SQLSMALLINT        cbFkTableName);
```

Function Arguments

Table 52. SQLForeignKeys Arguments

Data Type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle.
SQLCHAR *	szPkCatalogName	input	Catalog qualifier of the primary key table. This must be a NULL pointer or a zero length string.
SQLSMALLINT	cbPkCatalogName	input	Length of *szPkCatalogName*. This must be set to 0.
SQLCHAR *	szPkSchemaName	input	Schema qualifier of the primary key table.
SQLSMALLINT	cbPkSchemaName	input	Length of *szPkSchemaName*.
SQLCHAR *	szPkTableName	input	Name of the table name containing the primary key.
SQLSMALLINT	cbPkTableName	input	Length of *szPkTableName*.
SQLCHAR *	szFkCatalogName	input	Catalog qualifier of the table containing the foreign key. This must be a NULL pointer or a zero length string.
SQLSMALLINT	cbFkCatalogName	input	Length of *szFkCatalogName*. This must be set to 0.
SQLCHAR *	szFkSchemaName	input	Schema qualifier of the table containing the foreign key.
SQLSMALLINT	cbFkSchemaName	input	Length of *szFkSchemaName*.
SQLCHAR *	szFkTableName	input	Name of the table containing the foreign key.
SQLSMALLINT	cbFkTableName	input	Length of *szFkTableName*.

Usage

If *szPkTableName* contains a table name, and *szFkTableName* is an empty string, `SQLForeignKeys()` returns a result set containing the primary key of the specified table and all of the foreign keys (in other tables) that refer to it.

If *szFkTableName* contains a table name, and *szPkTableName* is an empty string, `SQLForeignKeys()` returns a result set containing all of the foreign keys in the specified table and the primary keys (in other tables) to which they refer.

If both *szPkTableName* and *szFkTableName* contain table names, `SQLForeignKeys()` returns the foreign keys in the table specified in *szFkTableName* that refer to the primary key of the table specified in *szPkTableName*. This should be one key at the most.

If the schema qualifier argument associated with a table name is not specified, then the schema name defaults to the one currently in effect for the current connection.

Table 53 lists the columns of the result set generated by the `SQLForeignKeys()` call. If the foreign keys associated with a primary key are requested, the result set is ordered by FKTABLE_CAT, FKTABLE_SCHEM, FKTABLE_NAME, and ORDINAL_POSITION. If the primary keys associated with a foreign key are requested, the result set is ordered by PKTABLE_CAT, PKTABLE_SCHEM, PKTABLE_NAME, and ORDINAL_POSITION.

The VARCHAR columns of the catalog functions result set are declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call `SQLGetInfo()` with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the associated TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change.

Table 53 (Page 1 of 2). Columns Returned By SQLForeignKeys

Column Number/Name	**Data Type**	**Description**
1 PKTABLE_CAT	VARCHAR(128)	This is always NULL.
2 PKTABLE_SCHEM	VARCHAR(128)	The name of the schema containing PKTABLE_NAME.
3 PKTABLE_NAME	VARCHAR(128) NOT NULL	Name of the table containing the primary key.
4 PKCOLUMN_NAME	VARCHAR(128) NOT NULL	Primary key column name.
5 FKTABLE_CAT	VARCHAR(128)	This is always NULL.
6 FKTABLE_SCHEM	VARCHAR(128)	The name of the schema containing FKTABLE_NAME.
7 FKTABLE_NAME	VARCHAR(128) NOT NULL	The name of the table containing the foreign key.

Table 53 (Page 2 of 2). Columns Returned By SQLForeignKeys

Column Number/Name	Data Type	Description
8 FKCOLUMN_NAME	VARCHAR(128) NOT NULL	Foreign key column name.
9 ORDINAL_POSITION	SMALLINT NOT NULL	The ordinal position of the column in the key, starting at 1.
10 UPDATE_RULE	SMALLINT	Action to be applied to the foreign key when the SQL operation is UPDATE: • SQL_RESTRICT • SQL_NO_ACTION The update rule for IBM DB2 DBMSs is always either RESTRICT or SQL_NO_ACTION. However, ODBC applications might encounter the following UPDATE_RULE values when connected to non-IBM RDBMSs: • SQL_CASCADE • SQL_SET_NULL
11 DELETE_RULE	SMALLINT	Action to be applied to the foreign key when the SQL operation is DELETE: • SQL_CASCADE • SQL_NO_ACTION • SQL_RESTRICT • SQL_SET_DEFAULT • SQL_SET_NULL
12 FK_NAME	VARCHAR(128)	Foreign key identifier. NULL if not applicable to the data source.
13 PK_NAME	VARCHAR(128)	Primary key identifier. NULL if not applicable to the data source.

Note: The column names used by DB2 CLI follow the X/Open CLI CAE specification style. The column types, contents and order are identical to those defined for the `SQLForeignKeys()` result set in ODBC.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 54 (Page 1 of 2). SQLForeignKeys SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	The arguments *szPkTableName* and *szFkTableName* are both NULL pointers.

Table 54 (Page 2 of 2). SQLForeignKeys SQLSTATEs

SQLSTATE	Description	Explanation
S1010	Function sequence error.	The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1014	No more handles.	DB2 CLI is not able to allocate a handle due to internal resources.
S1090	Invalid string or buffer length.	The value of one of the name length arguments was less than 0, but not equal SQL_NTS. The length of the table or owner name is greater than the maximum length supported by the server. Refer to "SQLGetInfo - Get General Information" on page 213.
S1C00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for table name.

Restrictions

None.

Example

```
 /******************************************************************/
 /*  DB2 for OS/390 Example:                                       */
 /*       Invokes SQLForeignKeys against PARENT Table. Find all    */
 /*       tables that contain foreign keys on PARENT.              */
 /******************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include "cli.h"
#include "sqlcli1.h"
#include "sqlcli1.h"

int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLHSTMT        hStmt   = SQL_NULL_HSTMT;
   SQLRETURN       rc      = SQL_SUCCESS;
   SQLINTEGER      RETCODE = 0;
   char            pTable [200];
   char            *pDSN = "STLEC1";
   SQLSMALLINT     update_rule;
   SQLSMALLINT     delete_rule;
   SQLINTEGER      update_rule_ind;
   SQLINTEGER      delete_rule_ind;
   char            update [25];
   char            delet  [25];

   typedef struct varchar    // define VARCHAR type
   {
     SQLSMALLINT  length;
     SQLCHAR      name [128];
     SQLINTEGER   ind;
   } VARCHAR;
   VARCHAR pktable_schem;
   VARCHAR pktable_name;
   VARCHAR pkcolumn_name;
   VARCHAR fktable_schem;
   VARCHAR fktable_name;
   VARCHAR fkcolumn_name;

   (void) printf ("**** Entering CLIP02.\n\n");

  /*****************************************************************/
  /* Allocate Environment Handle                                   */
  /*****************************************************************/

   RETCODE = SQLAllocEnv(&hEnv);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Allocate Connection Handle to DSN                             */
  /*****************************************************************/

   RETCODE = SQLAllocConnect(hEnv,
                             &hDbc);

   if( RETCODE != SQL_SUCCESS )      // Could not get a Connect Handle
     goto dberror;

  /*****************************************************************/
  /* CONNECT TO data source (STLEC1)                               */
  /*****************************************************************/

   RETCODE = SQLConnect(hDbc,        // Connect handle
                        (SQLCHAR *) pDSN, // DSN
                        SQL_NTS,     // DSN is nul-terminated
                        NULL,        // Null UID
                        0   ,
                        NULL,        // Null Auth string
                        0);

   if( RETCODE != SQL_SUCCESS )      // Connect failed
     goto dberror;

  /*****************************************************************/
  /* Allocate Statement Handle                                     */
  /*****************************************************************/

  rc = SQLAllocStmt (hDbc,
                     &hStmt);

  if (rc != SQL_SUCCESS)
    goto exit;
```

```
/*****************************************************************/
/* Invoke SQLForeignKeys against PARENT Table, specifying NULL   */
/* for table with foreign key.                                   */
/*****************************************************************/

rc = SQLForeignKeys (hStmt,
                     NULL,
                     0,
                     (SQLCHAR *) "ADMF001",
                     SQL_NTS,
                     (SQLCHAR *) "PARENT",
                     SQL_NTS,
                     NULL,
                     0,
                     NULL,
                     SQL_NTS,
                     NULL,
                     SQL_NTS);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** SQLForeignKeys Failed.\n");
  goto dberror;
}


/*****************************************************************/
/* Bind following columns of answer set:                         */
/*                                                               */
/*   2) pktable_schem                                            */
/*   3) pktable_name                                             */
/*   4) pkcolumn_name                                            */
/*   6) fktable_schem                                            */
/*   7) fktable_name                                             */
/*   8) fkcolumn_name                                            */
/*  10) update_rule                                              */
/*  11) delete_rule                                              */
/*                                                               */
/*****************************************************************/

rc = SQLBindCol (hStmt,           // bind pktable_schem
                 2,
                 SQL_C_CHAR,
                 (SQLPOINTER) pktable_schem.name,
                 128,
                 &pktable_schem.ind);

rc = SQLBindCol (hStmt,           // bind pktable_name
                 3,
                 SQL_C_CHAR,
                 (SQLPOINTER) pktable_name.name,
                 128,
                 &pktable_name.ind);

rc = SQLBindCol (hStmt,           // bind pkcolumn_name
                 4,
                 SQL_C_CHAR,
                 (SQLPOINTER) pkcolumn_name.name,
                 128,
                 &pkcolumn_name.ind);
```

```
rc = SQLBindCol (hStmt,           // bind fktable_schem
                 6,
                 SQL_C_CHAR,
                 (SQLPOINTER) fktable_schem.name,
                 128,
                 &fktable_schem.ind);

rc = SQLBindCol (hStmt,           // bind fktable_name
                 7,
                 SQL_C_CHAR,
                 (SQLPOINTER) fktable_name.name,
                 128,
                 &fktable_name.ind);

rc = SQLBindCol (hStmt,           // bind fkcolumn_name
                 8,
                 SQL_C_CHAR,
                 (SQLPOINTER) fkcolumn_name.name,
                 128,
                 &fkcolumn_name.ind);

rc = SQLBindCol (hStmt,           // bind update_rule
                 10,
                 SQL_C_SHORT,
                 (SQLPOINTER) &update_rule;
                 0,
                 &update_rule_ind);

rc = SQLBindCol (hStmt,           // bind delete_rule
                 11,
                 SQL_C_SHORT,
                 (SQLPOINTER) &delete_rule,
                 0,
                 &delete_rule_ind);

/*****************************************************************/
/* Retrieve all tables with foreign keys defined on PARENT      */
/*****************************************************************/

while ((rc = SQLFetch (hStmt)) == SQL_SUCCESS)
{
  (void) printf ("**** Primary Table Schema is %s. Primary Table Name is %s.\n",
                 pktable_schem.name, pktable_name.name);
  (void) printf ("**** Primary Table Key Column is %s.\n",
                 pkcolumn_name.name);
  (void) printf ("**** Foreign Table Schema is %s. Foreign Table Name is %s.\n",
                 fktable_schem.name, fktable_name.name);
  (void) printf ("**** Foreign Table Key Column is %s.\n",
                 fkcolumn_name.name);

  if (update_rule == SQL_RESTRICT)     // isolate update rule
    strcpy (update, "RESTRICT");
  else
  if (update_rule == SQL_CASCADE)
    strcpy (update, "CASCADE");
  else
    strcpy (update, "SET NULL");
    if (delete_rule == SQL_RESTRICT)     // isolate delete rule
      strcpy (delet, "RESTRICT");
    else
    if (delete_rule == SQL_CASCADE)
      strcpy (delet, "CASCADE");
    else
    if (delete_rule == SQL_NO_ACTION)
      strcpy (delet, "NO ACTION");
    else
      strcpy (delet, "SET NULL");

    (void) printf ("**** Update Rule is %s. Delete Rule is %s.\n",
                   update, delet);
  }

  /*****************************************************************/
  /* Deallocate Statement Handle                                   */
  /*****************************************************************/

  rc = SQLFreeStmt (hStmt,
                    SQL_DROP);

  /*****************************************************************/
  /* DISCONNECT from data source                                   */
  /*****************************************************************/

   RETCODE = SQLDisconnect(hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Deallocate Connection Handle                                  */
  /*****************************************************************/

   RETCODE = SQLFreeConnect (hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Free Environment Handle                                       */
  /*****************************************************************/

   RETCODE = SQLFreeEnv (hEnv);

   if (RETCODE == SQL_SUCCESS)
     goto exit;

   dberror:
   RETCODE=12;

   exit:

   (void) printf ("**** Exiting  CLIP02.\n\n");

   return RETCODE;
}
```

References

- “SQLPrimaryKeys - Get Primary Key Columns of A Table” on page 269
- “SQLStatistics - Get Index and Statistics Information For A Base Table” on page 325

SQLFreeConnect - Free Connection Handle

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLFreeConnect() invalidates and frees the connection handle. All DB2 CLI resources associated with the connection handle are freed.

SQLDisconnect() must be called before calling this function.

Syntax

```
SQLRETURN   SQLFreeConnect   (SQLHDBC          hdbc);
```

Function Arguments

Table 55. SQLFreeConnect Arguments

Data Type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Connection handle

Usage

If this function is called when a connection still exists, SQL_ERROR is returned, and the connection handle remains valid.

To continue termination, call SQLFreeEnv(), or, if a new connection handle is required, call SQLAllocConnect().

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE
- # SQL_SUCCESS_WITH_INFO

Diagnostics

Table 56. SQLFreeConnect SQLSTATEs

SQLSTATE	Description	Explanation
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	The function is called prior to SQLDisconnect() for the *hdbc*.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.

Restrictions

None.

Example

Refer to “Example” on page 79.

References

- “SQLDisconnect - Disconnect from a Data Source” on page 135
- “SQLFreeEnv - Free Environment Handle” on page 180

SQLFreeEnv - Free Environment Handle

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLFreeEnv() invalidates and frees the environment handle. All DB2 CLI resources associated with the environment handle are freed.

SQLFreeConnect() must be called before calling this function.

This function is the last DB2 CLI step an application needs to do before terminating.

Syntax

```
SQLRETURN   SQLFreeEnv       (SQLHENV          henv);
```

Function Arguments

Table 57. SQLFreeEnv Arguments

Data Type	Argument	Use	Description
SQLHENV	*henv*	input	Environment handle

Usage

If this function is called when there is still a valid connection handle, SQL_ERROR is returned, and the environment handle remains valid.

# The number of SQLFreeEnv() calls must equal the number of SQLAllocEnv() calls
# before the environment information is reset.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 58. SQLFreeEnv SQLSTATEs

SQLSTATE	Description	Explanation
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	There is an *hdbc* which is in allocated or connected state. Call SQLDisconnect() and SQLFreeConnect() for the *hdbc* before calling SQLFreeEnv().
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.

Restrictions

None.

Example

Refer to “Example” on page 79.

References

- “SQLFreeConnect - Free Connection Handle” on page 178

SQLFreeStmt - Free (or Reset) a Statement Handle

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLFreeStmt() ends processing on the statement referenced by the statement handle. Use this function to:

- Close a cursor
- Drop the statement handle and free the DB2 CLI resources associated with the statement handle.

SQLFreeStmt() is called after executing an SQL statement and processing the results.

Syntax

```
SQLRETURN   SQLFreeStmt      (SQLHSTMT         hstmt,
                              SQLUSMALLINT     fOption);
```

Function Arguments

Table 59. SQLFreeStmt Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle
SQLUSMALLINT	*fOption*	input	Option which specified the manner of freeing the statement handle. The option must have one of the following values: • SQL_CLOSE • SQL_DROP • SQL_UNBIND • SQL_RESET_PARAMS

Usage

SQLFreeStmt() can be called with the following options:

SQL_CLOSE The cursor (if any) associated with the statement handle (*hstmt*) is closed and all pending results are discarded. The application can reopen the cursor by calling SQLExecute() or SQLExecDirect() with the same or different values in the application variables (if any) that are bound to *hstmt*. The cursor name is retained until the statement handle is dropped or the next successful SQLSetCursorName() call. If a cursor is not associated with the statement handle, this option has no effect (no warning or error is generated).

SQL_DROP DB2 CLI resources associated with the input statement handle are freed, and the handle is invalidated. The open cursor, if any, is closed and all pending results are discarded.

SQL_UNBIND All the columns bound by previous SQLBindCol() calls on this statement handle are released (the association between application variables or file references and result set columns is broken).

SQL_RESET_PARAMS
All the parameters set by previous SQLBindParameter() calls on this statement handle are released (the association between application variables or file references and parameter markers in the SQL statement for the statement handle is broken).

You can reuse a statement handle to execute a different statement. If the handle is:

- Associated with a query, catalog function, or SQLGetTypeInfo(), you must close the cursor.
- Bound with a different number or type of parameters, the parameters must be reset.
- Bound with a different number or type of column bindings, the columns must be unbound.

Alternatively, you can drop the statement handle and allocate a new one.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

SQL_SUCCESS_WITH_INFO is not returned if *fOption* is set to SQL_DROP, since there would be no statement handle to use when SQLError() is called.

Diagnostics

Table 60. SQLFreeStmt SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
S1092	Option type out of range.	The value specified for the argument *fOption* is not SQL_CLOSE, SQL_DROP, SQL_UNBIND, or SQL_RESET_PARAMS.
S1506	Error closing a file.	Error encountered while trying to close a temporary file.

Restrictions

None.

Example

Refer to “Example” on page 167.

References

- “SQLAllocStmt - Allocate a Statement Handle” on page 84
- “SQLBindParameter - Binds A Parameter Marker to a Buffer” on page 91
- “SQLExtendedFetch - Extended Fetch (Fetch Array of Rows)” on page 157
- “SQLFetch - Fetch Next Row” on page 164
- “SQLSetParam - Binds A Parameter Marker to a Buffer” on page 310

SQLGetConnectOption - Returns Current Setting of A Connect Option

Purpose

Specification:	**ODBC** 1.0	**X/OPEN CLI**	

SQLGetConnectOption() returns the current settings for the specified connection option.

These options are set using the SQLSetConnectOption() function.

Syntax

```
SQLRETURN   SQLGetConnectOption (
                                SQLHDBC          hdbc,
                                SQLUSMALLINT     fOption,
                                SQLPOINTER       pvParam);
```

Function Arguments

Table 61. SQLGetConnectOption Arguments

Data Type	Argument	Use	Description
HDBC	hdbc	input	Connection handle.
SQLUSMALLINT	fOption	input	Option to set. Refer to Table 113 on page 299 for the complete list of connection options and their descriptions.
SQLPOINTER	pvParam	input/output	Value associated with *fOption*. Depending on the value of *fOption*, this can be a 32-bit integer value, or a pointer to a null terminated character string. The maximum length of any character string returned is SQL_MAX_OPTION_STRING_LENGTH bytes (excluding the null-terminator).

Usage

If SQLGetConnectOption() is called, and the specified *fOption* has not been set via SQLSetConnectOption and does not have a default, then SQLGetConnectOption() returns SQL_NO_DATA_FOUND.

Although SQLSetConnectOption() can be used to set statement options, SQLGetConnectOption() cannot be used to retrieve statement options, use SQLGetStmtOption() instead.

For a list of valid connect options, refer to Table 113 on page 299, in the function description for SQLSetConnectOption().

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 62. SQLGetConnectOption SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The function is called after the communication link source to which DB2 CLI is connected, failed during the processing of a previous request.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	The *pvParam* argument is NULL.
S1092	Option type out of range.	An invalid *fOption* value is specified.
S1C00	Driver not capable.	The *fOption* is recognized, but is not supported.

Restrictions

None.

Example

```
/* ... */
    rc = SQLGetConnectOption(hdbc, SQL_AUTOCOMMIT, &autocommit);
    printf("Autocommit is: ");
    if (autocommit == SQL_AUTOCOMMIT_ON)
       printf("ON\n");
    else
       printf("OFF\n");
/* ... */
```

References

- “SQLSetConnectOption - Set Connection Option” on page 298
- “SQLSetStmtOption - Set Statement Option” on page 315
- “SQLGetStmtOption - Returns Current Setting of A Statement Option” on page 236

SQLGetCursorName - Get Cursor Name

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLGetCursorName() returns the cursor name associated with the input statement handle. If a cursor name is explicitly set by calling SQLSetCursorName(), this name is returned; otherwise, an implicitly generated name is returned.

Syntax

```
SQLRETURN   SQLGetCursorName (SQLHSTMT          hstmt,
                              SQLCHAR     FAR   *szCursor,
                              SQLSMALLINT       cbCursorMax,
                              SQLSMALLINT FAR   *pcbCursor);
```

Function Arguments

Table 63. SQLGetCursorName Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle
SQLCHAR *	*szCursor*	output	Cursor name
SQLSMALLINT	*cbCursorMax*	input	Length of buffer *szCursor*
SQLSMALLINT *	*pcbCursor*	output	Number of bytes available to return for *szCursor*

Usage

SQLGetCursorName() returns the cursor name set explicitly with SQLSetCursorName(), or if no name is set, it returns the cursor name internally generated by DB2 CLI.

If a name is set explicitly using SQLSetCursorName(), this name is returned until the statement is dropped, or until another explicit name is set.

Internally generated cursor names always begin with SQLCUR or SQL_CUR. For query result sets, DB2 CLI also reserves SQLCURQRS as a cursor name prefix. Cursor names are always 18 characters or less, and are always unique within a connection.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 64. SQLGetCursorName SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The cursor name returned in *szCursor* is longer than the value in *cbCursorMax*, and is truncated to *cbCursorMax* - 1 bytes. The argument *pcbCursor* contains the length of the full cursor name available for return. The function returns SQL_SUCCESS_WITH_INFO.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.
S1092	Option type out of range.	The value specified for the argument *hstmt* is not valid.
S1015	No cursor name available.	There is no open cursor on the statement handle specified by *hstmt* and no cursor name is set with `SQLSetCursorName()`.
S1090	Invalid string or buffer length.	The value specified for the argument *cbCursorMax* is less than 0.

Restrictions

ODBC generated cursor names begin with SQL_CUR. X/Open CLI generated cursor names begin with either SQLCUR or SQL_CUR. DB2 CLI also generates a cursor name that begins with SQLCUR or SQL_CUR.

Example

```
 /******************************************************************/
 /*  DB2 for OS/ESA Example:                                        */
 /*        Performs a positioned update on a column of a cursor.   */
 /******************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include "sqlcli1.h"

int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLHSTMT        hStmt   = SQL_NULL_HSTMT, hStmt2 = SQL_NULL_HSTMT;
   SQLRETURN       rc      = SQL_SUCCESS, rc2 = SQL_SUCCESS;
   SQLINTEGER      RETCODE = 0;
   char            *pDSN = "STLEC1";
   SWORD           cbCursor;
   SDWORD          cbValue1;
   SDWORD          cbValue2;
   char            employee [30];
   int             salary = 0;
   char            cursor_name [20];
   char            update [200];

   char            *stmt = "SELECT NAME, SALARY FROM EMPLOYEE WHERE
                            SALARY > 100000 FOR UPDATE OF SALARY";


   (void) printf ("**** Entering CLIP04.\n\n");

  /*****************************************************************/
  /* Allocate Environment Handle                                   */
  /*****************************************************************/

   RETCODE = SQLAllocEnv(&hEnv);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Allocate Connection Handle to DSN                             */
  /*****************************************************************/

   RETCODE = SQLAllocConnect(hEnv,
                             &hDbc);

   if( RETCODE != SQL_SUCCESS )      // Could not get a Connect Handle
     goto dberror;

  /*****************************************************************/
  /* CONNECT TO data source (STLEC1)                               */
  /*****************************************************************/

   RETCODE = SQLConnect(hDbc,        // Connect handle
                        (SQLCHAR *) pDSN, // DSN
                        SQL_NTS,     // DSN is nul-terminated
                        NULL,        // Null UID
                        0   ,
                        NULL,        // Null Auth string
                        0);

   if( RETCODE != SQL_SUCCESS )      // Connect failed
     goto dberror;

  /*****************************************************************/
  /* Allocate Statement Handles                                    */
  /*****************************************************************/

  rc = SQLAllocStmt (hDbc,
                     &hStmt);

  if (rc != SQL_SUCCESS)
    goto exit;
```

```
rc = SQLAllocStmt (hDbc,
                   &hStmt2);

if (rc != SQL_SUCCESS)
  goto exit;


/*****************************************************************/
/* Execute query to retrieve employee nnames                     */
/*****************************************************************/

rc = SQLExecDirect (hStmt,
                    (SQLCHAR *) stmt,
                    strlen(stmt));

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** EMPLOYEE QUERY FAILED.\n");
  goto dberror;
}

/*****************************************************************/
/* Extract cursor name -- required to build UPDATE statement.    */
/*****************************************************************/

rc = SQLGetCursorName (hStmt,
                       (SQLCHAR *) cursor_name,
                       sizeof(cursor_name),
                       &cbCursor);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** GET CURSOR NAME FAILED.\n");
  goto dberror;
}

(void) printf ("**** Cursor Name is %s.\n");

rc = SQLBindCol (hStmt,           // bind employee name
                 1,
                 SQL_C_CHAR,
                 employee,
                 sizeof(employee),
                 &cbValue1);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** BIND OF NAME FAILED.\n");
  goto dberror;
}

rc = SQLBindCol (hStmt,           // bind employee salary
                 2,
                 SQL_C_LONG,
                 &salary,
                 0,
                 &cbValue2);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** BIND OF SALARY FAILED.\n");
  goto dberror;
}
```

```
/*****************************************************************/
/* Answer Set is available -- Fetch rows and update salary       */
/*****************************************************************/

while (((rc = SQLFetch (hStmt)) == SQL_SUCCESS) &&;
       (rc2 == SQL_SUCCESS))
{
  int new_salary = salary*1.1;

  (void) printf ("**** Employee Name %s with salary %d. New salary = %d.\n",
                 employee,
                 salary,
                 new_salary);

  sprintf (update,
          "UPDATE EMPLOYEE SET SALARY = %d WHERE CURRENT OF %s",
          new_salary,
          cursor_name);

  (void) printf ("***** Update statement is : %s\n", update);

  rc2 = SQLExecDirect (hStmt2,
                       (SQLCHAR *) update,
                       SQL_NTS);
}

if (rc2 != SQL_SUCCESS)
{
  (void) printf ("**** EMPLOYEE UPDATE FAILED.\n");
  goto dberror;
}

/*****************************************************************/
/* Reexecute query to validate that salary was updated           */
/*****************************************************************/

rc = SQLFreeStmt (hStmt,
                  SQL_CLOSE);

rc = SQLExecDirect (hStmt,
                    (SQLCHAR *) stmt,
                    strlen(stmt));

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** EMPLOYEE QUERY FAILED.\n");
  goto dberror;
}

while ((rc = SQLFetch (hStmt)) == SQL_SUCCESS)
{
  (void) printf ("**** Employee Name %s has salary %d.\n",
                 employee,
                 salary);
}
```

```
  /******************************************************************/
  /* Deallocate Statement Handles                                   */
  /******************************************************************/

  rc = SQLFreeStmt (hStmt,
                    SQL_DROP);

  rc = SQLFreeStmt (hStmt2,
                    SQL_DROP);

  /******************************************************************/
  /* DISCONNECT from data source                                    */
  /******************************************************************/

   RETCODE = SQLDisconnect(hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /******************************************************************/
  /* Deallocate Connection Handle                                   */
  /******************************************************************/

   RETCODE = SQLFreeConnect (hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /******************************************************************/
  /* Free Environment Handle                                        */
  /******************************************************************/

   RETCODE = SQLFreeEnv (hEnv);

   if (RETCODE == SQL_SUCCESS)
     goto exit;

   dberror:
   RETCODE=12;

   exit:

   (void) printf ("**** Exiting  CLIP04.\n\n");

   return RETCODE;
}
```

References

- “SQLExecute - Execute a Statement” on page 154
- “SQLExecDirect - Execute a Statement Directly” on page 149
- “SQLPrepare - Prepare a Statement” on page 261
- “SQLSetCursorName - Set Cursor Name” on page 304

SQLGetData - Get Data From a Column

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLGetData() retrieves data for a single column in the current row of the result set. This is an alternative to SQLBindCol(), which is used to transfer data directly into application variables on each SQLFetch() or SQLExtendedFetch() call. SQLGetData() can also be used to retrieve large data values in pieces.

SQLFetch() must be called before SQLGetData().

After calling SQLGetData() for each column, SQLFetch() or SQLExtendedFetch() is called to retrieve the next row.

Syntax

```
SQLRETURN   SQLGetData       (SQLHSTMT          hstmt,
                               SQLUSMALLINT      icol,
                               SQLSMALLINT       fCType,
                               SQLPOINTER        rgbValue,
                               SQLINTEGER        cbValueMax,
                               SQLINTEGER  FAR   *pcbValue);
```

Function Arguments

Table 65 (Page 1 of 2). SQLGetData Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle.
SQLUSMALLINT	*icol*	input	Column number for which the data retrieval is requested.
SQLSMALLINT	*fCType*	input	The C data type of the column identifier by *icol*. The following types are supported: • SQL_C_BINARY • SQL_C_BIT • SQL_C_CHAR • SQL_C_DATE • SQL_C_DBCHAR • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG • SQL_C_SHORT • SQL_C_TIME • SQL_C_TIMESTAMP • SQL_C_TINYINT Specifying SQL_C_DEFAULT results in the data being converted to its default C data type, refer to Table 3 on page 40 for more information.
SQLPOINTER	*rgbValue*	output	Pointer to buffer where the retrieved column data is to be stored.
SQLINTEGER	*cbValueMax*	input	Maximum size of the buffer pointed to by *rgbValue*.

Table 65 (Page 2 of 2). SQLGetData Arguments

Data Type	Argument	Use	Description
SQLINTEGER *	*pcbValue*	output	Pointer to value which indicates the number of bytes DB2 CLI has available to return in the *rgbValue* buffer. If the data is being retrieved in pieces, this contains the number of bytes still remaining. The value is SQL_NULL_DATA if the data value of the column is null. If this pointer is NULL and SQLFetch() has obtained a column containing null data, then this function fails because it has no means of reporting this. If SQLFetch() has fetched a column containing binary data, then the pointer to *pcbValue* must not be NULL or this function fails because it has no other means of informing the application about the length of the data retrieved in the *rgbValue* buffer.

Note: DB2 CLI provides some performance enhancement if *rgbValue* is placed consecutively in memory after *pcbValue.*

Usage

SQLGetData() can be used with SQLBindCol() for the same result set, as long as SQLFetch() and not SQLExtendedFetch() is used. The general steps are:

1. SQLFetch() - advances cursor to first row, retrieves first row, transfers data for bound columns.
2. SQLGetData() - transfers data for the specified column.
3. Repeat step 2 for each column needed.
4. SQLFetch() - advances cursor to next row, retrieves next row, transfers data for bound columns.
5. Repeat steps 2, 3 and 4 for each row in the result set, or until the result set is no longer needed.

SQLGetData() can also be used to retrieve long columns if the C data type (*fCType*) is SQL_C_CHAR, SQL_C_BINARY, SQL_C_DBCHAR, or if *fCType* is SQL_C_DEFAULT and the column type denotes a binary or character string.

Upon each SQLGetData() call, if the data available for return is greater than or equal to *cbValueMax*, truncation occurs. Truncation is indicated by a function return code of SQL_SUCCESS_WITH_INFO coupled with a SQLSTATE denoting data truncation. The application can call SQLGetData() again, with the same *icol* value, to get subsequent data from the same unbound column starting at the point of truncation. To obtain the entire column, the application repeats such calls until the function returns SQL_SUCCESS. The next call to SQLGetData() returns SQL_NO_DATA_FOUND.

Truncation is also affected by the SQL_MAX_LENGTH statement option. The application can specify that truncation is not to be reported by calling SQLSetStmtOption() with SQL_MAX_LENGTH and a value for the maximum length to return for any one column, and by allocating a *rgbValue* buffer of the same size (plus the null-terminator). If the column data is larger than the set maximum length, SQL_SUCCESS is returned and the maximum length, not the actual length is returned in *pcbValue*.

To discard the column data part way through the retrieval, the application can call SQLGetData() with *icol* set to the next column position of interest. To discard data that has not been retrieved for the entire row, the application should call SQLFetch() to advance the cursor to the next row; or, if it is not interested in any more data from the result set, call SQLFreeStmt() to close the cursor.

The *fCType* input argument determines the type of data conversion (if any) needed before the column data is placed into the storage area pointed to by *rgbValue*.

For SQL graphic column data:

- The length of the *rgbValue* buffer (*cbValueMax*) should be a multiple of 2. The application can determine the SQL data type of the column by first calling SQLDescribeCol() or SQLColAttributes().
- The pointer to *pcbValue* must not be NULL since DB2 CLI stores the number of octets stored in *rgbValue*.
- If the data is retrieved in piecewise fashion, DB2 CLI attempts to fill *rgbValue* to the nearest multiple of two octets that is still less than or equal to *cbValueMax*. This means if *cbValueMax* is not a multiple of two, the last byte in that buffer is untouched; DB2 CLI does not split a double-byte character.

The contents returned in *rgbValue* are always null-terminated unless the column data to be retrieved is binary, or if the SQL data type of the column is graphic (DBCS) and the C buffer type is SQL_C_CHAR. If the application is retrieving the data in multiple chunks, it should make the proper adjustments (for example, strip off the null-terminator before concatenating the pieces back together assuming the null termination environment attribute is in effect).

Truncation of numeric data types is reported as a warning if the truncation involves digits to the right of the decimal point. If truncation occurs to the left of the decimal point, an error is returned (refer to the 'Diagnostics' section).

Applications that use SQLExtendedFetch() to retrieve data should call SQLGetData() only when the rowset size is 1 (equivalent to issuing SQLFetch()). SQLGetData() can only retrieve column data for a row where the cursor is currently positioned.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

SQL_NO_DATA_FOUND is returned when the preceding SQLGetData() call has retrieved all of the data for this column.

SQL_SUCCESS is returned if a zero-length string is retrieved by SQLGetData(). If this is the case, *pcbValue* contains 0, and *rgbValue* contains a null terminator.

If the preceding call to SQLFetch() failed, SQLGetData() should not be called since the result is undefined.

Diagnostics

Table 66 (Page 1 of 2). SQLGetData SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	Data returned for the specified column (*icol*) is truncated. String or numeric values are right truncated. SQL_SUCCESS_WITH_INFO is returned.
07006	Invalid conversion.	The data value cannot be converted to the C data type specified by the argument *fCType*. The function has been called before for the same *icol* value but with a different *fCType* value.
22002	Invalid output or indicator buffer specified.	The pointer value specified for the argument pcbValue is a null pointer and the value of the column is null. There is no means to report SQL_NULL_DATA.
22003	Numeric value out of range.	Returning the numeric value (as numeric or string) for the column causes the whole part of the number to be truncated.
22005	Error in assignment.	A returned value is incompatible with the data type denoted by the argument *fCType*.
22008	Datetime field overflow.	Datetime field overflow occurred; for example, an arithmetic operation on a date or timestamp has a result that is not within the valid range of dates, or a datetime value cannot be assigned to a bound variable because it is too small.
24000	Invalid cursor state.	The previous `SQLFetch()` resulted in SQL_ERROR or SQL_NO_DATA found; as a result, the cursor is not positioned on a row.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1002	Invalid column number.	The specified column is less than 0 or greater than the number of result columns. The specified column is 0, but DB2 CLI does not support ODBC bookmarks (*icol* = 0). `SQLExtendedFetch()` is called for this result set.
S1003	Program type out of range.	*fCType* is not a valid data type or SQL_C_DEFAULT.
S1009	Invalid argument value.	The argument *rgbValue* is a null pointer. The argument *pcbValue* is a null pointer; the column SQL data type is graphic (DBCS); and *fcType* is set to SQL_C_CHAR.
S1010	Function sequence error.	The specified *hstmt* is not in a cursor positioned state. The function is called without first calling `SQLFetch()`. The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.

Table 66 (Page 2 of 2). SQLGetData SQLSTATEs

SQLSTATE	Description	Explanation
S1090	Invalid string or buffer length.	The value of the argument *cbValueMax* is less than 0 and the argument *fCType* is SQL_C_CHAR, SQL_C_BINARY, SQL_C_DBCHAR or (SQL_C_DEFAULT and the default type is one of SQL_C_CHAR, SQL_C_BINARY, or SQL_C_DBCHAR).
S1C00	Driver not capable.	The SQL data type for the specified data type is recognized but not supported by DB2 CLI. The requested conversion from the SQL data type to the application data *fCType* cannot be performed by DB2 CLI or the data source. SQLExtendedFetch() is called for the specified *hstmt*.

Restrictions

ODBC has defined column 0 for bookmarks. DB2 CLI does not support bookmarks.

Example

Refer to “Example” on page 167 for a comparison between using bound columns and using SQLGetData().

```
 /******************************************************************/
 /*  DB2 for OS/390 Example:                                       */
 /*        Populates BIOGRAPHY Table from flat file text. Inserts  */
 /*        VITAE in 80-byte pieces via SQLPutData. Also retrieve   */
 /*        NAME, UNIT and VITAE for all members. VITAE is retrieved*/
 /*        via SQLGetData.                                         */
 /******************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include "sqlcli1.h"

#define TEXT_SIZE 80

int insert_bio (SQLHSTMT hStmt,       // insert_bio prototype
                char     *bio,
                int       bcount);

int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLHSTMT        hStmt   = SQL_NULL_HSTMT, hStmt2 = SQL_NULL_HSTMT;
   SQLRETURN       rc      = SQL_SUCCESS;
   FILE            *fp;
   SQLINTEGER      RETCODE = 0;
   char            pTable [200];
   char            *pDSN = "STLEC1";
   UDWORD          pirow;
   SDWORD          cbValue;
   char            *i_stmt = "INSERT INTO BIOGRAPHY VALUES (?, ?, ?)";
   char            *query  = "SELECT NAME, UNIT, VITAE FROM BIOGRAPHY";
   char             text [TEXT_SIZE]; // file text
   char             vitae [3200];    // biography text
   char             Narrative [TEXT_SIZE];
   SQLINTEGER       vitae_ind = SQL_DATA_AT_EXEC; // bio data is
                                      // passed at execute time
                                      // via SQLPutData
   SQLINTEGER       vitae_cbValue = TEXT_SIZE;
   char             *t = NULL;
   char             *c = NULL;
   char             name [21];
   SQLINTEGER       name_ind = SQL_NTS;
   SQLINTEGER       name_cbValue = sizeof(name);
   char             unit [31];
   SQLINTEGER       unit_ind = SQL_NTS;
   SQLINTEGER       unit_cbValue = sizeof(unit);
   char             tmp [80];
   char             *token = NULL, *pbio = vitae;
   char             insert = SQL_FALSE;
   int              i, bcount = 0;

   (void) printf ("**** Entering CLIP09.\n\n");

  /*****************************************************************/
  /* Allocate Environment Handle                                   */
  /*****************************************************************/

   RETCODE = SQLAllocEnv(&hEnv);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Allocate Connection Handle to DSN                             */
  /*****************************************************************/

   RETCODE = SQLAllocConnect(hEnv,
                             &hDbc);

   if( RETCODE != SQL_SUCCESS )      // Could not get a Connect Handle
     goto dberror;

  /*****************************************************************/
  /* CONNECT TO data source (STLEC1)                               */
  /*****************************************************************/

   RETCODE = SQLConnect(hDbc,        // Connect handle
                        (SQLCHAR *) pDSN, // DSN
                        SQL_NTS,     // DSN is nul-terminated
                        NULL,        // Null UID
                        0   ,
                        NULL,        // Null Auth string
                        0);

   if( RETCODE != SQL_SUCCESS )      // Connect failed
     goto dberror;
```

```
/******************************************************************/
/* Allocate Statement Handles                                     */
/******************************************************************/

 rc = SQLAllocStmt (hDbc,
                    &hStmt);

 if (rc != SQL_SUCCESS)
 {
   (void) printf ("**** Allocate Statement Handle Failed.\n");
   goto dberror;
 }

 rc = SQLAllocStmt (hDbc,
                    &hStmt2);

 if (rc != SQL_SUCCESS)
 {
   (void) printf ("**** Allocate Statement Handle Failed.\n");
   goto dberror;
 }

/******************************************************************/
/* Prepare INSERT statement.                                      */
/******************************************************************/

 rc = SQLPrepare (hStmt,
                  (SQLCHAR *) i_stmt,
                  SQL_NTS);

 if (rc != SQL_SUCCESS)
 {
   (void) printf ("**** Prepare of INSERT Failed.\n");
   goto dberror;
 }

/******************************************************************/
/* Bind NAME and UNIT. Bind VITAE so that data can be passed      */
/* via SQLPutData.                                                */
/******************************************************************/

 rc = SQLBindParameter (hStmt,        // bind NAME
                        1,
                        SQL_PARAM_INPUT,
                        SQL_C_CHAR,
                        SQL_CHAR,
                        sizeof(name),
                        0,
                        name,
                        sizeof(name),
                        &name_ind);

 if (rc != SQL_SUCCESS)
 {
   (void) printf ("**** Bind of NAME Failed.\n");
   goto dberror;
 }
```

```
   rc = SQLBindParameter (hStmt,        // bind Branch
                          2,
                          SQL_PARAM_INPUT,
                          SQL_C_CHAR,
                          SQL_CHAR,
                          sizeof(unit),
                          0,
                          unit,
                          sizeof(unit),
                          &unit_ind);

   if (rc != SQL_SUCCESS)
   {
     (void) printf ("**** Bind of UNIT Failed.\n");
     goto dberror;
   }

   rc = SQLBindParameter (hStmt,        // bind Rank
                          3,
                          SQL_PARAM_INPUT,
                          SQL_C_CHAR,
                          SQL_LONGVARCHAR,
                          3200,
                          0,
                          (SQLPOINTER) 3,
                          0,
                          &vitae_ind);

   if (rc != SQL_SUCCESS)
   {
     (void) printf ("**** Bind of VITAE Failed.\n");
     goto dberror;
   }

  /******************************************************************/
  /* Read Biographical text from flat file                          */
  /******************************************************************/

   if ((fp = fopen ("DD:BIOGRAF", "r")) == NULL)  // open command file
   {
     rc = SQL_ERROR;                  // open failed
     goto exit;
   }

/******************************************************************/
/* Process file and insert Biographical text                      */
/******************************************************************/

   while (((t = fgets (text, sizeof(text), fp)) != NULL) &&;
          (rc == SQL_SUCCESS))
   {
     if (text[0] == #')      // if commander data
     {
       if (insert)                     // if BIO data to be inserted
       {
         rc = insert_bio (hStmt,
                          vitae,
                          bcount);     // insert row into BIOGRAPHY Table
         bcount = 0;                   // reset text line count
         pbio   = vitae;               // reset text pointer
       }
       token = strtok (text+1, ",");   // get member NAME
       (void) strcpy (name, token);
       token = strtok (NULL, "#");     // extract UNIT
       (void) strcpy (unit, token);    // copy to local variable
                                       // SQLPutData
       insert = SQL_TRUE;              // have row to insert
     }
     else
     {
       memset (pbio, ' ', sizeof(text));
       strcpy (pbio, text);            // populate text
       i = strlen (pbio);              // remove '\n' and '\0'
       pbio [i--] =' ';
       pbio [i]   =' ';
       pbio += sizeof (text);          // advance pbio
       bcount++;                       // one more text line
     }
   }

   if (insert)                         // if BIO data to be inserted
   {
     rc = insert_bio (hStmt,
                      vitae,
                      bcount);         // insert row into BIOGRAPHY Table
   }

   fclose (fp);                        // close text flat file

  /*****************************************************************/
  /* Commit Insert of rows                                         */
  /*****************************************************************/

   rc = SQLTransact (hEnv,
                     hDbc,
                     SQL_COMMIT);

   if (rc != SQL_SUCCESS)
   {
     (void) printf ("**** COMMIT FAILED.\n");
     goto dberror;
   }

  /*****************************************************************/
  /* Open query to retrieve NAME, UNIT and VITAE. Bind NAME and    */
  /* UNIT but leave VITAE unbound. Retrieved using SQLGetData.     */
  /*****************************************************************/

   RETCODE = SQLPrepare (hStmt2,
                         (SQLCHAR *)query,
                         strlen(query));

   if (RETCODE != SQL_SUCCESS)
   {
     (void) printf ("**** Prepare of Query Failed.\n");
     goto dberror;
   }
   RETCODE = SQLExecute (hStmt2);

   if (RETCODE != SQL_SUCCESS)
   {
     (void) printf ("**** Query Failed.\n");
     goto dberror;
   }

   RETCODE = SQLBindCol (hStmt2,            // bind NAME
                         1,
                         SQL_C_DEFAULT,
                         name,
                         sizeof(name),
                         &name_cbValue);

   if (RETCODE != SQL_SUCCESS)
   {
     (void) printf ("**** Bind of NAME Failed.\n");
     goto dberror;
   }

   RETCODE = SQLBindCol (hStmt2,            // bind UNIT
                         2,
                         SQL_C_DEFAULT,
                         unit,
                         sizeof(unit),
                         &unit_cbValue);

   if (RETCODE != SQL_SUCCESS)
   {
     (void) printf ("**** Bind of UNIT Failed.\n");
     goto dberror;
   }

   while ((RETCODE = SQLFetch (hStmt2)) != SQL_NO_DATA_FOUND)
   {
     (void) printf ("**** Name is %s. Unit is %s.\n\n", name, unit);
     (void) printf ("**** Vitae follows:\n\n");

     for (i = 0; (i < 3200 && RETCODE != SQL_NO_DATA_FOUND); i += TEXT_SIZE)
     {
       RETCODE = SQLGetData (hStmt2,
                             3,
                             SQL_C_CHAR,
                             Narrative,
                             sizeof(Narrative) + 1,
                             &vitae_cbValue);

       if (RETCODE != SQL_NO_DATA_FOUND)
         (void) printf ("%s\n", Narrative);
     }
   }

  /*****************************************************************/
  /* Deallocate Statement Handles                                  */
  /*****************************************************************/

   rc = SQLFreeStmt (hStmt,
                     SQL_DROP);

   rc = SQLFreeStmt (hStmt2,
                     SQL_DROP);
```

```
  /*****************************************************************/
  /* DISCONNECT from data source                                   */
  /*****************************************************************/

   RETCODE = SQLDisconnect(hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Deallocate Connection Handle                                  */
  /*****************************************************************/

   RETCODE = SQLFreeConnect (hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Free Environment Handle                                       */
  /*****************************************************************/

   RETCODE = SQLFreeEnv (hEnv);

   if (RETCODE == SQL_SUCCESS)
     goto exit;

   dberror:
   RETCODE=12;

   exit:

   (void) printf ("**** Exiting  CLIP09.\n\n");

   return RETCODE;
}

/*****************************************************************/
/* function insert_bio is invoked to insert one row into the     */
/* BIOGRAPHY Table. The biography text is inserted in sets of    */
/* 80 bytes via SQLPutData.                                      */
/*****************************************************************/

int insert_bio (SQLHSTMT hStmt,
                char     *vitae,
                int       bcount)
{
  SQLINTEGER       rc = SQL_SUCCESS;
  SQLPOINTER       prgbValue;
  int              i;
  char             *text;
  /*****************************************************************/
  /* NAME and UNIT are bound... VITAE is provided after execution  */
  /* of the INSERT using SQLPutData.                               */
  /*****************************************************************/

  rc = SQLExecute (hStmt);

  if (rc != SQL_NEED_DATA)     // expect SQL_NEED_DATA
  {
    rc = 12;
    (void) printf ("**** NEED DATA not returned.\n");
    goto exit;
  }

  /*****************************************************************/
  /* Invoke SQLParamData to position ODBC driver on input parameter*/
  /*****************************************************************/

  if ((rc = SQLParamData (hStmt,
                        &prgbValue)) != SQL_NEED_DATA)
  {
    rc = 12;
    (void) printf ("**** NEED DATA not returned.\n");
    goto exit;
  }

  /*****************************************************************/
  /* Iterate through VITAE in 80 byte increments.... pass to       */
  /* ODBC Driver via SQLPutData.                                   */
  /*****************************************************************/

  for (i = 0, text = vitae, rc = SQL_SUCCESS;
       (i < bcount) && (rc == SQL_SUCCESS);
       i++, text += TEXT_SIZE)
  {
    rc = SQLPutData (hStmt,
                     text,
                     TEXT_SIZE);
  }

  /*****************************************************************/
  /* Invoke SQLParamData to trigger ODBC driver to execute the     */
  /* statement.                                                    */
  /*****************************************************************/

  if ((rc = SQLParamData (hStmt,
                        &prgbValue)) != SQL_SUCCESS)
  {
    rc = 12;
    (void) printf ("**** INSERT Failed.\n");
  }

  exit:

  return (rc);
}
```

References

- “SQLExtendedFetch - Extended Fetch (Fetch Array of Rows)” on page 157
- “SQLFetch - Fetch Next Row” on page 164

SQLGetEnvAttr - Returns Current Setting of An Environment Attribute

Purpose

Specification:		X/OPEN CLI	ISO CLI

SQLGetEnvAttr() returns the current setting for the specified environment attribute.

These options are set using the SQLSetEnvAttr() function.

Syntax

```
SQLRETURN  SQLGetEnvAttr     (SQLHENV           henv,
                              SQLINTEGER        Attribute,
                              SQLPOINTER        Value,
                              SQLINTEGER        BufferLength,
                              SQLINTEGER  FAR   *StringLength);
```

Function Arguments

Table 67. SQLGetEnvAttr Arguments

Data Type	Argument	Use	Description
SQLHENV	henv	input	Environment handle.
SQLINTEGER	Attribute	input	Attribute to get. Refer to Table 118 on page 307 for the list of environment attributes and their descriptions.
SQLPOINTER	Value	output	The current value associated with *Attribute*. The type of the value returned depends on *Attribute*.
SQLINTEGER	BufferLength	input	Maximum size of buffer pointed to by *Value*, if the attribute value is a character string; otherwise, ignored.
SQLINTEGER *	StringLength	output	Length in bytes of the output data if the attribute value is a character string; otherwise, ignored.

If *Attribute* does not denote a string, then DB2 CLI ignores *BufferLength* and does not set *StringLength*.

Usage

SQLGetEnvAttr() can be called at any time between the allocation and freeing of the environment handle. It obtains the current value of the environment attribute.

For a list of valid environment attributes, refer to Table 118 on page 307.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 68. SQLGetEnvAttr SQLSTATEs

SQLSTATE	Description	Explanation
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1092	Option type out of range.	An invalid *Attribute* value was specified.

Restrictions

None.

Example

```
/* ... */
    rc = SQLGetEnvAttr(henv, SQL_ATTR_OUTPUT_NTS, &output_nts, 0, 0);
    printf("Null Termination of Output strings is: ");
    if (output_nts == SQL_TRUE)
       printf("True\n");
    else
       printf("False\n");
/* ... */
```

References

- “SQLSetEnvAttr - Set Environment Attribute” on page 307

SQLGetFunctions - Get Functions

Purpose

Specification:	**ODBC** 1.0	**X/OPEN CLI**	**ISO CLI**

`SQLGetFunctions()` to query whether a specific function is supported. This allows applications to adapt to varying levels of support when connecting to different database servers.

A connection to a database server must exist before calling this function.

Syntax

```
SQLRETURN   SQLGetFunctions  (SQLHDBC          hdbc,
                              SQLUSMALLINT     fFunction,
                              SQLUSMALLINT FAR  *pfExists);
```

Function Arguments

Table 69. SQLGetFunctions Arguments

Data Type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Database connection handle.
SQLUSMALLINT	*fFunction*	input	The function being queried. Valid *fFunction* values are shown in Figure 7 on page 209
SQLUSMALLINT *	*pfExists*	output	Pointer to location where this function returns SQL_TRUE or SQL_FALSE depending on whether the function being queried is supported.

Usage

Figure 7 on page 209 shows the valid values for the *fFunction* argument and whether the corresponding function is supported.

If *fFunction* is set to SQL_API_ALL_FUNCTIONS, then *pfExists* must point to an SQLSMALLINT array of 100 elements. The array is indexed by the *fFunction* values used to identify many of the functions. Some elements of the array are unused and reserved. Since some *fFunction* values are greater than 100, the array method can not be used to obtain a list of functions. The `SQLGetFunction()` call must be explicitly issued for all *fFunction* values equal to or above 100. The complete set of *fFunction* values is defined in `sqlcli1.h`.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

```
  SQL_API_SQLALLOCCONNECT      = TRUE   SQL_API_SQLALLOCENV          = TRUE
  SQL_API_SQLALLOCSTMT         = TRUE   SQL_API_SQLBINDCOL           = TRUE
  SQL_API_SQLBINDFILETOCOL     = FALSE  SQL_API_SQLBINDFILETOPARAM   = FALSE
  SQL_API_SQLBINDPARAMETER     = TRUE   SQL_API_SQLBROWSECONNECT     = FALSE
  SQL_API_SQLCANCEL            = TRUE   SQL_API_SQLCOLATTRIBUTES     = TRUE
  SQL_API_SQLCOLUMNPRIVILEGES  = TRUE   SQL_API_SQLCOLUMNS           = TRUE
  SQL_API_SQLCONNECT           = TRUE   SQL_API_SQLDATASOURCES       = TRUE
# SQL_API_SQLDESCRIBECOL       = TRUE   SQL_API_SQLDESCRIBEPARAM     = TRUE
  SQL_API_SQLDISCONNECT        = TRUE   SQL_API_SQLDRIVERCONNECT     = TRUE
  SQL_API_SQLERROR             = TRUE   SQL_API_SQLEXECDIRECT        = TRUE
  SQL_API_SQLEXECUTE           = TRUE   SQL_API_SQLEXTENDEDFETCH     = TRUE
  SQL_API_SQLFETCH             = TRUE   SQL_API_SQLFOREIGNKEYS       = TRUE
  SQL_API_SQLFREECONNECT       = TRUE   SQL_API_SQLFREEENV           = TRUE
  SQL_API_SQLFREESTMT          = TRUE   SQL_API_SQLGETCONNECTOPTION  = TRUE
  SQL_API_SQLGETCURSORNAME     = TRUE   SQL_API_SQLGETDATA           = TRUE
  SQL_API_SQLGETENVATTR        = TRUE   SQL_API_SQLGETFUNCTIONS      = TRUE
  SQL_API_SQLGETINFO           = TRUE   SQL_API_SQLGETLENGTH         = FALSE
  SQL_API_SQLGETPOSITION       = FALSE  SQL_API_SQLSQLGETSQLCA       = TRUE
  SQL_API_SQLGETSTMTOPTION     = TRUE   SQL_API_SQLGETSUBSTRING      = FALSE
  SQL_API_SQLGETTYPEINFO       = TRUE   SQL_API_SQLMORERESULTS       = TRUE
  SQL_API_SQLNATIVESQL         = TRUE   SQL_API_SQLNUMPARAMS         = TRUE
  SQL_API_SQLNUMRESULTCOLS     = TRUE   SQL_API_SQLPARAMDATA         = TRUE
  SQL_API_SQLPARAMOPTIONS      = TRUE   SQL_API_SQLPREPARE           = TRUE
  SQL_API_SQLPRIMARYKEYS       = TRUE   SQL_API_SQLPROCEDURECOLUMNS  = TRUE
  SQL_API_SQLPROCEDURES        = TRUE   SQL_API_SQLPUTDATA           = TRUE
  SQL_API_SQLROWCOUNT          = TRUE   SQL_API_SQLSETCOLATTRIBUTES  = TRUE
  SQL_API_SQLSETCONNECTION     = TRUE   SQL_API_SQLSETCONNECTOPTION  = TRUE
  SQL_API_SQLSETCURSORNAME     = TRUE   SQL_API_SQLSETENVATTR        = TRUE
  SQL_API_SQLSETPARAM          = TRUE   SQL_API_SQLSETPOS            = FALSE
  SQL_API_SQLSETSCROLLOPTIONS  = FALSE  SQL_API_SQLSETSTMTOPTION     = TRUE
  SQL_API_SQLSPECIALCOLUMNS    = TRUE   SQL_API_SQLSTATISTICS        = TRUE
  SQL_API_SQLTABLEPRIVILEGES   = TRUE   SQL_API_SQLTABLES            = TRUE
  SQL_API_TRANSACT             = TRUE
```

Figure 7. Supported Functions List

Diagnostics

Table 70. SQLGetFunctions SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	The argument pfExists was a null pointer.
S1010	Function sequence error.	`SQLGetFunctions()` was called before a database connection was established.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.

Restrictions

None.

Example

```
 /******************************************************************/
 /*  DB2 for OS/390 Example:                                       */
 /*        Executes SQLGetFunctions to verify that APIs required   */
 /*        by application are supported.                           */
 /******************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include "sqlcli1.h"

typedef struct odbc_api
{
  SQLUSMALLINT   api;
  char           api_name _40•;
} ODBC_API;

 /******************************************************************/
 /* CLI APIs required by application                               */
 /******************************************************************/

ODBC_API o_api [7] = {
   { SQL_API_SQLBINDPARAMETER, "SQLBindParameter" } ,
   { SQL_API_SQLDISCONNECT   , "SQLDisconnect"    } ,
   { SQL_API_SQLGETTYPEINFO  , "SQLGetTypeInfo"   } ,
   { SQL_API_SQLFETCH        , "SQLFetch"         } ,
   { SQL_API_SQLTRANSACT     , "SQLTransact"      } ,
   { SQL_API_SQLBINDCOL      , "SQLBindCol"       } ,
   { SQL_API_SQLEXECDIRECT   , "SQLExecDirect"    }
                                       } ;

  /******************************************************************/
  /* Validate that required APIs are supported.                    */
  /******************************************************************/

int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLRETURN       rc      = SQL_SUCCESS;
   SQLINTEGER      RETCODE = 0;
   int             i;

   // SQLGetFunctions parameters

   SQLUSMALLINT     fExists  = SQL_TRUE;
   SQLUSMALLINT    *pfExists = &fExists;


   (void) printf ("**** Entering CLIP05.\n\n");
```

```
/*****************************************************************/
/* Allocate Environment Handle                                   */
/*****************************************************************/

 RETCODE = SQLAllocEnv(&hEnv);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;

/*****************************************************************/
/* Allocate Connection Handle to DSN                             */
/*****************************************************************/

 RETCODE = SQLAllocConnect(hEnv,
                           &hDbc);

 if( RETCODE != SQL_SUCCESS )      // Could not get a Connect Handle
   goto dberror;

/*****************************************************************/
/* CONNECT TO data source (STLEC1)                               */
/*****************************************************************/

 RETCODE = SQLConnect(hDbc,        // Connect handle
                      (SQLCHAR *) "STLEC1", // DSN
                      SQL_NTS,     // DSN is nul-terminated
                      NULL,        // Null UID
                      0   ,
                      NULL,        // Null Auth string
                      0);

 if( RETCODE != SQL_SUCCESS )      // Connect failed
   goto dberror;

/*****************************************************************/
/* See if DSN supports required ODBC APIs                        */
/*****************************************************************/

 for (i = 0, (*pfExists = SQL_TRUE);
      (i < (sizeof(o_api)/sizeof(ODBC_API)) && (*pfExists) == SQL_TRUE);
      i++)
 {
   RETCODE = SQLGetFunctions (hDbc,
                              o_api[i].api,
                              pfExists);

   if (*pfExists == SQL_TRUE)      // if api is supported then print
   {
     (void) printf ("**** ODBC api %s IS supported.\n",
                    o_api[i].api_name);
   }
 }

 if (*pfExists == SQL_FALSE)      // a required api is not supported
 {
   (void) printf ("**** ODBC api %s not supported.\n",
                  o_api[i].api_name);
 }
```

```
  /*****************************************************************/
  /* DISCONNECT from data source                                   */
  /*****************************************************************/

   RETCODE = SQLDisconnect(hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Deallocate Connection Handle                                  */
  /*****************************************************************/

   RETCODE = SQLFreeConnect (hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Free Environment Handle                                       */
  /*****************************************************************/

   RETCODE = SQLFreeEnv (hEnv);

   if (RETCODE == SQL_SUCCESS)
     goto exit;

   dberror:
   RETCODE=12;

   exit:

   (void) printf("\n\n**** Exiting  CLIP05.\n\n   ");

   return(RETCODE);
}
```

References

None.

SQLGetInfo - Get General Information

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLGetInfo() returns general information, (including supported data conversions) about the DBMS that the application is currently connected to.

Syntax

```
SQLRETURN   SQLGetInfo       (SQLHDBC          hdbc,
                              SQLUSMALLINT     fInfoType,
                              SQLPOINTER       rgbInfoValue,
                              SQLSMALLINT      cbInfoValueMax,
                              SQLSMALLINT FAR  *pcbInfoValue);
```

Function Arguments

Table 71. SQLGetInfo Arguments

Data Type	Argument	Use	Description
SQLHDBC	*hdbc*	input	Database connection handle
SQLUSMALLINT	*fInfoType*	input	The type of information desired.
SQLPOINTER	*rgbInfoValue*	output (also input)	Pointer to buffer where this function stores the desired information. Depending on the type of information being retrieved, 5 types of information can be returned: • 16-bit integer value • 32-bit integer value • 32-bit binary value • 32-bit mask • Null-terminated character string
SQLSMALLINT	*cbInfoValueMax*	input	Maximum length of the buffer pointed to by *rgbInfoValue* pointer.
SQLSMALLINT *	*pcbInfoValue*	output	Pointer to location where this function returns the total number of bytes available to return the desired information. In the case of string output, this size does not include the null terminating character. If the value in the location pointed to by *pcbInfoValue* is greater than the size of the *rgbInfoValue* buffer as specified in *cbInfoValueMax*, then the string output information is truncated to *cbInfoValueMax* - 1 bytes and the function returns with SQL_SUCCESS_WITH_INFO.

Usage

Refer to Table 72 on page 214 for a list of the possible values of *fInfoType* and a description of the information that SQLGetInfo() would return for that value.

Table 72 (Page 1 of 13). Information Returned By SQLGetInfo

fInfoType	**Format**	**Description and Notes**
Note: Call Level Interface returns a value for each *fInfoType* in this table. If the *fInfoType* does not apply or is not supported, the result is dependent on the return type. If the return type is a: • Character string containing 'Y' or 'N', "N" is returned. • Character string containing a value other than just 'Y' or 'N', an empty string is returned. • 16-bit integer, 0 (zero). • 32-bit integer, 0 (zero). • 32-bit mask, 0 (zero).		
SQL_ACCESSIBLE_PROCEDURES [20]	string	A character string of "Y" indicates that the user can execute all procedures returned by the function `SQLProcedures()`. "N" indicates that procedures can be returned that the user cannot execute.
SQL_ACCESSIBLE_TABLES [19]	string	A character string of "Y" indicates that the user is guaranteed SELECT privilege to all tables returned by the function `SQLTables()`. "N" indicates that tables can be returned that the user cannot access.
SQL_ACTIVE_CONNECTIONS [0]	16-bit integer	The maximum number of active connections supported per application. Zero is returned, indicating that the limit is dependent on system resources. The MAXCONN keyword in the initialization file or the SQL_MAX_CONNECTIONS environment/connection option can be used to impose a limit on the number of connections. This limit is returned if it is set to any value other than zero.
SQL_ACTIVE_STATEMENTS [1]	16-bit integer	The maximum number of active statements per connection. Zero is returned, indicating that the limit is dependent on database system and Call Level Interface resources, and limits.
SQL_ALTER_TABLE [86]	32-bit mask	Indicates which clauses in ALTER TABLE are supported by the DBMS. • SQL_AT_ADD_COLUMN • SQL_AT_DROP_COLUMN
SQL_BOOKMARK_PERSISTENCE [82]	32-bit mask	Reserved option, zero is returned for the bit-mask.
SQL_COLUMN_ALIAS [87]	string	Returns "Y" if column aliases are supported, or "N" if they are not.
SQL_CONCAT_NULL_BEHAVIOR [22]	16-bit integer	Indicates how the concatenation of NULL valued character data type columns with non-NULL valued character data type columns is handled. • SQL_CB_NULL - indicates the result is a NULL value (this is the case for IBM RDBMs). • SQL_CB_NON_NULL - indicates the result is a concatenation of non-NULL column values.

Table 72 (Page 2 of 13). Information Returned By SQLGetInfo

fInfoType	**Format**	**Description and Notes**
SQL_CONVERT_BIGINT [53] SQL_CONVERT_BINARY [54] SQL_CONVERT_BIT [55] SQL_CONVERT_CHAR [56] SQL_CONVERT_DATE [57] SQL_CONVERT_DECIMAL [58] SQL_CONVERT_DOUBLE [59] SQL_CONVERT_FLOAT [60] SQL_CONVERT_INTEGER [61] SQL_CONVERT_LONGVARBINARY [71] SQL_CONVERT_LONGVARCHAR [62] SQL_CONVERT_NUMERIC [63] SQL_CONVERT_REAL [64] SQL_CONVERT_SMALLINT [65] SQL_CONVERT_TIME [66] SQL_CONVERT_TIMESTAMP [67] SQL_CONVERT_TINYINT [68] SQL_CONVERT_VARBINARY [69] SQL_CONVERT_VARCHAR [70]	32-bit mask	Indicates the conversions supported by the data source with the CONVERT scalar function for data of the type named in the *finfoType*. If the bitmask equals zero, the data source does not support any conversions for the data of the named type, including conversions to the same data type. For example, to find out if a data source supports the conversion of SQL_INTEGER data to the SQL_DECIMAL data type, an application calls `SQLGetInfo()` with *finfoType* of SQL_CONVERT_INTEGER. The application then ANDs the returned bitmask with SQL_CVT_DECIMAL. If the resulting value is nonzero then the conversion is supported. The following bitmasks are used to determine which conversions are supported: • SQL_CVT_BIGINT • SQL_CVT_BINARY • SQL_CVT_BIT • SQL_CVT_CHAR • SQL_CVT_DATE • SQL_CVT_DECIMAL • SQL_CVT_DOUBLE • SQL_CVT_FLOAT • SQL_CVT_INTEGER • SQL_CVT_LONGVARBINARY • SQL_CVT_LONGVARCHAR • SQL_CVT_NUMERIC • SQL_CVT_REAL • SQL_CVT_SMALLINT • SQL_CVT_TIME • SQL_CVT_TIMESTAMP • SQL_CVT_TINYINT • SQL_CVT_VARBINARY • SQL_CVT_VARCHAR
SQL_CONVERT_FUNCTIONS [48]	32-bit mask	Indicates the scalar conversion functions supported by the driver and associated data source.
SQL_CORRELATION_NAME [74]	16-bit integer	Indicates the degree of correlation name support by the server: • SQL_CN_ANY, supported and can be any valid user-defined name. • SQL_CN_NONE, correlation name not supported. • SQL_CN_DIFFERENT, correlation name supported but it must be different than the name of the table that it represents.
SQL_CLOSE_BEHAVIOR	32-bit integer	Indicates whether or not locks are released when the cursor is closed. The possible values are: • SQL_CC_NO_RELEASE: locks are not released when the cursor on this statement handle is closed. This is the default. • SQL_CC_RELEASE: locks are released when the cursor on this statement handle is closed. Typically cursors are explicitly closed when the function `SQLFreeStmt()` is called with the SQL_CLOSE or SQL_DROP option. In addition, the end of the transaction (when a commit or rollback is issued) can also cause the closing of the cursor (depending on the WITH HOLD attribute currently in use).

Table 72 (Page 3 of 13). Information Returned By SQLGetInfo

fInfoType	**Format**	**Description and Notes**
SQL_CURSOR_COMMIT_BEHAVIOR	16-bit integer	Indicates how a COMMIT operation affects cursors. A value of: • SQL_CB_DELETE, destroys cursors and drops access plans for dynamic SQL statements. • SQL_CB_CLOSE, destroys cursors, but retains access plans for dynamic SQL statements (including non-query statements) • SQL_CB_PRESERVE, retains cursors and access plans for dynamic statements (including non-query statements). Applications can continue to fetch data, or close the cursor and re-execute the query without re-preparing the statement. After COMMIT, a FETCH must be issued to reposition the cursor before actions such as positioned updates or deletes can be taken.
SQL_CURSOR_ROLLBACK_BEHAVIOR [24]	16-bit integer	Indicates how a ROLLBACK operation affects cursors. A value of: • SQL_CB_DELETE, destroys cursors and drops access plans for dynamic SQL statements. • SQL_CB_CLOSE, destroys cursors, but retains access plans for dynamic SQL statements (including non-query statements) • SQL_CB_PRESERVE, retains cursors and access plans for dynamic statements (including non-query statements). Applications can continue to fetch data, or close the cursor and re-execute the query without re-preparing the statement. DB2 servers do not have the SQL_CB_PRESERVE property.
SQL_DATA_SOURCE_NAME [2]	string	The name used as data source on the input to `SQLConnect()`, or the DSN keyword value in the `SQLDriverConnect()` connection string.
SQL_DATA_SOURCE_READ_ONLY [25]	string	A character string of "Y" indicates that the database is set to READ ONLY mode; an "N" indicates that it is not set to READ ONLY mode.
SQL_DATABASE_NAME [16]	string	The name of the current database in use. **Note:** Also returned by SELECT CURRENT SERVER on IBM DBMS's.
SQL_DBMS_NAME [17]	string	The name of the DBMS product being accessed. For example: • "DB2/6000" • "DB2/2"
SQL_DBMS_VER [18]	string	The Version of the DBMS product accessed. A string of the form 'mm.vv.rrrr' where mm is the major version, vv is the minor version and rrrr is the release. For example, "02.01.0000" translates to major version 2, minor version 1, release 0.

Table 72 (Page 4 of 13). Information Returned By SQLGetInfo

fInfoType	**Format**	**Description and Notes**
SQL_DEFAULT_TXN_ISOLATION [26]	32-bit mask	The default transaction isolation level supported. One of the following masks are returned: • SQL_TXN_READ_UNCOMMITTED = Changes are immediately perceived by all transactions (dirty read, non-repeatable read, and phantoms are possible). This is equivalent to IBM's UR level. • SQL_TXN_READ_COMMITTED = Row read by transaction 1 can be altered and committed by transaction 2 (non-repeatable read and phantoms are possible) This is equivalent to IBM's CS level. • SQL_TXN_REPEATABLE_READ = A transaction can add or remove rows matching the search condition or a pending transaction (repeatable read, but phantoms are possible) This is equivalent to IBM's RS level. • SQL_TXN_SERIALIZABLE = Data affected by pending transaction is not available to other transactions (repeatable read, phantoms are not possible) This is equivalent to IBM's RR level. • SQL_TXN_VERSIONING = Not applicable to IBM DBMSs. • SQL_TXN_NOCOMMIT = Any chnages are effectively committed at the end of a successful operation; no explicit commit or rollback is allowed. This is a DB2 for OS/400 isolation level. In IBM terminology, • SQL_TXN_READ_UNCOMMITTED is uncommitted read; • SQL_TXN_READ_COMMITTED is cursor stability; • SQL_TXN_REPEATABLE_READ is read stability; • SQL_TXN_SERIALIZABLE is repeatable read.
SQL_DRIVER_HDBC [3]	32 bits	Call Level Interface's current database handle.
SQL_DRIVER_HENV [4]	32 bits	Call Level Interface's environment handle.
SQL_DRIVER_HLIB [76]	32 bits	Reserved.
SQL_DRIVER_HSTMT [5]	32 bits	Call Level Interface's current statement handle for the current connection.
SQL_DRIVER_NAME [6]	string	The file name of the Call Level Interface implementation. DB2 CLI returns NULL.
SQL_DRIVER_ODBC_VER [77]	string	The version number of ODBC that the Driver supports. Call Level Interface returns "2.1".
SQL_DRIVER_VER [7]	string	The version of the CLI driver. A string of the form 'mm.vv.rrrr' where mm is the major version, vv is the minor version and rrrr is the release. For example, "02.01.0000" translates to major version 2, minor version 1, release 0.
SQL_EXPRESSIONS_IN_ORDERBY [27]	string	The character string "Y" indicates the database server supports the DIRECT specification of expressions in the ORDER BY list, "N" indicates that is does not.

Table 72 (Page 5 of 13). Information Returned By SQLGetInfo

fInfoType	**Format**	**Description and Notes**
SQL_FETCH_DIRECTION [8]	32-bit mask	The supported fetch directions. The following bit-masks are used in conjunction with the flag to determine which options are supported. • SQL_FD_FETCH_NEXT • SQL_FD_FETCH_FIRST • SQL_FD_FETCH_LAST • SQL_FD_FETCH_PREV • SQL_FD_FETCH_ABSOLUTE • SQL_FD_FETCH_RELATIVE • SQL_FD_FETCH_RESUME
SQL_FILE_USAGE [84]	16-bit integer	Reserved. Zero is returned.
SQL_GETDATA_EXTENSIONS [81]	32-bit mask	Indicates whether extensions to the `SQLGetData()` function are supported. The following extensions are currently identified and supported by Call Level Interface: • SQL_GD_ANY_COLUMN, `SQLGetData()` can be called for unbound columns that precede the last bound column. • SQL_GD_ANY_ORDER, `SQLGetData()` can be called for columns in any order. ODBC also defines SQL_GD_BLOCK and SQL_GD_BOUND; these bits are not returned by Call Level Interface.
SQL_GROUP_BY [88]	16-bit integer	Indicates the degree of support for the GROUP BY clause by the server: • SQL_GB_NO_RELATION, there is no relationship between the columns in the GROUP BY and in the SELECT list • SQL_GB_NOT_SUPPORTED, GROUP BY not supported • SQL_GB_GROUP_BY_EQUALS_SELECT, GROUP BY must include all non-aggregated columns in the select list. • SQL_GB_GROUP_BY_CONTAINS_SELECT, the GROUP BY clause must contain all non-aggregated columns in the SELECT list.
SQL_IDENTIFIER_CASE [28]	16-bit integer	Indicates case sensitivity of object names (such as table-name). A value of: • SQL_IC_UPPER = identifier names are stored in upper case in the system catalog. • SQL_IC_LOWER = identifier names are stored in lower case in the system catalog. • SQL_IC_SENSITIVE = identifier names are case sensitive, and are stored in mixed case in the system catalog. • SQL_IC_MIXED = identifier names are not case sensitive, and are stored in mixed case in the system catalog. **Note:** Identifier names in IBM DBMSs are not case sensitive.
SQL_IDENTIFIER_QUOTE_CHAR [29]	string	Indicates the character used to surround a delimited identifier.
SQL_KEYWORDS [89]	sting	This is a string of all the *keywords* at the DBMS that are not in the ODBC's list of reserved words.
SQL_LIKE_ESCAPE_CLAUSE [113]	string	A character string that indicates if an escape character is supported for the metacharacters percent and underscore in a LIKE predicate.
SQL_LOCK_TYPES [78]	32-bit mask	Reserved option, zero is returned for the bit-mask.
SQL_MAX_BINARY_LITERAL_LEN [112]	32-bit integer	A 32-bit integer value specifying the maximum length of a hexadecimal literal in a SQL statement.

Table 72 (Page 6 of 13). Information Returned By SQLGetInfo

fInfoType	**Format**	**Description and Notes**
SQL_MAX_CHAR_LITERAL_LEN [108]	32-bit integer	The maximum length of a character literal in an SQL statement (in bytes).
SQL_MAX_COLUMN_NAME_LEN [30]	16-bit integer	The maximum length of a column name (in bytes).
SQL_MAX_COLUMNS_IN_GROUP_BY [97]	16-bit integer	Indicates the maximum number of columns that the server supports in a GROUP BY clause. Zero if no limit.
SQL_MAX_COLUMNS_IN_INDEX [98]	16-bit integer	Indicates the maximum number of columns that the server supports in an index. Zero if no limit.
SQL_MAX_COLUMNS_IN_ORDER_BY [99]	16-bit integer	Indicates the maximum number of columns that the server supports in an ORDER BY clause. Zero if no limit.
SQL_MAX_COLUMNS_IN_SELECT [100]	16-bit integer	Indicates the maximum number of columns that the server supports in a select list. Zero if no limit.
SQL_MAX_COLUMNS_IN_TABLE [101]	16-bit integer	Indicates the maximum number of columns that the server supports in a base table. Zero if no limit.
SQL_MAX_CURSOR_NAME_LEN [31]	16-bit integer	The maximum length of a cursor name (in bytes).
SQL_MAX_INDEX_SIZE [102]	32-bit integer	Indicates the maximum size in bytes that the server supports for the combined columns in an index. Zero if no limit.
SQL_MAX_OWNER_NAME_LEN [32] SQL_MAX_SCHEMA_NAME_LEN	16-bit integer	The maximum length of a schema qualifier name (in bytes).
SQL_MAX_PROCEDURE_NAME_LEN [33]	16-bit integer	The maximum length of a procedure name (in bytes).
SQL_MAX_QUALIFIER_NAME_LEN [34] SQL_MAX_CATALOG_NAME_LEN	16-bit integer	The maximum length of a catalog qualifier name; first part of a 3 part table name (in bytes).
SQL_MAX_ROW_SIZE [104]	32-bit integer	Specifies the maximum length in bytes that the server supports in single row of a base table. Zero if no limit.
SQL_MAX_ROW_SIZE_INCLUDES_LONG [103]	string	Set to "Y" to indicate that the value returned by SQL_MAX_ROW_SIZE *fInfoType* includes the length of product-specific *long string* data types. Otherwise, set to "N".
SQL_MAX_STATEMENT_LEN [105]	32-bit integer	Indicates the maximum length of an SQL statement string in bytes, including the number of white spaces in the statement.
SQL_MAX_TABLE_NAME_LEN [35]	16-bit integer	The maximum length of a table name (in bytes).
SQL_MAX_TABLES_IN_SELECT [106]	16-bit integer	Indicates the maximum number of table names allowed in a FROM clause in a <query specification>.
SQL_MAX_USER_NAME_LEN [107]	16-bit integer	Indicates the maximum size allowed for a <user identifier> (in bytes).
SQL_MULT_RESULT_SETS [36]	string	The character string "Y" indicates that the database supports multiple result sets, "N" indicates that it does not.
# SQL_MULTIPLE_ACTIVE_TXN [37]	string	The character string "Y" indicates that active transactions on multiple connections are allowed. "N" indicates that only one connection at a time can have an active transaction.
SQL_NEED_LONG_DATA_LEN [111]	string	A character string reserved for the use of ODBC. "N is" always returned.
SQL_NON_NULLABLE_COLUMNS [75]	16-bit integer	Indicates whether non-nullable columns are supported: • SQL_NNC_NON_NULL, columns can be defined as NOT NULL. • SQL_NNC_NULL, columns can not be defined as NOT NULL.
SQL_NULL_COLLATION [85]	16-bit integer	Indicates where NULLs are sorted in a list: • SQL_NC_HIGH, null values sort high • SQL_NC_LOW, to indicate that null values sort low

Table 72 (Page 7 of 13). Information Returned By SQLGetInfo

fInfoType	**Format**	**Description and Notes**
SQL_NUMERIC_FUNCTIONS [49]	32-bit mask	Indicates the ODBC scalar numeric functions supported. These functions are intended to be used with the ODBC vendor escape sequence described in “Using Vendor Escape Clauses” on page 376. The following bit-masks are used to determine which numeric functions are supported: • SQL_FN_NUM_ABS • SQL_FN_NUM_ACOS • SQL_FN_NUM_ASIN • SQL_FN_NUM_ATAN • SQL_FN_NUM_ATAN2 • SQL_FN_NUM_CEILING • SQL_FN_NUM_COS • SQL_FN_NUM_COT • SQL_FN_NUM_DEGREES • SQL_FN_NUM_EXP • SQL_FN_NUM_FLOOR • SQL_FN_NUM_LOG • SQL_FN_NUM_LOG10 • SQL_FN_NUM_MOD • SQL_FN_NUM_PI • SQL_FN_NUM_POWER • SQL_FN_NUM_RADIANS • SQL_FN_NUM_RAND • SQL_FN_NUM_ROUND • SQL_FN_NUM_SIGN • SQL_FN_NUM_SIN • SQL_FN_NUM_SQRT • SQL_FN_NUM_TAN • SQL_FN_NUM_TRUNCATE
SQL_ODBC_API_CONFORMANCE [9]	16-bit integer	The level of ODBC conformance. • SQL_OAC_NONE • SQL_OAC_LEVEL1 • SQL_OAC_LEVEL2
SQL_ODBC_SAG_CLI_CONFORMANCE [12]	16-bit integer	The compliance to the functions of the SQL Access Group (SAG) CLI specification. A value of: • SQL_OSCC_NOT_COMPLIANT - the driver is not SAG-compliant. • SQL_OSCC_COMPLIANT - the driver is SAG-compliant.
SQL_ODBC_SQL_CONFORMANCE [15]	16-bit integer	A value of: • SQL_OSC_MINIMUM - means minimum ODBC SQL grammar supported • SQL_OSC_CORE - means core ODBC SQL Grammar supported • SQL_OSC_EXTENDED - means extended ODBC SQL Grammar supported For the definition of the above 3 types of ODBC SQL grammar, see *ODBC 2.0 Programmer's Reference and SDK Guide.*
SQL_ODBC_SQL_OPT_IEF [73]	string	The "Y" character string indicates that the data source supports Integrity Enhanced Facility (IEF) in SQL89 and in X/Open XPG4 Embedded SQL; an "N" indicates it does not.
SQL_ODBC_VER [10]	string	The version number of ODBC that the driver manager supports. Call Level Interface returns the string "02.10".

Table 72 (Page 8 of 13). Information Returned By SQLGetInfo

fInfoType	**Format**	**Description and Notes**
SQL_OJ_CAPABILITIES [65003]	32-bit mask	A 32-bit bit-mask enumerating the types of outer join supported. The bitmasks are: • SQL_OJ_LEFT : Left outer join is supported. • SQL_OJ_RIGHT : Right outer join is supported. • SQL_OJ_FULL : Full outer join is supported. • SQL_OJ_NESTED : Nested outer join is supported. • SQL_OJ_NOT_ORDERED : The order of the tables underlying the columns in the outer join ON clause need not be in the same order as the tables in the JOIN clause. • SQL_OJ_INNER : The inner table of an outer join can also be an inner join. • SQL_OJ_ALL_COMPARISONS_OPS : Any predicate can be used in the outer join ON clause. If this bit is not set, the equality (=) operator is the only valid comparison operator in the ON clause.
SQL_ORDER_BY_COLUMNS_IN_SELECT [90]	string	Set to "Y" if columns in the ORDER BY clauses must be in the select list; otherwise set to "N".
SQL_OUTER_JOINS [38]	string	The character string: • "Y" indicates that outer joins are supported, and Call Level Interface supports the ODBC outer join request syntax. • "N" indicates that it is not supported. (See “Using Vendor Escape Clauses” on page 376)
SQL_OWNER_TERM [39] SQL_SCHEMA_TERM	string	The database vendor's terminology for a schema (owner)
SQL_OWNER_USAGE [91]	32-bit mask	Indicates the type of SQL statements that have schema (owners) associated with them when these statements are executed. Schema qualifiers (owners) are: • SQL_OU_DML_STATEMENTS - supported in all DML statements. • SQL_OU_PROCEDURE_INVOCATION - supported in the procedure invocation statement. • SQL_OU_TABLE_DEFINITION - supported in all table definition statements. • SQL_OU_INDEX_DEFINITION - supported in all index definition statements. • SQL_OU_PRIVILEGE_DEFINITION - supported in all privilege definition statements (i.e. grant and revoke statements).
SQL_POS_OPERATIONS [79]	32-bit mask	Reserved option, zero is returned for the bit-mask.
SQL_POSITIONED_STATEMENTS [80]	32-bit mask	Indicates the degree of support for positioned UPDATE and positioned DELETE statements: • SQL_PS_POSITIONED_DELETE • SQL_PS_POSITIONED_UPDATE • SQL_PS_SELECT_FOR_UPDATE, indicates whether or not the server requires the FOR UPDATE clause to be specified on a <query expression> in order for a column to be updateable via the cursor.
SQL_PROCEDURE_TERM [40]	string	The name a database vendor uses for a procedure

Table 72 (Page 9 of 13). Information Returned By SQLGetInfo

fInfoType	**Format**	**Description and Notes**
SQL_PROCEDURES [21]	string	A character string of "Y" indicates that the data source supports procedures and Call Level Interface supports the ODBC procedure invocation syntax specified in "Using Stored Procedures" on page 361. "N" indicates that it does not.
SQL_QUALIFIER_LOCATION [114]	16-bit integer	A 16-bit integer value indicated the position of the qualifier in a qualified table name. Zero indicates that qualified names are not supported.
SQL_QUALIFIER_NAME_SEPARATOR [41] SQL_CATALOG_NAME_SEPARATOR	string	The characters used as a separator between a catalog name and the qualified name element that follows it.
SQL_QUALIFIER_TERM [42] SQL_CATALOG_TERM	string	The database vendor's terminology for a qualifier. The name that the vendor uses for the high order part of a three part name. Since Call Level Interface does not support three part names, a zero-length string is returned. For non-ODBC applications, the SQL_CATALOG_TERM symbolic name should be used instead of SQL_QUALIFIER_NAME.
SQL_QUALIFIER_USAGE [92] SQL_CATALOG_USAGE	32-bit mask	This is similar to SQL_OWNER_USAGE except that this is used for catalog.
SQL_QUOTED_IDENTIFIER_CASE [93]	16-bit integer	Returns: • SQL_IC_UPPER - quoted identifiers in SQL are case insensitive and stored in upper case in the system catalog. • SQL_IC_LOWER - quoted identifiers in SQL are case insensitive and are stored in lower case in the system catalog. • SQL_IC_SENSITIVE - quoted identifiers (delimited identifiers) in SQL are case sensitive and are stored in mixed case in the system catalog. • SQL_IC_MIXED - quoted identifiers in SQL are case insensitive and are stored in mixed case in the system catalog. This should be contrasted with the SQL_IDENTIFIER_CASE *fInfoType* which is used to determine how (unquoted) identifiers are stored in the system catalog.
SQL_ROW_UPDATES [11]	string	A character string of "Y" indicates changes are detected in rows between multiple fetches of the same rows, "N" indicates that changes are not detected.
SQL_SCROLL_CONCURRENCY [43]	32-bit mask	Indicates the concurrency options supported for the cursor. The following bit-masks are used in conjunction with the flag to determine which options are supported: • SQL_SCCO_READ_ONLY • SQL_SCCO_LOCK • SQL_SCCO_OPT_TIMESTAMP • SQL_SCCO_OPT_VALUES Call Level Interface returns SQL_SCCO_LOCK indicating that the lowest level of locking that is sufficient to ensure the row can be updated is used.

Table 72 (Page 10 of 13). Information Returned By SQLGetInfo

fInfoType	**Format**	**Description and Notes**
SQL_SCROLL_OPTIONS [44]	32-bit mask	The scroll options supported for scrollable cursors. The following bit-masks are used in conjunction with the flag to determine which options are supported: • SQL_SO_FORWARD_ONLY • SQL_SO_KEYSET_DRIVEN • SQL_SO_STATIC • SQL_SO_DYNAMIC • SQL_SO_MIXED Call Level Interface returns SQL_SO_FORWARD_ONLY, indicating that the cursor scrolls forward only.
SQL_SEARCH_PATTERN_ESCAPE [14]	string	Used to specify what the driver supports as an escape character for catalog functions such as (SQLTables(), SQLColumns()).
SQL_SERVER_NAME [13]	string	The name of DB2 subsystem to which the application is connected.
SQL_SPECIAL_CHARACTERS [94]	string	Contains all the characters in addition to a...z, A...Z, 0...9, and _ that the server allows in non-delimited identifiers.
SQL_STATIC_SENSITIVITY [83]	32-bit mask	Indicates whether changes made by an application with a positioned UPDATE or DELETE statement can be detected by that application: • SQL_SS_ADDITIONS: Added rows are visible to the cursor; the cursor can scroll to these rows. All DB2 servers see added rows. • SQL_SS_DELETIONS: Deleted rows are no longer available to the cursor and do not leave a hole in the result set; after the cursor scrolls from a deleted row, it cannot return to that row. • SQL_SS_UPDATES: Updates to rows are visible to the cursor; if the cursor scrolls from and returns to an updated row, the data returned by the cursor is the updated data, not the original data.

Table 72 (Page 11 of 13). Information Returned By SQLGetInfo

fInfoType	**Format**	**Description and Notes**
SQL_STRING_FUNCTIONS [50]	32-bit mask	Indicates which string functions are supported. The following bit-masks are used to determine which string functions are supported: • SQL_FN_STR_ASCII • SQL_FN_STR_CHAR • SQL_FN_STR_CONCAT • SQL_FN_STR_DIFFERENCE • SQL_FN_STR_INSERT • SQL_FN_STR_LCASE • SQL_FN_STR_LEFT • SQL_FN_STR_LENGTH • SQL_FN_STR_LOCATE • SQL_FN_STR_LOCATE_2 • SQL_FN_STR_LTRIM • SQL_FN_STR_REPEAT • SQL_FN_STR_REPLACE • SQL_FN_STR_RIGHT • SQL_FN_STR_RTRIM • SQL_FN_STR_SOUNDEX • SQL_FN_STR_SPACE • SQL_FN_STR_SUBSTRING • SQL_FN_STR_UCASE If an application can call the LOCATE scalar function with the *string1, string2,* and *start* arguments, the SQL_FN_STR_LOCATE bitmask is returned. If an application can only call the LOCATE scalar function with the *string1* and *string2*, the SQL_FN_STR_LOCATE_2 bitmask is returned. If the LOCATE scalar function is fully supported, both bitmasks are returned.
SQL_SUBQUERIES [95]	32-bit mask	Indicates which predicates support subqueries: • SQL_SQ_COMPARISION - the *comparison* predicate • SQL_SQ_CORRELATE_SUBQUERIES - all predicates • SQL_SQ_EXISTS - the *exists* predicate • SQL_SQ_IN - the *in* predicate • SQL_SQ_QUANTIFIED - the predicates containing a quantification scalar function.
SQL_SYSTEM_FUNCTIONS [51]	32-bit mask	Indicates which scalar system functions are supported. The following bit-masks are used to determine which scalar system functions are supported: • SQL_FN_SYS_DBNAME • SQL_FN_SYS_IFNULL • SQL_FN_SYS_USERNAME **Note:** These functions are intended to be used with the escape sequence in ODBC.
SQL_TABLE_TERM [45]	string	The database vendor's terminology for a table.

Table 72 (Page 12 of 13). Information Returned By SQLGetInfo

fInfoType	**Format**	**Description and Notes**
SQL_TIMEDATE_ADD_INTERVALS [109]	32-bit mask	Indicates whether or not the special ODBC system function TIMESTAMPADD is supported, and, if it is, which intervals are supported. The following bitmasks are used to determine which intervals are supported: • SQL_FN_TSI_FRAC_SECOND • SQL_FN_TSI_SECOND • SQL_FN_TSI_MINUTE • SQL_FN_TSI_HOUR • SQL_FN_TSI_DAY • SQL_FN_TSI_WEEK • SQL_FN_TSI_MONTH • SQL_FN_TSI_QUARTER • SQL_FN_TSI_YEAR
SQL_TIMEDATE_DIFF_INTERVALS [110]	32-bit mask	Indicates whether or not the special ODBC system function TIMESTAMPDIFF is supported, and, if it is, which intervals are supported. The following bitmasks are used to determine which intervals are supported: • SQL_FN_TSI_FRAC_SECOND • SQL_FN_TSI_SECOND • SQL_FN_TSI_MINUTE • SQL_FN_TSI_HOUR • SQL_FN_TSI_DAY • SQL_FN_TSI_WEEK • SQL_FN_TSI_MONTH • SQL_FN_TSI_QUARTER • SQL_FN_TSI_YEAR
SQL_TIMEDATE_FUNCTIONS [52]	32-bit mask	Indicates which time and date functions are supported. The following bit-masks are used to determine which date functions are supported: • SQL_FN_TD_CURDATE • SQL_FN_TD_CURTIME • SQL_FN_TD_DAYNAME • SQL_FN_TD_DAYOFMONTH • SQL_FN_TD_DAYOFWEEK • SQL_FN_TD_DAYOFYEAR • SQL_FN_TD_HOUR • SQL_FN_TD_JULIAN_DAY • SQL_FN_TD_MINUTE • SQL_FN_TD_MONTH • SQL_FN_TD_MONTHNAME • SQL_FN_TD_NOW • SQL_FN_TD_QUARTER • SQL_FN_TD_SECOND • SQL_FN_TD_SECONDS_SINCE_MIDNIGHT • SQL_FN_TD_TIMESTAMPADD • SQL_FN_TD_TIMESTAMPDIFF • SQL_FN_TD_WEEK • SQL_FN_TD_YEAR **Note:** These functions are intended to be used with the escape sequence in ODBC.

Table 72 (Page 13 of 13). Information Returned By SQLGetInfo

fInfoType	**Format**	**Description and Notes**
SQL_TXN_CAPABLE [46]	16-bit integer	Indicates whether transactions can contain DDL or DML or both. • SQL_TC_NONE = transactions not supported. • SQL_TC_DML = transactions can only contain DML statements (SELECT, INSERT, UPDATE, DELETE, etc.) DDL statements (CREATE TABLE, DROP INDEX, etc.) encountered in a transaction cause an error. • SQL_TC_DDL_COMMIT = transactions can only contain DML statements. DDL statements encountered in a transaction cause the transaction to be committed. • SQL_TC_DDL_IGNORE = transactions can only contain DML statements. DDL statements encountered in a transaction are ignored. • SQL_TC_ALL = transactions can contain DDL and DML statements in any order.
SQL_TXN_ISOLATION_OPTION [72]	32-bit mask	The transaction isolation levels available at the currently connected database server. The following masks are used in conjunction with the flag to determine which options are supported: • SQL_TXN_READ_UNCOMMITTED • SQL_TXN_READ_COMMITTED • SQL_TXN_REPEATABLE_READ • SQL_TXN_SERIALIZABLE • SQL_TXN_NOCOMMIT • SQL_TXN_VERSIONING For descriptions of each level refer to SQL_DEFAULT_TXN_ISOLATION.
SQL_UNION [96]	32-bit mask	Indicates if the server supports the UNION operator: • SQL_U_UNION - supports the UNION clause • SQL_U_UNION_ALL - supports the ALL keyword in the UNION clause If SQL_U_UNION_ALL is set, so is SQL_U_UNION.
SQL_USER_NAME [47]	string	The user name used in a particular database. This is the identifier specified on the `SQLConnect()` call.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 73. SQLGetInfo SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The requested information is returned as a string and its length exceeds the length of the application buffer as specified in *cbInfoValueMax*. The argument *pcbInfoValue* contains the actual (not truncated) length of the requested information. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection is closed.	The type of information requested in *fInfoType* requires an open connection. Only SQL_ODBC_VER does not require an open connection.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	The argument *rgbInfoValue* is a null pointer. The *fInfoType* is SQL_DRIVER_HSTMT and the value pointed to by *rgbInfoValue* is not a valid handle.
S1090	Invalid string or buffer length.	The value specified for argument *cbInfoValueMax* is less than 0.
S1096	Information type out of range.	An invalid *fInfoType* is specified.
S1C00	Driver not capable.	The value specified in the argument *fInfoType* is not supported by either DB2 CLI or the data source.

Restrictions

None.

Example

```
/* ... */
*/
    /* Check to see if SQLGetInfo() is supported */
    rc = SQLGetFunctions(hdbc, SQL_API_SQLGETINFO, &supported);

    if (supported == SQL_TRUE) { /* get information about current connection */

        rc = SQLGetInfo(hdbc, SQL_DATA_SOURCE_NAME, buffer, 255, &outlen);
        printf("      Server Name: %s\n", buffer);

        rc = SQLGetInfo(hdbc, SQL_DATABASE_NAME,    buffer, 255, &outlen);
        printf("    Database Name: %s\n", buffer);

        rc = SQLGetInfo(hdbc, SQL_SERVER_NAME,      buffer, 255, &outlen);
        printf("    Instance Name: %s\n", buffer);

        rc = SQLGetInfo(hdbc, SQL_DBMS_NAME,        buffer, 255, &outlen);
        printf("        DBMS Name: %s\n", buffer);

        rc = SQLGetInfo(hdbc, SQL_DBMS_VER,         buffer, 255, &outlen);
        printf("     DBMS Version: %s\n", buffer);

        rc = SQLGetInfo(hdbc, SQL_DRIVER_NAME,      buffer, 255, &outlen);
        printf("   CLI Driver Name: %s\n", buffer);

        rc = SQLGetInfo(hdbc, SQL_DRIVER_VER,       buffer, 255, &outlen);
        printf("CLI Driver Version: %s\n", buffer);

        rc = SQLGetInfo(hdbc, SQL_ODBC_SQL_CONFORMANCE, &output,
                        sizeof(output), &outlen);
        switch (output) {
        case 0:
            strcpy(buffer, "Minimum Grammar");
            break;
        case 1:
            strcpy(buffer, "Core Grammar");
            break;
        case 2:
            strcpy(buffer, "Extended Grammar");
            break;
        default:
            printf("Error calling getinfo!");
            return (SQL_ERROR);
        }
        printf("ODBC SQL Conformance Level: %s\n", buffer);
    } else {
        printf("SQLGetInfo is not supported!\n");
    }
/* ... */
```

References

- “SQLGetTypeInfo - Get Data Type Information” on page 238

SQLGetSQLCA - Get SQLCA Data Structure

Purpose

Specification:			

SQLGetSQLCA() is used to return the SQLCA associated with preparing and executing an SQL statement, fetching data, or closing a cursor. The SQLCA can return information that supplements the information obtained by using SQLError().

For a detailed description of the SQLCA structure, see Appendix C of *SQL Reference*.

An SQLCA is not available if a function is processed strictly on the application side, such as allocating a statement handle. In this case, an empty SQLCA is returned with all values set to zero.

Syntax

```
SQLRETURN  SQLGetSQLCA       (SQLHENV          henv,
                              SQLHDBC          hdbc,
                              SQLHSTMT         hstmt,
                              struct sqlca FAR  *pSqlca );
```

Function Arguments

Table 74. SQLGetSQLCA Arguments

Data Type	Argument	Use	Description
SQLHENV	henv	input	Environment Handle
SQLHDBC	hdbc	input	Connection Handle
SQLHSTMT	hstmt	input	Statement Handle
SQLCA *	pqlCA	output	SQL Communication Area

Usage

The handles are used in the same way as for the SQLError() function. To obtain the SQLCA associated with:

- An environment, pass a valid environment handle. Set *hdbc* and *hstmt* to SQL_NULL_HDBC and SQL_NULL_HSTMT respectively.
- A connection, pass a valid database connection handle, and set *hstmt* to SQL_NULL_HSTMT. The *henv* argument is ignored.
- A statement, pass a valid statement handle. The *henv* and *hdbc* arguments are ignored.

If diagnostic information generated by one DB2 CLI function is not retrieved before a function other than SQLError() is called with the same handle, the information for the previous function call is lost. This is true whether or not diagnostic information is generated for the second DB2 CLI function call.

If a DB2 CLI function is called that does not result in interaction with the DBMS, then the SQLCA contains all zeroes. Meaningful information is returned for the following functions:

- SQLCancel(),
- SQLConnect(), SQLDisconnect(),
- SQLExecDirect(), SQLExecute(),
- SQLFetch(),
- SQLPrepare(),
- SQLTransact()
- SQLColumns(),
- SQLConnect(),
- SQLSetConnectOption() (for SQL_AUTOCOMMIT),
- SQLStatistics(),
- SQLTables(),
- SQLColumnPrivileges(),
- SQLExtendedFetch(),
- SQLForeignKeys(),
- SQLMoreResults(),
- SQLPrimaryKeys(),
- SQLProcedureColumns(),
- SQLProcedures(),
- SQLTablePrivileges().

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

None.

Restrictions

None.

Example

```
 /******************************************************************/
 /*  DB2 for OS/390 Example:                                       */
 /*       Prepares a query and executes that query against a non- */
 /*       existent table. Then invoke SQLGetSQLCA to extract      */
 /*       native SQLCA data structure. Note that this API is NOT  */
 /*       defined within ODBC, i.e. this is unique to IBM CLI.    */
 /******************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include "sqlcli1.h"

void print_sqlca (SQLHENV,            // prototype for print_sqlca
                  SQLHDBC,
                  SQLHSTMT);

int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLHSTMT        hStmt   = SQL_NULL_HSTMT;
   SQLRETURN       rc      = SQL_SUCCESS;
   SQLINTEGER      RETCODE = 0;
   char            *pDSN = "STLEC1";
   SWORD           cbCursor;
   SDWORD          cbValue1;
   SDWORD          cbValue2;
   char            employee [30];
   int             salary = 0;
   int             param_salary = 30000;

   char            *stmt = "SELECT NAME, SALARY FROM EMPLOYEES WHERE SALARY > ?";


   (void) printf ("**** Entering CLIP11.\n\n");

  /******************************************************************/
  /* Allocate Environment Handle                                    */
  /******************************************************************/

   RETCODE = SQLAllocEnv(&hEnv);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /******************************************************************/
  /* Allocate Connection Handle to DSN                              */
  /******************************************************************/

   RETCODE = SQLAllocConnect(hEnv,
                             &hDbc);

   if( RETCODE != SQL_SUCCESS )      // Could not get a Connect Handle
     goto dberror;
```

```
/*****************************************************************/
/* CONNECT TO data source (STLEC1)                               */
/*****************************************************************/

 RETCODE = SQLConnect(hDbc,        // Connect handle
                      (SQLCHAR *) pDSN, // DSN
                      SQL_NTS,     // DSN is nul-terminated
                      NULL,        // Null UID
                      0   ,
                      NULL,        // Null Auth string
                      0);

 if( RETCODE != SQL_SUCCESS )      // Connect failed
   goto dberror;

/*****************************************************************/
/* Allocate Statement Handles                                    */
/*****************************************************************/

rc = SQLAllocStmt (hDbc,
                   &hStmt);

if (rc != SQL_SUCCESS)
  goto exit;

/*****************************************************************/
/* Prepare the query for multiple execution within current       */
/* transaction. Note that query is collapsed when transaction    */
/* is committed or rolled back.                                  */
/*****************************************************************/

rc = SQLPrepare (hStmt,
                 (SQLCHAR *) stmt,
                 strlen(stmt));

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** PREPARE OF QUERY FAILED.\n");
  (void) print_sqlca (hStmt,
                      hDbc,
                      hEnv);
  goto dberror;
}

rc = SQLBindCol (hStmt,           // bind employee name
                 1,
                 SQL_C_CHAR,
                 employee,
                 sizeof(employee),
                 &cbValue1);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** BIND OF NAME FAILED.\n");
  goto dberror;
}
```

```
rc = SQLBindCol (hStmt,           // bind employee salary
                 2,
                 SQL_C_LONG,
                 &salary,
                 0,
                 &cbValue2);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** BIND OF SALARY FAILED.\n");
  goto dberror;
}

/*****************************************************************/
/* Bind parameter to replace '?' in query. This has an initial   */
/* value of 30000.                                               */
/*****************************************************************/

rc = SQLBindParameter (hStmt,
                       1,
                       SQL_PARAM_INPUT,
                       SQL_C_LONG,
                       SQL_INTEGER,
                       0,
                       0,
                       &param_salary,
                       0,
                       NULL);

/*****************************************************************/
/* Execute prepared statement to generate answer set.            */
/*****************************************************************/

rc = SQLExecute (hStmt);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** EXECUTE OF QUERY FAILED.\n");
  (void) print_sqlca (hStmt,
                      hDbc,
                      hEnv);
  goto dberror;
}

/*****************************************************************/
/* Answer Set is available -- Fetch rows and print employees     */
/* and salary.                                                   */
/*****************************************************************/

(void) printf ("**** Employees whose salary exceeds %d follow.\n\n",
               param_salary);

while ((rc = SQLFetch (hStmt)) == SQL_SUCCESS)
{
  (void) printf ("**** Employee Name %s with salary %d.\n",
                 employee,
                 salary);
}
```

```
  /*****************************************************************/
  /* Deallocate Statement Handles -- statement is no longer in a   */
  /* Prepared state.                                               */
  /*****************************************************************/

  rc = SQLFreeStmt (hStmt,
                    SQL_DROP);

  /*****************************************************************/
  /* DISCONNECT from data source                                   */
  /*****************************************************************/

   RETCODE = SQLDisconnect(hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Deallocate Connection Handle                                  */
  /*****************************************************************/

   RETCODE = SQLFreeConnect (hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Free Environment Handle                                       */
  /*****************************************************************/

   RETCODE = SQLFreeEnv (hEnv);

   if (RETCODE == SQL_SUCCESS)
     goto exit;

   dberror:
   RETCODE=12;

   exit:

   (void) printf ("**** Exiting  CLIP11.\n\n");

   return RETCODE;
}

/*****************************************************************/
/* print_sqlca invokes SQLGetSQLCA and prints the native SQLCA.  */
/*****************************************************************/

void print_sqlca (SQLHENV  hEnv ,
                  SQLHDBC  hDbc ,
                  SQLHSTMT hStmt)
{
  SQLRETURN       rc      = SQL_SUCCESS;
  struct sqlca    sqlca;
  struct sqlca    *pSQLCA = &sqlca;
  int  code ;
  char state [6];
  char errp  [9];
  char tok   [40];
  int  count, len, start, end, i;
  if ((rc = SQLGetSQLCA (hEnv ,
                         hDbc ,
                         hStmt,
                         pSQLCA)) != SQL_SUCCESS)
  {
    (void) printf ("**** SQLGetSQLCA failed Return Code = %d.\n", rc);
    goto exit;
  }

  code = (int) pSQLCA->sqlcode;
  memcpy (state, pSQLCA->sqlstate, 5);
  state [5] = '\0';

  (void) printf ("**** sqlcode = %d, sqlstate = %s.\n", code, state);

  memcpy (errp, pSQLCA->sqlerrp, 8);
  errp [8] = '\0';
  (void) printf ("**** sqlerrp = %s.\n", errp);

  if (pSQLCA->sqlerrml == 0)
    (void) printf ("**** No tokens.\n");
  else
  {
    for (len = 0, count = 0; len < pSQLCA->sqlerrml; len = ++end)
    {
      start = end = len;
      while ((pSQLCA->sqlerrmc [end] != 0XFF) &&;
             (end < pSQLCA->sqlerrml))

        end++;
      if (start != end)
      {
        memcpy (tok, &pSQLCA->sqlerrmc[start],
                                                       (end-start));
        tok [end-start+1] = '\0';
        (void) printf ("**** Token # %d = %s.\n", count++, tok);
      }
    }
  }

  for (i = 0; i <= 5; i++)
    (void) printf ("**** sqlerrd # %d = %d.\n", i+1, pSQLCA->sqlerrd_i•);

  for (i = 0; i <= 10; i++)
    (void) printf ("**** sqwarn # %d = %c.\n", i+1, pSQLCA->sqlwarn_i•);

  exit:
  return;
}
```

References

- “SQLError - Retrieve Error Information” on page 143

SQLGetStmtOption - Returns Current Setting of A Statement Option

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	

SQLGetStmtOption() returns the current settings of the specified statement option.

These options are set using the SQLSetStmtOption() function.

Syntax

```
SQLRETURN   SQLGetStmtOption (SQLHSTMT         hstmt,
                              SQLUSMALLINT     fOption,
                              SQLPOINTER       pvParam);
```

Function Arguments

Table 75. SQLStmtOption Arguments

Data Type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle.
SQLUSMALLINT	fOption	input	Option to set. Refer to Table 123 on page 316 for the list of statement options and their descriptions.
SQLPOINTER	pvParam	output	Value of the option. Depending on the value of *fOption* this can be a 32-bit integer value, or a pointer to a null terminated character string. The maximum length of any character string returned is SQL_MAX_OPTION_STRING_LENGTH bytes (excluding the null-terminator).

Usage

See Table 123 on page 316 in the function description of SQLSetStmtOption() for a list of statement options. The following table lists the statement options that are read-only (can be read but not set).

Table 76. Statement Options

fOption	Contents
SQL_ROW_NUMBER	A 32-bit integer value that specifies the number of the current row in the entire result set. If the number of the current row cannot be determined or there is no current row, 0 is returned.

Note: ODBC also defines the read-only statement option SQL_GET_BOOKMARK. This option is not supported by Call Level Interface. If it is specified, this function returns SQL_ERROR (SQLSTATE **S1**011 -- Operation invalid at this time.)

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 77. SQLGetStmtOption SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	There is no open cursor on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	*pvParam* was null.
S1010	Function sequence error.	The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1092	Option type out of range.	An invalid *fOption* value was specified.
S1C00	Driver not capable.	DB2 CLI recognizes the option but does not support it.

Restrictions

None.

Example

```
/* ... */
    rc = SQLGetStmtOption(hstmt, SQL_CURSOR_HOLD, &cursor_hold);
    printf("Cursor With Hold is: ");
    if (cursor_hold == SQL_CURSOR_HOLD_ON )
       printf("ON\n");
    else
       printf("OFF\n");
/* ... */
```

References

- “SQLSetConnectOption - Set Connection Option” on page 298
- “SQLSetStmtOption - Set Statement Option” on page 315

SQLGetTypeInfo - Get Data Type Information

Purpose

Specification:	**ODBC** 1.0	**X/OPEN CLI**	**ISO CLI**

`SQLGetTypeInfo()` returns information about the data types that are supported by the DBMSs associated with DB2 CLI. The information is returned in an SQL result set. The columns can be received using the same functions that are used to process a query.

Syntax

```
SQLRETURN   SQLGetTypeInfo   (SQLHSTMT         hstmt,
                              SQLSMALLINT      fSqlType);
```

Function Arguments

Table 78. SQLGetTypeInfo Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle.
SQLSMALLINT	*fSqlType*	input	The SQL data type being queried. The supported types are: • SQL_ALL_TYPES • SQL_BINARY • SQL_CHAR • SQL_DATE • SQL_DECIMAL • SQL_DOUBLE • SQL_FLOAT • SQL_GRAPHIC • SQL_INTEGER • SQL_LONGVARBINARY • SQL_LONGVARCHAR • SQL_LONGVARGRAPHIC • SQL_NUMERIC • SQL_REAL • SQL_SMALLINT • SQL_TIME • SQL_TIMESTAMP • SQL_VARBINARY • SQL_VARCHAR • SQL_VARGRAPHIC If SQL_ALL_TYPES is specified, information about all supported data types is returned in ascending order by TYPE_NAME. All unsupported data types are absent from the result set.

Usage

Since `SQLGetTypeInfo()` generates a result set and is equivalent to executing a query, it generates a cursor and begins a transaction. To prepare and execute another statement on this statement handle, the cursor must be closed.

If `SQLGetTypeInfo()` is called with an invalid *fSqlType*, an empty result set is returned.

The columns of the result set generated by this function are described below.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change. The data types returned are those that can be used in a CREATE TABLE, ALTER TABLE, DDL statement. Non-persistent data types such as the locator data types are not part of the returned result set. User defined data types are not returned either.

Table 79 (Page 1 of 2). Columns Returned By SQLGetTypeInfo

Column Number/Name	Data Type	Description
1 TYPE_NAME	VARCHAR(128) NOT NULL	Character representation of the SQL data type name. For example, VARCHAR, DATE, INTEGER.
2 DATA_TYPE	SMALLINT NOT NULL	SQL data type define values. For example, SQL_VARCHAR, SQL_DATE, SQL_INTEGER.
3 COLUMN_SIZE	INTEGER	If the data type is a character or binary string, then this column contains the maximum length in bytes; if it is a graphic (DBCS) string, this is the number of double byte characters for the column. For date, time, timestamp data types, this is the total number of characters required to display the value when converted to character. For numeric data types, this is the total number of digits.
4 LITERAL_PREFIX	VARCHAR(128)	Character that DB2 recognizes as a prefix for a literal of this data type. This column is null for data types where a literal prefix is not applicable.
5 LITERAL_SUFFIX	VARCHAR(128)	Character that DB2 recognizes as a suffix for a literal of this data type. This column is null for data types where a literal prefix is not applicable.
6 CREATE_PARAMS	VARCHAR(128)	The text of this column contains a list of keywords, separated by commas, that correspond to each parameter the application can specify in parenthesis when using the name in the TYPE_NAME column as a data type in SQL. The keywords in the list can be any of the following: LENGTH, PRECISION, SCALE. They appear in the order that the SQL syntax requires that they be used. A NULL indicator is returned if there are no parameters for the data type definition, (such as INTEGER). **Note:** The intent of CREATE_PARAMS is to enable an application to customize the interface for a *DDL builder*. An application should expect, using this, only to be able to determine the number of arguments required to define the data type and to have localized text that could be used to label an edit control.

Table 79 (Page 2 of 2). Columns Returned By SQLGetTypeInfo

Column Number/Name	Data Type	Description
7 NULLABLE	SMALLINT NOT NULL	Indicates whether the data type accepts a NULL value • Set to SQL_NO_NULLS if NULL values are disallowed. • Set to SQL_NULLABLE if NULL values are allowed.
8 CASE_SENSITIVE	SMALLINT NOT NULL	Indicates whether the data type can be treated as case sensitive for collation purposes; valid values are SQL_TRUE and SQL_FALSE.
9 SEARCHABLE	SMALLINT NOT NULL	Indicates how the data type is used in a WHERE clause. Valid values are: • SQL_UNSEARCHABLE : if the data type cannot be used in a WHERE clause. • SQL_LIKE_ONLY : if the data type can be used in a WHERE clause only with the LIKE predicate. • SQL_ALL_EXCEPT_LIKE : if the data type can be used in a WHERE clause with all comparison operators except LIKE. • SQL_SEARCHABLE : if the data type can be used in a WHERE clause with any comparison operator.
10 UNSIGNED_ATTRIBUTE	SMALLINT	Indicates whether the data type is unsigned. The valid values are: SQL_TRUE, SQL_FALSE or NULL. A NULL indicator is returned if this attribute is not applicable to the data type.
11 FIXED_PREC_SCALE	SMALLINT NOT NULL	Contains the value SQL_TRUE if the data type is exact numeric and always has the same precision and scale; otherwise, it contains SQL_FALSE.
12 AUTO_INCREMENT	SMALLINT	Contains SQL_TRUE if a column of this data type is automatically set to a unique value when a row is inserted; otherwise, contains SQL_FALSE.
13 LOCAL_TYPE_NAME	VARCHAR(128)	This column contains any localized (native language) name for the data type that is different from the regular name of the data type. If there is no localized name, this column is NULL. This column is intended for display only. The character set of the string is locale-dependent and is typically the default character set of the database.
14 MINIMUM_SCALE	SMALLINT	The minimum scale of the SQL data type. If a data type has a fixed scale, the MINIMUM_SCALE and MAXIMUM_SCALE columns both contain the same value. NULL is returned where scale is not applicable.
15 MAXIMUM_SCALE	SMALLINT	The maximum scale of the SQL data type. NULL is returned where scale is not applicable. If the maximum scale is not defined separately in the DBMS, but is defined instead to be the same as the maximum length of the column, then this column contains the same value as the COLUMN_SIZE column.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 80. SQLGetTypeInfo SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle. *hstmt* is not closed.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1004	SQL data type out of range.	An invalid *fSqlType* is specified.
S1010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.

Restrictions

The following ODBC specified SQL data types (and their corresponding *fSqlType* define values) are not supported by any IBM RDBMS:

Data Type	***fSqlType***
TINY INT	SQL_TINYINT
BIG INT	SQL_BIGINT
BIT	SQL_BIT

Example

```
 /******************************************************************/
 /*  DB2 for OS/390 Example:                                       */
 /*        Invokes SQLGetTypeInfo to retrieve SQL data types sup-  */
 /*        ported.                                                 */
 /******************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include "sqlcli1.h"

  /******************************************************************/
  /* Invoke SQLGetTypeInfo to retrieve all SQL data types supported */
  /* by Data Source.                                                */
  /******************************************************************/

int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLHSTMT        hStmt   = SQL_NULL_HSTMT;
   SQLRETURN       rc      = SQL_SUCCESS;
   SQLINTEGER      RETCODE = 0;

   (void) printf ("**** Entering CLIP06.\n\n");

  /*****************************************************************/
  /* Allocate Environment Handle                                   */
  /*****************************************************************/

   RETCODE = SQLAllocEnv(&hEnv);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Allocate Connection Handle to DSN                             */
  /*****************************************************************/

   RETCODE = SQLAllocConnect(hEnv,
                             &hDbc);

   if( RETCODE != SQL_SUCCESS )      // Could not get a Connect Handle
     goto dberror;

  /*****************************************************************/
  /* CONNECT TO data source (STLEC1)                               */
  /*****************************************************************/

   RETCODE = SQLConnect(hDbc,        // Connect handle
                        (SQLCHAR *) "STLEC1", // DSN
                        SQL_NTS,     // DSN is null-terminated
                        NULL,        // Null UID
                        0   ,
                        NULL,        // Null Auth string
                        0);

   if( RETCODE != SQL_SUCCESS )      // Connect failed
     goto dberror;

  /*****************************************************************/
  /* Retrieve SQL data types from DSN                              */
  /*****************************************************************/
  // local variables to Bind to retrieve TYPE_NAME, DATA_TYPE,
  // COLUMN_SIZE and NULLABLE

   struct                    // TYPE_NAME is VARCHAR(128)
   {
     SQLSMALLINT  length;
     SQLCHAR      name [128];
     SQLINTEGER   ind;
   } typename;

   SQLSMALLINT data_type;    // DATA_TYPE is SMALLINT
   SQLINTEGER  data_type_ind;
   SQLINTEGER  column_size;  // COLUMN_SIZE is integer
   SQLINTEGER  column_size_ind;
   SQLSMALLINT nullable;     // NULLABLE is SMALLINT
   SQLINTEGER  nullable_ind;
```

```
/*****************************************************************/
/* Allocate Statement Handle                                     */
/*****************************************************************/

 rc = SQLAllocStmt (hDbc,
                    &hStmt);

 if (rc != SQL_SUCCESS)
   goto exit;

/*****************************************************************/
/*                                                               */
/* Retrieve native SQL types from DSN ------------>              */
/*                                                               */
/*  The result set consists of 15 columns. We only bind          */
/*  TYPE_NAME, DATA_TYPE, COLUMN_SIZE and NULLABLE. Note: Need   */
/*  not bind all columns of result set -- only those required.   */
/*                                                               */
/*****************************************************************/

 rc = SQLGetTypeInfo (hStmt,
                      SQL_ALL_TYPES);

 if (rc != SQL_SUCCESS)
   goto exit;

 rc = SQLBindCol (hStmt,          // bind TYPE_NAME
                  1,
                  SQL_CHAR,
                  (SQLPOINTER) typename.name,
                  128,
                  &typename.ind);

 if (rc != SQL_SUCCESS)
   goto exit;

 rc = SQLBindCol (hStmt,          // bind DATA_NAME
                  2,
                  SQL_C_DEFAULT,
                  (SQLPOINTER) &data_type,
                  sizeof(data_type),
                  &data_type_ind);

 if (rc != SQL_SUCCESS)
   goto exit;

 rc = SQLBindCol (hStmt,          // bind COLUMN_SIZE
                  3,
                  SQL_C_DEFAULT,
                  (SQLPOINTER) &column_size,
                  sizeof(column_size),
                  &column_size_ind);

 if (rc != SQL_SUCCESS)
   goto exit;

 rc = SQLBindCol (hStmt,          // bind NULLABLE
                  7,
                  SQL_C_DEFAULT,
                  (SQLPOINTER) &nullable,
                  sizeof(nullable),
                  &nullable_ind);
 if (rc != SQL_SUCCESS)
   goto exit;

/*****************************************************************/
/* Fetch all native DSN SQL Types and print Type Name, Type,     */
/* Precision and nullability.                                    */
/*****************************************************************/

 while ((rc = SQLFetch (hStmt)) == SQL_SUCCESS)
 {
   (void) printf ("**** Type Name is %s. Type is %d. Precision is %d.",
                  typename.name,
                  data_type,
                  column_size);
   if (nullable == SQL_NULLABLE)
     (void) printf (" Type is nullable.\n");
   else
     (void) printf (" Type is not nullable.\n");
 }

 if (rc == SQL_NO_DATA_FOUND)   // if result set exhausted reset
   rc = SQL_SUCCESS;            // rc to OK

/*****************************************************************/
/* Free Statement handle.                                        */
/*****************************************************************/

 rc = SQLFreeStmt (hStmt,
                   SQL_DROP);


 if (RETCODE != SQL_SUCCESS)       // An advertised API failed
   goto dberror;

/*****************************************************************/
/* DISCONNECT from data source                                   */
/*****************************************************************/

 RETCODE = SQLDisconnect(hDbc);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;

/*****************************************************************/
/* Deallocate Connection Handle                                  */
/*****************************************************************/

 RETCODE = SQLFreeConnect (hDbc);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;

/*****************************************************************/
/* Free Environment Handle                                       */
/*****************************************************************/

 RETCODE = SQLFreeEnv (hEnv);

 if (RETCODE == SQL_SUCCESS)
   goto exit;
   dberror:
   RETCODE=12;

   exit:

   (void) printf ("**** Exiting  CLIP06.\n\n");

   return(RETCODE);
}
```

References

- “SQLColAttributes - Get Column Attributes” on page 104
- “SQLExtendedFetch - Extended Fetch (Fetch Array of Rows)” on page 157
- “SQLGetInfo - Get General Information” on page 213

SQLMoreResults - Determine If There Are More Result Sets

Purpose

Specification:	**ODBC** 1.0		

SQLMoreResults() determines whether there is more information available on the statement handle which has been associated with:

- Array input of parameter values for a query, or
- A stored procedure that is returning result sets.

Syntax

```
SQLRETURN   SQLMoreResults   (SQLHSTMT         hstmt);
```

Function Arguments

Table 81. SQLMoreResults Arguments

Data Type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle.

Usage

This function is used to return multiple results set in a sequential manner upon the execution of:

- A parameterized query with an array of input parameter values specified with SQLParamOptions() and SQLBindParameter(), or
- A stored procedure containing SQL queries, the cursors of which have been left open so that the result sets remain accessible when the stored procedure has finished execution.

Refer to "Using Arrays to Input Parameter Values" on page 353 and "Returning Result Sets From Stored Procedures" on page 363 for more information.

After completely processing the first result set, the application can call SQLMoreResults() to determine if another result set is available. If the current result set has unfetched rows, SQLMoreResults() discards them by closing the cursor and, if another result set is available, returns SQL_SUCCESS.

If all the result sets have been processed, SQLMoreResults() returns SQL_NO_DATA_FOUND.

If SQLFreeStmt() is called with the SQL_CLOSE or SQL_DROP option, all pending result sets on this statement handle are discarded.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NO_DATA_FOUND

Diagnostics

Table 82. SQLMoreResults SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.

In addition `SQLMoreResults()` can return the SQLSTATEs associated with `SQLExecute()`.

Restrictions

The ODBC specification of `SQLMoreResults()` also allows counts associated with the execution of parameterized INSERT, UPDATE, and DELETE statements with arrays of input parameter values to be returned. However, DB2 CLI does not support the return of such count information.

Example

```
/* ... */
#define NUM_CUSTOMERS 25
    SQLCHAR         stmt[] =
    {  "WITH "  /* Common Table expression (or Define Inline View) */
         "order (ord_num, cust_num, prod_num, quantity, amount) AS "
         "( "
         "SELECT c.ord_num, c.cust_num, l.prod_num, l.quantity,  "
                 "price(char(p.price, '.'), p.units, char(l.quantity, '.')) "
            "FROM ord_cust c, ord_line l, product p  "
            "WHERE c.ord_num = l.ord_num AND l.prod_num = p.prod_num  "
             "AND cust_num = CNUM(cast (? as integer)) "
         "), "
         "totals (ord_num, total) AS "
         "( "
          "SELECT ord_num, sum(decimal(amount, 10, 2))  "
          "FROM order GROUP BY ord_num "
         ") "
```

```
       /* The 'actual' SELECT from the inline view */
       "SELECT order.ord_num, cust_num, prod_num, quantity,  "
              "DECIMAL(amount,10,2) amount, total "
        "FROM order, totals  "
        "WHERE order.ord_num = totals.ord_num "
     };
    /* Array of customers to get list of all orders for */
    SQLINTEGER     Cust[]=
    {
        10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
        110, 120, 130, 140, 150, 160, 170, 180, 190, 200,
        210, 220, 230, 240, 250
    };

#define  NUM_CUSTOMERS sizeof(Cust)/sizeof(SQLINTEGER)

    /* Row-Wise (Includes buffer for both column data and length) */
    struct {
        SQLINTEGER      Ord_Num_L;
        SQLINTEGER      Ord_Num;
        SQLINTEGER      Cust_Num_L;
        SQLINTEGER      Cust_Num;
        SQLINTEGER      Prod_Num_L;
        SQLINTEGER      Prod_Num;
        SQLINTEGER      Quant_L;
        SQLDOUBLE       Quant;
        SQLINTEGER      Amount_L;
        SQLDOUBLE       Amount;
        SQLINTEGER      Total_L;
        SQLDOUBLE       Total;
    }               Ord[ROWSET_SIZE];

    SQLUINTEGER     pirow = 0;
    SQLUINTEGER     pcrow;
    SQLINTEGER      i;
    SQLINTEGER      j;
/* ... */
    /* Get details and total for each order Row-Wise */
    rc = SQLAllocStmt(hdbc, &hstmt);

    rc = SQLParamOptions(hstmt, NUM_CUSTOMERS, &pirow);

    rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_LONG, SQL_INTEGER,
                          0, 0, Cust, 0, NULL);

    rc = SQLExecDirect(hstmt, stmt, SQL_NTS);

    /* SQL_ROWSET_SIZE sets the max number of result rows to fetch each time */
    rc = SQLSetStmtOption(hstmt, SQL_ROWSET_SIZE, ROWSET_SIZE);

    /* Set Size of One row, Used for Row-Wise Binding Only */
    rc = SQLSetStmtOption(hstmt, SQL_BIND_TYPE, sizeof(Ord) / ROWSET_SIZE);

    /* Bind column 1 to the Ord_num Field of the first row in the array*/
    rc = SQLBindCol(hstmt, 1, SQL_C_LONG, (SQLPOINTER) &Ord[0].Ord_Num, 0,
                    &Ord[0].Ord_Num_L);

    /* Bind remaining columns ... */
/* ... */
```

```
    /* NOTE: This sample assumes that an order never has more
             rows than ROWSET_SIZE.  A check should be added below to call
             SQLExtendedFetch multiple times for each result set.
    */
    do  /* for each result set .... */
    { rc = SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 0, &pcrow, NULL);

      if (pcrow > 0) /* if 1 or more rows in the result set */
      {
        i = j = 0;

        printf("*************************************\n");
        printf("Orders for Customer: %ld\n", Ord[0].Cust_Num);
        printf("*************************************\n");

        while (i < pcrow)
        {   printf("\nOrder #: %ld\n", Ord[i].Ord_Num);
            printf("     Product  Quantity         Price\n");
            printf("     -------- ---------------- ------------\n");
            j = i;
            while (Ord[j].Ord_Num == Ord[i].Ord_Num)
            {   printf("    %8ld %16.7lf %12.2lf\n",
                       Ord[i].Prod_Num, Ord[i].Quant, Ord[i].Amount);
                i++;
            }
            printf("                               ============\n");
            printf("                              %12.2lf\n", Ord[j].Total);
        } /* end while */
      } /* end if */
    }
    while ( SQLMoreResults(hstmt) == SQL_SUCCESS);
/* ... */
```

References

- “SQLParamOptions - Specify an Input Array for a Parameter” on page 259

SQLNativeSql - Get Native SQL Text

Purpose

Specification:	**ODBC** 1.0		

SQLNativeSql() is used to show how DB2 CLI interprets vendor escape clauses. If the original SQL string passed in by the application contains vendor escape clause sequences, then DB2 CLI returns the transformed SQL string that the data source sees (with vendor escape clauses either converted or discarded, as appropriate).

Syntax

```
SQLRETURN   SQLNativeSql     (SQLHDBC          hdbc,
                              SQLCHAR      FAR  *szSqlStrIn,
                              SQLINTEGER        cbSqlStrIn,
                              SQLCHAR      FAR  *szSqlStr,
                              SQLINTEGER        cbSqlStrMax,
                              SQLINTEGER   FAR  *pcbSqlStr);
```

Function Arguments

Table 83. SQLNativeSql Arguments

Data Type	**Argument**	**Use**	**Description**
SQLHDBC	hdbc	input	Connection handle.
SQLCHAR *	szSqlStrIn	input	Input SQL string.
SQLINTEGER	cbSqlStrIn	input	Length of *szSqlStrIn*.
SQLCHAR *	szSqlStr	output	Pointer to buffer for the transformed output string.
SQLINTEGER	cbSqlStrMax	input	Size of buffer pointed by *szSqlStr*.
SQLINTEGER *	pcbSqlStr	output	The total number of bytes (excluding the null-terminator) available to return in *szSqlStr*. If the number of bytes available to return is greater than or equal to *cbSqlStrMax*, the output SQL string in *szSqlStr* is truncated to *cbSqlStrMax - 1* bytes.

Usage

This function is called when the application wishes to examine or display the transformed SQL string that is passed to the data source by DB2 CLI. Translation (mapping) only occurs if the input SQL statement string contains vendor escape clause sequences. For more information on vendor escape clause sequences, refer to “Using Vendor Escape Clauses” on page 376.

DB2 CLI can only detect vendor escape clause syntax errors; since DB2 CLI does not pass the transformed SQL string to the data source for preparation, syntax errors that are detected by the DBMS are not generated at this time. (The statement is not passed to the data source for preparation because the preparation can potentially cause the initiation of a transaction.)

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 84. SQLNativeSql SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The buffer *szSqlStr* is not large enough to contain the entire SQL string, so truncation occurs. The argument *pcbSqlStr* contains the total length of the untruncated SQL string. (Function returns with SQL_SUCCESS_WITH_INFO)
08003	Connection is closed.	The *hdbc* does not reference an open database connection.
37000	Invalid SQL syntax.	The input SQL string in *szSqlStrIn* contained a syntax error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	The argument *szSqlStrIn* is a NULL pointer. The argument *szSqlStr* is a NULL pointer.
S1090	Invalid string or buffer length.	The argument *cbSqlStrIn* was less than 0, but not equal to SQL_NTS. The argument *cbSqlStrMax* was less than 0.

Restrictions

None.

Example

```
/* ... */
    SQLCHAR         in_stmt[1024];
    SQLCHAR         out_stmt[1024];
    SQLSMALLINT     pcPar;
    SQLINTEGER      indicator;
/* ... */
    /* Prompt for a statement to prepare */
    printf("Enter an SQL statement: \n");
    gets(in_stmt);

    /* prepare the statement */
    rc = SQLPrepare(hstmt, in_stmt, SQL_NTS);

    SQLNumParams(hstmt, &pcPar);

    SQLNativeSql(hstmt, in_stmt, SQL_NTS, out_stmt, 1024, &indicator);

    if (indicator == SQL_NULL_DATA)
    {  printf("Invalid statement\n"); }
    else
    {  printf(" Input Statement: \n %s \n", in_stmt);
       printf("Output Statement: \n %s \n", out_stmt);
       printf("Number of Parameter Markers = %ld\n", pcPar);
    }
    rc = SQLFreeStmt(hstmt, SQL_DROP);
/* ... */
```

References

- “Using Vendor Escape Clauses” on page 376

SQLNumParams - Get Number of Parameters in A SQL Statement

Purpose

Specification:	**ODBC** 1.0		

`SQLNumParams()` returns the number of parameter markers in a SQL statement.

Syntax

```
SQLRETURN   SQLNumParams     (SQLHSTMT         hstmt,
                              SQLSMALLINT  FAR  *pcpar);
```

Function Arguments

Table 85. SQLNumParams Arguments

Data Type	Argument	Use	Description
SQLHSTMT	hstmt	Input	Statement handle.
SQLSMALLINT *	pcpar	Output	Number of parameters in the statement.

Usage

This function can only be called after the statement associated with *hstmt* has been prepared. If the statement does not contain any parameter markers, *pcpar* is set to 0.

An application can call this function to determine how many `SQLBindParameter()` calls are necessary for the SQL statement associated with the statement handle.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 86 (Page 1 of 2). SQLNumParams SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	*pcpar* is null.
S1010	Function sequence error.	This function is called before `SQLPrepare()` is called for the specified *hstmt*. The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.

Table 86 (Page 2 of 2). SQLNumParams SQLSTATEs

SQLSTATE	Description	Explanation
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.

Restrictions

None.

Example

Refer to “Example” on page 252.

References

- “SQLBindParameter - Binds A Parameter Marker to a Buffer” on page 91
- “SQLPrepare - Prepare a Statement” on page 261

SQLNumResultCols - Get Number of Result Columns

Purpose

Specification:	**ODBC** 1.0	**X/OPEN CLI**	**ISO CLI**

`SQLNumResultCols()` returns the number of columns in the result set associated with the input statement handle.

`SQLPrepare()` or `SQLExecDirect()` must be called before calling this function.

After calling this function, you can call `SQLColAttributes()`, or one of the bind column functions.

Syntax

```
SQLRETURN   SQLNumResultCols (SQLHSTMT         hstmt,
                              SQLSMALLINT FAR   *pccol);
```

Function Arguments

Table 87. SQLNumResultCols Arguments

Data Type	**Argument**	**Use**	**Description**
SQLHSTMT	*hstmt*	input	Statement handle
SQLSMALLINT *	*pccol*	output	Number of columns in the result set

Usage

The function sets the output argument to zero if the last statement or function executed on the input statement handle did not generate a result set.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 88 (Page 1 of 2). SQLNumResultCols SQLSTATEs

SQLSTATE	**Description**	**Explanation**
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	*pcccol* is a null pointer.

Table 88 (Page 2 of 2). SQLNumResultCols SQLSTATEs

SQLSTATE	Description	Explanation
S1010	Function sequence error.	The function is called prior to calling `SQLPrepare()` or `SQLExecDirect()` for the *hstmt*. The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.

Restrictions

None.

Example

Refer to "Example" on page 130

References

- "SQLColAttributes - Get Column Attributes" on page 104
- "SQLDescribeCol - Describe Column Attributes" on page 128
- "SQLExecDirect - Execute a Statement Directly" on page 149
- "SQLGetData - Get Data From a Column" on page 193
- "SQLPrepare - Prepare a Statement" on page 261

SQLParamData - Get Next Parameter For Which A Data Value Is Needed

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLParamData() is used in conjunction with SQLPutData() to send long data in pieces. It can also be used to send fixed length data as well. For a description of the exact sequence of this input method, refer to "Sending/Retrieving Long Data in Pieces" on page 351.

Syntax

```
SQLRETURN   SQLParamData     (SQLHSTMT         hstmt,
                               SQLPOINTER  FAR   *prgbValue);
```

Function Arguments

Table 89. SQLParamData Arguments

Data Type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle.
SQLPOINTER *	prgbValue	output	Pointer to the value of the *rgbValue* argument specified on the SQLBindParameter() or SQLSetParam() call.

Usage

SQLParamData() returns SQL_NEED_DATA if there is at least one SQL_DATA_AT_EXEC parameter for which data is not assigned. This function returns an application provided value in *prgbValue* supplied by the application during the previous SQLBindParameter() call. SQLPutData() is called one or more times (in the case of long data) to send the parameter data. SQLParamData() is called to signal that all the data has been sent for the current parameter and to advance to the next SQL_DATA_AT_EXEC parameter. SQL_SUCCESS is returned when all the parameters have been assigned data values and the associated statement has been executed successfully. If any errors occur during or before actual statement execution, SQL_ERROR is returned.

If SQLParamData() returns SQL_NEED_DATA, then only SQLPutData() or SQLCancel() calls can be made. All other function calls using this statement handle fails. In addition, all function calls referencing the parent *hdbc* of *hstmt* fail if they involve changing any attribute or state of that connection; that is, the following function calls on the parent *hdbc* are also not permitted:

- SQLAllocConnect()
- SQLAllocStmt()
- SQLSetConnectOption()
- SQLNativeSql()
- SQLTransact()

Should they be invoked during an SQL_NEED_DATA sequence, these functions return SQL_ERROR with SQLSTATE of **S1**010 and the processing of the SQL_DATA_AT_EXEC parameters is not affected.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_NEED_DATA

Diagnostics

SQLParamData() can return any SQLSTATE returned by the SQLExecDirect() and SQLExecute() functions. In addition, the following diagnostics can also be generated:

Table 90. SQLParamData SQLSTATEs

SQLSTATE	Description	Explanation
40001	Transaction rollback.	The transaction to which this SQL statement belonged is rolled back due to a deadlock or timeout.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	SQLParamData() is called out of sequence. This call is only valid after an SQLExecDirect() or an SQLExecute(), or after an SQLPutData() call. Even though this function is called after an SQLExecDirect() or an SQLExecute() call, there are no SQL_DATA_AT_EXEC parameters (left) to process.

Restrictions

None.

Example

Refer to “Example” on page 289.

References

- “SQLBindParameter - Binds A Parameter Marker to a Buffer” on page 91
- “SQLCancel - Cancel Statement” on page 102
- “SQLExecDirect - Execute a Statement Directly” on page 149
- “SQLExecDirect - Execute a Statement Directly” on page 149
- “SQLPutData - Passing Data Value for A Parameter” on page 287
- “SQLSetParam - Binds A Parameter Marker to a Buffer” on page 310

SQLParamOptions - Specify an Input Array for a Parameter

Purpose

Specification:	**ODBC** 1.0		

SQLParamOptions() provides the ability to set multiple values for each parameter set by SQLBindParameter(). This allows the application to perform batched processing of the same SQL statement with one set of prepare, execute and SQLBindParameter() calls.

Syntax

```
SQLRETURN   SQLParamOptions  (SQLHSTMT         hstmt,
                              SQLUINTEGER      crow,
                              SQLUINTEGER  FAR  *pirow);
```

Function Arguments

Table 91. SQLParamOptions Arguments

Data Type	Argument	Use	Description
SQLHSTMT	hstmt	Input	Statement handle.
SQLUINTEGER	crow	Input	Number of values for each parameter. If this is greater than 1, then the *rgbValue* argument in SQLBindParameter() points to an array of parameter values, and *pcbValue* points to an array of lengths.
SQLUINTEGER *	pirow	Output (deferred)	Pointer to the buffer for the current parameter array index. As each set of parameter values is processed, *pirow* is set to the array index of that set. If a statement fails, *pirow* can be used to determine how many statements were successfully processed. Nothing is returned if the *pirow* pointer is NULL.

Usage

DB2 CLI prepares the statement, and executes it repeatedly for the array of parameter markers.

As a statement executes, *pirow* is set to the index of the current array of parameter values. If an error occurs during execution for a particular element in the array, execution halts and SQLExecute(), SQLExecDirect() or SQLParamData() returns SQL_ERROR.

The contents of *pirow* have the following uses:

- When SQLParamData() returns SQL_NEED_DATA, the application can access the value in *pirow* to determine which set of parameters is being assigned values.
- When SQLExecute() or SQLExecDirect() returns an error, the application can access the value in *pirow* to find out which element in the parameter value array failed.

- When SQLExecute(), SQLExecDirect(), SQLParamData(), or SQLPutData() succeeds, the value in *pirow* is set to the input value in *crow* to indicate that all elements of the array have been processed successfully.

The output argument *pirow* indicates how many sets of parameters were successfully processed. If the statement processed is a query, *pirow* indicates the array index associated with the current result set returned by SQLMoreResults() and is incremented each time SQLMoreResults() is called.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 92. SQLParamOptions SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
S1107	Row value out of range.	The value in the argument *crow* is less than 1.

Restrictions

None.

Example

Refer to “Array Input Example” on page 356.

References

- “SQLBindParameter - Binds A Parameter Marker to a Buffer” on page 91
- “SQLMoreResults - Determine If There Are More Result Sets” on page 246
- “SQLSetStmtOption - Set Statement Option” on page 315

SQLPrepare - Prepare a Statement

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLPrepare() associates an SQL statement with the input statement handle and sends the statement to the DBMS to be prepared. The application can reference this prepared statement by passing the statement handle to other functions.

If the statement handle has been previously used with a query statement (or any function that returns a result set), SQLFreeStmt() must be called to close the cursor, before calling SQLPrepare().

Syntax

```
SQLRETURN   SQLPrepare       (SQLHSTMT          hstmt,
                              SQLCHAR      FAR  *szSqlStr,
                              SQLINTEGER        cbSqlStr);
```

Function Arguments

Table 93. SQLPrepare Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle. There must not be an open cursor associated with *hstmt*.
SQLCHAR *	*szSqlStr*	input	SQL statement string.
SQLINTEGER	*cbSqlStr*	input	Length of contents of *szSqlStr* argument. This must be set to either the exact length of the SQL statement in *szSqlstr*, or to SQL_NTS if the statement text is null-terminated.

Usage

If the SQL statement text contains vendor escape clause sequences, DB2 CLI first modifies the SQL statement text to the appropriate DB2 specific format before submitting it to the database for preparation. If the application does not generate SQL statements that contain vendor escape clause sequences (see “Using Vendor Escape Clauses” on page 376); then the SQL_NOSCAN statement option should be set to SQL_NOSCAN_ON at the statement level so that DB2 CLI does not perform a scan for any vendor escape clauses.

When a statement is prepared using SQLPrepare(), the application can request information about the format of the result set (if the statement was a query) by calling:

- SQLNumResultCols()
- SQLDescribeCol()
- SQLColAttributes()

The SQL statement string can contain parameter markers and SQLNumParams() can be called to determine the number of parameter markers in the statement. A parameter marker is represented by a "?" character that indicates a position in the statement where an application supplied value is to be substituted when `SQLExecute()` is called. The bind parameter functions, `SQLBindParameter()` and `SQLSetParam()` are used to bind (associate) application values with each parameter marker and to indicate if any data conversion should be performed at the time the data is transferred.

All parameters must be bound before calling `SQLExecute()`. For more information refer to “SQLExecute - Execute a Statement” on page 154.

After the application processes the results from the `SQLExecute()` call, it can execute the statement again with new (or the same) parameter values.

The SQL statement cannot be a COMMIT or ROLLBACK. `SQLTransact()` must be called to issue COMMIT or ROLLBACK. For more information about SQL statements, that DB2 for OS/390 supports, see Table 1 on page 18.

If the SQL statement is a positioned DELETE or a positioned UPDATE, the cursor referenced by the statement must be defined on a separate statement handle under the same connection handle and same isolation level.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 94 (Page 1 of 2). SQLPrepare SQLSTATEs

SQLSTATE	Description	Explanation
01504	The UPDATE or DELETE statement does not include a WHERE clause.	*szSqlStr* contains an UPDATE or DELETE statement which did not contain a WHERE clause.
21S01	Insert value list does not match column list.	*szSqlStr* contains an INSERT statement and the number of values to be inserted did not match the degree of the derived table.
21S02	Degrees of derived table does not match column list.	*szSqlStr* contains a CREATE VIEW statement and the number of names specified is not the same degree as the derived table defined by the query specification.
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
34000	Invalid cursor name.	*szSqlStr* contains a positioned DELETE or a positioned UPDATE and the cursor referenced by the statement being executed is not open.
37xxx [a]	Invalid SQL syntax.	*szSqlStr* contains one or more of the following: • A COMMIT • A ROLLBACK • An SQL statement that the connected database server cannot prepare • A statement containing a syntax error
40001	Transaction rollback.	The transaction to which this SQL statement belongs is rolled back due to deadlock or timeout.

Table 94 (Page 2 of 2). SQLPrepare SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
42xxx [a]	Syntax error or access rule violation	**42**5xx indicates the authorization ID does not have permission to execute the SQL statement contained in *szSqlStr*. Other **42**xxx SQLSTATES indicate a variety of syntax or access problems with the statement.
58004	Unexpected system failure.	Unrecoverable system error.
S0001	Database object already exists.	*szSqlStr* contains a CREATE TABLE or CREATE VIEW statement and the table name or view name specified already exists.
S0002	Database object does not exist.	*szSqlStr* contains an SQL statement that references a table name or a view name that does not exist.
S0011	Index already exists.	*szSqlStr* contains a CREATE INDEX statement and the specified index name already exists.
S0012	Index not found.	*szSqlStr* contains a DROP INDEX statement and the specified index name does not exist.
S0021	Column already exists.	*szSqlStr* contains an ALTER TABLE statement and the column specified in the ADD clause is not unique or identifies an existing column in the base table.
S0022	Column not found.	*szSqlStr* contains an SQL statement that references a column name that does not exist.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	*szSqlStr* is a null pointer.
S1010	Function sequence error.	The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.
S1014	No more handles.	DB2 CLI is not able to allocate a handle due to internal resources.
S1090	Invalid string or buffer length.	The argument *cbSqlStr* is less than 1, but not equal to SQL_NTS.

Note:

a xxx refers to any SQLSTATE with that class code. Example, **37**xxx refers to any SQLSTATE in the **37** class.

Not all DBMSs report all of the above diagnostic messages at prepare time. Therefore, an application must also be able to handle these conditions when calling `SQLExecute()`.

Restrictions

None.

Example

```
 /******************************************************************/
 /*  DB2 for OS/390 Example:                                       */
 /*        Prepares a query and executes that query twice speci-  */
 /*        fying a unique value for the parameter marker.         */
 /******************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include "sqlcli1.h"

int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLHSTMT        hStmt   = SQL_NULL_HSTMT;
   SQLRETURN       rc      = SQL_SUCCESS;
   SQLINTEGER      RETCODE = 0;
   char            *pDSN = "STLEC1";
   SWORD           cbCursor;
   SDWORD          cbValue1;
   SDWORD          cbValue2;
   char            employee [30];
   int             salary = 0;
   int             param_salary = 30000;

   char            *stmt = "SELECT NAME, SALARY FROM EMPLOYEE WHERE SALARY > ?";


   (void) printf ("**** Entering CLIP07.\n\n");

  /*****************************************************************/
  /* Allocate Environment Handle                                   */
  /*****************************************************************/

   RETCODE = SQLAllocEnv(&hEnv);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Allocate Connection Handle to DSN                             */
  /*****************************************************************/

   RETCODE = SQLAllocConnect(hEnv,
                             &hDbc);

   if( RETCODE != SQL_SUCCESS )      // Could not get a Connect Handle
     goto dberror;
```

```
/******************************************************************/
/* CONNECT TO data source (STLEC1)                                */
/******************************************************************/

 RETCODE = SQLConnect(hDbc,        // Connect handle
                      (SQLCHAR *) pDSN, // DSN
                      SQL_NTS,     // DSN is nul-terminated
                      NULL,        // Null UID
                      0   ,
                      NULL,        // Null Auth string
                      0);
 if( RETCODE != SQL_SUCCESS )      // Connect failed
   goto dberror;

/******************************************************************/
/* Allocate Statement Handles                                     */
/******************************************************************/

rc = SQLAllocStmt (hDbc,
                   &hStmt);

if (rc != SQL_SUCCESS)
  goto exit;

/******************************************************************/
/* Preapare the query for multiple execution within current       */
/* transaction. Note that query is collapsed when transaction     */
/* is committed or rolled back.                                   */
/******************************************************************/

rc = SQLPrepare (hStmt,
                 (SQLCHAR *) stmt,
                 strlen(stmt));

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** PREPARE OF QUERY FAILED.\n");
  goto dberror;
}

rc = SQLBindCol (hStmt,           // bind employee name
                 1,
                 SQL_C_CHAR,
                 employee,
                 sizeof(employee),
                 &cbValue1);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** BIND OF NAME FAILED.\n");
  goto dberror;
}

rc = SQLBindCol (hStmt,           // bind employee salary
                 2,
                 SQL_C_LONG,
                 &salary,
                 0,
                 &cbValue2);
if (rc != SQL_SUCCESS)
```

```
{
  (void) printf ("**** BIND OF SALARY FAILED.\n");
  goto dberror;
}

/*****************************************************************/
/* Bind parameter to replace '?' in query. This has an initial   */
/* value of 30000.                                               */
/*****************************************************************/

rc = SQLBindParameter (hStmt,
                       1,
                       SQL_PARAM_INPUT,
                       SQL_C_LONG,
                       SQL_INTEGER,
                       0,
                       0,
                       &param_salary,
                       0,
                       NULL);

/*****************************************************************/
/* Execute prepared statement to generate answer set.            */
/*****************************************************************/

rc = SQLExecute (hStmt);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** EXECUTE OF QUERY FAILED.\n");
  goto dberror;
}

/*****************************************************************/
/* Answer Set is available -- Fetch rows and print employees     */
/* and salary.                                                   */
/*****************************************************************/

(void) printf ("**** Employees whose salary exceeds %d follow.\n\n",
               param_salary);

while ((rc = SQLFetch (hStmt)) == SQL_SUCCESS)
{
  (void) printf ("**** Employee Name %s with salary %d.\n",
                 employee,
                 salary);
}

/*****************************************************************/
/* Close query --- note that query is still prepared. Then change*/
/* bound parameter value to 100000. Then re-execute query.       */
/*****************************************************************/

rc = SQLFreeStmt (hStmt,
                  SQL_CLOSE);

param_salary = 100000;

rc = SQLExecute (hStmt);
if (rc != SQL_SUCCESS)
```

```
{
  (void) printf ("**** EXECUTE OF QUERY FAILED.\n");
  goto dberror;
}

/*****************************************************************/
/* Answer Set is available -- Fetch rows and print employees     */
/* and salary.                                                   */
/*****************************************************************/

(void) printf ("**** Employees whose salary exceeds %d follow.\n\n",
               param_salary);

while ((rc = SQLFetch (hStmt)) == SQL_SUCCESS)
{
  (void) printf ("**** Employee Name %s with salary %d.\n",
                 employee,
                 salary);
}

/*****************************************************************/
/* Deallocate Statement Handles -- statement is no longer in a   */
/* Prepared state.                                               */
/*****************************************************************/

rc = SQLFreeStmt (hStmt,
                  SQL_DROP);

/*****************************************************************/
/* DISCONNECT from data source                                   */
/*****************************************************************/

 RETCODE = SQLDisconnect(hDbc);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;

/*****************************************************************/
/* Deallocate Connection Handle                                  */
/*****************************************************************/

 RETCODE = SQLFreeConnect (hDbc);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;
  /*****************************************************************/
  /* Free Environment Handle                                       */
  /*****************************************************************/

   RETCODE = SQLFreeEnv (hEnv);

   if (RETCODE == SQL_SUCCESS)
     goto exit;

   dberror:
   RETCODE=12;

   exit:

   (void) printf ("**** Exiting  CLIP07.\n\n");

   return RETCODE;
}
```

References

- “SQLBindParameter - Binds A Parameter Marker to a Buffer” on page 91
- “SQLColAttributes - Get Column Attributes” on page 104
- “SQLDescribeCol - Describe Column Attributes” on page 128
- “SQLExecDirect - Execute a Statement Directly” on page 149
- “SQLExecute - Execute a Statement” on page 154
- “SQLNumParams - Get Number of Parameters in A SQL Statement” on page 253
- “SQLNumResultCols - Get Number of Result Columns” on page 255
- “SQLSetParam - Binds A Parameter Marker to a Buffer” on page 310

SQLPrimaryKeys - Get Primary Key Columns of A Table

Purpose

Specification:	ODBC 1.0		

SQLPrimaryKeys() returns a list of column names that comprise the primary key for a table. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN   SQLPrimaryKeys   (SQLHSTMT          hstmt,
                              SQLCHAR      FAR  *szCatalogName,
                              SQLSMALLINT       cbCatalogName,
                              SQLCHAR      FAR  *szSchemaName,
                              SQLSMALLINT       cbSchemaName,
                              SQLCHAR      FAR  *szTableName,
                              SQLSMALLINT       cbTableName);
```

Function Arguments

Table 95. SQLPrimaryKeys Arguments

Data Type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle.
SQLCHAR *	szCatalogName	input	Catalog qualifier of a 3 part table name. This must be a NULL pointer or a zero length string.
SQLSMALLINT	cbCatalogName	input	Length of *szCatalogName*.
SQLCHAR *	szSchemaName	input	Schema qualifier of table name.
SQLSMALLINT	cbSchemaName	input	Length of *szSchemaName*.
SQLCHAR *	szTableName	input	Table name.
SQLSMALLINT	cbTableName	input	Length of *szTableName*.

Usage

SQLPrimaryKeys() returns the primary key columns from a single table. Search patterns cannot be used to specify the schema qualifier or the table name.

The result set contains the columns listed in Table 96 on page 270, ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME and ORDINAL_POSITION.

Since calls to SQLPrimaryKeys() in many cases map to a complex and, thus, expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set have been declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call

SQLGetInfo() with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change.

Table 96. Columns Returned By SQLPrimaryKeys

Column Number/Name	Data Type	Description
1 TABLE_CAT	VARCHAR(128)	This is always null.
2 TABLE_SCHEM	VARCHAR(128)	The name of the schema containing TABLE_NAME.
3 TABLE_NAME	VARCHAR(128) NOT NULL	Name of the specified table.
4 COLUMN_NAME	VARCHAR(128) NOT NULL	Primary key column name.
5 ORDINAL_POSITION	SMALLINT NOT NULL	Column sequence number in the primary key, starting with 1.
6 PK_NAME	VARCHAR(128)	Primary key identifier. NULL if not applicable to the data source.

Note: The column names used by DB2 CLI follow the X/Open CLI CAE specification style. The column types, contents and order are identical to those defined for the SQLPrimaryKeys() result set in ODBC.

If the specified table does not contain a primary key, an empty result set is returned.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 97. SQLPrimaryKeys SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor was already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
S1014	No more handles.	DB2 CLI is not able to allocate a handle due to internal resources.
S1090	Invalid string or buffer length.	The value of one of the name length arguments is less than 0, but not equal SQL_NTS.
S1C00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for table name.

Restrictions

None.

Example

The following example uses `SQLPrimaryKeys` to locate a primary key for a table, and calls `SQLColAttributes` to find its data type.

```
/* ... */

#include <sqlcli1.h>

void main()
{
   SQLCHAR      rgbDesc_20•;
   SQLCHAR      szTableName_20•;
   SQLCHAR      szSchemaName_20•;
   SQLCHAR      rgbValue_20•;
   SQLINTEGER   pcbValue;
   SQLHENV      henv;
   SQLHDBC      hdbc;
   SQLHSTMT     hstmt;
   SQLSMALLINT  pscDesc;
   SQLINTEGER   pdDesc;
   SQLRETURN    rc;

   /*******************************************************************/
   /*   Initialization...                                             */
   /*******************************************************************/

   if( SQLAllocEnv( &henv ) != SQL_SUCCESS )
   {
       fprintf( stdout, "Error in SQLAllocEnv\n" );
       exit(1);
   }
   if( SQLAllocConnect( henv, &hdbc ) != SQL_SUCCESS )
   {
       fprintf( stdout, "Error in SQLAllocConnect\n" );
       exit(1);
   }
   if( SQLConnect( hdbc,
                   NULL, SQL_NTS,
                   NULL, SQL_NTS,
                   NULL, SQL_NTS ) != SQL_SUCCESS )
   {
       fprintf( stdout, "Error in SQLConnect\n" );
       exit(1);
   }
   if( SQLAllocStmt( hdbc, &hstmt ) != SQL_SUCCESS )
   {
       fprintf( stdout, "Error in SQLAllocStmt\n" );
       exit(1);
   }
```

```
/*******************************************************************/
/*  Get primary key for table 'myTable' by using SQLPrimaryKeys    */
/*******************************************************************/
rc = SQLPrimaryKeys( hstmt,
                     NULL, SQL_NTS,
                     (SQLCHAR*)szSchemaName, SQL_NTS,
                     (SQLCHAR*)szTableName, SQL_NTS );
if( rc != SQL_SUCCESS )
{
    goto exit;
}

/*
 *   Since all we need is the ordinal position, we'll bind column 5 from
 *   the result set.
 */
rc = SQLBindCol( hstmt,
                 5,
                 SQL_C_CHAR,
                 (SQLPOINTER)rgbValue,
                 20,
                 &pcbValue );
if( rc != SQL_SUCCESS )
{
    goto exit;
}
/*
 *   Fetch data...
 */
if( SQLFetch( hstmt ) != SQL_SUCCESS )
{
     goto exit;
}

/*******************************************************************/
/*  Get data type for that column by calling SQLColAttributes().   */
/*******************************************************************/

rc =  SQLColAttributes( hstmt,
                        pcbValue,
                        SQL_COLUMN_TYPE,
                        rgbDesc,
                        20,
                        &pcbDesc,
                        &pfDesc );
if( rc != SQL_SUCCESS )
{
    goto exit;
}

/*
 *   Display the data type.
 */

fprintf( stdout, "Data type ==> %s\n", rgbDesc );
```

```
exit:
   /*******************************************************************/
   /* Clean up the environment...                                     */
   /*******************************************************************/

   SQLTransact( henv,
                hdbc,
                SQL_ROLLBACK );

   SQLDisconnect( hdbc );

   SQLFreeConnect( hdbc );

   SQLFreeEnv( henv );

}
```

References

- “SQLForeignKeys - Get the List of Foreign Key Columns” on page 169
- “SQLStatistics - Get Index and Statistics Information For A Base Table” on page 325

SQLProcedureColumns - Get Input/Output Parameter Information for A Procedure

Purpose

Specification:	**ODBC** 1.0		

SQLProcedureColumns() returns a list of input and output parameters associated with a procedure. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN   SQLProcedureColumns (
                                   SQLHSTMT          hstmt,
                                   SQLCHAR      FAR  *szProcCatalog,
                                   SQLSMALLINT       cbProcCatalog,
                                   SQLCHAR      FAR  *szProcSchema,
                                   SQLSMALLINT       cbProcSchema,
                                   SQLCHAR      FAR  *szProcName,
                                   SQLSMALLINT       cbProcName,
                                   SQLCHAR      FAR  *szColumnName,
                                   SQLSMALLINT       cbColumnName);
```

Function Arguments

Table 98. SQLProcedureColumns Arguments

Data Type	**Argument**	**Use**	**Description**
SQLHSTMT	hstmt	input	Statement handle.
SQLCHAR *	szProcCatalog	input	Catalog qualifier of a 3 part procedure name. This must be a NULL pointer or a zero length string.
SQLSMALLINT	cbProcCatalog	input	Length of *szProcCatalog*. This must be set to 0.
SQLCHAR *	szProcSchema	input	Buffer that can contain a *pattern-value* to qualify the result set by schema name. For DB2 for OS/390, all the stored procedures are in one schema; the only acceptable value for the *szProcSchema* argument is a null pointer. For DB2 for common server, *szProcSchema* can contain a valid pattern value. For more information about valid search patterns, refer to "Querying System Catalog Information" on page 348.
SQLSMALLINT	cbProcSchema	input	Length of *szProcSchema*.
SQLCHAR *	szProcName	input	Buffer that can contain a *pattern-value* to qualify the result set by procedure name.
SQLSMALLINT	cbProcName	input	Length of *szProcName*.
SQLCHAR *	szColumnName	input	Buffer that can contain a *pattern-value* to qualify the result set by parameter name. This argument is to be used to further qualify the result set already restricted by specifying a non-empty value for szProcName and/or szProcSchema.
SQLSMALLINT	cbColumnName	input	Length of *szColumnName*.

Usage

If the stored procedure is at a DB2 for MVS/ESA Version 4 server or later, the name of the stored procedures must be registered in the server's SYSIBM.SYSPROCEDURES catalog table.

For versions of other DB2 servers that do not provide facilities for a stored procedure catalog, an empty result set is returned.

DB2 CLI returns information on the input, input/output, and output parameters associated with the stored procedure, but cannot return information on the descriptor information for any result sets returned.

SQLProcedureColumns() returns the information in a result set, ordered by PROCEDURE_CAT, PROCEDURE_SCHEM, PROCEDURE_NAME, and COLUMN_TYPE. Table 99 lists the columns in the result set.

Since calls to SQLProcedureColumns() in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set have been declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call SQLGetInfo() with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Applications should be aware that columns beyond the last column might be defined in future releases. Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change.

Table 99 (Page 1 of 3). Columns Returned By SQLProcedureColumns

Column Number/Name	Data Type	Description
1 PROCEDURE_CAT	VARCHAR(128)	The is always null.
2 PROCEDURE_SCHEM	VARCHAR(128)	The name of the schema containing PROCEDURE_NAME. (This is also NULL for DB2 for OS/390 SQLProcedureColumns() result sets.)
3 PROCEDURE_NAME	VARCHAR(128)	Name of the procedure.
4 COLUMN_NAME	VARCHAR(128)	Name of the parameter.

Table 99 (Page 2 of 3). Columns Returned By SQLProcedureColumns

Column Number/Name	Data Type	Description
5 COLUMN_TYPE	SMALLINT NOT NULL	Identifies the type information associated with this row. The values can be: • SQL_PARAM_TYPE_UNKNOWN: the parameter type is unknown. **Note:** This is not returned. • SQL_PARAM_INPUT: this parameter is an input parameter. • SQL_PARAM_INPUT_OUTPUT: this parameter is an input / output parameter. • SQL_PARAM_OUTPUT: this parameter is an output parameter. • SQL_RETURN_VALUE: the procedure column is the return value of the procedure. **Note:** This is not returned. • SQL_RESULT_COL: this parameter is actually a column in the result set. **Note:** This is not returned. **Note:** SQL_PARAM_OUTPUT and SQL_RETURN_VALUE are supported only on ODBC 2.0 or higher.
6 DATA_TYPE	SMALLINT NOT NULL	SQL data type.
7 TYPE_NAME	VARCHAR(128) NOT NULL	Character string representing the name of the data type corresponding to DATA_TYPE.
8 COLUMN_SIZE	INTEGER	If the DATA_TYPE column value denotes a character or binary string, then this column contains the maximum length in bytes; if it is a graphic (DBCS) string, this is the number of double byte characters for the parameter. For date, time, timestamp data types, this is the total number of bytes required to display the value when converted to character. For numeric data types, this is either the total number of digits, or the total number of bits allowed in the column, depending on the value in the NUM_PREC_RADIX column in the result set. See Table 144 on page 420.
9 BUFFER_LENGTH	INTEGER	The maximum number of bytes for the associated C buffer to store data from this parameter if SQL_C_DEFAULT is specified on the `SQLBindCol()`, `SQLGetData()` and `SQLBindParameter()` calls. This length excludes any null-terminator. For exact numeric data types, the length accounts for the decimal and the sign. See Table 146 on page 422.
10 DECIMAL_DIGITS	SMALLINT	The scale of the parameter. NULL is returned for data types where scale is not applicable. See Table 145 on page 421.

Table 99 (Page 3 of 3). Columns Returned By SQLProcedureColumns

Column Number/Name	Data Type	Description
11 NUM_PREC_RADIX	SMALLINT	Either 10 or 2 or NULL. If DATA_TYPE is an approximate numeric data type, this column contains the value 2, then the COLUMN_SIZE column contains the number of bits allowed in the parameter. If DATA_TYPE is an exact numeric data type, this column contains the value 10 and the COLUMN_SIZE and DECIMAL_DIGITS columns contain the number of decimal digits allowed for the parameter. For numeric data types, the DBMS can return a NUM_PREC_RADIX of either 10 or 2. NULL is returned for data types where radix is not applicable.
12 NULLABLE	SMALLINT NOT NULL	SQL_NO_NULLS if the parameter does not accept NULL values. SQL_NULLABLE if the parameter accepts NULL values.
13 REMARKS	VARCHAR(254)	Might contain descriptive information about the parameter.
14 ORDINAL_POSITION	INTEGER NOT NULL	Contains the ordinal position of the parameter given by COLUMN_NAME in this result set. This is the ordinal position of the argument provided on the CALL statement. The leftmost argument has an ordinal position of 1.

Note: The column names used by DB2 CLI follow the X/Open CLI CAE specification style. The column types, contents and order are identical to those defined for the `SQLProcedureColumns()` result set in ODBC.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 100 (Page 1 of 2). SQLProcedureColumns SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
42601	PARMLIST syntax error.	The PARMLIST value in the stored procedures catalog table contains a syntax error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1014	No more handles.	DB2 CLI is not able to allocate a handle due to internal resources.
S1090	Invalid string or buffer length	The value of one of the name length arguments is less than 0, but not equal SQL_NTS.

Table 100 (Page 2 of 2). SQLProcedureColumns SQLSTATEs

SQLSTATE	Description	Explanation
S1C00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for procedure name. The connected server does not support *schema* as a qualifier for procedure name.

Restrictions

`SQLProcedureColumns()` does not return information about the attributes of result sets that stored procedures can return.

If an application is connected to a DB2 server that does not provide support for stored procedures, or for a stored procedure catalog, `SQLProcedureColumns()` returns an empty result set.

Example

```
 /******************************************************************/
 /*  DB2 for OS/390 Example:                                       */
 /*       Invokes SQLProcedureColumns and enumerates all rows      */
 /*       retrieved.                                               */
 /******************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include "sqlcli1.h"

int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLHSTMT        hStmt   = SQL_NULL_HSTMT;
   SQLRETURN       rc      = SQL_SUCCESS;
   SQLINTEGER      RETCODE = 0;
   char            *pDSN = "STLEC1";
   char            procedure_name [20];
   char            parameter_name [20];
   char            ptype          [20];
   SQLSMALLINT     parameter_type = 0;
   SQLSMALLINT     data_type = 0;
   char            type_name      [20];
   SWORD           cbCursor;
   SDWORD          cbValue3;
   SDWORD          cbValue4;
   SDWORD          cbValue5;
   SDWORD          cbValue6;
   SDWORD          cbValue7;
   char            ProcCatalog [20] = {0};
   char            ProcSchema  [20] = {0};
   char            ProcName    [20] = {"DOIT%"};
   char            ColumnName  [20] = {"P%"};
   SQLSMALLINT     cbProcCatalog = 0;
   SQLSMALLINT     cbProcSchema  = 0;
   SQLSMALLINT     cbProcName    = strlen(ProcName);
   SQLSMALLINT     cbColumnName  = strlen(ColumnName);
   (void) printf ("**** Entering CLIP12.\n\n");

  /*****************************************************************/
  /* Allocate Environment Handle                                   */
  /*****************************************************************/

   RETCODE = SQLAllocEnv(&hEnv);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Allocate Connection Handle to DSN                             */
  /*****************************************************************/

   RETCODE = SQLAllocConnect(hEnv,
                             &hDbc);

   if( RETCODE != SQL_SUCCESS )      // Could not get a Connect Handle
     goto dberror;

  /*****************************************************************/
  /* CONNECT TO data source (STLEC1)                               */
  /*****************************************************************/

   RETCODE = SQLConnect(hDbc,        // Connect handle
                        (SQLCHAR *) pDSN, // DSN
                        SQL_NTS,     // DSN is nul-terminated
                        NULL,        // Null UID
                        0   ,
                        NULL,        // Null Auth string
                        0);

   if( RETCODE != SQL_SUCCESS )      // Connect failed
     goto dberror;

  /*****************************************************************/
  /* Allocate Statement Handles                                    */
  /*****************************************************************/

  rc = SQLAllocStmt (hDbc,
                     &hStmt);

  if (rc != SQL_SUCCESS)
    goto exit;

  /*****************************************************************/
  /* Invoke SQLProcedureColumns and retrieve all rows within       */
  /* answer set.                                                   */
  /*****************************************************************/

  rc = SQLProcedureColumns (hStmt                  ,
                            (SQLCHAR *) ProcCatalog,
                            cbProcCatalog          ,
                            (SQLCHAR *) ProcSchema ,
                            cbProcSchema           ,
                            (SQLCHAR *) ProcName   ,
                            cbProcName             ,
                            (SQLCHAR *) ColumnName ,
                            cbColumnName);
```

```
if (rc != SQL_SUCCESS)
{
  (void) printf ("**** SQLProcedureColumns Failed.\n");
  goto dberror;
}

rc = SQLBindCol (hStmt,             // bind procedure_name
                  3,
                  SQL_C_CHAR,
                  procedure_name,
                  sizeof(procedure_name),
                  &cbValue3);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** Bind of procedure_name Failed.\n");
  goto dberror;
}

rc = SQLBindCol (hStmt,             // bind parameter_name
                  4,
                  SQL_C_CHAR,
                  parameter_name,
                  sizeof(parameter_name),
                  &cbValue4);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** Bind of parameter_name Failed.\n");
  goto dberror;
}

rc = SQLBindCol (hStmt,             // bind parameter_type
                  5,
                  SQL_C_SHORT,
                  &parameter_type,
                  0,
                  &cbValue5);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** Bind of parameter_type Failed.\n");
  goto dberror;
}

rc = SQLBindCol (hStmt,             // bind SQL data type
                  6,
                  SQL_C_SHORT,
                  &data_type,
                  0,
                  &cbValue6);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** Bind of data_type Failed.\n");
  goto dberror;
}
```

```
rc = SQLBindCol (hStmt,           // bind type_name
                 7,
                 SQL_C_CHAR,
                 type_name,
                 sizeof(type_name),
                 &cbValue7);

if (rc != SQL_SUCCESS)
{
  (void) printf ("**** Bind of type_name Failed.\n");
  goto dberror;
}

/*****************************************************************/
/* Answer Set is available - Fetch rows and print parameters for */
/* all procedures.                                               */
/*****************************************************************/

while ((rc = SQLFetch (hStmt)) == SQL_SUCCESS)
{
  (void) printf ("**** Procedure Name = %s. Parameter %s",
                 procedure_name,
                 parameter_name);

  switch (parameter_type)
  {
    case SQL_PARAM_INPUT        :
      (void) strcpy (ptype, "INPUT");
      break;
    case SQL_PARAM_OUTPUT       :
      (void) strcpy (ptype, "OUTPUT");
      break;
    case SQL_PARAM_INPUT_OUTPUT :
      (void) strcpy (ptype, "INPUT/OUTPUT");
      break;
    default                     :
      (void) strcpy (ptype, "UNKNOWN");
      break;
  }

  (void) printf (" is %s. Data Type is %d. Type Name is %s.\n",
                 ptype     ,
                 data_type ,
                 type_name);
}

/*****************************************************************/
/* Deallocate Statement Handles -- statement is no longer in a   */
/* Prepared state.                                               */
/*****************************************************************/

rc = SQLFreeStmt (hStmt,
                  SQL_DROP);

/*****************************************************************/
/* DISCONNECT from data source                                   */
/*****************************************************************/

 RETCODE = SQLDisconnect(hDbc);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;
  /*****************************************************************/
  /* Deallocate Connection Handle                                  */
  /*****************************************************************/

   RETCODE = SQLFreeConnect (hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;

  /*****************************************************************/
  /* Free Environment Handle                                       */
  /*****************************************************************/

   RETCODE = SQLFreeEnv (hEnv);

   if (RETCODE == SQL_SUCCESS)
     goto exit;

   dberror:
   RETCODE=12;

   exit:

   (void) printf ("**** Exiting  CLIP12.\n\n");

   return RETCODE;
}
```

References

- “SQLProcedures - Get List of Procedure Names” on page 283

SQLProcedures - Get List of Procedure Names

Purpose

Specification:	ODBC 1.0		

SQLProcedures() returns a list of procedure names that have been registered at the server, and which match the specified search pattern.

The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN   SQLProcedures    (SQLHSTMT          hstmt,
                              SQLCHAR     FAR   *szProcCatalog,
                              SQLSMALLINT       cbProcCatalog,
                              SQLCHAR     FAR   *szProcSchema,
                              SQLSMALLINT       cbProcSchema,
                              SQLCHAR     FAR   *szProcName,
                              SQLSMALLINT       cbProcName);
```

Function Arguments

Table 101. SQLTables Arguments

Data Type	Argument	Use	Description
SQLHSTMT	hstmt	Input	Statement handle.
SQLCHAR *	szProcCatalog	Input	Catalog qualifier of a 3 part procedure name. This must be a NULL pointer or a zero length string.
SQLSMALLINT	cbProcCatalog	Input	Length of *szProcCatalog*. This must be set to 0.
SQLCHAR *	szProcSchema	Input	Buffer that can contain a *pattern-value* to qualify the result set by schema name. For DB2 for OS/390, all the stored procedures are in one schema; the only acceptable value for the *szProcSchema* argument is a null pointer. For DB2 for common server, *szProcSchema* can contain a valid pattern value. For more information about valid search patterns, refer to “Querying System Catalog Information” on page 348.
SQLSMALLINT	cbProcSchema	Input	Length of *szProcSchema*.
SQLCHAR *	szProcName	Input	Buffer that can contain a *pattern-value* to qualify the result set by table name.
SQLSMALLINT	cbProcName	Input	Length of *szProcName*.

Usage

If the stored procedure is at a DB2 for MVS/ESA Version 4 server or later, the name of the stored procedures must be registered in the server's SYSIBM.SYSPROCEDURES catalog table.

For other versions of DB2 servers that do not provide facilities for a stored procedure catalog, an empty result set is returned.

The result set returned by `SQLProcedures()` contains the columns listed in Table 102 in the order given. The rows are ordered by PROCEDURE_CAT, PROCEDURE_SCHEMA, and PROCEDURE_NAME.

Since calls to `SQLProcedures()` in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set have been declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call `SQLGetInfo()` with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change.

Table 102. Columns Returned By SQLProcedures

Column Number/Name	Data Type	Description
1 PROCEDURE_CAT	VARCHAR(128)	This is always null.
2 PROCEDURE_SCHEM	VARCHAR(128)	The name of the schema containing PROCEDURE_NAME.
3 PROCEDURE_NAME	VARCHAR(128) NOT NULL	The name of the procedure.
4 NUM_INPUT_PARAMS	INTEGER not NULL	Number of input parameters.
5 NUM_OUTPUT_PARAMS	INTEGER not NULL	Number of output parameters.
6 NUM_RESULT_SETS	INTEGER not NULL	Number of result sets returned by the procedure.
7 REMARKS	VARCHAR(254)	Contains the descriptive information about the procedure.
8 PROCEDURE_TYPE	SMALLINT	Defines the procedure type: • SQL_PT_UNKNOWN: It cannot be determined whether the procedure returns a value. • SQL_PT_PROCEDURE: The returned object is a procedure; that is, it does not have a return value. • SQL_PT_FUNCTION: The returned object is a function; that is, it has a return value. DB2 CLI always returns SQL_PT_PROCEDURE.

Note: The column names used by DB2 CLI follow the X/Open CLI CAE specification style. The column types, contents and order are identical to those defined for the `SQLProcedures()` result set in ODBC.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 103. SQLProcedures SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1014	No more handles.	DB2 CLI is not able to allocate a handle due to internal resources.
S1090	Invalid string or buffer length.	The value of one of the name length arguments is less than 0, but not equal to SQL_NTS.
S1C00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for procedure name. The connected server does not supported schema as a qualifier for procedure name.

Restrictions

If an application is connected to a DB2 server that does not provide support for stored procedures, or for a stored procedure catalog, `SQLProcedureColumns()` returns an empty result set.

Example

```
/* ... */

    printf("Enter Procedure Schema Name Search Pattern:\n");
    gets(proc_schem.s);

    rc = SQLProcedures(hstmt, NULL, 0, proc_schem.s, SQL_NTS, "%", SQL_NTS);

    rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) proc_schem.s, 129,
                    &proc_schem.ind);

    rc = SQLBindCol(hstmt, 3, SQL_C_CHAR, (SQLPOINTER) proc_name.s, 129,
                    &proc_name.ind);

    rc = SQLBindCol(hstmt, 7, SQL_C_CHAR, (SQLPOINTER) remarks.s, 255,
                    &remarks.ind);

    printf("PROCEDURE SCHEMA          PROCEDURE NAME            \n");
    printf("------------------------- ------------------------- \n");
    /* Fetch each row, and display */
    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
        printf("%-25s %-25s\n", proc_schem.s, proc_name.s);
        if (remarks.ind != SQL_NULL_DATA) {
            printf("  (Remarks) %s\n", remarks.s);
        }
    }                           /* endwhile */
/* ... */
```

References

- "SQLProcedureColumns - Get Input/Output Parameter Information for A Procedure" on page 274

SQLPutData - Passing Data Value for A Parameter

Purpose

Specification:	**ODBC** 1.0	**X/OPEN CLI**	**ISO CLI**

`SQLPutData()` is called following an `SQLParamData()` call returning SQL_NEED_DATA to supply parameter data values. This function can be used to send large parameter values in pieces.

The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN   SQLPutData       (SQLHSTMT         hstmt,
                              SQLPOINTER       rgbValue,
                              SQLINTEGER       cbValue);
```

Function Arguments

Table 104. SQLPutData Arguments

Data Type	Argument	Use	Description
SQLHSTMT	hstmt	Input	Statement handle.
SQLPOINTER	rgbValue	Input	Pointer to the actual data, or portion of data, for a parameter. The data must be in the form specified in the `SQLBindParameter()` call that the application used when specifying the parameter.
SQLINTEGER	cbValue	Input	Length of *rgbValue*. Specifies the amount of data sent in a call to `SQLPutData()` . The amount of data can vary with each call for a given parameter. The application can also specify SQL_NTS or SQL_NULL_DATA for *cbValue*. *cbValue* is ignored for all fixed length C buffer types, such as date, time, timestamp, and all numeric C buffer types. For cases where the C buffer type is SQL_C_CHAR or SQL_C_BINARY, or if SQL_C_DEFAULT is specified as the C buffer type and the C buffer type default is SQL_C_CHAR or SQL_C_BINARY, this is the number of bytes of data in the *rgbValue* buffer.

Usage

For a description on the `SQLParamData()` and `SQLPutData()` sequence, refer to “Sending/Retrieving Long Data in Pieces” on page 351.

The application calls `SQLPutData()` after calling `SQLParamData()` on a statement in the SQL_NEED_DATA state to supply the data values for an SQL_DATA_AT_EXEC parameter. Long data can be sent in pieces via repeated calls to `SQLPutData()`. After all the pieces of data for the parameter have been sent, the application calls `SQLParamData()` again to proceed to the next

SQL_DATA_AT_EXEC parameter, or, if all parameters have data values, to execute the statement.

SQLPutData() cannot be called more than once for a fixed length C buffer type, such as SQL_C_LONG.

After an SQLPutData() call, the only legal function calls are SQLParamData(), SQLCancel(), or another SQLPutData() if the input data is character or binary data. As with SQLParamData(), all other function calls using this statement handle fail. In addition, all function calls referencing the parent *hdbc* of *hstmt* fail if they involve changing any attribute or state of that connection; that is, the following function calls on the parent *hdbc* are also not permitted:

- SQLAllocConnect()
- SQLAllocStmt()
- SQLSetConnectOption()
- SQLNativeSql()
- SQLTransact()

Should they be invoked during an SQL_NEED_DATA sequence, these functions return SQL_ERROR with SQLSTATE of **S1**010 and the processing of the SQL_DATA_AT_EXEC parameters is not affected.

If one or more calls to SQLPutData() for a single parameter results in SQL_SUCCESS, attempting to call SQLPutData() with *cbValue* set to SQL_NULL_DATA for the same parameter results in an error with SQLSTATE of **22**005. This error does not result in a change of state; the statement handle is still in a *Need Data* state and the application can continue sending parameter data.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Some of the following diagnostic conditions are also reported on the final SQLParamData() call rather than at the time the SQLPutData() is called.

Table 105 (Page 1 of 2). SQLPutData SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	The data sent for a numeric parameter is truncated without the loss of significant digits. Timestamp data sent for a date or time column is truncated. Function returns with SQL_SUCCESS_WITH_INFO.
22001	String data right truncation.	More data is sent for a binary or char data than the data source can support for that column.
22003	Numeric value out of range.	The data sent for a numeric parameter cause the whole part of the number to be truncated when assigned to the associated column. SQLPutData() was called more than once for a fixed length parameter.

Table 105 (Page 2 of 2). SQLPutData SQLSTATEs

SQLSTATE	Description	Explanation
22005	Error in assignment.	The data sent for a parameter is incompatible with the data type of the associated table column.
22007	Invalid datetime format.	The data value sent for a date, time, or timestamp parameters is invalid.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	The argument *rgbValue* is a NULL pointer, and the argument *cbValue* is neither 0 nor SQL_NULL_DATA.
S1010	Function sequence error.	The statement handle *hstmt* must be in a need data state and must have been positioned on an SQL_DATA_AT_EXEC parameter via a previous `SQLParamData()` call.
S1090	Invalid string or buffer length.	The argument *rgbValue* is not a NULL pointer, and the argument *cbValue* is less than 0, but not equal to SQL_NTS or SQL_NULL_DATA.

Restrictions

A new value for *pcbValue*, SQL_DEFAULT_PARAM, was introduced in ODBC 2.0, to indicate that the procedure is to use the default value of a parameter, rather than a value sent from the application. Since the concept of default values does not apply to DB2 stored procedure arguments, specification of this value for the *pcbValue* argument results in an error when the CALL statement is executed because the SQL_DEFAULT_PARAM value is considered an invalid length.

ODBC 2.0 also introduced the SQL_LEN_DATA_AT_EXEC(*length*) macro to be used with the *pcbValue* argument. The macro is used to specify the sum total length of the entire data that would be sent for character or binary C data via the subsequent `SQLPutData()` calls. Since the DB2 ODBC driver does not need this information, the macro is not needed. An ODBC application calls `SQLGetInfo()` with the SQL_NEED_LONG_DATA_LEN option to check if the driver needs this information. The DB2 CLI ODBC driver returns 'N' to indicate that this information is not needed by `SQLPutData()`.

Example

Refer to “Example” on page 197.

References

- “SQLBindParameter - Binds A Parameter Marker to a Buffer” on page 91
- “SQLExecute - Execute a Statement” on page 154
- “SQLExecDirect - Execute a Statement Directly” on page 149
- “SQLParamData - Get Next Parameter For Which A Data Value Is Needed” on page 257
- “SQLCancel - Cancel Statement” on page 102

SQLRowCount - Get Row Count

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLRowCount() returns the number of rows in a table that were affected by an UPDATE, INSERT, or DELETE statement executed against the table, or a view based on the table.

SQLExecute() or SQLExecDirect() must be called before calling this function.

Syntax

```
SQLRETURN   SQLRowCount      (SQLHSTMT         hstmt,
                              SQLINTEGER  FAR   *pcrow);
```

Function Arguments

Table 106. SQLRowCount Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle
SQLINTEGER *	*pcrow*	output	Pointer to location where the number of rows affected is stored.

Usage

If the last executed statement referenced by the input statement handle was not an UPDATE, INSERT, or DELETE statement, or if it did not execute successfully, then the function sets the contents of *pcrow* to -1.

If SQLRowCount() is executed after the SQLExecDirect() or SQLExecute() of an SQL statement other than INSERT, UPDATE, or DELETE, it results in return code 0 and *pcrow* is set to -1.

Any rows in other tables that might be affected by the statement (for example, cascading deletes) are not included in the count.

If SQLRowCount() is executed after a built-in function (for example, SQLTables()), it results in return code -1 and SQLSTATE S1010.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 107. SQLRowCount SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	The function is called prior to calling `SQLExecute()` or `SQLExecDirect()` for the *hstmt*.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.

Restrictions

None.

Example

Refer to “Example” on page 130.

References

- “SQLExecDirect - Execute a Statement Directly” on page 149
- “SQLExecute - Execute a Statement” on page 154
- “SQLNumResultCols - Get Number of Result Columns” on page 255

SQLSetColAttributes - Set Column Attributes

Purpose

Specification:			

SQLSetColAttributes() sets the data source result descriptor (column name, type, precision, scale and nullability) for one column in the result set so that the DB2 CLI implementation does not have to obtain the descriptor information from the DBMS server.

Syntax

```
SQLRETURN  SQLSetColAttributes (SQLHSTMT        hstmt,
                               SQLUSMALLINT     icol,
                               SQLCHAR     FAR  *pszColName,
                               SQLSMALLINT      cbColName,
                               SQLSMALLINT      fSQLType,
                               SQLUINTEGER      cbColDef,
                               SQLSMALLINT      ibScale,
                               SQLSMALLINT      fNullable);
```

Function Arguments

Table 108 (Page 1 of 2). SQLSetColAttributes Arguments

Data Type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle.
SQLUSMALLINT	*icol*	input	Column number of result data, ordered sequentially left to right, starting at 1.
SQLCHAR *	szColName	input	Pointer to the column name. If the column is unnamed or is an expression, this pointer can be set to NULL, or an empty string can be used.
SQLSMALLINT	cbColName	input	Length of szColName buffer.

Table 108 (Page 2 of 2). SQLSetColAttributes Arguments

Data Type	Argument	Use	Description
SQLSMALLINT	fSqlType	input	The SQL data type of the column. The following values are recognized: • SQL_BINARY • SQL_CHAR • SQL_DATE • SQL_DECIMAL • SQL_DOUBLE • SQL_FLOAT • SQL_GRAPHIC • SQL_INTEGER • SQL_LONGVARBINARY • SQL_LONGVARCHAR • SQL_LONGVARGRAPHIC • SQL_NUMERIC • SQL_REAL • SQL_SMALLINT • SQL_TIME • SQL_TIMESTAMP • SQL_VARBINARY • SQL_VARCHAR • SQL_VARGRAPHIC
SQLUINTEGER	cbColDef	input	The precision of the column on the data source.
SQLSMALLINT	ibScale	input	The scale of the column on the data source. This is ignored for all data types except SQL_DECIMAL, SQL_NUMERIC, SQL_TIMESTAMP.
SQLSMALLINT	fNullable	input	Indicates whether the column allows NULL value. This must of one of the following values: • SQL_NO_NULLS - the column does not allow NULL values. • SQL_NULLABLE - the column allows NULL values.

Usage

This function is designed to help reduce the amount of network traffic that can result when an application is fetching result sets that contain an extremely large number of columns. If the application has advanced knowledge of the characteristics of the descriptor information of a result set (that is, the exact number of columns, column name, data type, nullability, precision, or scale), then it can inform DB2 CLI rather than having DB2 CLI obtain this information from the database, thus reducing the quantity of network traffic.

An application typically calls `SQLSetColAttributes()` after a call to `SQLPrepare()` and before the associated call to `SQLExecute()`. An application can also call `SQLSetColAttributes()` before a call to `SQLExecDirect()`. This function is valid only after the statement option SQL_NODESCRIBE has been set to SQL_NODESCRIBE_ON for this statement handle.

`SQLSetColAttributes()` informs DB2 CLI of the column name, type, and length that would be generated by the subsequent execution of the query. This allows DB2 CLI to determine whether any data conversion is necessary when the result is returned to the application. The application should only use this function if it has prior knowledge of the exact nature of the result set. The application **must** provide the result descriptor information for every column in the result set or an error occurs on the subsequent fetch (SQLSTATE **07**002). Using this function only benefits those applications that handle an extremely large number (hundreds) of columns in a result set, otherwise the effect is minimal.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 109. SQLSetColAttributes SQLSTATEs

SQLSTATE	Description	Explanation
01004	Data truncated.	szColName contains a column name that is too long. To obtain the maximum length of the column name, call SQLGetInfo with the fInfoType SQL_MAX_COLUMN_NAME_LEN.
24000	Invalid cursor state.	A cursor is already open on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1000	General error.	An error occurred for which there is no specific SQLSTATE and for which no implementation defined SQLSTATE is defined. The error message returned by SQLError in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1002	Invalid column number.	The value specified for the argument *icol* is less than 1 or greater than the maximum number of columns supported by the server.
S1004	SQL data type out of range.	The value specified for the argument *fSqlType* is not a valid SQL data type.
S1010	Function sequence error.	The function is called while in a data-at-execute (SQLParamData(), SQLPutData()) operation.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.
S1090	Invalid string or buffer length.	The value specified for the argument *cbColName* is less than 0 and not equal to SQL_NTS.
S1094	Invalid scale value.	The value specified for *fSqlType* is either SQL_DECIMAL or SQL_NUMERIC and the value specified for *ibScale* is less than 0 or greater than the value for the argument *cbColDef* (precision). The value specified for *fSqlType* is SQL_TIMESTAMP and the value for *ibScale* is less than 0 or greater than 6.
S1099	Nullable type out of range.	The value specified for *fNullable* is invalid.
S1104	Invalid precision value.	The value specified for *fSqlType* is either SQL_DECIMAL or SQL_NUMERIC and the value specified for *cbColDef* is less than 1.

Restrictions

None.

Example

```
/* ... */
    SQLCHAR         stmt[] =
    { "Select id, name from staff" };
/* ... */

  /* Tell DB2 CLI not to get Column Attribute from the server for this hstmt */
    rc = SQLSetStmtOption(hstmt, SQL_NODESCRIBE, SQL_NODESCRIBE_ON);

    rc = SQLPrepare(hstmt, stmt, SQL_NTS);

/* Provide the columns attributes to DB2 CLI for this hstmt */
    rc = SQLSetColAttributes(hstmt, 1, "-ID-", SQL_NTS, SQL_SMALLINT,
                             5, 0, SQL_NO_NULLS);
    rc = SQLSetColAttributes(hstmt, 2, "-NAME-", SQL_NTS, SQL_CHAR,
                             9, 0, SQL_NULLABLE);
    rc = SQLExecute(hstmt);

    print_results(hstmt); /* Call sample function to print column attributes
                             and fetch and print rows.  */

    rc = SQLFreeStmt(hstmt, SQL_DROP);

    rc = SQLTransact(henv, hdbc, SQL_COMMIT);

    printf("Disconnecting .....\n");
    rc = SQLDisconnect(hdbc);

    rc = SQLFreeConnect(hdbc);
    if (rc != SQL_SUCCESS)
        return (terminate(henv, rc));

    rc = SQLFreeEnv(henv);
    if (rc != SQL_SUCCESS)
        return (terminate(henv, rc));

    return (SQL_SUCCESS);
}                               /* end main */
```

References

- “SQLColAttributes - Get Column Attributes” on page 104
- “SQLDescribeCol - Describe Column Attributes” on page 128
- “SQLExecute - Execute a Statement” on page 154
- “SQLExecDirect - Execute a Statement Directly” on page 149
- “SQLPrepare - Prepare a Statement” on page 261

SQLSetConnection - Set Connection Handle

Purpose

Specification:			

This function is needed if the application needs to deterministically switch to a particular connection before continuing execution. It should only be used when the application is mixing DB2 CLI function calls with embedded SQL function calls and multiple connections are involved.

Syntax

```
SQLRETURN  SQLSetConnection  (SQLHDBC          hdbc);
```

Function Arguments

Table 110. SQLSetConnection Arguments

Data Type	**Argument**	**Use**	**Description**
SQLHDBC	hdbc	input	The connection handle associated with the connection to which the application wishes to switch.

Usage

Call Level Interface allows multiple concurrent connections. It is not clear which connection an embedded SQL routine uses when invoked. In practice, the embedded routine uses the connection associated with the most recent network activity. However, from the application's perspective, this is not always easy to determine and it is difficult to keep track of this information. `SQLSetConnection()` is used to allow the application to *explicitly* specify which connection is active. The application can then call the embedded SQL routine.

`SQLSetConnection()` is not needed at all if the application makes purely Call Level Interface calls. This is because each statement handle is implicitly associated with a connection handle and there is never any confusion as to which connection a particular DB2 CLI function applies.

For more information on using embedded SQL within DB2 CLI applications refer to “Mixing Embedded SQL and DB2 CLI” on page 373.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 111. SQLSetConnection SQLSTATEs

SQLSTATE	Description	Explanation
08003	Connection is closed.	The connection handle provided is not currently associated with an open connection to a database server.
S1000	General error.	An error occurred for which there is no specific SQLSTATE and for which the implementation does not define an SQLSTATE. SQLError returns an error message in the argument *szErrorMsg* that describes the error and its cause.

Restrictions

None.

Example

Refer to "Mixed Embedded SQL and DB2 CLI Example" on page 373.

References

- "SQLConnect - Connect to a Data Source" on page 120
- "SQLDriverConnect - (Expanded) Connect to a Data Source" on page 137

SQLSetConnectOption - Set Connection Option

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	

SQLSetConnectOption() sets connection attributes for a particular connection.

Syntax

```
SQLRETURN   SQLSetConnectOption(
                             SQLHDBC          hdbc,
                             SQLUSMALLINT     fOption,
                             SQLUINTEGER      vParam);
```

Function Arguments

Table 112. SQLSetConnectOption Arguments

Data Type	Argument	Use	Description
HDBC	hdbc	Input	Connection handle.
SQLUSMALLINT	fOption	Input	Connect option to set, refer to Table 113 on page 299 for the complete list of connect options and their description.
SQLUINTEGER	vParam	Input	Value associated with *fOption*. Depending on the option, this can be a 32-bit integer value, or a pointer to a null-terminated string.

Usage

The SQLSetConnectOption() can be used to specify statement options for *all* statement handles on this connection, as well as for all future statement handles on this connection. For a list of statement options, refer to “SQLSetStmtOption - Set Statement Option” on page 315.

All connection and statement options set via the SQLSetConnectOption() persist until SQLFreeConnect() is called or the next SQLSetConnectOption() call.

It is illegal to call SQLSetConnectOption() (SQLSTATE **S1**010) if any of the statement handles associated with this connection is in a need data state (that is, in the middle of an SQLParamData() -- SQLPutData() sequence to process SQL_DATA_AT_EXEC parameters). This sequence is described in “Sending/Retrieving Long Data in Pieces” on page 351.

The format of information set with *vParam* depends on the specified *fOption*. The option information can be either a 32-bit integer or a pointer to a null-terminated character string. In the case of the null-terminated character string, the maximum length of the string can be SQL_MAX_OPTION_STRING_LENGTH bytes (excluding the null-terminator).

Table 113 (Page 1 of 3). Connect Options

fOption	**Contents**
SQL_ACCESS_MODE	A 32-bit integer value which can be either: • SQL_MODE_READ_ONLY: Indicates that the application is not performing any updates on data from this point on. Therefore, a less restrictive isolation level and locking can be used on transactions; that is, uncommitted read (SQL_TXN_READ_UNCOMMITTED). DB2 CLI does not ensure that requests to the database are *read-only*. If an update request is issued, DB2 CLI processes it using the transaction isolation level it selected as a result of the SQL_MODE_READ_ONLY setting. • **SQL_MODE_READ_WRITE**: Indicates that the application is making updates on data from this point on. DB2 CLI goes back to using the default transaction isolation level for this connection. SQL_MODE_READ_WRITE is the default. There must not be any outstanding transactions on this connection.
SQL_AUTOCOMMIT	A 32-bit integer value that specifies whether to use auto-commit or manual commit mode: • SQL_AUTOCOMMIT_OFF: The application must manually, explicitly commit or rollback transactions with `SQLTransact()` calls. • **SQL_AUTOCOMMIT_ON**: DB2 CLI operates in auto-commit mode. Each statement is implicitly committed. Each statement, that is not a query, is committed immediately after it has been executed. Each query is committed immediately after the associated cursor is closed. SQL_AUTOCOMMIT_ON is the default. **Note:** If this is a coordinated distributed unit of work connection, then the default is **SQL_AUTOCOMMIT_OFF** When specifying auto-commit, the application can have only one outstanding statement per connection. For example, there must not be two open cursors, or unpredictable results can occur. An open cursor must be closed before another query is executed. Since in many DB2 environments, the execution of the SQL statements and the commit can be flowed separately to the database server, autocommit can be expensive. It is recommended that the application developer take this into consideration when selecting the auto-commit mode. Changing from manual-commit to auto-commit mode commits any open transaction on the connection.

Table 113 (Page 2 of 3). Connect Options

fOption	**Contents**
SQL_CONNECTTYPE	A 32-bit integer value that specifies whether this application is to operate in a coordinated or uncoordinated distributed environment. If the processing needs to be coordinated, then this option must be considered in conjunction with the SQL_SYNC_POINT connection option. The possible values are: • **SQL_CONCURRENT_TRANS**: The application can have concurrent multiple connections to any one database or to multiple databases. This option setting corresponds to the specification of the Type 1 CONNECT in embedded SQL. Each connection has its own commit scope. No effort is made to enforce coordination of transaction. The current setting of the SQL_SYNC_POINT option is ignored. This is the default. • SQL_COORDINATED_TRANS: The application wishes to have commit and rollbacks coordinated among multiple database connections. This option setting corresponds to the specification of the Type 2 CONNECT in embedded SQL and must be considered in conjunction with the SQL_SYNC_POINT connection option. In contrast to the SQL_CONCURRENT_TRANS setting described above, the application is permitted only one open connection per database. **Note:** This connection type results in the default for SQL_AUTOCOMMIT connection option to be SQL_AUTOCOMMIT_OFF. This option must be set before making a connect request; otherwise, the `SQLSetConnectOption()` call is rejected. All the connections within an application must have the same SQL_CONNECTTYPE and SQL_SYNC_POINT values. The first connection determines the acceptable attributes for the subsequent connections. We recommend that the application set the SQL_CONNECTTYPE attribute at the environment level rather than on a per connection basis. ODBC applications written to take advantage of coordinated DB2 transactions must set these attributes at the connection level for each connection as `SQLSetEnvAttr()` is not supported in ODBC. **Note:** This is an IBM-defined extension.
SQL_CURRENT_SCHEMA	A null-terminated character string containing the name of the schema to be used by DB2 CLI for the `SQLColumns()` call if the *szSchemaName* pointer is set to null. To reset this option, specify this option with a zero length or a null pointer for the *vParam* argument. This option is useful when the application developer has coded a generic call to `SQLColumns()` that does not restrict the result set by schema name, but needs to constrain the result set at isolated places in the code. This option can be set at any time and is effective on the next *SQLColumns()* call where the szSchemaName pointer is null. **Note:** This is an IBM-defined extension.

Table 113 (Page 3 of 3). Connect Options

fOption	**Contents**
SQL_MAXCONN	A 32-bit integer value corresponding to the number of maximum concurrent connections that an application wants to set up. The default value is **0**, which means no maximum - the application is allowed to set up as many connections as the system resources permit. The integer value must be 0 or a positive number. This can be used as a governor for the maximum number of connections on a per application basis. The value that is in effect when the first connection is established is the value that is used. When the first connection is established, attempts to change this value are rejected. We recommend that the application set SQL_MAXCONN at the environment level rather then on a connection basis. ODBC applications must set this attribute at the connection level since `SQLSetEnvAttr()` is not supported in ODBC. **Note:** This is an IBM-defined extension.
SQL_PARAMOPT_ATOMIC	If specified, DB2 CLI returns `S1C00` on `SQLSetConnectOption` and `S1011` on `SQLGetConnectOption`.
SQL_TXN_ISOLATION	A 32-bit bitmask that sets the transaction isolation level for the current connection referenced by *hdbc*. The valid values for *vParam* can be determined at runtime by calling `SQLGetInfo()` with *fInfoType* set to SQL_TXN_ISOLATION_OPTIONS. The following values are accepted by DB2 CLI, but each server might only support a subset of these isolation levels: • SQL_TXN_READ_UNCOMMITTED - Dirty reads, reads that cannot be repeated, and phantoms are possible. • **SQL_TXN_READ_COMMITTED** - Dirty reads are not possible. Reads that cannot be repeated, and phantoms are possible. This is the default. • SQL_TXN_REPEATABLE_READ - Dirty reads and reads that cannot be repeated are not possible. Phantoms are possible. • SQL_TXN_SERIALIZABLE - Transactions can be serialized. Dirty reads, non-repeatable reads, and phantoms are not possible. • SQL_TXN_NOCOMMIT - Any changes are effectively committed at the end of a successful operation; no explicit commit or rollback is allowed. This is analogous to autocommit. This is not an SQL92 isolation level, but an IBM defined extension, supported only by DB2 for OS/400. In IBM terminology, • SQL_TXN_READ_UNCOMMITTED is uncommitted read; • SQL_TXN_READ_COMMITTED is cursor stability; • SQL_TXN_REPEATABLE_READ is read stability; • SQL_TXN_SERIALIZABLE is repeatable read. For a detailed explanation of isolation levels, refer to *IBM SQL Reference* This option cannot be specified while there is an open cursor on any hstmt, or an outstanding transaction for this connection; otherwise, SQL_ERROR is returned on the function call (SQLSTATE **S1**011). **Note:** There is an IBM extension that permits the setting of transaction isolation levels on a per statement handle basis. See the SQL_STMTTXN_ISOLATION option in the function description for `SQLSetStmtOption()`.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 114. SQLSetConnectOption SQLSTATEs

SQLSTATE	Description	Explanation
Note: Since `SQLSetConnectOption()` can also be used to set statement options, SQLSTATES for `SQLSetConnectOption()` can also include those listed under "Diagnostics' for the `SQLSetStmtOption()` API.		
01000	Warning.	Informational message indicating an internal commit has been issued on behalf of the application as part of the processing to set the specified connection option.
01S02	Option value changed	SQL_CONNECTTYPE changed to SQL_CONCURRENT_TRANS when MULTICONTEXT=1 in use.
08003	Connection is closed.	An *fOption* is specified that required an open connection.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	Given the *fOption* value, an invalid value is specified for the argument *vParam*.
S1010	Function sequence error.	The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1011	Operation invalid at this time.	The *fOption* specified option cannot be set at this time: • SQL_CONNECTTYPE, SQL_LONGDATA_COMPAT: attempt is made to change the value of these options from their current value but the connection is open. • SQL_TXN_ISOLATION, SQL_ACCESS_MODE: a transaction is outstanding.
S1092	Option type out of range.	An invalid *fOption* value is specified.
S1C00	Driver not capable.	The specified *fOption* is not supported. Given specified *fOption* value, the value specified for the argument *vParam* is not supported.

Restrictions

For compatibility with ODBC applications, *fOption* values of SQL_CURRENT_QUALIFIER and SQL_PACKET_SIZE are also recognized, but not supported. If either of these two options are specified, SQL_ERROR is returned on the function call (SQLSTATE **S1**C00).

ODBC *fOption* values of SQL_TRANSLATE_DLL and SQL_TRANSLATE_OPTION are not supported since DB2 handles codepage conversion at the server, not the client.

Example

Refer to “Example” on page 122.

References

- “SQLGetConnectOption - Returns Current Setting of A Connect Option” on page 185
- “SQLGetStmtOption - Returns Current Setting of A Statement Option” on page 236
- “SQLSetStmtOption - Set Statement Option” on page 315

SQLSetCursorName - Set Cursor Name

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLSetCursorName() associates a cursor name with the statement handle. This function is optional since DB2 CLI implicitly generates a cursor name when each statement handle is allocated.

Syntax

```
SQLRETURN   SQLSetCursorName (SQLHSTMT          hstmt,
                              SQLCHAR     FAR   *szCursor,
                              SQLSMALLINT       cbCursor);
```

Function Arguments

Table 115. SQLSetCursorName Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle
SQLCHAR *	*szCursor*	input	Cursor name
SQLSMALLINT	*cbCursor*	input	Length of contents of *szCursor* argument

Usage

DB2 CLI always generates and uses an internally generated cursor name when a query is prepared or executed directly. SQLSetCursorName() allows an application defined cursor name to be used in an SQL statement (a positioned UPDATE or DELETE). DB2 CLI maps this name to the internal name. The name remains associated with the statement handle, until the handle is dropped, or another SQLSetCursorName() is called on this statement handle.

Although SQLGetCursorName() returns the name set by the application (if one is set), error messages associated with positioned UPDATE and DELETE statements refer to the internal name. For this reason, we recommend that you do not use SQLSetCursorName(). Instead, use the internal name which can be obtained by calling SQLGetCursorName().

Cursor names must follow these rules:

- All cursor names within the connection must be unique.
- Each cursor name must be less than or equal to 18 bytes in length. Any attempt to set a cursor name longer than 18 bytes results in truncation of that cursor name to 18 bytes. (No warning is generated.)
- Since internally generated names begin with SQLCUR, SQL_CUR, or SQLCURQRS, the application must not input a cursor name starting with either SQLCUR or SQL_CUR in order to avoid conflicts with internal names.
- Since a cursor name is considered an identifier in SQL, it must begin with an English letter (a-z, A-Z) followed by any combination of digits (0-9), English letters or the underscore character (_).

- To permit cursor names containing characters other than those listed above (such as National Language Set or Double Bytes Character Set characters), the application must enclose the cursor name in double quotes (").
- Unless the input cursor name is enclosed in double quotes, all leading and trailing blanks from the input cursor name string are removed.

For efficient processing, applications should not include any leading or trailing spaces in the *szCursor* buffer. If the *szCursor* buffer contains a delimited identifier, applications should position the first double quote as the first character in the *szCursor* buffer.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 116. SQLSetCursorName SQLSTATEs

SQLSTATE	Description	Explanation
34000	Invalid cursor name.	The cursor name specified by the argument *szCursor* is invalid. The cursor name either begins with SQLCUR, SQL_CUR, or SQLCURQRS or violates the cursor naming rules (Must begin with a-z or A-Z followed by any combination of English letters, digits, or the '_' character. The cursor name specified by the argument *szCursor* already exists. The cursor name length is greater than the value returned by `SQLGetInfo()` with the SQL_MAX_CURSOR_NAME_LEN argument.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	*szCursor* is a null pointer.
S1010	Function sequence error.	There is an open or positioned cursor on the statement handle. The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation called prior to `SQLSetCursorName()`.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.
S1090	Invalid string or buffer length.	The argument *cbCursor* is less than **0**, but not equal to SQL_NTS.

Restrictions

None.

Example

```
/* ... */
    SQLCHAR         sqlstmt[] =
                    "SELECT name, job FROM staff "
                    "WHERE job='Clerk'  FOR UPDATE OF job";
/* ... */
    /* allocate second statement handle for update statement */
    rc2 = SQLAllocStmt(hdbc, &hstmt2);

    /* Set Cursor for the SELECT statement's handle */
    rc = SQLSetCursorName(hstmt1, "JOBCURS", SQL_NTS);

    rc = SQLExecDirect(hstmt1, sqlstmt, SQL_NTS);

    /* bind name to first column in the result set */
    rc = SQLBindCol(hstmt1, 1, SQL_C_CHAR, (SQLPOINTER) name.s, 10,
                    &name.ind);

    /* bind job to second column in the result set */
    rc = SQLBindCol(hstmt1, 2, SQL_C_CHAR, (SQLPOINTER) job.s, 6,
                    &job.ind);

    printf("Job Change for all clerks\n");

    while ((rc = SQLFetch(hstmt1)) == SQL_SUCCESS) {
        printf("Name: %-9.9s Job: %-5.5s \n", name.s, job.s);
        printf("Enter new job or return to continue\n");
        gets(newjob);
        if (newjob[0] != '\0') {
            sprintf(updstmt,
                    "UPDATE staff set job = '%s' where current of JOBCURS",
                    newjob);
            rc2 = SQLExecDirect(hstmt2, updstmt, SQL_NTS);
        }
    }
    if (rc != SQL_NO_DATA_FOUND)
        check_error(henv, hdbc, hstmt1, rc, __LINE__, __FILE__);
/* ... */
```

References

- “SQLGetCursorName - Get Cursor Name” on page 187

SQLSetEnvAttr - Set Environment Attribute

Purpose

Specification:		X/OPEN CLI	ISO CLI

SQLSetEnvAttr() sets an environment attribute for the current environment.

Syntax

```
SQLRETURN  SQLSetEnvAttr     (SQLHENV          henv,
                              SQLINTEGER       Attribute,
                              SQLPOINTER       Value,
                              SQLINTEGER       StringLength);
```

Function Arguments

Table 117. SQLSetEnvAttr Arguments

Data Type	Argument	Use	Description
SQLHENV	henv	Input	Environment handle.
SQLINTEGER	Attribute	Input	Environment attribute to set. Refer to Table 118 for the list of attributes and their descriptions.
SQLPOINTER	Value	Input	The desired value for *Attribute*.
SQLINTEGER	StringLength	Input	Length of *Value* in bytes if the attribute value is a character string; if *Attribute* does not denote a string, then DB2 CLI ignores *StringLength*.

Usage

When set, the attribute's value affects all connections in this environment.

The application can obtain the current attribute value by calling SQLGetEnvAttr().

Table 118 (Page 1 of 2). Environment Attributes

Attribute	**Contents**
SQL_ATTR_OUTPUT_NTS	A 32-bit integer value which controls the use of null-termination in output arguments. The possible values are: • **SQL_TRUE**: DB2 CLI uses null termination to indicate the length of output character strings. This is the default. • SQL_FALSE: DB2 CLI does not use null termination in output character strings. The CLI functions affected by this attribute are all functions called for the environment (and for any connections and statements allocated under the environment) that have character string parameters. This attribute can only be set when there are no connection handles allocated under this environment.

Table 118 (Page 2 of 2). Environment Attributes

Attribute	**Contents**
SQL_CONNECTTYPE	A 32-bit integer value that specifies whether this application is to operate in a coordinated or uncoordinated distributed environment. The possible values are: • **SQL_CONCURRENT_TRANS**: Each connection has its own commit scope. No effort is made to enforce coordination of transaction. If an application issues a commit using the environment handle on `SQLTransact()` and not all of the connections commit successfully, the application is responsible for recovery. This corresponds to CONNECT (Type 1) semantics subject to the restrictions described in "DB2 CLI Restrictions on the ODBC Connection Model" on page 25. This is the default. • SQL_COORDINATED_TRANS: The application wishes to have commit and rollbacks coordinated among multiple database connections. In contrast to the SQL_CONCURRENT_TRANS setting described above, the application is permitted only one open connection per database. This attribute must be set before allocating any connection handles, otherwise, the `SQLSetEnvAttr()` call is rejected. All the connections within an application must have the same SQL_CONNECTTYPE and SQL_SYNCPOINT values. This attribute can also be set using the `SQLSetConnectOption` function. We recommend that the application set the SQL_CONNECTTYPE attribute at the environment level rather than on a per connection basis. ODBC applications written to take advantage of coordinated DB2 transactions must set these attributes at the connection level for each connection using `SQLSetConnectOption()` as `SQLSetEnvAttr()` is not supported in ODBC. **Note:** This is an IBM-defined extension.
SQL_MAXCONN	A 32-bit integer value corresponding to the number that maximum concurrent connections that an application wants to set up. The default value is **0**, which means no maximum - the application is allowed to set up as many connections as the system resources permit. The integer value must be 0 or a positive number. This can be used as a governor for the maximum number of connections on a per application basis. The value that is in effect when the first connection is established is the value that is used. When the first connection is established, attempts to change this value are rejected. We recommend that the application set SQL_MAXCONN at the environment level rather then on a connection basis. ODBC applications must set this attribute at the connection level since `SQLSetEnvAttr()` is not supported in ODBC. **Note:** This is an IBM-defined extension.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 119. SQLSetEnvAttr SQLSTATEs

SQLSTATE	Description	Explanation
# **01**S02	Option value changed	SQL_CONNECTTYPE changed to SQL_CONCURRENT_TRANS when MULTICONTEXT=1 in use.
S1009	Invalid argument value.	Given the *fOption* value, an invalid value is specified for the argument *vParam*.
S1011	Operation invalid at this time.	Applications cannot set environment attributes while connection handles are allocated on the environment handle.
S1092	Option type out of range.	An invalid *Attribute* value is specified.
S1C00	Driver not capable.	The specified *Attribute* is not supported by DB2 CLI. Given specified *Attribute* value, the value specified for the argument *Value* is not supported.

Restrictions

None.

Example

See also, “Distributed Unit of Work Example” on page 347.

```
/* ... */
int
main()
{
    SQLHENV         henv;
    SQLRETURN       rc;
    SQLINTEGER      output_nts = SQL_TRUE;

    rc = SQLAllocEnv(&henv);    /* allocate an environment handle    */
    if (rc == SQL_SUCCESS)
    {   rc = SQLSetEnvAttr(henv, SQL_ATTR_OUTPUT_NTS, output_nts,
                           0);
    }

    rc = SQLFreeEnv(henv);

}
/* ... */
```

References

- “SQLGetEnvAttr - Returns Current Setting of An Environment Attribute” on page 206

SQLSetParam - Binds A Parameter Marker to a Buffer

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	

Note: In ODBC 2.0, this function has been replaced by SQLBindParameter(). Refer to the restrictions section below for details.

SQLSetParam() is used to associate (bind) parameter markers in an SQL statement to application variables (storage buffers), for all data types. In this case data is transferred from the application to the DBMS when SQLExecute() or SQLExecDirect() is called. Data conversion can occur as the data is transferred.

Syntax

```
SQLRETURN   SQLSetParam      (SQLHSTMT          hstmt,
                              SQLUSMALLINT      ipar,
                              SQLSMALLINT       fCType,
                              SQLSMALLINT       fSqlType,
                              SQLUINTEGER       cbParamDef,
                              SQLSMALLINT       ibScale,
                              SQLPOINTER        rgbValue,
                              SQLINTEGER  FAR   *pcbValue);
```

Function Arguments

Table 120 (Page 1 of 3). SQLSetParam Arguments

Data Type	Argument	Use	Description
SQLHSTMT	*hstmt*	input	Statement handle.
SQLUSMALLINT	*ipar*	input	Parameter marker number, ordered sequentially left to right, starting at 1.
SQLSMALLINT	*fCType*	input	C data type of argument. The following types are supported: • SQL_C_BINARY • SQL_C_BIT • SQL_C_CHAR • SQL_C_DATE • SQL_C_DBCHAR • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG • SQL_C_SHORT • SQL_C_TIME • SQL_C_TIMESTAMP • SQL_C_TINYINT Specifying SQL_C_DEFAULT causes data to be transferred from its default C data type for the type indicated in *fSqlType*. Refer to Table 3 on page 40 for more information.

Table 120 (Page 2 of 3). SQLSetParam Arguments

Data Type	Argument	Use	Description
SQLSMALLINT	*fSqlType*	input	SQL Data Type of column. The supported types are: • SQL_BINARY • SQL_CHAR • SQL_DATE • SQL_DECIMAL • SQL_DOUBLE • SQL_FLOAT • SQL_GRAPHIC • SQL_INTEGER • SQL_LONGVARBINARY • SQL_LONGVARCHAR • SQL_LONGVARGRAPHIC • SQL_NUMERIC • SQL_REAL • SQL_SMALLINT • SQL_TIME • SQL_TIMESTAMP • SQL_VARBINARY • SQL_VARCHAR • SQL_VARGRAPHIC
SQLUINTEGER	*cbParamDef*	input	Precision of the corresponding parameter marker. If *fSqlType* denotes: • A binary or single byte character string (for example, SQL_CHAR, SQL_BINARY), this is the maximum length in bytes for this parameter marker. • A double byte character string (for example, SQL_GRAPHIC), this is the maximum length in double-byte characters for this parameter. • SQL_DECIMAL, SQL_NUMERIC, this is the maximum decimal precision. • Otherwise, this argument is ignored.
SQLSMALLINT	*ibScale*	input	Scale of the corresponding parameter marker if *fSqlType* is SQL_DECIMAL or SQL_NUMERIC. If *fSqlType* is SQL_TIMESTAMP, this is the number of digits to the right of the decimal point in the character representation of a timestamp (for example, the scale of yyyy-mm-dd hh:mm:ss.fff is 3). Other than for the *fSqlType* values mentioned here, *ibScale* is ignored.
SQLPOINTER	*rgbValue*	input (deferred)	Pointer to the location which contains (when the statement is executed) the actual values for the associated parameter marker.

Table 120 (Page 3 of 3). SQLSetParam Arguments

Data Type	Argument	Use	Description
SQLINTEGER *	*pcbValue*	input (deferred)	Pointer to the location which contains (when the statement is executed) the length of the parameter marker value stored at *rgbValue*. To specify a null value for a parameter marker, this storage location must contain SQL_NULL_DATA. If *fCType* is SQL_C_CHAR, this storage location must contain either the exact length of the data stored at *rgbValue*, or SQL_NTS if the contents at *rgbValue* are null-terminated. If *fCType* indicates character data (explicitly, or implicitly using SQL_C_DEFAULT), and this pointer is set to NULL, it is assumed that the application always provides a null-terminated string in *rgbValue*. This also implies that this parameter marker never contains a null value. If *fSqlType* indicates a graphic data type, and the *fCType* is SQL_C_CHAR, the pointer to *pcbValue* can never be NULL and the contents of *pcbValue* can never hold SQL_NTS. In general for graphic data types, this length should be the number of octets that the double byte data occupies; therefore, the length should always be a multiple of 2. In fact, if the length is odd, then an error occurs when the statement is executed.

Usage

A parameter marker is represented by a "?" character in an SQL statement and is used to indicate a position in the statement where an application supplied value is to be substituted when the statement is executed. This value can be obtained from an application variable. SQLSetParam() (or SQLBindParameter()) is used to bind the application storage area to the parameter marker.

The application must bind a variable to each parameter marker in the SQL statement before executing the SQL statement. For this function, *rgbValue* and *pcbValue* are deferred arguments. The storage locations must be valid and contain input data values when the statement is executed. This means either keeping the SQLExecDirect() or SQLExecute() call in the same procedure scope as the SQLBindParameter() calls, or, these storage locations must be dynamically allocated or declared statically or globally.

SQLSetParam() can be called before SQLPrepare() if the columns in the result set are known, otherwise the attributes of the result set can be obtained after the statement is prepared.

Parameter markers are referenced by number (*icol*) and are numbered sequentially from left to right, starting at 1.

All parameters bound by this function remain in effect until SQLFreeStmt() is called with either the SQL_DROP or SQL_RESET_PARAMS option, or until SQLSetParam() is called again for the same parameter *ipar* number.

After the SQL statement is executed, and the results processed, the application can reuse the statement handle to execute a different SQL statement. If the parameter marker specifications are different (number of parameters, length or type), then `SQLFreeStmt()` should be called with SQL_RESET_PARAMS to reset or clear the parameter bindings.

The C buffer data type given by *fCType* must be compatible with the SQL data type indicated by *fSqlType*, or an error occurs.

An application can pass the value for a parameter either in the *rgbValue* buffer or with one or more calls to `SQLPutData()`. In the latter case, these parameters are data-at-execution parameters. The application informs DB2 CLI of a data-at-execution parameter by placing the SQL_DATA_AT_EXEC value in the *pcbValue* buffer. It sets the *rgbValue* input argument to a 32-bit value which is returned on a subsequent `SQLParamData()` call and can be used to identify the parameter position.

Since the data in the variables referenced by *rgbValue* and *pcbValue* is not verified until the statement is executed, data content or format errors are not detected or reported until `SQLExecute()` or `SQLExecDirect()` is called.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 121 (Page 1 of 2). SQLSetParam SQLSTATEs

SQLSTATE	Description	Explanation
07006	Invalid conversion.	The conversion from the data value identified by the *fCType* argument to the data type identified by the *fSqlType* argument is not a meaningful conversion. (For example, conversion from SQL_C_DATE to SQL_DOUBLE.)
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1003	Program type out of range.	The value specified by the argument *fCType* is not a valid data type or SQL_C_DEFAULT.
S1004	SQL data type out of range.	The value specified for the argument *fSqlType* is not a valid SQL data type.
S1009	Invalid argument value.	The argument *rgbValue* is a null pointer.
S1010	Function sequence error.	There is an open or positioned cursor on the statement handle. The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation, called prior to `SQLSetCursorName()`.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.

Table 121 (Page 2 of 2). SQLSetParam SQLSTATEs

SQLSTATE	Description	Explanation
S1093	Invalid parameter number.	The value specified for the argument *ipar* is less than 1 or greater than the maximum number of parameters supported by the server.
S1094	Invalid scale value.	The value specified for *fSqlType* is either SQL_DECIMAL or SQL_NUMERIC and the value specified for *ibScale* is less than 0 or greater than the value for the argument *cbParamDef* (precision). The value specified for *fSqlType* is SQL_C_TIMESTAMP and the value for *fSqlType* is either SQL_CHAR or SQL_VARCHAR and the value for *ibScale* is less than 0 or greater than 6.
S1104	Invalid precision value.	The value specified for *cbParamDef* is either less than 0 or greater than the permissible range for the *fSQLType*.
S1C00	Driver not capable.	DB2 CLI or the data source does not support the conversion specified by the combination of the value specified for the argument *fCType* and the value specified for the argument *fSqlType*. The value specified for the argument *fCType* or *fSqlType* is not supported by either DB2 CLI or the data source.

Restrictions

In ODBC 2.0, `SQLSetParam()` has replaced by `SQLBindParameter()`.

`SQLSetParam()` cannot be used to:

- Bind application variables to parameter markers in a stored procedure CALL statement.
- Bind arrays of application variables when *SQLParamOptions()* has been used to specify multiple input parameter values.

SQLBindParameter() should be used instead in both of the above situations.

Example

Refer to “Example” on page 264.

References

- “SQLBindParameter - Binds A Parameter Marker to a Buffer” on page 91
- “SQLExecDirect - Execute a Statement Directly” on page 149
- “SQLExecute - Execute a Statement” on page 154
- “SQLPrepare - Prepare a Statement” on page 261

SQLSetStmtOption - Set Statement Option

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	

SQLSetStmtOption() sets an attribute of a specific statement handle. To set an option for all statement handles associated with a connection handle, the application can call SQLSetConnectOption() (refer to "SQLSetConnectOption - Set Connection Option" on page 298).

Syntax

```
SQLRETURN   SQLSetStmtOption (SQLHSTMT         hstmt,
                              SQLUSMALLINT     fOption,
                              SQLUINTEGER      vParam);
```

Function Arguments

Table 122. SQLSetStmtOption Arguments

Data Type	Argument	Use	Description
SQLHSTMT	hstmt	input	Statement handle.
SQLUSMALLINT	fOption	input	Option to set. Refer to Table 123 on page 316 for the list of statement options that can be set and their descriptions.
SQLUINTEGER	vParam	input	Value associated with *fOption*. *vParam* can be a 32-bit integer value or a pointer to a null-terminated string.

Usage

Statement options for an *hstmt* remain in effect until they are changed by another call to SQLSetStmtOption() or SQLSetConnectOption(), or the hstmt is dropped by calling SQLFreeStmt() with the SQL_DROP option. Calling SQLFreeStmt() with the SQL_CLOSE, SQL_UNBIND, or SQL_RESET_PARAMS options does not reset statement options.

The format of *vParam* depends on the value specified *fOption*. The format of each is noted in Table 123 on page 316. If the format denotes a pointer to a null-terminated character string the maximum length is SQL_MAX_OPTION_STRING_LENGTH (excluding the null terminator).

Note: Currently no statement option requires a string.

Table 123 (Page 1 of 3). Statement Options

fOption	**Contents**
Note: Values shown in **bold** are default values.	
SQL_BIND_TYPE	A 32-bit integer value that sets the binding orientation to be used when `SQLExtendedFetch()` is called with this statement handle. *Column-wise binding* is selected by supplying the value **SQL_BIND_BY_COLUMN** for the argument *vParam*. *Row-wise binding* is selected by supplying a value for *vParam* specifying the length of the structure or an instance of a buffer into which result columns are bound. For row-wise binding, the length specified in *vParam* must include space for all of the bound columns and any padding of the structure or buffer to ensure that when the address of a bound column is incremented with the specified length, the result points to the beginning of the same column in the next row. (When using the sizeof operator with structures or unions in ANSI C, this behavior is guaranteed.)
SQL_CLOSE_BEHAVIOR	A 32-bit integer that forces the release of locks upon an underlying CLOSE CURSOR operation. The possible values are: • **SQL_CC_NO_RELEASE:** locks are not released when the cursor on this statement handle is closed. • SQL_CC_RELEASE: locks are released when the cursor on this statement handle is closed. Typically cursors are explicitly closed when the function `SQLFreeStmt()` is called with the SQL_CLOSE or SQL_DROP option. In addition, the end of the transaction (when a commit or rollback is issued) can also close the cursor (depending on the WITH HOLD attribute currently in use).
SQL_CONCURRENCY	If specified, DB2 CLI returns `S1C00` on `SQLSetConnectOption` and `S1011` on `SQLGetConnectOption`.
SQL_CURSOR_HOLD	A 32-bit integer which specifies whether the cursor associated with this hstmt is preserved in the same position as before the COMMIT operation, and whether the application can fetch without executing the statement again. • **SQL_CUSROR_HOLD_ON** • SQL_CURSOR_HOLD_OFF The default value when an hstmt is first allocated is SQL_CURSOR_HOLD_ON. This option cannot be specified while the re is an open cursor on this hstmt.
SQL_CURSOR_TYPE	A 32-bit integer value that specifies the cursor type. The currently supported value is: • **SQL_CURSOR_FORWARD_ONLY** - Cursor behaves as a forward only scrolling cursor. This option cannot be set if there is an open cursor on the associated *hstmt*. **Note:** ODBC has also defined the following values, which are not supported by Call Level Interface: • SQL_CURSOR_STATIC - The data in the result set appears to be static. • SQL_CURSOR_KEYSET_DRIVEN - The keys for the number of rows specified in the SQL_KEYSET_SIZE option is stored. DB2 CLI does not support this option value. • SQL_CURSOR_DYNAMIC - The keys for the rows in the rowset are saved. DB2 CLI does not support this option value. If one of these values is used, SQL_SUCCESS_WITH_INFO (SQLSTATE **01**S02) is returned and the value remains unchanged.

Table 123 (Page 2 of 3). Statement Options

fOption	**Contents**
SQL_MAX_LENGTH	A 32-bit integer value corresponding to the maximum amount of data that can be retrieved from a single character or binary column. If data is truncated because the value specified for SQL_MAX_LENGTH is less than the amount of data available, an SQLGetData() call or fetch returns SQL_SUCCESS instead of returning SQL_SUCCESS_WITH_INFO and SQLSTATE **01**004 (data truncated). The default value for *vParam* is **0**; 0 means that DB2 CLI attempts to return all available data for character or binary type data.
SQL_MAX_ROWS	A 32-bit integer value corresponding to the maximum number of rows to return to the application from a query. The default value for *vParam* is **0**; 0 means all rows are returned.
SQL_NODESCRIBE	A 32-bit integer which specifies whether DB2 CLI should automatically describe the column attributes of the result set or wait to be informed by the application via SQLSetColAttributes(). • **SQL_NODESCRIBE_OFF** • SQL_NODESCRIBE_ON This option cannot be specified while there is an open cursor on this hstmt. This option is used in conjunction with the function SQLSetColAttributes() by an application which has prior knowledge of the exact nature of the result set to be returned and which does not wish to incur the extra network traffic associated with the descriptor information needed by DB2 CLI to provide client side processing. **Note:** This option is an IBM-defined extension.
SQL_NOSCAN	A 32-bit integer value that specifies whether DB2 CLI will scan SQL strings for escape clauses. The two permitted values are: • **SQL_NOSCAN_OFF** - SQL strings are scanned for escape clause sequences. • SQL_NOSCAN_ON - SQL strings are not scanned for escape clauses. Everything is sent directly to the server for processing. This application can choose to turn off the scanning if it never uses vendor escape sequences in the SQL strings that it sends. This eliminates some of the overhead processing associated with scanning.
SQL_RETRIEVE_DATA	A 32-bit integer value indicating whether DB2 CLI should actually retrieve data from the database when SQLExtendedFetch() is called. The possible values are: • **SQL_RD_ON**: SQLExtendedFetch() does retrieve data. • SQL_RD_OFF: SQLExtendedFetch() does not retrieve data. This is useful for verifying whether rows exist without incurring the overhead of sending long data from the database server. DB2 CLI internally retrieves all the fixed length columns, such as integer and smallint; so there is still some overhead. This option cannot be set if the cursor is open.
SQL_ROWSET_SIZE	A 32-bit integer value that specifies the number of rows in the rowset. A rowset is the array of rows returned by each call to SQLExtendedFetch(). The default value is **1**, which is equivalent to making a single SQLFetch(). This option can be specified even when the cursor is open and becomes effective on the next SQLExtendedFetch() call.

Table 123 (Page 3 of 3). Statement Options

fOption	**Contents**
SQL_STMTTXN_ISOLATION SQL_TXN_ISOLATION	A 32-bit integer value that sets the transaction isolation level for the current hstmt. This overrides the default value set at the connection level (refer also to "SQLSetConnectOption - Set Connection Option" on page 298 for the permitted values). This option cannot be set if there is an open cursor on this statement handle (SQLSTATE **24**000). The value SQL_STMTTXN_ISOLATION is synonymous with SQL_TXN_ISOLATION. **Note:** It is an IBM extension to allow setting this option at the statement level.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 124. SQLSetStmtOption SQLSTATEs

SQLSTATE	Description	Explanation
01S02	Option value changed	A recognized concurrency value is specified for the SQL_CONCURRENCY option, but is not supported by DB2 CLI.
24000	Invalid cursor state.	*fOption* is set to SQL_CONCURRENCY, SQL_CURSOR_TYPE, SQL_STMTTXN_ISOLATION, or SQL_TXN_ISOLATION and a cursor is already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1000	General error.	An error occurred for which there is no specific SQLSTATE and for which no implementation defined SQLSTATE is defined. The error message returned by SQLError in the argument *szErrorMsg* describes the error and its cause.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1009	Invalid argument value.	Given the specified fOption value, an invalid value is specified for the argument vParam.
S1010	Function sequence error.	The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation; called prior to `SQLSetCursorName()`.
S1011	Operation invalid at this time.	The *fOption* is SQL_CONCURRENCY, SQL_CURSOR_HOLD, SQL_NODESCRIBE, SQL_RETRIEVE_DATA, SQL_(STMT)TXN_ISOLATION, or SQL_CURSOR_TYPE and the statement is prepared.
S1092	Option type out of range.	An invalid fOption value is specified.
S1C00	Driver not capable.	The option or option value is not supported.

Restrictions

ODBC also defines statement options SQL_KEYSET_SIZE, SQL_BOOKMARKS and SQL_SIMULATE_CURSOR. These options are not supported by DB2 CLI. If either one is specified, SQL_ERROR (SQLSTATE **S1**C00) is returned.

Example

Refer to “Example” on page 295.

References

- “SQLColAttributes - Get Column Attributes” on page 104
- “SQLExtendedFetch - Extended Fetch (Fetch Array of Rows)” on page 157
- “SQLFetch - Fetch Next Row” on page 164
- “SQLGetConnectOption - Returns Current Setting of A Connect Option” on page 185
- “SQLGetData - Get Data From a Column” on page 193
- “SQLGetStmtOption - Returns Current Setting of A Statement Option” on page 236
- “SQLParamOptions - Specify an Input Array for a Parameter” on page 259
- “SQLSetConnectOption - Set Connection Option” on page 298

SQLSpecialColumns - Get Special (Row Identifier) Columns

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	

SQLSpecialColumns() returns unique row identifier information (primary key or unique index) for a table. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN   SQLSpecialColumns(SQLHSTMT          hstmt,
                              SQLUSMALLINT      fColType,
                              SQLCHAR     FAR   *szCatalogName,
                              SQLSMALLINT       cbCatalogName,
                              SQLCHAR     FAR   *szSchemaName,
                              SQLSMALLINT       cbSchemaName,
                              SQLCHAR     FAR   *szTableName,
                              SQLSMALLINT       cbTableName,
                              SQLUSMALLINT      fScope,
                              SQLUSMALLINT      fNullable);
```

Function Arguments

Table 125 (Page 1 of 2). SQLSpecialColumns Arguments

Data Type	Argument	Use	Description
SQLHSTMT	hstmt	Input	Statement handle.
SQLUSMALLINT	fColType	Input	Type of unique row identifier to return. Only the following type is supported: • SQL_BEST_ROWID Returns the optimal set of columns that can uniquely identify any row in the specified table. **Note:** For compatibility with ODBC applications, SQL_ROWVER is also recognized, but not supported; therefore, if SQL_ROWVER is specified, an empty result is returned.
SQLCHAR *	szCatalogName	Input	Catalog qualifier of a 3 part table name. This must be a null pointer or a zero length string.
SQLSMALLINT	cbCatalogName	Input	Length of *szCatalogName*. This must be a set to 0.
SQLCHAR *	szSchemaName	Input	Schema qualifier of the specified table.
SQLSMALLINT	cbSchemaName	Input	Length of *szSchemaName*.
SQLCHAR *	szTableName	Input	Table name.
SQLSMALLINT	cbTableName	Input	Length of *cbTableName*.

Table 125 (Page 2 of 2). SQLSpecialColumns Arguments

Data Type	Argument	Use	Description
SQLUSMALLINT	fScope	Input	Minimum required duration for which the unique row identifier is valid. *fScope* must be one of the following: • SQL_SCOPE_CURROW: The row identifier is guaranteed to be valid only while positioned on that row. A later re-select using the same row identifier values might not return a row if the row was updated or deleted by another transaction. • SQL_SCOPE_TRANSACTION: The row identifier is guaranteed to be valid for the duration of the current transaction. **Note:** This option is only valid if SQL_TXN_SERIALIZABLE and SQL_TXN_REPEATABLE_READ isolation options are set. • SQL_SCOPE_SESSION: The row identifier is guaranteed to be valid for the duration of the connection. **Note:** This option is not supported by DB2 for OS/390. The duration over which a row identifier value is guaranteed to be valid depends on the current transaction isolation level. For information and scenarios involving isolation levels, refer to *SQL Reference*.
SQLUSMALLINT	fNullable	Input	Determines whether to return special columns that can have a NULL value. Must be one of the following: • SQL_NO_NULLS - The row identifier column set returned cannot have any NULL values. • SQL_NULLABLE - The row identifier column set returned can include columns where NULL values are permitted.

Usage

If multiple ways exist to uniquely identify any row in a table (that is, if there are multiple unique indexes on the specified table), then DB2 CLI returns the *best* set of row identifier column sets based on its internal criterion.

If there is no column set that allows any row in the table to be uniquely identified, an empty result set is returned.

The unique row identifier information is returned in the form of a result set where each column of the row identifier is represented by one row in the result set. Table 126 on page 322 shows the order of the columns in the result set returned by `SQLSpecialColumns()`, sorted by SCOPE.

Since calls to `SQLSpecialColumns()` in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set are declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call `SQLGetInfo()` with the SQL_MAX_COLUMN_NAME_LEN to determine the actual length of the COLUMN_NAME column supported by the connected DBMS.

Although new columns might be added and the names of the columns changed in future releases, the position of the current columns does not change.

Table 126 (Page 1 of 2). Columns Returned By SQLSpecialColumns

Column Number/Name	**Data Type**	**Description**
1 SCOPE	SMALLINT	The duration for which the name in COLUMN_NAME is guaranteed to point to the same row. Valid values are the same as for the *fScope* argument: Actual scope of the row identifier. Contains one of the following values: • SQL_SCOPE_CURROW • SQL_SCOPE_TRANSACTION • SQL_SCOPE_SESSION Refer to *fScope* in Table 125 on page 320 for a description of each value.
2 COLUMN_NAME	VARCHAR(128) NOT NULL	Name of the column that is (or part of) the table's primary key.
3 DATA_TYPE	SMALLINT NOT NULL	SQL data type of the column. One of the values in the Symbolic SQL Data Type column in Table 3 on page 40.
4 TYPE_NAME	VARCHAR(128) NOT NULL	DBMS character string represented of the name associated with DATA_TYPE column value.
5 COLUMN_SIZE	INTEGER	If the DATA_TYPE column value denotes a character or binary string, then this column contains the maximum length in bytes; if it is a graphic (DBCS) string, this is the number of double byte characters for the parameter. For date, time, timestamp data types, this is the total number of bytes required to display the value when converted to character. For numeric data types, this is either the total number of digits, or the total number of bits allowed in the column, depending on the value in the NUM_PREC_RADIX column in the result set. See Table 144 on page 420.
6 BUFFER_LENGTH	INTEGER	The maximum number of bytes for the associated C buffer to store data from this column if SQL_C_DEFAULT is specified on the `SQLBindCol()`, `SQLGetData()` and `SQLBindParameter()` calls. This length does not include any null-terminator. For exact numeric data types, the length accounts for the decimal and the sign. See Table 146 on page 422.
7 DECIMAL_DIGITS	SMALLINT	The scale of the column. NULL is returned for data types where scale is not applicable. See Table 145 on page 421.

Table 126 (Page 2 of 2). Columns Returned By SQLSpecialColumns

Column Number/Name	Data Type	Description
8 PSEUDO_COLUMN	SMALLINT	Indicates whether or not the column is a pseudo-column. DB2 CLI only returns: • SQL_PC_NOT_PSEUDO DB2 DBMSs do not support pseudo columns. ODBC applications can receive the following values from other non-IBM RDBMS servers: • SQL_PC_UNKNOWN • SQL_PC_PSEUDO

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 127. SQLSpecialColumns SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1014	No more handles.	DB2 CLI is not able to allocate a handle due to internal resources.
S1090	Invalid string or buffer length.	The value of one of the length arguments is less than 0, but not equal to SQL_NTS. The value of one of the length arguments exceeded the maximum length supported by the DBMS for that qualifier or name.
S1097	Column type out of range.	An invalid *fColType* value is specified.
S1098	Scope type out of range.	An invalid *fScope* value is specified.
S1099	Nullable type out of range.	An invalid *fNullable* values is specified.
S1C00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for table name.

Restrictions

None.

Example

```
/* ... */
SQLRETURN
list_index_columns(SQLHDBC hdbc, SQLCHAR *schema, SQLCHAR *tablename )
{
/* ... */
    rc = SQLSpecialColumns(hstmt, SQL_BEST_ROWID, NULL, 0, schema, SQL_NTS,
                    tablename, SQL_NTS, SQL_SCOPE_CURROW, SQL_NULLABLE);

    rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) column_name.s, 129,
                    &column_name.ind);

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) type_name.s, 129,
                    &type_name.ind);

    rc = SQLBindCol(hstmt, 5, SQL_C_LONG, (SQLPOINTER) & precision,
                    sizeof(precision), &precision_ind);

    rc = SQLBindCol(hstmt, 7, SQL_C_SHORT, (SQLPOINTER) & scale,
                    sizeof(scale), &scale_ind);

    printf("Primary Key or Unique Index for %s.%s\n", schema, tablename);
    /* Fetch each row, and display */
    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
        printf("   %s, %s ", column_name.s, type_name.s);
        if (precision_ind != SQL_NULL_DATA) {
            printf(" (%ld", precision);
        } else {
            printf("(\n");
        }
        if (scale_ind != SQL_NULL_DATA) {
            printf(", %d)\n", scale);
        } else {
            printf(")\n");
        }
    }
/* ... */
```

References

- “SQLColumns - Get Column Information for a Table” on page 115
- “SQLStatistics - Get Index and Statistics Information For A Base Table” on page 325
- “SQLTables - Get Table Information” on page 334

SQLStatistics - Get Index and Statistics Information For A Base Table

Purpose

Specification:	**ODBC** 1.0	**X/OPEN CLI**	

SQLStatistics() retrieves index information for a given table. It also returns the cardinality and the number of pages associated with the table and the indexes on the table. The information is returned in a result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN   SQLStatistics    (SQLHSTMT          hstmt,
                              SQLCHAR     FAR   *szCatalogName,
                              SQLSMALLINT       cbCatalogName,
                              SQLCHAR     FAR   *szSchemaName,
                              SQLSMALLINT       cbSchemaName,
                              SQLCHAR     FAR   *szTableName,
                              SQLSMALLINT       cbTableName,
                              SQLUSMALLINT      fUnique,
                              SQLUSMALLINT      fAccuracy);
```

Function Arguments

Table 128 (Page 1 of 2). SQLStatistics Arguments

Data Type	**Argument**	**Use**	**Description**
SQLHSTMT	hstmt	Input	Statement handle.
SQLCHAR *	szCatalogName	Input	Catalog qualifier of a 3 part table name. This must be a null pointer or a zero length string.
SQLSMALLINT	cbCatalogName	Input	Length of *cbCatalogName*. This must be set to 0.
SQLCHAR *	szSchemaName	Input	Schema qualifier of the specified table.
SQLSMALLINT	cbSchemaName	Input	Length of *szSchemaName*.
SQLCHAR *	szTableName	Input	Table name.
SQLSMALLINT	cbTableName	Input	Length of *cbTableName*.
SQLUSMALLINT	fUnique	Input	Type of index information to return: • SQL_INDEX_UNIQUE Only unique indexes are returned. • SQL_INDEX_ALL All indexes are returned.

Table 128 (Page 2 of 2). SQLStatistics Arguments

Data Type	Argument	Use	Description
SQLUSMALLINT	fAccuracy	Input	Indicate whether the CARDINALITY and PAGES columns in the result set contain the most current information: • SQL_ENSURE : This value is reserved for future use, when the application requests the most up to date statistics information. **New applications should not use this value**. Existing applications specifying this value receive the same results as SQL_QUICK. • SQL_QUICK : Statistics which are readily available at the server are returned. The values might not be current, and no attempt is made to ensure that they be up to date.

Usage

`SQLStatistics()` returns two types of information:

- Statistics information for the table (if it is available):
  - when the TYPE column in the table below is set to SQL_TABLE_STAT, the number of rows in the table and the number of pages used to store the table.
  - when the TYPE column indicates an index, the number of unique values in the index, and the number of pages used to store the indexes.
- Information about each index, where each index column is represented by one row of the result set. The result set columns are given in Table 129 in the order shown; the rows in the result set are ordered by NON_UNIQUE, TYPE, INDEX_QUALIFIER, INDEX_NAME and ORDINAL_POSITION.

Since calls to `SQLStatistics()` in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set are declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call `SQLGetInfo()` with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_OWNER_SCHEMA_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change.

Table 129 (Page 1 of 2). Columns Returned By SQLStatistics

Column Number/Name	Data Type	Description
1 TABLE_CAT	VARCHAR(128)	The is always null.
2 TABLE_SCHEM	VARCHAR(128)	The name of the schema containing TABLE_NAME.

Table 129 (Page 2 of 2). Columns Returned By SQLStatistics

Column Number/Name	Data Type	Description
3 TABLE_NAME	VARCHAR(128) NOT NULL	Name of the table.
4 NON_UNIQUE	SMALLINT	Indicates whether the index prohibits duplicate values: • SQL_TRUE if the index allows duplicate values. • SQL_FALSE if the index values must be unique. • NULL is returned if the TYPE column indicates that this row is SQL_TABLE_STAT (statistics information on the table itself).
5 INDEX_QUALIFIER	VARCHAR(128)	The string is used to qualify the index name in the DROP INDEX statement. Appending a period (.) plus the INDEX_NAME results in a full specification of the index.
6 INDEX_NAME	VARCHAR(128)	The name of the index. If the TYPE column has the value SQL_TABLE_STAT, this column has the value NULL.
7 TYPE	SMALLINT NOT NULL	Indicates the type of information contained in this row of the result set: • SQL_TABLE_STAT - Indicates this row contains statistics information on the table itself. • SQL_INDEX_CLUSTERED - Indicates this row contains information on an index, and the index type is a clustered index. • SQL_INDEX_HASHED - Indicates this row contains information on an index, and the index type is a hashed index. • SQL_INDEX_OTHER - Indicates this row contains information on an index, and the index type is other than clustered or hashed.
8 ORDINAL_POSITION	SMALLINT	Ordinal position of the column within the index whose name is given in the INDEX_NAME column. A NULL value is returned for this column if the TYPE column has the value of SQL_TABLE_STAT.
9 COLUMN_NAME	VARCHAR(128)	Name of the column in the index. A NULL value is returned for this column if the TYPE column has the value of SQL_TABLE_STAT.
10 ASC_OR_DESC	CHAR(1)	Sort sequence for the column; "**A**" for ascending, "**D**" for descending. NULL value is returned if the value in the TYPE column is SQL_TABLE_STAT.
11 CARDINALITY	INTEGER	• If the TYPE column contains the value SQL_TABLE_STAT, this column contains the number of rows in the table. • If the TYPE column value is not SQL_TABLE_STAT, this column contains the number of unique values in the index. • A NULL value is returned if information is not available from the DBMS.
12 PAGES	INTEGER	• If the TYPE column contains the value SQL_TABLE_STAT, this column contains the number of pages used to store the table. • If the TYPE column value is not SQL_TABLE_STAT, this column contains the number of pages used to store the indexes. • A NULL value is returned if information is not available from the DBMS.
13 FILTER_CONDITION	VARCHAR(128)	If the index is a filtered index, this is the filter condition. Since DATABASE 2 servers do not support filtered indexes, NULL is always returned. NULL is also returned if TYPE is SQL_TABLE_STAT.

For the row in the result set that contains table statistics (TYPE is set to SQL_TABLE_STAT), the columns values of NON_UNIQUE, INDEX_QUALIFIER, INDEX_NAME, ORDINAL_POSITION, COLUMN_NAME, and ASC_OR_DESC are set to NULL. If the CARDINALITY or PAGES information cannot be determined, then NULL is returned for those columns.

Note: The accuracy of the information returned in the SQLERRD(3) and SQLERRD(4) fields is dependent on many factors such as the use of parameter markers and expressions within the statement. The main factor which can be controlled is the accuracy of the database statistics. That is, when the statistics were last updated, (for example, for DB2 for OS/390, the last time the RUNSTATS utility was run.)

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 130. SQLStatistics SQLSTATEs

SQLSTATE	Description	Explanation
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1014	No more handles.	DB2 CLI is not able to allocate a handle due to internal resources.
S1090	Invalid string or buffer length.	The value of one of the name length arguments was less than 0, but not equal to SQL_NTS. The valid of one of the name length arguments exceeded the maximum value supported for that data source. The maximum supported value can be obtained by calling the `SQLGetInfo()` function.
S1100	Uniqueness option type out of range.	An invalid *fUnique* value was specified.
S1101	Accuracy option type out of range.	An invalid *fAccuracy* value was specified.
S1C00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for table name.

Restrictions

None.

Example

```
/* ... */
SQLRETURN
list_stats(SQLHDBC hdbc, SQLCHAR *schema, SQLCHAR *tablename )
{
/* ... */
    rc = SQLStatistics(hstmt, NULL, 0, schema, SQL_NTS,
                    tablename, SQL_NTS, SQL_INDEX_UNIQUE, SQL_QUICK);

    rc = SQLBindCol(hstmt, 4, SQL_C_SHORT,
                          &non_unique, 2, &non_unique_ind);

    rc = SQLBindCol(hstmt, 6, SQL_C_CHAR,
                          index_name.s, 129, &index_name.ind);

    rc = SQLBindCol(hstmt, 7, SQL_C_SHORT,
                          &type, 2, &type_ind);

    rc = SQLBindCol(hstmt, 9, SQL_C_CHAR,
                          column_name.s, 129, &column_name.ind);

    rc = SQLBindCol(hstmt, 11, SQL_C_LONG,
                          &cardinality, 4, &card_ind);

    rc = SQLBindCol(hstmt, 12, SQL_C_LONG,
                          &pages, 4, &pages_ind);

    printf("Statistics for %s.%s\n", schema, tablename);

    while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS)
    {  if (type != SQL_TABLE_STAT)
       {   printf("  Column: %-18s Index Name: %-18s\n",
                  column_name.s, index_name.s);
       }
       else
       {   printf("  Table Statistics:\n");
       }
       if (card_ind != SQL_NULL_DATA)
          printf("    Cardinality = %13ld", cardinality);
       else
          printf("    Cardinality = (Unavailable)");

       if (pages_ind != SQL_NULL_DATA)
          printf(" Pages = %13ld\n", pages);
       else
          printf(" Pages = (Unavailable)\n");
    }
/* ... */
```

References

- “SQLColumns - Get Column Information for a Table” on page 115
- “SQLSpecialColumns - Get Special (Row Identifier) Columns” on page 320

SQLTablePrivileges - Get Privileges Associated With A Table

Purpose

Specification:	ODBC 1.0		

SQLTablePrivileges() returns a list of tables and associated privileges for each table. The information is returned in an SQL result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN SQLTablePrivileges (SQLHSTMT          hstmt,
                              SQLCHAR     FAR   *szCatalogName,
                              SQLSMALLINT       cbCatalogName,
                              SQLCHAR     FAR   *szSchemaName,
                              SQLSMALLINT       cbSchemaName,
                              SQLCHAR     FAR   *szTableName,
                              SQLSMALLINT       cbTableName);
```

Function Arguments

Table 131. SQLTablePrivileges Arguments

Data Type	Argument	Use	Description
SQLHSTMT	hstmt	Input	Statement handle.
SQLCHAR *	szTableQualifier	Input	Catalog qualifier of a 3 part table name. This must be a null pointer or a zero length string.
SQLSMALLINT	cbTableQualifier	Input	Length of *szCatalogName*. This must be set to 0.
SQLCHAR *	szSchemaName	Input	Buffer that can contain a *pattern-value* to qualify the result set by schema name.
SQLSMALLINT	cbSchemaName	Input	Length of *szSchemaName*.
SQLCHAR *	szTableName	Input	Buffer that can contain a *pattern-value* to qualify the result set by table name.
SQLSMALLINT	cbTableName	Input	Length of *szTableName*.

The *szSchemaName* and *szTableName* arguments accept search pattern. For more information about valid search patterns, refer to “Input Arguments on Catalog Functions” on page 349.

Usage

The results are returned as a standard result set containing the columns listed in the following table. The result set is ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and PRIVILEGE. If multiple privileges are associated with any given table, each privilege is returned as a separate row.

Since calls to SQLTablePrivileges() in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set are declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call `SQLGetInfo()` with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_OWNER_SCHEMA_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change.

Table 132. Columns Returned By SQLTablePrivileges

Column Number/Name	**Data Type**	**Description**
1 TABLE_CAT	VARCHAR(128)	The is always null.
2 TABLE_SCHEM	VARCHAR(128)	The name of the schema contain TABLE_NAME.
3 TABLE_NAME	VARCHAR(128) NOT NULL	The name of the table.
4 GRANTOR	VARCHAR(128)	Authorization ID of the user who granted the privilege.
5 GRANTEE	VARCHAR(128)	Authorization ID of the user to whom the privilege is granted.
6 PRIVILEGE	VARCHAR(128)	The table privilege. This can be one of the following strings: • ALTER • CONTROL • DELETE • INDEX • INSERT • REFERENCES • SELECT • UPDATE
7 IS_GRANTABLE	VARCHAR(3)	Indicates whether the grantee is permitted to grant the privilege to other users. This can be "YES", "NO" or NULL.

Note: The column names used by DB2 CLI follow the X/Open CLI CAE specification style. The column types, contents and order are identical to those defined for the `SQLProcedures()` result set in ODBC.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 133 (Page 1 of 2). SQLTablePrivileges SQLSTATEs

SQLSTATE	**Description**	**Explanation**
24000	Invalid cursor state.	A cursor is already opened on the statement handle.

Table 133 (Page 2 of 2). SQLTablePrivileges SQLSTATEs

SQLSTATE	Description	Explanation
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1014	No more handles.	DB2 CLI is not able to allocate a handle due to internal resources.
S1090	Invalid string or buffer length.	The value of one of the name length arguments is less than 0, but not equal to SQL_NTS. The value of one of the name length arguments exceeded the maximum value supported for that data source. The maximum supported value can be obtained by calling the `SQLGetInfo()` function.
S1C00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for table name.

Restrictions

None.

Example

```
/* ... */
SQLRETURN
list_table_privileges(SQLHDBC hdbc, SQLCHAR *schema,
                       SQLCHAR *tablename )
{
    SQLHSTMT        hstmt;
    SQLRETURN       rc;
    struct { SQLINTEGER ind;   /* Length & Indicator variable */
             SQLCHAR  s[129];  /* String variable */
           } grantor, grantee, privilege;

    struct { SQLINTEGER  ind;
             SQLCHAR     s[4];
            }is_grantable;

    SQLCHAR         cur_name[512] = "";  /* Used when printing the */
    SQLCHAR         pre_name[512] = "";  /* Result set */

    /* Allocate a statment handle to reference the result set */
    rc = SQLAllocStmt(hdbc, &hstmt);

    /* Create Table Privilges result set */
    rc = SQLTablePrivileges(hstmt, NULL, 0, schema, SQL_NTS,
                             tablename, SQL_NTS);

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) grantor.s, 129,
                    &grantor.ind);

/* Continue Binding, then fetch and display result set */
/* ... */
```

References

- “SQLTables - Get Table Information” on page 334

SQLTables - Get Table Information

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	

SQLTables() returns a list of table names and associated information stored in the system catalog of the connected data source. The list of table names is returned as a result set, which can be retrieved using the same functions that are used to process a result set generated by a query.

Syntax

```
SQLRETURN   SQLTables        (SQLHSTMT          hstmt,
                              SQLCHAR     FAR   *szCatalogName,
                              SQLSMALLINT       cbCatalogName,
                              SQLCHAR     FAR   *szSchemaName,
                              SQLSMALLINT       cbSchemaName,
                              SQLCHAR     FAR   *szTableName,
                              SQLSMALLINT       cbTableName,
                              SQLCHAR     FAR   *szTableType,
                              SQLSMALLINT       cbTableType);
```

Function Arguments

Table 134. SQLTables Arguments

Data Type	Argument	Use	Description
SQLHSTMT	hstmt	Input	Statement handle.
SQLCHAR *	szCatalogName	Input	Buffer that can contain a *pattern-value* to qualify the result set. *Catalog* is the first part of a 3 part table name. This must be a NULL pointer or a zero length string.
SQLSMALLINT	cbCatalogName	Input	Length of *szCatalogName*. This must be set to 0.
SQLCHAR *	szSchemaName	Input	Buffer that can contain a *pattern-value* to qualify the result set by schema name.
SQLSMALLINT	cbSchemaName	Input	Length of *szSchemaName*.
SQLCHAR *	szTableName	Input	Buffer that can contain a *pattern-value* to qualify the result set by table name.
SQLSMALLINT	cbTableName	Input	Length of *szTableName*.
SQLCHAR *	szTableType	Input	Buffer that can contain a *value list* to qualify the result set by table type. The value list is a list of upper-case comma-separated single quoted values for the table types of interest. Valid table type identifiers can include: TABLE, VIEW, SYSTEM TABLE, ALIAS, SYNONYM. If *szTableType* argument is a NULL pointer or a zero length string, then this is equivalent to specifying all of the possibilities for the table type identifier. If SYSTEM TABLE is specified, then both system tables and system views (if there are any) are returned.
SQLSMALLINT	cbTableType	Input	Size of *cbTableType*

Note that the *szCatalogName, szSchemaName*, and *szTableName* arguments accept search patterns. For more information about valid search patterns, refer to “Input Arguments on Catalog Functions” on page 349.

Usage

Table information is returned in a result set where each table is represented by one row of the result set. To determine the type of access permitted on any given table in the list, the application can call SQLTablePrivileges(). Otherwise, the application must be able to handle a situation where the user selects a table for which SELECT privileges are not granted.

To support obtaining just a list of schemas, the following special semantics for the *szSchemaName* argument can be applied: if *szSchemaName* is a string containing a single percent (%) character, and *szCatalogName* and *szTableName* are empty strings, then the result set contains a list of valid schemas in the data source.

If *szTableType* is a single percent character (%) and *szCatalogName, szSchemaName*, and *szTableName* are empty strings, then the result set contains a list of valid table types for the data source. (All columns except the TABLE_TYPE column contain NULLs.)

If *szTableType* is not an empty string, it must contain a list of upper-case, comma-separated values for the types of interest; each value can be enclosed in single quotes or unquoted. For example, "'TABLE','VIEW'" or "TABLE,VIEW". If the data source does not support or does not recognize a specified table type, nothing is returned for that type.

Sometimes, an application calls SQLTables() with null pointers for some or all of the *szSchemaName, szTableName*, and *szTableType* arguments so that no attempt is made to restrict the result set returned. For some data sources that contain a large quantity of tables, views, or aliases, this scenario maps to an extremely large result set and very long retrieval times. Three mechanisms are introduced to help the end user reduce the long retrieval times: three keywords (SCHEMALIST, SYSCHEMA, TABLETYPE) can be specified in the CLI initialization file to help restrict the result set when the application has supplied null pointers for either or both of *szSchemaName* and *szTableType*. These keywords and their usage are discussed in detail in “Initialization Keywords” on page 64. If the application did not specify a null pointer for *szSchemaName* or *szTableType* then the associated keyword specification in the CLI initialization file is ignored.

The result set returned by SQLTables() contains the columns listed in Table 135 on page 336 in the order given. The rows are ordered by TABLE_TYPE, TABLE_CAT, TABLE_SCHEM, and TABLE_NAME.

Since calls to SQLTables() in many cases map to a complex and thus expensive query against the system catalog, they should be used sparingly, and the results saved rather than repeating calls.

The VARCHAR columns of the catalog functions result set are declared with a maximum length attribute of 128 to be consistent with SQL92 limits. Since DB2 names are less than 128, the application can choose to always set aside 128 characters (plus the null-terminator) for the output buffer, or alternatively, call SQLGetInfo() with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_OWNER_SCHEMA_LEN, SQL_MAX_TABLE_NAME_LEN, and

SQL_MAX_COLUMN_NAME_LEN to determine respectively the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns supported by the connected DBMS.

Although new columns might be added and the names of the existing columns changed in future releases, the position of the current columns does not change.

Table 135. Columns Returned By SQLTables

Column Name	**Data Type**	**Description**
TABLE_CAT	VARCHAR(128)	The name of the catalog containing TABLE_SCHEM. This column contains a NULL value.
TABLE_SCHEM	VARCHAR(128)	The name of the schema containing TABLE_NAME.
TABLE_NAME	VARCHAR(128)	The name of the table, or view, or alias, or synonym.
TABLE_TYPE	VARCHAR(128)	Identifies the type given by the name in the TABLE_NAME column. It can have the string values 'TABLE', 'VIEW', 'INOPERATIVE VIEW', 'SYSTEM TABLE', 'ALIAS', or 'SYNONYM'.
REMARKS	VARCHAR(254)	Contains the descriptive information about the table.

Return Codes

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 136. SQLTables SQLSTATEs

SQLSTATE	**Description**	**Explanation**
24000	Invalid cursor state.	A cursor is already opened on the statement handle.
40003 **08**S01	Communication link failure.	The communication link between the application and data source fails before the function completes.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1010	Function sequence error.	The function is called while in a data-at-execute (`SQLParamData()`, `SQLPutData()`) operation.
S1014	No more handles.	DB2 CLI is not able to allocate a handle due to internal resources.
S1090	Invalid string or buffer length.	The value of one of the name length arguments is less than 0, but not equal to SQL_NTS. The value of one of the name length arguments exceeded the maximum value supported for that data source. The maximum supported value can be obtained by calling the `SQLGetInfo()` function.
S1C00	Driver not capable.	DB2 CLI does not support *catalog* as a qualifier for table name.

Restrictions

None.

Example

Also, refer to “Querying Environment Information Example” on page 46.

```
/* ... */
SQLRETURN init_tables(SQLHDBC hdbc )
{
    SQLHSTMT        hstmt;
    SQLRETURN       rc;

    SQLUSMALLINT     rowstat[MAX_TABLES];
    SQLUINTEGER      pcrow;

    rc = SQLAllocStmt(hdbc, &hstmt);

    /* SQL_ROWSET_SIZE sets the max number of result rows to fetch each time */
    rc = SQLSetStmtOption(hstmt, SQL_ROWSET_SIZE, MAX_TABLES);

    /* Set Size of One row, Used for Row-Wise Binding Only */
    rc = SQLSetStmtOption(hstmt, SQL_BIND_TYPE, sizeof(table) / MAX_TABLES);

    printf("Enter Search Pattern for Table Schema Name:\n");
    gets(table->schem);
    printf("Enter Search Pattern for Table Name:\n");
    gets(table->name);
    rc = SQLTables(hstmt, NULL, 0, table->schem, SQL_NTS,
                   table->name, SQL_NTS, NULL, 0);

    rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, (SQLPOINTER) &table->schem, 129,
                    &table->schem_l);

    rc = SQLBindCol(hstmt, 3, SQL_C_CHAR, (SQLPOINTER) &table->name, 129,
                    &table->name_l);

    rc = SQLBindCol(hstmt, 4, SQL_C_CHAR, (SQLPOINTER) &table->type, 129,
                    &table->type_l);

    rc = SQLBindCol(hstmt, 5, SQL_C_CHAR, (SQLPOINTER) &table->remarks, 255,
                    &table->remarks_l);

    /* Now fetch the result set */
/* ... */
```

References

- “SQLColumns - Get Column Information for a Table” on page 115
- “SQLTablePrivileges - Get Privileges Associated With A Table” on page 330

SQLTransact - Transaction Management

Purpose

Specification:	ODBC 1.0	X/OPEN CLI	ISO CLI

SQLTransact() commits or rolls back the current transaction in the specified connection. SQLTransact() can also be used to request that a commit or rollback be issued for each of the connections associated with the environment.

All changes to the database performed on the connection since connect time or the previous call to SQLTransact() (whichever is the most recent) are committed or rolled back.

If a transaction is active on a connection, the application must call SQLTransact() before it can disconnect from the database.

Syntax

```
SQLRETURN   SQLTransact      (SQLHENV          henv,
                              SQLHDBC          hdbc,
                              SQLUSMALLINT     fType);
```

Function Arguments

Table 137. SQLTransact Arguments

Data Type	Argument	Use	Description
SQLHENV	*henv*	input	Environment handle. If *hdbc* is a valid connection handle, *henv* is ignored.
SQLHDBC	*hdbc*	input	Database connection handle. If *hdbc* is set to SQL_NULL_HDBC, then *henv* must contain the environment handle that the connection is associated with.
SQLUSMALLINT	*fType*	input	The desired action for the transaction. The value for this argument must be one of: • SQL_COMMIT • SQL_ROLLBACK

Usage

In DB2 CLI, a transaction begins implicitly when an application that does not already have an active transaction, issues SQLPrepare(), SQLExecDirect(), SQLExecDirect(), SQLGetTypeInfo(), or one of the catalog functions. The transaction ends when the application calls SQLTransact() or disconnects from the data source.

If the input connection handle is SQL_NULL_HDBC and the environment handle is valid, then a commit or rollback is issued on each of the open connections in the environment. SQL_SUCCESS is returned only if success is reported on all the connections. If the commit or rollback fails for one or more of the connections, SQLTransact() returns SQL_ERROR. To determine which connections failed the commit or rollback operation, the application needs to call `SQLError()` on each connection handle in the environment.

It is important to note that unless the connection option SQL_CONNECTTYPE is set to SQL_COORDINATED_TRANS (to indicate coordinated distributed transactions), there is no attempt to provide coordinated global transaction with one-phase or two-phase commit protocols.

Completing a transaction has the following effects:

- Prepared SQL statements (via `SQLPrepare()`) survive transactions; they can be executed again without first calling `SQLPrepare()`.
- Cursor positions are maintained after a commit unless one or more of the following is true:
  - The server is SQL/DS.
  - The SQL_CURSOR_HOLD statement option for this handle is set to SQL_CURSOR_HOLD_OFF.
  - The CURSORHOLD keyword in the DB2 CLI initialization file is set so that cursor with hold is not in effect and this has not been overridden by resetting the SQL_CURSOR_HOLD statement option.
  - The CURSORHOLD keyword is present in a the connection string on the `SQLDriverConnect()` call that set up this connection, and it indicates cursor with hold is not in effect, and this has not been overridden by resetting the SQL_CURSOR_HOLD statement option.

  If the cursor position is not maintained due to any one of the above circumstances, the cursor is closed and all pending results are discarded.

  If the cursor position is maintained after a commit, the application must issue a fetch to re-position the cursor (to the next row) before continuing with processing of the remaining result set.

  To determine whether cursor position is maintained after a commit, call `SQLGetInfo()` with the SQL_CURSOR_COMMIT_BEHAVIOR information type.
- Cursors are closed after a rollback and all pending results are discarded.
- Statement handles are still valid after a call to `SQLTransact()`, and can be reused for subsequent SQL statements or de-allocated by calling `SQLFreeStmt()`.
- Cursor names, bound parameters, and column bindings survive transactions.

If no transaction is currently active on the connection, calling `SQLTransact()` has no effect on the database server and returns SQL_SUCCESS.

`SQLTransact()` can fail while executing the COMMIT or ROLLBACK due to a loss of connection. In this case the application might not be able to determine whether the COMMIT or ROLLBACK was processed, and a database administrator's help might be required. Refer to the DBMS product information for more information on transaction logs and other transaction management tasks.

Return Codes

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

Table 138. SQLTransact SQLSTATEs

SQLSTATE	Description	Explanation
08003	Connection is closed.	The *hdbc* is not in a connected state.
08007	Connection failure during transaction.	The connection associated with the *hdbc* failed during the execution of the function and it cannot be determined whether the requested COMMIT or ROLLBACK occurred before the failure.
58004	Unexpected system failure.	Unrecoverable system error.
S1001	Memory allocation failure.	DB2 CLI is not able to allocate memory required to support execution or completion of the function.
S1012	Invalid transaction code.	The value specified for the argument *fType* was neither SQL_COMMIT not SQL_ROLLBACK.
S1013	Unexpected memory handling error.	DB2 CLI is not able to access memory required to support execution or completion of the function.

Restrictions

| `SQLTransact()` can not be issued if the application is executing as a stored
| procedure.

Example

Refer to “Example” on page 167.

References

- “SQLSetStmtOption - Set Statement Option” on page 315
- “SQLGetInfo - Get General Information” on page 213

Chapter 6. Using Advanced Features

This section covers a series of advanced tasks.

Environment, Connection, and Statement Options

Environments, connections, and statements each have a defined set of options (or attributes). All attributes can be queried by the application, but only some attributes can be changed from their default values. By changing attribute values, the application can change the behavior of DB2 CLI.

An environment handle has attributes which affect the behavior of DB2 CLI functions under that environment. The application can specify the value of an attribute by calling `SQLSetEnvAttr()` and can obtain the current attribute value by calling `SQLGetEnvAttr()`. `SQLSetEnvAttr()` can only be called before connection handles have been allocated.

A connection handle has options which affect the behavior of DB2 CLI functions under that connection. Of the options that can be changed:

- Some can be set any time after the connection handle is allocated.
- Some can be set only before the actual connection is established.
- Some can be set only after the connection is established.
- Some can be set after the connection is established, but only while there are no outstanding transactions or open cursors.

The application can change the value of connection options by calling `SQLSetConnectOption()` and can obtain the current value of an option by calling `SQLGetConnectOption()`. An example of a connection option which can be set any time after a handle is allocated is the auto-commit option introduced in "Commit or Rollback" on page 34. For complete details on when each option can be set, refer to "SQLSetConnectOption - Set Connection Option" on page 298.

A statement handle has options which affect the behavior of CLI functions executed using that statement handle. Of the statement options that can be changed:

- Some options can be set, but currently can be set to only one specific value.
- Some options can be set any time after the statement handle is allocated.
- Some options can only be set if there is no open cursor on that statement handle.

The application can specify the value of any settable statement option by calling `SQLSetStmtOption()`, and can obtain the current value of an option by calling

SQLGetStmtOption(). For complete details on when each option can be set, refer to “SQLSetStmtOption - Set Statement Option” on page 315.

The SQLSetConnectOption() function can also be used to set statement options for **all** statement handles currently associated with the connection as well as for all future statement handles to be allocated under this connection. However, SQLGetConnectOption() can only be used to obtain connection options; SQLGetStmtOption() must be used to obtain the current value of a statement option.

Many applications use just the default option settings; however, there can be situations where some of these defaults are not suitable for a particular user of the application. DB2 CLI provides end users with two methods to change some of these default values at run time. The first method is to specify the new default attribute values in the connection string input to the SQLDriverConnect() function. The second method involves the specification of the new default attribute values in a DB2 CLI initialization file.

The DB2 CLI initialization file can be used to change default values for all DB2 CLI applications. This might be the end user's only means of changing the defaults if the application does not have a way for the user to provide default attribute values in the SQLDriverConnect() connection string. Default attribute values that are specified on SQLDriverConnect() override the values in the DB2 CLI initialization file for that particular connection. For information on how the end user can use the DB2 CLI initialization file as well as for a list of changeable defaults, refer to “DB2 CLI Initialization File” on page 62.

The mechanisms for changing defaults are intended for end user tuning; application developers must use the appropriate set-option function. If an application does call a set-option or attribute function with a value different from the initialization file or the connection string specification, then the initial default value is overridden and the new value takes effect.

The options that can be changed, are listed in the detailed function descriptions of the *set* option or attributes functions, see “Chapter 5. Functions” on page 73. The read-only options (if any exist) are listed with the detailed function descriptions of the *get* option or attribute functions.

For information on some commonly used options, refer to Appendix A, “Programming Hints and Tips” on page 393.

Figure 8 on page 343 shows the addition of the option or attribute functions to the basic connect scenario.

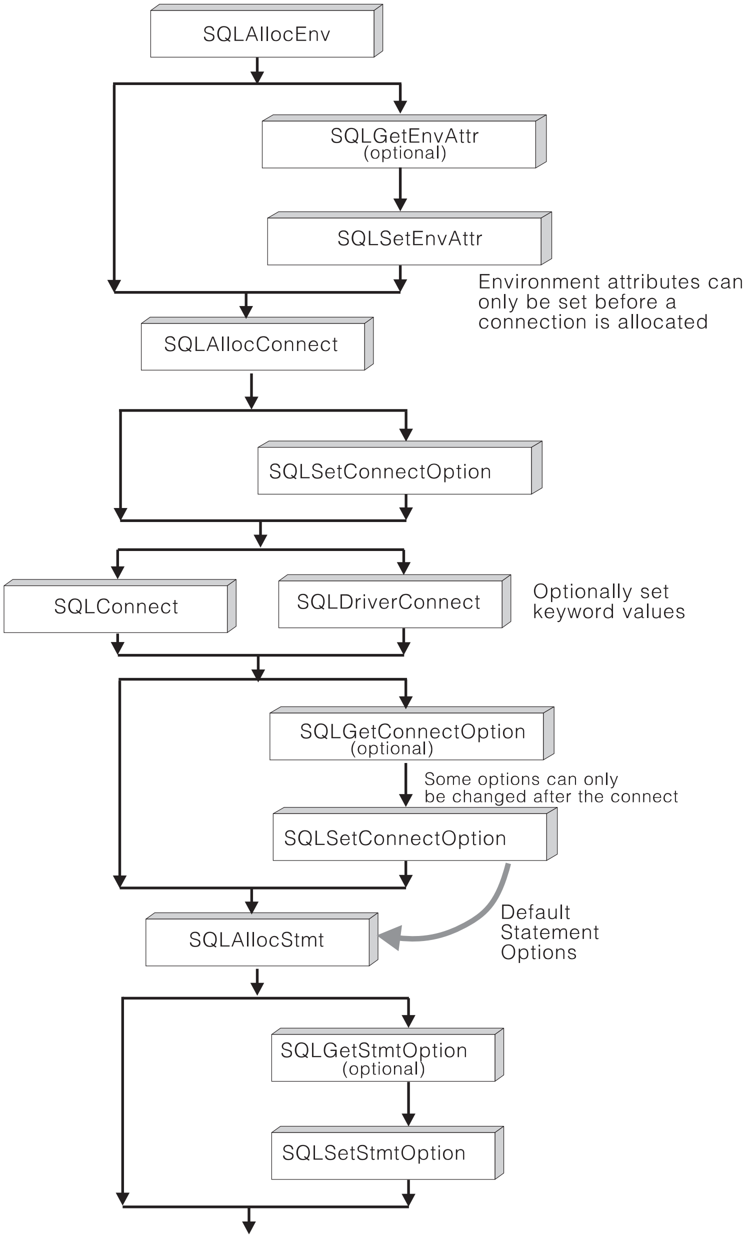

Figure 8. Setting and Retrieving Options (Attributes)

Distributed Unit of Work (Coordinated Distributed Transactions)

The transaction scenario described in “Connecting to One or More Data Sources” on page 26 portrays an application which interacts with only one database server in a transaction. Further, only one transaction (that associated with the current server) existed at any given time.

With distributed unit of work (coordinated distributed transactions), the application, if executing CONNECT (Type 2), is able to access multiple database servers from within the same coordinated transaction. This section describes how DB2 CLI applications can be written to use coordinated distributed unit of work.

First, consider the environment attribute (SQL_CONNECTTYPE) which controls whether the application is to operate in a coordinated or uncoordinated distributed environment. The two possible values for this attribute are:

- SQL_CONCURRENT_TRANS - supports the single data source per transaction semantics described in Chapter 2. Multiple (logical) concurrent connections to different data sources are permitted. This is the default.
- SQL_COORDINATED_TRANS - supports the multiple data sources per transaction semantics, as discussed below.

All connections within an application must have the same SQL_CONNECTTYPE setting. It is recommended that the application set this environment attribute, if necessary, as soon as `SQLAllocEnv()` is called successfully.

Options that Govern Distributed Unit of Work Semantics

A coordinated transaction means that commits or rollbacks among multiple data source connections are coordinated. The SQL_COORDINATED_TRANS setting of the SQL_CONNECTTYPE attribute corresponds to the CONNECT (Type 2) in IBM embedded SQL.

All the connections within an application must have the same SQL_CONNECTTYPE setting. After the first connection is established, all subsequent connect types must be the same as the first. Coordinated connections default to manual-commit mode (for discussion on auto-commit mode, see “Commit or Rollback” on page 34).

Figure 9 on page 345 shows the logical flow of an application executing statements on two SQL_CONCURRENT_TRANS connections (`'A'` and `'B'`), and indicates the scope of the transactions.

Figure 10 on page 346 shows the same statements being executed on two SQL_COORDINATED_TRANS connections ('A' and 'B'), and the scope of a coordinated distributed transaction.

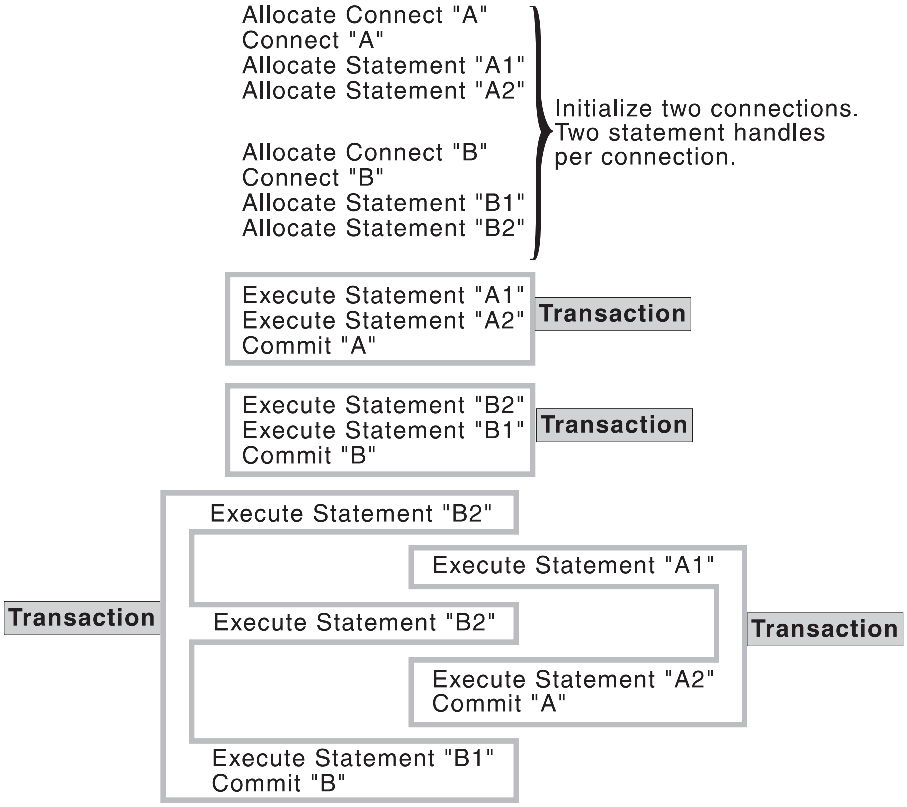

Figure 9. Multiple Connections with Concurrent Transactions

In Figure 9, within the context of the ODBC connection model, the third and fourth transactions can be interleaved as shown. That is, if the application has specified SQL_CONCURRENT_TRANS, then the ODBC model supports one transaction for each active connection. The third transaction, consisting of the execution of statements B2, B2 again and B1 at data source B, can be managed and committed independent of the fourth transaction, consisting of the execution of statements A1 and A2 at data source A. That is, the transactions at A and B are independent and exist concurrently.

If the application specifies SQL_CONCURRENT_TRANS and is executing with `MULTICONTEXT=0` specified in the initialization file, then DB2 for OS/390 allows any number of concurrent connection handles to be allocated, subject to the restriction that only one physical connection can exist at any given time. This behavior precludes support for the ODBC connection model, and consequently the behavior of the application is substantially different (than that described for the ODBC execution model described above.)

In particular, the third transaction is executed as three transactions. Prior to executing statement `'B2'`, DB2 CLI connects to B. This statement is executed and committed prior to reconnecting to data source A to execute `"A1"`. Similarly, this statement at data source A is committed prior to reconnecting to data source B to execute statement `"B2"`. This statement is then committed and a reconnection is made to A to execute `"A2"`. Next, another commit occurs and a reconnection to B to execute `"B1"`.

From an application point of view, the transaction at data source B, consisting of B2->B2->B1, is broken into three independent transactions: B2, B2 and B1. The fourth transaction at data source A, consisting of A1->A2, is broken into two independent transactions: A1 and A2.

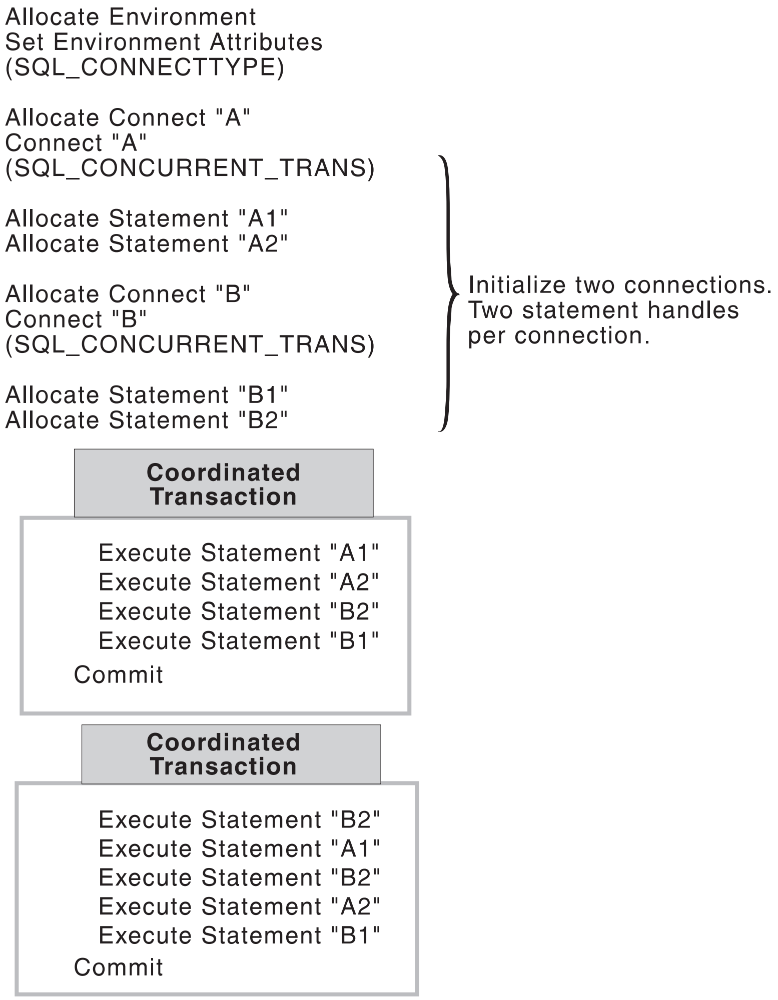

Figure 10. Multiple Connections with Coordinated Transactions

For a discussion of multiple active transaction support, see “DB2 CLI Support of Multiple Contexts” on page 368.

Establishing a Coordinated Transaction Connection

An application can establish coordinated transaction connections by calling the SQLSetEnvAttr() or SQLSetConnectOption() functions, or by setting the CONNECTTYPE keyword in the DB2 CLI initialization file or in the connection string for SQLDriverConnect(). The initialization file is intended for existing applications that do not use the SQLSetConnectOption() function. For information about the keywords, refer to “DB2 CLI Initialization File” on page 62.

An application cannot have a mixture of concurrent and coordinated connections; the type of the first connection determines the type of all subsequent connections. SQLSetEnvAttr() and SQLSetConnectOption() return an error if an application attempts to change the connect type while there is an active connection. When the connection type is established, it persists until SQLFreeEnv is called.

Distributed Unit of Work Example

The following example connects to two data sources using a SQL_CONNECTTYPE set to SQL_COORDINATED_TRANS (CONNECT (Type 2)).

```
/* ... */
#define MAX_CONNECTIONS   2

int
DBconnect(SQLHENV henv,
          SQLHDBC * hdbc,
          char    * server);

int
main()
{
    SQLHENV         henv;
    SQLHDBC         hdbc[MAX_CONNECTIONS];
    SQLRETURN       rc;

    char *          svr[MAX_CONNECTIONS] =
                    {
                      "KARACHI"   ,
                      "DAMASCUS"
                    }

    /* allocate an environment handle   */
    SQLAllocEnv(&henv);

    /* Before allocating any connection handles, set Environment wide
       Connect Options  */
    /* Set to Connect Type 2 */
    rc = SQLSetEnvAttr(henv, SQL_CONNECTTYPE,
                       (SQLPOINTER) SQL_COORDINATED_TRANS, 0);
/* ... */
    /* Connect to first data source */
    /* allocate a connection handle     */
    if (SQLAllocConnect(henv, &hdbc[0]) != SQL_SUCCESS) {
        printf(">---ERROR while allocating a connection handle-----\n");
        return (SQL_ERROR);
    }

    /* Connect to first data source (Type-II) */

    DBconnect (henv,
               &hdbc[0],
               svr[0]);


    /* allocate a second connection handle     */
    if (SQLAllocConnect(henv, &hdbc[1]) != SQL_SUCCESS) {
        printf(">---ERROR while allocating a connection handle-----\n");
        return (SQL_ERROR);
    }
    /* Connect to second data source (Type-II) */

    DBconnect (henv,
               &hdbc[1],
               svr[1]);
    /*********   Start Processing Step  *************************/
    /* allocate statement handle, execute statement, etc.       */
    /* Note that both connections participate in the disposition*/
    /* of the transaction. Note that a NULL connection handle   */
    /* is passed as all work is committed on all connections.   */
    /*********   End Processing Step  ***************************/

    (void) SQLTransact (henv,
                        SQL_NULL_HDBC,
                        SQL_COMMIT);

    /* Disconnect, free handles and exit */
}


/********************************************************************
**   Server is passed as a parameter. Note that USERID and PASSWORD**
**   are always NULL.                                              **
********************************************************************/

int
DBconnect(SQLHENV henv,
          SQLHDBC * hdbc,
          char    * server)
{
    SQLRETURN       rc;
    SQLCHAR         buffer[255];
    SQLSMALLINT     outlen;


    SQLAllocConnect(henv, hdbc);/* allocate a connection handle     */

    rc = SQLConnect(*hdbc, server, SQL_NTS, NULL, SQL_NTS, NULL, SQL_NTS);
    if (rc != SQL_SUCCESS) {
        printf(">--- Error while connecting to database: %s -------\n", server);
        return (SQL_ERROR);
    } else {
        printf(">Connected to %s\n", server);
        return (SQL_SUCCESS);
    }
}
/* ... */
```

Querying System Catalog Information

Often, one of the first tasks an application performs is to display to the user a list of tables from which one or more tables are selected by the user to work with. Although the application can issue its own queries against the database system catalog to get this type of catalog information, it is best that the application calls the DB2 CLI catalog functions instead. These catalog functions provide a generic interface to issue queries and return consistent result sets across the DB2 family of servers. In most cases, this allows the application to avoid server specific and release specific catalog queries.

The catalog functions operate by returning a result set to the application through a statement handle. Calling these functions is conceptually equivalent to using `SQLExecDirect()` to execute a SELECT against the system catalog tables. After calling these functions, the application can fetch individual rows of the result set as it would process column data from an ordinary SQLFetch(). The DB2 CLI catalog functions are:

- “SQLColumnPrivileges - Get Privileges Associated With The Columns of A Table” on page 110
- “SQLColumns - Get Column Information for a Table” on page 115
- “SQLForeignKeys - Get the List of Foreign Key Columns” on page 169
- “SQLPrimaryKeys - Get Primary Key Columns of A Table” on page 269
- “SQLProcedureColumns - Get Input/Output Parameter Information for A Procedure” on page 274
- “SQLProcedures - Get List of Procedure Names” on page 283
- “SQLSpecialColumns - Get Special (Row Identifier) Columns” on page 320
- “SQLStatistics - Get Index and Statistics Information For A Base Table” on page 325
- “SQLTablePrivileges - Get Privileges Associated With A Table” on page 330
- “SQLTables - Get Table Information” on page 334

The result sets returned by these functions are defined in the descriptions for each catalog function. The columns are defined in a specified order. The results sets for each API are fixed and cannot be changed. In future releases, other columns might be added to the end of each defined result set, therefore applications should be written in a way that would not be affected by such changes.

Some of the catalog functions result in execution of fairly complex queries, and for this reason should only be called when needed. It is recommended that the application save the information returned rather than making repeated calls to get the same information.

Input Arguments on Catalog Functions

All of the catalog functions have *CatalogName* and *SchemaName* (and their associated lengths) on their input argument list. Other input arguments can also include *TableName*, *ProcedureName*, or *ColumnName* (and their associated lengths). These input arguments are used to either identify or constrain the amount of information to be returned. *CatalogName*, however, must always be a null pointer (with its length set to 0) as DB2 CLI does not support three-part naming.

In the Function Arguments sections for these catalog functions in “Chapter 5. Functions” on page 73, each of the above input arguments are described either as a pattern-value or just as an ordinary argument. For example, SQLColumnPrivileges() treats *SchemaName* and *TableName* as ordinary arguments and *ColumnName* as a pattern-value.

Inputs treated as ordinary arguments are taken literally and the case of letters is significant. The argument does not qualify a query but rather identifies the information desired. If the application passes a null pointer for this argument, the results can be unpredictable.

Inputs treated as pattern-values are used to constrain the size of the result set by including only matching rows as though the underlying query were qualified by a WHERE clause. If the application passes a null pointer for a pattern-value input, the argument is not used to restrict the result set (that is, there is no WHERE clause). If a catalog function has more than one pattern-value input argument, they are treated as though the WHERE clauses in the underlying query were joined by AND; a row appears in this result set only if it meets all the conditions of the WHERE clauses.

Each pattern-value argument can contain:

- The underscore (_) character which stands for any single character.
- The percent (%) character which stands for any sequence of zero or more characters.
- Characters which stand for themselves. The case of a letter is significant.

These argument values are used on conceptual LIKE predicates in the WHERE clause. To treat the metadata characters (_, %) as themselves, an escape character must immediately precede the _ or %. The escape character itself can be specified as part of the pattern by including it twice in succession. An application can determine the escape character by calling `SQLGetInfo()` with SQL_SEARCH_PATTERN_ESCAPE.

Catalog Functions Example

In the sample application:

- A list of all tables are displayed for the specified schema (qualifier) name or search pattern.
- Column, special column, foreign key, and statistics information is returned for a selected table.

DB2 for common server is the data source in this example. Output from a sample is shown below. Relevant segments of the sample are listed for each of the catalog functions.

```
Enter Search Pattern for Table Schema Name:
STUDENT
Enter Search Pattern for Table Name:
%
### TABLE SCHEMA              TABLE_NAME                TABLE_TYPE
    ------------------------- ------------------------- ----------
1   STUDENT                   CUSTOMER                  TABLE
2   STUDENT                   DEPARTMENT                TABLE
3   STUDENT                   EMP_ACT                   TABLE
4   STUDENT                   EMP_PHOTO                 TABLE
5   STUDENT                   EMP_RESUME                TABLE
6   STUDENT                   EMPLOYEE                  TABLE
7   STUDENT                   NAMEID                    TABLE
8   STUDENT                   ORD_CUST                  TABLE
9   STUDENT                   ORD_LINE                  TABLE
10  STUDENT                   ORG                       TABLE
11  STUDENT                   PROD_PARTS                TABLE
12  STUDENT                   PRODUCT                   TABLE
13  STUDENT                   PROJECT                   TABLE
14  STUDENT                   STAFF                     TABLE
Enter a table Number and an action:(n [Q | C | P | I | F | T |O | L])
|Q=Quit     C=cols     P=Primary Key I=Index   F=Foreign Key |
|T=Tab Priv O=Col Priv S=Stats       L=List Tables           |
1c
Schema: STUDENT  Table Name: CUSTOMER
   CUST_NUM, NOT NULLABLE, INTeger (10)
   FIRST_NAME, NOT NULLABLE, CHARacter (30)
   LAST_NAME, NOT NULLABLE, CHARacter (30)
   STREET, NULLABLE, CHARacter (128)
   CITY, NULLABLE, CHARacter (30)
   PROV_STATE, NULLABLE, CHARacter (30)
   PZ_CODE, NULLABLE, CHARacter (9)
   COUNTRY, NULLABLE, CHARacter (30)
   PHONE_NUM, NULLABLE, CHARacter (20)
>> Hit Enter to Continue<<

1p
Primary Keys for STUDENT.CUSTOMER
 1  Column: CUST_NUM            Primary Key Name: = NULL
>> Hit Enter to Continue<<

1f
Primary Key and Foreign Keys for STUDENT.CUSTOMER
  CUST_NUM  STUDENT.ORD_CUST.CUST_NUM
      Update Rule SET NULL , Delete Rule: NO ACTION
>> Hit Enter to Continue<<
```

Sending/Retrieving Long Data in Pieces

When manipulating long data, it might not be feasible for the application to load the entire parameter data value into storage at the time the statement is executed, or when the data is fetched from the database. A method is provided to allow the application to handle the data in pieces. The technique to send long data in pieces is called *Specifying Parameter Values at Execute Time* because it can also be used to specify values for fixed size non-character data types such as integers.

Specifying Parameter Values at Execute Time

A bound parameter for which value is prompted at execution time instead of stored in memory before calling `SQLExecute()` or `SQLExecDirect()` is called a data-at-execute parameter. To indicate such a parameter on an `SQLBindParameter()` or `SQLSetParam()` call, the application:

- Sets the input data length pointer to point to a variable that, at execute time, contains the value SQL_DATA_AT_EXEC.
- If there is more than one data-at-execute parameter, sets each input data pointer argument to some value that it recognizes as uniquely identifying the field in question.

If there are any data-at-execute parameters when the application calls `SQLExecDirect()` or `SQLExecute()`, the call returns with SQL_NEED_DATA to prompt the application to supply values for these parameters. The application responds as follows:

1. It calls `SQLParamData()` to conceptually advance to the first such parameter. `SQLParamData()` returns SQL_NEED_DATA and provides the contents of the input data pointer argument specified on the associated `SQLBindParameter()` or `SQLSetParam()` call to help identify the information required.

2. It calls SQLPutData() to pass the actual data for the parameter. Long data can be sent in pieces by calling SQLPutData() repeatedly.
3. It calls SQLParamData() again after it has provided the entire data for this data-at-execute parameter. If more data-at-execute parameters exist, SQLParamData() again returns SQL_NEED_DATA and the application repeats steps 2 and 3 above.

When all data-at-execute parameters are assigned values, SQLParamData() completes execution of the SQL statement and produces a return value and diagnostics as the original SQLExecDirect() or SQLExecute() would have produced. The right side of Figure 11 on page 353 illustrates this flow.

While the data-at-execution flow is in progress, the only DB2 CLI functions the application can call are:

- SQLParamData() and SQLPutData() as given in the sequence above.
- The SQLCancel() function which is used to cancel the flow and force an exit from the loops on the right side of Figure 11 on page 353 without executing the SQL statement.
- The SQLError() function. The application also must not end the transaction nor set any connection attributes.

Fetching Data in Pieces

Typically, based on its knowledge of a column in the result set (via SQLDescribeCol() or prior knowledge), the application can choose to allocate the maximum memory the column value can occupy and bind it using SQLBindCol(). However, in the case of character and binary data, the column can be long. If the length of the column value exceeds the length of the buffer the application can allocate, or afford to allocate, a feature of SQLGetData() lets the application use repeated calls to obtain in sequence the value of a single column in more manageable pieces.

Basically, as shown on the left side of Figure 11 on page 353, a call to SQLGetData() returns SQL_SUCCESS_WITH_INFO (with SQLSTATE 01004) to indicate more data exists for this column. SQLGetData() is called repeatedly to get the remaining pieces of data until it returns SQL_SUCCESS, signifying that the entire data were retrieved for this column.

Input and Retrieval Example

See “DB2 CLI Application” on page 443 for a detailed example that uses SQLGetData() and SQLPutData() to input and retrieve data.

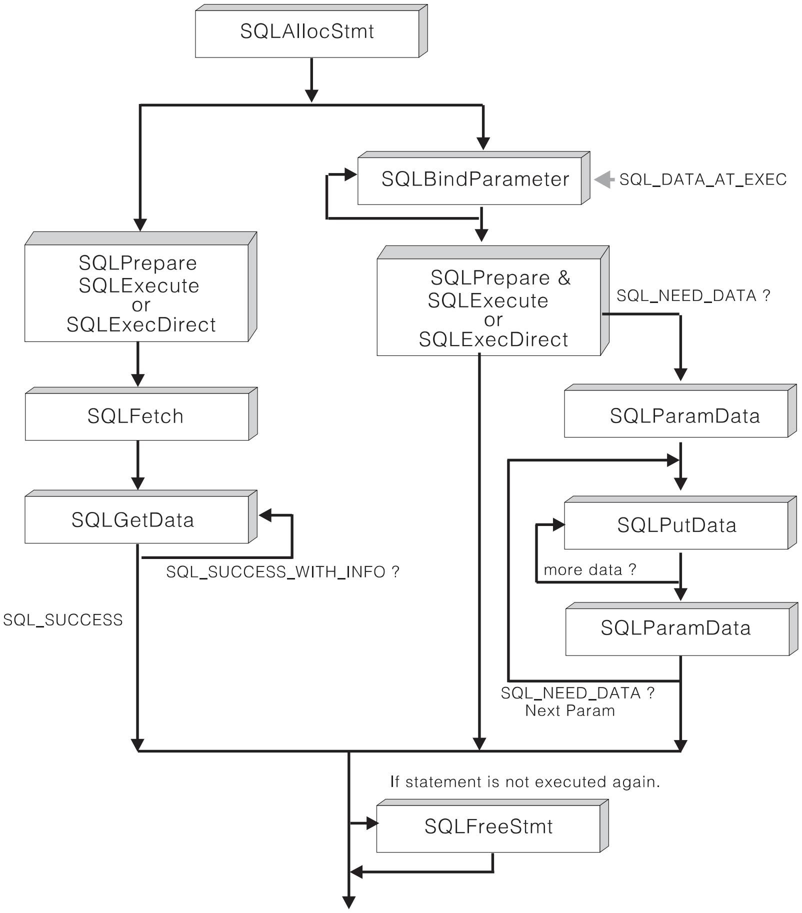

Figure 11. Input and Retrieval of Data in Pieces

Using Arrays to Input Parameter Values

For some data entry and update applications, users might often insert, delete, or change many cells in a data entry form and then ask for the data to be sent to the database. For these situations of bulk insert, delete, or update, DB2 CLI provides an array input method to save the application from having to call SQLExecute() repeatedly on the same INSERT, DELETE, or UPDATE statement. In addition, there can be savings in network flows.

This method involves the binding of parameter markers to arrays of storage locations via the SQLBindParameter() call. For character and binary input data, the application uses the maximum input buffer size argument (*cbValueMax*) on SQLBindParameter() call to indicate to DB2 CLI the location of values in the input array. For other input data types, the length of each element in the array is assumed to be the size of the C data type. SQLParamOptions() is called to specify how many elements are in the array before the execution of the SQL statement.

Suppose for Figure 12 on page 355 there is an application that allows the user to change values in the OVERTIME_WORKED and OVERTIME_PAID columns of a time sheet data entry form. Also suppose that the primary key of the underlying EMPLOYEE table is EMPLOY_ID. The application can then request to prepare the following SQL statement:

```
UPDATE EMPLOYEE SET OVERTIME_WORKED= ? and OVERTIME_PAID= ?
WHERE EMPLOY_ID=?
```

When the user enters all the changes, the application counts that *n* rows are to change and allocates m=3 arrays to store the changed data and the primary key. Then, the application makes calls to functions as follows:

1. SQLBindParameter() to bind the three parameter markers to the location of three arrays in memory.
2. SQLParamOptions() to specify the number of rows to change (the size of the array).
3. SQLExecute() once and all the updates are sent to the database.

This is the flow shown on the right side of Figure 12 on page 355.

The basic method is shown on the left side of Figure 12 on page 355 where SQLBindParameter() is called to bind the three parameter markers to the location of three variables in memory. SQLExecute() is called to send the first set of changes to the database. The variables are updated to reflect values for the next row of changes and again SQLExecute() is called. This method has n-1 extra SQLExecute() calls. For insert and update, use SQLRowCount to verify the number of changed rows.

Note: SQLSetParam() must not be used to bind an array storage location to a parameter marker. In the case of character or binary input data, there is no method to specify the size of each element in the input array.

For queries with parameter markers on the WHERE clauses, an array of input values generate multiple sequential result sets. Each result set can be processed before moving onto the next one by calling SQLMoreResults(). See "SQLMoreResults - Determine If There Are More Result Sets" on page 246 for more information and an example.

Figure 12 on page 355 shows the two methods of executing a statement with *m* parameters *n* times. The basic method shown on the left calls SQLExecute() once for each set of parameter values. The array method on the right, calls SQLParamOptions() to specify the number of rows (*n*), then calls SQLExecute() once. Both methods must call SQLBindParameter() once for each parameter.

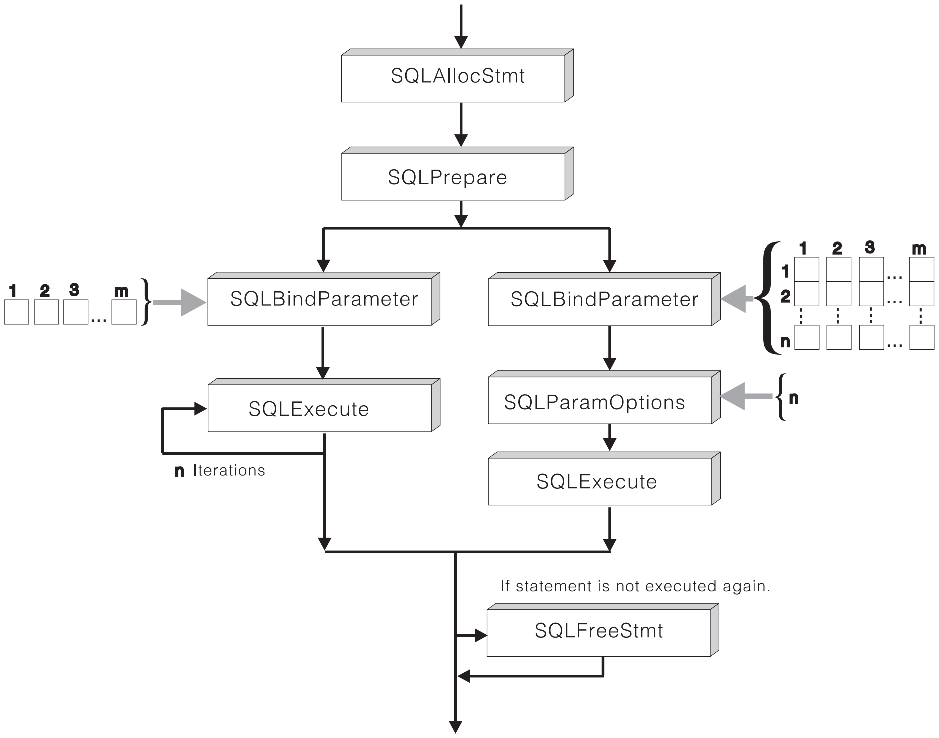

Figure 12. Array Insert

Array Input Example

This example shows an array INSERT statement. For an example of an array query statement, refer to “SQLMoreResults - Determine If There Are More Result Sets” on page 246.

```
/* ... */
    SQLUINTEGER pirow = 0;
    SQLCHAR         stmt[] =
    "INSERT INTO CUSTOMER ( Cust_Num, First_Name, Last_Name) "
      "VALUES (?, ?, ?)";

    SQLINTEGER      Cust_Num[25] = {
        10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
        110, 120, 130, 140, 150, 160, 170, 180, 190, 200,
        210, 220, 230, 240, 250
    };

    SQLCHAR         First_Name[25][31] = {
        "EVA",     "EILEEN",     "THEODORE", "VINCENZO",  "SEAN",
        "DOLORES", "HEATHER",    "BRUCE",    "ELIZABETH", "MASATOSHI",
        "MARILYN", "JAMES",      "DAVID",    "WILLIAM",   "JENNIFER",
        "JAMES",    "SALVATORE", "DANIEL",   "SYBIL",     "MARIA",
        "ETHEL",    "JOHN",      "PHILIP",   "MAUDE",     "BILL"
    };

    SQLCHAR         Last_Name[25][31] = {
        "SPENSER", "LUCCHESI", "O'CONNELL", "QUINTANA",
        "NICHOLLS", "ADAMSON", "PIANKA", "YOSHIMURA",
        "SCOUTTEN", "WALKER", "BROWN", "JONES",
        "LUTZ", "JEFFERSON", "MARINO", "SMITH",
        "JOHNSON", "PEREZ", "SCHNEIDER", "PARKER",
        "SMITH", "SETRIGHT", "MEHTA", "LEE",
        "GOUNOT"
    };
/* ... */
    /* Prepare the statement */
    rc = SQLPrepare(hstmt, stmt, SQL_NTS);

    rc = SQLParamOptions(hstmt, 25, &pirow);

    rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER,
                          0, 0, Cust_Num, 0, NULL);

    rc = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                          31, 0, First_Name, 31, NULL);

    rc = SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                         31, 0, Last_Name, 31, NULL);

    rc = SQLExecute(hstmt);
    printf("Inserted %ld Rows\n", pirow);

/* ... */
```

Retrieving A Result Set Into An Array

One of the most common tasks performed by an application is to issue a query statement, and then fetch each row of the result set into application variables that were bound using `SQLBindCol()`. If the application requires that each column or each row of the result set be stored in an array, each fetch must be followed by either a data copy operation or a new set of `SQLBindCol()` calls to assign new storage areas for the next fetch.

Alternatively, applications can eliminate the overhead of extra data copies or extra `SQLBindCol()` calls by retrieving multiple rows of data (called a rowset) at a time into an array. `SQLBindCol()` is also used to assign storage for application array variables. By default, the binding of rows is in column-wise fashion: this is symmetrical to using `SQLBindParameter()` to bind arrays of input parameter values as described in the previous section.

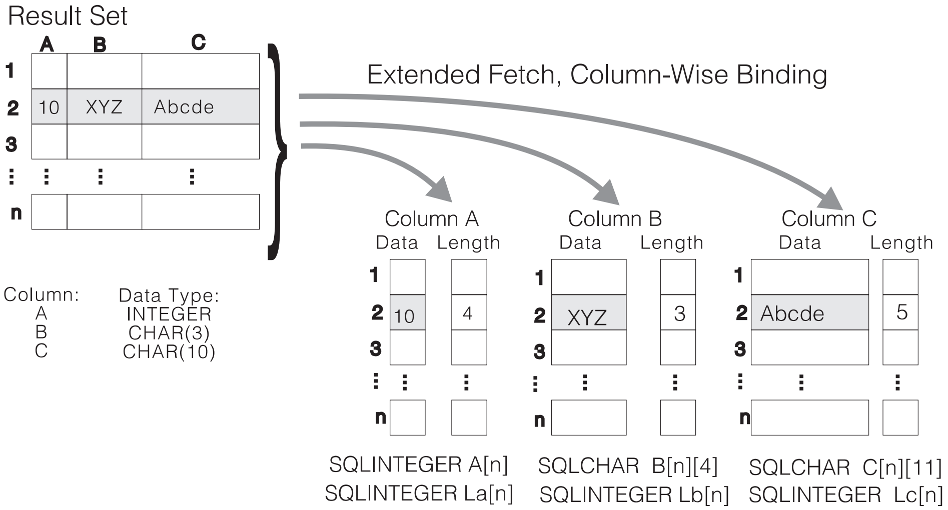

Figure 13. Column-Wise Binding

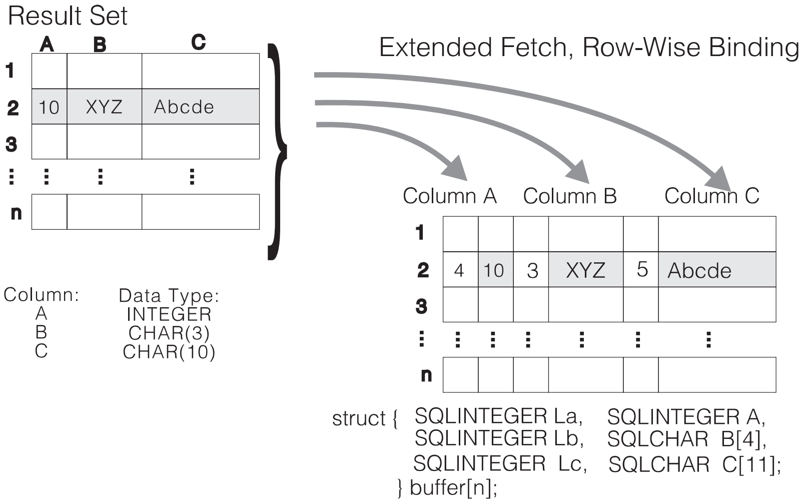

Figure 14. Row-Wise Binding

Returning Array Data for Column-Wise Bound Data

Figure 13 on page 357 is a logical view of column-wise binding. The right side of Figure 15 on page 359 shows the function flows for column-wise retrieval.

To specify column-wise array retrieval, the application calls SQLSetStmtOption() with the SQL_ROWSET_SIZE attribute to indicate how many rows to retrieve at a time. When the value of the SQL_ROWSET_SIZE attribute is greater than 1, DB2 CLI knows to treat the deferred output data pointer and length pointer as pointers to arrays of data and length rather than to one single element of data and length of a result set column.

The application then calls SQLExtendedFetch() to retrieve the data. When returning data, DB2 CLI uses the maximum buffer size argument (*cbValueMax*) on SQLBindCol() to determine where to store successive rows of data in the array; the number of bytes available for return for each element is stored in the deferred length array. If the number of rows in the result set is greater than the SQL_ROWSET_SIZE attribute value, multiple calls to SQLExtendedFetch() are required to retrieve all the rows.

Returning Array Data for Row-Wise Bound Data

The application can also do row-wise binding which associates an entire row of the result set with a structure. In this case the rowset is retrieved into an array of structures, each of which holds the data in one row and the associated length fields. Figure 14 on page 357 gives a pictorial view of row-wise binding.

To perform row-wise array retrieval, the application needs to call SQLSetStmtOption() with the SQL_ROWSET_SIZE attribute to indicate how many rows to retrieve at a time. In addition, it must call SQLSetStmtOption() with the SQL_BIND_TYPE attribute value set to the size of the structure to which the result columns are bound. DB2 CLI treats the deferred output data pointer of SQLBindCol() as the address of the data field for the column in the first element of the array of these structures. It treats the deferred output length pointer as the address of the associated length field of the column.

The application then calls SQLExtendedFetch() to retrieve the data. When returning data, DB2 CLI uses the structure size provided with the SQL_BIND_TYPE attribute to determine where to store successive rows in the array of structures.

Figure 15 on page 359 shows the required functions for each method. The left side shows *n* rows being selected, and retrieved one row at a time into m application variables. The right side shows the same n rows being selected, and retrieved directly into an array.

Note:

- The diagram shows *m* columns bound, so *m* calls to SQLBindCol() are required in both cases.
- If arrays of less than *n* elements had been allocated, then multiple SQLExtendedFetch() calls would be required.

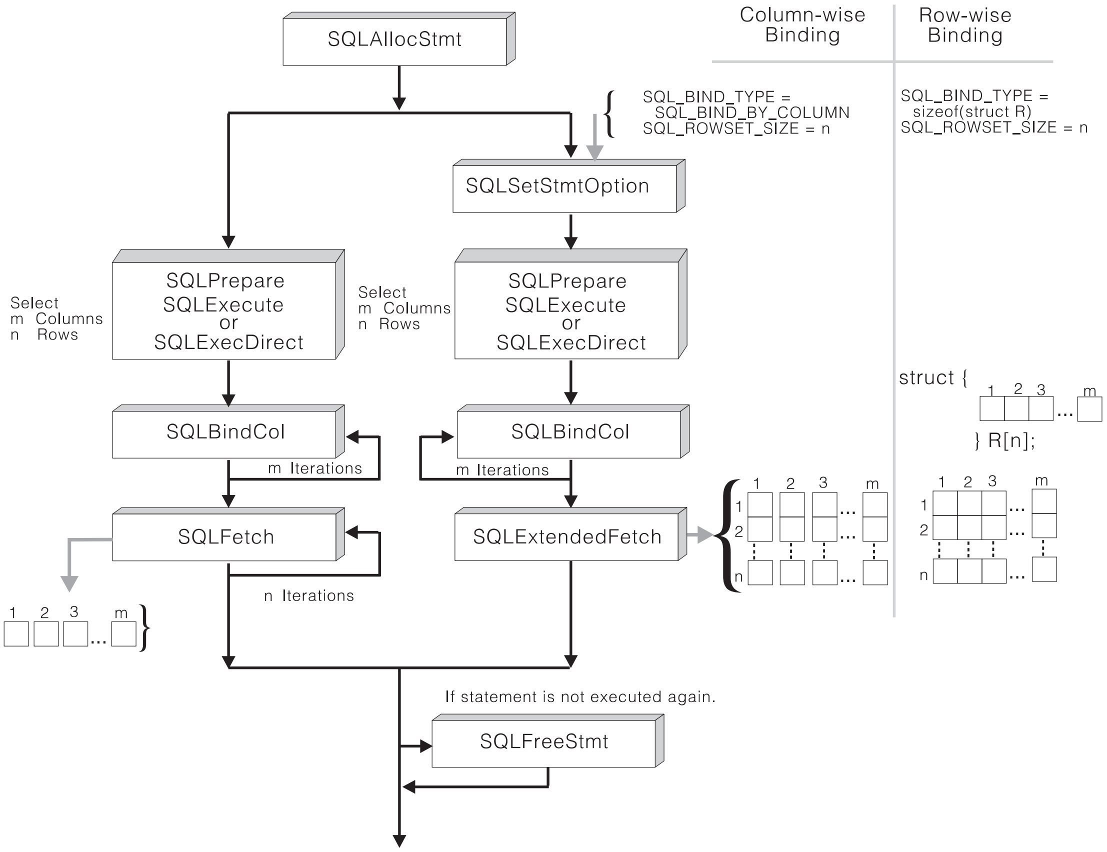

Figure 15. Array Retrieval

Column-Wise, Row-Wise Binding Example

```
/* ... */
#define NUM_CUSTOMERS 25
    SQLCHAR         stmt[] =
    {  "WITH "  /* Common Table expression (or Define Inline View) */
         "order (ord_num, cust_num, prod_num, quantity, amount) AS "
         "( "
         "SELECT c.ord_num, c.cust_num, l.prod_num, l.quantity,  "
                 "price(char(p.price, '.'), p.units, char(l.quantity, '.')) "
            "FROM ord_cust c, ord_line l, product p  "
            "WHERE c.ord_num = l.ord_num AND l.prod_num = p.prod_num  "
             "AND cust_num = CNUM(cast (? as integer)) "
         "), "
         "totals (ord_num, total) AS "
         "( "
          "SELECT ord_num, sum(decimal(amount, 10, 2))  "
          "FROM order GROUP BY ord_num "
         ") "
```

```
       /* The 'actual' SELECT from the inline view */
       "SELECT order.ord_num, cust_num, prod_num, quantity,  "
              "DECIMAL(amount,10,2) amount, total "
        "FROM order, totals  "
        "WHERE order.ord_num = totals.ord_num "
     };
    /* Array of customers to get list of all orders for */
    SQLINTEGER     Cust[]=
    {
        10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
        110, 120, 130, 140, 150, 160, 170, 180, 190, 200,
        210, 220, 230, 240, 250
    };

#define  NUM_CUSTOMERS sizeof(Cust)/sizeof(SQLINTEGER)

    /* Row-Wise (Includes buffer for both column data and length) */
    struct {
        SQLINTEGER      Ord_Num_L;
        SQLINTEGER      Ord_Num;
        SQLINTEGER      Cust_Num_L;
        SQLINTEGER      Cust_Num;
        SQLINTEGER      Prod_Num_L;
        SQLINTEGER      Prod_Num;
        SQLINTEGER      Quant_L;
        SQLDOUBLE       Quant;
        SQLINTEGER      Amount_L;
        SQLDOUBLE       Amount;
        SQLINTEGER      Total_L;
        SQLDOUBLE       Total;
    }               Ord[ROWSET_SIZE];

    SQLUINTEGER     pirow = 0;
    SQLUINTEGER     pcrow;
    SQLINTEGER      i;
    SQLINTEGER      j;
/* ... */
    /* Get details and total for each order Row-Wise */
    rc = SQLAllocStmt(hdbc, &hstmt);

    rc = SQLParamOptions(hstmt, NUM_CUSTOMERS, &pirow);

    rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_LONG, SQL_INTEGER,
                          0, 0, Cust, 0, NULL);

    rc = SQLExecDirect(hstmt, stmt, SQL_NTS);

    /* SQL_ROWSET_SIZE sets the max number of result rows to fetch each time */
    rc = SQLSetStmtOption(hstmt, SQL_ROWSET_SIZE, ROWSET_SIZE);

    /* Set Size of One row, Used for Row-Wise Binding Only */
    rc = SQLSetStmtOption(hstmt, SQL_BIND_TYPE, sizeof(Ord) / ROWSET_SIZE);

    /* Bind column 1 to the Ord_num Field of the first row in the array*/
    rc = SQLBindCol(hstmt, 1, SQL_C_LONG, (SQLPOINTER) &Ord[0].Ord_Num, 0,
                    &Ord[0].Ord_Num_L);

    /* Bind remaining columns ... */
/* ... */
```

```
    /* NOTE: This sample assumes that an order never has more
             rows than ROWSET_SIZE.  A check should be added below to call
             SQLExtendedFetch multiple times for each result set.
    */
    do  /* for each result set .... */
    { rc = SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 0, &pcrow, NULL);

      if (pcrow > 0) /* if 1 or more rows in the result set */
      {
        i = j = 0;

        printf("**************************************\n");
        printf("Orders for Customer: %ld\n", Ord[0].Cust_Num);
        printf("**************************************\n");

        while (i < pcrow)
        {   printf("\nOrder #: %ld\n", Ord[i].Ord_Num);
            printf("     Product  Quantity         Price\n");
            printf("     -------- ---------------- ------------\n");
            j = i;
            while (Ord[j].Ord_Num == Ord[i].Ord_Num)
            {   printf("    %8ld %16.7lf %12.2lf\n",
                       Ord[i].Prod_Num, Ord[i].Quant, Ord[i].Amount);
                i++;
            }
            printf("                               ============\n");
            printf("                              %12.2lf\n", Ord[j].Total);
        } /* end while */
      } /* end if */
    }
    while ( SQLMoreResults(hstmt) == SQL_SUCCESS);
/* ... */
```

Using Stored Procedures

An application can be designed to run in two parts, one on the client and the other on the server. The stored procedure is a server application that runs at the database within the same transaction as the client application. Stored procedures can be written in either embedded SQL or using the DB2 CLI functions (see “Writing a DB2 CLI Stored Procedure” on page 363).

Both the main application that calls a stored procedure and a stored procedure itself can be a either a CLI application or a standard DB2 precompiled application. Any combination of embedded SQL and CLI applications can be used. Figure 16 illustrates this concept.

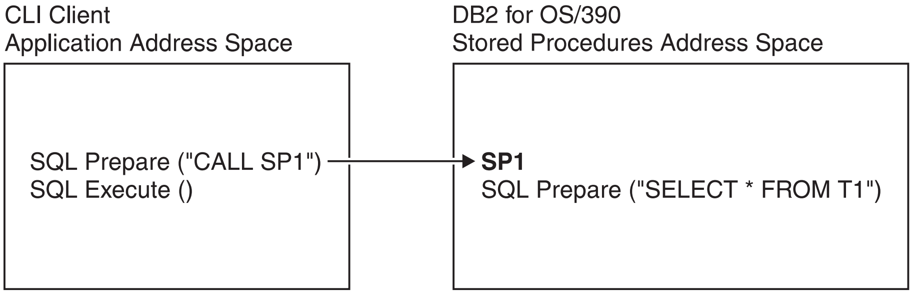
CLI Client
Application Address Space
DB2 for OS/390
Stored Procedures Address Space
SQL Prepare ("CALL SP1")
SQL Execute ()
SP1
SQL Prepare ("SELECT * FROM T1")

Figure 16. Running Stored Procedures

Advantages of Using Stored Procedures

In general, stored procedures have the following advantages:

- Avoid network transfer of large amounts of data obtained as part of intermediate results in a long sequence of queries.
- Deployment of client database applications into client/server pieces.

In addition, stored procedures written in embedded static SQL have the following advantages:

- Performance - static SQL is prepared at precompile time and has no run time overhead of access plan (package) generation.
- Encapsulation (information hiding) - users do not need to know the details about database objects in order to access them. Static SQL can help enforce this encapsulation.
- Security - users access privileges are encapsulated within the packages associated with the stored procedures, so there is no need to grant explicit access to each database object. For example, you can grant a user run access for a stored procedure that selects data from tables for which the user does not have select privilege.

Catalog Table for Stored Procedures

If the server is DB2 for common server Version 2 Release 1 or later, or DB2 for MVS/ESA Version 4 or later, an application can call `SQLProcedureColumns()` to determine the type of a parameter in a procedure call.

If the stored procedure resides on a DB2 for MVS/ESA Version 4 or later server, the name of the stored procedure must be defined in the SYSIBM.SYSPROCEDURES catalog table. (The pseudo catalog table used by DB2 for common server is a derivation and extension of the DB2 for OS/390 SYSIBM.SYSPROCEDURES catalog table).

For more information, refer to “Catalog Table for Stored Procedures” and “SQLProcedureColumns - Get Input/Output Parameter Information for A Procedure” on page 274.

Calling Stored Procedures from a DB2 CLI Application

Stored procedures are invoked from a DB2 CLI application by passing the following CALL statement syntax to `SQLExecDirect()` or to `SQLPrepare()` followed by `SQLExecute()`.

```
                                  ┌─,───┐
►►──CALL──procedure-name──(──▼─────┴──)──────────────────────────────►◄
                                  └─?─┘
```

procedure-name
The name of the stored procedure to execute.

Note: Although the CALL statement cannot be prepared dynamically, DB2 CLI accepts the CALL statement as if it could be dynamically prepared.

Stored procedures can also be called using the ODBC vendor escape sequence shown in “Stored Procedure Call Syntax” on page 378.

For more information regarding the use of the CALL statement and stored procedures, refer to *SQL Reference* and *Application Programming and SQL Guide*.

If the server is DB2 for common server Version 2 Release 1 or later, or DB2 for MVS/ESA Version 4 or later, SQLProcedures() can be called to obtain a list of stored procedures available at the database.

The ? in the CALL statement syntax diagram denotes parameter markers corresponding to the arguments for a stored procedure. All arguments must be passed using parameter markers; literals, the NULL keyword, and special registers are not allowed. However, literals can be used if a vendor escape clause for the CALL statement is used. See “Using Vendor Escape Clauses” on page 376.

The parameter markers in the CALL statement are bound to application variables using SQLBindParameter(). Although stored procedure arguments can be used both for input and output, in order to avoid sending unnecessary data between the client and the server, the application should specify on SQLBindParameter() the parameter type of an input argument to be SQL_PARAM_INPUT and the parameter type of an output argument to be SQL_PARAM_OUTPUT. Those arguments that are both input and output have a parameter type of SQL_PARAM_INPUT_OUTPUT. Literals are considered type SQL_PARAM_INPUT only.

Writing a DB2 CLI Stored Procedure

Although embedded SQL stored procedures provide the most advantages, application developers who have existing DB2 CLI applications might wish to move components of the application to run on the server. In order to minimize the required changes to the code and logic of the application, these components can be implemented by writing stored procedures using DB2 CLI.

Auto-commit must be off. This is acheived by using the AUTOCOMMIT keyword in the initialization file or by using the SQLSetConnectOption API with the SQL_AUTOCOMMIT connect option SQL_AUTOCOMMIT_OFF.

SQLConnect() should be a null connect. Since all the internal information related to a DB2 CLI connection is referenced by the connection handle, and since a stored procedure runs under the same connection and transaction as the client application, a stored procedure written using DB2 CLI must make a null SQLConnect() call to associate a connection handle with the underlying connection of the client application. In a null SQLConnect() call, the *szDSN*, *szUID*, and *szAuthStr* argument pointers are all set to NULL and their respective length arguments all set to 0.

For stored procedures written in DB2 CLI, the COMMIT_ON _RETURN option has no effect on DB2 CLI rules; set it to 'N'. However, be aware that setting this option to 'N' overrides the manual-commit mode set in the client application.

Returning Result Sets From Stored Procedures

DB2 CLI provides the ability to retrieve one or more result sets from a stored procedure call, provided the stored procedure is coded such that one or more cursors, each associated with a query, is opened and left opened when the stored procedure exits. If more than one cursor is left open, multiple result sets are returned.

Stored procedures written to return one or more result sets to a DB2 CLI application should indicate the maximum number of result sets that can be returned in the RESULT_SETS column of the SYSIBM.SYSPROCEDURES table. A zero in this column indicates that open cursors returned no result sets.

Programming Stored Procedures to Return Result Sets

To return one or more result sets to a DB2 CLI application the stored procedure must satisfy the following requirements:

- The stored procedure indicates that it wants a result set returned by declaring a cursor on the result set, opening a cursor on the result set (that is, executing the query), and leaving the cursor open when exiting the stored procedure.
- For every cursor that is left open, a result set is returned to the application.
- If more than one cursor is left open, the result sets are returned in the order in which their cursors were opened in the stored procedure.
- In a stored procedure, DB2 CLI uses a cursor declared WITH RETURN. If the cursor is closed before the stored procedure exit, it is a local cursor. If the cursor remains open upon stored procedure exit, it returns a query result set (also called a multiple result set) to the client application.
- If the stored procedure calls `SQLFreeStmt()` with either SQL_DROP or SQL_CLOSE, then the cursor for the current result set is closed and the rows are flushed. Note that this is also the case for all other cursors associated with other result sets generated by this same stored procedure call. If the stored procedure is going to return result sets, it must terminate with the statement handle in a valid state, either after an `SQLExecute()` or `SQLFetch()` of an SQL query.
- Only unread rows are passed back. For example, if the result set of a cursor has 500 rows, and 150 of those rows were already read by the stored procedure when it terminated, then rows 151 through 500 are returned to the stored procedure. This can be useful if the stored procedure wishes to filter out some initial rows and not return them to the application.

Restrictions on Stored Procedures Returning Result Sets

In general, calling a stored procedure that returns a result set is equivalent to executing a query statement. The following restrictions apply:

- Column names are not returned by either `SQLDescribeCol()` or `SQLColAttributes()` for static query statements. In this case, the ordinal position of the column is returned instead.
- All result sets are read-only.
- Schema functions (such as `SQLTables()`) cannot be used to return a result set. If schema functions are used within a stored procedure, all of the cursors for the associated statement handles must be closed before returning, otherwise extraneous result sets might be returned.
- When a query is prepared, result set column information is available before the execute. When a stored procedure is prepared, the result set column information is not available until the CALL statement is executed.

Programming DB2 CLI Client Applications to Receive Result Sets

DB2 CLI applications can retrieve result sets after the execution of a stored procedure that leaves cursors open. The following guidelines explain the process and requirements.

- Before the stored procedure is called, ensure that there are no open cursors associated with the statement handle.
- Call the stored procedure.
- The execution of the stored procedure CALL statement effectively causes the cursors associated with the result sets to open.
- Examine any output parameters that are returned by the stored procedure. For example, the procedure might be designed so that there is an output parameter that indicates exactly how many result sets are generated.
- The DB2 CLI application can then process a query as it normally does. If the application does not know the nature of the result set or the number of columns returned, it can call `SQLNumResultCols()`, `SQLDescribeCol()` or `SQLColAttributes()`. Next, the application can use any permitted combination of `SQLBindCol()`, `SQLFetch()`, and `SQLGetData()` to obtain the data in the result set.
- When `SQLFetch()` returns SQL_NO_DATA_FOUND or if the application is finished with the current result set, the application can call `SQLMoreResults()` to determine if there are more result sets to retrieve. Calling `SQLMoreResults()` closes the current cursor and advances processing to the next cursor that was left open by the stored procedure.
- If there is another result set, then `SQLMoreResults()` returns success; otherwise, it returns an SQL_NO_DATA_FOUND.
- Result sets must be processed in serial fashion by the application, that is, one at a time in the order that they were opened in the stored procedure.

Stored Procedure Example with Query Result Set

A detailed stored procedure example is provided in Appendix F, “Example Code” on page 443.

Writing Multithreaded Applications

This section explains DB2 CLI's support of multithreading and multiple contexts, and provides guidelines for programming techniques.

DB2 CLI Support of Multiple LE Threads

All DB2 CLI applications have at least one LE thread created automatically in the application's LE *enclave*. A multithreaded DB2 CLI application creates additional LE threads using the *POSIX Pthread* function **pthread_create()**. These additional threads share the same reentrant copy of DB2 CLI code within the LE enclave.

DB2 CLI code is *reentrant* but uses shared storage areas that must be protected if multiple LE threads are running concurrently in the same enclave. The quality of being reentrant and correctly handling shared storage areas is referred to as *threadsafe*. This quality is required by multithreaded applications.

DB2 CLI supports concurrent execution of LE threads by making all of the DB2 CLI function calls threadsafe. This threadsafe quality of function calls is only available if DB2 CLI has access to OpenEdition system services. DB2 CLI uses the Pthread *mutex* functions of OpenEdition system services to provide threadsafe function calls by serializing critical sections of DB2 CLI code. See “Initialization Keywords” on page 64 for a description of the `THREADSAFE` keyword.

Because OpenEdition system services are present, threadsafe capability is available by default when executing a DB2 CLI application in the following environments:

- The OpenEdition shell
- TSO or batch for HFS-resident applications using the IBM supplied BPXBATCH program. (See *OS/390 OpenEdition Command Reference* for more information about BPXBATCH).
- TSO or batch for applications that are not HFS-resident if the LE runtime option POSIX(ON) is specified when the application runs.

For example, to specify POSIX(ON) in TSO, you can invoke the DB2 CLI application APP1 in the MVS dataset USER.RUNLIB.LOAD as follows:

```
CALL 'USER.RUNLIB.LOAD(APP1)' 'POSIX(ON)/'
```

Using batch JCL, you can invoke the same application:

```
//STEP1 EXEC PGM=APP1,PARM='POSIX(ON)/'
//STEPLIB DD DSN=USER.RUNLIB.LOAD,DISP=SHR
//        DD ...other libraries needed at runtime...
```

For more OpenEdition information relating to DB2 CLI, see “Special Considerations for OS/390 OpenEdition” on page 57. Also, see *Language Environment for OS/390 & VM Programming Guide* for more information about running programs that use OpenEdition system services.

Multithreaded applications allow threads to perform work in parallel on different connections as shown in Figure 17 on page 367.

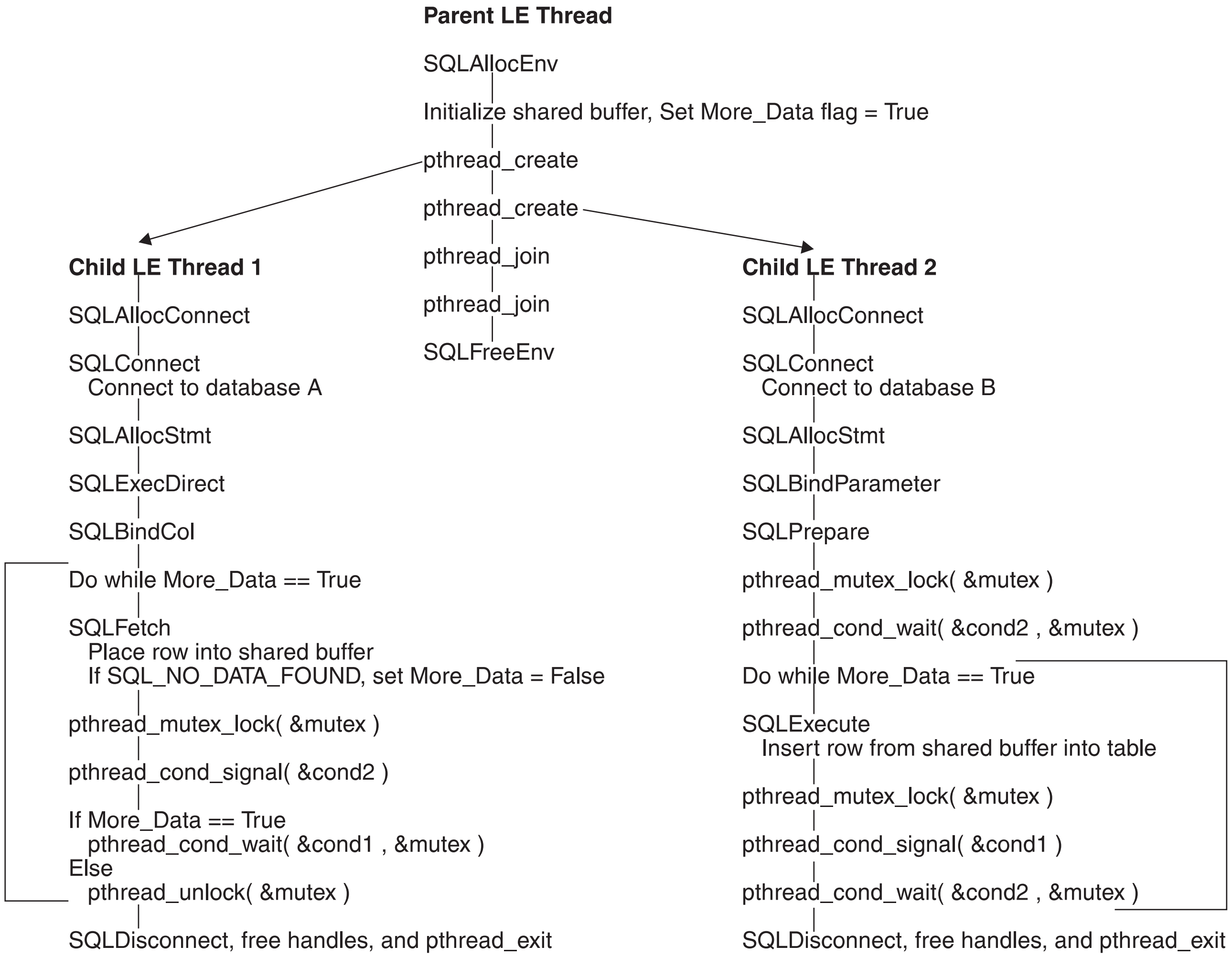
Parent LE Thread
SQLAllocEnv
Initialize shared buffer, Set More_Data flag = True
pthread_create
pthread_create
pthread_join
pthread_join
SQLFreeEnv
Child LE Thread 1
SQLAllocConnect
SQLConnect
Connect to database A
SQLAllocStmt
SQLExecDirect
SQLBindCol
Do while More_Data == True
SQLFetch
Place row into shared buffer
If SQL_NO_DATA_FOUND, set More_Data = False
pthread_mutex_lock(&mutex)
pthread_cond_signal(&cond2)
If More_Data == True
pthread_cond_wait(&cond1 , &mutex)
Else
pthread_unlock(&mutex)
SQLDisconnect, free handles, and pthread_exit
Child LE Thread 2
SQLAllocConnect
SQLConnect
Connect to database B
SQLAllocStmt
SQLBindParameter
SQLPrepare
pthread_mutex_lock(&mutex)
pthread_cond_wait(&cond2 , &mutex)
Do while More_Data == True
SQLExecute
Insert row from shared buffer into table
pthread_mutex_lock(&mutex)
pthread_cond_signal(&cond1)
pthread_cond_wait(&cond2 , &mutex)
SQLDisconnect, free handles, and pthread_exit

Figure 17. Multithreaded Application

In Figure 17, an application implements a database-to-database copy as follows:

- A parent LE thread creates two child LE threads. The parent thread remains present for the duration of the activity of the child threads. DB2 CLI requires that the thread which establishes the environment using `SQLAllocEnv()` must remain present for the duration of the application, so that DB2 language interface routines remain resident in the LE enclave.
- One child LE thread connects to database A and uses `SQLFetch()` calls to read data from one connection into a shared application buffer.
- Another child LE thread connects to database B and concurrently reads from the shared buffer, inserting the data into database B.
- Pthread functions are used to synchronize the use of the shared application buffer. See *OS/390 C/C++ Run-Time Library Reference* for a description of the Pthread functions.

When to Use Multiple LE Threads

Detailed discussion of evaluating application requirements and making decisions about whether or not to use multithreading is beyond the scope of this book. However, there are some general application types that are well-suited to multithreading. For example, applications that handle asynchronous work requests are candidates for multithreading.

An application that handles asynchronous work requests can take the form of a parent/child threading model in which the parent LE thread creates child LE threads to handle incoming work. The parent thread can then act as a dispatcher of these work requests as they arrive, directing them to child threads that are not currently busy servicing other work.

DB2 CLI Support of Multiple Contexts

The context consists of the application's logical connection to the data source and associated internal DB2 CLI connection information that allows the application to direct its operations to a data source. In DB2 CLI, the context is established when `SQLAllocConnect()` is issued.

The context is the DB2 CLI equivalent of a DB2 thread. DB2 CLI always creates a context when a successful `SQLAllocConnect()` is first issued by an application LE thread. If DB2 CLI support for multiple contexts is not enabled, only the first `SQLAllocConnect()` for a LE thread establishes a context. With support for multiple contexts, DB2 CLI establishes a separate context (and DB2 thread) each time the `SQLAllocConnect()` function is invoked.

If the initialization file specifies `MULTICONTEXT=0` (see “Initialization Keywords” on page 64), there can only be one context for each LE thread that the application creates. This single context per thread can provide only the simulated support of the ODBC connection model, explained in “DB2 CLI Restrictions on the ODBC Connection Model” on page 25.

When the initialization file specifies `MULTICONTEXT=1`, a distinct context is established for each connection handle that is allocated when `SQLAllocConnect()` is issued. Using `MULTICONTEXT=1` requires:

- The RRSAF attachment facility, specified by `MVSATTACHTYPE=RRSAF` in the initialization file.
- OS/390 Unauthorized Context Services, available in OS/390 Version 2 Release 5 and higher releases.

Consistent with the ODBC connection model, the use of `MULTICONTEXT=1` implies `CONNECTTYPE=1`. The connections are independently handled by `SQLTransact` for both commit and rollback.

The creation of a context for each connection is consistent with, and provides full support for, the ODBC connection model.

The context is established with `SQLAllocConnect()` and deleted by `SQLFreeConnect()`. All `SQLConnect()` and `SQLDisconnect()` operations that use the same connection handle belong to the same context. Although there can be only one active connection to a data source at any given time for the duration of the context, the target data source can be changed by `SQLDisconnect()` and `SQLConnect()`, subject to the rules of `CONNECTTYPE=1`.

With `MULTICONTEXT=1` specified, DB2 CLI automatically uses OS/390 Unauthorized Context Services to create and manage contexts for the application. However, DB2 CLI does not perform context management for the application if any of the following are true:

- The CLI application created a DB2 thread before invoking DB2 CLI. This is always the case for a stored procedure using DB2 CLI.
- The CLI application created and switched to a private context before invoking DB2 CLI. For example, an application that is explicitly using OS/390 Context Services and issues `ctxswch` to switch to a private context prior to invoking DB2 CLI cannot take advantage of `MULTICONTEXT=1`.
- The CLI application started a unit of recovery with any RRS resource manager before invoking DB2 CLI.
- `MVSATTACHTYPE=CAF` is specified in the initialization file.
- The OS/390 operating system level does not support Unauthorized Context Services.

The application can use the `SQLGetInfo()` function with *finfoType*=`SQL_MULTIPLE_ACTIVE_TXN` to determine if `MULTICONTEXT=1` is active for the DB2 CLI application. See "SQLGetInfo - Get General Information" on page 213 for the description of `SQLGetInfo()`.

Table 139. Connection Characteristics

Settings		Results		
MULTICONTEXT	**CONNECTTYPE**	**Can LE thread have more than one CLI connection with an outstanding unit of work?**	**Can LE thread commit/rollback CLI connection independently?**	**Number of DB2 created by DB2 CLI on behalf of application**
0	2	Y	N	1 per LE thread
0	1	N	Y	1 per LE thread
1 [1]	1 or 2 [2]	Y	Y	1 per CLI connection handle

Note:

1. `MULTICONTEXT=1` requires `MVSATTACHTYPE=RRSAF` and OS/390 Version 2 Release 5 or higher.
2. `MULTICONTEXT=1` implies `CONNECTTYPE=1` characteristics. With `MULTICONTEXT=1` and `CONNECTTYPE=2` specified in the initialization file, DB2 CLI ignores `CONNECTTYPE=2`. With `MULTICONTEXT=1` specified, any attempts to set `CONNECTTYPE=2` using `SQLSetEnvAttr()`, `SQLSetConnectOptions()`, or `SQLDriverConnect()` are rejected with SQLSTATE=01S02.
   - All connections in a DB2 CLI application have the same `CONNECTTYPE` and `MULTICONTEXT` characteristics. `CONNECTTYPE` is established at the first `SQLConnect()`. `MULTICONTEXT` is established at `SQLAllocEnv()`.
   - For `CONNECTTYPE=1` or `MULTICONTEXT=1`, the `AUTOCOMMIT` default is `ON`. For `CONNECTTYPE=2` or `MULTICONTEXT=0`, the `AUTOCOMMIT` default is `OFF`.

Multiple Contexts, One LE Thread

When using the initialization file setting `MULTICONTEXT=1`, a DB2 CLI application can create multiple independent connections for a LE thread. Figure 18 is an example of a multicontext, one LE thread application.

```
/*  Get an environment handle (henv).             */

SQLAllocEnv( &henv );

/*
 *  Get two connection handles, hdbc1 and hdbc2, which
 *  represent two independent DB2 threads.
 */
SQLAllocConnect( henv , &hdbc1 );
SQLAllocConnect( henv , &hdbc2 );

/*  Set autocommit off for both connections.     */
/*  This is done only to emphasize the           */
/*  independence of the connections for purposes */
/*  of this example, and is not intended as      */
/*  a general recommendation.                    */

SQLSetConnectOption( hdbc1 , SQL_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF );
SQLSetConnectOption( hdbc2 , SQL_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF );

/*  Perform SQL under DB2 thread 1 at STLEC1.    */

SQLConnect( hdbc1, (SQLCHAR *) "STLEC1", ... );
SQLAllocStmt  ...
SQLExecDirect ...
      .
      .
/*  Perform SQL under DB2 thread 2 at STLEC1.    */

SQLConnect( hdbc2, (SQLCHAR *) "STLEC1", ... );
SQLAllocStmt  ...
SQLExecDirect ...
     .
     .
/*  Commit changes on connection 1.              */

SQLTransact( henv , hdbc1 , SQL_COMMIT );

/*  Rollback changes on connection 2.            */

SQLTransact( henv , hdbc2 , SQL_ROLLBACK );
     .
     .
```

Figure 18. Example of independent connections on a single LE thread.

Multiple Contexts, Multiple LE Threads

Using the initialization file setting `MULTICONTEXT=1`, combined with the default `THREADSAFE=1`, the application can create multiple independent connections under multiple LE threads. This capability can support complex DB2 CLI server applications that handle multiple incoming work requests by using a fixed number of threads.

The multiple context, multiple LE thread capability requires some special considerations for the application using it. The Pthread functions should be used by the application for serialization of the use of connection handles and the associated statement handles. As an example of what can go wrong without proper serialization, see Figure 19 on page 371.

```
LE_Thread_1                          LE_Thread_2
   .                                    .
   .                                    .
   .                                    .
   .                                    .
   .                                    .
rc = SQLExecDirect( hstmt1 , ... );     .
   .                                    .
   .                                    .
   .                                 SQLFreeStmt( hstmt1 , SQL_DROP);
   .                                    .
   .                                    .
rc = SQLExecDirect( hstmt1 , ... );     .

  rc has the value SQL_INVALID_HANDLE because
  LE_Thread_2 has freed statement handle hstmt1.
```

Figure 19. Example of improper serialization.

The suggested design is to map one LE thread per connection by establishing a pool of connections as shown in Figure 20.

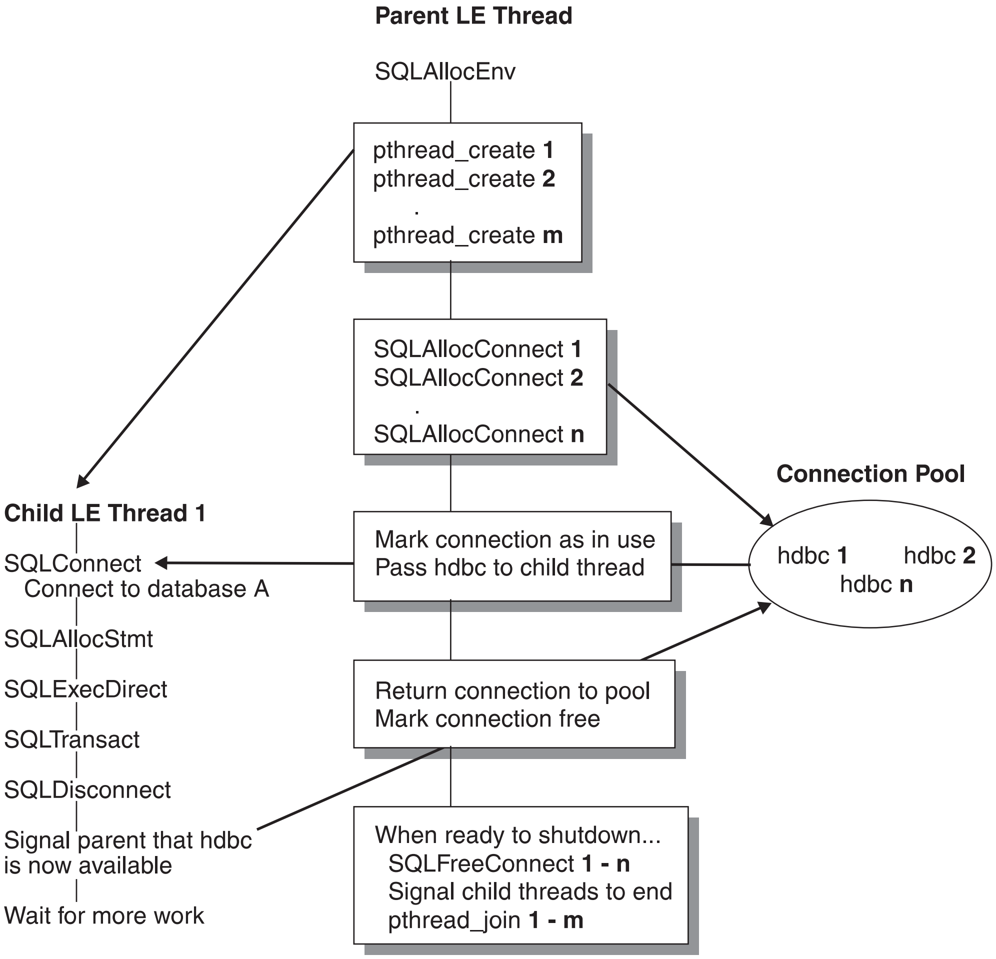

Figure 20. Model for Multithreading with Connection Pooling (MULTICONTEXT=1)

In Figure 20, a pool of connections is established as follows:

- Designate a parent LE thread which allocates:
  - *m* child LE threads
  - *n* connection handles
- Each task that requires a connection is executed by one of the child LE threads, and is given one of the *n* connections by the parent LE thread. The parent thread remains active, acting as a dispatcher of tasks.
- The parent LE thread marks a connection as being in use until the child thread returns it to the connection pool.
- The parent LE thread frees the connections using `SQLFreeConnect()` when the parent thread is ending.

DB2 CLI requires that the LE thread which establishes the environment using `SQLAllocEnv()` must remain present for the duration of the application, so that DB2 language interface routines will remain resident in the LE enclave.

This suggested design allows the parent LE thread to create more LE threads than connections if the threads are also used to perform non-SQL related tasks, or more connections than threads if the application should maintain a pool of active connections but limit the number of active tasks.

Connections can move from one application LE thread to another as the connections in the pool are assigned to child threads, returned to the pool, and assigned again.

The use of this design prevents two LE threads from trying to use the same connection (or an associated statement handle) at the same time. Although DB2 CLI controls access to its internal resources, the application resources such as bound columns, parameter buffers, and files are not controlled by DB2 CLI. If it is necessary for two threads to share an application resource, the application must implement some form of synchronization mechanism. For example, the database copy scenario in Figure 17 on page 367 uses Pthread functions to synchronize use of the shared buffer.

Application Deadlocks

The possibility of timeouts, deadlocks, and general contention for database resources exists when multiple connections are used to access the same database resources concurrently.

An application that creates multiple connections by using multithreading or multiple context support can potentially create deadlocks with shared resources in the database.

A DB2 subsystem can detect deadlocks and rollback one or more connections to resolve them. An application can still deadlock if the following sequence occurs:

- Two LE threads connect to the same data source using two DB2 threads.
- One LE thread holds an internal application resource (such as a mutex) while its DB2 thread waits for access to a database resource.
- The other LE thread has a lock on a database resource while waiting for the internal application resource.

# In this case the DB2 subsystem does not detect a deadlock since the application's
# internal resources cannot be monitored by a DB2 subsystem. However, the
# application is still subject to the DB2 subsystem detecting and handling any DB2
# thread timeouts.

Mixing Embedded SQL and DB2 CLI

It is possible, and sometimes desirable for an application to use DB2 CLI in conjunction with embedded static SQL. Consider the scenario where the application developer wishes to take advantage of the ease of use provided by the DB2 CLI catalog functions and maximize the portion of the application's processing where performance is critical. In order to mix the use of DB2 CLI and embedded SQL, the application must comply to the following rules:

- All connection management and transaction management must be performed completely using either DB2 CLI or embedded SQL. Either the DB2 CLI application performs all the connects and commits/rollback and calls functions written using embedded SQL; or an embedded SQL application performs all the connects and commits/rollback and calls functions written in DB2 CLI which use a null connection (see “Writing a DB2 CLI Stored Procedure” on page 363 for details on null connections).
- Query statement processing must not and cannot straddle across DB2 CLI and embedded SQL interfaces for the same statement; for example, the application cannot open a cursor in an embedded SQL routine, and then call the DB2 CLI SQLFetch() function to retrieve row data.

Since DB2 CLI permits multiple connections, the SQLSetConnection() function must be called prior to making a function call to a routine written in embedded SQL. This allows the application to explicitly specify the connection under which the embedded SQL routine should perform its processing. If the application only ever sets up one connection, or if the application is written entirely in DB2 CLI, then calls to SQLSetConnection() are not needed.

Mixed Embedded SQL and DB2 CLI Example

The following example demonstrates an application that connects to two data sources, and executes both embedded SQL and dynamic SQL using DB2 CLI.

```
/* ... */
    /* allocate an environment handle   */
    SQLAllocEnv(&henv);

    /* Connect to first data source */
    DBconnect(henv, &hdbc[0]);

    /* Connect to second data source */
    DBconnect(henv, &hdbc[1]);

    /*********   Start Processing Step  *************************/
    /* NOTE: at this point there are two active connections */

    /* set current connection to the first database */
    if ( (rc = SQLSetConnection(hdbc[0])) != SQL_SUCCESS )
         printf("Error setting connection 1\n");

    /* call function that contains embedded SQL */
    if ((rc = Create_Tab() ) != 0)
         printf("Error Creating Table on 1st connection, RC=%ld\n", rc);

   /*  Commit transation on connection 1 */
    SQLTransact(henv, hdbc[0], SQL_COMMIT);

    /* set current connection to the second database */
    if ( (rc = SQLSetConnection(hdbc[1])) != SQL_SUCCESS )
         printf("Error setting connection 2\n");

    /* call function that contains embedded SQL */
    if ((rc = Create_Tab() ) != 0)
         printf("Error Creating Table on 2nd connection, RC=%ld\n", rc);

   /*  Commit transation on connection 2 */
    SQLTransact(henv, hdbc[1], SQL_COMMIT);

    /* Pause to allow the existance of the tables to be verified. */
    printf("Tables created, hit Return to continue\n");
    getchar();

    SQLSetConnection(hdbc[0]);
    if (( rc = Drop_Tab() ) != 0)
        printf("Error dropping Table on 1st connection, RC=%ld\n", rc);
   /*  Commit transation on connection 1 */
    SQLTransact(henv, hdbc[0], SQL_COMMIT);

    SQLSetConnection(hdbc[1]);
    if (( rc = Drop_Tab() ) != 0)
        printf("Error dropping Table on 2nd connection, RC=%ld\n", rc);

   /*  Commit transation on connection 2 */
    SQLTransact(henv, hdbc[1], SQL_COMMIT);

    printf("Tables dropped\n");
    /*********   End Processing Step  ****************************/

/* ... */
/*************   Embedded SQL Functions  *******************************
** This would normally be a separate file to avoid having to           *
** keep precompiling the embedded file in order to compile the DB2 CLI *
** section50                                                           *
************************************************************************/

EXEC SQL INCLUDE SQLCA;

int
Create_Tab( )
{

   EXEC SQL CREATE TABLE mixedup
            (ID INTEGER, NAME CHAR(10));

   return( SQLCODE);
}

int
Drop_Tab( )
{
   EXEC SQL DROP TABLE mixedup;

   return( SQLCODE);
}
/* ... */
```

Using Vendor Escape Clauses

The X/Open SQL CAE specification defines an *escape clause* as: “a syntactic mechanism for vendor-specific SQL extensions to be implemented in the framework of standardized SQL”. Both DB2 CLI and ODBC support vendor escape clauses as defined by X/Open.

Note: ODBC defines short forms of vendor escape clauses that are not defined by X/Open.

Currently, escape clauses are used extensively by ODBC to define SQL extensions. DB2 CLI translates the ODBC extensions into the correct DB2 syntax. The `SQLNativeSql()` function can be used to display the resulting syntax.

If an application is only going to access DB2 data sources, then there is no reason to use the escape clauses. If an application is going to access other data sources that offer the same support, but uses different syntax, then the escape clauses increase the portability of the application.

Escape Clause Syntax

The format of an X/Open SQL escape clause definition is:

```
--(*vendor(vendor-identifier),
      product(product-identifier) extended SQL text*)--
```

vendor-identifier Vendor identification that is consistent across all of that vendor's SQL products (for example, IBM).

product-identifier Identifies an SQL product (for example, DB2).

These two parts make up the *SQL-escape-identification*.

Using ODBC Defined SQL Extensions

ODBC has used a vendor escape clause of:

```
--(* vendor(Microsoft), product(ODBC) extended SQL text*)--
```

to define the following SQL extensions (these extensions are not defined by X/Open):

- Extended date, time, timestamp data
- Outer join
- LIKE predicate
- Call stored procedure
- Extended scalar functions
  - Numeric functions
  - String functions
  - System functions

ODBC also defines a shorthand syntax for specifying these extensions:

```
{ extended SQL text }
```

X/Open does not support this shorthand syntax; however, it is widely used by ODBC applications.

ODBC Date, Time, Timestamp Data

The ODBC escape clauses for date, time, and timestamp data are:

```
--(*vendor(Microsoft),product(ODBC) d 'value'*)--
--(*vendor(Microsoft),product(ODBC) t 'value'*)--
--(*vendor(Microsoft),product(ODBC) ts 'value'*)--
```

d indicates *value* is a date in the *yyyy-mm-dd* format,

t indicates *value* is a time in the *hh:mm:ss* format

ts indicates *value* is a timestamp in the *yyyy-mm-dd hh:mm:ss.ffffff* format.

The shorthand syntax for date, time, and timestamp data is:

```
{d 'value'}
{t 'value'}
{ts 'value'}
```

For example, each of the following statements can be used to issue a query against the **EMPLOYEE** table:

```
SELECT * FROM EMPLOYEE
WHERE HIREDATE=--(*vendor(Microsoft),product(ODBC) d '1994-03-29' *)--

SELECT * FROM EMPLOYEE WHERE HIREDATE={d '1994-03-29'}
```

DB2 CLI translates either of the above statements to a DB2 format. `SQLNativeSql()` can be used to return the translated statement.

The ODBC escape clauses for date, time, and timestamp literals can be used in input parameters with a C data type of SQL_C_CHAR.

ODBC Outer Join Syntax

The ODBC escape clause for outer join is:

```
--(*vendor(Microsoft),product(ODBC) oj  outer join*)--
```

where *outer join* is:

```
  table-name  {LEFT | RIGHT | FULL} OUTER JOIN
              {table-name | outer-join}
              ON search-condition
```

Or alternatively, the ODBC shorthand syntax is:

```
{oj outer-join}
```

For example, DB2 CLI translates the following two statements:

```
SELECT * FROM
 --(*vendor(Microsoft),product(ODBC) oj
    T1 LEFT OUTER JOIN T2 ON T1.C1=T2.C3*)--
    WHERE T1.C2>20

SELECT * FROM {oj T1 LEFT OUTER JOIN T2 ON T1.C1=T2.C3}
    WHERE T1.C2>20
```

to IBM's format, which corresponds to the SQL92 outer join syntax.

```
SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.C1=T2.C3 WHERE T1.C2>20
```

Note: Not all servers support outer join. To determine if the current server supports outer joins, call SQLGetInfo() with the SQL_OUTER_JOIN and SQL_OJ_CAPABILITIES options.

LIKE Predicate Escape Clauses

In an SQL LIKE predicate, the metacharacter **%** matches zero or more of any character and the metacharacter _ retry matches any one character. The ESCAPE clause allows the definition of patterns intended to match values that contain the actual percent and underscore characters by preceding them with an escape character. The escape clause ODBC uses to define the LIKE predicate escape character is:

```
--(*vendor(Microsoft),product(ODBC) escape 'escape-character'*)--
```

where *escape-character* is any character supported by the DB2 rules governing the use of the ESCAPE clause. The shorthand syntax for the LIKE predicate escape character is:

```
{escape 'escape-character'}
```

Applications that are not concerned about portability across different vendor DBMS products should pass the ESCAPE clause directly to the data source. To determine when LIKE predicate escape characters are supported by a particular data source, an application should call SQLGetInfo() with the SQL_LIKE_ESCAPE_CLAUSE information type.

Stored Procedure Call Syntax

The ODBC escape clause for calling a stored procedure is:

```
--(*vendor(Microsoft),product(ODBC),
 [?=]call procedure-name[([parameter][,[parameter]]...)]*)--
```

procedure-name
Specifies the name of a procedure stored at the data source.

parameter
Specifies a procedure parameter.

A procedure can have zero or more parameters. The short form syntax is:

```
{[?=]call procedure-name[([parameter][,[parameter]]...)]}
```

(The square brackets ([ ]) indicate optional arguments.

ODBC specifies the optional parameter **?=** to represent the procedure's return value, which, if present, is stored in the location specified by the first parameter marker defined by SQLBindParameter(). DB2 CLI returns the SQLCODE as the procedure's return value if **?=** is present in the escape clause. If **?=** is not present, then the application can retrieve the SQLCA by using the SQLGetSQLCA() function. Unlike ODBC, DB2 CLI does not support literals as procedure arguments. Parameter markers must be used.

For more information about stored procedures, refer to “Using Stored Procedures” on page 361 or *Application Programming and SQL Guide*.

For example, DB2 CLI translates the following two statements:

```
--(*vendor(Microsoft),product(ODBC) CALL NEBT94(?,?,?)*)--
```

```
{CALL NETB94(?,?,?)}
```

To an internal CALL statement format:

```
CALL NEBT94(?, ?, ?)
```

ODBC Scalar Functions

Scalar functions such as string length, substring, or trim can be used on columns of a result sets and on columns that restrict rows of a result set. The ODBC escape clauses for scalar functions and its shorthand are:

```
--(*vendor(Microsoft),product(ODBC) fn scalar-function*)--
```

```
or,
{fn scalar-function}
```

Where, *scalar-function* can be any function listed in Appendix C, “Extended Scalar Functions” on page 405.

For example, Call Level Interface translates both of the following statements:

```
SELECT --(*vendor(Microsoft),product(ODBC) fn CONCAT(FIRSTNAME,LASTNAME) *)--
FROM EMPLOYEE
```

```
SELECT {fn CONCAT(FIRSTNAME,LASTNAME)} FROM EMPLOYEE
```

to:

```
SELECT FIRSTNAME CONCAT LASTNAME FROM EMPLOYEE
```

`SQLNativeSql()` can be called to obtain the translated SQL statement.

To determine which scalar functions are supported by the current server referenced by a specific connection handle, call `SQLGetInfo()` with the SQL_NUMERIC_FUNCTIONS, SQL_STRING_FUNCTIONS, SQL_SYSTEM_FUNCTIONS, and SQL_TIMEDATE_FUNCTIONS options.

Chapter 7. Problem Diagnosis

This section provides guidelines for working with the DB2 CLI traces and information about general diagnosis, debugging, and abends. You can obtain traces for DB2 CLI applications and diagnostics and DB2 CLI stored procedures.

Tracing

DB2 CLI provides two traces that differ in purpose:

- An application trace intended for debugging user applications, described in “Application Trace.”
- A service trace for problem diagnosis, described in “Diagnostic Trace” on page 383.

Application Trace

The DB2 CLI application trace is enabled using the `CLITRACE` and `TRACEFILENAME` keywords in the DB2 CLI initialization file.

The `CLITRACE` keyword is intended for customer application debugging. This trace records data information at the DB2 CLI API interface; it is specifically designed to trace CLI API calls. The trace is written to the file specified on the `TRACEFILENAME` keyword. We strongly recommend that you use this trace

Specifying the Trace File Name

You can use a JCL DD card format or an OS/390 OpenEdition HFS file name format to specify the `TRACEFILENAME` keyword setting. The primary use of the JCL DD card format is write to an MVS preallocated sequential data set. You can also specify OS/390 OpenEdition HFS files on a DD statement. The OS/390 OpenEdition HFS file name format is used strictly for writing to HFS files.

JCL DD Card Format: The JCL DD card format is `TRACEFILENAME="DD:ddname"`. The `ddname` value is the name of the DD card specified in your job or TSO logon procedure.

Examples: Assume the keyword setting is `TRACEFILENAME="DD:APPLDD"`. You can use the following JCL DD statement examples in your job or TSO logon procedure to specify the MVS trace data set.

Example 1: Write to preallocated MVS sequential data set USER01.MYTRACE.

```
//APPLDD   DD      DISP=SHR,DSN=USER01.MYTRACE
```

Example 2: Write to preallocated OS/390 OpenEdition HFS file MYTRACE in directory /usr/db2.

```
//APPLDD        DD      PATH='/usr/db2/MYTRACE'
```

Example 3: Allocate OS/390 OpenEdition HFS file MYTRACE in directory /usr/db2 specifying permission for the file owner to read from (SIRUSR) and write to (SIWUSR) the trace file:

```
//APPLDD     DD  PATH='/usr/db2/MYTRACE',
          PATHOPTS=(ORDWR,OCREAT,OTRUNC),
          PATHMODE=(SIRUSR,SIWUSR)
```

OS/390 OpenEdition HFS File Name Format: The OS/390 OpenEdition HFS file name format is `TRACEFILENAME=hfs_filename`. The `hfs_filename` value specifies the path and file name for the HFS file. The HFS file does not have to be preallocated. If the file name does not exist in the specified directory, the file is dynamically allocated.

Examples: The following examples use the `TRACEFILENAME` keyword to specify an OS/390 OpenEdition HFS trace file.

Example 1: Create and write to HFS file named APPLTRC1 in the fully qualified directory /usr/db2.

```
TRACEFILENAME=/usr/db2/APPLTRC1
```

Example 2: Create and write to HFS file named APPLTRC1 in the current working directory of the application.

```
TRACEFILENAME=./APPLTRC1
```

Example 3: Create and write to HFS file named APPLTRC1 in the parent directory of the current working directory.

```
TRACEFILENAME=../APPLTRC1
```

Application Trace Output

The following example of application trace output shows how DB2 CLI follows the APIs invoked, indicates values used, data pointers, etc. Errors are also indicated.

```
SQLAllocEnv( phEnv=&6b7e77c )
SQLAllocEnv( phEnv=1 )
    ---> SQL_SUCCESS

SQLAllocConnect( hEnv=1, phDbc=&6b7e778 )
SQLAllocConnect( phDbc=1 )
    ---> SQL_SUCCESS

SQLConnect( hDbc=1, szDSN=Null Pointer, cbDSN=0, szUID=Null Pointer, cbUID=0,
szAuthStr=Null Pointer, cbAuthStr=0 )
SQLConnect( )
    ---> SQL_SUCCESS

SQLAllocStmt( hDbc=1, phStmt=&6b7e774 )
SQLAllocStmt( phStmt=1 )
    ---> SQL_SUCCESS

SQLExecDirect( hStmt=1, pszSqlStr="SELECT NAME FROM SYSIBM.SYSPLAN", cbSqlStr=-3 )
SQLExecDirect( )
    ---> SQL_SUCCESS

SQLFetch( hStmt=1 )
SQLFetch( )
    ---> SQL_SUCCESS

SQLTransact( hEnv=1, hDbc=1, fType=SQL_COMMIT )
SQLTransact( )
    ---> SQL_SUCCESS

SQLFreeStmt( hStmt=1, fOption=SQL_DROP )
SQLFreeStmt( )
    ---> SQL_SUCCESS

SQLDisconnect( hDbc=1 )
SQLDisconnect( )
    ---> SQL_SUCCESS

SQLFreeConnect( hDbc=1 )
SQLFreeConnect( )
    ---> SQL_SUCCESS

SQLFreeEnv( hEnv=1 )
SQLFreeEnv( )
    ---> SQL_SUCCESS
```

For more information about how to specify the CLITRACE and TRACEFILENAME keywords, see “DB2 CLI Initialization File” on page 62.

Diagnostic Trace

The DB2 CLI diagnostic trace captures information to use in DB2 CLI problem determination. The trace is intended for use under the direction of the IBM Support Center; it is not intended to assist in debugging user written DB2 CLI applications. You can view this trace to obtain information about the general flow of an application, such as commit information. However, this trace is intended for IBM service information only and is therefore subject to change.

You can activate the diagnostic trace by either the DSNAOTRC command or the

TRACE keyword in the DB2 CLI initialization file.

If you activate the diagnostic trace using the TRACE keyword in the initialization file, you must also allocate a DSNAOTRC DD statement in your job or TSO logon procedure. You can use one of the following methods to allocate a DSNAOTRC DD statement:

- Specify a DSNAOTRC DD JCL statement in your job or TSO logon procedure
- Use the TSO/E ALLOCATE command
- Use dynamic allocation in your CLI application

Specifying the Diagnostic Trace File

The diagnostic trace data can be written to an MVS sequential data set or an OS/390 OpenEdition HFS file.

An MVS data set must be preallocated with the following data set attributes:

- Sequential data set organization
- Fixed-block 80 record format

When you execute an CLI application in OS/390 OpenEdition and activate the diagnostic trace using the TRACE keyword in the initialization file, DB2 writes the diagnostic data to a dynamically allocated file, DD:DSNAOTRC. This file is located in the current working directory of the application if the DSNAOTRC DD statement is not available to the CLI application. You can format DD:DSNAOTRC using the trace formatting program.

Examples: The following JCL examples use a DSNAOTRC JCL DD card to specify the diagnostic trace file.

Example 1: Write to preallocated MVS sequential data set USER01.DIAGTRC.

```
//DSNAOTRC DD      DISP=SHR,DSN=USER01.DIAGTRC
```

Example 2: Write to preallocated OS/390 OpenEdition HFS file DIAGTRC in directory /usr/db2.

```
//DSNAOTRC      DD      PATH='/usr/db2/DIAGTRC'
```

Example 3: Allocate OS/390 OpenEdition HFS file DIAGTRC in directory /usr/db2 specifying permission for the file owner to read from (SIRUSR) and write to (SIWUSR) the trace file.

```
//DSNAOTRC        DD   PATH='/usr/db2/DIAGTRC',
              PATHOPTS=(ORDWR,OCREAT,OTRUNC),
              PATHMODE=(SIRUSR,SIWUSR)
```

For more information about the TRACE keyword, see “DB2 CLI Initialization File” on page 62.

Using the Diagnostic Trace Command: DSNAOTRC

You can use the DSNAOTRC command to:

- Manually start or stop the recording of memory resident diagnostic trace records.
- Query the current status of the diagnostic trace.
- Capture the memory resident trace table to an MVS data set or OS/390 OpenEdition HFS file.

- Format the DB2 CLI diagnostic trace.

Special OS/390 OpenEdition Considerations: You can issue the DSNAOTRC command from the OS/390 OpenEdition shell command line to activate the diagnostic trace prior to executing an CLI application. Under the direction of IBM support only, you must store the `dsnaotrc` program load module in an OS/390 OpenEdition HFS file.

Use the TSO/E command, OPUTX, to store the `dsnaotrc` load module in an HFS file. The following example uses the OPUTX command to store load module `dsnaotrc` from MVS partitioned data set DB2A.DSNLOAD to HFS file DSNAOTRC in directory /usr/db2:

```
OPUTX  'DB2A.DSNLOAD(DSNAOTRC)'  /usr/db2/dsnaotrc
```

After storing the `dsnaotrc` program module in an HFS file, follow these steps at the OS/390 OpenEdition shell to activate, dump, and format the diagnostic trace:

1. Enable the shared address space environment variable for the OS/390 OpenEdition shell. Issue the following `export` statement at the command line or specify it in your $HOME/.profile:

   ```
   export _BPX_SHAREAS=YES
   ```

   Setting this environment variable allows the OMVS command and the OS/390 OpenEdition shell to run in the same TSO address space.
2. Go to the directory that contains the `dsnaotrc` module.
3. Verify that execute permission is established for the `dsnaotrc` load module. If execute permission was not granted, use the `chmod` command to set execute permission for the `dsnaotrc` load module.
4. Issue `dsnaotrc on`. The options for activating the diagnostic trace are optional.
5. Execute the CLI application.
6. Issue `dsnaotrc dmp "raw_trace_file"`. The `raw_trace_file` value is the name of the output file to which DB2 writes the raw diagnostic trace data.
7. Issue `dsnaotrc off` to deactivate the diagnostic trace.
8. Issue `dsnaotrc  fmt  "raw_trace_file"  "fmt_trace_file"` to format the raw trace data records from input file `"raw_trace_file"` to output file `"fmt_trace_file"`.

After successfully formatting the diagnostic trace data, delete the `dsnaotrc` program module from your OS/390 OpenEdition directory. Do not attempt to maintain a private copy of the dsnaotrc program module in your HFS directory.

### Syntax

```
►►──DSNAOTRC───────────────────────────────────────────────────────────►

►──┬─ON──┬──────────────────┬──────────────────────────────┬──────────►◄
   │     ├─ -L──buffer size─┤                              │
   │     └─ -I──buffer size─┘                              │
   ├─OFF───────────────────────────────────────────────────┤
   ├─INF───────────────────────────────────────────────────┤
   ├─DMP──trace data set spec──────────────────────────────┤
   ├─FMT──┬───────────────────────────────────────────────┬┤
   │      └─input data set spec──┬──────────────────────┬─┘│
   │                             └─output data set spec─┘  │
   └─FLW──input data set spec──┬──────────────────────┬────┘
                               └─output data set spec─┘
```

### Option Descriptions

ON
Start the DB2 CLI diagnostic trace.

-L *buffer size*
L = Last. The trace wraps; it captures the last, most current trace records.

buffer size is the number of bytes to allocate for the trace buffer. This value is required. The buffer size is rounded to a multiple of 65536 (64K).

-I *buffer size*
I = Initial. The trace does not wrap; it captures the initial trace records.

buffer size is the number of bytes to allocate for the trace buffer. This value is required. The buffer size is rounded to a multiple of 65536 (64K).

OFF
Stop the DB2 CLI diagnostic trace.

INF
Display information about the currently active DB2 CLI diagnostic trace.

DMP
Dump the currently active DB2 CLI diagnostic trace.

trace data set spec
Specifies the MVS data set or OS/390 OpenEdition HFS file to which DB2 writes the raw DB2 CLI diagnostic trace data. The data set specification can be either an MVS data set name, an OS/390 OpenEdition HFS file name, or a currently allocated JCL DD card name.

FMT
Generate a formatted detail report of the DB2 CLI diagnostic trace contents.

FLW
Generate a formatted flow report of the DB2 CLI diagnostic trace contents.

input data set spec
The data set that contains the raw DB2 CLI diagnostic trace data to be formatted. This is the data set that was generated as the result of a DSNAOTRC DMP command or due to the DSNAOTRC DD card if the trace was started by using the `TRACE` initialization keyword. The data set specification can be either an MVS data set name, an OS/390 OpenEdition

HFS file name, or a currently allocated JCL DD card name. If this parameter is not specified, then DSNAOTRC attempts to format the memory resident DSNAOTRC that is currently active.

output data set spec
The data set to which the formatted DB2 CLI diagnostic trace records are written. The data set specification can be either an MVS data set name, an OS/390 OpenEdition HFS file name, or a currently allocated JCL DD card name. If you specify an MVS data set or OS/390 OpenEdition HFS file that does not exist, DB2 allocates it dynamically. If this parameter is not specified, the output is written to standard output ("`STDOUT`").

Examples: The following examples show how to code the data set specifications.

- Trace data set specification:

  Example 1: Currently allocated JCL DD card name TRACEDD.

  ```
  DSNAOTRC DMP DD:TRACEDD
  ```

  Example 2: MVS sequential data set USER01.DIAGTRC.

  ```
  DSNAOTRC DMP "USER01.DIAGTRC"
  ```

  Example 3: OS/390 OpenEdition HFS file named DIAGTRC in directory /usr/db2:

  ```
  DSNAOTRC DMP "/usr/db2/DIAGTRC"
  ```

- Input data set specification:

  Example 1: Currently allocated JCL DD card name INPDD.

  ```
  DSNAOTRC FLW DD:INPDD output-dataset-spec
  ```

  Example 2: MVS sequential data set USER01.DIAGTRC.

  ```
  DSNAOTRC FLW "USER01.DIAGTRC" output-dataset-spec
  ```

  Example 3: OS/390 OpenEdition HFS file DIAGTRC in directory /usr/db2.

  ```
  DSNAOTRC FLW "/usr/db2/DIAGTRC" output-dataset-spec
  ```

- Output data set specification:

  Example 1: Currently allocated JCL DD card name OUTPDD.

  ```
  DSNAOTRC FLW input-dataset-spec DD:OUTPDD
  ```

  Example 2: MVS sequential data set USER01.TRCFLOW.

  ```
  DSNAOTRC FLW input-dataset-spec "USER01.TRCFLOW"
  ```

  Example 3: OS/390 OpenEdition HFS file TRCFLOW in directory /usr/db2.

  ```
  DSNAOTRC FLW input-dataset-spec "/usr/db2/TRCFLOW"
  ```

Stored Procedure Trace

This section describes the steps required to obtain an application trace or a diagnostic trace of a DB2 CLI stored procedure. DB2 CLI stored procedures run in either a DB2-established stored procedures address space or a WLM-established address space. Both the main application that calls the stored procedure (client application), and the stored procedure itself, can be either a DB2 CLI application or a standard DB2 precompiled application.

If the client application and the stored procedure are DB2 CLI application programs, you can trace:

- A client application only
- A stored procedure only
- Both the client application and stored procedure

More than one address spaces can not share write access to a single data set. Therefore, you must use the appropriate JCL DD statements to allocate a unique trace data set for each stored procedures address space that uses the DB2 CLI application trace or diagnostic trace.

Tracing a Client Application

This section explains how to obtain an application trace and a diagnostic trace for a client application.

Application Trace: Follow these steps to obtain an application trace.

1. Set `CLITRACE=1` and `TRACEFILENAME="DD:DDNAME"` in the common section of the DB2 CLI initialization file as follows:

   ```
   [COMMON]
   CLITRACE=1
   TRACEFILENAME="DD:APPLTRC"
   ```

   `DDNAME` is the name of an OS/390 JCL DD statement specified in the JCL for the application job or your TSO logon procedure.

2. Specify an OS/390 JCL DD statement in the JCL for the application job or your TSO logon procedure. The DD statement references a pre-allocated OS/390 sequential data set with DCB attributes `RECFM=VBA,LRECL=137`, an OS/390 OpenEdition HFS file to contain the client application trace, as shown in the following examples:

   ```
   //APPLTRC DD DISP=SHR,DSN=CLI.APPLTRC
   ```

   ```
   //APPLTRC DD PATH='/u/cli/appltrc'
   ```

Diagnostic Trace: When tracing only the client application, you can activate the diagnostic trace by using the `TRACE` keyword in the DB2 CLI initialization file or the DSNAOTRC command. See “Diagnostic Trace” on page 383 for information about obtaining a diagnostic trace of the client application.

Tracing a Stored Procedure

This section explains how to obtain an application trace and a diagnostic trace for a stored procedure.

Application Trace: Follow these steps to obtain an application trace.

1. Set `CLITRACE=1` and `TRACEFILENAME="DD:DDNAME"` in the common section of the DB2 CLI initialization file as follows:

   ```
   [COMMON]
   CLITRACE=1
   TRACEFILENAME="DD:APPLTRC"
   ```

   `DDNAME` is the name of an OS/390 JCL DD statement specified in the JCL for the stored procedures address space.

2. Specify an OS/390 JCL DD statement in the JCL for the stored procedures address space The DD statement references a pre-allocated OS/390 sequential

data set with DCB attributes `RECFM=VBA,LRECL=137` or OS/390 OpenEdition HFS file to contain the client application trace, as shown in the following examples:

```
//APPLTRC DD DISP=SHR,DSN=CLI.APPLTRC
```

```
//APPLTRC DD PATH='/u/cli/appltrc'
```

Diagnostic Trace: Follow these steps to obtain a diagnostic trace.

1. Set `TRACE=1`, `TRACE_BUFFER_SIZE=nnnnnnn`, and `TRACE_NO_WRAP=0|1` in the common section of the DB2 CLI initialization file. For example:

   ```
   [COMMON]
   TRACE=1
   TRACE_BUFFER_SIZE=2000000
   TRACE_NO_WRAP=1
   ```

   `nnnnnnn` is the number of bytes to allocate for the diagnostic trace buffer.

2. Specify an OS/390 DSNAOINI JCL DD statement in the JCL for the stored procedures address space. The DD statement references the DB2 CLI initialization file, as shown in the following examples:

   ```
   //DSNAOINI DD DISP=SHR,DSN=CLI.DSNAOINI
   ```

   ```
   //DSNAOINI DD PATH='/u/cli/dsnaoini'
   ```

3. Specify an OS/390 DSNAOTRC JCL DD statement in the JCL for the stored procedures space. The DD statement references a pre-allocated OS/390 sequential data set with DCB attributes `RECFM=FB,LRECL=80`, or an OS/390 OpenEdition HFS file to contain the unformatted diagnostic data, as shown in the following examples:

   ```
   //DSNAOTRC DD DISP=SHR,DSN=CLI.DIAGTRC
   ```

   ```
   //DSNAOTRC DD PATH='/u/cli/diagtrc'
   ```

4. Execute the client application that calls the stored procedure.

5. After the DB2 CLI stored procedure executes, stop the stored procedures address space.

   - For DB2-established address spaces, use the DB2 command, STOP PROCEDURE.

   - For WLM-established address spaces operating in WLM goal mode, use the MVS command, "`VARY WLM,APPLENV=name,QUIESCE`". `name` is the WLM application environment name.

   - For WLM-established address spaces operating in WLM compatibility mode, use the MVS command, "`CANCEL address-space-name`". `address-space-name` is the name of the WLM-established address space.

6. You can submit either the formatted or unformatted diagnostic trace data to the IBM Support Center. To format the raw trace data at your site, run the DSNAOTRC FMT or DSNAOTRC FLW command against the diagnostic trace data set.

Tracing both a Client Application and a Stored Procedure

This section explains how to obtain an application trace and a diagnostic trace for both a client application and a stored procedure.

Application Trace: Follow these steps to obtain an application trace.

1. Set `CLITRACE=1` and `CLITRACEFILENAME="DD:DDNAME"` in the common section of the DB2 CLI initialization file as follows:

   ```
   [COMMON]
   CLITRACE=1
   TRACEFILENAME="DD:APPLTRC"
   ```

   `DDNAME` is the name of an OS/390 JCL DD statement specified in both the JCL for the client application job and the stored procedures address space.

2. Specify an OS/390 JCL DD statement in the JCL for the client application. The DD statement references a pre-allocated OS/390 sequential data set with DCB attributes `RECFM=VBA,LRECL=137`, or an OS/390 OpenEdition HFS file to contain the client application trace, as shown in the following examples:

   ```
   //APPLTRC DD DISP=SHR,DSN=CLI.APPLTRC.CLIENT
   ```

   ```
   //APPLTRC DD PATH='/u/cli/appltrc.client'
   ```

   You must allocate a separate application trace data set, or an HFS file for the client application. Do not attempt to write to the same application trace data set or HFS file used for the stored procedure.

3. Specify an OS/390 JCL DD statement in the JCL for the stored procedures address space. The DD statement references a pre-allocated OS/390 sequential data set, or an OS/390 OpenEdition HFS file to contain the stored procedure application trace, as shown in the following examples:

   ```
   //APPLTRC DD DISP=SHR,DSN=CLI.APPLTRC.SPROC
   ```

   ```
   //APPLTRC DD PATH='/u/cli/appltrc.sproc'
   ```

   You must allocate a separate trace data set or HFS file for the stored procedure. Do not attempt to write to the same application trace data set or HFS file used for the client application.

Diagnostic Trace: Follow these steps to obtain a diagnostic trace.

1. Set `TRACE=1`, `TRACE_BUFFER_SIZE=nnnnnnn`, and `TRACE_NO_WRAP=0|1` in the common section of the DB2 CLI initialization file. For example:

   ```
   [COMMON]
   TRACE=1
   TRACE_BUFFER_SIZE=2000000
   TRACE_NO_WRAP=1
   ```

   `nnnnnnn` is the number of bytes to allocate for the diagnostic trace buffer.

2. Specify an OS/390 DSNAOINI JCL DD statement in the JCL for the stored procedures address space. The DD statement references the DB2 CLI initialization file, as shown in the following examples:

   ```
   //DSNAOINI DD DISP=SHR,DSN=CLI.DSNAOINI
   ```

   ```
   //DSNAOINI DD PATH='/u/cli/dsnaoini'
   ```

3. Specify an OS/390 DSNAOTRC JCL DD statement in JCL for the client application job. The DD statement references a pre-allocated OS/390 sequential data set with DCB attributes `RECFM=FB,LRECL=80`, or an OS/390

OpenEdition HFS file to contain the unformatted diagnostic data, as shown in the following examples:

```
//DSNAOTRC DD DISP=SHR,DSN=CLI.DIAGTRC.CLIENT
```

```
//DSNAOTRC DD PATH='/u/cli/diagtrc.client'
```

4. Specify an OS/390 DSNAOTRC JCL DD statement in the JCL for the stored procedures address space. The DD statement references a pre-allocated OS/390 sequential data set with DCB attributes `RECFM=FB,LRECL=80`, or an OS/390 OpenEdition HFS file to contain the stored procedure's unformatted diagnostic data, as shown in the following examples:

   ```
   //DSNAOTRC DD DISP=SHR,DSN=CLI.DIAGTRC.SPROC
   ```

   ```
   //DSNAOTRC DD PATH='/u/cli/diagtrc.sproc'
   ```

5. Execute the client application that calls the stored procedure.

6. After the DB2 CLI stored procedure executes, stop the stored procedures address space.

   - For DB2-established address spaces, use the DB2 command, STOP PROCEDURE.
   - For WLM-established address spaces operating in WLM goal mode, use the MVS command, "`VARY WLM,APPLENV=name,QUIESCE`". `name` is the WLM application environment name.
   - For WLM-established address spaces operating in WLM compatibility mode, use the MVS command, "`CANCEL address-space-name`". `address-space-name` is the name of the WLM-established address space.

7. You can submit either the formatted or unformatted diagnostic trace data to the IBM Support Center. To format the raw trace data at your site, run the DSNAOTRC FMT or DSNAOTRC FLW command against the client application's diagnostic trace data set and the stored procedure's diagnostic trace data set.

Debugging

You can debug DB2 for OS/390 CLI applications debug tool shipped with your the C or C^{++} language compiler. For detailed instructions on debugging DB2 stored procedures, including DB2 CLI stored procedures, see Section 6 of *Application Programming and SQL Guide*.

Abnormal Termination

Language Environment reports abends since DB2 CLI runs under Language Environment. Typically, Language Environment reports the type of abend that occurs and the function that is active in the address space at the time of the abend.

DB2 CLI has no facility for abend recovery. When an abend occurs, DB2 CLIterminates. DBMSs follow the normal recovery process for any outstanding DB2 unit of work.

"CEE" is the prefix for all Language Environment messages. If the prefix of the active function is "CLI", then DB2 CLI had control during the abend which indicates that this can be a DB2 CLI, a DB2, or a user error.

The following example shows an abend:

```
CEE3250C The system or user abend S04E  R=00000000 was issued.
      From entry point CLI_mvsCallProcedure(CLI_CONNECTINFO*,...
      +091A2376 at address 091A2376...
```

In this message, you can determine what caused the abend as follows:

- "CEE" indicates that Language Environment is reporting the abend.
- The entry point shows that DB2 CLI is the active module.
- Abend code "S04E" means that this is a DB2 system abend.

For further information on debugging, see *OS/390 Language Environment for OS/390 & VM Debugging Guide*. For further information on the DB2 recovery process, see Section 4 (Volume 1) of *Administration Guide*.

Internal Error Code

DB2 CLI provides an internal error code for CLI diagnosis that is intended for use under the guidance of IBM service. This unique error location, ERRLOC, is a good tool for APAR searches. The following example of a failed `SQLAllocConnect()` shows an error location:

```
DB2 CLI Sample SQLError Information
DB2 CLI Sample SQLSTATE          : 58004
DB2 CLI Sample Native Error Code : -99999
DB2 CLI Sample Error message text:
  {DB2 for OS/390}{CLI Driver}  SQLSTATE=58004  ERRLOC=2:170:4;
  RRS "IDENTIFY" failed using DB2 system:V61A,
  RC=08 and REASON=00f30091
```

Appendix A. Programming Hints and Tips

This section provides some hints and tips to help improve DB2 CLI and ODBC application performance and portability.

Avoiding Common Initialization File Problems

You can avoid two common problems when using the DB2 CLI initialization file by ensuring that these contents are accurate.

Square brackets

The square brackets in the initialization file must consist of the correct EBCDIC characters. The open square bracket must use the hexadecimal characters `X'AD'`. The close square bracket must use the hexadecimal characters `X'BD'`. DB2 CLI does not recognize brackets if coded differently.

Sequence numbers

The initialization file cannot accept sequence numbers. All sequence numbers must be removed.

Setting Common Connection Options

The following connection options might need to be set (or considered) by DB2 CLI applications.

SQL_AUTOCOMMIT

Generally this option should be set to SQL_AUTOCOMMIT_OFF, since each commit request can generate extra network flow. Only leave SQL_AUTOCOMMIT on if specifically needed.

Note: The default is SQL_AUTOCOMMIT_ON.

SQL_TXN_ISOLATION

This connection (and statement) option determines the isolation level at which the connection or statement operate. The isolation level determines the level of concurrency possible, and the level of locking required to execute the statement. Applications need to choose an isolation level that maximizes concurrency, yet ensures data consistency.

See Section 4 of *Application Programming and SQL Guide* for a complete discussion of isolation levels and their effect.

Setting Common Statement Options

The following statement options might need to be set by DB2 CLI applications.

SQL_MAX_ROWS

Setting this option limits the number of rows returned to the application. This can be used to avoid an application from being overwhelmed with a very large result set generated inadvertently, especially for applications on clients with limited memory resources.

Note: The full result set is still generated at the server. DB2 CLI only fetches up to SQL_MAX_ROWS rows.

SQL_CURSOR_HOLD

This statement option determines if the cursor for this statement is defined with the equivalent of the CURSOR WITH HOLD clause.

Resources associated with statement handles can be better utilized by DB2 CLI if the statements that do not require CURSOR WITH HOLD are set to SQL_CURSOR_HOLD_OFF.

Note: Many ODBC applications expect a default behavior where the cursor position is maintained after a commit.

SQL_STMTTXN_ISOLATION

DB2 CLI allows the isolation level to be set at the statement level (however, we recommend that the isolation level be set at the connection level). The isolation level determines the level of concurrency possible, and the level of locking required to execute the statement.

Resources associated with statement handles can be better utilized by DB2 CLI if statements are set to the required isolation level, rather than leaving all statements at the default isolation level. This should only be attempted with a thorough understanding of the locking and isolation levels of the connected DBMS. Refer to *SQL Reference* for a complete discussion of isolation levels and their effect.

Applications should use the minimum isolation level possible to maximize concurrency.

Using SQLSetColAttributes() to Reduce Network Flow

Each time a query statement is prepared or executed directly, DB2 CLI retrieves information about the SQL data type, and its size from the data source. If the application *knows* this information ahead of time, `SQLSetColAttributes()` can be used to provide DB2 CLI with the information. This can significantly reduce the network flow from remote data sources if the result set coming back contains a very large number (hundreds) of columns.

Note: The application must provide DB2 CLI with exact result descriptor information for ALL columns; otherwise, an error occurs when the data is fetched.

Queries that generate result sets that contain a large number of columns, but relatively small number of rows, have the most to gain from using `SQLSetColAttributes()`.

Comparing Binding and SQLGetData

Generally it is more efficient to bind application variables to result sets than to use `SQLGetData()`. Use `SQLGetData()` when the data value is large variable-length data that:

- Must be received in pieces, or
- Might not need to be retrieved (dependent on another application action.)

Increasing Transfer Efficiency

The efficiency of transferring character data between bound application variables and DB2 CLI can be increased if the *pcbValue* and *rgbValue* arguments are contiguous in memory. (This allows DB2 CLI to fetch both values with one copy operation.)

For example:

```
struct {  SQLINTEGER  pcbValue;
          SQLCHAR     rgbValue[MAX_BUFFER];
       } column;
```

Limiting Use of Catalog Functions

In general, try to limit the number of times the catalog functions are called, and limit the number of rows returned.

The number of catalog function calls can be reduced by calling the function once, and storing the information at the application.

The number of rows returned can be limited by specifying a:

- Schema name or pattern for all catalog functions
- Table name or pattern for all catalog functions other than `SQLTables`
- Column name or pattern for catalog functions that return detailed column information.

Remember, although an application can be developed and tested against a data source with hundreds of tables, it can be run against a database with thousands of tables. Plan ahead.

Close any open cursors (call `SQLFreeStmt()` with SQL_CLOSE) for statement handles used for catalog queries to release any locks against the catalog tables. Outstanding locks on the catalog tables can prevent CREATE, DROP or ALTER statements from executing.

Using Column Names of Function Generated Result Sets

The column names of the result sets generated by catalog and information functions can change as the X/Open and ISO standards evolve. The *position* of the columns, however, does not change.

Any application dependency should be based on the column position (*icol* parameter) and not the name.

Making use of Dynamic SQL Statement Caching

To make use of *dynamic caching* (when the server caches a prepared version of a dynamic SQL statement), the application must use the same statement handle for the same SQL statement.

For example, if an application routinely uses a set of 10 SQL statements, 10 statement handles should be allocated and associated with each of those statements. Do not free the statement handle while the statement can still be executed. (The transaction can still be rolled back or committed without affecting any of the prepared statements). The application continues to prepare and execute the statements in a normal manner. DB2 CLI determines if the prepare is actually needed.

To reduce function call overhead, the statement can be prepared once, and executed repeatedly throughout the application.

Optimizing Insertion and Retrieval of Data

The methods described in “Using Arrays to Input Parameter Values” on page 353 and “Retrieving A Result Set Into An Array” on page 357 optimize the network flow.

Use these methods as much as possible.

Using SQLDriverConnect Instead of SQLConnect

| Using `SQLDriverConnect()` gives an application the flexibility to override any or all
| of the initialization keyword values specified for the target data source.

Turning Off Statement Scanning

DB2 CLI by default, scans each SQL statement searching for vendor escape clause sequences.

If the application does not generate SQL statements that contain vendor escape clause sequences (“Using Vendor Escape Clauses” on page 376), then the SQL_NO_SCAN statement option should be set to SQL_NOSCAN_ON at the connection level so that DB2 CLI does not perform a scan for vendor escape clauses.

Problem Solving and Debugging

This section provides guidelines for working with traces and abends.

Use of Trace Keywords

The DB2 CLI initialization file contains two trace keywords: TRACE and CLITRACE. These keywords differ in purpose.

TRACE

The TRACE keyword in the DB2 CLI initialization file is a service trace that IBM uses for problem diagnosis. You can view this trace to obtain information about the general flow of an application, such as commit information. However, this trace is intended for IBM service information only and is therefore subject to change.

CLITRACE

The CLITRACE keyword in the DB2 CLI initialization file is intended for customer application debugging. This trace records data information at the DB2 CLI API interface.

Abnormal Termination

Language Environment reports abends since DB2 CLI runs under Language Environment. Typically, Language Environment reports the type of abend that occurs and the function that is active in the address space at the time of the abend.

DB2 CLI has no facility for abend recovery. When an abend occurs, DB2 CLI terminates. DBMSs follow the normal recovery process for any outstanding DB2 unit of work.

"CEE" is the prefix for all all Language Environment messages. If the prefix of the active function is "CLI", then DB2 CLI had control during the abend which indicates that this can be a DB2 CLI, a DB2, or a user error.

The following example shows an abend:

```
CEE3250C The system or user abend S04E  R=00000000 was issued.
         From entry point CLI_mvsCallProcedure(CLI_CONNECTINFO*,...
         +091A2376 at address 091A2376...
```

In this message, you can determine what caused the abend as follows:

- "CEE" indicates that Language Environment is reporting the abend.
- The entry point shows that DB2 CLI is the active module.
- Abend code "S04E" means that this is a DB2 system abend.

For further information on debugging, see *Language Environment for MVS & VM Debugging and Run-Time Messages Guide*. For further information on the DB2 recovery process, see Section 4 (Volume 1) of *Administration Guide*.

Appendix B. DB2 CLI and ODBC

This appendix discusses the support provided by the DB2 ODBC driver, and how it differs from DB2 CLI.

Figure 21 below compares DB2 CLI and the DB2 ODBC driver.

1. An ODBC driver under the ODBC driver manager
2. DB2 CLI, callable interface designed for DB2 specific applications.

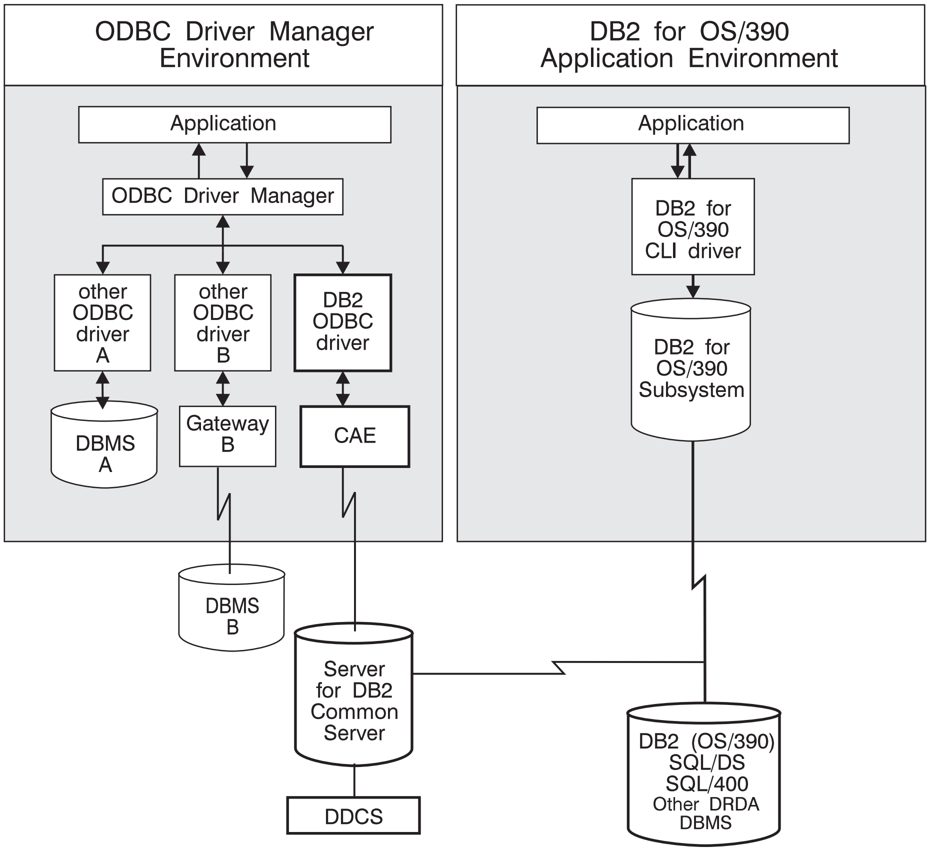
ODBC Driver Manager Environment
DB2 for OS/390 Application Environment
Application
ODBC Driver Manager
other ODBC driver A
other ODBC driver B
DB2 ODBC driver
DBMS A
Gateway B
CAE
Application
DB2 for OS/390 CLI driver
DB2 for OS/390 Subsystem
DBMS B
Server for DB2 Common Server
DDCS
DB2 (OS/390) SQL/DS SQL/400 Other DRDA DBMS

Figure 21. DB2 CLI and ODBC.

In an ODBC environment, the driver manager provides the interface to the application. It also dynamically loads the necessary *driver* for the database server to which the application connects. It is the driver that implements the ODBC function set, with the exception of some extended functions implemented by the driver manager.

The DB2 CLI driver does not execute in this environment. Rather, DB2 CLI is a self-sufficient driver which supports a subset of the functions provided by the ODBC driver.

DB2 CLI applications interface directly with the CLI driver which executes within the application address space. Applications do not interface with a driver manager. The capabilities provided to the application are a subset of the Microsoft ODBC Version 2 specifications.

The following sections compare DB2 CLI support with the ODBC Version 2.0 support.

- “ODBC APIs and Data Types”
- “ODBC Function List” on page 401

In addition “Isolation Levels” on page 404 compares IBM with ODBC isolation levels.

ODBC APIs and Data Types

Table 140 summarizes the ODBC Version 2 application programming interfaces, ODBC SQL data types and ODBC C data types and whether those functions, and data types are supported by DB2 CLI. Table 141 on page 401 provides a complete list of ODBC 2.0 functions, and indicates if they are supported.

Table 140 (Page 1 of 2). DB2 CLI ODBC Support

ODBC Features	**DB2 CLI**
Core Level Functions	All
Level 1 Functions	All
Level 2 Functions	All, except for: • SQLBrowseConnect() • SQLSetPos() • SQLSetScrollOptions()
Additional DB2 CLI Functions	• SQLCancel() • SQLSetConnection() • SQLGetEnvAttr() • SQLSetEnvAttr() • SQLSetColAttributes() • SQLGetSQLCA()
Minimum SQL Data Types	• SQL_CHAR • SQL_LONGVARCHAR • SQL_VARCHAR
Core SQL Data Types	• SQL_DECIMAL • SQL_NUMERIC • SQL_SMALLINT • SQL_INTEGER • SQL_REAL • SQL_FLOAT • SQL_DOUBLE
Extended SQL Data Types	• SQL_BIT • SQL_TINYINT • SQL_BIGINT (NOT SUPPORTED) • SQL_BINARY • SQL_DATE • SQL_LONGVARBINARY • SQL_TIME • SQL_TIMESTAMP • SQL_VARBINARY
ODBC Version 3 SQL Data Types	• SQL_GRAPHIC • SQL_LONGVARGRAPHIC • SQL_VARGRAPHIC

Table 140 (Page 2 of 2). DB2 CLI ODBC Support

ODBC Features	DB2 CLI
Core C Data Types	• SQL_C_CHAR • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG(SLONG, ULONG) • SQL_C_SHORT (SSHORT, USHORT)
Extended C Data Types	• SQL_C_BINARY • SQL_C_BIT • SQL_C_DATE • SQL_C_TIME • SQL_C_TIMESTAMP • SQL_C_TINYINT
ODBC Version 3 C Data Types	• SQL_C_DBCHAR (SQL_C_WCHAR)
Return Codes	• SQL_SUCCESS • SQL_SUCCESS_WITH_INFO • SQL_NEED_DATA • SQL_NO_DATA_FOUND • SQL_ERROR • SQL_INVALID_HANDLE
SQLSTATES	Mapped to X/Open SQLSTATES with additional IBM SQLSTATES
Multiple connections per application	Supported but Type 1 connections, SQL_CONNECTTYPE = SQL_CONCURRENT_TRANS. Must be on a transaction boundary prior to an SQLConnect or SQLSetConnection.

For more information on ODBC, refer to *ODBC 2.0 Programmer's Reference and SDK Guide*.

ODBC Function List

The following table is a complete list of all Microsoft's ODBC 2.0 functions. The ODBC conformance level and whether it is supported by DB2 CLI is shown for each function. For a complete list of DB2 CLI functions and information about X/Open and ISO callable SQL standards, see "DB2 CLI Function Summary" on page 74. xproc=display. proc=display.

Table 141 (Page 1 of 4). ODBC Function List

Task Function Name	Conformance	DB2 CLI Support	Purpose
Note: The DB2 CLI Support column indicates the **first** Version in which the function was supported.			
Connecting to a Data Source			
SQLAllocEnv	Core	V5	Obtains an environment handle. One environment handle is used for one or more connections.
SQLAllocConnect	Core	V5	Obtains a connection handle.
SQLConnect	Core	V5	Connects to specific driver by data source name, user ID, and password.

Table 141 (Page 2 of 4). ODBC Function List

Task Function Name	Conformance	DB2 CLI Support	Purpose
SQLDriverConnect	Level 1	V5	Connects to a specific driver by connection string or requests that the driver manager and driver display connection dialogs for the user. **Note:** This function is also extended by the additional IBM keywords supported in the ODBC.INI file in the DB2 for common server CLI environment. Within the DB2 for OS/390 V5 CLI environment, there is no equivalent of the ODBC.INI file.
SQLBrowseConnect	Level2	No	Returns successive levels of connection attributes and valid attribute values. When a value is specified for each connection attribute, connects to the data source.
Obtaining Information about a Driver and Data Source			
SQLDataSources	Level 2	V5	Returns the list of available data sources.
SQLDrivers	Level 2	No	Returns the list of installed drivers and their attributes (ODBC 2.0). **Note:** This function is implemented within the ODBC driver manager and is therefore not applicable within the DB2 for OS/390 V5 CLI environment.
SQLGetInfo	Level 1	V5	Returns information about a specific driver and data source.
SQLGetFunctions	Level 1	V5	Returns supported driver functions.
SQLGetTypeInfo	Level 1	V5	Returns information about supported data types.
Setting and Retrieving Driver Options			
SQLSetConnectOption	Level 1	V5	Sets a connection option.
SQLGetConnectOption	Level 1	V5	Returns the value of a connection option.
SQLSetStmtOption	Level 1	V5	Sets a statement option.
SQLGetStmtOption	Level 1	V5	Returns the value of a statement option.
Preparing SQL Requests			
SQLAllocStmt	Core	V5	Allocates a statement handle.
SQLPrepare	Core	V5	Prepares an SQL statement for later execution.
SQLBindParameter	Level 1	V5	Assigns storage for a parameter in an SQL statement (ODBC 2.0)
SQLSetParam	Core	V5	Assigns storage for a parameter in an SQL statement (ODBC 2.0). In ODBC, SQLBindParameter replaces this function.
SQLParamOptions	Level 2	V5	Specifies the use of multiple values for parameters.
SQLGetCursorName	Core	V5	Returns the cursor name associated with a statement handle.
SQLSetCursorName	Core	V5	Specifies a cursor name.
SQLSetScrollOptions	Level 2	No	Sets options that control cursor behavior.
Submitting Requests			
SQLExecute	Core	V5	Executes a prepared statement.
SQLExecDirect	Core	V5	Executes a statement.

Table 141 (Page 3 of 4). ODBC Function List

Task Function Name	Conformance	DB2 CLI Support	Purpose
SQLNativeSql	Level 2	V5	Returns the text of an SQL statement as translated by the driver.
SQLDescribeParam	Level 2	No	Returns the description for a specific parameter in a statement.
SQLNumParams	Level 2	V5	Returns the number of parameters in a statement.
SQLParamData	Level 1	V5	Used in conjunction with `SQLPutData()` to supply parameter data at execution time. (Useful for long data values.)
SQLPutData	Level 1	V5	Send part or all of a data value for a parameter. (Useful for long data values.)
Retrieving Results and Information about Results			
SQLRowCount	Core	V5	Returns the number of rows affected by an insert, update, or delete request.
SQLNumResultCols	Core	V5	Returns the number of columns in the result set.
SQLDescribeCol	Core	V5	Describes a column in the result set.
SQLColAttributes	Core	V5	Describes attributes of a column in the result set.
SQLBindCol	Core	V5	Assigns storage for a result column and specifies the data type.
SQLFetch	Core	V5	Returns a result row.
SQLExtendedFetch	Level 2	V5	Returns multiple result rows.
SQLGetData	Level 1	V5	Returns part or all of one column of one row of a result set. (Useful for long data values.)
SQLSetPos	Level 2	No	Positions a cursor within a fetched block of data.
SQLMoreResults	Level 2	V5	Determines whether there are more result sets available and, if so, initializes processing for the next result set.
SQLError	Core	V5	Returns additional error or status information.
Obtaining information about the data source's system tables (catalog functions)			
SQLColumnPrivileges	Level 2	V5	Returns a list of columns and associated privileges for a table.
SQLColumns	Level 1	V5	Returns the list of column names in specified tables.
SQLForeignKeys	Level 2	V5	Returns a list of column names that comprise foreign keys, if they exist for a specified table.
SQLPrimaryKeys	Level 2	V5	Returns the list of column names that comprise the primary key for a table.
SQLProcedureColumns	Level 2	V5	Returns the list of input and output parameters, as well as the columns that make up the result set for the specified procedures.
SQLProcedures	Level 2	V5	Returns the list of procedure names stored in a specific data source.
SQLSpecialColumns	Level 1	V5	Returns information about the optimal set of columns that uniquely identifies a row in a specified table, or the columns that are automatically updated when any value in the row is updated by a transaction.

Table 141 (Page 4 of 4). ODBC Function List

Task Function Name	Conformance	DB2 CLI Support	Purpose
SQLStatistics	Level 1	V5	Returns statistics about a single table and the list of indexes associated with the table.
SQLTablePrivileges	Level 2	V5	Returns a list of tables and the privileges associated with each table.
SQLTables	Level 1	V5	Returns the list of table names stored in a specific data source.
Terminating a Statement			
SQLFreeStmt	Core	V5	End statement processing and closes the associated cursor, discards pending results, and, optionally, frees all resources associated with the statement handle.
SQLCancel	Core	Yes	Cancels an SQL statement.
SQLTransact	Core	V5	Commits or rolls back a transaction.
Terminating a Connection			
SQLDisconnect	Core	V5	Closes the connection.
SQLFreeConnect	Core	V5	Releases the connection handle.
SQLFreeEnv	Core	V5	Releases the environment handle.

Isolation Levels

The following table maps IBM RDBMs isolation levels to ODBC transaction isolation levels. The `SQLGetInfo()` function, indicates which isolation levels are available.

Table 142. Isolation Levels Under ODBC

IBM Isolation Level	ODBC Isolation Level
Cursor Stability	SQL_TXN_READ_COMMITTED
Repeatable Read	SQL_TXN_SERIALIZABLE_READ
Read Stability;	SQL_TXN_REPEATABLE_READ
Uncommitted Read	SQL_TXN_READ_UNCOMMITTED
No Commit	(no equivalent in ODBC)

Note: `SQLSetConnectOption()` and `SQLSetStmtOption` return SQL_ERROR with an SQLSTATE of **S1**009 if you try to set an unsupported isolation level.

Appendix C. Extended Scalar Functions

The following functions are defined by ODBC using vendor escape clauses. Each function can be called using the escape clause syntax, or calling the equivalent DB2 function.

These functions are presented in the following categories:

- "String Functions"
- "Date and Time Functions" on page 406
- "System Functions" on page 407

For more information about vendor escape clauses, refer to "ODBC Scalar Functions" on page 379.

The tables in the following sections indicate for which servers (and the earliest versions) that the function can be accessed when called from an application using DB2 CLI.

All errors detected by the following functions, when connected to a DB2 for common server Version 2 server, return SQLSTATE 38552. The text portion of the message is of the form SYSFUN:*nn* where *nn* is one of the following reason codes:

01 Numeric value out of range
02 Division by zero
03 Arithmetic overflow or underflow
04 Invalid date format
05 Invalid time format
06 Invalid timestamp format
07 Invalid character representation of a timestamp duration
08 Invalid interval type (must be one of 1, 2, 4, 8, 16, 32, 64, 128, 256)
09 String too long
10 Length or position in string function out of range
11 Invalid character representation of a floating point number

String Functions

The string functions in this section are supported by DB2 CLI and defined by ODBC using vendor escape clauses.

Note:

- Character string literals used as arguments to scalar functions must be bounded by single quotes.
- Arguments denoted as *string_exp* can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as SQL_CHAR, SQL_VARCHAR, or SQL_LONGVARCHAR. .
- Arguments denoted as *start, length, code* or *count* can be a numeric literal or the result of another scalar function, where the underlying data type is integer based (SQL_SMALLINT, SQL_INTEGER).
- The first character in the string is considered to be at position 1.

ASCII(*string_exp* **)**
Returns the ASCII code value of the leftmost character of *string_exp* as an integer.

CONCAT(*string_exp1*, *string_exp2* **)**
Returns a character string that is the result of concatenating *string_exp2* to *string_exp1*.

INSERT(*string_exp1, start, length, string_exp2* **)**
Returns a character string where *length* number of characters beginning at *start* is replaced by *string_exp2* which contains *length* characters.

LEFT(*string_exp,count* **)**
Returns the leftmost *count* of characters of *string_exp*.

LENGTH(*string_exp* **)**
Returns the number of characters in *string_exp*, excluding trailing blanks and the string termination character.

REPEAT(*string_exp*, *count* **)**
Returns a character string composed of *string_exp* repeated *count* times.

RIGHT(*string_exp*, *count* **)**
Returns the rightmost count of characters of *string_exp*.

SUBSTRING(*string_exp*, *start*, *length* **)**
Returns a character string that is derived from *string_exp* beginning at the character position specified by *start* for *length* characters.

Date and Time Functions

The date and time functions in this section are supported by DB2 CLI and defined by ODBC using vendor escape clauses.

Note:

- Arguments denoted as *timestamp_exp* can be the name of a column, the result of another scalar function, or a time, date, or timestamp literal.
- Arguments denoted as *date_exp* can be the name of a column, the result of another scalar function, or a date or timestamp literal, where the underlying data type can be character based, or date or timestamp based.
- Arguments denoted as *time_exp* can be the name of a column, the result of another scalar function, or a time or timestamp literal, where the underlying data types can be character based, or time or timestamp based.

CURDATE()
Returns the current date as a date value.

CURTIME()
Returns the current local time as a time value.

DAYOFMONTH (*date_exp* **)**
Returns the day of the month in *date_exp* as an integer value in the range of 1-31.

HOUR(*time_exp* **)**
Returns the hour in *time_exp* as an integer value in the range of 0-23.

MINUTE(*time_exp* **)**
Returns the minute in *time_exp* as integer value in the range of 0-59.

MONTH(*date_exp* **)**
Returns the month in *date_exp* as an integer value in the range of 1-12.

NOW()
Returns the current date and time as a timestamp value.

SECOND(*time_exp* **)**
Returns the second in *time_exp* as an integer value in the range of 0-59.

System Functions

The system functions in this section are supported by DB2 CLI and defined by ODBC using vendor escape clauses.

- Arguments denoted as *exp* can be the name of a column, the result of another scalar function, or a literal.
- Arguments denoted as *value* can be a literal constant.

DATABASE()
Returns the name of the database corresponding to the connection handle (*hdbc*). (The name of the database is also available via `SQLGetInfo()` by specifying the information type SQL_DATABASE_NAME.)

IFNULL(*exp***,** *value* **)**
If *exp* is null, *value* is returned. If *exp* is not null, *exp* is returned. The possible data types of *value* must be compatible with the data type of *exp*.

USER()
Returns the user's authorization name. (The user's authorization name is also available via `SQLGetInfo()` by specifying the information type SQL_USER_NAME.)

Appendix D. Appendix D. SQLSTATE Cross Reference

This table is a cross-reference of all the SQLSTATEs listed in the 'Diagnostics' section of each function description in "Chapter 5. Functions" on page 73.

Note: DB2 CLI can also return SQLSTATEs generated by the server that are not listed in this table. If the returned SQLSTATE is not listed here, refer to the documentation for the server for additional SQLSTATE information.

Table 143 (Page 1 of 10). SQLSTATE Cross Reference

SQLSTATE	Description	Functions
01000	Warning.	• SQLSetConnectOption()
01002	Disconnect error.	• SQLDisconnect()
01004	Data truncated.	• SQLColAttributes() • SQLDataSources() • SQLDescribeCol() • SQLDriverConnect() • SQLExecDirect() • SQLExecute() • SQLExtendedFetch() • SQLFetch() • SQLGetCursorName() • SQLGetData() • SQLGetInfo() • SQLNativeSql() • SQLPutData() • SQLSetColAttributes()
01504	The UPDATE or DELETE statement does not include a WHERE clause.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
01508	Statement disqualified for blocking.	• SQLExecDirect() • SQLExecute() • SQLPrepare()
01S00	Invalid connection string attribute.	• SQLDriverConnect()
01S01	Error in row.	• SQLExtendedFetch()
01S02	Option value changed	• SQLDriverConnect() • SQLSetConnectOption() • SQLSetEnvAttr() • SQLSetStmtOption()
07001	Wrong number of parameters.	• SQLExecDirect() • SQLExecute() • SQLParamData()
07002	Too many columns.	• SQLExtendedFetch() • SQLFetch()
07005	The statement did not return a result set.	• SQLColAttributes() • SQLDescribeCol()
07006	Invalid conversion.	• SQLBindParameter() • SQLExecDirect() • SQLExecute() • SQLParamData() • SQLExtendedFetch() • SQLFetch() • SQLGetData() • SQLSetParam()
08001	Unable to connect to data source.	• SQLConnect() • SQLDriverConnect()

Table 143 (Page 2 of 10). SQLSTATE Cross Reference

SQLSTATE	Description	Functions
08002	Connection in use.	• SQLConnect() • SQLDriverConnect()
08003	Connection is closed.	• SQLAllocStmt() • SQLDisconnect() • SQLGetConnectOption() • SQLGetInfo() • SQLNativeSql() • SQLSetConnection() • SQLSetConnectOption() • SQLTransact()
08004	The application server rejected establishment of the connection.	• SQLConnect() • SQLDriverConnect()
08007	Connection failure during transaction.	• SQLTransact()
21S01	Insert value list does not match column list.	• SQLExecDirect() • SQLExecute() • SQLParamData() • SQLPrepare()
21S02	Degrees of derived table does not match column list.	• SQLExecDirect() • SQLExecute() • SQLParamData() • SQLPrepare()
22001	String data right truncation.	• SQLPutData()
22002	Invalid output or indicator buffer specified.	• SQLExtendedFetch() • SQLFetch() • SQLGetData()
22003	Numeric value out of range.	• SQLExecDirect() • SQLExecute() • SQLParamData() • SQLExtendedFetch() • SQLFetch() • SQLGetData() • SQLPutData()
22005	Error in assignment.	• SQLExecDirect() • SQLExecute() • SQLParamData() • SQLExtendedFetch() • SQLFetch() • SQLGetData() • SQLPutData()
22008	Datetime field overflow.	• SQLExecDirect() • SQLExecute() • SQLParamData() • SQLExtendedFetch() • SQLFetch() • SQLGetData()
22012	Division by zero is invalid.	• SQLExecDirect() • SQLExecute() • SQLParamData() • SQLExtendedFetch() • SQLFetch()
23000	Integrity constraint violation.	• SQLExecDirect() • SQLExecute() • SQLParamData()

Table 143 (Page 3 of 10). SQLSTATE Cross Reference

SQLSTATE	Description	Functions
24000	Invalid cursor state.	• SQLColumns() • SQLColumnPrivileges() • SQLExecDirect() • SQLExecute() • SQLParamData() • SQLExtendedFetch() • SQLFetch() • SQLForeignKeys() • SQLGetData() • SQLGetStmtOption() • SQLGetTypeInfo() • SQLPrepare() • SQLPrimaryKeys() • SQLProcedures() • SQLProcedureColumns() • SQLSetColAttributes() • SQLSetStmtOption() • SQLSpecialColumns() • SQLStatistics() • SQLTables() • SQLTablePrivileges()
24504	The cursor identified in the UPDATE, DELETE, SET, or GET statement is not positioned on a row.	• SQLExecDirect() • SQLExecute()
25000	Invalid transaction state.	• SQLDisconnect()
25501	Invalid transaction state.	• SQLDisconnect()
28000	Invalid authorization specification.	• SQLConnect() • SQLDriverConnect()
34000	Invalid cursor name.	• SQLExecDirect() • SQLExecute() • SQLParamData() • SQLPrepare() • SQLSetCursorName()
37XXX	Invalid SQL syntax.	• SQLExecDirect() • SQLExecute() • SQLParamData() • SQLPrepare()
37000	Invalid SQL syntax.	• SQLNativeSql()
38552	Error in function listed in SYSFUN schema	All scalar functions, and SQLFetch() where the SQL statement references an ODBC scalar function.
40001	Transaction rollback.	• SQLExecDirect() • SQLExecute() • SQLPrepare()

Table 143 (Page 4 of 10). SQLSTATE Cross Reference

SQLSTATE	Description	Functions
40003	Communication link failure.	• SQLAllocStmt() • SQLBindCol() • SQLBindParameter() • SQLCancel() • SQLColumns() • SQLColumnPrivileges() • SQLColAttributes() • SQLDescribeCol() • SQLExecDirect() • SQLExecute() • SQLParamData() • SQLExtendedFetch() • SQLFetch() • SQLForeignKeys() • SQLFreeStmt() • SQLGetConnectOption() • SQLGetCursorName() • SQLGetData() • SQLGetFunctions() • SQLGetInfo() • SQLGetStmtOption() • SQLGetTypeInfo() • SQLMoreResults() • SQLNumParams() • SQLNumResultCols() • SQLParamData() • SQLParamOptions() • SQLPrepare() • SQLPrimaryKeys() • SQLProcedures() • SQLProcedureColumns() • SQLPutData() • SQLRowCount() • SQLSetColAttributes() • SQLSetConnectOption() • SQLSetCursorName() • SQLSetParam() • SQLSetStmtOption() • SQLSpecialColumns() • SQLStatistics() • SQLTables() • SQLTablePrivileges()
42XXX	Syntax error or access rule violation	• SQLExecDirect() • SQLExecute() • SQLParamData() • SQLPrepare()
425XX	Syntax error or access rule violation	• SQLExecDirect() • SQLExecute() • SQLParamData() • SQLPrepare()
42601	PARMLIST syntax error.	• SQLProcedureColumns()
42895	The value of a host variable in the EXECUTE or OPEN statement cannot be used because of its data type	• SQLExecDirect() • SQLExecute() • SQLParamData()
44000	Integrity constraint violation.	• SQLExecDirect() • SQLExecute() • SQLParamData()

Table 143 (Page 5 of 10). SQLSTATE Cross Reference

SQLSTATE	Description	Functions
56084	LOB data is not supported in DRDA.	• SQLExecDirect() • SQLExecute() • SQLParamData() • SQLExtendedFetch() • SQLFetch()
# **58**004	Unexpected system failure.	• SQLAllocConnect() • SQLAllocStmt() • SQLBindCol() • SQLBindParameter() • SQLColAttributes() • SQLConnect() • SQLDriverConnect() • SQLDataSources() • SQLDescribeCol() • SQLDisconnect() • SQLExecDirect() • SQLExecute() • SQLParamData() • SQLExtendedFetch() • SQLFetch() • SQLFreeConnect() • SQLFreeEnv() • SQLFreeStmt() • SQLGetCursorName() • SQLGetData() • SQLGetFunctions() • SQLGetInfo() • SQLMoreResults() • SQLNumResultCols() • SQLPrepare() • SQLRowCount() • SQLSetCursorName() • SQLSetParam() • SQLTransact()
S0001	Database object already exists.	• SQLExecDirect() • SQLExecute() • SQLParamData() • SQLPrepare()
S0002	Database object does not exist.	• SQLExecDirect() • SQLExecute() • SQLParamData() • SQLPrepare()
S0011	Index already exists.	• SQLExecDirect() • SQLExecute() • SQLParamData() • SQLPrepare()
S0012	Index not found.	• SQLExecDirect() • SQLExecute() • SQLParamData() • SQLPrepare()
S0021	Column already exists.	• SQLExecDirect() • SQLExecute() • SQLParamData() • SQLPrepare()
S0022	Column not found.	• SQLExecDirect() • SQLExecute() • SQLParamData() • SQLPrepare()

Table 143 (Page 6 of 10). SQLSTATE Cross Reference

SQLSTATE	Description	Functions
S1C00	Driver not capable.	• SQLBindCol() • SQLBindParameter() • SQLColumns() • SQLColumnPrivileges() • SQLColAttributes() • SQLDescribeCol() # • SQLDescribeParam() • SQLExtendedFetch() • SQLFetch() • SQLForeignKeys() • SQLGetConnectOption() • SQLGetData() • SQLGetInfo() • SQLGetStmtOption() • SQLPrimaryKeys() • SQLProcedures() • SQLProcedureColumns() • SQLSetConnectOption() • SQLSetEnvAttr() • SQLSetParam() • SQLSetStmtOption() • SQLSpecialColumns() • SQLStatistics() • SQLTables() • SQLTablePrivileges()
S1000	General error.	• SQLDataSources() # • SQLDescribeParam() • SQLDriverConnect() • SQLSetColAttributes() • SQLSetConnection() • SQLSetStmtOption()
S1001	Memory allocation failure.	All functions.
S1002	Invalid column number.	• SQLBindCol() • SQLColAttributes() • SQLDescribeCol() • SQLGetData() • SQLSetColAttributes()
S1003	Program type out of range.	• SQLBindCol() • SQLBindParameter() • SQLGetData() • SQLSetParam()
S1004	SQL data type out of range.	• SQLBindParameter() • SQLGetTypeInfo() • SQLSetColAttributes() • SQLSetParam()

Table 143 (Page 7 of 10). SQLSTATE Cross Reference

SQLSTATE	Description	Functions
S1009	Invalid argument value.	• SQLAllocConnect() • SQLAllocStmt() • SQLBindParameter() • SQLColumnPrivileges() • SQLConnect() • SQLDriverConnect() • SQLExecDirect() • SQLForeignKeys() • SQLGetConnectOption() • SQLGetData() • SQLGetFunctions() • SQLGetInfo() • SQLGetStmtOption() • SQLNativeSql() • SQLNumParams() • SQLNumResultCols() • SQLPrepare() • SQLPutData() • SQLSetConnectOption() • SQLSetCursorName() • SQLSetEnvAttr() • SQLSetParam() • SQLSetStmtOption()
S1010	Function sequence error.	• SQLBindCol() • SQLBindParameter() • SQLColumns() • SQLColAttributes() • SQLDescribeCol() # • SQLDescribeParam() • SQLDisconnect() • SQLExecute() • SQLExtendedFetch() • SQLFetch() • SQLForeignKeys() • SQLFreeConnect() • SQLFreeEnv() • SQLFreeStmt() • SQLGetCursorName() • SQLGetData() • SQLGetFunctions() • SQLGetStmtOption() • SQLGetTypeInfo() • SQLMoreResults() • SQLNumParams() • SQLNumResultCols() • SQLParamData() • SQLParamOptions() • SQLPrepare() • SQLPrimaryKeys() • SQLProcedures() • SQLProcedureColumns() • SQLPutData() • SQLRowCount() • SQLSetColAttributes() • SQLSetConnectOption() • SQLSetCursorName() • SQLSetParam() • SQLSetStmtOption() • SQLSpecialColumns() • SQLStatistics() • SQLTables() • SQLTablePrivileges()

Table 143 (Page 8 of 10). SQLSTATE Cross Reference

SQLSTATE	Description	Functions
S1011	Operation invalid at this time.	• SQLSetConnectOption() • SQLSetEnvAttr() • SQLSetStmtOption()
S1012	Invalid transaction code.	• SQLTransact()
S1013	Unexpected memory handling error.	• SQLAllocConnect() • SQLAllocStmt() • SQLBindCol() • SQLBindParameter() • SQLCancel() • SQLColAttributes() • SQLConnect() • SQLDriverConnect() • SQLDataSources() • SQLDescribeCol() • SQLDisconnect() • SQLExecDirect() • SQLExecute() • SQLParamData() • SQLExtendedFetch() • SQLFetch() • SQLFreeConnect() • SQLFreeEnv() • SQLGetCursorName() • SQLGetData() • SQLGetFunctions() • SQLMoreResults() • SQLNumParams() • SQLNumResultCols() • SQLPrepare() • SQLRowCount() • SQLSetColAttributes() • SQLSetCursorName() • SQLSetParam() • SQLTransact()
S1014	No more handles.	• SQLAllocConnect() • SQLAllocStmt() • SQLColumns() • SQLColumnPrivileges() • SQLExecDirect() • SQLForeignKeys() • SQLPrepare() • SQLPrimaryKeys() • SQLProcedures() • SQLProcedureColumns() • SQLSpecialColumns() • SQLStatistics() • SQLTables() • SQLTablePrivileges()

Table 143 (Page 9 of 10). SQLSTATE Cross Reference

SQLSTATE	Description	Functions
S1090	Invalid string or buffer length.	• SQLBindCol() • SQLBindParameter() • SQLColumns() • SQLColumnPrivileges() • SQLColAttributes() • SQLConnect() • SQLDriverConnect() • SQLDataSources() • SQLDescribeCol() • SQLDriverConnect() • SQLExecDirect() • SQLExecute() • SQLParamData() • SQLForeignKeys() • SQLGetCursorName() • SQLGetData() • SQLGetInfo() • SQLNativeSql() • SQLPrepare() • SQLPrimaryKeys() • SQLProcedures() • SQLProcedureColumns() • SQLPutData() • SQLSetColAttributes() • SQLSetCursorName() • SQLSpecialColumns() • SQLStatistics() • SQLTables() • SQLTablePrivileges()
S1091	Descriptor type out of range.	• SQLColAttributes()
S1092	Option type out of range.	• SQLFreeStmt() • SQLGetConnectOption() • SQLGetEnvAttr() • SQLExecDirect() • SQLExecute() • SQLExtendedFetch() • SQLFetch() • SQLGetStmtOption() • SQLParamData() • SQLSetConnectOption() • SQLSetEnvAttr() • SQLSetStmtOption()
S1093	Invalid parameter number.	• SQLBindParameter() • SQLDescribeParam() • SQLSetParam()
S1094	Invalid scale value.	• SQLBindParameter() • SQLSetColAttributes() • SQLSetParam()
S1096	Information type out of range.	• SQLGetInfo()
S1097	Column type out of range.	• SQLSpecialColumns()
S1098	Scope type out of range.	• SQLSpecialColumns()
S1099	Nullable type out of range.	• SQLSetColAttributes() • SQLSpecialColumns()
S1100	Uniqueness option type out of range.	• SQLStatistics()
S1101	Accuracy option type out of range.	• SQLStatistics()
S1103	Direction option out of range.	• SQLDataSources() • SQLGetInfo()

Table 143 (Page 10 of 10). SQLSTATE Cross Reference

SQLSTATE	Description	Functions
S1104	Invalid precision value.	• SQLBindParameter() • SQLSetColAttributes() • SQLSetParam()
S1105	Invalid parameter type.	• SQLBindParameter()
S1106	Fetch type out of range.	• SQLExtendedFetch()
S1107	Row value out of range.	• SQLParamOptions()
S1110	Invalid driver completion.	• SQLDriverConnect()
S1501	Invalid data source name.	• SQLConnect() • SQLDriverConnect()
S1503	Invalid file name length.	• SQLExecDirect() • SQLExecute() • SQLParamData()
S1506	Error closing a file.	• SQLCancel() • SQLFreeStmt() • SQLParamData()
S1509	Error deleting a file.	• SQLParamData()

Appendix E. Data Conversion

This section contains tables used for data conversion between C and SQL data types. This includes:

- Precision, scale, length, and display size of each data type
- Conversion from SQL to C data types
- Conversion from C to SQL data types

For a list of SQL and C data types, their symbolic types, and the default conversions, refer to Table 3 on page 40 and Table 4 on page 41. Supported conversions are shown in Table 7 on page 43.

Data Type Attributes

Information is shown for the following data type attributes:

- "Precision"
- "Scale" on page 421
- "Length" on page 422
- "Display Size" on page 423

Precision

The precision of a numeric column or parameter refers to the maximum number of digits used by the data type of the column or parameter. The precision of a non-numeric column or parameter generally refers to the maximum length or the defined length of the column or parameter. The following table defines the precision for each SQL data type.

Table 144. Precision

fSqlType	Precision
SQL_CHAR SQL_VARCHAR	The defined length of the column or parameter. For example, the precision of a column defined as CHAR(10) is 10.
SQL_LONGVARCHAR	The maximum length of the column or parameter. [a]
SQL_DECIMAL SQL_NUMERIC	The defined maximum number of digits. For example, the precision of a column defined as NUMERIC(10,3) is 10.
SQL_SMALLINT [b]	5
SQL_INTEGER [b]	10
SQL_FLOAT [b]	15
SQL_REAL [b]	7
SQL_DOUBLE [b]	15
SQL_BINARY SQL_VARBINARY	The defined length of the column or parameter. For example, the precision of a column defined as CHAR(10) FOR BIT DATA, is 10.
SQL_LONGVARBINARY	The maximum length of the column or parameter.
SQL_DATE [b]	10 (the number of characters in the yyyy-mm-dd format).
SQL_TIME [b]	8 (the number of characters in the hh:mm:ss format).
SQL_TIMESTAMP	The number of characters in the "yyyy-mm-dd hh:mm:ss[.fff[fff]]" or "yyyy-mm-dd.hh.mm.ss.fff[fff]]" format used by the TIMESTAMP data type. For example, if a timestamp does not use seconds or fractional seconds, the precision is 16 (the number of characters in the "yyyy-mm-dd hh:mm" format). If a timestamp uses thousandths of a second, the precision is 26 (the number of characters in the "yyyy-mm-dd hh:mm:ss.ffffff" format). The maximum for fractional seconds is 6 digits.
SQL_GRAPHIC SQL_VARGRAPHIC	The defined length of the column or parameter. For example, the precision of a column defined as GRAPHIC(10) is 10.
SQL_LONGVARGRAPHIC	The maximum length of the column or parameter.

Note:

[a] When defining the precision of a parameter of this data type with `SQLBindParameter()` or `SQLSetParam()`, *cbParamDef* should be set to the total length of the data, not the precision as defined in this table.

[b] The *cbParamDef* argument of `SQLBindParameter()` or `SQLSetParam()` is ignored for this data type.

Scale

The scale of a numeric column or parameter refers to the maximum number of digits to the right of the decimal point. Note that, for approximate floating point number columns or parameters, the scale is undefined, since the number of digits to the right of the decimal place is not fixed. The following table defines the scale for each SQL data type.

Table 145. Scale

fSqlType	**Scale**
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Not applicable.
SQL_DECIMAL SQL_NUMERIC	The defined number of digits to the right of the decimal place. For example, the scale of a column defined as NUMERIC(10, 3) is 3.
SQL_SMALLINT SQL_INTEGER	0
SQL_REAL SQL_FLOAT SQL_DOUBLE	Not applicable.
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY	Not applicable.
SQL_DATE SQL_TIME	Not applicable.
SQL_TIMESTAMP	The number of digits to the right of the decimal point in the "yyyy-mm-dd hh:mm:ss[fff[fff]]” format. For example, if the TIMESTAMP data type uses the "yyyy-mm-dd hh:mm:ss.fff” format, the scale is 3. The maximum for fractional seconds is 6 digits.
SQL_GRAPHIC SQL_VARGRAPHIC SQL_LONGVARGRAPHIC	Not applicable.

Length

The length of a column is the maximum number of *bytes* returned to the application when data is transferred to its default C data type. For character data, the length does not include the null termination byte. Note that the length of a column can be different than the number of bytes required to store the data on the data source. For a list of default C data types, see the "Default C Data Types" section.

The following table defines the length for each SQL data type.

Table 146. Length

fSqlType	**Length**
SQL_CHAR SQL_VARCHAR	The defined length of the column. For example, the length of a column defined as CHAR(10) is 10.
SQL_LONGVARCHAR	The maximum length of the column.
SQL_DECIMAL SQL_NUMERIC	The maximum number of digits plus two. Since these data types are returned as character strings, characters are needed for the digits, a sign, and a decimal point. For example, the length of a column defined as NUMERIC(10,3) is 12.
SQL_SMALLINT	2 (two bytes).
SQL_INTEGER	4 (four bytes).
SQL_REAL	4 (four bytes).
SQL_FLOAT	8 (eight bytes).
SQL_DOUBLE	8 (eight bytes).
SQL_BINARY SQL_VARBINARY	The defined length of the column. For example, the length of a column defined as CHAR(10) FOR BIT DATA is 10.
SQL_LONGVARBINARY	The maximum length of the column.
SQL_DATE SQL_TIME	6 (the size of the DATE_STRUCT or TIME_STRUCT structure).
SQL_TIMESTAMP	16 (the size of the TIMESTAMP_STRUCT structure).
SQL_GRAPHIC SQL_VARGRAPHIC	The defined length of the column times 2. For example, the length of a column defined as GRAPHIC(10) is 20.
SQL_LONGVARGRAPHIC	The maximum length of the column times 2.

Display Size

The display size of a column is the maximum number of *bytes* needed to display data in character form. The following table defines the display size for each SQL data type.

Table 147. Display Size

fSqlType	Display Size
SQL_CHAR SQL_VARCHAR	The defined length of the column. For example, the display size of a column defined as CHAR(10) is 10.
SQL_LONGVARCHAR	The maximum length of the column.
SQL_DECIMAL SQL_NUMERIC	The precision of the column plus two (a sign, precision digits, and a decimal point). For example, the display size of a column defined as NUMERIC(10,3) is 12.
SQL_SMALLINT	6 (a sign and 5 digits).
SQL_INTEGER	11 (a sign and 10 digits).
SQL_REAL	13 (a sign, 7 digits, a decimal point, the letter E, a sign, and 2 digits).
SQL_FLOAT SQL_DOUBLE	22 (a sign, 15 digits, a decimal point, the letter E, a sign, and 3 digits).
SQL_BINARY SQL_VARBINARY	The defined length of the column times 2 (each binary byte is represented by a 2 digit hexadecimal number). For example, the display size of a column defined as CHAR(10) FOR BIT DATA is 20.
SQL_LONGVARBINARY	The maximum length of the column times 2.
SQL_DATE	10 (a date in the format yyyy-mm-dd).
SQL_TIME	8 (a time in the format hh:mm:ss).
SQL_TIMESTAMP	19 (if the scale of the timestamp is 0) or 20 plus the scale of the timestamp (if the scale is greater than 0). This is the number of characters in the "yyyy-mm-dd hh:mm:ss[fff[fff]]” or "yyyy-mm-dd.hh.mm.ss.fff[fff]]” format. For example, the display size of a column storing thousandths of a second is 23 (the number of characters in "yyyy-mm-dd hh:mm:ss.ffffff”). The maximum for fractional seconds is 6 digits.
SQL_GRAPHIC SQL_VARGRAPHIC	The defined length of the column or parameter. For example, the display size of a column defined as GRAPHIC(10) is 20.
SQL_LONGVARGRAPHIC	The maximum length of the column or parameter.

Converting Data from SQL to C Data Types

For a given SQL data type:

- The first column of the table lists the legal input values of the *fCType* argument in `SQLBindCol()` and `SQLGetData()`.
- The second column lists the outcomes of a test, often using the *cbValueMax* argument specified in `SQLBindCol()` or `SQLGetData()`, which the driver performs to determine if it can convert the data.
- The third and fourth columns list the values (for each outcome) of the *rgbValue* and *pcbValue* arguments specified in the `SQLBindCol()` or `SQLGetData()` after the driver has attempted to convert the data.
- The last column lists the SQLSTATE returned for each outcome by `SQLFetch()`, `SQLExtendedFetch()`, or `SQLGetData()`.

The tables list the conversions defined by ODBC to be valid for a given SQL data type.

If the *fCType* argument in `SQLBindCol()` or `SQLGetData()` contains a value not shown in the table for a given SQL data type, `SQLFetch()`, or `SQLGetData()` returns the SQLSTATE 07006 (restricted data type attribute violation).

If the *fCType* argument contains a value shown in the table but which specifies a conversion not supported by the driver, `SQLFetch()`, or `SQLGetData()` returns SQLSTATE S1C00 (driver not capable).

Though it is not shown in the tables, the *pcbValue* argument contains SQL_NULL_DATA when the SQL data value is NULL. For an explanation of the use of *pcbValue* when multiple calls are made to retrieve data, see `SQLGetData()`.

When SQL data is converted to character C data, the character count returned in *pcbValue* does not include the null termination byte. If *rgbValue* is a null pointer, `SQLBindCol()` or `SQLGetData()` returns SQLSTATE S1009 (Invalid argument value).

In the following tables:

Length of data
The total length of the data after it has been converted to the specified C data type (excluding the null termination byte if the data was converted to a string). This is true even if data is truncated before it is returned to the application.

Significant digits
The minus sign (if needed) and the digits to the left of the decimal point.

Display size
The total number of bytes needed to display data in the character format.

proc=display. xproc=display.

Converting Character SQL Data to C Data

The character SQL data types are:

SQL_CHAR
SQL_VARCHAR
SQL_LONGVARCHAR

Table 148. Converting Character SQL Data to C Data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Length of data < cbValueMax	Data	Length of data	00000
	Length of data >= cbValueMax	Truncated data	Length of data	01004
SQL_C_BINARY	Length of data <= cbValueMax	Data	Length of data	00000
	Length of data > cbValueMax	Truncated data	Length of data	01004
SQL_C_SHORT SQL_C_LONG SQL_C_FLOAT SQL_C_DOUBLE SQL_C_TINYINT SQL_C_BIT	Data converted without truncation [a]	Data	Size of the C data type	00000
	Data converted with truncation, but without loss of significant digits [a]	Data	Size of the C data type	01004
	Conversion of data would result in loss of significant digits [a]	Untouched	Size of the C data type	22003
	Data is not a number [a]	Untouched	Size of the C data type	22005
SQL_C_DATE	Data value is a valid date [a]	Data	6 [b]	00000
	Data value is not a valid date [a]	Untouched	6 [b]	22008
SQL_C_TIME	Data value is a valid time [a]	Data	6 [b]	00000
	Data value is not a valid time [a]	Untouched	6 [b]	22008
SQL_C_TIMESTAMP	Data value is a valid timestamp [a]	Data	16 [b]	00000
	Data value is not a valid timestamp [a]	Untouched	16 [b]	22008

Note:

[a] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

[b] This is the size of the corresponding C data type.

SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

Converting Graphic SQL Data to C Data

The graphic SQL data types are:

SQL_GRAPHIC
SQL_VARGRAPHIC
SQL_LONGVARGRAPHIC

Table 149. Converting Graphic SQL Data to C Data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Number of double byte characters * 2 <= cbValueMax	Data	Length of data(octets)	00000
	Number of double byte characters * 2 <= cbValueMax	Truncated data, to the nearest even byte that is less than *cbValueMax*.	Length of data(octets)	01004
SQL_C_DBCHAR	Number of double byte characters * 2 < cbValueMax	Data	Length of data(octets)	00000
	Number of double byte characters * 2 >= cbValueMax	Truncated *cbValueMax*. data, to the nearest even byte that is less than *cbValueMax*.	Length of data(octets)	01004

Note:

SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

Converting Numeric SQL Data to C Data

The numeric SQL data types are:

SQL_DECIMAL
SQL_NUMERIC
SQL_SMALLINT
SQL_INTEGER
SQL_REAL
SQL_FLOAT
SQL_DOUBLE

Table 150. Converting Numeric SQL Data to C Data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Display size < cbValueMax	Data	Length of data	00000
	Number of significant digits < cbValueMax	Truncated data	Length of data	01004
	Number of significant digits >= cbValueMax	Untouched	Length of data	22003
SQL_C_SHORT SQL_C_LONG SQL_C_FLOAT SQL_C_DOUBLE SQL_C_TINYINT SQL_C_BIT	Data converted without truncation [a]	Data	Size of the C data type	00000
	Data converted with truncation, but without loss of significant digits [a]	Truncated data	Size of the C data type	01004
	Conversion of data would result in loss of significant digits [a]	Untouched	Size of the C data type	22003

Note:

[a] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

Converting Binary SQL Data to C Data

The binary SQL data types are:

SQL_BINARY
SQL_VARBINARY
SQL_LONGVARBINARY

Table 151. Converting Binary SQL Data to C Data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	(Length of data) < cbValueMax	Data	Length of data	N/A
	(Length of data) >= cbValueMax	Truncated data	Length of data	01004
SQL_C_BINARY	Length of data <= cbValueMax	Data	Length of data	N/A
	Length of data > cbValueMax	Truncated data	Length of data	01004

Converting Date SQL Data to C Data

The date SQL data type is:

SQL_DATE

Table 152. Converting Date SQL Data to C Data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	cbValueMax >= 11	Data	10	00000
	cbValueMax < 11	Untouched	10	22003
SQL_C_DATE	None [a]	Data	6 [b]	00000
SQL_C_TIMESTAMP	None [a]	Data [c]	16 [b]	00000
SQL_C_BINARY	Length of data <= cbValueMax	Data	Length of data	00000
	Length of data > cbValueMax	Untouched	Untouched	22003

Note:

[a] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

[b] This is the size of the corresponding C data type.

[c] The time fields of the TIMESTAMP_STRUCT structure are set to zero.

SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

When the date SQL data type is converted to the character C data type, the resulting string is in the "yyyy-mm-dd” format.

Converting Time SQL Data to C Data

The time SQL data type is:

SQL_TIME

Table 153. Converting Time SQL Data to C Data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	cbValueMax >= 9	Data	8	00000
	cbValueMax < 9	Untouched	8	22003
SQL_C_TIME	None [a]	Data	6 [b]	00000
SQL_C_TIMESTAMP	None [a]	Data [c]	16 [b]	00000

Note:

[a] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

[b] This is the size of the corresponding C data type.

[c] The date fields of the TIMESTAMP_STRUCT structure are set to the current system date of the machine that the application is running, and the time fraction is set to zero.

SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

When the time SQL data type is converted to the character C data type, the resulting string is in the "hh:mm:ss” format.

Converting Timestamp SQL Data to C Data

The timestamp SQL data type is:

SQL_TIMESTAMP

Table 154. Converting Timestamp SQL Data to C Data

fCType	Test	rgbValue	pcbValue	SQLSTATE
SQL_C_CHAR	Display size < cbValueMax	Data	Length of data	00000
	19 <= cbValueMax <= Display size	Truncated data [b]	Length of data	01004
	cbValueMax < 19	Untouched	Length of data	22003
SQL_C_DATE	None [a]	Truncated data [c]	6 [e]	01004
SQL_C_TIME	None [a]	Truncated data [d]	6 [e]	01004
SQL_C_TIMESTAMP	None [a]	Data	16 [e]	00000
	Fractional seconds portion of timestamp is truncated.[a]	Data [b]	16	01004

Note:

[a] The value of *cbValueMax* is ignored for this conversion. The driver assumes that the size of *rgbValue* is the size of the C data type.

[b] The fractional seconds of the timestamp are truncated.

[c] The time portion of the timestamp is deleted.

[d] The date portion of the timestamp is deleted.

[e] This is the size of the corresponding C data type.

SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

When the timestamp SQL data type is converted to the character C data type, the resulting string is in the "yyyy-mm-dd hh:mm:ss[.fff[fff]]” format (regardless of the precision of the timestamp SQL data type).

SQL to C Data Conversion Examples

Table 155. SQL to C Data Conversion Examples

SQL Data Type	SQL Data Value	C Data Type	cbValueMax	rgbValue	SQL STATE
SQL_CHAR	abcdef	SQL_C_CHAR	7	abcdef\0 [a]	00000
SQL_CHAR	abcdef	SQL_C_CHAR	6	abcde\0 [a]	01004
SQL_DECIMAL	1234.56	SQL_C_CHAR	8	1234.56\0 [a]	00000
SQL_DECIMAL	1234.56	SQL_C_CHAR	5	1234\0 [a]	01004
SQL_DECIMAL	1234.56	SQL_C_CHAR	4	---	22003
SQL_DECIMAL	1234.56	SQL_C_FLOAT	Ignored	1234.56	00000
SQL_DECIMAL	1234.56	SQL_C_SHORT	Ignored	1234	01004
SQL_DATE	1992-12-31	SQL_C_CHAR	11	1992-12-31\0 [a]	00000
SQL_DATE	1992-12-31	SQL_C_CHAR	10	---	22003
SQL_DATE	1992-12-31	SQL_C_TIMESTAMP	Ignored	1992,12,31, 0,0,0,0 [b]	00000
SQL_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	23	1992-12-31 23:45:55.12\0 [a]	00000
SQL_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	22	1992-12-31 23:45:55.1\0 [a]	01004
SQL_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	18	---	22003

Note:

[a] "\0" represents a null termination character.

[b] The numbers in this list are the numbers stored in the fields of the TIMESTAMP_STRUCT structure.

SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

Converting Data from C to SQL Data Types

For a given C data type:

- The first column of the table lists the legal input values of the *fSqlType* argument in `SQLBindParameter()` or `SQLSetParam()`.
- The second column lists the outcomes of a test, often using the length of the parameter data as specified in the *pcbValue* argument in `SQLBindParameter()` or `SQLSetParam()`, which the driver performs to determine if it can convert the data.
- The third column lists the SQLSTATE returned for each outcome by `SQLExecDirect()` or `SQLExecute()`.

  Note: Data is sent to the data source only if the SQLSTATE is 00000 (success).

The tables list the conversions defined by ODBC to be valid for a given SQL data type.

If the *fSqlType* argument in `SQLBindParameter()` or `SQLSetParam()` contains a value not shown in the table for a given C data type, SQLSTATE 07006 is returned (Restricted data type attribute violation).

If the *fSqlType* argument contains a value shown in the table but which specifies a conversion not supported by the driver, `SQLBindParameter()` or `SQLSetParam()` returns SQLSTATE S1C00 (Driver not capable).

If the *rgbValue* and *pcbValue* arguments specified in `SQLBindParameter()` or `SQLSetParam()` are both null pointers, that function returns SQLSTATE S1009 (Invalid argument value).

Length of data
The total length of the data after it has been converted to the specified SQL data type (excluding the null termination byte if the data was converted to a string). This is true even if data is truncated before it is sent to the data source.

Column length
The maximum number of bytes returned to the application when data is transferred to its default C data type. For character data, the length does not include the null termination byte.

Display size
The maximum number of bytes needed to display data in character form.

Significant digits
The minus sign (if needed) and the digits to the left of the decimal point.

Converting Character C Data to SQL Data

The character C data type is:

SQL_C_CHAR

Table 156. Converting Character C Data to SQL Data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Length of data <= Column length	00000
	Length of data > Column length	01004
SQL_DECIMAL SQL_NUMERIC SQL_SMALLINT SQL_INTEGER SQL_REAL SQL_FLOAT SQL_DOUBLE	Data converted without truncation	00000
	Data converted with truncation, but without loss of significant digits	01004
	Conversion of data would result in loss of significant digits	22003
	Data value is not a numeric value	22005
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY	(Length of data) < Column length	N/A
	(Length of data) >= Column length	01004
	Data value is not a hexadecimal value	22005
SQL_DATE	Data value is a valid date	00000
	Data value is not a valid date	22008
SQL_TIME	Data value is a valid time	00000
	Data value is not a valid time	22008
	Data value is a valid timestamp; time portion is non-zero	01004
SQL_TIMESTAMP	Data value is a valid timestamp	00000
	Data value is a valid timestamp; fractional seconds portion is non-zero	01004
	Data value is not a valid timestamp	22008
	Data value is a valid timestamp; fractional seconds portion is non-zero	01004
SQL_GRAPHIC SQL_VARGRAPHIC SQL_LONGVARGRAPHIC	Length of data / 2 <= Column length	00000
	Length of data / 2 < Column length	01004

Note:

SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

Converting Numeric C Data to SQL Data

The numeric C data types are:

SQL_C_SHORT
SQL_C_LONG
SQL_C_FLOAT
SQL_C_DOUBLE
SQL_C_TINYINT
SQL_C_BIT

Table 157. Converting Numeric C Data to SQL Data

fSQLType	Test	SQLSTATE
SQL_DECIMAL SQL_NUMERIC SQL_SMALLINT SQL_INTEGER SQL_REAL SQL_FLOAT SQL_DOUBLE	Data converted without truncation	00000
	Data converted with truncation, but without loss of significant digits	01004
	Conversion of data would result in loss of significant digits	22003
SQL_CHAR SQL_VARCHAR	Data converted without truncation.	00000
	Conversion of data would result in loss of significant digits.	22003

Note:

SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

Converting Binary C Data to SQL Data

The binary C data type is:

SQL_C_BINARY

Table 158. Converting Binary C Data to SQL Data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Length of data <= Column length	N/A
	Length of data > Column length	01004
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY	Length of data <= Column length	N/A
	Length of data > Column length	01004

Converting DBCHAR C Data to SQL Data

The double byte C data type is:

SQL_C_DBCHAR

Table 159. Converting DBCHAR C Data to SQL Data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Length of data <= Column length x 2	N/A
	Length of data > Column length x 2	01004
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY	Length of data <= Column length x 2	N/A
	Length of data > Column length x 2	01004

Converting Date C Data to SQL Data

The date C data type is:

SQL_C_DATE

Table 160. Converting Date C Data to SQL Data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR	Column length >= 10	00000
	Column length < 10	22003
SQL_DATE	Data value is a valid date	00000
	Data value is not a valid date	22008
SQL_TIMESTAMP [a]	Data value is a valid date	00000
	Data value is not a valid date	22008

Note:

[a] The time component of TIMESTAMP is set to zero.

SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

Converting Time C Data to SQL Data

The time C data type is:

SQL_C_TIME

Table 161. Converting Time C Data to SQL Data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR	Column length >= 8	00000
	Column length < 8	22003
SQL_TIME	Data value is a valid time	00000
	Data value is not a valid time	22008
SQL_TIMESTAMP [a]	Data value is a valid time	00000
	Data value is not a valid time	22008

Note:

[a] The date component of TIMESTAMP is set to the system date of the machine at which the application is running.

SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

Converting Timestamp C Data to SQL Data

The timestamp C data type is:

SQL_C_TIMESTAMP

Table 162. Converting Timestamp C Data to SQL Data

fSQLType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR	Column length >= Display size	00000
	19 <= Column length < Display size [a]	01004
	Column length < 19	22003
SQL_DATE	Data value is a valid date [b]	01004
	Data value is not a valid date	22008
SQL_TIME	Data value is a valid time [c]	01004
	Data value is not a valid time	22008
	Fractional seconds fields are non-zero	01004
SQL_TIMESTAMP	Data value is a valid timestamp	00000
	Data value is not a valid timestamp	22008

Note:

[a] The fractional seconds of the timestamp are truncated.
[b] The time portion of the timestamp is deleted.
[c] The date portion of the timestamp is deleted.

SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

C to SQL Data Conversion Examples

Table 163. C to SQL Data Conversion Examples

C Data Type	C Data Value	SQL Data Type	Column Length	SQL Data Value	SQLSTATE
SQL_C_CHAR	abcdef\0	SQL_CHAR	6	abcdef	00000
SQL_C_CHAR	abcdef\0	SQL_CHAR	5	abcde	01004
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	6	1234.56	00000
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	5	1234.5	01004
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	3	---	22003
SQL_C_FLOAT	1234.56	SQL_FLOAT	Not applicable	1234.56	00000
SQL_C_FLOAT	1234.56	SQL_INTEGER	Not applicable	1234	01004

Note:

SQLSTATE **00**000 is not returned by `SQLError()`, rather it is indicated when the function returns SQL_SUCCESS.

Appendix F. Example Code

This section provides two extensive examples.

- “DB2 CLI Application” shows an application that creates two tables, populates them, and reads them.
- “Stored Procedure” on page 472 shows a DB2 CLI client application calling a DB2 CLI stored procedure.

DB2 CLI Application

The application in this example consists of three parts.

1. In the first step, the application creates two tables: the COMMANDERS table and the BATTLES table.
2. In the second step, the application populates these tables with long data in pieces using the `SQLPutData()` API.
3. In the last step, the application retrieves data from the tables using the `SQLGetData()` API.

STEP 1. Create COMMANDERS and BATTLES tables.

```
 /******************************************************************/
 /* DB2 for OS/390 CLI Sample:                                     */
 /*    Exercises the ODBC Version 2.0 APIs necessary to create     */
 /*    tables. The tables are formulated using ODBC SQL data types */
 /*    and text is generated to create a COMMANDERS table and a    */
 /*    BATTLES table.                                              */
 /******************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include <sqlcli1.h>

typedef struct Column
{
  UCHAR            Colname [16];        // columns name
  SWORD            SQLType;             // ODBC SQL type
  SWORD            Precision;           // length
  SWORD            Nullable;            // nullable (true) or non-nullabe (false
} COLUMN, *pCOLUMN;

typedef struct DSNtype
{
  UCHAR            Typename [40];       // DSN type name
  SWORD            SQLType;             // DSN SQL type
  SWORD            Precision;           // length
  SWORD            Nullable;            // nullable (true) or non-nullabe (false
  struct DSNtype  *next;                // forward link
} DSNTYPE, *pDSNTYPE;
```

```
typedef struct Table
{
  UCHAR            TableName [16];      // table name
  SWORD            NumColumns;          // number of columns
  pCOLUMN          TableCols;           // address of COLUMN array
} TABLE, *pTABLE;
 /******************************************************************/
 /* Define BATTLES Table                                           */
 /******************************************************************/
COLUMN BATTLES [5] =    {
                         { "Battlename", SQL_CHAR       , 20,    0},
                         { "Date"      , SQL_DATE       ,  0,    0},
                         { "Winner"    , SQL_CHAR       , 20,    0},
                         { "Loser"     , SQL_CHAR       , 20,    0},
                         { "Narrative" , SQL_LONGVARCHAR,  0,    0}
                        } ;
 /******************************************************************/
 /* Define COMMANDERS Table                                        */
 /******************************************************************/
COLUMN COMMANDERS [4] = {
                         { "CINC"      , SQL_CHAR       , 20,    1},
                         { "Branch"    , SQL_CHAR       , 30,    1},
                         { "Rank"      , SQL_CHAR       , 20,    1},
                         { "Cur_Vitae" , SQL_LONGVARCHAR,  0,    0}
                        } ;

TABLE TBDEF [2]       = {
                         { "Commanders", 4, COMMANDERS },
                         { "Battles"   , 5, BATTLES    }
                        } ;
 /******************************************************************/
 /* CLI APIs required                                              */
 /******************************************************************/
SQLUSMALLINT ODBC_api [7] = {
                             SQL_API_SQLBINDPARAMETER,
                             SQL_API_SQLDISCONNECT   ,
                             SQL_API_SQLGETTYPEINFO  ,
                             SQL_API_SQLFETCH        ,
                             SQL_API_SQLTRANSACT     ,
                             SQL_API_SQLBINDCOL      ,
                             SQL_API_SQLEXECDIRECT
                            } ;
  /******************************************************************/
  /* GetDSNTypes and CreateTable prototypes                         */
  /******************************************************************/
int GetDSNTypes (SQLHDBC   hdbc,
                 pDSNTYPE *dsntype);

int CreateTable (SQLHDBC   hdbc,
                 pDSNTYPE  dsntype,
                 pTABLE    pTab);
  /******************************************************************/
  /* CLI T3: Create Table main entry point                          */
  /******************************************************************/
int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLRETURN       rc      = SQL_SUCCESS;
   SQLINTEGER      RETCODE = SQL_SUCCESS;
   SQLSMALLINT     t3_small= 0;
   int             i;
   pDSNTYPE        dsntype_list = NULL;
   pDSNTYPE        dsnp;
   pTABLE          pTab;

   // SQLDataSources parameters

   SQLCHAR         szDSN [19] = {0};
   SQLSMALLINT     fDirection = SQL_FETCH_FIRST;
   SQLSMALLINT     cbDSNMax   = sizeof(szDSN);
   SQLSMALLINT     cbDSN      = 0;
   SQLSMALLINT     *pcbDSN    = &cbDSN;
   SQLCHAR         szDescription [40];
   SQLSMALLINT     cbDescriptionMax = sizeof(szDescription);
   SQLSMALLINT     *pcbDescription = &t3_small;

   // SQLGetFunctions parameters
   SQLUSMALLINT     fExists  = SQL_TRUE;
   SQLUSMALLINT    *pfExists = &fExists;

   (void) printf ("**** DB2 for OS/390 CLI: Create Table.\n\n");
  /*****************************************************************/
  /* Allocate Environment Handle                                   */
  /*****************************************************************/
   RETCODE = SQLAllocEnv(&hEnv);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;
  /*****************************************************************/
  /* See if DSN= STLEC1 is known                                   */
  /*****************************************************************/
   RETCODE = SQLDataSources (hEnv,
                             fDirection,
                             szDSN,
                             cbDSNMax,
                             pcbDSN,
                             szDescription,
                             cbDescriptionMax,
                             pcbDescription); // fetch first DSN

   while ((RETCODE                            == SQL_SUCCESS) &&
          (memcmp(szDSN, "STLEC1", (*pcbDSN)) != 0))
   {
     RETCODE = SQLDataSources (hEnv,
                               SQL_FETCH_NEXT,
                               szDSN,
                               cbDSNMax,
                               pcbDSN,
                               szDescription,
                               cbDescriptionMax,
                               pcbDescription); // fetch next DSN
   }
```

```
     if (RETCODE != SQL_SUCCESS)
     {
       (void) printf ("**** DSN = STLEC1 not known.\n");
       goto dberror;
     }
     else
       (void) printf ("**** Found DSN = STLEC1.\n");
    /*****************************************************************/
    /* Allocate Connection Handle to DSN                             */
    /*****************************************************************/
     RETCODE = SQLAllocConnect(hEnv,
                               &hDbc);

     if( RETCODE != SQL_SUCCESS )      // Could not get a Connect Handle
       goto dberror;
    /*****************************************************************/
    /* CONNECT TO data source (STLEC1)                               */
    /*****************************************************************/
     RETCODE = SQLConnect(hDbc,        // Connect handle
                          szDSN,       // DSN
                          (*pcbDSN),   // length of DSN
                          NULL,        // Null UID
                          0   ,

#  NULL,        //
#  Null Auth string
                          0);

     if( RETCODE != SQL_SUCCESS )      // Connect failed
       goto dberror;
     else
       (void) printf ("**** Connect to %s OK.\n", (char *) szDSN);
    /*****************************************************************/
    /* See if DSN supports required ODBC APIs                        */
    /*****************************************************************/
     for (i = 0, (*pfExists = SQL_TRUE);
          (i < 7 && (*pfExists) == SQL_TRUE);
          i++)
     {
       RETCODE = SQLGetFunctions (hDbc,
                                  ODBC_api[i],
                                  pfExists);
     }

     if (*pfExists == SQL_FALSE)      // a required API is not supported
       goto dberror;
    /*****************************************************************/
    /* Retrieve SQL data types from DSN                              */
    /*****************************************************************/
     RETCODE = GetDSNTypes (hDbc,
                            &dsntype_list);

     if (RETCODE != SQL_SUCCESS)       // An advertised API failed
       goto dberror;
```

```
  /*****************************************************************/
  /* Create COMMANDERS and BATTLES Tables                          */
  /*****************************************************************/
   for (i = 0, pTab = TBDEF;
        (i < 2 && RETCODE == SQL_SUCCESS);
        i++, pTab++)
   {
     RETCODE = CreateTable (hDbc,
                            dsntype_list,
                            pTab);
   }
  /*****************************************************************/
  /* DISCONNECT from data source                                   */
  /*****************************************************************/
   RETCODE = SQLDisconnect(hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;
  /*****************************************************************/
  /* Deallocate Connection Handle                                  */
  /*****************************************************************/
   RETCODE = SQLFreeConnect (hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;
  /*****************************************************************/
  /* Free Environment Handle                                       */
  /*****************************************************************/
   RETCODE = SQLFreeEnv (hEnv);

   if (RETCODE == SQL_SUCCESS)
     goto exit;

   dberror:
   RETCODE=12;

   exit:
  /*****************************************************************/
  /* Deallocate DSN Type List                                      */
  /*****************************************************************/
   for (dsnp = dsntype_list; dsnp != NULL; dsnp = dsntype_list)
   {
     dsntype_list = dsnp->next;
     (void) free ( (void *) dsnp);
   }

   (void) printf("\n\nDB2 for OS/390 CLI: Create Table TERMINATION\n\n   ");

   if (RETCODE!=0)
     (void) printf("\n\nDB2 for OS/390 CLI: WAS UNSUCCESSFUL\n");
   else
     (void) printf("\n\nDB2 for OS/390 CLI: WAS SUCCESSFUL\n\n");

   return(RETCODE);
}
```

```
/******************************************************************/
/* Function GetDSNTypes creates a linked list of all SQL data     */
/* types supported by the DSN.                                    */
/******************************************************************/
int GetDSNTypes (SQLHDBC   hDbc,
                 pDSNTYPE *pdsntype)
{
  SQLINTEGER       rc = SQL_SUCCESS;
  SQLHSTMT         hStmt = SQL_NULL_HSTMT;
  pDSNTYPE         pdsn  = NULL;
  int              count= 1;
  // local variables to Bind to retrieve TYPE_NAME, DATA_TYPE,
  // COLUMN_SIZE and NULLABLE

  struct                    // TYPE_NAME is VARCHAR(128)
  {
    SQLSMALLINT  length;
    SQLCHAR      name [128];
    SQLINTEGER   ind;
  } typename;

  SQLSMALLINT data_type;    // DATA_TYPE is SMALLINT
  SQLINTEGER  data_type_ind;
  SQLINTEGER  column_size;  // COLUMN_SIZE is integer
  SQLINTEGER  column_size_ind;
  SQLSMALLINT nullable;     // NULLABLE is SMALLINT
  SQLINTEGER  nullable_ind;
  /******************************************************************/
  /* Allocate Statement Handle                                      */
  /******************************************************************/
  rc = SQLAllocStmt (hDbc,
                     &hStmt);

  if (rc != SQL_SUCCESS)
    goto exit;
  /******************************************************************/
  /* Retrieve native SQL types from DSN ------------>               */
  /*  The result set consists of 15 columns. We will only bind      */
  /*  TYPE_NAME, DATA_TYPE, COLUMN_SIZE and NULLABLE. Note: Need    */
  /*  not bind all columns of result set -- only those required.    */
  /******************************************************************/
  rc = SQLGetTypeInfo (hStmt,
                       SQL_ALL_TYPES);

  if (rc != SQL_SUCCESS)
    goto exit;

  rc = SQLBindCol (hStmt,           // bind TYPE_NAME
                   1,
                   SQL_CHAR,
                   (SQLPOINTER) typename.name,
                   128,
                   &typename.ind);

  if (rc != SQL_SUCCESS)
    goto exit;
  rc = SQLBindCol (hStmt,           // bind DATA_NAME
                   2,
                   SQL_C_DEFAULT,
                   (SQLPOINTER) &data_type,
                   sizeof(data_type),
                   &data_type_ind);

  if (rc != SQL_SUCCESS)
    goto exit;

  rc = SQLBindCol (hStmt,           // bind COLUMN_SIZE
                   3,
                   SQL_C_DEFAULT,
                   (SQLPOINTER) &column_size,
                   sizeof(column_size),
                   &column_size_ind);

  if (rc != SQL_SUCCESS)
    goto exit;

  rc = SQLBindCol (hStmt,           // bind NULLABLE
                   7,
                   SQL_C_DEFAULT,
                   (SQLPOINTER) &nullable,
                   sizeof(nullable),
                   &nullable_ind);

  if (rc != SQL_SUCCESS)
    goto exit;
  /******************************************************************/
  /* Fetch all native DSN types and allocate a DSNTYPE structure   */
  /* and chain onto linked list passed via pdsntype                */
  /******************************************************************/
  while ((rc = SQLFetch (hStmt)) == SQL_SUCCESS)
  {
    pdsn = (pDSNTYPE) malloc (sizeof(DSNTYPE));
    strcpy ((char *) pdsn->Typename, (char *) typename.name);
    pdsn->SQLType                  = data_type;
    pdsn->Precision                = column_size;
    pdsn->Nullable                 = nullable;
    pdsn->next                     = (*pdsntype);
    (*pdsntype)                    = pdsn;
  }

  if (rc == SQL_NO_DATA_FOUND)   // if result set exhausted reset
    rc = SQL_SUCCESS;            // rc to OK
  /******************************************************************/
  /* Free Statement handle.                                         */
  /******************************************************************/
  rc = SQLFreeStmt (hStmt,
                    SQL_DROP);
  exit:

  return (rc);
}
```

```
/*****************************************************************/
/* Function CreateTable processes the TABLE passed, generating   */
/* a CREATE TABLE statement. It then executes the statement.     */
/*****************************************************************/
int CreateTable (SQLHDBC  hDbc,
                 pDSNTYPE pdsnt,
                 pTABLE   pTab)
{
  SQLINTEGER rc = SQL_SUCCESS;
  SQLHSTMT   hStmt = SQL_NULL_HSTMT;
  char       pTable[200];              // buffer for CREATE TABLE
  char       precision [10];
  pCOLUMN    pCol;                     // Column Pointer
  int        i;
  char       temp [10];
  /*****************************************************************/
  /* Allocate Statement Handle                                     */
  /*****************************************************************/
  rc = SQLAllocStmt (hDbc,
                     &hStmt);

  if (rc != SQL_SUCCESS)
    goto exit;
  /*****************************************************************/
  /* First DROP Table                                              */
  /*****************************************************************/
  (void) strcpy (pTable, "DROP TABLE ");
  (void) strcat (pTable, (char *) pTab->TableName);

  rc = SQLExecDirect (hDbc,
                      (SQLCHAR *) pTable,
                      strlen(pTable));
  /*****************************************************************/
  /* Build CREATE TABLE text                                       */
  /*****************************************************************/
  (void) strcpy (pTable, "CREATE TABLE ");
  (void) strcat (pTable, (char *) pTab->TableName);
  (void) strcat (pTable, " ( ");
  /*****************************************************************/
  /* Iterate through all ODBC column types and find matching       */
  /* SQL type on native SQL type linked list. If not found cannot  */
  /* create table.                                                 */
  /*****************************************************************/
  for (i = 0, pCol = pTab->TableCols;
       (i < pTab->NumColumns);
       i++, pCol++)
  {
    pDSNTYPE dsnp = pdsnt;           // point to start of list

    while ((dsnp != NULL) &&
           (dsnp->SQLType != pCol->SQLType))
      dsnp = dsnp->next;

    if (dsnp == NULL)                // SQL type not supported
    {
      rc = SQL_ERROR;                // cannot build CREATE TABLE
      goto exit;
    }
    else
    {
      strcat (pTable, (char *) pCol->Colname);  // append Column name
      strcat (pTable, " ");
      strcat (pTable, (char *)dsnp->Typename);  // append DSN Type Name
      if (pCol->Precision != 0)          // if length required
      {
        strcat (pTable, " (");           // convert to CHAR and
        sprintf (precision, "%d", pCol->Precision); // append
        strcat (pTable, precision);
        strcat (pTable, ")");
      }

      if (!pCol->Nullable)               // if column is NOT NULLABLE
        strcat (pTable, " NOT NULL ");
      if (i == (pTab->NumColumns - 1))   // if last column
        strcat (pTable, " )");           // close parens
      else
        strcat (pTable, " , ");          // append comma delimiter
    }
  }
  /*****************************************************************/
  /* CREATE TABLE text is constructed. Execute statement          */
  /*****************************************************************/
  (void) printf ("**** DB2 for OS/390 CLI: Table text = %s.\n",
                 pTable);

  rc = SQLExecDirect (hDbc,
                      (SQLCHAR *) pTable,
                      strlen(pTable));

  if (rc != SQL_SUCCESS)
    goto exit;
  /*****************************************************************/
  /* Deallocate Statement Handle                                   */
  /*****************************************************************/
  rc = SQLFreeStmt (hDbc,
                    SQL_DROP);
  exit:

  return (rc);
}
```

STEP 2. Populate tables.

```
 /******************************************************************/
 /* DB2 for OS/390 CLI Sample:                                     */
 /*    Exercises the ODBC Version 2.0 APIs necessary to populate   */
 /*    the COMMANDERS and BATTLES tables. The biographic text of   */
 /*    the COMMANDERS table and the Narrative text of the BATTLES  */
 /*    table is inserted in 80-character pieces.                   */
 /*                                                                */
 /*    The source for the tables are two flat files read via the   */
 /*    ANSI C file APIs.                                           */
 /******************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlcli1.h>
 /******************************************************************/
 /* Structure to represent one line of text                        */
 /******************************************************************/
typedef struct Text
{
  char             text_line [80];      // line of text
  struct Text     *next;                // forward link
} TEXT, *pTEXT;
 /******************************************************************/
 /* Structure to represent one Commander                           */
 /******************************************************************/
typedef struct Commander
{
  char             Cname  [20];         // Commander Name
  char             Branch [30];         // Branch of service
  char             Rank   [20];         // Rank in service
  pTEXT            CVitae;              // Curriculm Vitae text chain
  SQLINTEGER       Cname_ind;           // Cname length
  SQLINTEGER       Branch_ind;          // Branch length
  SQLINTEGER       Rank_ind;            // Rank length
  SQLINTEGER       CVitae_ind;          // CVitae indicator
  struct Commander *next;               // forward link
} COMMANDER, *pCOMMANDER;
 /******************************************************************/
 /* Structure to represent one Battle                              */
 /******************************************************************/
typedef struct Battle
{
  char             Bname     [20];      // Battle Name
  char             Date      [30];      // Date
  char             Winner    [20];      // Winner
  char             Loser     [20];      // Loser
  pTEXT            Narrative;           // Curriculm Vitae text chain
  SQLINTEGER       Bname_ind;           // Bname length
  SQLINTEGER       Date_ind;            // Date length
  SQLINTEGER       Winner_ind;          // Winner length
  SQLINTEGER       Loser_ind;           // Loser  length
  SQLINTEGER       Narrative_ind;       // Narrative indicator
  struct Battle   *next;                // forward link
} BATTLE, *pBATTLE;
```

```
 /******************************************************************/
 /* CLI APIs required                                              */
 /******************************************************************/
SQLUSMALLINT ODBC_api [7] = {
                             SQL_API_SQLBINDPARAMETER,
                             SQL_API_SQLDISCONNECT   ,
                             SQL_API_SQLFETCH        ,
                             SQL_API_SQLTRANSACT     ,
                             SQL_API_SQLBINDCOL      ,
                             SQL_API_SQLEXECDIRECT   ,
                             SQL_API_SQLPUTDATA
                            } ;
  /******************************************************************/
  /* GetCommanders, GetBattles , PopCommanders and PopBattles       */
  /* Function prototypes                                            */
  /******************************************************************/
int GetCommanders (pCOMMANDER *pCmd,
                   FILE       *fp);

int GetBattles (pBATTLE *pBat,
                FILE    *fp);

int PopCommanders (SQLHDBC    hdbc,
                   pCOMMANDER pCmd);

int PopBattles    (SQLHDBC  hdbc,
                   pBATTLE  pBat);

int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLRETURN       rc      = SQL_SUCCESS;
   SQLSMALLINT     t3_small= 0;
   SQLINTEGER      RETCODE = 0;
   int             i;
   pCOMMANDER      pCmdTop = NULL;    // linked list of Commander structures
   pCOMMANDER      pCmd    = NULL;
   pBATTLE         pBatTop = NULL;    // linked list of Battles structures
   pBATTLE         pBat    = NULL;
   pTEXT           ptxt    = NULL;
   FILE           *fp = NULL;

   // SQLDataSources parameters

   SQLCHAR         szDSN [19]• = {0};
   SQLSMALLINT     fDirection = SQL_FETCH_FIRST;
   SQLSMALLINT     cbDSNMax   = sizeof(szDSN);
   SQLSMALLINT     cbDSN      = 0;
   SQLSMALLINT     *pcbDSN    = &cbDSN;
   SQLCHAR         szDescription [40];
   SQLSMALLINT     cbDescriptionMax = sizeof(szDescription);
   SQLSMALLINT     *pcbDescription = &t3_small;

   // SQLGetFunctions parameters

   SQLUSMALLINT     fExists  = SQL_TRUE;
   SQLUSMALLINT    *pfExists = &fExists;


   (void) printf ("**** DB2 for OS/390 CLI: Table Population.\n\n");
```

```
/*****************************************************************/
/* Allocate Environment Handle                                   */
/*****************************************************************/
 RETCODE = SQLAllocEnv(&hEnv);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;
/*****************************************************************/
/* See if STLEC1 is a known DSN                                  */
/*****************************************************************/
 RETCODE = SQLDataSources (hEnv,
                           fDirection,
                           szDSN,
                           cbDSNMax,
                           pcbDSN,
                           szDescription,
                           cbDescriptionMax,
                           pcbDescription); // fetch first DSN

 while ((RETCODE                            == SQL_SUCCESS) &&
        (memcmp(szDSN, "STLEC1", (*pcbDSN)) != 0))
 {
   RETCODE = SQLDataSources (hEnv,
                             SQL_FETCH_NEXT,
                             szDSN,
                             cbDSNMax,
                             pcbDSN,
                             szDescription,
                             cbDescriptionMax,
                             pcbDescription); // fetch next DSN
 }

 if (RETCODE != SQL_SUCCESS)
 {
   (void) printf ("**** DSN = STLEC1 not known.\n");
   goto dberror;
 }
/*****************************************************************/
/* Allocate Connection Handle to DSN                             */
/*****************************************************************/
 RETCODE = SQLAllocConnect(hEnv,
                           &hDbc);

 if( RETCODE != SQL_SUCCESS )      // Could not get a Connect Handle
   goto dberror;
/*****************************************************************/
/* CONNECT TO data source (STLEC1)                               */
/*****************************************************************/
 RETCODE = SQLConnect(hDbc,        // Connect handle
                      szDSN,       // DSN
                      (*pcbDSN),   // length of DSN
                      NULL,        // Null UID
                      0   ,
                      NULL,        // Null Auth string
                      0);

 if( RETCODE != SQL_SUCCESS )      // Connect failed
   goto dberror;
```

```
/*****************************************************************/
/* See if DSN supports required ODBC APIs                        */
/*****************************************************************/
 for (i = 0, (*pfExists = SQL_TRUE);
      (i < 7 && (*pfExists) == SQL_TRUE);
      i++)
 {
   RETCODE = SQLGetFunctions (hDbc,
                              ODBC_api[i],
                              pfExists);
 }

 if (*pfExists == SQL_FALSE)      // a required API is not supported
   goto dberror;
/*****************************************************************/
/* Read Commanders File and build Commander List                 */
/*****************************************************************/
 RETCODE = GetCommanders (&pCmdTop,
                          fp);

 if (RETCODE != SQL_SUCCESS)       // List Creation failed
   goto dberror;

/*****************************************************************/
/* Populate Commanders Table                                     */
/*****************************************************************/
 RETCODE = PopCommanders (hDbc,
                          pCmdTop);

 if (RETCODE != SQL_SUCCESS)       // Population failed
   goto dberror;
/*****************************************************************/
/* Read Battles File and build Battles List                      */
/*****************************************************************/
 RETCODE = GetBattles (&pBatTop,
                       fp);

 if (RETCODE != SQL_SUCCESS)       // List Creation failed
   goto dberror;
/*****************************************************************/
/* Populate Battles Table                                        */
/*****************************************************************/
 RETCODE = PopBattles (hDbc,
                       pBatTop);

 if (RETCODE != SQL_SUCCESS)       // Population failed
   goto dberror;
/*****************************************************************/
/* DISCONNECT from data source                                   */
/*****************************************************************/
 RETCODE = SQLDisconnect(hDbc);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;
/*****************************************************************/
/* Deallocate Connection Handle                                  */
/*****************************************************************/
 RETCODE = SQLFreeConnect (hDbc);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;
```

```
  /******************************************************************/
  /* Free Environment Handle                                        */
  /******************************************************************/
   RETCODE = SQLFreeEnv (hEnv);

   if (RETCODE == SQL_SUCCESS)
     goto exit;

   dberror:
   RETCODE=12;

   exit:
  /******************************************************************/
  /* Deallocate Commander and Battle List and associated text lines*/
  /******************************************************************/
   for (pCmd = pCmdTop; pCmd != NULL; pCmd = pCmdTop)
   {
     for (ptxt = pCmd->CVitae; ptxt != NULL; ptxt = pCmd->CVitae)
     {
       pCmd->CVitae = ptxt->next;
       (void) free ( (void *) ptxt);
     }

     pCmdTop = pCmdTop->next;
     (void) free ( (void *) pCmd);
   }

   for (pBat = pBatTop; pBat != NULL; pBat = pBatTop)
   {
     for (ptxt = pBat->Narrative; ptxt != NULL; ptxt = pBat->Narrative)
     {
       pBat->Narrative = ptxt->next;
       (void) free ( (void *) ptxt);
     }

     pBatTop = pBatTop->next;
     (void) free ( (void *) pBat);
   }

   (void) printf("\n\nDB2 for OS/390 CLI: Table Population TERMINATION\n\n   ");

   if (RETCODE!=0)
     (void) printf("\n\nDB2 for OS/390 CLI: Table Population WAS UNSUCCESSFUL\n"
   else
     (void) printf("\n\nDB2 for OS/390 CLI: Table Population WAS SUCCESSFUL\n\n"

   return(RETCODE);
}
```

```
/*****************************************************************/
/* Function GetCommanders reads the "command" file and creates a */
/* linked list of COMMANDER structures which is returned to the  */
/* caller.                                                       */
/*****************************************************************/
int GetCommanders (pCOMMANDER *pCmd,
                   FILE       *fp)
{
  SQLINTEGER       rc = SQL_SUCCESS;
  pCOMMANDER       pCmd2 = NULL;
  char             text [80];        // file text
  char             *t = NULL;
  char             *c = NULL;
  char             *token = NULL;
  char             temp [80];
  pTEXT            ptxt, ptxt2;
  int              i;

  if ((fp = fopen ("DD:COMMAND", "r")) == NULL)  // open command file
  {
    rc = SQL_ERROR;                  // open failed
    goto exit;
  }
/*****************************************************************/
/* Process file and create COMMANDER structures                  */
/*****************************************************************/
  while ((t = fgets (text, sizeof(text), fp)) != NULL)
  {
    if (text[0] == '#')               // if commander data
    {
       if (pCmd2 != NULL)            // if Commander structure allocated
       {
         pCmd2->next = (*pCmd);      // then insert LIFO
         (*pCmd)     = pCmd2;
       }

       pCmd2 = (pCOMMANDER) malloc (sizeof(COMMANDER));
       pCmd2->next = NULL;
       token = strtok (text+1, ",");  // get commander name
       (void) strcpy (pCmd2->Cname, token); // copy to structure
       pCmd2->Cname_ind = SQL_NTS;    // string is null terminated
       token = strtok (NULL, ",");    // extract Branch
       (void) strcpy (pCmd2->Branch, token); // copy to structure
       pCmd2->Branch_ind = SQL_NTS;   // string is null-terminated
       token = strtok (NULL, "#");    // extract Rank
       (void) strcpy (pCmd2->Rank, token);// copy to structure
       pCmd2->Rank_ind = SQL_NTS;     // string is null-terminated
       pCmd2->CVitae_ind = SQL_DATA_AT_EXEC; // will provide data via
       pCmd2->CVitae = NULL;          // no text
                                      // SQLPutData
    }
    else
    {
      ptxt = (pTEXT) malloc (sizeof(TEXT)); // allocate text structure
      memset (ptxt->text_line, ' ', sizeof(ptxt->text_line));
      ptxt->next = NULL;
      strcpy (ptxt->text_line, text); // populate text
      i = strlen (ptxt->text_line);   // remove '\n' and '\0'
      ptxt->text_line [i--] =' ';
      ptxt->text_line [i]   =' ';
      ptxt2 = pCmd2->CVitae;          // point to 1st text line
      if (ptxt2 == NULL)              // if text is empty
        pCmd2->CVitae = ptxt;         // insert 1st line
      else
      {
        while (ptxt2->next != NULL)   // else find last
          ptxt2 = ptxt2->next;
        ptxt2->next = ptxt;
      }
    }
  }

  if (pCmd2 != NULL)                 // if Commander structure allocated
  {
    pCmd2->next = (*pCmd);           // then insert LIFO
    (*pCmd)     = pCmd2;
  }

  exit:
/*****************************************************************/
/* Close the Commander file if necessary.                        */
/*****************************************************************/
  if (fp != NULL)
    (void) fclose (fp);

  return (rc);
}
/*****************************************************************/
/* Function GetBattles reads the "battle" file and creates a     */
/* linked list of BATTLE structures which is returned to the     */
/* caller.                                                       */
/*****************************************************************/
int GetBattles (pBATTLE *pBat,
                FILE    *fp)
{
  SQLINTEGER       rc = SQL_SUCCESS;
  pBATTLE          pBat2 = NULL;
  char             text [80];        // file text
  char             *t = NULL;
  char             *c = NULL;
  char             *token = NULL;
  pTEXT            ptxt, ptxt2;
  int              i;

  if ((fp = fopen ("DD:BATTLE", "r")) == NULL)  // open command file
  {
    rc = SQL_ERROR;                  // open failed
    goto exit;
  }
/*****************************************************************/
/* Process file and create BATTLES structures                    */
/*****************************************************************/
  while ((t = fgets (text, sizeof(text), fp)) != NULL)
  {
    if (text[0] == '#')               // if battle data
    {
       if (pBat2 != NULL)            // if Battle structure allocated
       {
         pBat2->next = (*pBat);      // then insert LIFO
         (*pBat)     = pBat2;
       }
       pBat2 = (pBATTLE) malloc (sizeof(BATTLE));
       pBat2->next = NULL;
       token = strtok (text+1, ",");  // get battle name
       (void) strcpy (pBat2->Bname, token); // copy to structure
       pBat2->Bname_ind = SQL_NTS;    // string is null terminated
       token = strtok (NULL, ",");    // extract Date
       (void) strcpy (pBat2->Date, token); // copy to structure
       pBat2->Date_ind = SQL_NTS;     // string is null terminated
       token = strtok (NULL, ",");    // extract Winner
       (void) strcpy (pBat2->Winner, token);// copy to structure
       pBat2->Winner_ind = SQL_NTS;   // string is null terminated
       token = strtok (NULL, "#");    // extract Loser
       (void) strcpy (pBat2->Loser, token);// copy to structure
       pBat2->Loser_ind = SQL_NTS;     // string is null terminated
       pBat2->Narrative_ind = SQL_DATA_AT_EXEC; // will provide data
                                       // in 80-character subsets
       pBat2->Narrative = NULL;        // no text yet
    }
    else
    {
      ptxt = (pTEXT) malloc (sizeof(TEXT)); // allocate text structure
      memset (ptxt->text_line, ' ', sizeof(ptxt->text_line));
      ptxt->next = NULL;
      strcpy (ptxt->text_line, text); // populate text
      i = strlen (ptxt->text_line);   // remove '\n' and '\0'
      ptxt->text_line [i--] =' ';
      ptxt->text_line [i]   =' ';
      ptxt2 = pBat2->Narrative;       // point to 1st text line
      if (ptxt2 == NULL)              // if text is empty
        pBat2->Narrative = ptxt;      // insert 1st line
      else
      {
        while (ptxt2->next != NULL)   // else find last
          ptxt2 = ptxt2->next;
        ptxt2->next = ptxt;
      }
    }
  }

  if (pBat2 != NULL)                 // if Battle structure allocated
  {
    pBat2->next = (*pBat);           // then insert LIFO
    (*pBat)     = pBat2;
  }

  exit:
/*****************************************************************/
/* Close the Battles file if necessary.                          */
/*****************************************************************/
  if (fp != NULL)
    (void) fclose (fp);

  return (rc);
}
```

```
/*****************************************************************/
/* Function PopCommanders processes the linked list of Commander */
/* structures and for each node, binds each column and then      */
/* Executes the Insert statement. Note that the Cur_Vitae column */
/* is populated by inserting 80 byte subsets via SQLPutData.     */
/*****************************************************************/
int PopCommanders (SQLHDBC    hDbc,
                   pCOMMANDER pCmd)
{
  SQLINTEGER       rc = SQL_SUCCESS;
  SQLHSTMT         hStmt = SQL_NULL_HSTMT;
  pCOMMANDER       pCmd2;
  int              i;
  SQLPOINTER       prgbValue;
  pTEXT            ptxt;

  // INSERT text with five parameter markers

  char            *pSQLStmt =
   "INSERT INTO COMMANDERS VALUES (?, ?, ?, ?)";
  /*****************************************************************/
  /* Allocate Statement Handle                                     */
  /*****************************************************************/
  rc = SQLAllocStmt (hDbc,
                     &hStmt);

  if (rc != SQL_SUCCESS)
    goto exit;
  /*****************************************************************/
  /* Prepare Statement.............                                */
  /*****************************************************************/
  rc = SQLPrepare (hStmt,
                   (SQLCHAR *) pSQLStmt,
                   SQL_NTS);

  if (rc != SQL_SUCCESS)
    goto exit;
  /*****************************************************************/
  /* Iterate through Commander List                                */
  /*****************************************************************/
  for (pCmd2 = pCmd; pCmd2 != NULL; pCmd2 = pCmd2->next)
  {
    rc = SQLBindParameter (hStmt,        // bind CINC
                           1,
                           SQL_PARAM_INPUT,
                           SQL_C_CHAR,
                           SQL_CHAR,
                           sizeof(pCmd2->Cname),
                           0,
                           pCmd2->Cname,
                           sizeof(pCmd2->Cname),
                           &pCmd2->Cname_ind);

    if (rc != SQL_SUCCESS)
      goto exit;
    rc = SQLBindParameter (hStmt,        // bind Branch
                           2,
                           SQL_PARAM_INPUT,
                           SQL_C_CHAR,
                           SQL_CHAR,
                           sizeof(pCmd2->Branch),
                           0,
                           pCmd2->Branch,
                           sizeof(pCmd2->Branch),
                           &pCmd2->Branch_ind);

    if (rc != SQL_SUCCESS)
      goto exit;

    rc = SQLBindParameter (hStmt,        // bind Rank
                           3,
                           SQL_PARAM_INPUT,
                           SQL_C_CHAR,
                           SQL_CHAR,
                           sizeof(pCmd2->Rank),
                           0,
                           pCmd2->Rank,
                           sizeof(pCmd2->Rank),
                           &pCmd2->Rank_ind);

    if (rc != SQL_SUCCESS)
      goto exit;

    (void) printf ("**** Inserting CINC = %s.\n", pCmd2->Cname);

    rc = SQLBindParameter (hStmt,        // bind CVitae, rgbValue
                           4,            // is specified as parm #
                           SQL_PARAM_INPUT, // and pcbValue is set
                           SQL_C_CHAR,      // to SQL_DATA_AT_EXEC
                           SQL_LONGVARCHAR, // execution will be
                           3200,            // deferred until
                           0,               // SQLParamData returns
                           (SQLPOINTER) 4,  // SQL_SUCCESS
                           0,
                           &pCmd2->CVitae_ind);
  /*****************************************************************/
  /* Three columns are bound... CVitae will be provided later....  */
  /* Execute statement.                                            */
  /*****************************************************************/
    rc = SQLExecute (hStmt);

    if (rc != SQL_NEED_DATA)     // expect SQL_NEED_DATA
      goto exit;
  /*****************************************************************/
  /* Invoke SQLParamData to position ODBC driver on input parameter*/
  /*****************************************************************/
    if ((rc = SQLParamData (hStmt,
                            &prgbValue)) != SQL_NEED_DATA)
      goto exit;
  /*****************************************************************/
  /* Iterate through CVitae in 80 byte increments.... pass to      */
  /* ODBC Driver via SQLPutData.                                   */
  /*****************************************************************/
    for (rc = SQL_SUCCESS, ptxt = pCmd2->CVitae;
         ((ptxt != NULL) && (rc == SQL_SUCCESS));
         ptxt = ptxt->next)
    {
      rc = SQLPutData (hStmt,
                       ptxt->text_line,
                       sizeof(ptxt->text_line));
    }
  /*****************************************************************/
  /* Invoke SQLParamData to trigger ODBC driver to execute the     */
  /* statement.                                                    */
  /*****************************************************************/
    if ((rc = SQLParamData (hStmt,
                            &prgbValue)) != SQL_SUCCESS)
      goto exit;

  }
  /*****************************************************************/
  /* Free Statement handle.                                        */
  /*****************************************************************/
  rc = SQLFreeStmt (hStmt,
                    SQL_DROP);
  exit:

  return (rc);
}
/*****************************************************************/
/* Function PopBattle processes the linked list of Battle        */
/* structures and for each node, binds each column and then      */
/* Executes the Insert statement.                                */
/*****************************************************************/
int PopBattles (SQLHDBC  hDbc,
                pBATTLE  pBat)
{
  SQLINTEGER       rc = SQL_SUCCESS;
  SQLHSTMT         hStmt = SQL_NULL_HSTMT;
  pBATTLE          pBat2;
  int              i;
  SQLPOINTER       prgbValue;
  pTEXT            ptxt;

  // INSERT text with five parameter markers

  char            *pSQLStmt =
   "INSERT INTO BATTLES VALUES (?, ?, ?, ?, ?)";
  /*****************************************************************/
  /* Allocate Statement Handle                                     */
  /*****************************************************************/
  rc = SQLAllocStmt (hDbc,
                     &hStmt);

  if (rc != SQL_SUCCESS)
    goto exit;
  /*****************************************************************/
  /* Prepare Statement.............                                */
  /*****************************************************************/
  rc = SQLPrepare (hStmt,
                  (SQLCHAR *) pSQLStmt,
                   SQL_NTS);

  if (rc != SQL_SUCCESS)
    goto exit;
```

```
/******************************************************************/
/* Iterate through Battles List                                   */
/******************************************************************/
for (pBat2 = pBat; pBat2 != NULL; pBat2 = pBat2->next)
{
  rc = SQLBindParameter (hStmt,       // bind Battlename
                         1,
                         SQL_PARAM_INPUT,
                         SQL_C_CHAR,
                         SQL_CHAR,
                         sizeof(pBat2->Bname),
                         0,
                         pBat2->Bname,
                         sizeof(pBat2->Bname),
                         &pBat2->Bname_ind);

  if (rc != SQL_SUCCESS)
    goto exit;

  rc = SQLBindParameter (hStmt,       // bind Date
                         2,
                         SQL_PARAM_INPUT,
                         SQL_C_CHAR,
                         SQL_CHAR,
                         sizeof(pBat2->Date),
                         0,
                         pBat2->Date,
                         sizeof(pBat2->Date),
                         &pBat2->Date_ind);

  if (rc != SQL_SUCCESS)
    goto exit;

  rc = SQLBindParameter (hStmt,       // bind Winner
                         3,
                         SQL_PARAM_INPUT,
                         SQL_C_CHAR,
                         SQL_CHAR,
                         sizeof(pBat2->Winner),
                         0,
                         pBat2->Winner,
                         sizeof(pBat2->Winner),
                         &pBat2->Winner_ind);

  if (rc != SQL_SUCCESS)
    goto exit;

  rc = SQLBindParameter (hStmt,       // bind Loser
                         4,
                         SQL_PARAM_INPUT,
                         SQL_C_CHAR,
                         SQL_CHAR,
                         sizeof(pBat2->Loser),
                         0,
                         pBat2->Loser,
                         sizeof(pBat2->Loser),
                         &pBat2->Loser_ind);

  if (rc != SQL_SUCCESS)
    goto exit;

  (void) printf ("**** Inserting Battle = %s.\n", pBat2->Bname);
```

```
    rc = SQLBindParameter (hStmt,        // bind Narrative, rgbValue
                           5,            // is specified as parm #
                           SQL_PARAM_INPUT, // and pcbValue is set
                           SQL_C_CHAR,   // to SQL_DATA_AT_EXEC
                           SQL_LONGVARCHAR, // execution will be
                           3200,            // deferred until
                           0,               // SQLParamData returns
                           (SQLPOINTER) 4,  // SQL_SUCCESS
                           0,
                           &pBat2->Narrative_ind);
  /*****************************************************************/
  /* Four columns are bound... Narrative will be provided later.   */
  /* Execute statement.                                            */
  /*****************************************************************/
    rc = SQLExecute (hStmt);

    if (rc != SQL_NEED_DATA)     // expect SQL_NEED_DATA
      goto exit;
  /*****************************************************************/
  /* Invoke SQLParamData to position ODBC driver on input parameter*/
  /*****************************************************************/
    if ((rc = SQLParamData (hStmt,
                            &prgbValue)) != SQL_NEED_DATA)
      goto exit;
  /*****************************************************************/
  /* Iterate through Narrative in 80 byte increments.... pass to   */
  /* ODBC Driver via SQLPutData.                                   */
  /*****************************************************************/
    for (rc = SQL_SUCCESS, ptxt = pBat2->Narrative;
         ((ptxt != NULL) && (rc == SQL_SUCCESS));
         ptxt = ptxt->next)
    {
      rc = SQLPutData (hStmt,
                       ptxt->text_line,
                       sizeof(ptxt->text_line));
    }
  /*****************************************************************/
  /* Invoke SQLParamData to trigger ODBC driver to execute the    */
  /* statement.                                                    */
  /*****************************************************************/
    if ((rc = SQLParamData (hStmt,
                            &prgbValue)) != SQL_SUCCESS)
      goto exit;

  }
  /*****************************************************************/
  /* Free Statement handle.                                        */
  /*****************************************************************/
  rc = SQLFreeStmt (hStmt,
                    SQL_DROP);
  exit:

  return (rc);
}
```

STEP 3. Read tables.

```
 /******************************************************************/
 /*  DB2 for OS/390 CLI Sample:                                    */
 /*     Exercises the ODBC Version 2.0 APIs necessary to perform   */
 /*     an outer join of the COMMANDERS and BATTLES tables to      */
 /*     identify the winners of all battles that occurred after    */
 /*     January 1 1920. Then retrieves the results.  The Narrative */
 /*     is retrieved via SQLGetData.                               */
 /******************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlcli1.h>

 // global static variables

static char Bname     [20];
static char Rank      [20];
static char CINC      [20];
static char Branch    [30];
static char Narrative [80];
 /******************************************************************/
 /* Structure to represent one column of answer set -- expected    */
 /* results ---------------------->                                */
 /******************************************************************/
typedef struct Column
{
  char             Col_name [20];       // column name
  SQLINTEGER       SQLType;             // SQL Type
  char            *pSelect_item;        // Select item
  void            *rgbValue;            // target for SQLBindCol
  SQLINTEGER       cbValueMax;          // Maximum size of bound column
  SQLINTEGER       cbValue;             // return size
} COLUMN, *pCOLUMN;
 /******************************************************************/
 /* Expected Columns from ODBC Outer Join                          */
 /******************************************************************/
COLUMN EXP_COL [5] = {
  { "BATTLENAME", SQL_CHAR        , "Battles.Battlename" , (void *)Bname  , size
  { "RANK"      , SQL_CHAR        , "Commanders.Rank"    , (void *)Rank   , size
  { "CINC"      , SQL_CHAR        , "Commanders.CINC"    , (void *)CINC   , size
  { "BRANCH"    , SQL_CHAR        , "Commanders.Branch"  , (void *)Branch , size
  { "NARRATIVE" , SQL_LONGVARCHAR , "Battles.Narrative"  , NULL           , 0
                     } ;
 /******************************************************************/
 /* CLI APIs required                                              */
 /******************************************************************/
SQLUSMALLINT ODBC_api [6] = {
                             SQL_API_SQLDISCONNECT   ,
                             SQL_API_SQLFETCH        ,
                             SQL_API_SQLTRANSACT     ,
                             SQL_API_SQLBINDCOL      ,
                             SQL_API_SQLEXECUTE      ,
                             SQL_API_SQLGETDATA
                            } ;
  /******************************************************************/
  /* StripBlanks functions prototype                                */
  /******************************************************************/
void Strip_Blanks (char *ptext,
                   int   size);
int main( )
{
   SQLHENV         hEnv    = SQL_NULL_HENV;
   SQLHDBC         hDbc    = SQL_NULL_HDBC;
   SQLHSTMT        hStmt   = SQL_NULL_HSTMT;
   SQLRETURN       rc      = SQL_SUCCESS;
   SQLSMALLINT     t3_small= 0;
   SQLINTEGER      RETCODE = 0;
   int             i,j;
   char            pSELECT [255];
   pCOLUMN         pCol;
   char            *outer_join =
    "{ oj Battles LEFT OUTER JOIN Commanders ON Battles.Winner=Commanders.CINC }

   char            *predicate =
    " WHERE Battles.Date > {d '1920-01-01'};";

   // SQLDataSources parameters

   SQLCHAR         szDSN [19] = {0};
   SQLSMALLINT     fDirection = SQL_FETCH_FIRST;
   SQLSMALLINT     cbDSNMax   = sizeof(szDSN);
   SQLSMALLINT     cbDSN      = 0;
   SQLSMALLINT     *pcbDSN    = &cbDSN;
   SQLCHAR         szDescription [40];
   SQLSMALLINT     cbDescriptionMax = sizeof(szDescription);
   SQLSMALLINT     *pcbDescription = &t3_small;

   // SQLGetFunctions parameters

   SQLUSMALLINT     fExists  = SQL_TRUE;
   SQLUSMALLINT    *pfExists = &fExists;

   // SQLGetInfo      parameters

   char            oj[2];
   SQLSMALLINT     cbInfoValue;

   // SQLGetData      parameters

   SQLINTEGER      cbValue;

   // SQLDescribeCol parameters

   SQLCHAR          szColName [20];
   SQLSMALLINT      cbColName;
   SQLSMALLINT      fSqlType;
   SQLUINTEGER      cbColDef;
   SQLSMALLINT      ibScale;
   SQLSMALLINT      fNullable;

   (void) printf ("**** DB2 for OS/390 CLI: Identify Winners.\n\n");
  /******************************************************************/
  /* Allocate Environment Handle                                    */
  /******************************************************************/
   RETCODE = SQLAllocEnv(&hEnv);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;
```

```
/*****************************************************************/
/* See if STLEC1 is a known DSN                                  */
/*****************************************************************/
 RETCODE = SQLDataSources (hEnv,
                           fDirection,
                           szDSN,
                           cbDSNMax,
                           pcbDSN,
                           szDescription,
                           cbDescriptionMax,
                           pcbDescription); // fetch first DSN

 while ((RETCODE                            == SQL_SUCCESS) &&
        (memcmp(szDSN, "STLEC1", (*pcbDSN)) != 0))
 {
   RETCODE = SQLDataSources (hEnv,
                             SQL_FETCH_NEXT,
                             szDSN,
                             cbDSNMax,
                             pcbDSN,
                             szDescription,
                             cbDescriptionMax,
                             pcbDescription); // fetch next DSN
 }

 if (RETCODE != SQL_SUCCESS)
 {
   (void) printf ("**** DSN = STLEC1 not known.\n");
   goto dberror;
 }
/*****************************************************************/
/* Allocate Connection Handle to DSN                             */
/*****************************************************************/
 RETCODE = SQLAllocConnect(hEnv,
                           &hDbc);

 if( RETCODE != SQL_SUCCESS )      // Could not get a Connect Handle
   goto dberror;
/*****************************************************************/
/* CONNECT TO data source (STLEC1)                               */
/*****************************************************************/
 RETCODE = SQLConnect(hDbc,        // Connect handle
                      szDSN,       // DSN
                      (*pcbDSN),   // length of DSN
                      NULL,        // Null UID
                      0   ,
                      NULL,        // Null Auth string
                      0);

 if( RETCODE != SQL_SUCCESS )      // Connect failed
   goto dberror;
/*****************************************************************/
/* See if DSN supports required ODBC APIs                        */
/*****************************************************************/
 for (i = 0, (*pfExists = SQL_TRUE);
      (i < 6 && (*pfExists) == SQL_TRUE);
      i++)
 {
   RETCODE = SQLGetFunctions (hDbc,
                              ODBC_apiõi•,
                              pfExists);
 }
 if (*pfExists == SQL_FALSE)      // a required API is not supported
   goto dberror;
/*****************************************************************/
/* See if DSN supports OUTER JOIN ---------------->              */
/*****************************************************************/
 if ((RETCODE = SQLGetInfo(hDbc,
                           SQL_OUTER_JOINS,
                           oj,
                           sizeof(oj),
                           &cbInfoValue)) != SQL_SUCCESS)
   goto dberror;

 if (strcmp(oj, "N") == 0)
 {
   RETCODE = SQL_ERROR;
   goto dberror;
 }
/*****************************************************************/
/* DSN supports Outer Join. Allocate statement handle and build  */
/* OUTER JOIN text ---------------->                             */
/*****************************************************************/
 RETCODE = SQLAllocStmt (hDbc,
                         &hStmt);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;

 (void) strcpy (pSELECT, "SELECT ");

 for (i = 0, pCol = EXP_COL;
      (i < 5);
      i++, pCol++)
 {
   (void) strcat (pSELECT, pCol->pSelect_item);
   if (i <=3)
     (void) strcat (pSELECT, ",");
 }

 (void) strcat (pSELECT, " FROM ");
 (void) strcat (pSELECT, outer_join);
 (void) strcat (pSELECT, predicate);
/*****************************************************************/
/* OUTER JOIN text is complete ---> Prepare statement ---->      */
/*****************************************************************/
 RETCODE = SQLPrepare (hStmt,
                       (SQLCHAR *)pSELECT,
                       strlen(pSELECT));

 if (RETCODE != SQL_SUCCESS)
   goto dberror;
/*****************************************************************/
/* Validate that column names and types are as expected --->     */
/*****************************************************************/
 for (i = 1, pCol = EXP_COL;
      (i <= 5) && (RETCODE == SQL_SUCCESS);
      i++, pCol++)
 {
   RETCODE = SQLDescribeCol (hStmt,
                             i,
                             szColName        ,
                             sizeof(szColName),
                             &cbColName       ,
                             &fSqlType        ,
                             &cbColDef        ,
                             &ibScale         ,
                             &fNullable);

   if ((RETCODE != SQL_SUCCESS)                          ||
       (strcmp ((char *)szColName, pCol->Col_name) != 0) ||
       (fSqlType != pCol->SQLType))
   {
     RETCODE = SQL_ERROR;
     goto dberror;
   }
 }
/*****************************************************************/
/* Bind 4 of the columns --- leaving Narrative unbound           */
/*****************************************************************/
 for (pCol = EXP_COL, i = 1;
      (i < 5) && (RETCODE == SQL_SUCCESS);
      i++, pCol++)
 {
   RETCODE = SQLBindCol (hStmt,             // bind Column
                         i,
                         SQL_C_DEFAULT,
                         (SQLPOINTER) pCol->rgbValue,
                         pCol->cbValueMax,
                         &pCol->cbValue);
 }
/*****************************************************************/
/* Execute the statement generating a results set -------------> */
/*****************************************************************/
 RETCODE = SQLExecute (hStmt);

 if (RETCODE != SQL_SUCCESS)
   goto dberror;
/*****************************************************************/
/* Fetch Battle name, Commander Rank, Commander Name and Branch  */
/* via SQLFetch. Then retrieve Battle Narrative via SQLGetData   */
/* in subsets of 80 characters.                                  */
/*****************************************************************/
 while ((RETCODE = SQLFetch (hStmt)) != SQL_NO_DATA_FOUND)
 {
   (void) Strip_Blanks (Bname,
                        sizeof(Bname)-1);

   (void) printf ("**** Battle = %s\n",  // null-terminate string
                  Bname);

   (void) Strip_Blanks (Rank,            // null-terminate string
                        sizeof(Rank)-1);

   (void) Strip_Blanks (CINC,            // null-terminate string
                        sizeof(CINC)-1);

   (void) Strip_Blanks (Branch,          // null-terminate string
                        sizeof(Branch)-1);

   (void) printf ("**** Winner is %s, %s, %s\n\n",
                  Rank,
                  CINC,
                  Branch);

   (void) printf ("**** Battle Narrative follows ----->\n\n");
```

```
     for (i = 0; (i < 3200 && RETCODE != SQL_NO_DATA_FOUND); i += 80)
     {
       RETCODE = SQLGetData (hStmt,
                             5,
                             SQL_C_CHAR,
                             Narrative,
                             sizeof(Narrative)+1,
                             &cbValue);

       if (RETCODE != SQL_NO_DATA_FOUND)
         (void) printf ("%s\n", Narrative);
     }
   }
  /*******************************************************************/
  /* Free Statement handle.                                          */
  /*******************************************************************/
   RETCODE = SQLFreeStmt (hStmt,
                          SQL_DROP);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;
  /*******************************************************************/
  /* DISCONNECT from data source                                     */
  /*******************************************************************/
   RETCODE = SQLDisconnect(hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;
  /*******************************************************************/
  /* Deallocate Connection Handle                                    */
  /*******************************************************************/
   RETCODE = SQLFreeConnect (hDbc);

   if (RETCODE != SQL_SUCCESS)
     goto dberror;
  /*******************************************************************/
  /* Free Environment Handle                                         */
  /*******************************************************************/
   RETCODE = SQLFreeEnv (hEnv);

   if (RETCODE == SQL_SUCCESS)
     goto exit;

   dberror:
   RETCODE=12;

   exit:

   (void) printf("\n\nDB2 for OS/390 CLI: Identify Winners TERMINATION\n\n   ");

   if (RETCODE!=0)
     (void) printf("\n\nDB2 for OS/390 CLI: Identify Winners WAS UNSUCCESSFUL\n"
   else
     (void) printf("\n\nDB2 for OS/390 CLI: Identify Winners WAS SUCCESSFUL\n\n"

   return(RETCODE);
}
```

```
/*****************************************************************/
/* Function StripBlanks removes trailing blanks from a fixed     */
/* CHAR field.                                                   */
/*****************************************************************/
void Strip_Blanks (char *ptext,
                   int   size)
{
  int i = size;
  while (ptext [size--] != ' ');    // move backwards until 1st
                                    // non-blank
  ptext [size + 1] = '\0';          // append null terminator
  return;
}
```

Stored Procedure

This example shows a DB2 CLI client application (APD29) calling a DB2 CLI stored procedure (SPD29). It includes very fundamental processing of query result sets (a query cursor opened in a stored procedure and return to client for fetching). For completeness, the CREATE TABLE, data INSERTs and SYSIBM.SYSPROCEDURES declaration is provided.

STEP 1. CREATE TABLE

```
    printf("\nAPDDL SQLExecDirect  stmt=%d",__LINE__);
   strcpy((char *)sqlstmt,
"CREATE TABLE TABLE2A (INT4 INTEGER,SMINT SMALLINT,FLOAT8 FLOAT");
   strcat((char *)sqlstmt,
",DEC312 DECIMAL(31,2),CHR10 CHARACTER(10),VCHR20 VARCHAR(20)");
   strcat((char *)sqlstmt,
",LVCHR LONG VARCHAR,CHRSB CHAR(10),CHRBIT CHAR(10) FOR BIT DATA");
   strcat((char *)sqlstmt,
",DDATE DATE,TTIME TIME,TSTMP TIMESTAMP)");
   printf("\nAPDDL sqlstmt=%s",sqlstmt);
   rc=SQLExecDirect(hstmt,sqlstmt,SQL_NTS);
   if( rc != SQL_SUCCESS ) goto dberror;
```

STEP 2. INSERT 101 ROWS INTO TABLE

```
   /* insert 100 rows into table2a */
   for (jx=1;jx<=100 ;jx++ ) {
     printf("\nAPDIN SQLExecDirect  stmt=%d",__LINE__);
     strcpy((char *)sqlstmt,"insert into table2a values(");
     sprintf((char *)sqlstmt+strlen((char *)sqlstmt),"%ld",jx);
     strcat((char *)sqlstmt,
",4,8.2E+30,1515151515151.51,'CHAR','VCHAR','LVCCHAR','SBCS'");
     strcat((char *)sqlstmt,
",'MIXED','01/01/1991','3:33 PM','1999-09-09-09.09.09.090909')");
     printf("\nAPDIN sqlstmt=%s",sqlstmt);
     rc=SQLExecDirect(hstmt,sqlstmt,SQL_NTS);
     if( rc != SQL_SUCCESS ) goto dberror;
   } /* endfor */
```

STEP 3. DEFINE STORED PROCEDURE IN SYSIBM.SYSPROCEDURES

```
DELETE FROM  SYSIBM.SYSPROCEDURES
 WHERE PROCEDURE='SPD29';

INSERT INTO  SYSIBM.SYSPROCEDURES
 VALUES('SPD29',
        ' ',
        ' ',
        'SPD29',
        ' ',
        'DSNAOCLI',
        'C',
        0,
        'Y',
        'N',
        ' ',
        'INTEGER INOUT',2,' ','M','N','N');
```

STEP 4. STORED PROCEDURE

```
  /*START OF SPD29****************************************************/
   /* PRAGMA TO CALL PLI SUBRTN CSPSUB TO ISSUE CONSOLE MSGS        */
   #pragma options (rent)
   #pragma runopts(plist(os))
   /******************************************************************/
   /* Include the 'C' include files                                  */
   /******************************************************************/
   #include <stdio.h>
   #include <string.h>
   #include <stdlib.h>
   #include "sqlcli1.h"
   #include <sqlca.h>
   #include <decimal.h>
   #include <wcstr.h>
   /******************************************************************/
   /* Variables for COMPARE routines                                 */
   /******************************************************************/
#ifndef NULL
#define NULL   0
#endif

    SQLHENV henv = SQL_NULL_HENV;
    SQLHDBC hdbc = SQL_NULL_HDBC;
    SQLHSTMT hstmt = SQL_NULL_HSTMT;
    SQLHSTMT hstmt2 = SQL_NULL_HSTMT;
    SQLRETURN rc = SQL_SUCCESS;
    SQLINTEGER      id;
    SQLCHAR         name[51];
    SQLINTEGER      namelen, intlen, colcount;
    SQLSMALLINT     scale;
    struct sqlca    sqlca;
    SQLCHAR   server[18];
    SQLCHAR   uid[30];
    SQLCHAR   pwd[30];
    SQLCHAR   sqlstmt[500];
    SQLCHAR   sqlstmt2[500];
    SQLSMALLINT  pcpar=0;
    SQLSMALLINT  pccol=0;
    SQLCHAR         cursor[19];
    SQLSMALLINT   cursor_len;

   SQLINTEGER    SPCODE;
   struct {
     SQLSMALLINT LEN;
     SQLCHAR  DATA_200•; }         STMTSQL;

   SQLSMALLINT                 H1SMINT;
   SQLINTEGER                  H1INT4;
   SQLDOUBLE               H1FLOAT8;
   SQLDOUBLE               H1DEC312;
   SQLCHAR                  H1CHR10[11];
   SQLCHAR                  H1VCHR20[21];
   SQLCHAR                  H1LVCHR[21];
   SQLCHAR                  H1CHRSB[11];
   SQLCHAR                  H1CHRBIT[11];
   SQLCHAR                  H1DDATE[11];
   SQLCHAR                  H1TTIME[9];
   SQLCHAR                  H1TSTMP[27];
```

```
SQLSMALLINT                 I1SMINT;
SQLSMALLINT                 I1INT4;
SQLSMALLINT                 I1FLOAT8;
SQLSMALLINT                 I1DEC312;
SQLSMALLINT                 I1CHR10;
SQLSMALLINT                 I1VCHR20;
SQLSMALLINT                 I1LVCHR;
SQLSMALLINT                 I1CHRSB;
SQLSMALLINT                 I1CHRBIT;
SQLSMALLINT                 I1DDATE;
SQLSMALLINT                 I1TTIME;
SQLSMALLINT                 I1TSTMP;

SQLINTEGER                  LEN_H1SMINT;
SQLINTEGER                  LEN_H1INT4;
SQLINTEGER                  LEN_H1FLOAT8;
SQLINTEGER                  LEN_H1DEC312;
SQLINTEGER                  LEN_H1CHR10;
SQLINTEGER                  LEN_H1VCHR20;
SQLINTEGER                  LEN_H1LVCHR;
SQLINTEGER                  LEN_H1CHRSB;
SQLINTEGER                  LEN_H1CHRBIT;
SQLINTEGER                  LEN_H1DDATE;
SQLINTEGER                  LEN_H1TTIME;
SQLINTEGER                  LEN_H1TSTMP;

SQLSMALLINT                 H2SMINT;
SQLINTEGER                  H2INT4;
SQLDOUBLE                H2FLOAT8;
SQLCHAR                  H2CHR10[11];
SQLCHAR                  H2VCHR20[21];
SQLCHAR                  H2LVCHR[21];
SQLCHAR                  H2CHRSB[11];
SQLCHAR                  H2CHRBIT[11];
SQLCHAR                  H2DDATE[11];
SQLCHAR                  H2TTIME[9];
SQLCHAR                  H2TSTMP[27];

SQLSMALLINT                 I2SMINT;
SQLSMALLINT                 I2INT4;
SQLSMALLINT                 I2FLOAT8;
SQLSMALLINT                 I2CHR10;
SQLSMALLINT                 I2VCHR20;
SQLSMALLINT                 I2LVCHR;
SQLSMALLINT                 I2CHRSB;
SQLSMALLINT                 I2CHRBIT;
SQLSMALLINT                 I2DDATE;
SQLSMALLINT                 I2TTIME;
SQLSMALLINT                 I2TSTMP;

SQLINTEGER                  LEN_H2SMINT;
SQLINTEGER                  LEN_H2INT4;
SQLINTEGER                  LEN_H2FLOAT8;
SQLINTEGER                  LEN_H2CHR10;
SQLINTEGER                  LEN_H2VCHR20;
SQLINTEGER                  LEN_H2LVCHR;
SQLINTEGER                  LEN_H2CHRSB;
SQLINTEGER                  LEN_H2CHRBIT;
SQLINTEGER                  LEN_H2DDATE;
SQLINTEGER                  LEN_H2TTIME;
SQLINTEGER                  LEN_H2TSTMP;
```

```
   SQLCHAR locsite[18] =  "stlec1";
   SQLCHAR remsite[18] =  "stlec1b";

   SQLCHAR    spname[8];
   SQLINTEGER    ix,jx,locix;
   SQLINTEGER     result;
   SQLCHAR    state_blank[6] ="     ";

SQLRETURN
check_error(SQLHENV henv,
            SQLHDBC hdbc,
            SQLHSTMT hstmt,
            SQLRETURN frc);
SQLRETURN
prt_sqlca();
   /******************************************************************/
   /* Main Program                                                   */
   /******************************************************************/
SQLINTEGER
main(SQLINTEGER argc, SQLCHAR *argv[] )
{
 printf("\nSPD29 INITIALIZATION");
 scale = 0;
 rc=0;

 rc=0;
 SPCODE=0;

 /* argv0 = sp module name */
 if (argc != 2)
  {
   printf("SPD29 parm number error\n   ");
   printf("SPD29 EXPECTED =%d\n",3);
   printf("SPD29 received =%d\n",argc);
   goto dberror;
  }
  strcpy((char *)spname,(char *)argv[0]);
  result = strncmp((char *)spname,"SPD29",5);
  if (result != 0)
   {
    printf("SPD29 argv0 sp name  error\n   ");
    printf("SPD29 compare rusult =%i\n",result);
    printf("SPD29 expected =%s\n","SPD29");
    printf("SPD29 received spname=%s\n",spname);
    printf("SPD29 received argv0 =%s\n",argv[0]);
    goto dberror;
   }
  /* get input spcode value */
  SPCODE        = *(SQLINTEGER *)  argv[1];
 printf("\nSPD29 SQLAllocEnv        number=     1\n");
 henv=0;
 rc = SQLAllocEnv(&henv);
 if( rc != SQL_SUCCESS ) goto dberror;
 printf("\nSPD29-henv=%i",henv);
 /*****************************************************************/
 printf("\nSPD29 SQLAllocConnect                             ");
 hdbc=0;
 SQLAllocConnect(henv, &hdbc);
 if( rc != SQL_SUCCESS ) goto dberror;
 printf("\nSPD29-hdbc=%i",hdbc);
```

```
/*****************************************************************/
/* Make sure no autocommits after cursors are allocated, commits */
/* cause sp failure. No-autocommit could also be specified in the*/
/* INI file.                                                     */
/* Also, sp could be defined with COMMIT_ON_RETURN in the        */
/* DB2 catalog table SYSIBM.SYSPROCEDURES, but be wary that this */
/* removes control from the client appl to control commit scope. */
/*****************************************************************/
printf("\nSPD29 SQLSetConnectOption-no autocommits in stored procs");
rc = SQLSetConnectOption(hdbc,SQL_AUTOCOMMIT,SQL_AUTOCOMMIT_OFF);
if( rc != SQL_SUCCESS ) goto dberror;
/*****************************************************************/
printf("\nSPD29 SQLConnect  NULL connect in stored proc      ");
strcpy((char *)uid,"cliuser");
strcpy((char *)pwd,"password");
printf("\nSPD29 server=%s",NULL);
printf("\nSPD29 uid=%s",uid);
printf("\nSPD29 pwd=%s",pwd);
rc=SQLConnect(hdbc, NULL, 0, uid, SQL_NTS, pwd, SQL_NTS);
if( rc != SQL_SUCCESS ) goto dberror;
/*****************************************************************/
/* Start SQL statements *****************************************/
/*****************************************************************/
switch(SPCODE)
{
 /****************************************************************/
 /* CASE(SPCODE=0) do nothing and return                    *****/
 /****************************************************************/
  case 0:
    break;
  case 1:
 /****************************************************************/
 /* CASE(SPCODE=1)                                          *****/
 /*  -sqlprepare/sqlexecute insert int4=200                 *****/
 /*  -sqlexecdirect         insert int4=201                 *****/
 /* *validated in client appl that inserts occur            *****/
 /****************************************************************/
    SPCODE=0;

    printf("\nSPD29 SQLAllocStmt                                \n");
    hstmt=0;
    rc=SQLAllocStmt(hdbc, &hstmt);
    if( rc != SQL_SUCCESS ) goto dberror;
    printf("\nSPD29-hstmt=%i\n",hstmt);

    printf("\nSPD29 SQLPrepare                                  \n");
    strcpy((char *)sqlstmt,
    "insert into TABLE2A(int4) values(?)");
    printf("\nSPD29 sqlstmt=%s",sqlstmt);
    rc=SQLPrepare(hstmt,sqlstmt,SQL_NTS);
    if( rc != SQL_SUCCESS ) goto dberror;

    printf("\nSPD29 SQLNumParams                                \n");
    rc=SQLNumParams(hstmt,&pcpar);
    if( rc != SQL_SUCCESS) goto dberror;
    if (pcpar!=1) {
       printf("\nSPD29 incorrect pcpar=%d",pcpar);
       goto dberror;
      }
```

```
   printf("\nSPD29 SQLBindParameter   int4                        \n");
   H1INT4=200;
   LEN_H1INT4=sizeof(H1INT4);
   rc=SQLBindParameter(hstmt,1,SQL_PARAM_INPUT,SQL_C_LONG,
   SQL_INTEGER,0,0,&H1INT4,0,(SQLINTEGER *)&LEN_H1INT4);
   if( rc != SQL_SUCCESS) goto dberror;

   printf("\nSPD29 SQLExecute                                   \n");
   rc=SQLExecute(hstmt);
   if( rc != SQL_SUCCESS) goto dberror;

   printf("\nSPD29  SQLFreeStmt                                  \n");
   rc=SQLFreeStmt(hstmt, SQL_DROP);
   if( rc != SQL_SUCCESS ) goto dberror;
/******************************************************************/
  printf("\nAPDIN SQLAllocStmt   stmt=%d",__LINE__);
  hstmt=0;
  rc=SQLAllocStmt(hdbc, &hstmt);
  if( rc != SQL_SUCCESS ) goto dberror;
  printf("\nAPDIN-hstmt=%i\n",hstmt);

  jx=201;
  printf("\nAPDIN SQLExecDirect  stmt=%d",__LINE__);
  strcpy((char *)sqlstmt,"insert into table2a values(");
  sprintf((char *)sqlstmt+strlen((char *)sqlstmt),"%ld",jx);
  strcat((char *)sqlstmt,
  ",4,8.2E+30,1515151515151.51,'CHAR','VCHAR','LVCCHAR','SBCS'");
    strcat((char *)sqlstmt,
  ",'MIXED','01/01/1991','3:33 PM','1999-09-09-09.09.09.090909')");
  printf("\nAPDIN sqlstmt=%s",sqlstmt);
  rc=SQLExecDirect(hstmt,sqlstmt,SQL_NTS);
  if( rc != SQL_SUCCESS ) goto dberror;

  break;
/******************************************************************/
 case 2:
/*****************************************************************/
/* CASE(SPCODE=2)                                            *****/
/*  -sqlprepare/sqlexecute select int4  from table2a         *****/
/*  -sqlprepare/sqlexecute select chr10 from table2a         *****/
/* *qrs cursors should be allocated and left open by CLI     *****/
/*****************************************************************/
   SPCODE=0;

/* generate 1st  queryresultset  */
  printf("\nSPD29 SQLAllocStmt                                  \n");
  hstmt=0;
  rc=SQLAllocStmt(hdbc, &hstmt);
  if( rc != SQL_SUCCESS ) goto dberror;
  printf("\nSPD29-hstmt=%i\n",hstmt);

  printf("\nSPD29 SQLPrepare                                    \n");
  strcpy((char *)sqlstmt,
    "SELECT INT4 FROM TABLE2A");
  printf("\nSPD29 sqlstmt=%s",sqlstmt);
  rc=SQLPrepare(hstmt,sqlstmt,SQL_NTS);
  if( rc != SQL_SUCCESS ) goto dberror;

  printf("\nSPD29 SQLExeccute                                   \n");
  rc=SQLExecute(hstmt);
  if( rc != SQL_SUCCESS ) goto dberror;
 /* allocate 2nd stmt handle for 2nd queryresultset */
 /* generate 2nd  queryresultset  */
   printf("\nSPD29 SQLAllocStmt                                 \n");
   hstmt=0;
   rc=SQLAllocStmt(hdbc, &hstmt2);
   if( rc != SQL_SUCCESS ) goto dberror;
   printf("\nSPD29-hstmt2=%i\n",hstmt2);

   printf("\nSPD29 SQLPrepare                                   \n");
   strcpy((char *)sqlstmt2,
    "SELECT CHR10 FROM TABLE2A");
   printf("\nSPD29 sqlstmt2=%s",sqlstmt2);
   rc=SQLPrepare(hstmt2,sqlstmt2,SQL_NTS);
   if( rc != SQL_SUCCESS ) goto dberror;

   printf("\nSPD29 SQLExeccute                                  \n");
   rc=SQLExecute(hstmt2);
   if( rc != SQL_SUCCESS ) goto dberror;

   /*leave queryresultset cursor open for fetch back at client appl */

   break;
 /****************************************************************/
  default:
   {
    printf("SPD29 INPUT SPCODE INVALID\n");
    printf("SPD29...EXPECTED SPCODE=0-2\n");
    printf("SPD29...RECIEVED SPCODE=%i\n",SPCODE);
    goto dberror;
    break;
    }
}
/*****************************************************************/
/* End SQL statements ********************************************/
/*****************************************************************/
/*Be sure NOT to put a SQLTransact with SQL_COMMIT in a DB2/MVS  */
/*  stored procedure.  Commit is not allowed in a DB2/MVS        */
/*  stored procedure.  Use SQLTransact with SQL_ROLLBACK to      */
/*  force a must rollback condition for this sp and calling      */
/*  client application.                                          */
/*****************************************************************/
printf("\nSPD29 SQLDisconnect      number=     4\n");
rc=SQLDisconnect(hdbc);
if( rc != SQL_SUCCESS ) goto dberror;
/*****************************************************************/
printf("\nSPD29 SQLFreeConnect     number=     5\n");
rc = SQLFreeConnect(hdbc);
if( rc != SQL_SUCCESS ) goto dberror;
/*****************************************************************/
printf("\nSPD29 SQLFreeEnv         number=     6\n");
rc = SQLFreeEnv(henv);
if( rc != SQL_SUCCESS ) goto dberror;
/*****************************************************************/
goto pgmend;

dberror:
printf("\nSPD29 entry dberror label");
printf("\nSPD29 rc=%d",rc);
check_error(henv,hdbc,hstmt,rc);
printf("\nSPD29 SQLFreeEnv         number=     7\n");
rc = SQLFreeEnv(henv);
```

```
 printf("\nSPD29 rc=%d",rc);
 rc=12;
 rc=12;
 SPCODE=12;
 goto pgmend;

 pgmend:

 printf("\nSPD29  TERMINATION   ");
 if (rc!=0)
  {
  printf("\nSPD29 WAS NOT NOT NOT SUCCESSFUL");
  printf("\nSPD29 SPCODE  = %i", SPCODE      );
  printf("\nSPD29 rc          = %i", rc  );
  }
 else
 {
  printf("\nSPD29 WAS SUCCESSFUL");
 }
 /* assign output spcode value */
 *(SQLINTEGER  *) argv[1] = SPCODE;
 exit;
}  /*END MAIN*/
/*******************************************************************
** check_error - call print_error(), checks severity of return code
*******************************************************************/
SQLRETURN
check_error(SQLHENV henv,
            SQLHDBC hdbc,
            SQLHSTMT hstmt,
            SQLRETURN frc )
{
    SQLCHAR         buffer[SQL_MAX_MESSAGE_LENGTH + 1];
    SQLCHAR         cli_sqlstate[SQL_SQLSTATE_SIZE + 1];
    SQLINTEGER     cli_sqlcode;
    SQLSMALLINT   length;

    printf("\nSPD29 entry check_error rtn");

    switch (frc) {
    case SQL_SUCCESS:
        break;
    case SQL_INVALID_HANDLE:
        printf("\nSPD29 check_error> SQL_INVALID HANDLE ");
    case SQL_ERROR:
        printf("\nSPD29 check_error> SQL_ERROR ");
        break;
    case SQL_SUCCESS_WITH_INFO:
        printf("\nSPD29 check_error>  SQL_SUCCESS_WITH_INFO");
        break;
    case SQL_NO_DATA_FOUND:
        printf("\nSPD29 check_error> SQL_NO_DATA_FOUND ");
        break;
    default:
        printf("\nSPD29 check_error> Invalid rc from api rc=%i",frc);
        break;
    } /*end switch*/
    printf("\nSPD29 SQLError  ");
    while ((rc=SQLError(henv, hdbc, hstmt, cli_sqlstate, &cli_sqlcode,
      buffer,SQL_MAX_MESSAGE_LENGTH + 1, &length)) == SQL_SUCCESS) {
        printf("         SQLSTATE: %s", cli_sqlstate);
        printf("Native Error Code: %ld", cli_sqlcode);
        printf("%s ", buffer);
    };
    if (rc!=SQL_NO_DATA_FOUND)
      printf("SQLError api call failed rc=%d",rc);

    printf("\nSPD29 SQLGetSQLCA  ");
    rc = SQLGetSQLCA(henv, hdbc, hstmt, &sqlca);
    if( rc == SQL_SUCCESS )
      prt_sqlca();
    else
      printf("\n  SPD29-check_error SQLGetSQLCA failed rc=%i",rc);

    return (frc);
}
  /*******************************************************************/
  /*                  P r i n t   S Q L C A                           */
  /*******************************************************************/
SQLRETURN
  prt_sqlca()
  {
    SQLINTEGER i;
    printf("\nlSPD29 entry prts_sqlca rtn");
    printf("\r\r*** Printing the SQLCA:\r");
    printf("\nSQLCAID .... %s",sqlca.sqlcaid);
    printf("\nSQLCABC .... %d",sqlca.sqlcabc);
    printf("\nSQLCODE .... %d",sqlca.sqlcode);
    printf("\nSQLERRML ... %d",sqlca.sqlerrml);
    printf("\nSQLERRMC ... %s",sqlca.sqlerrmc);
    printf("\nSQLERRP  ... %s",sqlca.sqlerrp);
    for (i = 0; i < 6; i++)
      printf("\nSQLERRD%d ... %d",i+1,sqlca.sqlerrd??(i??));
    for (i = 0; i < 10; i++)
      printf("\nSQLWARN%d ... %c",i,sqlca.sqlwarn[i]);
    printf("\nSQLWARNA ... %c",sqlca.sqlwarn[10]);
    printf("\nSQLSTATE ... %s",sqlca.sqlstate);

    return(0);
  }                                               /* End of prtsqlca */
  /*******************************************************************/
  /*END OF SPD29 *****************************************************/
```

STEP 5. CLIENT APPLICATION

```
 /********************************************************************/
 /*START OF SPD29*****************************************************/
 /*  SCEANRIO PSEUDOCODE:                                            */
 /*  APD29(CLI CODE CLIENT APPL)                                     */
 /*       -CALL SPD29 (CLI CODE STORED PROCEDURE APPL)                */
 /*         -SPCODE=0                                                */
 /*             -PRINTF MSGS (CHECK SDSF FOR SPAS ADDR TO VERFIFY)   */
 /*         -SPCODE=1                                                */
 /*             -PRINTF MSGS (CHECK SDSF FOR SPAS ADDR TO VERFIFY)   */
 /*             -SQLPREPARE/EXECUTE INSERT INT4=200                  */
 /*             -SQLEXECDIRECT      INSERT INT4=201                  */
 /*         -SPCODE=2                                                */
 /*             -PRINTF MSGS (CHECK SDSF FOR SPAS ADDR TO VERFIFY)   */
 /*             -SQLPREPARE/EXECUTE SELECT INT4 FROM TABLE2A         */
 /*             -SQLPREPARE/EXECUTE SELECT CHR10 FROM TABLE2A        */
 /*              (CLI CURSORS OPENED 'WITH RETURN')...               */
 /*         -RETURN                                                  */
 /*       -FETCH QRS FROM SP CURSOR                                  */
 /*       -COMMIT                                                    */
 /*       -VERFIFY INSERTS BY SPD29                                  */
 /********************************************************************/
   /* Include the 'C' include files                                  */
   /******************************************************************/
   #include <stdio.h>
   #include <string.h>
   #include <stdlib.h>
   #include "sqlcli1.h"
   #include <sqlca.h>
   /******************************************************************/
   /* Variables for COMPARE routines                                 */
   /******************************************************************/
#ifndef NULL
#define NULL   0
#endif

    SQLHENV henv = SQL_NULL_HENV;
    SQLHDBC hdbc = SQL_NULL_HDBC;
    SQLHSTMT hstmt = SQL_NULL_HSTMT;
    SQLRETURN rc = SQL_SUCCESS;
    SQLINTEGER      id;
    SQLCHAR         name[51];
    SQLINTEGER      namelen, intlen, colcount;
    SQLSMALLINT     scale;
    struct sqlca    sqlca;
    SQLCHAR   server[18];
    SQLCHAR   uid[30];
    SQLCHAR   pwd[30];
    SQLCHAR   sqlstmt[250];
    SQLSMALLINT  pcpar=0;
    SQLSMALLINT  pccol=0;

   SQLINTEGER    SPCODE;
   struct {
     SQLSMALLINT LEN;
     SQLCHAR  DATA[200]; }         STMTSQL;
```

```
SQLSMALLINT                 H1SMINT;
SQLINTEGER                  H1INT4;
SQLDOUBLE               H1FLOAT8;
SQLDOUBLE               H1DEC312;
SQLCHAR                  H1CHR10[11];
SQLCHAR                  H1VCHR20[21];
SQLCHAR                  H1LVCHR[21];
SQLCHAR                  H1CHRSB[11];
SQLCHAR                  H1CHRBIT[11];
SQLCHAR                  H1DDATE[11];
SQLCHAR                  H1TTIME[9];
SQLCHAR                  H1TSTMP[27];

SQLSMALLINT                 I1SMINT;
SQLSMALLINT                 I1INT4;
SQLSMALLINT                 I1FLOAT8;
SQLSMALLINT                 I1DEC312;
SQLSMALLINT                 I1CHR10;
SQLSMALLINT                 I1VCHR20;
SQLSMALLINT                 I1LVCHR;
SQLSMALLINT                 I1CHRSB;
SQLSMALLINT                 I1CHRBIT;
SQLSMALLINT                 I1DDATE;
SQLSMALLINT                 I1TTIME;
SQLSMALLINT                 I1TSTMP;

SQLINTEGER                  LNH1SMINT;
SQLINTEGER                  LNH1INT4;
SQLINTEGER                  LNH1FLOAT8;
SQLINTEGER                  LNH1DEC312;
SQLINTEGER                  LNH1CHR10;
SQLINTEGER                  LNH1VCHR20;
SQLINTEGER                  LNH1LVCHR;
SQLINTEGER                  LNH1CHRSB;
SQLINTEGER                  LNH1CHRBIT;
SQLINTEGER                  LNH1DDATE;
SQLINTEGER                  LNH1TTIME;
SQLINTEGER                  LNH1TSTMP;

SQLSMALLINT                 H2SMINT;
SQLINTEGER                  H2INT4;
SQLDOUBLE                H2FLOAT8;
SQLCHAR                  H2CHR10[11];
SQLCHAR                  H2VCHR20[21];
SQLCHAR                  H2LVCHR[21];
SQLCHAR                  H2CHRSB[11];
SQLCHAR                  H2CHRBIT[11];
SQLCHAR                  H2DDATE[11];
SQLCHAR                  H2TTIME[9];
SQLCHAR                  H2TSTMP[27];

SQLSMALLINT                 I2SMINT;
SQLSMALLINT                 I2INT4;
SQLSMALLINT                 I2FLOAT8;
SQLSMALLINT                 I2CHR10;
SQLSMALLINT                 I2VCHR20;
SQLSMALLINT                 I2LVCHR;
SQLSMALLINT                 I2CHRSB;
SQLSMALLINT                 I2CHRBIT;
SQLSMALLINT                 I2DDATE;
SQLSMALLINT                 I2TTIME;
SQLSMALLINT                 I2TSTMP;
```

```
   SQLINTEGER                  LNH2SMINT;
   SQLINTEGER                  LNH2INT4;
   SQLINTEGER                  LNH2FLOAT8;
   SQLINTEGER                  LNH2CHR10;
   SQLINTEGER                  LNH2VCHR20;
   SQLINTEGER                  LNH2LVCHR;
   SQLINTEGER                  LNH2CHRSB;
   SQLINTEGER                  LNH2CHRBIT;
   SQLINTEGER                  LNH2DDATE;
   SQLINTEGER                  LNH2TTIME;
   SQLINTEGER                  LNH2TSTMP;

   SQLCHAR locsite[18] =  "stlec1";
   SQLCHAR remsite[18] =  "stlec1b";

   SQLINTEGER    ix,jx,locix;
   SQLINTEGER     result;
   SQLCHAR    state_blank[6] ="     ";

SQLRETURN
check_error(SQLHENV henv,
            SQLHDBC hdbc,
            SQLHSTMT hstmt,
            SQLRETURN frc);
SQLRETURN
prt_sqlca();
   /******************************************************************/
   /* Main Program                                                   */
   /******************************************************************/
SQLINTEGER
main()
{
    printf("\nAPD29 INITIALIZATION");
    scale = 0;
    rc=0;

    printf("\nAPD29 SQLAllocEnv   stmt=%d",__LINE__);
    henv=0;
    rc = SQLAllocEnv(&henv);
    if( rc != SQL_SUCCESS ) goto dberror;
    printf("\nAPD29-henv=%i",henv);

for (locix=1;locix<=2;locix++)
{
 /* Start SQL statements ******************************************/
 /*****************************************************************/
    printf("\nAPD29 SQLAllocConnect                              ");
    hdbc=0;
    SQLAllocConnect(henv, &hdbc);
    if( rc != SQL_SUCCESS ) goto dberror;
    printf("\nAPD29-hdbc=%i",hdbc);
 /*****************************************************************/
    printf("\nAPD29 SQLConnect                                   ");
    if (locix == 1)
    {
     strcpy((char *)server,(char *)locsite);
    }
    else
     {
      strcpy((char *)server,(char *)remsite);
     }
    strcpy((char *)uid,"cliuser");
    strcpy((char *)pwd,"password");
    printf("\nAPD29 server=%s",server);
    printf("\nAPD29 uid=%s",uid);
    printf("\nAPD29 pwd=%s",pwd);
    rc=SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
    if( rc != SQL_SUCCESS ) goto dberror;
 /*****************************************************************/
 /* CASE(SPCODE=0)  QRS RETURNED=0  COL=0  ROW=0                  */
 /*****************************************************************/
 printf("\nAPD29 SQLAllocStmt   stmt=%d",__LINE__);
 hstmt=0;
 rc=SQLAllocStmt(hdbc, &hstmt);
 if( rc != SQL_SUCCESS ) goto dberror;
 printf("\nAPD29-hstmt=%i\n",hstmt);

 SPCODE=0;
 printf("\nAPD29 call sp SPCODE =%i\n",SPCODE);
 printf("\nAPD29 SQLPrepare  stmt=%d",__LINE__);
 strcpy((char*)sqlstmt,"CALL SPD29(?)");
 printf("\nAPD29 sqlstmt=%s",sqlstmt);
 rc=SQLPrepare(hstmt,sqlstmt,SQL_NTS);
 if( rc != SQL_SUCCESS ) goto dberror;

 printf("\nAPD29 SQLBindParameter  stmt=%d",__LINE__);
 rc = SQLBindParameter(hstmt,
                       1,
                       SQL_PARAM_INPUT_OUTPUT,
                       SQL_C_LONG,
                       SQL_INTEGER,
                       0,
                       0,
                       &SPCODE,
                       0,
                       NULL);
 if( rc != SQL_SUCCESS ) goto dberror;

 printf("\nAPD29 SQLExecute  stmt=%d",__LINE__);
 rc=SQLExecute(hstmt);
 if( rc != SQL_SUCCESS ) goto dberror;
 if( SPCODE != 0 )
 {
   printf("\nAPD29 SPCODE not zero, spcode=%i\n",SPCODE);
   goto dberror;
 }

 printf("\nAPD29 SQLTransact stmt=%d",__LINE__);
 rc=SQLTransact(henv, hdbc, SQL_COMMIT);
 if( rc != SQL_SUCCESS ) goto dberror;

 printf("\nAPD29  SQLFreeStmt stmt=%d",__LINE__);
 rc=SQLFreeStmt(hstmt, SQL_DROP);
 if( rc != SQL_SUCCESS ) goto dberror;
 /*****************************************************************/
 /* CASE(SPCODE=1)  QRS RETURNED=0  COL=0  ROW=0                  */
 /*****************************************************************/
 printf("\nAPD29 SQLAllocStmt   stmt=%d",__LINE__);
 hstmt=0;
 rc=SQLAllocStmt(hdbc, &hstmt);
 if( rc != SQL_SUCCESS ) goto dberror;
 printf("\nAPD29-hstmt=%i\n",hstmt);
```

```
SPCODE=1;
printf("\nAPD29 call sp SPCODE =%i\n",SPCODE);
printf("\nAPD29 SQLPrepare  stmt=%d",__LINE__);
strcpy((char*)sqlstmt,"CALL SPD29(?)");
printf("\nAPD29 sqlstmt=%s",sqlstmt);
rc=SQLPrepare(hstmt,sqlstmt,SQL_NTS);
if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPD29 SQLBindParameter  stmt=%d",__LINE__);
rc = SQLBindParameter(hstmt,
                      1,
                      SQL_PARAM_INPUT_OUTPUT,
                      SQL_C_LONG,
                      SQL_INTEGER,
                      0,
                      0,
                      &SPCODE,
                      0,
                      NULL);
if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPD29 SQLExecute  stmt=%d",__LINE__);
rc=SQLExecute(hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
if( SPCODE != 0 )
{
  printf("\nAPD29 SPCODE not zero, spcode=%i\n",SPCODE);
  goto dberror;
}

printf("\nAPD29 SQLTransact stmt=%d",__LINE__);
rc=SQLTransact(henv, hdbc, SQL_COMMIT);
if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPD29  SQLFreeStmt stmt=%d",__LINE__);
rc=SQLFreeStmt(hstmt, SQL_DROP);
if( rc != SQL_SUCCESS ) goto dberror;
/*****************************************************************/
/* CASE(SPCODE=2)  QRS RETURNED=2  COL=1(int4/chr10)  ROW=100+   */
/*****************************************************************/
printf("\nAPD29 SQLAllocStmt         number=   18\n");
hstmt=0;
rc=SQLAllocStmt(hdbc, &hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
printf("\nAPD29-hstmt=%i\n",hstmt);

SPCODE=2;
printf("\nAPD29 call sp SPCODE =%i\n",SPCODE);
printf("\nAPD29 SQLPrepare           number=   19\n");
strcpy((char*)sqlstmt,"CALL SPD29(?)");
printf("\nAPD29 sqlstmt=%s",sqlstmt);
rc=SQLPrepare(hstmt,sqlstmt,SQL_NTS);
if( rc != SQL_SUCCESS ) goto dberror;
```

```
printf("\nAPD29 SQLBindParameter     number=   20\n");
rc = SQLBindParameter(hstmt,
                      1,
                      SQL_PARAM_INPUT_OUTPUT,
                      SQL_C_LONG,
                      SQL_INTEGER,
                      0,
                      0,
                      &SPCODE,
                      0,
                      NULL);
if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPD29 SQLExecute           number=   21\n");
rc=SQLExecute(hstmt);
if( rc != SQL_SUCCESS ) goto dberror;
if( SPCODE != 0 )
{
  printf("\nAPD29 spcode incorrect");
  goto dberror;
}

 printf("\nAPD29 SQLNumResultCols     number=   22\n");
 rc=SQLNumResultCols(hstmt,&pccol);
 if (pccol!=1)
 {
   printf("APD29 col count wrong=%i\n",pccol);
   goto dberror;
 }

printf("\nAPD29 SQLBindCol           number=   23\n");
rc=SQLBindCol(hstmt,
              1,
              SQL_C_LONG,
              (SQLPOINTER) &H1INT4,
              (SQLINTEGER)sizeof(SQLINTEGER),
              (SQLINTEGER *) &LNH1INT4       );
if( rc != SQL_SUCCESS ) goto dberror;

jx=0;
printf("\nAPD29 SQLFetch             number=   24\n");
while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS)
{
 jx++;
 printf("\nAPD29 fetch loop jx =%i\n",jx);
 if ( (H1INT4<=0) ||  (H1INT4>=202)
      || (LNH1INT4!=4 && LNH1INT4!=-1)  )
  { /* data error */
    printf("\nAPD29 H1INT4=%i\n",H1INT4);
    printf("\nAPD29 LNH1INT4=%i\n",LNH1INT4);
    goto dberror;
  }
 printf("\nAPD29 SQLFetch             number=   24\n");
} /* end while loop */

if( rc != SQL_NO_DATA_FOUND )
{
  printf("\nAPD29 invalid end of data\n");
  goto dberror;
}
```

```
printf("\nAPD29 SQLMoreResults       number=   25\n");
rc=SQLMoreResults(hstmt);
if(rc != SQL_SUCCESS) goto dberror;

printf("\nAPD29 SQLNumResultCols     number=   26\n");
rc=SQLNumResultCols(hstmt,&pccol);
if (pccol!=1) {
  printf("APD29 col count wrong=%i\n",pccol);
  goto dberror;
}

printf("\nAPD29 SQLBindCol           number=   27\n");
rc=SQLBindCol(hstmt,
              1,
              SQL_C_CHAR,
              (SQLPOINTER) H1CHR10,
              (SQLINTEGER)sizeof(H1CHR10),
              (SQLINTEGER *) &LNH1CHR10       );
if( rc != SQL_SUCCESS ) goto dberror;

jx=0;
while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS)
{
 jx++;
 printf("\nAPD29 fetch loop jx =%i\n",jx);
  result=strcmp((char *)H1CHR10,"CHAR      ");
  if ( (result!=0)
      || (LNH1INT4!=4 && LNH1INT4!=-1)  )
  {
    printf("\nAPD29 H1CHR10=%s\n",H1CHR10);
    printf("\nAPD29 result=%i\n",result);
    printf("\nAPD29 LNH1CHR10=%i\n",LNH1CHR10);
    printf("\nAPD29 strlen(H1CHR10)=%i\n",strlen((char *)H1CHR10));
    goto dberror;
  }
 printf("\nAPD29 SQLFetch             number=   24\n");
} /* end while loop */

if( rc != SQL_NO_DATA_FOUND )
  goto dberror;

printf("\nAPD29 SQLMoreResults       number=   29\n");
rc=SQLMoreResults(hstmt);
if( rc != SQL_NO_DATA_FOUND) goto dberror;

printf("\nAPD29 SQLTransact          number=   30\n");
rc=SQLTransact(henv, hdbc, SQL_COMMIT);
if( rc != SQL_SUCCESS ) goto dberror;

printf("\nAPD29  SQLFreeStmt         number=   31\n");
rc=SQLFreeStmt(hstmt, SQL_DROP);
if( rc != SQL_SUCCESS ) goto dberror;
/*****************************************************************/
   printf("\nAPD29 SQLDisconnect  stmt=%d",__LINE__);
   rc=SQLDisconnect(hdbc);
   if( rc != SQL_SUCCESS ) goto dberror;
/*****************************************************************/
   printf("\nSQLFreeConnect  stmt=%d",__LINE__);
   rc=SQLFreeConnect(hdbc);
   if( rc != SQL_SUCCESS ) goto dberror;
 /*****************************************************************/
 /* End SQL statements *******************************************/
```

```
} /* end for each site perform these stmts */

for (locix=1;locix<=2;locix++)
{
 /******************************************************************/
    printf("\nAPD29 SQLAllocConnect                              ");
    hdbc=0;
    SQLAllocConnect(henv, &hdbc);
    if( rc != SQL_SUCCESS ) goto dberror;
    printf("\nAPD29-hdbc=%i",hdbc);
 /******************************************************************/
    printf("\nAPD29 SQLConnect                                   ");
    if (locix == 1)
    {
     strcpy((char *)server,(char *)locsite);
    }
    else
     {
      strcpy((char *)server,(char *)remsite);
     }

    strcpy((char *)uid,"cliuser");
    strcpy((char *)pwd,"password");
    printf("\nAPD29 server=%s",server);
    printf("\nAPD29 uid=%s",uid);
    printf("\nAPD29 pwd=%s",pwd);
    rc=SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
    if( rc != SQL_SUCCESS ) goto dberror;
 /******************************************************************/
 /* Start validate SQL statements **********************************/
 /******************************************************************/
    printf("\nAPD01 SQLAllocStmt                                  \n");
    hstmt=0;
    rc=SQLAllocStmt(hdbc, &hstmt);
    if( rc != SQL_SUCCESS ) goto dberror;
    printf("\nAPD01-hstmt=%i\n",hstmt);

    printf("\nAPD01 SQLExecDirect                                 \n");
    strcpy((char *)sqlstmt,
    "SELECT INT4 FROM TABLE2A WHERE INT4=200");
    printf("\nAPD01 sqlstmt=%s",sqlstmt);
    rc=SQLExecDirect(hstmt,sqlstmt,SQL_NTS);
    if( rc != SQL_SUCCESS ) goto dberror;

    printf("\nAPD01 SQLBindCol                                    \n");
    rc=SQLBindCol(hstmt,
                  1,
                  SQL_C_LONG,
                  (SQLPOINTER) &H1INT4,;
                  (SQLINTEGER)sizeof(SQLINTEGER),
                  (SQLINTEGER *) &LNH1INT4       );
    if( rc != SQL_SUCCESS ) goto dberror;

    printf("\nAPD01 SQLFetch                                      \n");
    rc=SQLFetch(hstmt);
    if( rc != SQL_SUCCESS ) goto dberror;
    if ((H1INT4!=200) || (LNH1INT4!=4))
     {
     printf("\nAPD01 H1INT4=%i\n",H1INT4);
     printf("\nAPD01 LNH1INT4=%i\n",LNH1INT4);
     goto dberror;
     }
    printf("\nAPD01 SQLTransact                                  \n");
    rc=SQLTransact(henv, hdbc, SQL_COMMIT);
    if( rc != SQL_SUCCESS ) goto dberror;

    printf("\nAPD01  SQLFreeStmt                                 \n");
    rc=SQLFreeStmt(hstmt, SQL_CLOSE);
    if( rc != SQL_SUCCESS ) goto dberror;

    printf("\nAPD01 SQLExecDirect                                \n");
    strcpy((char *)sqlstmt,
    "SELECT INT4 FROM TABLE2A WHERE INT4=201");
    printf("\nAPD01 sqlstmt=%s",sqlstmt);
    rc=SQLExecDirect(hstmt,sqlstmt,SQL_NTS);
    if( rc != SQL_SUCCESS ) goto dberror;

    printf("\nAPD01 SQLFetch                                     \n");
    rc=SQLFetch(hstmt);
    if( rc != SQL_SUCCESS ) goto dberror;
    if ((H1INT4!=201) || (LNH1INT4!=4))
     {
     printf("\nAPD01 H1INT4=%i\n",H1INT4);
     printf("\nAPD01 LNH1INT4=%i\n",LNH1INT4);
     goto dberror;
     }

    printf("\nAPD01 SQLTransact                                  \n");
    rc=SQLTransact(henv, hdbc, SQL_COMMIT);
    if( rc != SQL_SUCCESS ) goto dberror;

    printf("\nAPD01  SQLFreeStmt                                 \n");
    rc=SQLFreeStmt(hstmt, SQL_DROP);
    if( rc != SQL_SUCCESS ) goto dberror;
 /****************************************************************/
 /* End validate SQL statements **********************************/
 /****************************************************************/
    printf("\nAPD29 SQLDisconnect  stmt=%d",__LINE__);
    rc=SQLDisconnect(hdbc);
    if( rc != SQL_SUCCESS ) goto dberror;
 /****************************************************************/
    printf("\nSQLFreeConnect  stmt=%d",__LINE__);
    rc=SQLFreeConnect(hdbc);
    if( rc != SQL_SUCCESS ) goto dberror;

} /* end for each site perform these stmts */
 /****************************************************************/
    printf("\nSQLFreeEnv  stmt=%d",__LINE__);
    rc=SQLFreeEnv(henv);
    if( rc != SQL_SUCCESS ) goto dberror;
 /****************************************************************/
 goto pgmend;

 dberror:
 printf("\nAPD29 entry dberror label");
 printf("\nAPD29 rc=%d",rc);
 check_error(henv,hdbc,hstmt,rc);
 printf("\nAPDXX SQLFreeEnv         number=     6\n");
 rc=SQLFreeEnv(henv);
 printf("\nAPDXX FREEENV rc =%d",rc);
 rc=12;
 printf("\nAPDXX DBERROR set rc =%d",rc);
 goto pgmend;
 pgmend:
     printf("\nAPD29  TERMINATION   ");
     if (rc!=0)
      {
      printf("\nAPD29 WAS NOT NOT NOT SUCCESSFUL");
      printf("\nAPD29 SPCODE  = %i", SPCODE      );
      printf("\nAPD29 rc          = %i", rc  );
      }
     else
      printf("\nAPD29 WAS SUCCESSFUL");

     return(rc);

}  /*END MAIN*/
/*******************************************************************
** check_error - call print_error(), checks severity of return code
*******************************************************************/
SQLRETURN
check_error(SQLHENV henv,
            SQLHDBC hdbc,
            SQLHSTMT hstmt,
            SQLRETURN frc )
{

   SQLCHAR         buffer_SQL_MAX_MESSAGE_LENGTH + 1•;
    SQLCHAR         cli_sqlstate_SQL_SQLSTATE_SIZE + 1•;
    SQLINTEGER     cli_sqlcode;
    SQLSMALLINT   length;

    printf("\nAPD29 entry check_error rtn");

    switch (frc) {
    case SQL_SUCCESS:
        break;
    case SQL_INVALID_HANDLE:
        printf("\nAPD29 check_error> SQL_INVALID HANDLE ");
    case SQL_ERROR:
        printf("\nAPD29 check_error> SQL_ERROR ");
        break;
    case SQL_SUCCESS_WITH_INFO:
        printf("\nAPD29 check_error>  SQL_SUCCESS_WITH_INFO");
        break;
    case SQL_NO_DATA_FOUND:
        printf("\nAPD29 check_error> SQL_NO_DATA_FOUND ");
        break;
    default:
        printf("\nAPD29 check_error> Invalid rc from api rc=%i",frc);
        break;
    } /*end switch*/

    printf("\nAPD29 SQLError  ");
    while ((rc=SQLError(henv, hdbc, hstmt, cli_sqlstate, &cli_sqlcode,
      buffer,SQL_MAX_MESSAGE_LENGTH + 1, &length)) == SQL_SUCCESS) {
        printf("        SQLSTATE: %s", cli_sqlstate);
        printf("Native Error Code: %ld", cli_sqlcode);
        printf("%s ", buffer);
    };
    if (rc!=SQL_NO_DATA_FOUND)
      printf("SQLError api call failed rc=%d",rc);
    printf("\nAPD29 SQLGetSQLCA  ");
    rc = SQLGetSQLCA(henv, hdbc, hstmt, &sqlca);
    if( rc == SQL_SUCCESS )
      prt_sqlca();
    else
      printf("\n  APD29-check_error SQLGetSQLCA failed rc=%i",rc);

    return (frc);
}
  /*******************************************************************/
  /*                  P r i n t   S Q L C A                          */
  /*******************************************************************/
SQLRETURN
  prt_sqlca()
  {
    SQLINTEGER i;
    printf("\nlAPD29 entry prts_sqlca rtn");
    printf("\r\r*** Printing the SQLCA:\r");
    printf("\nSQLCAID .... %s",sqlca.sqlcaid);
    printf("\nSQLCABC .... %d",sqlca.sqlcabc);
    printf("\nSQLCODE .... %d",sqlca.sqlcode);
    printf("\nSQLERRML ... %d",sqlca.sqlerrml);
    printf("\nSQLERRMC ... %s",sqlca.sqlerrmc);
    printf("\nSQLERRP  ... %s",sqlca.sqlerrp);
    for (i = 0; i < 6; i++)
      printf("\nSQLERRD%d ... %d",i+1,sqlca.sqlerrd??(i??));
    for (i = 0; i < 10; i++)
      printf("\nSQLWARN%d ... %c",i,sqlca.sqlwarn[i]);
    printf("\nSQLWARNA ... %c",sqlca.sqlwarn[10]);
    printf("\nSQLSTATE ... %s",sqlca.sqlstate);

    return(0);
  }                                               /* End of prtsqlca */
  /*END OF APD29******************************************************/
```

Glossary

The following terms and abbreviations are defined as they are used in the DB2 library. If you do not find the term you are looking for, refer to the index or to *Dictionary of Computing*.

A

address space. A range of virtual storage pages identified by a number (ASID) and a collection of segment and page tables which map the virtual pages to real pages of the computer's memory.

address space connection. The result of connecting an allied address space to DB2. Each address space containing a task connected to DB2 has exactly one address space connection, even though more than one task control block (TCB) can be present. See *allied address space* and *task control block*.

allied address space. An area of storage external to DB2 that is connected to DB2 and is therefore capable of requesting DB2 services.

American National Standards Institute (ANSI). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

ANSI. American National Standards Institute.

API. Application programming interface.

application. A program or set of programs that perform a task; for example, a payroll application.

application plan. The control structure produced during the bind process and used by DB2 to process SQL statements encountered during statement execution.

application program interface (API). A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or licensed program.

application requester (AR). See *requester*.

AR. application requester. See *requester*.

ASCII. An encoding scheme used to represent strings in many environments, typically on PCs and workstations. Contrast with *EBCDIC*.

attachment facility. An interface between DB2 and TSO, IMS, CICS, or batch address spaces. An attachment facility allows application programs to access DB2.

authorization ID. A string that can be verified for connection to DB2 and to which a set of privileges are allowed. It can represent an individual, an organizational group, or a function, but DB2 does not determine this representation.

B

base table. A table created by the SQL CREATE TABLE statement that is used to hold persistent data. Contrast with *result table* and *temporary table*.

bind. The process by which the output from the DB2 precompiler is converted to a usable control structure called a package or an application plan. During the process, access paths to the data are selected and some authorization checking is performed.

> **automatic bind**. (More correctly *automatic rebind*). A process by which SQL statements are bound automatically (without a user issuing a BIND command) when an application process begins execution and the bound application plan or package it requires is not valid.
> **dynamic bind**. A process by which SQL statements are bound as they are entered.
> **incremental bind**. A process by which SQL statements are bound during the execution of an application process, because they could not be bound during the bind process, and VALIDATE(RUN) was specified.
> **static bind**. A process by which SQL statements are bound after they have been precompiled. All static SQL statements are prepared for execution at the same time. Contrast with *dynamic bind*.

built-in function. Scalar function or column function.

C

CAF. Call attachment facility.

call attachment facility (CAF). A DB2 attachment facility for application programs running in TSO or MVS batch. The CAF is an alternative to the DSN command processor and allows greater control over the execution environment.

call level interface (CLI). A callable application program interface (API) for database access, which is an alternative to using embedded SQL. In contrast to embedded SQL, DB2 CLI does not require the user to precompile or bind applications, but instead provides a standard set of functions to process SQL statements and related services at run time.

catalog. In DB2, a collection of tables that contains descriptions of objects such as tables, views, and indexes.

catalog table. Any table in the DB2 catalog.

CLI. See *call level interface*.

client. See *requester*.

column function. An SQL operation that derives its result from a collection of values across one or more rows. Contrast with *scalar function*.

commit. The operation that ends a unit of work by releasing locks so that the database changes made by that unit of work can be perceived by other processes.

common server. Describes the set of DB2 products that run on various platforms and have the same source code. These platforms include OS/2, Windows, and UNIX.

connection handle. The data object that contains information associated with a connection managed by DB2 CLI. This includes general status information, transaction status, and diagnostic information.

constant. A language element that specifies an unchanging value. Constants are classified as string constants or numeric constants. Contrast with *variable*.

context. The application's logical connection to the data source and associated internal DB2 CLI connection information that allows the application to direct its operations to a data source. A DB2 CLI context represents a DB2 thread.

cursor. A named control structure used by an application program to point to a row of interest within some set of rows, and to retrieve rows from the set, possibly making updates or deletions.

D

database management system (DBMS). A software system that controls the creation, organization, and modification of a database and access to the data stored within it.

DBMS. Database management system.

DB2 thread. The DB2 structure that describes an application's connection, traces its progress, processes resource functions, and delmits its accessibility to DB2 resources. and services.

distributed relational database architecture (DRDA). A connection protocol for distributed relational database processing that is used by IBM's relational database products. DRDA includes protocols for communication between an application and a remote relational database management system, and for communication between relational database management systems.

DRDA. Distributed relational database architecture.

dynamic SQL. SQL statements that are prepared and executed within an application program while the program is executing. In dynamic SQL, the SQL source is contained in host language variables rather than being coded into the application program. The SQL statement can change several times during the application program's execution.

E

EBCDIC. Extended binary coded decimal interchange code. An encoding scheme used to represent character data in the MVS, VM, VSE, and OS/400 environments. Contrast with *ASCII*.

embedded SQL. SQL statements coded within an application program. See *static SQL*.

environment. A collection of names of logical and physical resources that are used to support the performance of a function.

environment handle. In DB2 CLI, the data object that contains global information regarding the state of the application. An environment handle must be allocated before a connection handle can be allocated. Only one environment handle can be allocated per application.

equi-join. A join operation in which the join-condition has the form *expression* = *expression*.

F

foreign key. A key that is specified in the definition of a referential constraint. Because of the foreign key, the table is a dependent table. The key must have the same number of columns, with the same descriptions, as the primary key of the parent table.

full outer join. The result of a join operation that includes the matched rows of both tables being joined and preserves the unmatched rows of both tables. See also *join*.

function. A scalar function or column function. Same as *built-in function*.

H

handle. In DB2 CLI, a variable that refers to a data structure and associated resources. See *statement handle*, *connection handle*, and *environment handle*.

I

initialization file. For DB2 CLI applications, a file containing values that can be set to adjust the performance of the database manager.

inner join. The result of a join operation that includes only the matched rows of both tables being joined. See also *join*.

J

JCL. Job control language.

join. A relational operation that allows retrieval of data from two or more tables based on matching column values. See also *full outer join, inner join, left outer join, outer join, right outer join, equi-join*.

L

left outer join. The result of a join operation that includes the matched rows of both tables being joined, and preserves the unmatched rows of the first table. See also *join*.

link-edit. To create a loadable computer program using a linkage editor.

load module. A program unit that is suitable for loading into main storage for execution. The output of a linkage editor.

local. Refers to any object maintained by the local DB2 subsystem. A *local table*, for example, is a table maintained by the local DB2 subsystem. Contrast with *remote*.

M

multithreading. Multiple TCBs executing one copy of DB2 CLI code concurrently (sharing a processor) or in parallel (on separate central processors).

mutex. Pthread mutual exclusion; a lock. A Pthread mutex variable is used as a locking mechanism to allow serialization of critical sections of code by temporarily blocking the execution of all but one thread.

MVS/ESA. Multiple Virtual Storage/Enterprise Systems Architecture.

N

NUL. In C, a single character that denotes the end of the string.

null. A special value that indicates the absence of information.

NUL-terminated host variable. A varying-length host variable in which the end of the data is indicated by the presence of a NUL terminator.

NUL terminator. In C, the value that indicates the end of a string. For character strings, the NUL terminator is `X'00'`.

O

ODBC. See *Open Database Connectivity*.

ODBC driver. A dynamically-linked library (DLL) that implements ODBC function calls and interacts with a data source.

Open Database Connectivity (ODBC). A Microsoft database application programming interface (API) for C that allows access to database management systems by using callable SQL. ODBC does not require the use of an SQL preprocessor. In addition, ODBC provides an architecture that lets users add modules called *database drivers* that link the application to their choice of database management systems at run time. This means that applications no longer need to be directly linked to the modules of all the database management systems that are supported.

outer join. The result of a join operation that includes the matched rows of both tables being joined and preserves some or all of the unmatched rows of the tables being joined. See also *join*.

P

plan. See *application plan*.

plan name. The name of an application plan.

POSIX. Portable Operating System Interface. The IEEE operating system interface standard which defines the Pthread standard of threading. See *Pthread*.

precompilation. A processing of application programs containing SQL statements that takes place before compilation. SQL statements are replaced with statements that are recognized by the host language compiler. Output from this precompilation includes source code that can be submitted to the compiler and the database request module (DBRM) that is input to the bind process.

prepare. The first phase of a two-phase commit process in which all participants are requested to prepare for commit.

prepared SQL statement. A named object that is the executable form of an SQL statement that has been processed by the PREPARE statement.

primary key. A unique, nonnull key that is part of the definition of a table. A table cannot be defined as a parent unless it has a unique key or primary key.

Pthread. The POSIX threading standard model for splitting an application into subtasks. The Pthread standard includes functions for creating threads, terminating threads, synchronizing threads through locking, and other thread control facilities.

R

RDBMS. Relational database management system.

reentrant. Executable code that can reside in storage as one shared copy for all threads. Reentrant code is not self-modifying and provides separate storage areas for each thread. Reentrancy is a compiler and operating system concept, and reentrancy alone is not enough to guarantee logically consistent results when multithreading. See *threadsafe*.

relational database management system (RDBMS). A relational database manager that operates consistently across supported IBM systems.

remote. Refers to any object maintained by a remote DB2 subsystem; that is, by a DB2 subsystem other than the local one. A *remote view*, for instance, is a view maintained by a remote DB2 subsystem. Contrast with *local*.

requester. Also *application requester (AR)*. The source of a request to a remote RDBMS, the system that requests the data.

result set. The set of rows returned to a client application by a stored procedure.

result set locator. A 4-byte value used by DB2 to uniquely identify a query result set returned by a stored procedure.

result table. The set of rows specified by a SELECT statement.

right outer join. The result of a join operation that includes the matched rows of both tables being joined and preserves the unmatched rows of the second join operand. See also *join*.

rollback. The process of restoring data changed by SQL statements to the state at its last commit point. All locks are freed. Contrast with *commit*.

S

scalar function. An SQL operation that produces a single value from another value and is expressed as a function name followed by a list of arguments enclosed in parentheses. See also *column function*.

SQL. Structured Query Language.

SQL authorization ID (SQL ID). The authorization ID that is used for checking dynamic SQL statements in some situations.

SQL Communication Area (SQLCA). A structure used to provide an application program with information about the execution of its SQL statements.

SQL Descriptor Area (SQLDA). A structure that describes input variables, output variables, or the columns of a result table.

SQLCA. SQL communication area.

SQLDA. SQL descriptor area.

SQL/DS. SQL/Data System. Also known as *DB2/VSE & VM*.

statement handle. In DB2 CLI, the data object that contains information about an SQL statement that is managed by DB2 CLI. This includes information such as dynamic arguments, bindings for dynamic arguments and columns, cursor information, result values and status information. Each statement handle is associated with the connection handle.

static SQL. SQL statements, embedded within a program, that are prepared during the program preparation process (before the program is executed). After being prepared, the SQL statement does not change (although values of host variables specified by the statement might change).

stored procedure. A user-written application program, that can be invoked through the use of the SQL CALL statement.

Structured Query Language (SQL). A standardized language for defining and manipulating data in a relational database.

T

table. A named data object consisting of a specific number of columns and some number of unordered rows. Synonymous with *base table* or *temporary table*.

task control block (TCB). A control block used to communicate information about tasks within an address space that are connected to DB2. An address space can support many task connections (as many as one per task), but only one address space connection. See *address space connection*.

TCB. MVS task control block.

TCP/IP. A network communication protocol used by computer systems to exchange information across telecommunication links.

temporary table. A table created by the SQL CREATE GLOBAL TEMPORARY TABLE statement that is used to hold temporary data. Contrast with *result table* and *temporary table*.

threadsafe. Characteristic of code that allows multithreading both by providing private storage areas for each thread, and by properly serializing shared (global) storage areas.

timestamp. A seven-part value that consists of a date and time expressed in years, months, days, hours, minutes, seconds, and microseconds.

trace. A DB2 facility that provides the ability to monitor and collect DB2 monitoring, auditing, performance, accounting, statistics, and serviceability (global) data.

V

variable. A data element that specifies a value that can be changed. A COBOL elementary data item is an example of a variable. Contrast with *constant*.

X

X/Open. An independent, worldwide open systems organization that is supported by most of the world's largest information systems suppliers, user organizations, and software companies. X/Open's goal is to increase the portability of applications by combining existing and emerging standards.

Bibliography

DB2 for OS/390 Version 5

- *Administration Guide, SC26-8957*
- *Application Programming and SQL Guide, SC26-8958*
- *Call Level Interface Guide and Reference, SC26-8959*
- *Command Reference, SC26-8960*
- *Data Sharing: Planning and Administration, SC26-8961*
- *Data Sharing Quick Reference Card, SX26-3841*
- *Diagnosis Guide and Reference, LY27-9659*
- *Diagnostic Quick Reference Card, LY27-9660*
- *Installation Guide, GC26-8970*
- *Application Programming Guide and Reference for Java™, SC26-9547*
- *Licensed Program Specifications, GC26-8969*
- *Messages and Codes, GC26-8979*
- *Reference for Remote DRDA Requesters and Servers, SC26-8964*
- *Reference Summary, SX26-3842*
- *Release Guide, SC26-8965*
- *SQL Reference, SC26-8966*
- *Utility Guide and Reference, SC26-8967*
- *What' s New?, GC26-8971*
- *Program Directory*

DB2 PM for OS/390 Version 5

- *Batch User's Guide, SC26-8991*
- *Command Reference, SC26-8987*
- *General Information, GC26-8982*
- *Getting Started on the Workstation, SC26-8989*
- *Master Index, SC26-8984*
- *Messages Manual, SC26-8988*
- *Online Monitor User's Guide, SC26-8990*
- *Report Reference Volume 1, SC26-8985*
- *Report Reference Volume 2, SC26-8986*
- *Program Directory*

Ada/370

- *IBM Ada/370 Language Reference, SC09-1297*
- *IBM Ada/370 Programmer' s Guide, SC09-1414*
- *IBM Ada/370 SQL Module Processor for DB2 Database Manager User' s Guide, SC09-1450*

APL2

- *APL2 Programming Guide, SH21-1072*
- *APL2 Programming: Language Reference, SH21-1061*
- *APL2 Programming: Using Structured Query Language (SQL), SH21-1057*

AS/400

- *DB2 for OS/400 SQL Programming, SC41-4611*
- *DB2 for OS/400 SQL Reference, SC41-4612*

BASIC

- *IBM BASIC/MVS Language Reference, GC26-4026*
- *IBM BASIC/MVS Programming Guide, SC26-4027*

C/370

- *IBM SAA AD/Cycle C/370 Programming Guide, SC09-1356*
- *IBM SAA AD/Cycle C/370 Programming Guide for Language Environment/370, SC09-1840*
- *IBM SAA AD/Cycle C/370 User' s Guide, SC09-1763*
- *SAA CPI C Reference, SC09-1308*

Character Data Representation Architecture

- *Character Data Representation Architecture Overview, GC09-2207*
- *Character Data Representation Architecture Reference, SC09-2190*

CICS/ESA

- *CICS/ESA Application Programming Guide, SC33-1169*
- *CICS/ESA Application Programming Reference, SC33-1170*
- *CICS/ESA CICS - RACF Security Guide, SC33-1185*
- *CICS/ESA CICS-Supplied Transactions, SC33-1168*
- *CICS/ESA Customization Guide, SC33-1165*
- *CICS/ESA Data Areas, LY33-6083*
- *CICS/ESA Installation Guide, SC33-1163*
- *CICS/ESA Intercommunication Guide, SC33-1181*
- *CICS/ESA Messages and Codes, SC33-1177*
- *CICS/ESA Operations and Utilities Guide, SC33-1167*
- *CICS/ESA Performance Guide, SC33-1183*
- *CICS/ESA Problem Determination Guide, SC33-1176*
- *CICS/ESA Resource Definition Guide, SC33-1166*
- *CICS/ESA System Definition Guide, SC33-1164*
- *CICS/ESA System Programming Reference, GC33-1171*

CICS/MVS

- *CICS/MVS Application Programming Primer, SC33-0139*
- *CICS/MVS Application Programmer's Reference, SC33-0512*
- *CICS/MVS Facilities and Planning Guide, SC33-0504*
- *CICS/MVS Installation Guide, SC33-0506*
- *CICS/MVS Operations Guide, SC33-0510*
- *CICS/MVS Problem Determination Guide, SC33-0516*
- *CICS/MVS Resource Definition (Macro), SC33-0509*
- *CICS/MVS Resource Definition (Online), SC33-0508*

IBM C/C++ for MVS/ESA or OS/390

- *IBM C/C++ for MVS/ESA Library Reference, SC09-1995*
- *IBM C/C++ for MVS/ESA Programming Guide, SC09-1994*
- *IBM C/C++ for OS/390 User's Guide, SC09-2361*

IBM COBOL for MVS & VM

- *IBM COBOL for MVS & VM Language Reference, SC26-4769*
- *IBM COBOL for MVS & VM Programming Guide, SC26-4767*

Conversion Guides

- *DBMS Conversion Guide: DATACOM/DB to DB2, GH20-7564*
- *DBMS Conversion Guide: IDMS to DB2, GH20-7562*
- *DBMS Conversion Guide: Model 204 to DB2 or SQL/DS, GH20-7565*
- *DBMS Conversion Guide: VSAM to DB2, GH20-7566*
- *IMS-DB and DB2 Migration and Coexistence Guide, GH21-1083*

Cooperative Development Environment

- *CoOperative Development Environment/370: Debug Tool, SC09-1623*

DATABASE 2 for Common Servers

- *DATABASE 2 Administration Guide for common servers, S20H-4580*
- *DATABASE 2 Application Programming Guide for common servers, S20H-4643*
- *DATABASE 2 Software Developer's Kit for AIX: Building Your Applications, S20H-4780*
- *DATABASE 2 Software Developer's Kit for OS/2: Building Your Applications, S20H-4787*
- *DATABASE 2 SQL Reference for common servers, S20H-4665*
- *DATABASE 2 Call Level Interface Guide and Reference for common servers, S20H-4644*

Data Extract (DXT)

- *Data Extract Version 2: General Information, GC26-4666*
- *Data Extract Version 2: Planning and Administration Guide, SC26-4631*

DataPropagator NonRelational

- *DataPropagator NonRelational MVS/ESA Administration Guide, SH19-5036*
- *DataPropagator NonRelational MVS/ESA Reference, SH19-5039*

DataPropagator Relational

- *DataPropagator Relational User's Guide, SC26-3399*
- *IBM An Introduction to DataPropagator Relational, GC26-3398*

Data Facility Data Set Services

- *Data Facility Data Set Services: User's Guide and Reference, SC26-4125*

Database Design

- *DB2 Database Design and Implementation Using DB2, SH24-6101*
- *DB2 Design and Development Guide, Gabrielle Wiorkowski and David Kull, Addison Wesley*
- *Handbook of Relational Database Design, C. Fleming and B Von Halle, Addison Wesley*
- *Principles of Database Systems, Jeffrey D. Ullman, Computer Science Press*

DataHub

- *IBM DataHub General Information, GC26-4874*

DB2 Universal Database

- *DB2 Universal Database Administration Guide, S10J-8157*
- *DB2 Universal Database API Reference, S10J-8167*
- *DB2 Universal Database Building Applications for UNIX Environments, S10J-8161*
- *DB2 Universal Database Building Applications for Windows and OS/2 Environments, S10J-8160*
- *DB2 Universal Database CLI Guide and Reference, S10J-8159*
- *DB2 Universal Database SQL Reference, S10J-8165*

Device Support Facilities

- *Device Support Facilities User's Guide and Reference, GC35-0033*

DFSMS/MVS

- *DFSMS/MVS: Access Method Services for the Integrated Catalog, SC26-4906*

- *DFSMS/MVS: Access Method Services for VSAM Catalogs, SC26-4905*
- *DFSMS/MVS: Administration Reference for DFSMSdss, SC26-4929*
- *DFSMS/MVS: DFSMShsm Managing Your Own Data, SH21-1077*
- *DFSMS/MVS: Diagnosis Reference for DFSMSdfp, LY27-9606*
- *DFSMS/MVS: Macro Instructions for Data Sets, SC26-4913*
- *DFSMS/MVS: Managing Catalogs, SC26-4914*
- *DFSMS/MVS: Program Management, SC26-4916*
- *DFSMS/MVS: Storage Administration Reference for DFSMSdfp, SC26-4920*
- *DFSMS/MVS: Using Advanced Services for Data Sets, SC26-4921*
- *DFSMS/MVS: Utilities, SC26-4926*
- *MVS/DFP: Managing Non-VSAM Data Sets, SC26-4557*

DFSORT

- *DFSORT Application Programming: Guide, SC33-4035*

Distributed Relational Database

- *Data Stream and OPA Reference, SC31-6806*
- *Distributed Relational Database Architecture: Application Programming Guide, SC26-4773*
- *Distributed Relational Database Architecture: Connectivity Guide, SC26-4783*
- *Distributed Relational Database Architecture: Evaluation and Planning Guide, SC26-4650*
- *Distributed Relational Database Architecture: Problem Determination Guide, SC26-4782*
- *Distributed Relational Database: Every Manager's Guide, GC26-3195*
- *IBM SQL Reference, SC26-8416*
- *Open Group Technical Standard (the Open Group presently makes the following books available through their website at www.opengroup.org):*
  - *DRDA Volume 1: Distributed Relational Database Architecture (DRDA), ISBN 1-85912-295-7*
  - *DRDA Volume 3: Distributed Database Management (DDM) Architecture, ISBN 1-85912-206-X*

Education

- *Dictionary of Computing, SC20-1699*
- *IBM Enterprise Systems Training Solutions Catalog, GR28-5467*

Enterprise System/9000 and Enterprise System/3090

- *Enterprise System/9000 and Enterprise System/3090 Processor Resource/System Manager Planning Guide, GA22-7123*

FORTRAN

- *VS FORTRAN Version 2: Language and Library Reference, SC26-4221*
- *VS FORTRAN Version 2: Programming Guide for CMS and MVS, SC26-4222*

High Level Assembler

- *High Level Assembler/MVS and VM and VSE Language Reference, SC26-4940*
- *High Level Assembler/MVS and VM and VSE Programmer's Guide, SC26-4941*

Parallel Sysplex Library

- *System/390 MVS Sysplex Application Migration, GC28-1211*
- *System/390 MVS Sysplex Hardware and Software Migration, GC28-1210*
- *System/390 MVS Sysplex Overview: An Introduction to Data Sharing and Parallelism, GC28-1208*
- *System/390 MVS Sysplex Systems Management, GC28-1209*
- *System/390 MVS 9672/9674 System Overview, GA22-7148*

ICSF/MVS

- *ICSF/MVS General Information, GC23-0093*

IMS/ESA

- *IMS Batch Terminal Simulator General Information, GH20-5522*
- *IMS/ESA Administration Guide: System, SC26-8013*
- *IMS/ESA Application Programming: Database Manager, SC26-8727*
- *IMS/ESA Application Programming: Design Guide, SC26-8016*
- *IMS/ESA Application Programming: Transaction Manager, SC26-8729*
- *IMS/ESA Customization Guide, SC26-8020*
- *IMS/ESA Installation Volume 1: Installation and Verification, SC26-8023*
- *IMS/ESA Installation Volume 2: System Definition and Tailoring, SC26-8024*
- *IMS/ESA Messages and Codes, SC26-8028*
- *IMS/ESA Operator's Reference, SC26-8030*
- *IMS/ESA Utilities Reference: System, SC26-8035*

ISPF

- *ISPF Version 4 Messages and Codes, SC34-4450*
- *ISPF Version 4 for MVS Dialog Management Guide, SC34-4213*
- *ISPF/PDF Version 4 for MVS Guide and Reference, SC34-4258*
- *ISPF and ISPF/PDF Version 4 for MVS Planning and Customization, SC34-4134*

Language Environment for MVS & VM

- *Language Environment for MVS & VM Concepts Guide, GC26-4786*
- *Language Environment for MVS & VM Debugging and Run-Time Messages Guide, SC26-4829*
- *Language Environment for MVS & VM Installation and Customization, SC26-4817*
- *Language Environment for MVS & VM Programming Guide, SC26-4818*
- *Language Environment for MVS & VM Programming Reference, SC26-3312*

MVS/ESA

- *MVS/ESA Analyzing Resource Measurement Facility Monitor I and Monitor II Reference and User' s Guide, LY28-1007*
- *MVS/ESA Analyzing Resource Measurement Facility Monitor III Reference and User' s Guide, LY28-1008*
- *MVS/ESA Application Development Reference: Assembler Callable Services for OpenEdition MVS, SC23-3020*
- *MVS/ESA Data Administration: Utilities, SC26-4516*
- *MVS/ESA Diagnosis: Procedures, LY28-1844*
- *MVS/ESA Diagnosis: Tools and Service Aids, LY28-1845*
- *MVS/ESA Initialization and Tuning Guide, SC28-1451*
- *MVS/ESA Initialization and Tuning Reference, SC28-1452*
- *MVS/ESA Installation Exits, SC28-1459*
- *MVS/ESA JCL Reference, GC28-1479*
- *MVS/ESA JCL User' s Guide, GC28-1473*
- *MVS/ESA JES2 Initialization and Tuning Guide, SC28-1453*
- *MVS/ESA MVS Configuration Program, GC28-1615*
- *MVS/ESA Planning: Global Resource Serialization, GC28-1450*
- *MVS/ESA Planning: Operations, GC28-1441*
- *MVS/ESA Planning: Workload Management, GC28-1493*
- *MVS/ESA Programming: Assembler Services Guide, GC28-1466*
- *MVS/ESA Programming: Assembler Services Reference, GC28-1474*
- *MVS/ESA Programming: Authorized Assembler Services Guide, GC28-1467*
- *MVS/ESA Programming: Authorized Assembler Services Reference, Volumes 1-4, GC28-1475, GC28-1476, GC28-1477, GC28-1478*
- *MVS/ESA Programming: Extended Addressability Guide, GC28-1468*
- *MVS/ESA Programming: Sysplex Services Guide, GC28-1495*
- *MVS/ESA Programming: Sysplex Services Reference, GC28-1496*
- *MVS/ESA Programming: Workload Management Services, GC28-1494*
- *MVS/ESA Routing and Descriptor Codes, GC28-1487*
- *MVS/ESA Setting Up a Sysplex, GC28-1449*
- *MVS/ESA SPL: Application Development Guide, GC28-1852*
- *MVS/ESA System Codes, GC28-1486*
- *MVS/ESA System Commands, GC28-1442*
- *MVS/ESA System Management Facilities (SMF), GC28-1457*
- *MVS/ESA System Messages Volume 1, GC28-1480*
- *MVS/ESA System Messages Volume 2, GC28-1481*
- *MVS/ESA System Messages Volume 3, GC28-1482*
- *MVS/ESA Using the Subsystem Interface, SC28-1502*

Net.Data for OS/390

- *Net.Data Language Environment Guide, http://www.ibm.com/software/net.data/docs* # #
- *Net.Data Programming Guide, http://www.ibm.com/software/net.data/docs* # #
- *Net.Data Reference Guide, http://www.ibm.com/software/net.data/docs* # #

NetView

- *NetView Installation and Administration Guide, SC31-8043*
- *NetView User' s Guide, SC31-8056*

ODBC

- *ODBC 2.0 Programmer' s Reference and SDK Guide, ISBN 1-55615-658-8*
- *Inside ODBC, ISBN 1-55615-815-7*

OS/390

- *OS/390 C/C++ Programming Guide, SC09-2362*
- *OS/390 C/C++ Run-Time Library Reference, SC28-1663*
- *OS/390 Information Roadmap, GC28-1727*
- *OS/390 Introduction and Release Guide, GC28-1725*
- *OS/390 JES2 Initialization and Tuning Guide, SC28-1791*
- *OS/390 JES3 Initialization and Tuning Guide, SC28-1802*
- *OS/390 Language Environment for OS/390 & VM Concepts Guide, GC28-1945*
- *OS/390 Language Environment for OS/390 & VM Customization, SC28-1941*
- *OS/390 Language Environment for OS/390 & VM Debugging Guide, SC28-1942*
- *OS/390 Language Environment for OS/390 & VM Programming Guide, SC28-1939*
- *OS/390 Language Environment for OS/390 & VM Programming Reference, SC28-1940*
- *OS/390 MVS Diagnosis: Procedures, LY28-1082*
- *OS/390 MVS Diagnosis: Tools and Service Aids, LY28-1085*

- *OS/390 MVS Initialization and Tuning Guide, SC28-1751*
- *OS/390 MVS Initialization and Tuning Reference, SC28-1752*
- *OS/390 MVS Installation Exits, SC28-1753*
- *OS/390 MVS JCL Reference, GC28-1757*
- *OS/390 MVS JCL User's Guide, GC28-1758*
- *OS/390 MVS Planning: Global Resource Serialization, GC28-1759*
- *OS/390 MVS Planning: Operations, GC28-1760*
- *OS/390 MVS Planning: Workload Management, GC28-1761*
- *OS/390 MVS Programming: Assembler Services Guide, GC28-1762*
- *OS/390 MVS Programming: Assembler Services Reference, GC28-1910*
- *OS/390 MVS Programming: Authorized Assembler Services Guide, GC28-1763*
- *OS/390 MVS Programming: Authorized Assembler Services Reference, Volumes 1-4, GC28-1764, GC28-1765, GC28-1766, GC28-1767*
- *OS/390 MVS Programming: Callable Services for High-Level Languages, GC28-1768*
- *OS/390 MVS Programming: Extended Addressability Guide, GC28-1769*
- *OS/390 MVS Programming: Sysplex Services Guide, GC28-1771*
- *OS/390 MVS Programming: Sysplex Services Reference, GC28-1772*
- *OS/390 MVS Programming: Workload Management Services, GC28-1773*
- *OS/390 MVS Routing and Descriptor Codes, GC28-1778*
- *OS/390 MVS Setting Up a Sysplex, GC28-1779*
- *OS/390 MVS System Codes, GC28-1780*
- *OS/390 MVS System Commands, GC28-1781*
- *OS/390 MVS System Messages Volume 1, GC28-1784*
- *OS/390 MVS System Messages Volume 2, GC28-1785*
- *OS/390 MVS System Messages Volume 3, GC28-1786*
- *OS/390 MVS System Messages Volume 4, GC28-1787*
- *OS/390 MVS System Messages Volume 5, GC28-1788*
- *OS/390 Security Server (RACF) Auditor's Guide, SC28-1916*
- *OS/390 Security Server (RACF) Command Language Reference, SC28-1919*
- *OS/390 Security Server (RACF) General User's Guide, SC28-1917*
- *OS/390 Security Server (RACF) Security Administrator's Guide, SC28-1915*
- *OS/390 Security Server (RACF) System Programmer's Guide, SC28-1913*
- *OS/390 SMP/E Reference, SC28-1806*
- *OS/390 SMP/E User's Guide, SC28-1740*
- *OS/390 RMF User's Guide, SC28-1949*
- *OS/390 TSO/E CLISTS, SC28-1973*
- *OS/390 TSO/E Command Reference, SC28-1969*
- *OS/390 TSO/E Customization, SC28-1965*
- *OS/390 TSO/E Messages, GC28-1978*
- *OS/390 TSO/E Programming Guide, SC28-1970*
- *OS/390 TSO/E Programming Services, SC28-1971*
- *OS/390 TSO/E REXX Reference, SC28-1975*
- *OS/390 TSO/E User's Guide, SC28-1968*

OS/390 OpenEdition

- *OS/390 OpenEdition DCE Administration Guide, SC28-1584*
- *OS/390 OpenEdition DCE Introduction, GC28-1581*
- *OS/390 R1 OE DCE Messages and Codes, ST01-0920*
- *OS/390 OpenEdition Command Reference, SC28-1892*
- *OS/390 OpenEdition Messages and Codes, SC28-1908*
- *OS/390 OpenEdition Planning, SC28-1890*
- *OS/390 OpenEdition User's Guide, SC28-1891*

PL/I for MVS & VM

- *IBM PL/I MVS & VM Language Reference, SC26-3114*
- *IBM PL/I MVS & VM Programming Guide, SC26-3113*

OS PL/I

- *OS PL/I Programming Language Reference, SC26-4308*
- *OS PL/I Programming Guide, SC26-4307*

PROLOG

- *IBM SAA AD/Cycle Prolog/MVS & VM Programmer's Guide, SH19-6892*

Query Management Facility

- *Query Management Facility: Managing QMF for MVS, SC26-8218*
- *Query Management Facility: Reference, SC26-4716*
- *Query Management Facility: Using QMF, SC26-8078*

Remote Recovery Data Facility

- *Remote Recovery Data Facility Program Description and Operations, LY37-3710*

Resource Access Control Facility (RACF)

- *External Security Interface (RACROUTE) Macro Reference for MVS and VM, GC28-1366*
- *Resource Access Control Facility (RACF) Auditor's Guide, SC28-1342*
- *Resource Access Control Facility (RACF) Command Language Reference, SC28-0733*
- *Resource Access Control Facility (RACF) General Information Manual, GC28-0722*

- *Resource Access Control Facility (RACF) General User' s Guide, SC28-1341*
- *Resource Access Control Facility (RACF) Security Administrator' s Guide, SC28-1340*
- *Recource Access Control Facility (RACF) System Programmer' s Guide, SC28-1343*

Storage Management

- *MVS/ESA Storage Management Library: Implementing System-Managed Storage, SC26-3123*
- *MVS/ESA Storage Management Library: Leading an Effective Storage Administration Group, SC26-3126*
- *MVS/ESA Storage Management Library: Managing Data, SC26-3124*
- *MVS/ESA Storage Management Library: Managing Storage Groups, SC26-3125*
- *MVS Storage Management Library: Storage Management Subsystem Migration Planning Guide, SC26-4659*

System/370 and System/390

- *IBM System/370 ESA Principles of Operation, SA22-7200*
- *IBM System/390 ESA Principles of Operation, SA22-7205*
- *System/390 MVS Sysplex Hardware and Software Migration, GC28-1210*

System Modification Program Extended (SMP/E)

- *System Modification Program Extended (SMP/E) Reference, SC28-1107*
- *System Modification Program Extended (SMP/E) User' s Guide, SC28-1302*

System Network Architecture (SNA)

- *SNA Formats, GA27-3136*
- *SNA LU 6.2 Peer Protocols Reference, SC31-6808*
- *SNA Transaction Programmer' s Reference Manual for LU Type 6.2, GC30-3084*
- *SNA/Management Services Alert Implementation Guide, GC31-6809*

TCP/IP

- *IBM TCP/IP for MVS: Customization & Administration Guide, SC31-7134*
- *IBM TCP/IP for MVS: Diagnosis Guide, LY43-0105*
- *IBM TCP/IP for MVS: Messages and Codes, SC31-7132*
- *IBM TCP/IP for MVS: Planning and Migration Guide, SC31-7189*

TSO Extensions

- *TSO/E CLISTS, SC28-1876*
- *TSO/E Command Reference, SC28-1881*
- *TSO/E Customization, SC28-1872*
- *TSO/E Messages, GC28-1885*
- *TSO/E Programming Guide, SC28-1874*
- *TSO/E Programming Services, SC28-1875*
- *TSO/E User' s Guide, SC28-1880*

VS COBOL II

- *VS COBOL II Application Programming Guide for MVS and CMS, SC26-4045*
- *VS COBOL II Application Programming: Language Reference, SC26-4047*
- *VS COBOL II Installation and Customization for MVS, SC26-4048*

VTAM

- *Planning for NetView, NCP, and VTAM, SC31-8063*
- *VTAM for MVS/ESA Diagnosis, LY43-0069*
- *VTAM for MVS/ESA Messages and Codes, SC31-6546*
- *VTAM for MVS/ESA Network Implementation Guide, SC31-6548*
- *VTAM for MVS/ESA Operation, SC31-6549*
- *VTAM for MVS/ESA Programming, SC31-6550*
- *VTAM for MVS/ESA Programming for LU 6.2, SC31-6551*
- *VTAM for MVS/ESA Resource Definition Reference, SC31-6552*

Index

We'd Like to Hear from You

DB2 for OS/390
Version 5
Call Level Interface Guide and Reference

Publication No. SC26-8959-02

Please use one of the following ways to send us your comments about this book:

- Mail—Use the Readers' Comments form on the next page. If you are sending the form from a country other than the United States, give it to your local IBM branch office or IBM representative for mailing.
- Fax—Use the Readers' Comments form on the next page and fax it to this U.S. number: 800-426-7773 or (408) 463-4393.
- Electronic mail—Use one of the following network IDs:
  - IBMMail: USIBMXFC @ IBMMAIL
  - IBMLink: DB2PUBS @ STLVM27
  - Internet: DB2PUBS@VNET.IBM.COM

  Be sure to include the following with your comments:

  - Title and publication number of this book
  - Your name, address, and telephone number or your name and electronic address if you would like a reply

Your comments should pertain only to the information in this book and the way the information is presented. To request additional publications, or to comment on other IBM information or the function of IBM products, please give your comments to your IBM representative or to your IBM authorized remarketer.

IBM may use or distribute your comments without obligation.

Readers' Comments

DB2 for OS/390
Version 5
Call Level Interface Guide and Reference

Publication No. SC26-8959-02

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	□	□	□	□	□
Complete	□	□	□	□	□
Easy to find	□	□	□	□	□
Easy to understand	□	□	□	□	□
Well organized	□	□	□	□	□
Applicable to your tasks	□	□	□	□	□
Grammatically correct and consistent	□	□	□	□	□
Graphically well designed	□	□	□	□	□
Overall satisfaction	□	□	□	□	□

Please tell us how we can improve this book:

May we contact you to discuss your comments? □ Yes □ No

Name

Address

Company or Organization

Phone No.

Readers' Comments
SC26-8959-02

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department BWE/H3
PO Box 49023
San Jose, CA 95161-9945

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

IBM®

Program Number: 5655-DB2

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

DB2 for OS/390
Version 5

SC26-8957	Administration Guide
SC26-8958	Application Programming and SQL Guide
SC26-8959	Call Level Interface Guide and Reference
SC26-8960	Command Reference
SC26-8961	Data Sharing: Planning and Administration
SX26-3841	Data Sharing Quick Reference
LY27-9659	Diagnosis Guide and Reference
LY27-9660	Diagnosis Quick Reference
GC26-8970	Installation Guide
GC26-8979	Master Index
SC26-8962	Messages and Codes
SC26-8964	Reference for Remote DRDA Requesters and Servers
SX26-3842	Reference Summary
SC26-8965	Release Guide
SC26-8966	SQL Reference
SC26-8967	Utility Guide and Reference
GC26-8971	What's New?

SC26-8959-02