SYBASE®

OmniConnect™
Component Integration Services User's Guide for Sybase® Adaptive Server™ Enterprise and OmniConnect

Omni
Connect™

September 1997

Document ID: 32702-01-1150

Table of Contents

List of Figures

List of Tables

About This Book

Audience

This book is written for Sybase® Adaptive Server™ Enterprise and OmniConnect™ System Administrators, database administrators, and users.

How to Use This Book

This guide will assist you in configuring and using Component Integration Services. The book includes the following chapters:

- Chapter 1, "Introduction," provides an overview of Component Integration Services.
- Chapter 2, "Understanding Component Integration Services," provides a framework for understanding how Component Integration works. This chapter includes both basic concepts and in-depth topics.
- Chapter 3, "Using Component Integration Services," includes a tutorial designed to help new users get Component Integration Services up and running, and provides configuration and tuning information.
- Chapter 4, "Server Classes," describes the Component Integration Services server classes required to access remote databases.
- Chapter 5, "Utility Programs,"describes the utilities that are specific to Component Integration Services.
- Appendix A, "Troubleshooting," provides troubleshooting tips if you encounter a problem with Component Integration Services.

Adaptive Server Enterprise Documents

The following documents comprise the Sybase Adaptive Server Enterprise documentation:

- The *Release Bulletin* for your platform – contains last-minute information that was too late to be included in the books.

  A more recent version of the *Release Bulletin* may be available on the World Wide Web. To check for critical product or document information that was added after the release of the product CD, use SyBooks™-on-the-Web.

- The Adaptive Server installation documentation for your platform – describes installation and upgrade procedures for all Adaptive Server and related Sybase products.
- The Adaptive Server configuration documentation for your platform – describes configuring a server, creating network connections, configuring for optional functionality, such as auditing, installing most optional system databases, and performing operating system administration tasks.
- *What's New in Adaptive Server Enterprise Release 11.5?* – describes the new features in Adaptive Server release 11.5, the system changes added to support those features, and the changes that may affect your existing applications.
- *Navigating the Documentation for Adaptive Server* – an electronic interface for using Adaptive Server. This online document provides links to the concepts and syntax in the documentation that are relevant to each task.
- *Transact-SQL User's Guide* – documents Transact-SQL®, Sybase's enhanced version of the relational database language. This manual serves as a textbook for beginning users of the database management system. This manual also contains descriptions of the *pubs2* and *pubs3* sample databases.
- *System Administration Guide* – provides in-depth information about administering servers and databases. This manual includes instructions and guidelines for managing physical resources and user and system databases, and specifying character conversion, international language, and sort order settings.
- *Adaptive Server Reference Manual* – contains detailed information about all Transact-SQL commands, functions, procedures, and datatypes. This manual also contains a list of the Transact-SQL reserved words and definitions of system tables.
- *Performance and Tuning Guide* – explains how to tune Adaptive Server for maximum performance. This manual includes information about database design issues that affect performance, query optimization, how to tune Adaptive Server for very large databases, disk and cache issues, and the effects of locking and cursors on performance.

- The *Utility Programs* manual for your platform – documents the Adaptive Server utility programs, such as **isql** and **bcp**, which are executed at the operating system level.
- *Security Administration Guide* – explains how to use the security features provided by Adaptive Server to control user access to data. This manual includes information about how to add users to Adaptive Server, administer both system and user-defined roles, grant database access to users, and manage remote Adaptive Servers.
- *Security Features User's Guide* – provides instructions and guidelines for using the security options provided in Adaptive Server from the perspective of the non-administrative user.
- *Error Messages* and *Troubleshooting Guide* – explains how to resolve frequently occurring error messages and describes solutions to system problems frequently encountered by users.
- *Component Integration Services User's Guide for Adaptive Server Enterprise and OmniConnect* – explains how to use the Adaptive Server Component Integration Services feature to connect remote Sybase and non-Sybase databases.
- *Adaptive Server Glossary* – defines technical terms used in the Adaptive Server documentation.
- *Master Index for Adaptive Server Publications* – combines the indexes of the *Adaptive Server Reference Manual*, *Component Integration Services User Guide*, *Performance and Tuning Guide*, *Security Administration Guide*, *Security Features User's Guide*, *System Administration Guide*, and *Transact-SQL User's Guide*.

Other Sources of Information

Use the SyBooks™ and SyBooks-on-the-Web online resources to learn more about your product:

- SyBooks documentation is on the CD that comes with your software. The DynaText browser, also included on the CD, allows you to access technical information about your product in an easy-to-use format.

  Refer to *Installing SyBooks* in your documentation package for instructions on installing and starting SyBooks.
- SyBooks-on-the-Web is an HTML version of SyBooks that you can access using a standard Web browser.

To use SyBooks-on-the-Web, go to http://www.sybase.com, and choose Documentation.

Conventions

What you type to the computer screen is shown as:

```
Enter text in an entry field
```

Computer output is shown as:

```
OmniConnect returns results.
```

Command arguments you replace with a non-generic value are shown in italics:

machine_name

If You Need Help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, ask a designated person at your site to contact Sybase Technical Support.

1 Introduction

Overview of Component Integration Services

Component Integration Services is a feature that extends Adaptive Server capabilities and provides enhanced interoperability. It is the core interoperability feature of OmniConnect.

Component Integration Services allows Adaptive Server and OmniConnect to present a uniform view of enterprise data to client applications and provides location transparency to enterprise-wide data sources.

Component Integration Services allows users to access both Sybase and non-Sybase databases on different servers. These external data sources include host data files and tables, views and RPCs (remote procedure calls) in database systems such as Adaptive Server, Oracle, and DB2, as shown in Figure 1-1.

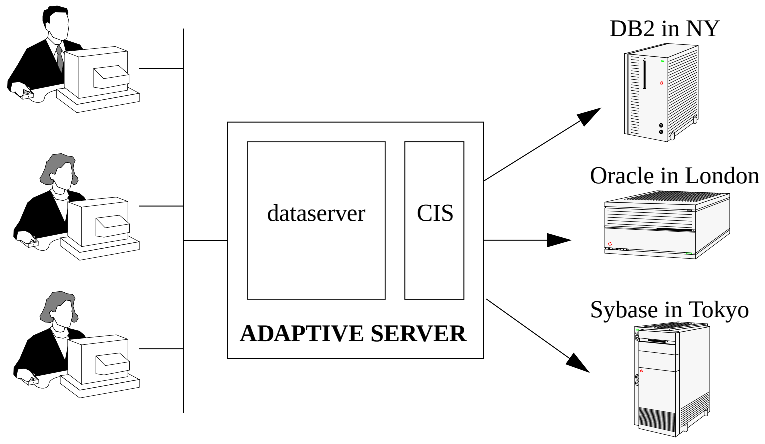


Figure 1-1: Component Integration Services connects to multiple vendor databases

Using Component Integration Services, you can:

- Access tables in remote servers as if the tables were local.
- Perform joins between tables in multiple remote, heterogeneous servers. For example, it is possible to join tables between an Oracle database management system (DBMS) and an Adaptive Server, and between tables in multiple Adaptive Servers.

- Transfer the contents of one table into a new table on any supported remote server by means of a **select into** statement.
- Provide applications, such as PowerBuilder®, Microsoft Access, and DataEase, with transparent access to heterogeneous data.
- Maintain referential integrity across heterogeneous data sources.
- Access native remote server capabilities using the Component Integration Services passthrough mode.

Who Can Use Component Integration Services

Component Integration Services can be used by anyone who needs to access multiple data sources or legacy data. It can also be used by anyone who needs to migrate data from one server to another.

A single server is often used to access data on multiple external servers. Component Integration Services manages the data regardless of the location of the external servers. Data management is transparent to the client application.

Component Integration Services, in combination with EnterpriseConnect™ products, provides transparent access to a wide variety of data sources, including:

- Oracle
- Ingres
- Informix
- Rdb
- IBM databases including:
  - DB2 for MVS
  - DB2/400
  - DB2/2
  - DB2 for VM (SQL/DS)
- Microsoft SQL Server
- Adaptive Server Enterprise
- Adaptive Server Anywhere™
- Mainframe data, including:
  - ADABAS
  - IDMS

- IMS
- VSAM

The list of certified and supported sources and front-end tools is increasing. For current information on all data sources, versions supported, and products required for support, please call the Sybase FAX on Demand at 1-800-423-8737. Request the "Partner Certification Report."

Steps Needed to Use Component Integration Services

To get Component Integration Services running:

- Make sure you have installed the Component Integration Services or the OmniConnect option.
- Issue the command **sp_configure "enable cis", 1**
- Restart Adaptive Server
- Install DirectConnect server(s) or gateways for the external data sources you choose to access (Oracle, Ingres, DB2, Informix, Rdb).
- Configure the server to access remote objects as described in Chapter 3, "Using Component Integration Services."

2 Understanding Component Integration Services

This chapter discusses some of the essential features of Component Integration Services. It is intended to help you understand how Adaptive Server works with the Component Integration Services option configured. The chapter includes the following topics:

- Basic Concepts – is of particular interest to new users; it describes steps the user takes to define objects and create tables.
- Topics and Issues – describes key features of Component Integration Services—the **create existing table** command, the **connect** command, passthrough mode, *text* and *image* datatype processing, and transaction management.
- Internal Operations – describes the underlying operations performed on behalf of an application.
- Adaptive Server Features Not Supported by CIS - describes Adaptive Server features that are not supported by Component Integration Services.

Basic Concepts

Before accessing remote tables with Component Integration Services, you must have a valid *interfaces* file (or *sql.ini* file if you are using Windows NT). For information on setting up an interfaces file, see the configuration documentation for your platform.

Remote Table Access

The ability to access remote (or external) tables as if they were local is a hallmark of Component Integration Services. Component Integration Services presents tables to a client application as if all the data in the tables were stored locally. Internally, when a query involving remote tables is executed, the storage location is determined, and the remote location is accessed so that data can be retrieved.

The access method used to retrieve remote data is determined by two attributes of the external object:

- The server class associated with the remote object
- The object type

To achieve location transparency, which means remote tables appear as local tables to the client, tables must first be mapped to their corresponding external locations. This mapping is performed by means of stored procedures. See the *Adaptive Server Reference Manual* for more information on stored procedures.

Access Methods

Access methods form the interface between the server and an external object. For each server class, there is a separate access method that handles all interaction between Adaptive Server and remote servers of the same class.

Server Classes

A server class must be assigned to each server when it is added by means of the system procedure **sp_addserver**. There are seven server classes, each of which specifies the access method used to interact with the remote server. The server classes are:

- *sql_server* – indicates that the server is a Sybase SQL Server™ or an Adaptive Server or a Microsoft SQL Server. Component Integration Services determines whether the Sybase server is a release 10.0 or later server (supports cursors and dynamic SQL) or a pre-release 10.0 server (does not support cursors or dynamic SQL).
- *local* – the local server. There can be only one.
- *direct_connect* – indicates that the server is an Open Server™ application that conforms to the interface requirements of a DirectConnect™ server.
- *access_server* – a synonym for server class *direct_connect* for compatibility with previous releases.
- *db2* – indicates that the server is a gateway to DB2 or DB2-compatible databases. Net-Gateway™ and Database Gateway for DB2, AS / 400, InfoHub, and SQL / DS fall into this category.
- *generic* – indicates that the server is an Open Server application that conforms to the interface requirements of a Generic Access Module.
- *sds* – indicates that the server conforms to the interface requirements of a Specialty Data Store.

Object Types

The server presents a number of object types to client applications as if they were local tables. Supported object types are:

- *table* – The object in a remote server of any class is a relational table. This is the default type.
- *view* – The object in a remote server of any class is a view. Component Integration Services treats views as if they were local tables without any indexes.
- *rpc* – The object in a remote server of any class is an RPC. Component Integration Services treats the result set from the RPC as a read-only table.

Interface to Remote Servers

The interface between the server and remote servers is handled by the Open Client software, Client-Library™. The Client-Library features that are used to implement the interface are dependent upon the class of server with which Component Integration Services is interacting.

For example, if the server class is *direct_connect (access_server)*, a number of release 11.0 features such as cursor and dynamic requests are used. These features are not used by a server of class *generic*.

Before the server can interact with a remote server, you need to configure the following:

- Remote server addition to the *interfaces* file (or *sql.ini* file if you are using Windows NT)
- Remote server definition
- Remote server login information
- Remote object definition

Remote Server Definition

Remote servers are defined by means of the stored procedure **sp_addserver**. This procedure is documented in the *Adaptive Server Reference Manual*.

Logging into Remote Servers

Once the remote server has been configured, login information must be provided. By default, the server uses the names and passwords of its clients whenever it connects to a remote server on behalf of those clients. However, this default can be overridden by the use of the stored procedure **sp_addexternlogin**. This procedure allows a system administrator to define the name and password for each user who connects to a remote server.

Using **connect to *server_name***, you can verify that the server configuration is correct. This command establishes a passthrough mode connection to the remote server. Passthrough mode allows clients to communicate with remote servers in native syntax. This passthrough mode remains in effect until you issue a **disconnect** command.

Defining Remote Objects

Once a remote server has been properly configured, objects in that remote server cannot be accessed as tables until a mapping between them and a local object (proxy table) has been established.

You can create new tables on remote servers, and you can define the schema for an existing object in a remote server. The procedures for both are similar.

You can use one of two methods for defining the storage location of remote objects:

1. Define the storage location of individual objects
2. Define the default location of all objects in a database

Defining the Storage Location of Individual Objects

Defining individual object storage locations is done by means of the system procedure **sp_addobjectdef**. This procedure allows you to associate a remote object with a local proxy table name. The remote object may or may not exist before you do the mapping. Complete syntax for **sp_addobjectdef** is provided in the *Adaptive Server Reference Manual*.

Defining the Default Storage Location for Tables

Defining the default storage location for all tables in a given database is done by means of the stored procedure **sp_defaultloc**. This procedure establishes the location of tables that are to be created for a given database. The remote objects may or may not already exist. You can override **sp_defaultloc** with **sp_addobjectdef** for individual tables. **sp_defaultloc** can not be used when local tables are required. The syntax for **sp_defaultloc** is provided in the *Adaptive Server Reference Manual*.

Using the *create [existing] table* Command

Once you have defined the storage location, you can create the table as a new or existing object. If the table does not already exist at the remote location, use the **create table** syntax. If it already exists, use the **create existing table** syntax. If the object type is *rpc*, only the **create existing table** syntax is allowed.

When a **create existing table** statement is received, and the object type is either *table* or *view*, the existence of the remote object is checked by means of the catalog stored procedure **sp_tables**. If the object exists, then its column and index attributes are obtained. Column attributes are compared with those defined for the object in the **create existing table** statement. Column name, type, length, and null property are checked. Index attributes are added to the *sysindexes* system table.

Once the object has been created, either as a new or an existing object, the remote object can be queried by using its local name.

Topics and Issues

The following topics, commands, and processes are key features of Component Integration Services and are described in the following sections:

- “Using the create existing table Command”
- “auto identity Option”
- “Passthrough Mode”
- “Transaction Management”
- “RPCs As Read-Only Tables”
- “text and image Datatypes”

Using the *create existing table* Command

The **create existing table** command allows the definition of existing tables (proxy tables). The syntax for this option is similar to the **create table** command and reads as follows:

```
create existing table table_name (column_list)
   [ on segment_name ]
```

The action taken by the server when it receives this command is quite different from the action it takes when it receives the **create table** command, however. In this case, a new table is not created at the remote location; instead, the table mapping is checked, and the existence of the underlying object is verified. If the object does not exist (either host data file or remote server object), the command is rejected with an error message.

If the object does exist, its attributes are obtained and used to update system tables *sysobjects, syscolumns,* and *sysindexes*.

- The nature of the existing object is determined.
- For remote server objects (other than RPCs), column attributes found for the table or view are compared with those defined in the *column_list*. Column names must match identically (although case is ignored), column types and lengths must match, or at least be convertible, and the NULL attributes of the columns must match. (See the data type conversion tables in Chapter 4, "Server Classes."*)*
- Index information from the host data file or remote server table is extracted and used to create rows for the system table *sysindexes*. This defines indexes and keys in server terms and enables the query optimizer to consider any indexes that may exist on this table.
- The **on *segment_name*** clause is processed locally and is not passed to a remote server.
- Referential constraints are passed to the remote location when appropriate. See Chapter 4, "Server Classes."

After successfully defining an existing table, issue an **update statistics** command for the table.This allows the query optimizer to make intelligent choices regarding index selection and join order.

Datatype Conversions

When you use the **create table** or **create existing table** commands, you must specify all datatypes, using recognized Adaptive Server datatypes. If the remote server tables reside on a class of server that is heterogeneous, the datatypes of the remote table are converted into the specified Adaptive Server types automatically when the data is retrieved. If the conversion cannot be made, the **create table** or **create existing table** commands do not allow the table to be created or defined.

Chapter 4, “Server Classes,” contains a section for each supported server class that describes all possible datatype conversions that are implicitly performed by the server.

Example of Remote Table Definition

The following example illustrates the steps necessary to define the remote Adaptive Server table, *authors*, starting with the server definition:

1. Define a server named SYBASE. Its server class is *sql_server*, and its name in the interfaces file is SYBASE:

   ```
   exec sp_addserver SYBASE, sql_server, SYBASE
   ```

2. Define a remote login alias. This step is optional. User “sa” is known to remote server SYBASE as user “sa,” password “timothy”:

   ```
   exec sp_addexternlogin SYBASE, sa, sa, timothy
   ```

3. Add an object definition for the remote *authors* table, to be known locally as *authors*:

   ```
   exec sp_addobjectdef authors,
   "SYBASE.pubs2.dbo.authors", "table"
   ```

4. Define the remote *authors* table:

   ```
   create existing table authors
   (
       au_id           id              not null,
       au_lname        varchar(40)     not null,
       au_fname        varchar(20)     not null,
       phone           char(12)        not null,
       address         varchar(40)     null,
       city            varchar(20)     null,
       state           char(2)         null,
       country         varchar(12)     null,
       postalcode      char(10)        null
   )
   ```

5. Update statistics in tables to ensure reasonable choices by the query optimizer:

```
update statistics authors
```

6. Execute a query to test the configuration:

```
select * from authors where au_lname = 'Carson'
```

auto identity Option

When the Adaptive Server **auto identity** database option is enabled, an IDENTITY column is added to any tables that are created in the database. The column name is *CIS_IDENTITY_COL*, for proxy tables, or *SYB_IDENTITY_COL*, for local tables. In either case, the column can be referenced using the **syb_identity** keyword.

Passthrough Mode

Passthrough mode is provided within Component Integration Services as a means of enabling a user to perform native operations on the server to which the user is being "passed through."

For example, requesting passthrough mode for an Oracle server, allows you to send native Oracle SQL statements to the Oracle DBMS. Results are converted into a form that is usable by the Open Client™ application and passed back to the user.

The Transact-SQL® parser and compiler are bypassed in this mode, and each language batch received from the user is passed directly to the server to which the user is connected in passthrough mode. Results from each batch are returned to the client.

There are several ways to use passthrough mode:

- The **connect to** command
- The **sp_autoconnect** stored procedure
- The **sp_passthru** stored procedure
- The **sp_remotesql** procedure

The *connect to* Command

The **connect to** command enables users to specify the server to which a passthrough connection is required. The syntax of the command is as follows:

```
connect to server_name
```

where *server_name* is the name of a server added to the *sysservers* table, with its server class and network name defined. See **sp_addserver** in the *Adaptive Server Reference Manual.*

When establishing a connection to *server_name* on behalf of the user, the server uses:

- A remote login alias set using **sp_addexternlogin**, or
- The name and password used to communicate with the Adaptive Server.

In either case, if the connection cannot be made to the server specified, the reason is contained in a message returned to the user.

Once a passthrough connection has been made, the Transact-SQL parser and compiler are bypassed when subsequent language text is received. Any statements received by the server are passed directly to the specified remote server.

➤ ***Note***

Some database management systems do not recognize more than one statement at a time and produce syntax errors if, for example, multiple **select** statements were received as part of a single language text buffer.

After statements are passed to the requested server, any results are converted into a form that can be recognized by the Open Client interface and sent back to the client program.

To exit from passthrough mode, issue the **disconnect**, or **disc**, command. Subsequent language text from this client is then processed using the Transact-SQL parser and compiler.

Permission to use the **connect to** command must be explicitly granted by the System Administrator. The syntax is:

```
grant connect to user_name
```

To revoke permission to use the **connect to**, the syntax is:

```
revoke connect from user_name
```

The **connect to** permissions are stored in the *master* database. To globally grant or revoke permissions to “public”, the System Administrator sets the permissions in the *master* database; the effect is server-wide, regardless of what database is being used. The System Administrator can only grant or revoke permissions to or from a user, if the user is a valid user of the *master* database.

The System Administrator can grant or revoke "all" permissions to or from "public" within any database. If the permissions are in the *master* database, "all" includes the **connect to** command. If they are in another database, "all" does not include the **connect to** command.

Example

The System Administrator wants to revoke permission from "public" and wants only the user "fred" to be able to execute the **connect to** command. "fred" must be made a valid user of *master*. To do this, the System Administrator issues the following commands in *master*:

```
revoke connect from public
sp_adduser fred
grant connect to fred
```

sp_autoconnect

Some users may always require a passthrough connection to a given server. If this is the case, Component Integration Services can be configured so that it automatically connects these users to a specified remote server in passthrough mode when the users connect to the server. This feature is enabled and disabled by the system procedure **sp_autoconnect** using the following syntax:

sp_autoconnect *server_name*, true|false [, *loginname*]

Before using **sp_autoconnect**, add the *server_name* to *sysservers* by using **sp_addserver.**

A user can request automatic connection to a server using **sp_autoconnect**, but only the System Administrator can enable or disable automatic passthrough connection for another user. Thus, only the System Administrator can specify a third argument to this procedure.

If the second argument is **true**, the **autoconnect** feature is enabled for the current user (or the user specified in the third argument). If the second argument is **false**, the **autoconnect** feature is disabled.

Anytime a user connects to the server, that user's *autoconnect* status in *syslogins* is checked. If enabled, the *server_name*, also found in *syslogins* (placed there by **sp_autoconnect**), is checked for validity. If the server is valid, the user is automatically connected to that server, and a passthrough status is established. Subsequent language statements received by the server from this user are handled exactly as if the user explicitly entered the **connect** command. This user then views the server very much like a passthrough gateway to the remote server.

When an “autoconnected” user executes a **disconnect**, she or he is returned normally to the server.

If the remote server cannot be reached, the user (unless the user is assigned the “sa” role) will not be connected to the local Adaptive Server. A “login failed” error message is returned.

sp_passthru

The **sp_password** procedure allows the user to pass a SQL command buffer to a remote server. The syntax of the SQL statement(s) being passed is assumed to be the syntax native to the class of server receiving the buffer; no translation or interpretation is performed. Results from the remote server are optionally placed in output parameters. The syntax for **sp_passthru** follows:

```
sp_passthru server, command, errcode, errmsg, rowcount
    [, arg1, arg2, ... argn]
```

where:

- *server* is the name of the server that is to receive the SQL command buffer; the datatype is *varchar(30)*.
- *command* is the SQL command buffer; the datatype is *varchar(255)*.
- *errcode* is the error code returned by the remote server; the datatype is *int output*.
- *errmsg* is the error message returned by the remote server; the datatype is *varchar(255) output*.
- *rowcount* is the number of rows affected by the last command in the command buffer; the datatype is *int output*.
- *arg1–argn* are optional parameters. If provided, these output parameters will receive the results from the last row returned by the last command in the command buffer. The datatypes may vary. All must be *output* parameters.

Example

```
sp_passthru ORACLE, "select date from dual",
@errcode output, @errmsg output, @rowcount output,
@oradate output
```

This example returns the date from the Oracle server in the output parameter *@oradate*. If an Oracle error occurs, the error code is placed in *@errcode* and the corresponding message is placed in *@errmsg*. The *@rowcount* parameter is set to 1.

For more information on **sp_passthru** and its return status, refer to the *Adaptive Server Reference Manual*.

sp_remotesql

sp_remotesql allows you to pass native syntax to a remote server. The procedure establishes a connection to a remote server, passes a query buffer, and relays the results back to the client. The syntax for **sp_remotesql** is as follows:

```
sp_remotesql server_name, query_buf1
   [, query_buf2, ... , query_buf254]
```

where:

- *server_name* is the name of a server that has been defined using **sp_addserver**. *server_name* is a *varchar(30)* field. If *server_name* is not defined or is not available, the connection fails, and the procedure is aborted. This parameter is required.
- *query_buf1* is a query buffer of type *char* or *varchar* with a maximum length of 255 bytes. This parameter is required.

Each additional buffer is *char* or *varchar* with a maximum length of 255 bytes. If supplied, these optional arguments are concatenated with the contents of *query_buf1* into a single query buffer.

Example

```
   sp_remotesql freds_server, "select @@version"
```

In this example, the server passes the query buffer to *freds_server*, which interprets the **select** *@@version* syntax and returns version information to the client. The returned information is not interpreted by the server.

For more information on **sp_remotesql** and its return codes, refer to the *Adaptive Server Reference Manual*.

Transaction Management

Transactions provide a way to group Transact-SQL statements so that they are treated as a unit—either all work performed by the statements is committed to the database, or none of it is.

For the most part, transaction management with Component Integration Services is the same as transaction management in Adaptive Server, but there are some differences. They are discussed in the following section, “Overview.”

Overview

Component Integration Services makes every effort to manage user transactions reliably. However, the different access methods incorporated into the server allow varying degrees of support for this capability. The general logic described below is employed by server classes *direct_connect (access_server)*, *sql_server* (when the server involved is release 10.0 or later), and *sds* if the Specialty Data Store supports transaction management.

The method for managing transactions involving remote servers uses a two-phase commit protocol. Adaptive Server 11.5 implements a strategy that ensures transaction integrity for most scenarios. However, there is still a chance that a distributed unit of work will be left in an undetermined state. Even though two-phase commit protocol is used, no recovery process is included.

The general logic for managing a user transaction is as follows:

Component Integration Services prefaces work to a remote server with a **begin transaction** notification. When the transaction is ready to be committed, Component Integration Services sends a **prepare transaction** notification to each remote server that has been part of the transaction. The purpose of **prepare transaction** is to “ping” the remote server to determine that the connection is still viable. If a **prepare transaction** request fails, all remote servers are told to roll back the current transaction. If all **prepare transaction** requests are successful, the server sends a **commit transaction** request to each remote server involved with the transaction.

Any command preceded by **begin transaction** can begin a transaction. Other commands are sent to a remote server to be executed as a single, remote unit of work.

Transactional RPCs

The server allows RPCs to be included within the unit of work initiated by the current transaction.

Before using transactional RPCs, issue the **set transactional_rpc on** or **set cis_rpc_handling on** command.

Assuming that the remote server can support the inclusion of RPCs within transactions, the following syntax shows how this capability might be used:

```
begin transaction
   insert into t1 values (1)
   update t2 set c1 = 10
   execute @status = RMTSERVER.pubs2.dbo.myproc
   if @status = 1
       commit transction
   else
       rollback transaction
```

In this example, the work performed by the procedure *myproc* in server RMTSERVER is included in the unit of work that began with the **begin transaction** command. This example requires that the remote procedure *myproc* return a status of "1" for success. The application controls whether the work is committed or rolled back as a complete unit.

The server that is to receive the RPC must allow RPCs to be included in the same transactional context as Data Manipulation Language (DML) commands (**select**, **insert**, **delete**, **update**). This is true for Adaptive Server and is expected to be true for most DirectConnect products being released by Sybase. However, some database management systems may not support this capability.

Restrictions on Transaction Management

Restrictions on transaction management are as follows:

- Savepoints are not propagated to remote servers.
- If nested **begin transaction** and **commit transaction** statements are included in a transaction that involves remote servers, only the outermost set of statements is processed. The innermost set, containing the **begin transaction** and **commit transaction** statements, is not transmitted to remote servers.
- The transaction model described in "Overview" on page 2-13 is not supported in server class *generic* or server class *db2*. It is also not supported in server class *sql_server* when the remote server is a pre-release 10.0 SQL Server or a Microsoft SQL Server. In these cases, the transactions are committed after each statement is completed.

RPCs As Read-Only Tables

Component Integration Services users can map remote or external objects of the type *rpc* to local proxy tables. If a table is created in this way, it can be referenced only by the **select** and **drop** commands. The commands **insert**, **delete**, and **update** generate error messages, since the table is assumed to be read-only.

If an object of the type *rpc* has been defined within the server, a query is not issued to the remote server on which the object resides. Instead, the server issues an RPC and treats the results from the RPC as a read-only table.

Examples

```
sp_addobjectdef rtable, "RMTSERVER...myproc", "rpc"

create existing table rtable
(   col1    int,
    col2    datetime,
    col3    varchar(30)
)

select * from rtable
```

When this query is issued, the server sends the RPC named *myproc* to server RMTSERVER. Row results are treated like the results from any other table; they can be sorted, joined with other tables, grouped, inserted into another table, and so forth.

RPC parameters should represent arguments that restrict the result set. If the RPC is issued without parameters, the entire result set of the object is returned. If the RPC is issued with parameters, each parameter further limits the result set. For example, the following query:

```
select * from rtable where col1 = 10
```

results in a single parameter, named *@col1*, that is sent along with the RPC. Its value is 10.

Component Integration Services attempts to pass as many of the search arguments as possible to the remote server, but depending on the SQL statement being executed, Component Integration Services might perform the result set calculation itself. Each parameter represents a search for an exact match, for example, the **=** operator.

The following are rules which define the parameters sent to the RPC. If an RPC will be used as a Component Integration Services object, these rules should be kept in mind during development.

- Component Integration Services sends **=** operators in the **where** clause as parameters. For example, the query:

  ```
  select * from rpc1 where a = 3 and b = 2
  ```

  results in Component Integration Services sending two parameters. Parameter *a* has a value of 3 and parameter *b* has a value of 2. The RPC is expected to return only result rows in which column *a* has a value of 3 and column *b* has a value of 2.

- Component Integration Services will not send any parameters for a **where** clause, or portion of a **where** clause, if there is not an exact search condition. For example:

  ```
  select * from rpc1 where a = 3 or b = 2
  ```

  Component Integration Services will not send parameters for *a* or *b* because of the **or** clause.

  Another example:

  ```
  select * from rpc1 where a = 2 and b < 3
  ```

  Component Integration Services will not send parameters because there is nothing in the **where** clause representing an exact search condition. Component Integration Services will perform the result set calculation locally.

text and *image* Datatypes

The *text* datatype is used to store printable character data which can be more than 255 bytes. The *image* datatype is used to store more than 255 bytes of hexadecimal-encoded binary data. The maximum length for *text* and *image* data is defined by the server class of the remote server to which the column is mapped:

- For servers of class *sql_server*, the maximum is 2147MB.
- For Open Server applications of class *direct_connect (access_server)* the maximum byte count is defined by the functionality of the DirectConnect server.

Restrictions on *text* and *image* Columns

text and *image* columns cannot be used:

- As parameters to stored procedures. *text* or *image* values cannot be passed to stored procedures.
- As local variables.

- In **order by, compute**, or **group by** clauses.
- In indexes.
- In subqueries.
- In **where** clauses, except with the keyword **like.**
- In joins.

Limits of @@*textsize*

select statements return *text* and *image* data up to the limit specified in the global variable *@@textsize*. The **set textsize** command is used to change this limit. The initial value of *@@textsize* is 32K; the maximum value for *@@textsize* is 2147MB.

Odd Bytes Padded

image values of less than 255 bytes that have an odd number of bytes are padded with a leading zero (an insert of "0xaaabb" becomes "0x0aaabb"). It is an error to **insert** an *image* value of more than 255 bytes if the value has an odd number of bytes.

Converting *text* and *image* Datatypes

You can explicitly convert *text* values to *char* or *varchar* and *image* values to *binary* or *varbinary* with the **convert** function, but you are limited to the maximum length of the *character* and *binary* datatypes, 255 bytes. If you do not specify the length, the converted value has a default length of 30 bytes. Implicit conversion is not supported.

Pattern Matching with *text* Data

Use the **patindex** function to search for the starting position of the first occurrence of a specified pattern in a *text*, *varchar*, or *char* column. The % wildcard character must precede and follow the pattern (except when you are searching for the first or last character).

You can use the **like** keyword to search for a particular pattern. The following example selects each *text* data value from the *blurb* column of the *texttest* table that contains the pattern “Straight Talk%”:

```
select blurb from texttest
where blurb like "Straight Talk%"
```

Entering *text* and *image* values

The DB-Library™ functions **dbwritetext** and **dbmoretext** and the Client-Library function **ct_send_data** are the most efficient ways to enter *text* and *image* values.

When inserting *text* or *image* values using the **insert** command, the length of the data is limited to 450 bytes.

readtext using bytes

If you use the **readtext using bytes** command on a *text* column, and the combination of size and offset result in the transmission of a partial character, then errors result.

text and *image* with *bulk copy*

When you use **bulk copy** to copy *text* and *image* values to a remote server, the server must store the values in data pages before sending them to the remote server. Once the values have been issued to the remote server, the data pages are released. Data pages are allocated and released row by row. Users must be aware of this for the following reasons:

- The overhead of allocating and releasing data pages impacts performance.
- The data pages are allocated in the database where the table resides, so the database must be large enough to accommodate enough data pages for the largest *text* and *image* values that exist for any given row.

Error Logging

Processing of *text* and *image* data (with remote servers only) can be logged by using trace flag 11207.

text and *image* Data with Server Class *sql_server*

- A pointer in a *text* or *image* column is assigned when the column is initialized. Before you can enter *text* or *image* data into a column, the column must be initialized. This causes a 2K page to be allocated on the remote or Adaptive Server. To initialize *text* or *image* columns, use the **update** or a non-null **insert** command. See **writetext** for more information.

- Before you use **writetext** to enter *text* data or **readtext** to read it, the *text* column must be initialized. Use **update** or **insert** non-null data to initialize the *text* column, and then use **writetext** and **readtext**.
- Using **update** to replace existing *text* and *image* data with NULL, reclaims all of the allocated data pages, except the first page, in the remote server.
- **writetext**, **select into**, DB-Library functions, or Client-Library functions must be used to enter *text* or *image* values that are larger than 450 bytes.
- **insert select** cannot be used to insert *text* or *image* values.
- **readtext** is the most efficient way to access *text* and *image* data.

text and *image* Data with Server Class *direct_connect (access_server)*

- Specific DirectConnect servers support *text* and *image* data to varying degrees. Refer to the DirectConnect documentation for information on *text* and *image* support.
- The server uses the length defined in the global variable *@@textsize* for the column length. Before issuing **create table**, the client application should set *@@textsize* to the required length by invoking the **set textsize** command.
- For DirectConnect servers that support *text* and *image* datatypes but do not support text pointers, the following restrictions apply:
  - The **writetext** command is not supported.
  - The **readtext** command is not supported.
  - Client-Library functions that use text pointers are not supported.
  - DB-Library functions that use text pointers are not supported.
- For DirectConnect servers that support *text* and *image* datatypes but do not support text pointers, some additional processing is performed to allow the following functions to be used:
  - **patindex**
  - **char_length**
  - **datalength**

  If text pointers are supported, the server performs these functions by issuing an RPC to the DirectConnect server.

- For DirectConnect servers that do not support text pointers, the server stores data in the *sysattributes* system table. Data pages are preallocated on a per column per row basis. The column size is determined by the *@@textsize* global variable. If this value is not sufficient an error is returned.
- Specific DirectConnect servers may or may not support pattern matching against the *text* datatype. If a DirectConnect server does not support this pattern matching, the server copies the *text* value to internal data pages and performs the pattern matching internally. The best performance is seen when pattern matching is performed by the DirectConnect server.
- **writetext**, **select into**, or **insert...select** must be used to enter *text* or *image* values that exceed 450 bytes.
- **select into** and **insert...select** can be used to insert *text* or *image* values, but the table must have a unique index.

db2 Server Issues

text and *image* datatypes for a server of class *db2* are not supported. If you need *text* and *image* datatypes, you must use a DirectConnect server.

Internal Operations

This section describes the underlying operations on remote servers performed by Component Integration Services on behalf of client applications.

Connection Management

When connecting to a remote server on behalf of a client, the server uses Client-Library functions. Once the first connection to a remote server is established for a given client, that connection remains open until the client disconnects from Component Integration Services.

For servers of class *direct_connect (access_server)* and *sql_server* (release 10.0 and later), only one connection is established to that server for each client that requires access to that server. All interaction with these servers is done within this single connection context.

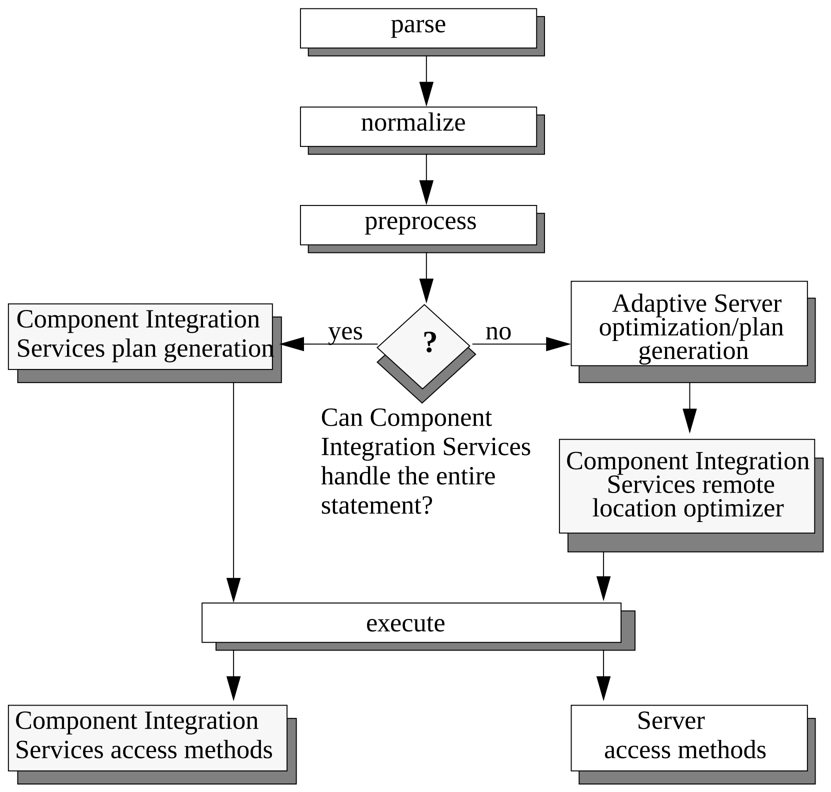
However, for pre-release 10.0 SQL Server, and servers of class *db2* and *generic*, it may be necessary to establish more than one connection to that server in order to process a single client request. In this case, multiple connections are established as needed, and all but one are closed when the Transact-SQL command requiring them has completed.

Query Processing

The query processing steps taken when Component Integration Services is enabled are similar to the steps taken by Adaptive Server, except for the following:

- If a client connection is made in passthrough mode, the Adaptive Server query processing is bypassed and the SQL text is forwarded to the remote server for execution.
- When **select**, **insert**, **delete** or **update** statements are submitted to the server for execution, additional steps may be taken by Component Integration Services to improve the query's performance, if local proxy tables are referenced.

The query processing steps are shown in Figure 2-1. An overview of these steps follows.


parse
normalize
preprocess
Component Integration Services plan generation
yes
?
no
Adaptive Server optimization/plan generation
Can Component Integration Services handle the entire statement?
Component Integration Services remote location optimizer
execute
Component Integration Services access methods
Server access methods

▢ Shaded boxes indicate steps taken by Component Integration Services.

Figure 2-1: Query processing steps

Query Parsing

The SQL parser checks the syntax of incoming SQL statements, and raises an error if the SQL being submitted for execution is not recognized by the Transact-SQL parser.

Query Normalization

During query normalization, each object referenced in the SQL statement is validated. Query normalization verifies the objects referenced in the statement exist, and the datatypes are compatible with values in the statement.

Example

```
select * from t1 where c1 = 10
```

The query normalization stage verifies that table *t1* with a column named *c1* exists in the system catalogs. It also verifies that the datatype of column *c1* is compatible with the value 10. If the column's datatype is *datetime*, for example, this statement is rejected.

Query Preprocessing

Query preprocessing prepares the query for optimization. It may change the representation of a statement such that the SQL statement Component Integration Services generates will be syntactically different from the original statement.

Preprocessing performs view expansion, so that a query can operate on tables referenced by the view. It also takes steps such as reordering expressions and transforming subqueries to improve processing efficiency. For example, subquery transformation may convert some subqueries into joins.

Decision Point

After preprocessing, a decision is made as to whether Component Integration Services or the standard Adaptive Server query optimizer will handle optimization.

Component Integration Services will handle optimization (using a feature known as quickpass mode) when:

- Every table represented in the SQL statement resides within a single remote server.
- The remote server is capable of processing all the syntax represented by the statement.

  Component Integration Services determines the query processing capabilities of the remote server by its server class. Servers with server class *sql_server*, *db2*, or *generic* have implied capabilities. For example, Component Integration Services assumes that any server configured as server class *sql_server* is capable of processing all Transact-SQL syntax.

For remote servers with server class *access_server* or *direct_connect*, Component Integration Services issues an RPC to ask the remote server for its capabilities the first time a connection is made to the server. Based on the server's response to the RPC, Component Integration Services determines the syntax of the SQL it will forward to the remote server.

- The following is true of the SQL statement:
  - It is a **select**, **insert**, **delete**, or **update** statement
  - If it is an **insert**, **update**, or **delete** statement, there are no *identity* or *timestamp* columns, or referential constraints
  - It contains no *text* or *image* columns
  - It contains no **compute by** clauses
  - It contains no **for browse** clauses
  - It is not a **select...into** statement
  - It is not a cursor-related statement (for example, **fetch**, **declare**, **open**, **close**, **deallocate**, **update** or **delete** statements that include **where current of cursor**)

If the above conditions are not met, quickpass mode cannot be used, and the standard Adaptive Server query optimizer handles optimization.

Component Integration Services Plan Generation

If quickpass mode can be used, Component Integration Services produces a simplified query plan. When statements contain proxy tables, they are executed more quickly when processed by the remote server than when processed through the Adaptive Server plan generation phase.

Adaptive Server Optimization and Plan Generation

Adaptive Server optimization and plan generation evaluates the optimal path for executing a query and produces a query plan that tells the Adaptive Server how to execute the query.

If the **update statistics** command has been run for the tables in the query, the optimizer has sufficient data on which to base decisions regarding join order. If the **update statistics** command has not been run, the Adaptive Server defaults apply.

For more information on Adaptive Server optimization, refer to Chapter 7, “The Adaptive Server Query Optimizer,” in the *Performance and Tuning Guide*.

Component Integration Services Remote Location Optimizer

Adaptive Server generates a query plan containing the optimal join order for a multitable query **without** regard to the storage location of each table. If remote tables are represented in the query, Component Integration Services, which takes the storage location into account, performs additional optimization for the following conditions:

- Join processing
- Aggregate processing

In order to make intelligent evaluations of a query to improve performance in the above areas, statistics are required. These are obtained by executing the command **update statistics** for a specific table.

update statistics

When updating statistics on a remote table, Component Integration Services intercepts the request and provides meaningful statistics for the remote table and all of its indexes (if any). The result of executing an **update statistics** command is a distribution statistics page stored in the database, for each index.

In Adaptive Server, data used to create this distribution page comes from local index pages. When you are updating statistics on a remote table, the data used to create the distribution statistics page comes from the keys used to make up the index on the remote table.

The server issues a query to the remote server to obtain all columns making up the index, sorted according to position within the index. For example, if *table1* has an index made up of two columns, *col1* and *col2*, then the query to that server is sent as follows when **update statistics** is executed:

```
select col1, col2 from table1 order by col1, col2
```

The results are then used to construct a distribution page in the format needed by the optimizer.

The detailed distribution statistics are used to determine optimal join order. This gives the server the ability to generate optimal queries against remote databases that may not support cost-based query optimization.

On large tables, **update statistics** can take a long time. To speed up the process, turn on trace flag 11209 before executing **update statistics**. This trace flag instructs **update statistics** to obtain only row counts on remote tables. The Adaptive Server query optimizer uses the row count information to make assumptions about the selectivity of a particular index. While these assumptions are not as complete as the full distribution statistics, they provide the minimal information needed to handle query optimization.

Join Processing

Component Integration Services remote location optimizer isolates join conditions represented in the query plan. For each remote server that is represented by two or more tables in the join, Component Integration Services modifies the query plan to appear as though a single virtual table is being processed for that server. Component Integration Services then forwards the join conditions to the remote server during query execution.

For example, if a query involves four tables, two that are located on the remote server SERVERA and two that are located on the remote server SERVERB, Component Integration Services processes the query as though it were a two-way join. The following query:

```
select * from A1, A2, B1, B2
where A1.id = A2.id and A2.id = B1.id
and B1.id = B2 id
```

gets converted to:

```
select * from V1, V2 where V1.id = V2.id
```

V1 is the virtual table representing the results of the join between *A1* and *A2* (processed by SERVERA), and *V2* is the virtual table representing the results of the join between *B1* and *B2* (processed by SERVERB). Since the Adaptive Server uses nested iteration (looping) to process inner tables of a join, the query is processed as follows:

```
open cursor on V1
fetch V1 row
for each row in V1
    open a cursor on V2
    fetch V2
    route results V1, V2 to client
    close cursor on V2
```

Aggregate Processing

Component Integration Services optimizes queries containing ungrouped aggregate functions (**min**, **max**, **sum,** and **count**) by passing the aggregate to the remote server if the remote server is capable of performing the function.

For example, consider the following query on the remote table *A1*:

```
select count(*) from A1 where id > 100
```

The **count(*)** aggregate is forwarded to the remote server that owns *A1*.

Query Execution

The query execution stage receives a query plan, generated either as a result of an adhoc query or a stored procedure, and executes each step of the plan, according to the information stored in the plan. Query plan structures are tagged with information that indicates which access method is to be invoked. If a table is local, then normal Adaptive Server access methods used to process a query are activated as required by the plan execution logic. If the table is remote, then Component Integration Services access methods are invoked to process each table (or virtual table) represented in the query.

Component Integration Services Access Methods

The Component Integration Services access methods interact with the remote servers that contain objects represented in a query. In Adaptive Server 11.5, all interaction is done through Client-Library.

When an entire statement can be forwarded to the remote server, the statement is taken from the query plan. After any parameters have been substituted into the text of the statement, the entire statement is forwarded to the appropriate remote server.

When the Adaptive Server optimizer and plan generator are involved, the statement or fragment of a statement that is to be executed remotely is constructed from data structures contained within the query plan. The statement or fragment of a statement is then forwarded to the appropriate remote server.

The results from the remote servers are then converted into the necessary internal data types, and processed as if they were derived from local tables.

When an **order by** is processed by the remote server, the results may be different from what Adaptive Server would return for the same query, because the sort order is determined by the remote server, not by Adaptive Server.

Query Plan Execution

Any command that could affect a table is checked by the server to determine whether the object has a local or remote storage location. If the storage location is remote, then the appropriate access method is invoked when the query plan is executed in order to apply the requested operation to the remote objects. The following commands are affected if they operate on objects that are mapped to a remote storage location:

- **alter table**
- **begin transaction**
- **commit**
- **create index**
- **create table**
- **create existing table**
- **deallocate table**
- **declare cursor**
- **delete**
- **drop table**
- **drop index**
- **execute**
- **fetch**
- **insert**
- **open**
- **prepare transaction**
- **readtext**
- **rollback**
- **select**
- **set**
- **setuser**

- **truncate table**
- **update**
- **update statistics**
- **writetext**

create table Command

When the server receives a **create table** command, the command is interpreted as a request for new table creation. The server invokes the access method appropriate for the server class of the table that is to be created, if it is remote, and then creates the table. If this command is successful, system catalogs are updated, and the object appears to clients as a local table in the database in which it was created.

The **create table** command is reconstructed in a syntax that is appropriate for the server class. For example, if the server class is *db2*, then the command is reconstructed using DB2 syntax before being passed to the remote server. Datatype conversions are made for datatypes that are unique to the Adaptive Server environment.

Datatype conversion charts for each server class are provided in Chapter 4. Some server classes have restrictions on what datatypes can and cannot be supported. These are also described in Chapter 4, “Server Classes.”

The **create table** command is passed to remote servers as a language request.

create existing table Command

When a **create existing table** command is received, it is interpreted as a request to import metadata from the remote or external location of the object for updating system catalogs. Importing this metadata is performed by means of three RPCs sent to the remote server with which the object has been associated:

- **sp_tables** – verifies that the remote object actually exists.
- **sp_columns** – obtains column attributes of the remote object for comparison with those defined in the **create existing table** command.
- **sp_statistics** – obtains index information in order to update the local system table, *sysindexes*.

alter table Command

When the server receives the **alter table** command, it passes the command to an appropriate access method if:

- The object on which the command is to operate has been associated with a remote or external storage location.
- The command consists of an **add column** request. Requests to add or drop constraints are not passed to the access methods; instead, they are handled locally.

The **alter table** command is passed to remote servers as a language request.

create index Command

When the server receives the **create index** command, it passes the command to an appropriate access method, if the object on which the command is to operate has been associated with a remote or external storage location.

The command is reconstructed using a syntax appropriate for the class and is passed to the remote server for execution.

The **create index** command is passed to remote servers as a language request.

drop table Command

When the server receives the **drop table** command for a remote table, a check is made to determine whether the table to be dropped has been created with the **existing** option. If so, references to the object within the system tables are removed, and the operation is complete.

If the table was not created with the **existing** option, the command is passed to an appropriate access method, if the object on which the command is to operate has been associated with a remote or external storage location.

The **drop table** command is reconstructed using a syntax appropriate for the class and is passed to the remote server for execution.

This command is passed to remote servers as a language request.

In all cases, references to the object from within the system catalogs are removed.

drop index Command

When the server receives the **drop index** command, it passes the command to an appropriate access method, if the object on which the command is to operate has been associated with a remote or external storage location.

The **drop index** command is reconstructed using a syntax appropriate for the class and is passed to the remote server for execution.

This command is passed to remote servers as a language request.

truncate table Command

When the server receives the **truncate table** command, it passes the command to an appropriate access method, if the object on which the command is to operate has been associated with a remote or external storage location.

The command is reconstructed using a syntax appropriate for the class and is passed to the remote server for execution. Since this syntax is unique to the Adaptive Server environment, a server of class *db2* would receive a **delete** command with no qualifying **where** clause:

```
delete from t1
```

The **truncate table** command is passed to remote servers as a language request.

Triggers

Component Integration Services allows triggers on proxy tables; however their usefulness is limited. It is possible to **create** a trigger on a proxy table and the trigger will be invoked just as it would be for a normal Adaptive Server table. However, before and after *image* data is not written to the log for proxy tables because the **insert**, **update** and **delete** commands are passed to the remote server. The *inserted* or *deleted* tables, which are actually views into the log, contain no data for proxy tables. Users cannot examine the rows being inserted, deleted, or updated, so a trigger with a proxy table has limited value.

Referential Integrity

You can use Component Integration Services to maintain referential integrity between remote tables. See the section on constraints in the

Transact-SQL User's Guide. During **update**, **insert**, and **delete** operations, Component Integration Services checks the referenced table. If the check fails, the transaction is rolled back.

Security Issues

When establishing a connection to a remote Adaptive Server, Client-Library functions are used instead of a site handler when either **cis_rpc_handling** or **set transactional_rpc** is **on**. This method of establishing connections prevents the remote server from distinguishing these connections from those of other clients. Thus, any remote server security configured on the remote server to allow or disallow connections from a given server does not take effect.

Another Adaptive Server with Component Integration Services enabled cannot use **trusted mode** for remote server connections. This forces the Adaptive Server to be configured with all possible user accounts if it is going to be used with Component Integration Services.

Passwords are stored internally in encrypted form.

Trusted Mode

Trusted mode can be used only between two servers with site handlers. When Component Integration Services establishes a Client-Library connection to a remote server, trusted mode cannot be used. If a trusted mode connection is needed, use **set cis_rpc_handling off**.

For more information about trusted mode, see Chapter 8, "Managing Remote Servers," in the *Security Administration Guide.*

Adaptive Server Features Not Supported by CIS

Parallel Processing

Parallel processing is disabled while Component Integration Services accesses remote tables.

External Security

Clients connecting to Adaptive Server can use external security features when Component Integration Services is enabled. However, Component Integration Services does not incorporate security features when communicating with remote servers. Clients using external security features should use **sp_addexternlogin** to access remote servers.

Directory Services

The Directory Services feature is not used by Component Integration Services.

3 Using Component Integration Services

This chapter provides information on defining objects, configuring, tuning, and using Component Integration Services.

- Getting Started with Component Integration Services – is a tutorial, to assist first-time users in completing the basic Component Integration Services configuration steps.
- Configuration and Tuning – provides information for System Administrators. See the *System Administration Guide,* the *Performance and Tuning Guide,* and the *Adaptive Server Reference Manual* for additional information.

Getting Started with Component Integration Services

This section is intended to help first-time users get Component Integration Services running quickly. It provides a step-by-step guide to configuring the server to access remote data sources. It includes instructions for:

- Adding a remote server
- Mapping remote objects to local proxy tables
- Performing joins between remote tables

Routine system administration tasks such as starting and stopping Adaptive Server, creating logins, creating groups, adding users, granting permissions, and password administration are explained in the Adaptive Server documentation.

Adding a Remote Server

You can use the server to access data on remote servers. Before you can do this, you must configure Component Integration Services.

Follow these steps to configure the server to access remote data:

Overview of the Procedure

1. Add the remote server to the interfaces file, using the **dsedit** or **dscp** utility.
2. Add the name, server class, and network name of the remote server to system tables, using the system procedure **sp_addserver**.

3. Assign an alternate login name and password, using the system procedure **sp_addexternlogin**. This step is optional.

Step 1: Add the Remote Server to the Interfaces File

Use the **dsedit** or **dscp** utility to edit the interfaces file located in the $*SYBASE* directory on the UNIX platform:

- In UNIX, the interfaces file is called *interfaces.*
- In Windows NT, the interfaces file is called *sql.ini.*

For a complete discussion of the interfaces file, see the Adaptive Server configuration guide for your platform.

Step 2: Create Server Entries in System Tables

Use the system procedure **sp_addserver** to add entries to the *sysservers* table. **sp_addserver** creates entries for the local server and an entry for each remote server that is to be called. The **sp_addserver** syntax is:

```
sp_addserver server_name [,server_class
   [,network_name]]
```

where:

- *server_name* is the name used to identify the server. It must be unique.
- *server_class* is one of the supported server classes. Server classes are defined in Chapter 4, "Server Classes." The default value is *sql_server.* If *server_class* is set to *local*, *network_name* is ignored.
- *network_name* is the server name in the interfaces file. This name may be the same as *server_name*, or it may differ. The *network_name* is sometimes referred to as the *physical name*.

Example

The following examples create entries for the local server named DOCS and for the remote server CTOSDEMO with server class *sql_server.*

```
sp_addserver DOCS, local

sp_addserver CTOSDEMO, sql_server, CTOSDEMO
```

Step 3: Add an Alternate Login and Password

Use the system procedure **sp_addexternlogin** to assign an alternate login name and password to be used when communicating with a remote server. This step is optional. The syntax for **sp_addexternlogin** is:

```
sp_addexternlogin remote_server, login_name,
   remote_name [, remote_password]
```

where:

- *remote_server* is the name of the remote server. The *remote_server* must be known to the local server by an entry in the *master.dbo.sysservers* table.
- *login_name* is an account known to the local server. *login_name* must be represented by an entry in the *master.dbo.syslogins* table. The "sa" account, the "sso" account, and the *login_name* account are the only users authorized to modify remote access for a given local user.
- *remote_name* is an account known to the *remote_server* and must be a valid account on the node where the *remote_server* runs. This is the account used for logging into the *remote_server*.
- *remote_password* is the password for *remote_name*.

Examples

```
sp_addexternlogin FRED, sa, system, sys_pass
```

Allows the local server to gain access to remote server FRED using the remote name "system" and the remote password "sys_pass" on behalf of user "sa".

```
sp_addexternlogin OMNI1012, bobj, jordan, hitchpost
```

Tells the local server that when the login name "bobj" logs in, access to the remote server OMNI1012 is by the remote name "jordan" and the remote password "hitchpost". Only the "bobj" account, the "sa" account, and the "sso" account have the authority to add or modify a remote login for the login name "bobj".

Verifying Connectivity

Use the **connect to** *server_name* command to verify that the configuration is correct. **connect to** requires that "sa" explicitly grant connect authority to users other than "sa." The **connect to** command establishes a passthrough mode connection to the remote server. This passthrough mode remains in effect until you issue a **disconnect** command.

Mapping Remote Objects to Local Proxy Tables

Location transparency of remote data is enabled through remote object mapping.

Once a remote server has been properly configured, users can reference the remote objects that have been defined. Users can create new tables on remote servers and can define the schema for an existing table on a remote server.

Overview of the Procedure

1. Use the stored procedure **sp_addobjectdef** to define the storage location of a remote object.
2. Use the **create table** or the **create existing table** command to map the remote table schema to the server.

Step 1: Define the Storage Location of a Remote Object

The stored procedure **sp_addobjectdef** defines the storage location of a remote object. This procedure allows the user to associate a remote object name with a local table name. The remote object may or may not exist before the storage location is defined. The syntax for **sp_addobjectdef** is:

```
sp_addobjectdef object_name, "object_loc"
   [,"object_type"]
```

where:

- *object_name* is the local proxy table name to be used by subsequent statements. *object_name* takes the form:

  ```
  dbname.owner.object
  ```

  where *dbname* and *owner* are optional and represent the local database and owner name. If not present, the object is defined in the current database owned by the current owner. If either *dbname* or *owner* is specified, the entire *object_name* must be enclosed in quotes. If only *dbname* is present, a placeholder is required for *owner.*

- *object_loc* is the storage location of the remote object. It takes the form:

```
server_name.dbname.owner.object;aux1.aux2
```

where:

- *server_name* is the name of the server that contains this remote object (required.)
- *dbname* is the name of the database managed by the remote server that contains this object (optional). If the server is class *db2*, this is the *location_name* portion of a DB2 table name.
- *owner* is the name of the remote server user that owns the remote object (optional). If the server is class *db2*, this is the DB2 authorization ID.
- *object* is the name of the remote *table*, *view*, or *rpc*.
- *aux1.aux2* is a string of characters that is passed to the remote server during a **create table** or **create index** command as the segment name; the meaning of this string is dependent upon the class of the server that receives it. If the server is class *db2*, *aux1* is the DB2 database in which to place the table, and *aux2* is the DB2 tablespace in which to place the table. *aux1.aux2* is optional.

• *object_type* is the type of remote object. It can be *table*, *view*, or *rpc*. This parameter is optional; the default is *table*. When present, the *object_type* option must be enclosed in quotes.

Example

To map the proxy table *authors* to the remote *authors* table, use the following syntax for the database shown in Figure 3-1:

```
sp_addobjectdef authors, "ORACLEDC...authors", "table"
```

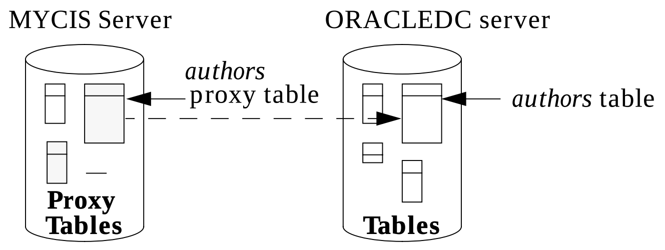


Figure 3-1: Using sp_addobjectdef to map a remote table to a proxy table

Step 2: Map Remote Table Schema to Adaptive Server

Once you have defined the storage location, you can create the table as a new object or as an existing object. If the table does not exist at the remote storage location, use the **create table** syntax. If it already exists, use the **create existing table** syntax. If the object type is *rpc*, only the **create existing table** syntax is allowed.

When a **create existing table** statement is received and the object type is either *table* or *view*, the existence of the remote object is checked using the catalog stored procedure **sp_tables**.

If the object exists, column and index attributes are obtained and compared with those defined for the object in the **create existing table** command. The server checks the column name, type, length and null property and adds index attributes to the *sysindexes* system table.

Once the object has been created, either as a new or existing object, users can query the remote object by using the local proxy name.

See **create table** and **create index in** the *Adaptive Server Reference Manual*.

Join Between Two Remote Tables

With Component Integration Services, you can perform joins across remote tables. The following steps show how to join two Adaptive Server tables:

Overview of the Procedure

1. Add the remote servers to the interfaces file.
2. Define each remote server using **sp_addserver.**
3. Define each remote object using **sp_addobjectdef**.
4. Map the remote tables to the server using **create existing table**.
5. Perform the join using **select**.

Step 1: Add the Remote Servers to the Interfaces File

Edit the interfaces file using the **dsedit** utility.

Step 2: Define the Remote Servers

Use the system procedure **sp_addserver** to add entries to the *sysservers* system table. On the server originating the call, there must be an entry for each remote server that is to be called. The **sp_addserver** syntax is:

```
sp_addserver server_name [,server_class]
   [,network_name]
```

where:

- *server_name* is the name used to identify the server. It must be unique.
- *server_class* is one of the supported server classes, defined in Chapter 4, "Server Classes." The default value is *sql_server*. If the value is *local*, *network_name* is ignored.
- *network_name* is the server name in the interfaces file. This name may be the same as the *server_name* specification, or it may be different. If *network_name* is not provided, the default value is the *server_name*.

Example

The following examples create entries for the local server named DOCS and for the remote server SYBASE of class *sql_server*.

```
sp_addserver DOCS, local
sp_addserver CTOSDEMO, sql_server, SYBASE
```

Step 3: Define the Remote Objects

Use **sp_addobjectdef** to map a local object to an external storage location.

The syntax for the **sp_addobjectdef** procedure is as follows:

```
sp_addobjectdef object_name, "object_loc"
[,"object_type"]
```

The *object_name* argument identifies a table that does not yet exist, but is about to be created. *object_name* takes the form:

```
dbname.owner.object
```

dbname and *owner* are optional. Only the System Administrator may use an owner name other than his or her own. The *object_loc* identifies the storage location of the remote object and takes one of two forms, depending on the value of *object_type*. It takes the form:

```
server_name.dbname.owner.object
```

server_name and *object* are required. *dbname* and *owner* are optional.

Example:

```
sp_addobjectdef accounts,
"SYBASE.pubs.dbo.accounts", "table"
```

Maps the table a*ccounts* to the remote object *pubs.dbo.accounts* in the remote server named SYBASE.

See the *Adaptive Server Reference Manual* for a complete discussion of **sp_addobjectdef**.

Step 4: Map the Remote Tables to Adaptive Server

The **create existing table** command enables the definition of existing (proxy) tables. The syntax for this option is similar to the **create table** command and reads as follows:

```
create existing table table_name (column_list)
   [ on segment_name ]
```

When the server processes this command, it does not create a new table. Instead, it checks the table mapping and verifies the existence of the underlying object. If the object does not exist (either host data file or remote server object), the server rejects the command and returns an error message to the client.

After you define an existing table, it is good practice to issue an **update statistics** command for that table. This helps the query optimizer make intelligent choices regarding index selection and join order.

Example

Figure 3-2 illustrates the remote Adaptive Server tables *publishers* and *titles* in the sample *pubs2* database mapped to a local server.

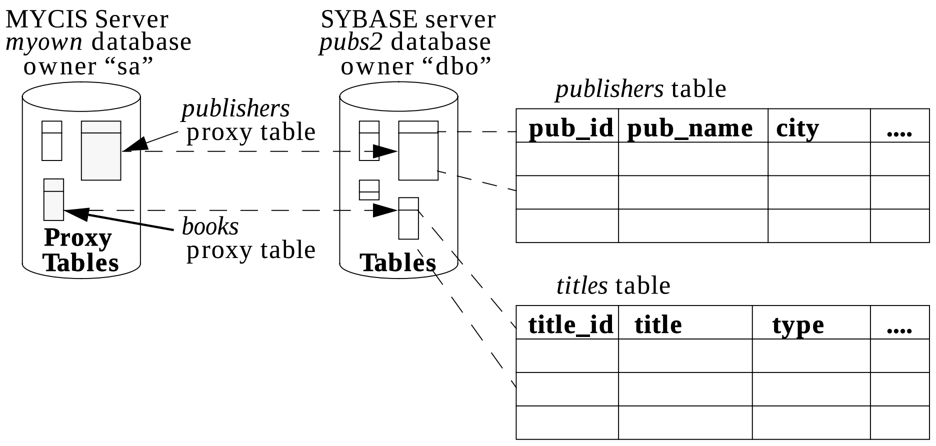


Figure 3-2: Defining remote tables in a local server

Mapping the Remote Tables

The steps required to produce the mapping illustrated above are as follows:

1. Define a server named SYBASE. Its server class is *sql_server*, and its name in the interfaces file is SYBASE:

```
exec sp_addserver SYBASE, sql_server, SYBASE
```

2. Define a remote login alias. This step is optional. User "sa" is known to remote server SYBASE as user "sa," password "timothy":

```
exec sp_addexternlogin SYBASE, sa, sa, timothy
```

3. Add an object definition for the remote *publishers* table:

```
exec sp_addobjectdef publishers,
"SYBASE.pubs2.dbo.publishers", "table"
```

4. Add an object definition for the remote *titles* table, to be known locally as *books*:

```
exec sp_addobjectdef books,
"SYBASE.pubs2.dbo.titles", "table"
```

5. Define the remote *publishers* table:

```
create existing table publishers
(
    pub_id      char(4)     not null,
    pub_name    varchar(40) null,
    city        varchar(20) null,
    state       char(2)     null
)
```

6. Define the remote *titles* table:

```
create existing table books
(
    title_id    tid         not null,
    title       varchar(80) not null,
    type        char(12)    not null,
    pub_id      char(4)     null,
    price       money       null,
    advance     money       null,
    total_sales int         null,
    notes       varchar(200)null,
    pubdate     datetime    not null,
    contract    bit         not null
)
```

7. Update statistics in both tables to ensure reasonable choices by the query optimizer:

```
update statistics publishers
```

```
update statistics books
```

Step 5: Perform the Join

Use the **select** statement to perform the join.

```
select Publisher = p.pubname, Title = b.title
from publishers p, books b
where p.pub_id = b.pub_id
order by p.pubname
```

Configuration and Tuning

This section is intended for System Administrators. It provides information about configuration, tuning, trace flags, backup and recovery, and security issues.

The System Administrator or database owner may elect to use the server in such a way as to optimize performance or to allow use by a required number of clients. Configuration choices might involve being able to review total numbers of reads and writes for a given SQL command.

Once an application is up and running, the System Administrator should monitor performance and may choose to customize and fine-tune the system. The server provides tools for these purposes. This section explains:

- Changing system parameters with the **sp_configure** procedure
- Using **update statistics** to ensure that Component Integration Services makes the best use of existing indexes
- Monitoring server activity with the **dbcc** command.
- Setting trace flags
- Executing **ddlgen** and related backup and recovery issues
- Determining database size requirements

Using *sp_configure*

The configuration parameters in the **sp_configure** system procedure control resource allocation and performance. The System Administrator can reset these configuration parameters in order to tune performance and redefine storage allocation. In the absence of intervention by the System Administrator, the server supplies default values for all the parameters.

The procedure for resetting configuration parameters is:

- Execute the system procedure **sp_configure,** which updates the values field of the system table *master..sysconfigures*.
- Restart the server if you have reset any of the static configuration parameters. The parameters listed below are dynamic; all others are static:

  cis rpc handling

  cis cursor rows

  cis connect timeout

  cis bulk insert batch size

  cis packet size

sysconfigures Table

The *master..sysconfigures* system table stores all configuration options. It contains columns identifying the minimum and maximum values possible for each configuration parameter, as well as the configured value and run value for each parameter.

The *status* column in *sysconfigures* cannot be updated by the user. Status 1 means dynamic, indicating that new values for these configuration parameters take effect immediately. The rest of the configuration parameters (those with status 0) take effect only after the **reconfigure** command has been issued and the server restarted.

You can display the configuration parameters currently in use (run values) by executing the system procedure **sp_configure** without giving it any parameters.

Changing the Configuration Parameters

The stored procedure **sp_configure** displays all the configuration values when it is used without an argument. When used with an option name and a value, the server resets the configuration value of that option in the system tables.

See the *System Administration Guide* for a complete discussion of **sp_configure** with syntax options.

To see the Component Integration Services options enter:

```
sp_configure "Component Integration Services"
```

To change the current value of a configuration parameter, execute **sp_configure** as follows:

```
sp_configure "parameter", value
```

Component Integration Services Configuration Parameters

The following configuration parameters are unique to Component Integration Services:

- **enable cis**
- **max cis remote connections**
- **max cis remote servers**
- **cis bulk insert batch size**
- **cis connect timeout**
- **cis cursor rows**

- **cis packet size**
- **cis rpc handling**

enable cis

Use this parameter with **sp_configure** to enable Component Integration Services as follows:

1. Log into Adaptive Server as the System Administrator and issue the following command:

   ```
   sp_configure "enable cis", 1
   ```

2. Restart Adaptive Server.

Issuing the command **sp_configure "enable cis", 0** disables Component Integration Services after restarting the server.

max cis remote connections

The server establishes Client-Library connections to remote servers on behalf of clients. More than one connection per client may be required if multiple servers are being accessed by that client. By default, Component Integration Services allows up to 4 connections per user to be made simultaneously to remote servers. For example, if you set the maximum number of users to 25, up to 100 simultaneous Client-Library connections are allowed by Component Integration Services.

If this number does not meet the needs of your installation, you can override the setting by specifying how many outgoing Client-Library connections you want the server to be able to make at one time.

max cis remote servers

This configuration parameter allows you to specify how many concurrent servers can be accessed from within the server using Client-Library connections. The default is 25.

cis bulk insert batch size

This configuration parameter determines how many rows from the source table(s) are to be bulk copied into the target table as a single batch using **select into**, when the target table resides in an Adaptive Server or in a DirectConnect server that supports a bulk copy interface.

If left at zero (the default), all rows are copied as a single batch. Otherwise, after the count of rows specified by this parameter has been copied to the target table, the server issues a bulk commit to the target server, causing the batch to be committed.

If a normal client-generated bulk copy operation (such as that produced by the **bcp** utility) is received, the client is expected to control the size of the bulk batch, and the server ignores the value of this configuration parameter.

cis connect timeout

This configuration parameter determines the wait time in seconds for a successful Client-Library connection. By default, no timeout is provided.

cis cursor rows

This configuration parameter allows users to specify the cursor row count for **cursor open** and **cursor fetch** operations. Increasing this value means more rows will be fetched in one operation. This increases speed but requires more memory. The default is 50.

cis packet size

This configuration parameter allows you to specify the size of Tabular Data Stream™ (TDS) packets that are exchanged between the server and a remote server when connection is initiated.

The default packet size on most systems is 512 bytes, which is adequate for most applications. However, larger packet sizes may result in significantly improved query performance, especially when *text* and *image* or bulk data is involved.

If a packet size larger than the default is specified, and the requested server is release 10.0 or later, then the target server must be configured to allow variable-length packet sizes. Adaptive Server configuration parameters of interest in this case are:

- **additional netmem**
- **maximum network packet size**

Refer to the *System Administration Guide* for a complete explanation of these configuration parameters.

cis rpc handling

This global configuration parameter determines whether Component Integration Services will handle outbound RPC requests by default. When this is enabled using **sp_configure "cis rpc handling" 1**, all outbound RPCs are handled by Component Integration Services. When you use **sp_configure "cis rpc handling" 0**, the Adaptive Server site handler is used. The thread cannot override it with **set cis_rpc_handling on**. If the global property is disabled, a thread can enable or disable the capability, as required.

For more information on using the Adaptive Server site handler vs. using Component Integration Services to handle outbound RPCs, see "RPC Handling and Component Integration Services" on page 3-15.

RPC Handling and Component Integration Services

When Component Integration Services is enabled, you can choose between the site handler or Component Integration Services to handle outbound remote procedure calls (RPCs). Each of these mechanisms is described in the following sections.

Site Handler Handling Outbound RPCs

Within an Adaptive Server, outgoing RPCs are transmitted by means of a site handler, which multiplexes multiple requests through a single physical connection to a remote server. The RPC is handled as part of a multistep operation:

1. Establish connection – The Adaptive Server site handler establishes a single physical connection to the remote server. Each RPC requires that a logical connection be established over this physical connection. The logical connection is routed through the site handler of the intended remote server.

   The connection validation process for these connect requests is different than that of normal client connections. First, the remote server must determine if the server from which the connect request originated is configured in its *sysservers* table. If so, then the system table *sysremotelogins* is checked to determine how the connect request should be handled. If trusted mode is configured, password checking is not performed. (For more information about trusted mode, see "Trusted Mode" on page 2-32.)

2. Transmit the RPC – The RPC request is transmitted over the logical connection.
3. Process results – All results from the RPC are relayed from the logical connection to the client.
4. Disconnect – The logical connection is terminated.

Because of the logical connect and disconnect steps, site handler RPCs can be slow.

Component Integration Services Handling Outbound RPCs

If Component Integration Services has been enabled, a client can use one of two methods to request that Component Integration Services handle outbound RPC requests:

- Configure Component Integration Services to handle outbound RPCs as the default for all clients by issuing:

  ```
  sp_configure "cis rpc handling", 1
  ```

  If you use this method to set the **cis rpc handling** configuration parameter, all client connections inherit this behavior, and outbound RPC requests are handled by Component Integration Services. The client can, if necessary, revert back to the default Adaptive Server behavior by issuing the command:

  ```
  set cis_rpc_handling off
  ```

- Configure Component Integration Services to handle outbound RPCs for the current connection only by issuing:

  ```
  set cis_rpc_handling on
  ```

  This command enables **cis rpc handling** for the current thread only, and will not affect the behavior of other threads.

When **cis rpc handling** is enabled, outbound RPC requests are not routed through the Adaptive Server’s site handler. Instead, they are routed through Component Integration Services, which uses persistent Client-Library connections to handle the RPC request. Using this mechanism, Component Integration Services handles outbound RPCs as follows:

1. Determines whether the client already has a Client-Library connection to the server in which the RPC is intended. If not, establish one.
2. Sends the RPC to the remote server using Client-Library functions.

3. Relays the results from the remote server back to the client program that issued the RPC using Client-Library functions.

RPCs can be included within a user-defined transaction. In fact, all work performed by Component Integration Services on behalf of its client can be performed within a single connection context. This allows RPCs to be included in a transaction's unit of work, and the work performed by the RPC can be committed or rolled back with the other work performed within the transaction. This transactional RPC capability is supported only when release 10.0 or later Servers or DirectConnect servers are involved.

The side effects of using Component Integration Services to handle outbound RPC requests are as follows:

- Client-Library connections are persistent so that subsequent RPC requests can use the same connection to the remote server. This can result in substantial RPC performance improvements, since the connect and disconnect logic is bypassed for all but the first RPC.
- Work performed by an RPC can be included in a transaction, and is committed or rolled back with the rest of the work performed by the transaction. This transactional RPC behavior is currently supported only when the server receiving the RPC is another Adaptive Server or a DirectConnect which supports transactional RPCs.
- Connect requests appear to a remote server as ordinary client connections. The remote server cannot distinguish the connection from a normal application's connection. This affects the remote server management capabilities of an Adaptive Server, since no verification is performed against *sysremotelogins*, and all connections must have valid Adaptive Server login accounts established prior to the connect request (trusted mode cannot be used in this case). For more information about trusted mode, see "Trusted Mode" on page 2-32.

dbcc Commands

All **dbcc** commands used by Component Integration Services are available with a single **dbcc** entry point.

The syntax for **dbcc cis** is:

```
dbcc cis ("subcommand"[, vararg1, vararg2...])
```

If Component Integration Services is not configured or loaded, the command will result in a run-time error.

The use of the **dbcc cis** command is unrestricted.

dbcc Options

The following **dbcc** options are unique to Component Integration Services.

remcon

remcon displays a list of all remote connections made by all Component Integration Services clients. It takes no arguments.

rusage

rusage returns a report describing the total memory used by each Component Integration Services resource utilizing shared memory. The report describes total configured items, number of items used, number of items available, and total memory used for each resource.

srvdes

srvdes returns a formatted list of all in-memory SRVDES structures, if no argument is provided. If an argument is provided, this command syncs the in-memory version of a SRVDES with information found in *sysservers*. The command takes an optional argument as follows:

```
srvdes, [ srvid ]
```

Trace Flags

The **dbcc traceon** option allows the System Administrator to turn on trace flags within Component Integration Services. Trace flags enable the logging of certain events when they occur within Component Integration Services. Each trace flag is uniquely identified by a number. Some are global to Component Integration Services while others are *spid*-based and affect only the user who enabled the trace flag. **dbcc traceoff** turns off trace flags.

The syntax is:

```
dbcc traceon (traceflag [, traceflag...])
```

Trace flags and their meanings are shown in Table 3-1:

Table 3-1: Component Integration Services trace flags

Trace Flag	Description
11201	Logs client connect events, disconnect events, and attention events. (global)
11202	Logs client language, cursor declare, dynamic prepare, and dynamic execute-immediate text. (global)
11203	Logs client rpc events. (global)
11204	Logs all messages routed to client. (global)
11205	Logs all interaction with remote server. (global)
11206	Not used.
11207	Logs *text* and *image* processing. (global)
11208	Prevents the **create index** and **drop table** statements from being transmitted to a remote server. *sysindexes* is updated anyway. (*spid*)
11209	Instructs **update statistics** to obtain just row counts rather than complete distribution statistics, from a remote table. (*spid*)
11210	Disables Component Integration Services enhanced remote query optimization. (*spid*)
11211	Not used.
11212	Prevents escape on underscores ("_") in table names. (*spid*)
11213	Prevents generation of column and table constraints. (*spid*)
11214	Disables Component Integration Services recovery at start-up. (global)
11215	Sets enhanced remote optimization for servers of class *db2*. (global)
11216	Disables enhanced remote optimization. (*spid*)
11217	Disables enhanced remote optimization. (global)

Using *update statistics*

The **update statistics** command helps the server make the best decisions about which indexes to use when it processes a query, by providing information about the distribution of the key values in the indexes. **update statistics** is **not** automatically run when you create or re-create an index on a table that already contains data. It can be used when a large amount of data in an indexed column has been added, changed, or deleted. The crucial element in the optimization of your queries is the accuracy of the distribution steps. Therefore, if there are significant changes in the key values in your index, rerun **update statistics** on that index.

The syntax is:

```
update statistics table_name [index_name]
```

If you do not specify an index name, the command updates the distribution statistics for all the indexes in the specified table. Giving an index name updates statistics for that index only.

Try to run **update statistics** at a time when the tables you need to specify are not heavily used. **update statistics** acquires locks on the remote tables and indexes as it reads the data. If trace flag 11209 is used, tables will not be locked.

The server performs a table scan for each index specified in the **update statistics** command.

Since Transact-SQL does not require index names to be unique in a database, you must give the name of the table with which the index is associated.

After running **update statistics**, run **sp_recompile** so that triggers and procedures that use the indexes will use the new distribution:

```
sp_recompile authors
```

Finding Index Names

You can find the names of indexes by using the **sp_helpindex** system procedure. This procedure takes a table name as a parameter.

To list the indexes for the *authors* table, type:

```
sp_helpindex authors
```

To update the statistics for all of the indexes in the table, type:

```
update statistics authors
```

To update the statistics only for the index on the *au_id* column, type:

```
update statistics authors auidind
```

Shared Memory Requirements

When configured, the Component Integration Services shared library draws memory from the shared memory pool initialized by the server during start-up. The amount of memory required by the shared library varies, depending on configuration values. Five resources are managed by the shared library. To view the resources, execute the following **dbcc** command:

```
dbcc cis("rusage")
```

With 25 users, the output might look like this:

```
Resource     Configured  Available   Memory(Bytes)
----------- ----------   ---------   -------------
CIS SRVDES   25          24          4900
CIS DES      500         498         134000
CIS PSS      49          48          2940
CIS RDES     100         100         20800
CIS CURSOR   400         400         49600
```

With 50 users, the output might look like this:

```
Resource     Configured  Available   Memory(Bytes)
----------- ----------   ---------   -------------
CIS SRVDES   25          25          4900
CIS DES      500         500         134000
CIS PSS      74          73          4440
CIS RDES     100         100         20800
CIS CURSOR   800         800         99200
```

The resources are configured as follows:

- CIS SRVDES – configured via the **max cis remote servers** configuration parameter. The default is 25. Each additional SRVDES requires approximately 196 bytes of memory.
- CIS DES – configured indirectly via the **open objects** configuration parameter. For each open object, a CIS DES is allocated. Each CIS DES requires 268 bytes of memory.
- CIS PSS – configured indirectly via the **user connections** configuration parameter. For each user connection, a CIS PSS is allocated. Each CIS PSS requires 60 bytes of memory.
- CIS RDES – configured indirectly via the **max cis remote connections** configuration parameter. There is one CIS RDES for each remote connection.
- CIS CURSOR – configured indirectly via the **user connections** configuration parameter. The number of CIS CURSOR resources is calculated as:

  ```
  4 * user connections
  ```

  Each CIS CURSOR requires 124 bytes of memory.

Additional Component Integration Services Memory Requirements

In addition to the shared memory used by Component Integration Services, dynamic memory which is not accounted for by any configuration value is also used. Dynamic memory is used for:

- The shared library – When Component Integration Services is configured and loaded, the shared library adds approximately 550K to the size of the server executable. This additional memory usage does not appear anywhere, except in operating system commands (for example, the UNIX **ps** command).
- Dynamic Client-Library memory – When connections to remote servers are necessary, Component Integration Services uses Client-Library to establish them. During query and results processing, Client-Library dynamically allocates additional memory and then frees it when the connection ends or statement completes.

Backing Up Your System

There are two ways to back up objects when using Component Integration Services:

- Using the **backup** utility. See the *System Administration Guide* for details.
- Using the **ddlgen** utility. Use **ddlgen** only when you want to back up remote schema information.

Objects Recoverable Through *ddlgen*

The **ddlgen** utility can be used to back up the following objects:

- Servers
- Database definitions
- Logins
- Remote logins
- External logins
- Database options
- Object ownership
- Users and groups

- Aliases
- Rules and defaults
- User datatypes
- Rules and defaults bound to user datatypes
- Table definitions
- Rules and defaults bound to table columns
- Local views
- Triggers
- Procedures
- Permissions

Recovering Component Integration Services Objects

Execute the **ddlgen** utility regularly to ensure that database and schema changes can be restored in the event of a failure. Follow these steps to recover Component Integration Services objects:

1. Create a new installation.
2. Edit the most recent **ddlgen** script to initialize required devices and to update **create database** statements so that objects are created on the correct devices.
3. Apply the modified **ddlgen** script to the new installation through the **isql** utility.

Transaction Log Issues

There is no automatic method for backing up transaction logs associated with Component Integration Services. If the database has only proxy tables, the System Administrator should use the **truncate log on chkpt** option, to truncate the transaction log, every time the server performs an automatic **checkpoint**. Execute this command from the master database:

```
sp_dboption database_name, "trunc log on chkpt",
true
```

This helps avoid filling the device where the transaction log is defined.

Databases with local tables should be backed up using the standard Adaptive Server backup procedures.

Elements Not Restored

ddlgen does not restore local tables. Any local tables should be backed up using the **backup** server.

Additionally, **ddlgen** does not back up the settings of configuration parameters. To back up the configuration file, simply copy it.

4 Server Classes

This chapter provides reference material on the server classes supported by Component Integration Services.

Each server class has a set of unique characteristics that System Administrators and programmers need to know about in order to configure the server for remote data access. These properties are:

- Types of servers that each server class supports
- Datatype conversions specific to the server class
- Restrictions on Transact-SQL statements that apply to the server class

Defining Remote Servers

Use the system procedure **sp_addserver** to add entries to the *sysservers* table for the local server and for each remote server that is to be called. The **sp_addserver** syntax is:

```
sp_addserver server_name [,server_class
   [,network_name]]
```

where:

- *server_name* is the name used to identify the server. It must be unique.
- *server_class* is the type of server. The supported server classes with the types of servers that are in each class are described in the following sections. The default is server class *sql_server.*
- *network_name* is the server name in the interfaces file. This name may be the same as *server_name*, or it may differ. The *network_name* is sometimes referred to as the *physical name*. The default is the same name as *server_name*.

Server Class *sql_server*

A server with server class *sql_server* is:

- SQL Server release 4.9 or later
- SQL Anywhere release 5.5.01 or later
- Microsoft SQL Server version 4.2 or later

- OmniConnect release 10.5 or later
- OmniSQL Server 10.1.2
- Sybase IQ™ version 11.2.1 or later

Server Class *db2*

A server with server class *db2* is an IBM DB2 database accessed through:

- Net-Gateway™ release 3.0 or later (Net-Gateway release 3.01 can also be configured as server class *direct_connect*)
- Database Gateway (Database Gateway release 2.5 or later should be defined as server class *access_server*, not *db2*)
- DB2-DRDA Gateway
- AS / 400
- SQL / DS-DRDA
- InfoHub

Server Class *direct_connect* (*access_server)*

A server with server class *direct_connect* is an Open Server-based application that conforms to the *direct_connect* interface specification. Server class *access_server is* synonymous with server class *direct_connect*. It is used for compatibility with previous releases.

Open Server-based applications using server class *direct_connect* are the preferred means of accessing all external, non-Sybase data sources.

Figure 4-1 illustrates the manner in which Adaptive Server with Component Integration Services enabled interacts with clients and

Open Server-based applications. The data sources are not limited to those in this diagram:

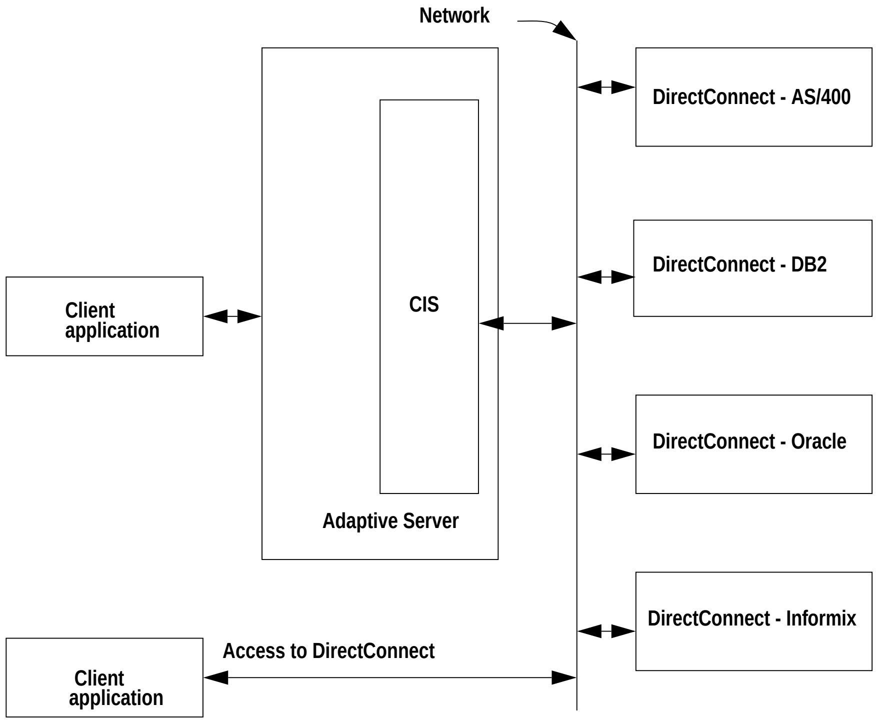


Figure 4-1: **Adaptive Server with CIS interacts with clients and other servers**

Server Class *sds*

A server with server class *sds* conforms to the interface requirements of a Specialty Data Store™ as described in the *Adaptive Server Specialty Data Store Developer's Kit* manual. A Specialty Data Store is an Open Server application you design to interface with Adaptive Server.

In this release, server class *sds* is synonymous to server class *direct_connect*.

Server Class *generic*

The server class *generic* allows customers to build their own Open Server applications to communicate with the server. The customer-built application must conform to the interface described in the *OmniSQL Server Generic Access Module Reference Manual.* The OmniSQL Server™ Rdb Access Module and OmniSQL Server™ Informix Access Module are written to this specification.

The server class *generic* is supported for compatibility with existing Open Server applications. Server class *sds* replaces server class *generic*. New Open Server applications that are compatible with Component Integration Services should follow the interface spcification in the *Adaptive Server Specialty Data Store Developer's Kit* manual.

Datatype Conversions

Datatype conversion can take place whenever the server receives data from a remote source, be it DB2, Adaptive Server, or an Open Server-based application.

Depending on the remote datatype of each column, data is converted from the native datatype on the remote server to a form that the local server supports.

Datatype conversions are made when the **create table, alter table** and **create existing table** commands are processed. The datatype conversions are dependent on the server's server class. See the **create table, alter table** and **create existing table** commands in the following reference pages for tables that illustrate the datatype conversions that take place for each server class when the commands are processed.

Remote Server Capabilities

The first time Adaptive Server establishes a connection to a remote server of class *sds*, *direct_connect*, or *access_server,* it issues an RPC named **sp_capabilities** and expects a set of results in return. This result set describes functional capabilities of the remote server so that Component Integration Services can adjust its interaction with that remote server to take advantage of available features. Component Integration Services forwards as much syntax as possible to a remote server, according to its capabilities.

Transact-SQL Commands

The following pages are reference pages, presented in alphabetical order, which discuss Transact-SQL commands that either directly or indirectly affect external tables, and, as a result, Component Integration Services. For each command, a description of its effect on Component Integration Services, and the manner in which Component Integration Services processes the command, is described. For a complete description of each command, see the *Adaptive Server Reference Manual*.

If Component Integration Services does not pass all of a command's syntax to a remote server (such as all clauses of a **select** statement), the syntax that is passed along is described for each server class.

Each command has several sections that describe it:

Function - contains a brief description of the command.

Syntax - contains a description of the full Transact-SQL syntax of the command.

Comments - contains a general, server class-independent description of handling by Component Integration Services.

Server Class *sql_server* - contains a description of handling specific to server class *sql_server*. This includes syntax that is forwarded to a remote server of class *sql_server*.

Server Class *direct_connect* - contains a description of handling specific to server class *direct_connect* (*access_server*). This includes syntax that is forwarded to a remote server of class *direct_connect* (*access_server*). In this release, all comments that apply to server class *direct_connect*, also apply to server class *sds*.

Server Class *db2* - contains a description of handling specific to server class *db2*. This includes syntax that is forwarded to a remote server of class *db2*.

Server Class *generic* - contains a description of handling specific to server class *generic*. This includes syntax that is forwarded to a remote server of class *generic*.

alter table

Function

Adds new columns to an existing table; adds, changes, or drops constraints on an existing table; partitions or unpartitions an existing table.

Syntax

```
alter table [database.[owner].]table_name
   {add column_name datatype
       [default {constant_expression | user | null}]
       {[{identity | null}]
       | [[constraint constraint_name]
         {{unique | primary key}
           [clustered | nonclustered]
           [with {fillfactor | max_rows_per_page} = x]
                    [on segment_name]
           | references [[database.]owner.]ref_table
              [(ref_column)]
           | check (search_condition)}]}...
       {[, next_column]}...

   | add {[constraint constraint_name]
       {unique | primary key}
         [clustered | nonclustered]
         (column_name [{, column_name}...])
         [with {fillfactor | max_rows_per_page} = x]
              [on segment_name]
      | foreign key (column_name [{, column_name}...])
           references [[database.]owner.]ref_table
               [(ref_column [{, ref_column}...])]
       | check (search_condition)}

   | drop constraint constraint_name

   | replace column_name
       default {constant_expression | user | null}

   | partition number_of_partitions

   | unpartition}
```

Comments

- Component Integration Services processes the **alter table** command when the table on which it operates has been created as a proxy table. Component Integration Services forwards the request (or part of it) to the server that owns the actual object.

- Only the **add** ***column_name*** syntax is forwarded to a remote location. Adding and dropping constraints are local operations, which are not processed by Component Integration Services.
- When Component Integration Services forwards the **alter table** command to a remote server, the table name used is the remote table name, and the column names used are the remote column names. These names may not be the same as the local proxy table names.

Server Class *sql_server*

- Component Integration Services forwards the following syntax to a server configured as class *sql_server*:

```
alter table [database.[owner].]table_name
add column_name datatype [{identity | null}]
       {[, next_column]}...
```

- When a user defines a column with the **alter table** command, Component Integration Services passes the datatype of each column to the remote server without conversions.

Server Class *direct_connect*

- Component Integration Services forwards the following syntax to a remote server configured as class *direct_connect*:

```
alter table [database.[owner].]table_name
add column_name datatype [{identity | null}]
       {[, next_column]}...
```

- Although Component Integration Services requests a capabilities response from a server with class *direct_connect*, support for **alter table** is not optional. Component Integration Services forwards the **alter table** command to the remote server regardless of the capabilities response.
- The behavior of the server with class *direct_connect* is database dependent.
- If the syntax capability of the remote server indicates Sybase Transact-SQL, Adaptive Server datatypes are sent to the remote

server. If the syntax capability indicates DB2 SQL, DB2 datatypes are sent. The mapping for these datatypes is shown in Table 4-1.

Table 4-1: DirectConnect datatype conversions for alter table

Adaptive Server Datatype	DirectConnect Default Datatype	DirectConnect DB2 Syntax Mode Datatype
binary(n)	*binary(n)*	*char(n)* for *bit* data
bit	*bit*	*char*(1)
char	*char*	*char*
datetime	*datetime*	*timestamp*
decimal(p, s)	*decimal(p, s)*	*decimal(p, s)*
float	*float*	*float*
image	*image*	*varchar(n)* for *bit* data; the value of *n* is determined by the global variable *@@textsize*
int	*int*	*int*
money	*money*	*float*
numeric(p, s)	*numeric(p, s)*	*decimal(p, s)*
nchar(n)	*nchar(n)*	*graphic(n)*
nvarchar(n)	*nvarchar(n)*	*vargraphic(n)*
real	*real*	*real*
smalldatetime	*smalldatetime*	*timestamp*
smallint	*smallint*	*smallint*
smallmoney	*smallmoney*	*float*
timestamp	*timestamp*	*varbinary*(8)
tinyint	*tinyint*	*smallint*
text	*text*	*varchar(n)*; the value of *n* is determined by the global variable *@@textsize*
varbinary(n)	*varbinary(n)*	*varchar(n)* for *bit* data
varchar(n)	*varchar(n)*	*varchar(n)*

Server Class *db2*

- Component Integration Services forwards the following syntax to a remote server configured as class *db2*:

```
alter table [database.[owner].]table_name
add column_name datatype [null]
      {[, next_column]}...
```

- *text* and *image* datatypes are not supported by server class *db2*. If *text* and *image* datatypes are used, Component Integration Services raises Error 11205:

```
Datatype <typename> is unsupported for server
<servername>.
```

The datatype specification contains DB2 datatypes that are mapped from Adaptive Server datatypes. The datatype conversions are shown in Table 4-2.

Table 4-2: DB2 datatype conversions for alter table

Adaptive Server Datatype	DB2 Datatype
binary(n)	*char(n)* for *bit* data, where *n* <= 254
bit	*char(1)*
char(n)	*char(n)*, where *n* <= 254
datetime	*timestamp*
decimal(p, s)	*decimal(p, s)*
float	*float*
image	Not supported
int	*int*
money	*float*
nchar	*char(n)*
nvarchar	*varchar(n)*
numeric(p, s)	*decimal(p, s)*
real	*real*
smalldatetime	*timestamp*
smallint	*smallint*
smallmoney	*float*
tinyint	*smallint*
text	Not supported

Table 4-2: DB2 datatype conversions for alter table (continued)

Adaptive Server Datatype	DB2 Datatype
varbinary(n)	*varchar(n)* for *bit* data, where *n* <=254
varchar(n)	*varchar(n)*, where *n* <= 254

Server Class *generic*

- Component Integration Services forwards the following syntax to a server with server class *generic*, unless a *text*, *image*, *decimal*, or *numeric* datatypes is specified:

  ```
  alter table [database.[owner].]table_name
  add column_name datatype [{identity | null}]
         {[, next_column]}...
  ```

  If a *text*, *image*, *decimal*, or *numeric* datatype is used, Component Integration Services raises Error 11205:

  ```
  Datatype <typename> is unsupported for server
  <servername>.
  ```

- When a user defines a column with the **alter table** command, a datatype must be provided. The server passes the datatype name of each column to the Generic Access Module without conversion.

See Also

alter table in the *Adaptive Server Reference Manual*.

begin transaction

Function

Marks the starting point of a user-defined transaction.

Syntax

```
begin tran[saction] [transaction_name]
```

Comments

- When Adaptive Server receives a **begin transaction** command, an internal state is set which marks the beginning of a transaction. At this point, Component Integration Services is not involved, and the command is not immediately forwarded to remote locations.
- *transaction_name* is not used by Component Integration Services in this release.

Server Class *sql_server*

- Component Integration Services checks the transaction state of the connection to a server of class *sql_server*. If the internal transaction state indicates that a transaction is in progress, and the state of the connection to the server indicates that no transaction is in progress, Component Integration Services forwards the **begin transaction** command to the server prior to forwarding the first command to that server. In the example below, assume tables *t1* and *t2* are both located on the same remote SQL Server:

  ```
  begin transaction
      insert into t1 values (...)
      update t2 ...
  commit transaction
  ```

  At the time the **begin transaction** command is processed, no interaction with the remote SQL Server occurs.

  When the **insert** command is processed, the transaction state of the connection to the server that owns *t1* is checked. Since this is the first command within the transaction, the connection is in a NO TRANSACTION ACTIVE state, and the **begin transaction** command is forwarded to the server. The **insert** command is then forwarded to the remote location, and the transaction state for the connection is marked as TRANSACTION ACTIVE.

When processing the **update** command, the transaction state of the server that owns table *t2* is checked. Since it is the same server that owns table *t1*, it is in the TRANSACTION ACTIVE state, and the **begin transaction** command is not forwarded.

➤ ***Note***

These comments apply only to release 10.0 or later, which supports cursors. For pre-release 10.0 SQL Server and Microsoft SQL Server, transaction handling is similar to server class *db2*, described below.

Server Class *direct_connect*

- Transaction processing for servers in class *direct_connect* is identical to that of server class *sql_server* (release 10.0 or later).

Server Class *db2*

- Transactions are supported only at the statement level for servers in class *db2*. When the internal state of a client connection indicates that there is an active transaction, Component Integration Services precedes each statement forwarded to the server with a **begin transaction** command. Component Integration Services then issues a **commit** or **rollback transaction** (depending on the success or failure of the statement) immediately after the statement is complete.

Server Class *generic*

- Transactions are supported only at the statement level for servers in class *generic*. When the internal state of a client connection indicates that there is an active transaction, Component Integration Services precedes each statement forwarded to the server with the RPC **gen_begin_xact**. It then issues a **gen_commit_xact** or **gen_rollback_xact** RPC (depending on the success or failure of the statement) immediately after the statement is complete. Each statement executes completely or not at all.

See Also

begin transaction in the *Adaptive Server Reference Manual*.

close

Function

Deactivates a cursor.

Syntax

```
close cursor_name
```

Comments

- If the cursor specified by *cursor_name* contains references to proxy tables, Adaptive Server notifies Component Integration Services to close and deallocate its remote cursors for those tables.
- Component Integration Services uses Client-Library to manage cursor operations to a remote server. When Component Integration Services receives a **close** command, it uses the following Client-Library functions to interact with the remote server:

```
ct_cursor(command, CS_CURSOR_CLOSE, NULL,
   CS_UNUSED, NULL, CS_UNUSED, CS_UNUSED)
ct_cursor(command, CS_CURSOR_DEALLOC, NULL,
   CS_UNUSED, NULL, CS_UNUSED, CS_UNUSED)
```

- If the cursor contains references to more than one proxy table, Component Integration Services must close a remote cursor for each server represented by the proxy tables.

See Also

deallocate cursor, **declare cursor**, **fetch**, **open** in this chapter.

close in the *Adaptive Server Reference Manual*.

commit transaction

Function

Marks the successful ending point of a user-defined transaction.

Syntax

```
commit [tran[saction] | work] [transaction_name]
```

Comments

- When Adaptive Server receives the **commit transaction** command, it notifies Component Integration Services, and Component Integration Services attempts to commit work associated with remote servers involved in the current transaction.
- Multiple remote servers can be involved in a single transaction, each with their own unit of work which is associated with the Adaptive Server unit of work.
- Remote work is committed before local work. If the remote servers do not respond, or respond with errors, the transaction is aborted, including any local work.
- Work performed by transactional RPC's must be part of an explicit transaction.
- *transaction_name* is not used by Component Integration Services in this release.

Server Class *sql_server*

- When Component Integration Services receives notification to commit a transaction, it checks the TRANSACTION ACTIVE state of all remote connections associated with the client application. If there is more than one remote server involved in a transaction, Component Integration Services first sends a **prepare transaction** command to each connection with an active transaction. If all remote servers respond with no error, Component Integration Services sends a **commit transaction** command to each server involved in the transaction. If all remote servers again respond with no error, Component Integration Services notifies the Adaptive Server that it can commit local work.

  This process applies to release 10.0 or later. Transaction handling is the same as server class *db2*, described below, if the server represented by server class *sql_server* is:

- Pre-release 10.0 SQL Server
- Microsoft SQL Server (any version)
- Sybase IQ
- OmniConnect 10.1.2

Server Class *direct_connect*

- Transaction processing for servers in class *direct_connect* is identical to that of server class *sql_server* (release 10.0 or later).

Server Class *db2*

- Transactions are supported only at the statement level for servers in class *db2*. When the internal state of a client connection indicates that there is an active transaction, a **begin transaction** command precedes all **insert**, **update** and **delete** commands. Component Integration Services issues a **commit** or **rollback transaction** (depending on the success or failure of the statement) immediately after the statement is complete.

Server Class *generic*

- Transactions are supported only at the statement level for servers in class *generic*. When the internal state of a client connection indicates that there is an active transaction, the RPC **gen_begin_xact** precedes all **insert**, **update** and **delete** commands. Component Integration Services issues a **gen_commit_xact** or **gen_rollback_xact** RPC (depending on the success or failure of the statement) immediately after the statement is complete.

See Also

commit in the *Adaptive Server Reference Manual*.

create existing table

Function

Creates a new proxy table representing an existing object in a remote server.

Syntax

```
create existing table [database.[owner].]table_name
  (column_name datatype
      [default {constant_expression | user | null}]
      {[{identity | null | not null}]
      | [[constraint constraint_name]
          {{unique | primary key}
           [clustered | nonclustered]
           [with {fillfactor |max_rows_per_page}= x]
          [on segment_name]
          | references [[database.]owner.]ref_table
              [(ref_column)]
          | check (search_condition)}]}...

  | [constraint constraint_name]
      {{unique | primary key}
          [clustered | nonclustered]
          (column_name [{, column_name}...])
           [with {fillfactor |max_rows_per_page}= x]
           [on segment_name]
      | foreign key (column_name [{,
  column_name}...])
          references [[database.]owner.]ref_table
              [(ref_column [{, ref_column}...])]
       | check (search_condition)}

  [{, {next_column | next_constraint}}...])

  [with max_rows_per_page = x] [on segment_name]
```

Comments

- Adaptive Server processes the **create existing table** command as if the table being created is a new local table.
- After creating the local table, Adaptive Server passes the **create existing table** command to Component Integration Services, with the external location for the existing remote object.

Component Integration Services verifies that the table exists by issuing the **sp_tables** RPC to the remote server that owns the existing object.

- Component Integration Services verifies the column list by sending the **sp_columns** RPC to the remote server. Column names, datatypes, lengths, identity property, and null properties are checked for the following:
    - Datatypes in the **create existing table** command must match or be convertible to the datatypes of the column on the remote location. For example, a local column datatype might be defined as *money*, while the remote column datatype might be *numeric*. This is a legal conversion, therefore, no errors are reported.
    - Each column's null property is checked. If the local column's null property is not identical to the remote column's null property, a warning message is issued, but the command is not aborted.
    - Each column's length is checked. If the length of *char*, *varchar*, *binary*, *varbinary*, *decimal* and *numeric* columns do not match, a warning message is issued, but the command is not aborted.
- For each server class, different rules apply to column name matching. These rules are described in the following sections.

Server Class *sql_server*

- Column names are not checked, but are compared by position (column ID).
- The proxy table must contain the exact number of columns as found in the remote table, or a column count mismatch error is issued, and the command is aborted.
- Column names do not need to be identical. The remote column name is stored in *syscolumns.remote_name* and is used during query processing when a statement is forwarded to the remote server.
- Column datatypes do not need to be identical, but they must be convertible in both directions, or a column datatype error is raised, and the command is aborted.

- Table 4-3 describes the allowable datatypes that can be used when mapping remote Adaptive Server columns to local proxy table columns:

Table 4-3: Adaptive Server datatype conversions for create existing table

Remote Adaptive Server Datatype	Allowable Adaptive Server Datatypes
binary(*n*)	*image, binary*(*n*), and *varbinary*(*n*); if not *image*, the length must match
bit	*bit*, *decimal*, *float*, *int*, *money*, *numeric*, *real*, *smallint*, *smallmoney*, and *tinyint*
char(*n*)	*text, nchar*(*n*), *nvarchar*(*n*), *char*(*n*), *varchar*(*n*); if not *text*, the length must match
datetime	*datetime* and *smalldatetime*
decimal(p, s)	*bit*, *decimal*, *float*, *int*, *money*, *numeric*, *real*, *smallint*, *smallmoney*, and *tinyint*
float	*bit*, *decimal*, *float*, *int*, *money*, *numeric*, *real*, *smallint*, *smallmoney*, and *tinyint*
image	*image*
int	*bit*, *decimal*, *float*, *int*, *money*, *numeric*, *real*, *smallint*, *smallmoney*, and *tinyint*
money	*bit*, *decimal*, *float*, *int*, *money*, *numeric*, *real*, *smallint*, *smallmoney*, and *tinyint*
nchar(*n*)	*text, nchar*(*n*), *nvarchar*(*n*), *char*(*n*), *varchar*(*n*); if not *text*, the length must match
numeric(p, s)	*bit*, *decimal*, *float*, *int*, *money*, *numeric*, *real*, *smallint*, *smallmoney*, and *tinyint*
nvarchar(*n*)	*text, nchar*(*n*), *nvarchar*(*n*), *char*(*n*), *varchar*(*n*); if not *text*, the length must match
real	*bit*, *decimal*, *float*, *int*, *money*, *numeric*, *real*, *smallint*, *smallmoney*, and *tinyint*
smalldatetime	*datetime* and *smalldatetime*
smallint	*bit*, *decimal*, *float*, *int*, *money*, *numeric*, *real*, *smallint*, *smallmoney*, and *tinyint*
smallmoney	*bit*, *decimal*, *float*, *int*, *money*, *numeric*, *real*, *smallint*, *smallmoney*, and *tinyint*
text	*text*
timestamp	*timestamp*

Table 4-3: Adaptive Server datatype conversions for create existing table

Remote Adaptive Server Datatype	Allowable Adaptive Server Datatypes
tinyint	*bit*, *decimal*, *float*, *int*, *money*, *numeric*, *real*, *smallint*, *smallmoney*, and *tinyint*
varbinary(*n*)	*image, binary*(*n*), and *varbinary*(*n*); if not *image*, the length must match
varchar(*n*)	*text*, *nchar*(*n*), *nvarchar*(*n*), *char*(*n*), *varchar*(*n*); if not *text*, the length must match

Server Class *direct_connect*

- The RPC **sp_columns** queries the datatypes of the columns in the existing table.
- Column names are not checked, but are compared by position (column ID).
- The proxy table must contain the exact number of columns as found in the remote table, or a column count mismatch error is issued, and the command is aborted.
- Local column datatypes do not need to be identical to remote column datatypes, but they must be convertible as shown in Table 4-4. If not, a column type error is raised, and the command is aborted.

Table 4-4: DirectConnect datatype conversions for create existing table

DirectConnect Datatype	Allowable Adaptive Server Datatypes
binary(*n*)	*image*, *binary*(*n*), *varbinary*(*n*); if the length does not match, the command is aborted
binary(16)	*timestamp*
bit	*bit*, *decimal*, *float*, *int*, *money*, *numeric*, *real*, *smallint*, *smallmoney*, and *tinyint*
char(*n*)	*text*, *nchar*(*n*), *nvarchar*(*n*), *char*(*n*) and *varchar*(*n*); if the length does not match, the command is aborted
datetime	*datetime*, *smalldatetime*
decimal(*p*, *s*)	*bit*, *decimal*, *float*, *int*, *money*, *numeric*, *real*, *smallint*, *smallmoney*, and *tinyint*
float	*bit*, *decimal*, *float*, *int*, *money*, *numeric*, *real*, *smallint*, *smallmoney*, and *tinyint*

Table 4-4: DirectConnect datatype conversions for create existing table

DirectConnect Datatype	Allowable Adaptive Server Datatypes
image	*image*
int	*bit*, *decimal*, *float*, *int*, *money*, *numeric*, *real*, *smallint*, *smallmoney*, and *tinyint*
money	*bit*, *decimal*, *float*, *int*, *money*, *numeric*, *real*, *smallint*, *smallmoney*, and *tinyint*
nchar(*n*)	*text, nchar*(*n*), *nvarchar*(*n*), *char*(*n*) and *varchar*(*n*); if the length does not match, the command is aborted
numeric(p, s)	*bit*, *decimal*, *float*, *int*, *money*, *numeric*, *real*, *smallint*, *smallmoney*, and *tinyint*
nvarchar(*n*)	*text, nchar*(*n*), *nvarchar*(*n*), *char*(*n*) and *varchar*(*n*); if the length does not match, the command is aborted
real	*bit*, *decimal*, *float*, *int*, *money*, *numeric*, *real*, *smallint*, *smallmoney*, and *tinyint*
smalldatetime	*datetime, smalldatetime*
smallint	*bit*, *decimal*, *float*, *int*, *money*, *numeric*, *real*, *smallint*, *smallmoney*, and *tinyint*
smallmoney	*bit*, *decimal*, *float*, *int*, *money*, *numeric*, *real*, *smallint*, *smallmoney*, and *tinyint*
text	*text*
timestamp	*timestamp, binary*(8), *varbinary*(8)

- The column length defined for columns of type *char, varchar, binary,* and *varbinary* must match the length of the corresponding columns in the remote table.
- Scale and precision of columns of type *numeric* or *decimal* must match the scale and precision of the corresponding columns in the remote table.
- If the null property is not identical to the remote column's null property, a warning message is issued, but the command is not aborted.
- Datatype information is passed in the CS_DATAFMT structure associated with the parameter. The following fields of the structure contain datatype information:
  - *datatype* – the CS_Library datatype representing the Adaptive Server datatype. For example, CS_INT_TYPE.

- *usertype* – the native DBMS datatype. **sp_columns** passes this datatype back to Component Integration Services during a **create existing table** command as part of its result set (see **sp_columns** in the *Adaptive Server Reference Manual*). Adaptive Server returns this datatype in the *usertype* field of parameters to assist the DirectConnect in datatype conversions.

Server Class *db2*

- Column names are checked in a case-insensitive manner. If there is no match, a column name error is raised, and the command is aborted.

➤ ***Note***

The Adaptive Server table can contain fewer columns than the remote table, but each column in the Adaptive Server table must have a matching column in the remote table.

- *text* and *image* datatypes are not supported by server class *db2*.
- When a **create existing table** command is processed, the datatype for each column specifies the type of conversion to perform between the DB2 and Adaptive Server datatypes during query processing. Table 4-5 describes the allowable Adaptive Server datatypes that can be used for existing DB2 datatypes:

Table 4-5: DB2 datatype conversions for create existing table

DB2 Datatype	Allowable Adaptive Server Datatypes
int	*int*
smallint	*int*, *smallint*, and *tinyint*; if length does not match, a warning message is issued
tinyint	*int*, *smallint*, and *tinyint*; if length does not match, a warning message is issued
float	*real, float,* and *money*
double precision	*real, float,* and *money*
real	*real, float,* and *money*
decimal(scale > 0)	*float, money, decimal*, and *numeric*; for *decimal* and *numeric*, scale and precision must match
decimal (scale = 0)	*float, money, decimal*, and *numeric*; for *decimal* and *numeric*, scale and precision must match

Table 4-5: DB2 datatype conversions for create existing table (continued)

DB2 Datatype	Allowable Adaptive Server Datatypes
numeric (scale > 0)	*float, money, decimal*, and *numeric*; for *decimal* and *numeric*, scale and precision must match
numeric (scale = 0)	*float, money, decimal*, and *numeric*; for *decimal* and *numeric*, scale and precision must match
char	*char, varchar, bit, binary, varbinary, text* and *image*; if not *text* or *image*, length must match
char(*n*) for *bit* data	*binary*(*n*), *varbinary*(*n*), and *image*; if not *image*, length must match
varchar	*char, varchar, bit, binary, varbinary, text* and *image*; if not *text* or *image*, length must match
varchar(*n*) for *bit* data	*binary*(*n*), *varbinary*(*n*), and *image*; if not *image*, length must match
long varchar (length truncated to 255)	*char, varchar, bit, binary, varbinary, text* and *image*; if not *text* or *image*, length must match
date	*char(10), varchar(10)*, and *datetime* (time set to 12:00AM)
time	*char(8), varchar(8)*, and *datetime* (date set to 1/1/1900)
timestamp	*char(26), varchar(26), datetime*, and *smalldatetime*
graphic	Not supported
vargraphic	Not supported
long vargraphic	Not supported

- If a local column's datatype cannot be converted to the remote column's datatype, a column type error is raised, and the command is aborted.
- If the data contained in a *long varchar* column exceeds 255 bytes, it is truncated, or, if the gateway is so configured, an error is returned.
- DB2 table names are limited to 18 characters.
- DB2 authorization IDs (owner names) are limited to 8 characters.
- The maximum string length for columns returned by DB2 is 254 characters for *char* and *varchar* datatypes. For *long varcha*r, the length is 32,704 bytes.

- DB2 can return date values that are not within the range of the Adaptive Server *datetime* datatype. DB2's range is 0001-01-01 to 9999-12-31. The Adaptive Server's range is 1753-01-01 to 9999-12-31. When a date earlier than 1753-01-01 is retrieved from DB2, it is converted to 1753-01-01.
- Check DB2 documentation for the maximum number of columns per DB2 table. This varies with the DB2 version.

Server Class *generic*

- Column names are checked in a case-insensitive manner. If there is no match, a column name error is raised, and the command is aborted.

➤ ***Note***

The Adaptive Server table can contain fewer columns than the remote table, but each column in the Adaptive Server table must have a matching column in the remote table.

- *text*, *image*, *decimal*, and *numeric* datatypes are not supported by the server class *generic*.
- Table 4-6 illustrates datatype compatibility when the **create existing table** command is processed. When the server encounters a datatype shown in the "ODBC Datatype" column, it allows any of the datatypes shown in the "Allowable Adaptive Server Datatypes" column. When a datatype other than an allowable datatype is encountered, Adaptive Server returns an error message and the **create existing table** command is aborted.

Table 4-6: ODBC datatype conversions for create existing table

ODBC Datatype	Allowable Adaptive Server Datatypes
int	*int*
smallint	*smallint*
tinyint	*tinyint*
float	*float*, *money*, and *smallmoney*
double precision	*float*, *money*, and *smallmoney*
real	*real*, *money*, and *smallmoney*

Table 4-6: ODBC datatype conversions for create existing table (continued)

ODBC Datatype	Allowable Adaptive Server Datatypes
decimal(p,s) (scale less than 0 or precision greater than or equal to 10)	*float, money*
decimal(p,s) (scale equal to 0 or precision greater than or equal to 10	*int, float, money*
numeric(p,s) (scale less than 0 or precision greater than or equal to 10)	*float, money*
numeric(p,s) (scale equal to 0 or precision greater than or equal to 10)	*int, float, money*
char(n)	*char*(*n*), *varchar*(*n*) (*n* truncated to 255 bytes)
long varchar(n), varchar(n)	*char*(*n*), *varchar*(*n*) (*n* truncated to 255 bytes)
date	*datetime* (time set to 12:00AM)
time	*datetime* (date set to 1/1/1900)
timestamp	*datetime*
bit	*bit*
binary(n)	*binary(n), varbinary(n)*; length must match
varbinary(n)	*binary(n), varbinary(n)*; length must match
long varbinary(n)	*binary(255), varbinary(255)*

See Also

create existing table in the *Adaptive Server Reference Manual*.

create index

Function

Creates an index on one or more columns in a table.

Syntax

```
create [unique] [clustered | nonclustered]
      index index_name
   on [[database.]owner.]table_name (column_name
       [, column_name]...)
   [with {{fillfactor | max_rows_per_page} = x,
       ignore_dup_key, sorted_data,
       [ignore_dup_row | allow_dup_row]}]
   [on segment_name]
```

Comments

- Component Integration Services processes the **create index** command when the table involved has been created as a proxy table. The actual table resides on a remote server, and Component Integration Services forwards the request to the remote server after Adaptive Server catalogs are updated to represent the new index.
- Trace flag 11208 changes the behavior of the **create index** command. If trace flag 11208 is turned on, Component Integration Services does not send the **create index** command to the remote server. Instead, Adaptive Server processes the command locally, as if the table on which it operates is local. This is useful for creating an index on a proxy table that maps to a remote view.
- Adaptive Server performs all system catalog updates in order to identify the index. However, just as there are no data pages in the server for proxy tables, there are no index pages.
- When Component Integration Services forwards the **create index** command to a remote server, the table name used is the remote table name, and the column names used are the remote column names. These names may not be the same as the local proxy table names.

Server Class *sql_server*

- Component Integration Services forwards everything except the **on *segment_name*** clause to the remote server.

- For pre-release 10.0 SQL Server or Microsoft SQL Server 6.5, neither the **max_rows_per_page** or **on *segment_name*** clause is forwarded to the remote server.

Server Class *direct_connect*

- When the language capability is set to "Transact-SQL", Component Integration Services forwards all syntax except the **max_rows_per_page** and **on *segment_name*** clauses to the remote server.
- When the language capability is set to "DB2", the behavior is the same as for server class *db2*.
- The DirectConnect must either translate the Sybase extensions to equivalent native syntax or ignore them.

Server Class *db2*

- Component Integration Services does not forward the following clauses to the remote server:
  - **max_rows_per_page**
  - **ignore_dup_key**
  - **ignore_dup_row**
  - **allow_dup_row**
- Component Integration Services converts the **fillfactor** option to **pctfree** and then forwards it to the remote server.

Server Class *generic*

- Component Integration Services forwards all syntax except the **max_rows_per_page** and **on *segment_name*** clauses to the remote server.
- The Generic Access Module must either translate the Sybase extensions to equivalent native syntax or ignore them.

See Also

create index in the *Adaptive Server Reference Manual*.

create table

Function

Creates new tables and optional integrity constraints.

Syntax

```
create table [database.[owner].]table_name
  (column_name datatype
      [default {constant_expression | user | null}]
      {[{identity | null | not null}]
      | [[constraint constraint_name]
          {{unique | primary key}
           [clustered | nonclustered]
           [with {fillfactor |max_rows_per_page}= x]
          [on segment_name]
          | references [[database.]owner.]ref_table
              [(ref_column)]
          | check (search_condition)}]}...

  | [constraint constraint_name]
      {{unique | primary key}
          [clustered | nonclustered]
          (column_name [{, column_name}...])
           [with {fillfactor |max_rows_per_page}= x]
           [on segment_name]
      | foreign key (column_name [{,
  column_name}...])
          references [[database.]owner.]ref_table
              [(ref_column [{, ref_column}...])]
       | check (search_condition)}

  [{, {next_column | next_constraint}}...])

  [with max_rows_per_page = x] [on segment_name]
```

Comments

- If the table being created is mapped to a remote location, a proxy table is created. A proxy table is identical to a local table, except that the *sysobjects.sysstat2* column contains a status flag that indicates the table is mapped to an external location.
- The external location must be previously defined using the **sp_addobjectdef** system procedure.

- After the Adaptive Server processes the **create table** command, it notifies Component Integration Services of the need to forward the command to the remote location (if a location has been previously specified).

  Component Integration Services reconstructs the SQL necessary to create the table, and forwards the SQL to the remote server. It does not forward all the original syntax to the remote server. The following clauses are processed by Adaptive Server:

  - **on *segment name***
  - **check** constraints
  - **default**
  - **with max_rows_per_page**
- Trace flag 11213 changes the behavior of the **create table** command. Referential constraints and unique or primary key constraints are forwarded to the remote server unless trace flag 11213 is turned on, in which case they are processed locally.
- For each column, the column name, datatype, length, identity property, and null property are reconstructed from the original statement.
- Component Integration Services passes a NULL *char* column as a NULL *varchar* column.
- Component Integration Services passes a NULL *binary* column as a NULL *varbinary* column.

Server Class *sql_server*

- When a user defines a column with the **create table** command, Component Integration Services passes the datatype of each column to the remote server without conversions.

Server Class *direct_connect*

- When a user defines a column with the **create table** command, a datatype for the column must be provided. Component Integration Services reconstructs the **create table** command and passes commands to the targeted DirectConnect. The gateway transforms the commands into a form that the underlying DBMS recognizes.
- Some DirectConnects support DB2 syntax mode, which is described in the DirectConnect documentation. When the DirectConnect enables DB2 syntax mode, Component

Integration Services constructs DB2 SQL syntax and converts the column to a datatype DB2 supports.

- Adaptive Server datatypes are converted to either the DirectConnect or DB2 syntax mode datatypes shown in Table 4-7, depending on whether the DirectConnect supports DB2 syntax mode.

Table 4-7: DirectConnect datatype conversions for create table

Adaptive Server Datatype	DirectConnect Default Datatype	DirectConnect DB2 Syntax Mode Datatype
binary(n)	*binary(n)*	*char(n)* for *bit* data
bit	*bit*	*char*(1)
char	*char*	*char*
datetime	*datetime*	*timestamp*
decimal(p, s)	*decimal(p, s)*	*decimal(p, s)*
float	*float*	*float*
image	*image*	*varchar(n)* for *bit* data; the value of *n* is determined by the global variable *@@textsize*
int	*int*	*int*
money	*money*	*float*
numeric(p, s)	*numeric(p, s)*	*decimal(p, s)*
nchar(n)	*nchar(n)*	*graphic(n)*
nvarchar(n)	*nvarchar(n)*	*vargraphic(n)*
real	*real*	*real*
smalldatetime	*smalldatetime*	*timestamp*
smallint	*smallint*	*smallint*
smallmoney	*smallmoney*	*float*
timestamp	*timestamp*	*varbinary*(8)
tinyint	*tinyint*	*smallint*
text	*text*	*varchar(n)*; the value of *n* is determined by the global variable *@@textsize*
varbinary(n)	*varbinary(n)*	*varchar(n)* for *bit* data

Table 4-7: DirectConnect datatype conversions for create table (continued)

Adaptive Server Datatype	DirectConnect Default Datatype	DirectConnect DB2 Syntax Mode Datatype
varchar(n)	*varchar(n)*	*varchar(n)*

Server Class *db2*

Table 4-8 shows the datatype conversions that are performed when a **create table** command is processed. Adaptive Server datatypes are converted to the DB2 datatypes shown.

Table 4-8: DB2 datatype conversions for create table

Adaptive Server Datatype	DB2 Datatype
binary(n)	*char(n)* for *bit* data, where *n* <= 254
bit	*char(1)*
char(n)	*char(n)*, where *n* <= 254
datetime	*timestamp*
decimal(p, s)	*decimal(p, s)*
float	*float*
image	Not supported
int	*int*
money	*float*
nchar	*char(n)*
nvarchar	*varchar(n)*
numeric(p, s)	*decimal(p, s)*
real	*real*
smalldatetime	*timestamp*
smallint	*smallint*
smallmoney	*float*
tinyint	*smallint*
text	Not supported
varbinary(n)	*varchar(n)* for *bit* data, where *n* <=254
varchar(n)	*varchar(n)*, where *n* <= 254

Server Class *generic*

- When you define a column with the **create table** command, a datatype must be provided. The server passes the datatype name of each column to the Generic Access Module without conversion.
- The *generic* server class does not allow *text*, *image*, *decimal* or *numeric* datatypes. Use of these datatypes results in an error.

See Also

create table in the *Adaptive Server Reference Manual*.

deallocate cursor

Function

Makes a cursor inaccessible and releases all memory resources committed to that cursor.

Syntax

```
deallocate cursor cursor_name
```

Comments

- If the cursor specified by *cursor_name* contains references to proxy tables, Adaptive Server notifies Component Integration Services to deallocate its remote cursors for those tables.
- If the remote cursor is not closed, Component Integration Services closes and deallocates it. If the remote cursor is already closed, no additional actions are taken.
- Component Integration Services uses Client-Library to manage cursor operations to a remote server. When Component Integration Services receives a **deallocate cursor** command and the cursor has not been explicitly closed with a **close** command, Component Integration Services uses the following Client-Library functions to interact with the remote server:

```
ct_cursor(command, CS_CURSOR_CLOSE, NULL,
   CS_UNUSED, NULL, CS_UNUSED, CS_UNUSED)
ct_cursor(command, CS_CURSOR_DEALLOC, NULL,
   CS_UNUSED, NULL, CS_UNUSED, CS_UNUSED)
```

- If the cursor contains references to more than one proxy table, Component Integration Services must deallocate a remote cursor for each server represented by the proxy tables.

See Also

close, **declare cursor**, **fetch**, **open** in this chapter.

deallocate cursor in the *Adaptive Server Reference Manual*.

declare cursor

Function

Defines a cursor.

Syntax

```
declare cursor_name cursor
   for select_statement
   [for {read only | update [of column_name_list]}]
```

Comments

- If the cursor specified by *cursor_name* contains references to proxy tables, Adaptive Server notifies Component Integration Services to establish a connection to the remote servers referenced by the proxy tables.

  A separate connection is required for each server represented by all proxy tables. For example, if all proxy tables in the cursor reference the same remote server, only one connection is required while the **declare cursor** command is processed. However, if two or more servers are referenced by the proxy tables, a separate connection to each server is required.

See Also

close, **deallocate cursor**, **fetch**, **open** in this chapter.

declare cursor in the *Adaptive Server Reference Manual*.

delete

Function

Removes rows from a table.

Syntax

```
delete [from]
   [[database.]owner.]{view_name|table_name}
   [where search_conditions]

delete [[database.]owner.]{table_name | view_name}
   [from [[database.]owner.]{view_name|table_name
    [(index index_name [ prefetch size ][lru|mru])]}
    [, [[database.]owner.]{view_name|table_name
    (index index_name [ prefetch size ][lru|mru])]}
   ]...]
   [where search_conditions]

delete [from]
   [[database.]owner.]{table_name|view_name}
   where current of cursor_name
```

Comments

- Component Integration Services processes the **delete** command when the table on which it operates has been created as a proxy table. Component Integration Services forwards the entire request (or part of it) to the server that owns the actual object.
- Component Integration Services executes the **delete** command using one of two methods:
  1. The entire command is forwarded to the remote server as a single statement in close to its original syntax. If the syntax and remote capabilities match, the entire statement is forwarded and processed remotely. This is referred to as **quickpass mode**.
  2. If the entire command cannot be forwarded to a remote server, Component Integration Services declares and opens one or more cursors in update mode, and begins a scan on the remote table. Each cursor forwards as much of the original statement's predicates to the remote server as possible. For each row fetched that meets the search criteria, a positioned delete is executed.
- When Component Integration Services forwards the **delete** command to a remote server, the table name used is the remote table name, and the column names used are the remote column

names. These names may not be the same as the local proxy table names.

- Component Integration Services generally passes the original **delete** syntax to remote servers as a single statement, but the following conditions will likely cause the statement to be executed using method 2, above:
  - The statement contains multiple tables that are not located in the same remote server
  - The statement contains local tables (including temporary tables)
  - The statement contains **case** expressions
  - The statement contains *text* or *image* columns
  - The statement contains certain referential integrity checks
  - The statement contains system functions in the predicate list
  - The statement contains syntax that the remote server does not support
- The format involving **where current of** is never forwarded to a remote server and causes the statement to be executed using method 2 above.
- If Component Integration Services cannot pass the entire statement to a remote server, a unique index must exist on the table.

Server Class *sql_server*

- If Component Integration Services cannot forward the original query without alteration, it performs the delete using method 2.

Server Class *direct_connect*

- The syntax forwarded to servers of class *direct_connect* is dependent on the capabilities negotiation which occurs when Component Integration Services first connects to the remote DirectConnect. Examples of negotiable capabilities include: subquery support, **group by** support, and built-in support.
- A DirectConnect can request that the **delete** command be generated in DB2 syntax.
- Component Integration Services passes data values as parameters to either a cursor or a dynamic SQL statement. Language statements can also be used if the DirectConnect

supports it. The parameters are in the datatype native to Adaptive Server and must be converted by the DirectConnect into formats appropriate for the target DBMS.

Server Class *db2*

- Server's of class *db2* do not contain the capabilities negotiation features of server class *direct_connect*, so the syntax passed to the remote server is simpler than that allowed by Transact-SQL. The syntax does not contain the following:
  - Search conditions containing subqueries, **group by**, or **order by** clauses
  - Transact-SQL built-in functions
  - Transact-SQL operators (such as bitwise operators)
  - Syntax not allowed by DB2

  Component Integration Services processes the **delete** command using method 2, described above, when the statement is complex.
- If the server is a DB2 system, use traceflag 11215 to instruct Component Integration Services that the remote server is capable of handling all DB2 syntax. This assumption is not made automatically because not all gateways using the *db2* server class are actually connected to DB2 systems. When trace flag 11215 is turned on, **quickpass mode** is used unless the following conditions exist:
  - The statement cannot be expressed in DB2 syntax
  - The statement contains outer joins
  - The statement contains **like** clauses with Sybase extensions
  - The statement contains built-in functions that are not supported by DB2

Server Class *generic*

- Server's of class *generic* do not contain the capabilities negotiation features of server class *direct_connect*, so the syntax passed to the remote server is simpler than that allowed by Transact-SQL. The syntax does not contain the following:
  - Search conditions containing subqueries, **group by**, or **order by** clauses
  - Transact-SQL built-in functions

    - Transact-SQL operators (such as bitwise operators)
    - Syntax not allowed by the *generic* server class
- Complex statements cause Component Integration Services to perform a **select** statement followed by the **delete** statement when qualifying rows are found.

See Also

delete in the *Adaptive Server Reference Manual*.

drop database

Function

Removes one or more databases from Adaptive Server.

Syntax

```
drop database database_name [, database_name]...
```

Comments

- For each database being dropped, Component Integration Services scans *sysobjects* to check for proxy tables in the database. Each proxy table that was not created with the **existing** keyword is dropped in the remote server that owns the object.

Server Class *sql_server*

- Component Integration Services issues a **drop table** command for each table that was not created with the **existing** keyword.

Server Class *direct_connect*

- Component Integration Services issues a **drop table** command for each table that was not created with the **existing** keyword.

Server Class *db2*

- Component Integration Services issues a **drop table** command for each table that was not created with the **existing** keyword.

Server Class *generic*

- Component Integration Services issues the following RPC for each proxy table that was not created with the **existing** keyword:

```
gen_drop_table table_name, owner_name,
   database_name
```

See Also

drop database in the *Adaptive Server Reference Manual*.

drop index

Function

Removes an index from a table in the current database.

Syntax

```
drop index table_name.index_name
   [, table_name.index_name]...
```

Comments

- Component Integration Services processes the **drop index** command when the table involved has been created as a proxy table. The actual table and index reside on a remote server. Component Integration Services forwards the request to the remote server, and removes the index from the proxy table.
- When Component Integration Services forwards the **drop index** command to a remote server, the table name used is the remote table name, and the index names used are the remote index names. These names may not be the same as the local proxy table names.
- If multiple indexes are dropped in a single command, each index is sent as an individual **drop index** command.
- Trace flag 11208 changes the behavior of the **drop index** command. If trace flag 11208 is turned on, the **drop index** command is not sent to the remote server. Instead, Adaptive Server processes the command locally, as if the table on which it operates is local. This is useful for synchronizing the local Adaptive Server schema with the schema of the remote database.

Server Class *sql_server*

- Component Integration Services forwards the following **drop index** syntax to a remote server configured as class *sql_server*:

```
drop index table_name.index_name
```

Component Integration Services precedes this statement with a **use *database*** command since the **drop index** syntax does not allow you to specify the database name.

Server Class *direct_connect*

- Component Integration Services forwards the following **drop index** syntax to a remote server configured as class *direct_connect*:

```
drop index table_name.index_name
```

Server Class *db2*

- Component Integration Services forwards the following **drop index** syntax to a remote server configured as class *db2*:

```
drop index index_name
```

Server Class *generic*

- Component Integration Services forwards the following RPC to a remote server configured as class *generic*:

```
gen_drop_index index_name, table_name,owner_name,
   database_name
```

See Also

drop index in the *Adaptive Server Reference Manual*.

drop table

Function

Removes a table definition and all of its data, indexes, triggers, and permissions from the database.

Syntax

```
drop table [[database.]owner.]table_name
   [, [[database.]owner.]table_name ]...
```

Comments

- Component Integration Services processes the **drop table** command when the table on which it operates has been created as a proxy table. Component Integration Services forwards the entire request (or part of it) to the server that owns the actual object if the table was not created with the **existing** keyword.
- When Component Integration Services forwards the **drop table** command to a remote server, the table name used is the remote table name. This name may not be the same as the local proxy table name.
- If multiple tables are dropped in a single command, each table is sent as an individual **drop table** command.
- A table in use by another user or process cannot be dropped and an error stating that the table is in use is returned.

Server Class *sql_server*

- Component Integration Services forwards the following **drop table** syntax to a remote server configured as class *sql_server*:

```
drop table database.owner.table_name
```

Server Class *direct_connect*

- Component Integration Services requests a capabilities response from a remote server with server class *direct_connect*, but support for **drop table** is not optional. The behavior of the DirectConnect is database dependent.

Server Class *db2*

- Component Integration Services forwards the following **drop table** syntax to a remote server configured as class *db2*:

```
drop table owner.table_name
```

Server Class *generic*

- Component Integration Services forwards the following RPC to a remote server configured as class *generic*:

```
gen_drop_table table_name, owner, database
```

See Also

drop table in the *Adaptive Server Reference Manual*.

execute

Function

Runs a system procedure or a user-defined stored procedure.

Syntax

```
[execute] [@return_status = ]
   [[[server.]database.]owner.]procedure_name[;number]
       [[@parameter_name =] value |
           [@parameter_name =] @variable [output]
       [,[@parameter_name =] value |
           [@parameter_name =] @variable [output]...]]
   [with recompile]
```

Comments

- When the **execute** command is used to issue an RPC to a remote server, Adaptive Server issues the RPC via one of two methods. The method used to issue the RPC determines whether the work performed by the RPC can be part of an on-going transaction. The two methods are as follows:
  1. The RPC is issued via the Adaptive Server's site handler. This is the Adaptive Server's default method of issuing RPCs. In this case, the RPC cannot be part of an on-going transaction.
  2. The RPC is issued via Component Integration Services. In this case, the RPC can be part of an on-going transaction. To issue RPCs using this method, **cis rpc handling** must be turned on. This is done via the **set** command or the **sp_configure** system procedure.

See Also

"RPC Handling and Component Integration Services" on page 3-15.

set in this chapter.

execute in the *Adaptive Server Reference Manual.*

fetch

Function

Returns a row or a set of rows from a cursor result set.

Syntax

```
fetch cursor_name [ into fetch_target_list ]
```

Comments

- When the first **fetch** is received, Component Integration Services constructs the query defined by the **declare cursor** command and sends it to the remote server.

  If the remote server supports Client-Library cursors, Component Integration Services takes the following steps:

  1. Declares a cursor:

```
ct_cursor(command, CS_CURSOR_DECLARE...)
```

  2. Establishes the cursor row count:

```
ct_cursor(command, CS_CURSOR_ROWS,...
   cursor_row_count)
```

  3. Opens a Client-Library client cursor to the remote server:

```
ct_cursor(command, CS_CURSOR_OPEN...)
```

  If the remote server does not support Client-Library cursors, Component Integration Services sends a language request to the server. This may require an additional connection to that server.

- If the **declare cursor** command included a **for update** clause, the cursor row count is set to 1; otherwise, it is set to the value of the configuration parameter **cis_cursor_rows**.
- After the cursor is opened or the language request is sent, Component Integration Services issues a Client-Library **ct_fetch** command to obtain the first row. Client-Library array binding is used to establish the buffer in which to place the fetched results, whether Client-Library cursors or language requests are used to generate the fetchable results. The number of rows that are buffered by a single fetch is determined by the cursor row count discussed above.

  Subsequent **fetch** requests retrieve rows from the buffered results, until the end of the buffer is reached. At that time, Component Integration Services issues another Client-Library **ct_fetch** command to the remote server.

- A **fetch** against a cursor that has no remaining rows in its result set causes Component Integration Services to close the remote cursor.

Server Class *sql_server*

- For pre-version 10.0 SQL Server, Component Integration Services sends a language request to the remote server when the first **fetch** is received after the cursor is opened.
- For version 10.0 or later Servers, Component Integration Services declares a cursor to the remote server by means of Client-Library.

Server Class *direct_connect*

- Component Integration Services treats servers in class *direct_connect* as if they were version 10.0 or later of class *sql_server*.

Server Class *db2*

- Component Integration Services sends a language request to the remote server when the first **fetch** is requested after the cursor is opened.

Server Class *generic*

- Cursors are not supported by server class *generic*. Component Integration Services sends a language request to the remote server when the first **fetch** is requested.
- The configuration parameter **cis cursor rows** determines how many rows are returned from a single fetch sent to the remote server. If this number is greater than 1, the rows are buffered by Client-Library. Subsequent **fetch** requests retrieves rows from the buffer until it is empty, at which time Component Integration Services issues another **fetch** to the remote server.

See Also

close, **deallocate cursor**, **declare cursor**, **open** in this chapter.

fetch in the *Adaptive Server Reference Manual*.

insert

Function

Adds new rows to a table or view.

Syntax

```
insert [into]
   [database.[owner.]]{table_name|view_name}
   [(column_list)]
   {values (expression [, expression]...)
       |select_statement }
```

Comments

- Component Integration Services processes the **insert** command when the table on which it operates has been created as a proxy table. Component Integration Services forwards the entire request (or part of it) to the server that owns the actual object.
- When Component Integration Services forwards the **insert** command to a remote server, the table name used is the remote table name, and the column names used are the remote column names. These names may not be the same as the local proxy table names.

Server Class *sql_server*

- **insert** commands using the **values** keyword are fully supported.
- **insert** commands using a **select** command are supported for all datatypes except *text* and *image*. *text* and *image* columns are only supported when they contain null values.
- If all **insert** and **select** tables reside on the same remote server, the entire statement is forwarded to the remote server for execution. This is referred to as **quickpass mode**. Quickpass mode is not used if the **select** statement does not conform to all the quickpass rules for a **select** command (see “select” on page 4-56).
- If the **select** tables reside on one remote server, and the **insert** table resides on a different server, Component Integration Services selects each row from the source tables, and inserts the row into the target table.

Server Class *direct_connect*

- **insert** commands using the **values** keyword are fully supported.

- **insert** commands using a **select** command are fully supported, but the table must have a unique index if the table has *text* or *image* columns. When using **insert** with a **select** command, the entire command is sent to the remote server if:
  - All tables referenced in the command reside on the remote server
  - The capabilities response from the DirectConnect indicates that **insert-select** commands are supported

  If both conditions are not met, Component Integration Services selects each row from the source tables, and inserts the row into the target table.
- Component Integration Services passes data values as parameters to either a cursor or a dynamic SQL statement. Language statements can also be used if the DirectConnect supports it. The parameters are in the datatype native to Adaptive Server and must be converted by the DirectConnect into formats appropriate for the target DBMS.

Server Class *db2*

- **insert** commands using the **values** keyword are fully supported for all valid DB2 datatypes.
- **insert** commands using a **select** command are fully supported for all valid DB2 datatypes.
- When using **insert** with a **select** command, the entire statement is sent to the remote server if:
  - All tables referenced in the statement reside on the remote server
  - Trace flag 11215 is enabled

  If both conditions are not met, Component Integration Services selects each row from the source tables, and inserts the rows into the target table.

Server Class *generic*

- **insert** commands using the **values** keyword are supported for all valid datatypes in server class *generic*.
- **insert** commands using a **select** command are supported for all valid datatypes in server class *generic*. When using **insert** with a **select** command, Component Integration Services selects each

row from the source tables, and then inserts the row into the target table.

See Also

insert in the *Adaptive Server Reference Manual*.

open

Function

Opens a cursor for processing.

Syntax

```
open cursor_name
```

Comments

- Component Integration Services takes no additional action for the **open** command.
- Adaptive Server processes the **open** command locally—it sets cursor state information not used by Component Integration Services until the first **fetch** command is received.
- Refer also to the **declare**, **fetch**, **close** and **deallocate** commands.

See Also

close, **deallocate cursor**, **declare cursor**, **fetch** in this chapter.

open in the *Adaptive Server Reference Manual*.

prepare transaction

Function

Used by two-phase commit applications to see if a server is prepared to commit a transaction.

Syntax

```
prepare tran[saction]
```

Comments

- Adaptive Server notifies Component Integration Services when it receives a **prepare transaction** command so that remote servers involved in the current transaction can enter the prepared state.
- For each server that is involved in the current transaction, Component Integration Services sends a **prepare transaction** command and monitors the response. If there are no errors reported, each remote server is assumed to be in a prepared state and Component Integration Services returns control to the Adaptive Server. Adaptive Server then enters a prepared state for local work performed by the transaction.
- This behavior applies only to servers in class *sql_server* and *direct_connect*; **prepare transaction** is ignored by servers in other server classes that are involved in the current transaction.

Server Class *sql_server*

- Component Integration Services sends a **prepare transaction** command to each server in class *sql_server* that is version 10.0 or later.
- The **prepare transaction** command is not sent to the following types of servers:
  - Sybase IQ 11.x
  - Microsoft SQL Server
  - Pre-version 10.0 SQL Server
  - OmniSQL Server 10.1.2

Server Class *direct_connect*

- Handling of the **prepare transaction** command for servers in class *direct_connect* is identical to that of server class *sql_server* (version 10.0 or later).

Server Class *db2*

- Component Integration Services does not send the **prepare transaction** command to servers in class *db2*.

Server Class *generic*

- Component Integration Services does not send the **prepare transaction** command to servers in class *generic*

See Also

prepare transaction in the *Adaptive Server Reference Manual*.

readtext

Function

Reads *text* and *image* values, starting from a specified offset and reading a specified number of bytes or characters.

Syntax

```
readtext [[database.]owner.]table_name.column_name
   text_pointer offset size [holdlock]
   [using {bytes | chars | characters}]
   [at isolation {read uncommitted | read committed |
       serializable}]
```

Comments

- Component Integration Services processes the **readtext** command when the table on which it operates has been created as a proxy table. Component Integration Services forwards the entire request (or part of it) to the server that owns the actual object.
- When Component Integration Services forwards the **readtext** command to a remote server, the table name used is the remote table name, and the column names used are the remote column names. These names may not be the same as the local proxy table names.
- The **using bytes** and **at isolation** clauses are ignored.

Server Class *sql_server*

- Component Integration Services forwards the following syntax to the remote server when the underlying table is a proxy table:

```
readtext [[database.]owner.]table_name.column_name
   text_pointer offset size
   [using {chars | characters}]
```

Server Class *direct_connect*

- If the DirectConnect does not support text pointers, **readtext** cannot be sent and its use results in errors.
- If the DirectConnect does support text pointers, Component Integration Services forwards the following syntax to the remote server:

```
readtext
   [[database.]owner.]table_name.column_name
   text_pointer offset size
   [using {chars | characters}]
```

- **readtext** is issued anytime *text* or *image* data must be read. **readtext** is called when a **select** command refers to a *text* or *image* column in the select list, or when a **where** clause refers to a *text* or *image* column.

  For example, you have a proxy table *books* that is mapped to the *books* table on the remote server *foo*. The columns are *id*, *name*, and the *text* column *blurb*. When the following statement is issued:

```
select * from books
```

  Component Integration Services sends the following syntax to the remote server:

```
select id, name, textptr(blurb) from foo_books
```

```
readtext foo_books.blurb @p1 0 0 using chars
```

Server Class *db2*

- **readtext** is not supported since *text* and *image* datatypes are not supported for servers in class *db2*.

Server Class *generic*

- **readtext** is not supported since *text* and *image* datatypes are not supported for servers in class *generic*.

See Also

readtext in the *Adaptive Server Reference Manual*.

rollback transaction

Function

Rolls a user-defined transaction back to the last savepoint inside the transaction or to the beginning of the transaction.

Syntax

```
rollback {transaction | tran | work}
   [transaction_name | savepoint_name ]
```

Comments

- Adaptive Server notifies Component Integration Services when it receives a **rollback transaction** command and Component Integration Services attempts to rollback work associated with remote servers in the current transaction.
- Multiple remote servers can be involved in a single transaction, each with their own unit of work which is associated with the Adaptive Server unit of work.
- Remote work is rolled back before local work.
- Work performed by transactional RPC's is included in the local transaction and can be rolled back if the remote server supports RPC's within transactions.
- *transaction_name* and *savepoint_name* is not used by Component Integration Services in this release.

Server Class *sql_server*

- When Component Integration Services receives notification that a transaction is to be rolled back, it checks the TRANSACTION ACTIVE state of all remote connections associated with the client application. For each connection with an active transaction, Component Integration Services sends a **rollback transaction**. If all remote servers respond with no error, Component Integration Services notifies the Adaptive Server that it can begin to roll back local work.

  This process applies to version 10.0 or later, but not to the following servers represented by server class *sql_server* is:

  - Pre-version 10.0 SQL Server
  - Microsoft SQL Server (any version)
  - Sybase IQ

- OmniConnect 10.1.2

For these types of servers, transaction handling is similar to server class *db2*, described below.

Server Class *direct_connect*

- Transaction processing for servers in class *direct_connect* is identical to that of server class *sql_server* (version 10.0 or later).

Server Class *db2*

- Transactions are supported only at the statement level for servers in class *db2*. When the internal state of a client connection indicates that there is an active transaction, Component Integration Services precedes each **insert**, **update** and **delete** command with a **begin transaction** command. It then issues a **commit** or **rollback transaction** (depending on the success or failure of the statement) immediately after the statement is complete.

Server Class *generic*

- Transactions are supported only at the statement level for servers in class *db2*. When the internal state of a client connection indicates that there is an active transaction, Component Integration Services precedes each **insert**, **update** and **delete** command with a **gen_begin_xact** RPC. It then issues a **gen_commit_xact** or **gen_rollback_xact** RPC (depending on the success or failure of the statement) immediately after the statement is complete.

See Also

rollback in the *Adaptive Server Reference Manual*.

select

Function

Retrieves rows from database objects.

Syntax

```
select [all | distinct] select_list
   [into [[database.]owner.]table_name]
   [from [[database.]owner.]{view_name|table_name
     [(index index_name [ prefetch size ][lru|mru])]}
           [holdlock | noholdlock] [shared]
       [,[[database.]owner.]{view_name|table_name
     [(index index_name [ prefetch size ][lru|mru])]}
           [holdlock | noholdlock] [shared]]... ]

   [where search_conditions]

   [group by [all] aggregate_free_expression
       [, aggregate_free_expression]... ]
   [having search_conditions]

   [order by
   {[[[database.]owner.]{table_name.|view_name.}]
       column_name | select_list_number | expression}
           [asc | desc]
   [,{[[[database.]owner.]{table_name|view_name.}]
       column_name | select_list_number | expression}
           [asc | desc]]...]

   [compute row_aggregate(column_name)
           [, row_aggregate(column_name)]...
       [by column_name [, column_name]...]]

   [for {read only | update [of column_name_list]}]

   [at isolation {read uncommitted | read committed |
       serializable}]

   [for browse]
```

Comments

- Component Integration Services processes the **select** command when any table on which it operates has been created as a proxy table. When possible, Component Integration Services forwards the entire syntax of a **select** command to a single remote server. This is referred to as **quickpass mode**.

- When Component Integration Services forwards the **select** command to a remote server, the table name used is the remote table name, and the column names used are the remote column names. These names may not be the same as the local proxy table names.
- The following keywords are ignored, but they do not prevent Component Integration Services from using quickpass mode:
  - **shared**
  - **prefetch**
  - **at isolation**
  - **index**
  - **lru | mru**
- The following keywords are never forwarded to a remote server and they do prevent Component Integration Services from using quickpass mode:
  - **compute by**
  - **for browse**
  - **into**
- Quickpass mode is not used if any of the following conditions exist:
  - All tables referenced in the **from** clause do not reside on the same remote server
  - Any tables are local (including temporary tables)
  - The query contains syntax that the remote server does not support
  - The query contains *text* or *image* columns
  - The query contains system functions in the select list or in search conditions
- **select** commands in a **union** operation can all be forwarded to a remote server, including the **union** operator, if all tables in the **select** commands reside on the same remote server.
- If the **select** command returns a sorted result set involving a character column from a remote server (for example, in a **union** operation, a **group by** clause, or an **order by** clause), the rows may be returned in an unexpected sort order if the remote server is

configured with a different sort order than Adaptive Server. You can rerun the query with traceflag 11216 turned on to receive the expected sort order. This traceflag is global and should be turned off as soon as the query is executed.

Server Class *sql_server*

- All syntax is supported. Since the remote server is assumed to have all capabilities necessary to process Transact-SQL syntax, all elements of a **select** command, except those mentioned above, are forwarded to a remote server, using quickpass mode.
- A bulk copy transfer is used to copy data into the new table when a **select...into** command is issued and the **into** table resides on a remote Adaptive Server. Both the local and remote databases must be configured with **dboption** set to **select into / bulkcopy.**

Server Class *direct_connect*

- The first time Component Integration Services requires a connection to a server in class *direct_connect*, a request for capabilities is made of the DirectConnect. Based on the response, Component Integration Services determines the parts of a **select** command to forward to the DirectConnect. In most cases, this is determined by the capabilities of the DBMS with which the DirectConnect is interfacing.
- If the entire statement cannot be forwarded to the DirectConnect using quickpass mode, Component Integration Services compensates for the functionality that cannot be forwarded. For example, if the remote server cannot handle the **order by** clause, quickpass is not used and Component Integration Services performs a sort on the result set.
- Component Integration Services passes data values as parameters to either a cursor or a dynamic SQL statement. Language statements can also be used if the DirectConnect supports it. The parameters are in the datatype native to Adaptive Server and must be converted by the DirectConnect into formats appropriate for the target DBMS.
- The **select...into** command is supported, but the table must have a unique index if the table has *text* or *image* columns.
- If the **select...into** format is used and the **into** table is accessed through a DirectConnect, bulk inserts are not used. Instead, Component Integration Services uses Client-Library to prepare a

parameterized dynamic **insert** command, and executes it for each row returned by the **select** portion of the command.

Server Class *db2*

- By default, Component Integration Services does not forward syntax involving **order by, group by, union, distinct, all**, and expressions that involve more than column names.
- When you turn traceflag 11215 on, the full capabilities of a DB2 database are assumed, and Component Integration Services forwards as much syntax to the remote server (gateway) as DB2 can process, including **order by, group by, union**, and so forth.

Server Class *generic*

- Component Integration Services only forwards the syntax to servers in class *generic* as is documented in the *Generic Access Module Reference Manual.* Quickpass mode is not used when servers in class *generic* are involved in a query.

See Also

select in the *Adaptive Server Reference Manual*.

set

Function

Sets Adaptive Server query processing options for the duration of the user's work session. The subset of options listed below affects behavior unique to Component Integration Services. For a complete list of options, see the *Adaptive Server Reference Manual*.

Syntax

```
set cis_rpc_handling {on | off}
set transactional_rpc {on | off}
set textsize value
```

Comments

- Normally, all outbound RPCs are routed through Adaptive Server's site handler. These RPCs cannot participate in any transactions, and the performance characteristics of routing many RPCs through the site handler may necessitate the use of an alternate method for RPC handling.
- Component Integration Services provides an alternate means of handling outbound RPCs. If **cis_rpc_handling** is **on**, outbound RPCs are routed through a Client-Library connection that is persistent through the life of the client's connection to the Adaptive Server. This means that any number of RPCs can be routed through the same connection, without a connect and disconnect between each RPC. This connection is the same connection used by Component Integration Services to handle all interaction with the remote server, including processing of **select**, **insert**, **delete** and **update** commands.
- The client application issues **set cis_rpc_handling on** or **off** to control whether an outbound RPC is to be routed through the Adaptive Server's site handler or through a Component Integration Services connection. If **cis_rpc_handling** is **on**, Component Integration Services processes the RPC request; if **cis_rpc_handling** is **off**, the site handler processes the RPC.
- When a client application makes a new connection to Adaptive Server, the connection inherits the setting for the configuration parameter **cis rpc handling** (default is **off**). This determines the default handling for outbound RPCs.

- Setting **transactional_rpc on** results in the same behavior as setting **cis_rpc_handling on**, except that RPCs that are issued outside of a transaction will continue to be routed through the site handler.
- If both **transactional_rpc** and **cis_rpc_handling** are **on**, then **cis_rpc_handling** has precedence.

Server Class *direct_connect*

- Component Integration Services uses the **textsize** option to specify in bytes the memory to allocate for the retrieval of *text* and *image* data. Memory is allocated only for remote servers that cannot perform *text* and *image* handling with text pointers. For additional information on *text* and *image* handling, refer to “text and image Datatypes” on page 2-16.
- When a server with server class *direct_connect* requests DB2 syntax (for example, Net Gateway configured as server class *direct_connect*), Component Integration Services uses the **textsize** option to set the size of the *long char* column when a *text* column is mapped to a *long char* column.

See Also

set in the *Adaptive Server Reference Manual*.

setuser

Function

Allows a Database Owner to impersonate another user.

Syntax

```
setuser ["user_name"]
```

Comments

- The Database Owner uses the **setuser** command to adopt the identity of another user in order to use another user's database objects. When using Component Integration Services, these objects can be either local or remote.
- Component Integration Services processes the **setuser** command—it does **not** forward the command to the remote server. Component Integration Services drops all current connections that have been made on behalf of the current user.
- The **setuser** command cannot be executed when a transaction is current.
- Permissions that are set on a remote server override permissions set by Component Integration Services. Component Integration Services cannot change permissions of a user on a remote server.
- Prior to using the **setuser** command, the user to be impersonated must have an external login mapped to the remote server. This is set by the **sp_addexternlogin** system procedure (for more information on **sp_addexternlogin**, see the *Adaptive Server Reference Manual*).

See Also

setuser in the *Adaptive Server Reference Manual*.

truncate table

Function

Removes all rows from a table.

Syntax

```
truncate table [[database.]owner.]table_name
```

Comments

- Component Integration Services processes the **truncate table** command when the table on which it operates has been created as a proxy table.
- When Component Integration Services forwards the **truncate table** command to a remote server, the table name used is the remote table name. This name may not be the same as the local proxy table name.

Server Class *sql_server*

- Component Integration Services forwards the **truncate table** command to servers of class *sql_server.*

Server Class *direct_connect* and *sds*

- If the remote server has requested DB2 syntax, the following statement is forwarded:

```
delete from [owner.]table_name
```

Otherwise, Transact-SQL syntax is sent:

```
truncate table [[database.]owner.]table_name
```

Server Class *db2*

- The following syntax is forwarded to the remote server:

```
delete from [owner.]table_name
```

Server Class *generic*

- Component Integration Services processes the **truncate table** command using an RPC call to the procedure **gen_truncate_table.**

See Also

truncate table in the *Adaptive Server Reference Manual.*

update

Function

Changes data in existing rows, either by adding data or by modifying existing data.

Syntax

```
update [[database.]owner.]{table_name | view_name}
   set [[[database.]owner.]{table_name.|view_name.}]
       column_name1 =
           {expression1|NULL|(select_statement)}
       [, column_name2 =
           {expression2|NULL|(select_statement)}]...
   [from [[database.]owner.]{view_name|table_name
   [(index index_name [ prefetch size ][lru|mru])]}

        [,[[database.]owner.]{view_name|table_name
   [(index index_name [ prefetch size ][lru|mru])]}]
   ...]
   [where search_conditions]
update [[database.]owner.]{table_name | view_name}
   set [[[database.]owner.]{table_name.|view_name.}]
       column_name1 =
          {expression1|NULL|(select_statement)}
       [, column_name2 =
          {expression2|NULL|(select_statement)}]...
   where current of cursor_name
```

Comments

- Component Integration Services processes the **update** command when the table on which it operates has been created as a proxy table. Component Integration Services forwards the entire request (or part of it) to the server that owns the actual object.
- The **update** command specifies the row or rows you want to change, and the new data. The new data can be a constant, an expression, or data pulled from other tables.
- Component Integration Services executes the **update** command using one of two methods:
  1. The entire command is forwarded to the remote server as a single statement in close to its original syntax. If the syntax and remote capabilities match, the entire statement is forwarded and processed remotely. This is referred to as **quickpass mode**.

2. If the entire command cannot be forwarded to a remote server, Component Integration Services declares and opens one or more cursors in update mode, and begins a scan on the remote table. Each cursor forwards as much of the original statement's predicates to the remote server as possible. For each row fetched that meets the search criteria, a positioned update is executed.

- When Component Integration Services forwards the **update** command to a remote server, the table name used is the remote table name, and the column names used are the remote column names. These names may not be the same as the local proxy table names.
- Component Integration Services generally passes the original **update** syntax to remote servers as a single statement, but the following conditions will likely cause the statement to be executed using method 2, above:
  - The statement contains multiple tables that are not located in the same remote server
  - The statement contains local tables (including temporary tables)
  - The statement contains **case** expressions
  - The statement contains *text* or *image* columns
  - The statement contains certain referential integrity checks
  - The statement contains system functions in the predicate list
  - The statement contains syntax that the remote server does not support
- The following keywords are ignored and do not prevent Component Integration Services from using quickpass mode:
  - **prefetch**
  - **index**
  - **lru | mru**
- The format involving **where current of** is never forwarded to a remote server and causes the statement to be executed using method 2 above.
- If Component Integration Services cannot pass the entire statement to a remote server, a unique index must exist on the table.

Server Class *sql_server*

- The **update** command is fully supported for all datatypes except *text* and *image*. *text* and *image* data cannot be changed with the **update** command, except when setting the *text* or *image* value to null. Use the **writetext** command instead.
- If quickpass mode is not used, data is retrieved from the source tables, and the values in the target table are updated.

Server Class *direct_connect*

- The following syntax is supported by servers of class *direct_connect*:

```
update [[database.]owner.]{table_name | view_name}
   set [[[database.]owner.]{table_name.|view_name.}]
       column_name1 =
           {expression1|NULL|(select_statement)}
       [, column_name2 =
           {expression2|NULL|(select_statement)}]...
[where search_conditions]
```

  update commands that conform to this syntax use quickpass mode, if the capabilities response from the remote server indicates that all elements of the command are supported. Examples of negotiable capabilities include: subquery support, **group by** support, and built-in support.

- If the remote server does not support all elements of the command, or the command contains a **from** clause, Component Integration Services issues a query to obtain the values for the **set** clause, and then issues an **update** command to the remote server.
- Component Integration Services passes data values as parameters to either a cursor or a dynamic SQL statement. Language statements can also be used if the DirectConnect supports it. The parameters are in the datatype native to Adaptive Server and must be converted by the DirectConnect into formats appropriate for the target DBMS.

Server Class *db2*

- The following syntax is supported by servers of class *db2*:

```
update [[database.]owner.]{table_name | view_name}
   set [[[database.]owner.]{table_name.|view_name.}]
       column_name1 =
           {expression1|NULL|(select_statement)}
       [, column_name2 =
           {expression2|NULL|(select_statement)}]...
[where search_conditions]
```

- Server's of class *db2* do not contain the capabilities negotiation features of server class *direct_connect*, so the syntax passed to the remote server is simpler than that allowed by Transact-SQL. The syntax does not contain the following:
  - Search conditions containing subqueries, **group by**, or **order by** clauses
  - Transact-SQL built-in functions
  - Transact-SQL operators (such as bitwise operators)
  - Syntax not allowed by DB2

  Component Integration Services processes the **update** command using method 2, described above, when the statement is complex.
- If the server is a DB2 system, use traceflag 11215 to instruct Component Integration Services that the remote server is capable of handling all DB2 syntax. This assumption is not made automatically because not all gateways using the *db2* server class are actually connected to DB2 systems. When trace flag 11215 is turned on, quickpass mode is used unless the following conditions exist:
  - The statement cannot be expressed in DB2 syntax
  - The statement contains outer joins
  - The statement contains **like** clauses with Sybase extensions
  - The statement contains built-in functions that are not supported by DB2
- When an **update** statement contains a **select** statement, Component Integration Services issues a query to obtain the values for the **set** clause, and then issues an **update** command to the remote server, unless trace flag 11215 is enabled.
- When an **update** statement contains a **from** clause, Component Integration Services issues a query to obtain the values for the **set** clause, and then issues an **update** command to the remote server.

Server Class *generic*

The following syntax is supported by servers of class *generic*:

```
update [[database.]owner.]{table_name | view_name}
   set [[[database.]owner.]{table_name.|view_name.}]
       column_name1 =
           {expression1|NULL}
       [, column_name2 =
           {expression2|NULL}]...
[where search_conditions]
```

- Server's of class *generic* do not contain the capabilities negotiation features of server class *direct_connect*, so the syntax passed to the remote server is simpler than that allowed by Transact-SQL. The syntax does not contain the following:
  - Search conditions containing subqueries, **group by**, or **order by** clauses
  - Transact-SQL built-in functions
  - Transact-SQL operators (such as bitwise operators)
  - Syntax not allowed by the *generic* server class

  Component Integration Services processes the **update** command using method 2, described above, when the statement is complex.
- When an **update** statement contains a complex search condition, a **select** statement, or a **from** clause, Component Integration Services issues a query to obtain the values for the **set** clause, and then issues an **update** command to the remote server.

See Also

update in the *Adaptive Server Reference Manual*.

update statistics

Function

Updates information about the distribution of key values in specified indexes. Also updates row count information.

Syntax

```
update statistics table_name [index_name]
```

Comments

- When the **update statistics** command is issued against a proxy table, Component Integration Services provides meaningful statistics on the remote table and the given index or on all indexes if no index is specified. The results are used to construct a distribution page for each index. This distribution page is stored in the database. When a new distribution page is created for an index, any previous distribution page for that index is freed.
- Using **update statistics**, Component Integration Services creates extremely accurate distribution statistics for remote tables. This information is used to determine the optimal join order, giving Component Integration Services the ability to generate optimal queries against remote databases which may not support cost-based query optimization.
- When Component Integration Services forwards the command to a remote server, the table name used is the remote table name, and the column names used are the remote column names. These names may not be the same as the local proxy table names.
- Obtaining information on an index, and especially on a number of indexes, can be time consuming on large tables. Trace flag 11209 can be used to indicate that **update statistics** is to obtain row count only. When this flag is on, previous distribution pages for indexes are not replaced.
- Component Integration Services retrieves row count information even if no indexes exist.

Server Class *sql_server*

- If the table on which the statistics are requested has no indexes, Component Integration Services issues the following command:

```
select count(*) from table_name
```

It is also the only command issued when trace flag 11209 is on.

- If the table has an index and the index is specified in the command, Component Integration Services issues the following commands:

```
select count(*) from table_name

select column_name [,column_name, ...]
   from table_name
   order by column_name [,column_name, ..]
```

  The column name(s) represent the column or columns that make up the index.

  For example, when the following command is issued:

```
update statistics customers ind_name
```

  Component Integration Services issues:

```
select count(*) from customers

select last_name, first_name
   from customers
   order by last_name, first_name
```

- If the table has one or more indexes but no index is specified in the statement, Component Integration Services issues the **select count (*)** once, and the **select/order by** commands for each index.

Server Class *direct_connect*

- The processing of **update statistics** in server class *direct_connect* is identical to that of server class *sql_server* described above.

Server Class *db2*

- The processing of **update statistics** in server class *db2* is identical to that of server class *sql_server* described above.

Server Class *generic*

- When an **update statistics** command is issued, Component Integration Services issues the following command:

```
     select count(*) from table_name
```

- No distribution statistics are calculated for tables owned by servers of this class.

See Also

update statistics in the *Adaptive Server Reference Manual.*

writetext

Function

Permits non-logged, interactive updating of an existing *text* or *image* column.

Syntax

```
writetext [[database.]owner.]table_name.column_name
   text_pointer [with log] data
```

Comments

- Component Integration Services processes the **writetext** command when the table on which it operates has been created as a proxy table.
- If the remote server referenced by the proxy table does not support text pointers, **writetext** is not supported.
- To process the **writetext** command, Component Integration Services issues the following Client Library commands using the connection established to the remote server:

```
ct_command(command, CS_SEND_DATA_CMD, NULL,
   CS_UNUSED, CS_COLUMN_DATA);
ct_data_info(command, CS_SET, CS_UNUSED, iodesc)
ct_send_data(command, (CS_VOID *) start, length)
```

Server Class *sql_server*

- The **writetext** command is processed using a separate connection to the remote server.

Server Class *direct_connect*

- If the DirectConnect supports text pointers, Component Integration Services treats the DirectConnect as if it were a server in class *sql_server.*

Server Class *db2*

- **writetext** is not supported for tables owned by servers in this class.

Server Class *generic*

- **writetext** is not supported for tables owned by servers in this class.

See Also

writetext in the *Adaptive Server Reference Manual.*

5 Utility Programs

This chapter describes the utility programs that are unique to Component Integration Services. Many of the standard Adaptive Server utility programs, such as **bcp**, **isql** and **startserver** are also required for Component Integration Services users. See the *Utility Programs* manual for your platform for more information.

Utility programs are commands that you invoke directly from the operating system.

The UNIX system shell interprets the utility commands. Place characters with special meaning to the shell, such as the backslash (\), asterisk (*), slash (/), and spaces, in quotes. You can precede some special characters with the backslash (\) to “escape” them. This prevents the shell from interpreting the special characters.

ddlgen

Function

The **ddlgen** utility is used to back up data definition statements associated with Component Integration Services or to migrate data definitions for databases from OmniConnect release 10.5 to Adaptive Server release 11.5. It generates an **isql** script containing data definition language statements for objects defined in Component Integration Services. See also Chapter 3, "Backing Up Your System."

Syntax

The parameters listed with accompanying text require an argument. Those listed without text do not require or accept an argument.

```
ddlgen
    [-S server]
    [-P [password]]
    [-F output_file_name]
    [-V]
    [-d [database_name]]
    [-e exclude_server]
    [-x]
    [-v]
    [-a display_charset]
    [-J client_charset]
    [-z language]
```

Parameters

-S *server* – specifies the name of the server to which you want to connect. This is the name that **ddlgen** looks for in the interfaces file. If **-S** is omitted, **ddlgen** looks for the server specified with your DSQUERY environment variable. If DSQUERY is not defined, SYBASE is used.

-P *password* – specifies the password of the "sa" user. If you do not specify the **-P** parameter, **ddlgen** prompts for a password. If the password is NULL, use the **-P** parameter without any password.

-F – specifies the name of the operating system file in which to store the output from **ddlgen**. If this parameter is omitted, output is directed to the file *ddlgen.sql*.

-V – runs **ddlgen** in verbose mode. Verbose mode causes **ddlgen** to print informational messages as it performs its tasks.

-d – specifies the database to be processed. There are three available choices:

- If the **-d** parameter is omitted, **ddlgen** backs up all the databases.
- If the **-d** parameter is used with no argument, **ddlgen** issues a prompt prior to processing each database, and once before processing all logins. This allows you to select which databases **ddlgen** processes.
- If the **-d** parameter is followed by a database_name, the named database is processed. Each database specified must be preceded with "-d". Multiple database names are permitted. This option is used as follows:

```
-d database_a -d database_b -d database_c
```

-e *exclude_server* – specifies a server, and all objects mapped to that server that are to be excluded from processing. Multiple server names are permitted. Each server specified must be preceded with "-e". If no servers are specified all servers are processed.

-x – creates a trace file called *debug*. The trace file records:

- Entry and exit from internal routines
- SQL statements presented to the source server
- Objects and definitions found in the source server
- Objects and definitions not processed in the source server

-v – prints the **ddlgen** version and copyright strings. No other operations are performed, and **ddlgen** exits.

-a *display_charset* – allows you to run **ddlgen** from a terminal whose character set differs from that of the machine on which **ddlgen** is running. **-a** in conjunction with **-J** specifies the character set translation file (*.xlt* file) required for the conversion. Use **-a** without **-J** only if the client character set is the same as the default character set.

-J *client_charset* – specifies that the server convert to and from *client_charset*, the character set used on the client. **-J** with no argument sets character conversion to NULL. No conversion takes place. Use **-J** if the client and server use the same character set. Omitting **-J** sets the character set to the default for the platform. The default may not necessarily be the character set that the client is using.

-z *language* – specifies the official name of the alternate language in which to display **ddlgen** messages. Without **-z, ddlgen** uses the server's default language.

Examples

The following example shows the two-step process to use **ddlgen** as a migration tool:

1. Direct **ddlgen** to the OmniConnect server OMNI105 using the following assumptions:
   - The password is NULL
   - The output script file is named **sample.sql**
   - Only record objects from the *ident* database
   - Exclude any objects that map to the server OMNINEW

   The syntax looks like this:

```
ddlgen -S omni105 -P -F sample.sql -d ident -e omninew
```

2. Direct the **sample.sql** script to the server OMNINEW:

   ```
   isql -Usa -P -Somninew < sample.sql
   ```

Objects from the *ident* database in OMNI105 are migrated to the server OMNINEW.

Comments

- The following process outlines a **ddlgen** session:
  - User submits **ddlgen** with parameters
  - **ddlgen** logs into -S server
  - User responds to prompts (if any)
  - **ddlgen** generates script
  - User reviews script and customizes
  - User directs script to -S server or other server
- The **ddlgen** utility is used to migrate or back up data definitions associated with Component Integration Services. **ddlgen** connects to a server that must be either an Adaptive Server with Component Integration Services enabled or an OmniConnect release 10.5 server. The **ddlgen** utility examines the version string

and determines the action to take based on the server type as shown in Table 5-1:

Table 5-1: ddlgen action by server type

Server Type	*ddlgen* Action
OmniConnect release 10.5	A migrate operation is required.
Adaptive Server 11.5 with Component Integration Services configured	A backup operation is required.
Adaptive Server 11.5 with Component Integration Services **not** configured	Inform the user that Component Integration Services is not configured, and that **ddlgen** cannot process this server.
OmniSQL Server, pre-release 11.0 SQL Server, or any other target that **ddlgen** 11.5 cannot process	Inform the user that processing cannot proceed.

- **ddlgen** connects to the source server as user "sa".
- **ddlgen** preserves the owner of tables and other objects by issuing **setuser** statements to the output script.
- The **ddlgen** utility generates an **isql** script containing data definition language (DDL) statements for objects in the server. The **isql** script can be used to restore a server by reinstalling the server and then running the script using **isql**. The **isql** script can also be used to replicate databases or servers by editing the script and running it against other servers.
- **ddlgen** records user-written stored procedures and permissions for the procedures to the output script. For the *master* database and *sybsystemprocs* database, stored procedures are written to the output script if the create date column (*crdate*) contains a date more than 5 seconds after Component Integration Services install scripts were run. This allows processing of user-written procedures and permissions for the procedures, and avoids processing Sybase-supplied stored procedures. Sites should preserve user_written procedures is a script in case **installmaster** is re-run after the initial installation.
- When **ddlgen** is used as a migration tool, **sp_configure** statements are **not** written to the output script. For using **ddlgen** as a backup tool with release 11.5, Sybase recommends saving the *severname*.cfg file in the SYBASE directory, and reapplying these configuration properties after the server is restored.
- The **ddlgen** utility writes **create database** statements at the beginning of the **isql** script file for the databases that are being processed. These statements should be reviewed when **ddlgen** is used as a

backup tool. The script must be modified to specify the devices and sizes of the databases that will be created prior to running the script when using **ddlgen** to migrate data.

- The actual passwords for logins are not written to the script file. Instead, login passwords are shown as “restored”, and external login passwords are shown as NULL. The passwords in the **sp_addlogin** and **sp_addexternlogin** statements can be modified before the script is executed, or the passwords can be changed after the script has been run by using the **sp_password** procedure to change login passwords or by executing another **sp_addexternlogin** statement to change external login passwords.
- Any syntax error results in a formatted display of valid usage.
- **ddlgen** does not write data definition statements for local tables to the output script. Sybase recommends that the data definition statements and the data in local tables be backed up using the standard Adaptive Server backup utilities.
- **ddlgen** checks remote table names and column names against a list of Sybase keywords. If the remote object name is a Sybase keyword, **ddlgen** issues **set quoted_identifer on** before using the keyword in SQL syntax. The keyword is enclosed in double quotes. Any non-keyword syntax requiring quotes employs single quotes. After issuing the SQL syntax with Sybase keywords, **ddlgen** writes **set quoted_identifier off** to the output script.

Example:

```
set quoted_identifier on
create existing table contract_status
(
contract_num int,
"confirm"    char(10)
...
)
set quoted_identifier off
```

defgen

Function

Creates table definition statements for tables owned by servers in classes of types *sql_server*, *generic*, *db2*, and *direct_connect* (*access_server*). This utility is used to define data located on remote servers quickly by generating the **sp_addobjectdef**, **create existing table** and **update statistics** statements.

Syntax

The parameters listed with accompanying text require a value. Those listed without text do not require or accept a value.

```
defgen [-Uusername]
       [-Ppassword]
       [-Sserver]
       [-Ddatabase]
        -sforeign_server
       [-dforeign_database]
       [-nforeign_owner]
       [-Foutput_file]
       [-V]
       [-L]
       [-Ttable_prefix]
       [-ewildcard_escape_character]
       [-v]
       [-x]
       [-o]
       [-adisplay_charset]
       [-Jclient_charset]
       [-z language]
       [[owner.]tablename [[owner.]tablename...]]
```

Parameters

-U*username* – specifies a login name for the server. Logins are case-sensitive. If no parameter is specified *username* defaults to operating system user name.

-P*password* – specifies the user's password. If no **-P** parameter is given, **defgen** prompts the user for a password. Passwords are case-sensitive and can be 1–30 characters in length or NULL.

-S*server* – specifies the name of the server to which you want to connect. This is the name that **defgen** looks for in the interfaces file. If **-S** is omitted, **defgen** looks for the server specified with your DSQUERY environment variable. If DSQUERY is not defined, SYBASE is used.

-D*database* – specifies the name of the database in which the specified tables are to be placed. If not specified, the user's default database is used.

-s*foreign_server* – specifies the name of the foreign server in the *sysservers* system table. This parameter is required.

-d*foreign_database* – specifies the name of the database in *foreign_server* that contains the specified tables. This is allowed if *foreign_server* is of class s*ql_server, generic*, or *direct_connect* (*access_server)*.

-n*foreign_owner* – qualifies the tables to be defined by selecting only those tables owned by *foreign_owner.* When specific tables are not supplied, definitions are generated for all the tables owned by *foreign_owner.* This parameter defaults to the user name used to access the foreign server. For DB2, this parameter is synonymous with authorization ID. Wildcard pattern matching characters, as supported by *foreign_server,* may be used in *foreign_owner.*

-F*output_file* – specifies the name of the operating system file in which to store the output from **defgen**. If this parameter is not used, output is directed to the file *defgen.sql.*

-V – runs **defgen** in verbose mode. Verbose mode causes **defgen** to print informational messages as it performs its tasks.

-L – specifies that all table names and column names for the tables created within the server are in lowercase.

-T*table_prefix* – specifies a string used to prefix all table definitions for tables to be created by running the script.

-e*wildcard_escape_character* – specifies a wildcard escape character to precede underscores in owner and table names when issuing **sp_tables** and **sp_columns.**

-v – prints the **defgen** version and copyright strings. No other operations are performed, and **defgen** exits.

-x – creates a trace file called *debug.*

-o– instructs **defgen** to build the **create existing table** statement using the same column order as the actual table. This is the default for all servers except DB2 servers.

-a*display_charset* – allows you to run **defgen** from a terminal whose character set differs from that of the machine on which **defgen** is running. **-a** in conjunction with **-J** specifies the character set translation file (*.xlt* file) required for the conversion. Use **-a** without **-J** only if the client character set is the same as the default character set.

-J*client_charset* – specifies that the OmniConnect convert to and from *client_charset*, the character set used on the client. **-J** with no argument sets character conversion to NULL. No conversion takes place. Use **-J** if the client and server use the same character set. Omitting **-J** sets the character set to the default for the platform. The default may not necessarily be the character set that the client is using.

-z*language* – specifies the official name of the alternate language in which to display **defgen** messages. Without **-z**, **defgen** uses the server's default language.

[[owner.]table [[owner.]table...]] – If one or more tables are listed on the command line, then table definitions are generated only for those tables. If no tables are listed, table definitions are generated for all tables.

Examples

1. **`defgen -Usa -P -SOMNI -Dtestdb -sSYBASE -dpubs -ndbo -Tsyb_ -Fsybpubs.sql`**

   Builds an output file for all tables in the *pubs* database owned by "dbo", puts them in a database called *testdb,* prefixes all table definitions with the string "syb_", and names the output file *sybpubs.sql*.

2. **`defgen -Usa -P -SOMNI -Dtestdb -sORACLE -nscott -Tora_ -Fora.sql`**

   Builds an output file for all tables owned by "scott" in a DirectConnect server called ORACLE, puts them in a database called *testdb*, prefixes all table definitions with the string "ora_", and names the output file *ora.sql*.

3. **`defgen -Usa -P -SOMNI -Ddb2_tables -sDB2 -Tdb2_ -Fdb2.sql dsn8220.act dsn8220.emp`**

Builds an output file for two tables (*dsn8220.act* and *dsn8220.emp*) in a server of class *db2* called DB2, puts them in a database called *db2_tables*, prefixes all table definitions with the string “db2_”, and names the output file *db2.sql*.

4. **`defgen -Usa -Psapw -SOMNI -Dinf_app -sINFMX -Tinf_ -Finf_app.sql inf_dbo.clients inf_dbo.sales`**

   Builds an output file for two tables (*inf_dbo.clients* and *inf_dbo.sales*) in a DirectConnect server called INFMX, puts them in a database called *inf_app*, prefixes all table definitions with the string “inf_”, and names the output file *inf_app.sql*.

5. **`defgen -Usa -P -SOMNI -Drdb_tables -sRDB -TRDB_ -Frdb.sql -dpersonnel`**

   Builds an output file for all tables in the *personnel* database in a server of class *generic* called RDB, puts them in an Adaptive Server database called *rdb_tables*, prefixes all table definitions with the string “RDB_”, and names the output file *rdb.sql*.

Comments

- The following process outlines a **defgen** session:
  - User submits **defgen** with parameters
  - **defgen** logs into -S server
  - -S server logs into -s foreign server
  - **defgen** generates script
  - User reviews script and customizes
  - User directs script to -S server or other server
- The **defgen** utility allows the System Administrator to set up a server quickly and easily. It generates the required **sp_addobjectdef**, **create existing table**, and **update statistics** commands necessary for the server to communicate with foreign database tables and views. These generated statements are placed in the output file defined by the **-F** option or in *defgen.sql* (the default).
- **defgen** logs into the server and obtains the server class of the server defined with the **-s** option. For release 10.0 and later Servers, system catalogs are queried for matching table and view information. For all other servers, **defgen** issues ODBC catalog stored procedure commands. For example:

  ```
  exec DB2...sp_tables "EMP", "DNS8220", ""
  ```

- The system catalogs in the remote server are scanned for matching entries. For each table or view found, corresponding column name, type, lengths, and NULL attributes are obtained. The native server datatypes are converted to Adaptive Server datatypes, according to the conversion tables shown in sections that follow.
- After running **defgen**, the output file can be executed by using the **isql** utility. This SQL output file updates the system catalogs with information about each remote server table that **defgen** selected from the remote server:

  ```
  isql -Uname -Ppasswd -SOMNI <defgen.sql >defgen.out
  ```
- After directing the output file to the server, the tables can be used in SQL statements, stored procedures, views, and so on, as if they were all local to the server.
- The output of **defgen** can be used to update multiple servers with the same information, allowing any number of copies of the server to access the same remote servers.

Adaptive Server Datatype Mapping

The same datatype is used in the server table definition as is found in the remote Adaptive Server.

If **defgen** encounters a user-defined datatype on the target table, it uses the underlying storage type and places the user-defined datatype within a comment.

DB2 Datatype Mapping

Table 5-2 lists the Adaptive Server datatypes that can be used for existing DB2 datatypes. Note that for the unsupported DB2 types *graphic*, *vargraphic*, and *long vargraphic*, **defgen** does not generate an error; instead, these DB2 datatype names are added to the table definition in the output file, and errors are raised when the output file is executed.

Table 5-2: DB2 to Adaptive Server default datatype mapping

DB2 Datatype	Adaptive Server Datatype
int	*int*
smallint	*smallint*
float	*float*
double precision	*float*

Table 5-2: DB2 to Adaptive Server default datatype mapping (continued)

DB2 Datatype	Adaptive Server Datatype
real	*real*
decimal	*decimal*
number	*numeric*
char(n)	*char(n)*
varchar(n)	*varchar(n)*
long varchar	*varchar(255)*
date	*datetime*
time	*datetime*
timestamp	*datetime*
graphic	Not supported
vargraphic	Not supported
long vargraphic	Not supported

Generic Datatype Mapping

The following table lists the Adaptive Server datatypes that can be used with ODBC datatypes for servers in server class *generic*.

Table 5-3: Generic to Adaptive Server default datatype mapping

ODBC Datatype	Adaptive Server Datatype
int	*int*
smallint	*smallint*
tinyint	*tinyint*
float	*float, money*
double precision	*float, money*
real	*real, smallmoney*
decimal (scale > 0 or precision >=10)	*float, money*
decimal (scale = 0 and precision <=9)	*int, float, money*
number (scale > 0 or precision >=10)	*float, money*
number (scale = 0 and precision <=9)	*int, float, money*
char	*char, varchar*
varchar	*char, varchar*

Table 5-3: Generic to Adaptive Server default datatype mapping (continued)

ODBC Datatype	Adaptive Server Datatype
date	*datetime* (time set to 12:00AM)
time	*datetime* (date set to 1/1/1900)
timestamp	*datetime*
bit	*bit*
binary	*binary, varbinary*
varbinary	*binary, varbinary*

DirectConnect Datatype Mapping

Table 5-4 lists the Adaptive Server datatypes that can be used with *direct_connect* (*access_server)* datatypes.

Table 5-4: DirectConnect default datatype mapping

ODBC Datatype	Adaptive Server Datatype
bigint	*decimal or numeric*
int	*int*
smallint	*smallint*
tinyint	*tinyint*
float	*float, money*
double precision	*float, money*
real	*real, smallmoney*
decimal	*decimal*
numeric	*numeric*
char (precision < 256)	*char, varchar*
char (precision >= 256)	*text*
varchar (precision < 256)	*char, varchar*
varchar (precision >= 256)	*text*
date	*datetime* (time set to 12:00AM)
time	*datetime* (date set to 1/1/1900)
timestamp	*datetime*
bit	*bit*
binary (precision < 256)	*binary, varbinary*

Table 5-4: DirectConnect default datatype mapping (continued)

ODBC Datatype	Adaptive Server Datatype
binary (precision >=256)	*image*
varbinary (precision < 256)	*binary, varbinary*
varbinary (precision >=256)	*image*

See Also

isql in the *Utility Programs* manual

sp_addserver, create existing table, and **sp_addobjectdef** in the *Adaptive Server Reference Manual*.

A Troubleshooting

This appendix provides troubleshooting tips for problems that you may encounter when using Component Integration Services. The purpose of this chapter is:

- To provide enough information about certain error conditions so that you can resolve problems without help from Technical Support
- To provide lists of information that you can gather before calling Technical Support, which will help resolve your problem quickly
- To provide you with a greater understanding of Component Integration Services

Error Messages and the *Troubleshooting Guide* should also be used for troubleshooting. While this appendix provides troubleshooting tips for most frequently asked Component Integration Services questions, *Error Messages* lists all error messages with a one-line recovery procedure; the *Troubleshooting Guide* provides tips on SQL Server problems that are not specific to Component Integration Services.

For the most up-to-date information on troubleshooting and technical tips, refer to Sybase's electronic services. See "Other Sources of Information" on page -xvii.

Problems Accessing Component Integration Services

If you issue a command that accesses a remote object and Component Integration Services is not found, the following error message appears:

```
4050 cis extension not enabled or installed
```

Do the following:

- Verify that the **enable cis** configuration parameter is set to 1 by running:

  `sp_configure "enable cis"`

  sp_configure returns the following row for the **enable cis** parameter:

```
name                   min      max      config value     run value
enable cis              0       1            1                1
```

Both “config value” and “run value” should be 1. If both values are 0, set the **enable cis** configuration parameter to 1, and restart the server. Use the syntax:

```
sp_configure "enable cis" 1
```

If “config value” is 1 and “run value” is 0, the **enable cis** configuration parameter is set, but will not take effect until the server is restarted.

- Check the error log. If Component Integration Services loaded correctly, you will see the following line at the start of the error log:

  ```
  Distributed services option loaded.
  ```

  If there was a problem loading Component Integration Services, the message stating the problem is displayed instead. Contact Sybase Technical Support to correct the problem. (See “If You Need Help” on page A-8.)

Problems Using Component Integration Services

This section provides tips on how to correct problems you may encounter when using Component Integration Services.

Unable to Access Remote Server

When you cannot access a remote server, the following error message is returned:

```
11206 Unable to connect to server server_name.
```

The message will be preceded by one of the following Client-Library messages:

```
Requested server name not found
Driver call to connect two endpoints failed
Login failed
```

The Client-Library message indicates why you cannot access the remote server as described in the following sections.

Requested Server Name Not Found

The server is not defined in the interfaces file when the following messages display:

```
Requested server name not found
11206 Unable to connect to server server_name.
```

When a remote server is added using the **sp_addserver** stored procedure, the interfaces file is not checked. It is checked the first time you try to make a connection to the remote server. To correct this problem, add the remote server to the interfaces file that is being used by Component Integration Services.

Driver Call to Connect Two Endpoints Failed

If the remote server is defined in the interfaces file, but no response was received from the connect request, the following messages are displayed:

```
Driver call to connect two endpoints failed
11206 Unable to connect to server server_name.
```

Check the following:

- Is your environment set up correctly?

  To test this, try to connect directly to the remote server using **isql** or a similar tool. Do this by following these steps:

  - Log into the machine where Component Integration Services is running.
  - Set the SYBASE environment variable to the same location that was used when Component Integration Services was started. Component Integration Services uses the interfaces file in the directory specified by the SYBASE environment variable, unless it is overridden in the *runserver* file by the **-i** argument.

➤ ***Note***

These first two steps are important to ensure that the test environment is the same environment that Component Integration Services was using when you could not connect to the remote server.

  - Use **isql** or a similar tool to connect directly to the remote server.

  If the environment is set up correctly and the connection fails, continue through this list. If the connection is made, there is a problem with the environment being used by Component Integration Services.

- Is the remote server up and running?

Log into the machine where the remote server is located to verify the server is running. If the server is running, continue through this list. If the server is hung, restart the server and try your query again.

- Is the entry for the remote server in the interfaces file correct:
  - Is the machine name the correct name for the machine the software is loaded on?
  - If the interfaces file is a text file, do the query and master lines start with a tab and not spaces?
  - Is the port number available? Check the *services* file in the */etc* directory to ensure that the port number is not reserved for another process.

    If the port is available, is it already in use? To determine this on UNIX, run the command:

netstat -a

Login Failed

If the remote server is accessed, but the login name and password are not correct, the following messages display:

```
Login failed
11206 Unable to connect to server server_name.
```

Check to see if there is an external login established for the remote server by executing:

```
exec sp_helpexternlogin server_name
```

If no external login is defined, Component Integration Services uses the user login name and password that was used to connect to Adaptive Server. For example, if the user connected to Adaptive Server using the “sa” account, Component Integration Services uses the login name “sa” when making a remote connection. Unless the remote server is another Adaptive Server, the "sa" account probably does not exist, and an external login must be added using **sp_addexternlogin**.

If an external login is defined, verify that the user’s login name is correct. Remote server logins are case sensitive; for example, DB2 logins are all uppercase. Is the case correct for the user login name you are using and the entry in *externlogins*?

If the login name is correct, the password might be incorrect. It is not possible to display the password. If the user login name is incorrect or if the password might be incorrect, drop the existing external login and redefine it by executing the commands:

```
exec sp_dropexternlogin server_name, login_name
go
exec sp_addexternlogin server_name, login_name,
remote_login, remote_password
go
```

Unable to Access Remote Object

When you are unable to access a remote object, the following error message appears:

```
Error 11214  Remote object object does not exist.
```

The problem may be in the local proxy table definition or in the table itself on the remote server.

Verify the following:

- Has the object been defined in Component Integration Services?

  To confirm, run:

  ```
  sp_help object_name
  ```

  If the object does not exist, create the object in Component Integration Services (see “Mapping Remote Objects to Local Proxy Tables” on page 3-4).

- If the object has been defined in Component Integration Services, is the definition correct?

  Table names can have four parts with the format *server.dbname.owner.tablename*. The *dbname* part is not valid for DB2, Oracle or InfoHUB servers.

  If the object definition is incorrect, delete it using **sp_dropobjectdef**, and define correctly using **sp_addobjectdef**.

- If the local object definition is correct, check the table on the remote server:
  - Are permissions set to allow access to both the database and table?
  - Has the database been marked suspect?
  - Is the database available?
  - Can you access the remote table using a native tool (for example, SQL on Rdb or SQL*Plus on Oracle)?

Problem Retrieving Data From Remote Objects

When you receive error messages pertaining to mismatches in remote objects, the Component Integration Services object definition does not match the remote object definition. This happens if:

- The object definition was altered outside of Component Integration Services
- An index was added or dropped outside of Component Integration Services

Object Is Altered Outside Component Integration Services

Once an object is defined in Component Integration Services, alterations made to an object at the remote server are not made to the local proxy object definition. If an object is altered outside of Component Integration Services, the steps to correct the problem differ, depending on whether **create existing table** or **create table** was used to define the object.

To determine which method was used to define the object, run the statement:

```
sp_help object_name
```

If the object was defined via the **create existing table** command, the following message is returned in the result set:

```
Object existed prior to CIS.
```

If this message is not displayed, the object was defined via the **create table** command.

If **create existing table** was used to create the table in Component Integration Services:

1. Use the **drop table** command in Component Integration Services.
2. Create the table again in Component Integration Services using **create existing table.** This creates the table using the new version of the table on the remote server.

If the table was created in Component Integration Services using **create table**, you will drop the remote object when you use **drop table**. To prevent this, follow these steps:

1. Rename the table on the remote server so the table is not deleted when you use **drop table**.
2. Create a table on the remote server using the original name.

3. Use **drop table** in Component Integration Services to drop the table in Component Integration Services and on the remote server.
4. Rename the saved table in step 1 with its original name on the remote server.
5. Create the table again in Component Integration Services using **create existing table.**

◆ ***WARNING!***

Do not use drop table in Component Integration Services prior to renaming the table on the remote server, or you will delete the table on the remote server.

A good rule to follow is to create the object on the remote server, and then do a **create existing table** to create the object in Component Integration Services. This enables you to correct mismatch problems with fewer steps and with no chance of deleting objects on the remote server.

Index Is Added or Dropped Outside CIS

Component Integration Services is unaware of indexes that are added or dropped outside Component Integration Services. Verify that the indexes used by Component Integration Services are the same as the indexes used on the remote server. Use **sp_help** to see the indexes used by Component Integration Services. Use the appropriate command on your remote server to verify the indexes used by the remote server. For example, you can use the **describe** command with an Oracle server or **select * from syscolumns, sysindexes** for a DB2 server.

If the indexes are not the same, the steps to correct the problem differ, depending on whether **create existing table** or **create table** was used to define the object.

To determine which method was used to define the object, run the statement:

```
sp_help object_name
```

If the object was defined via the **create existing table** command, the following message is returned in the result set:

```
Object existed prior to CIS.
```

If this message is not displayed, the object was defined via the **create table** command.

If **create existing table** was used to create the object:

1. Use **drop table** in Component Integration Services.
2. Re-create the table in Component Integration Services using **create existing table**. This will update the indexes to match the indexes on the remote table.

If **create table** was used to create the object:

1. Use **drop table** to drop the index from the remote table.
2. Re-create the index in Component Integration Services using **create index**. This creates the index in Component Integration Services and the remote server.

An alternative method if **create table** was used to define the object is to turn on trace flag 11208. This trace flag prevents the **create index** statement from transmitting to the remote server. To use trace flag 11208, follow these steps:

1. Turn on trace flag 11208:

   dbcc traceon(11208)

2. Create the index using **create index**.
3. Turn off trace flag 11208:

   dbcc traceoff(11208)

If You Need Help

If you encounter a problem that you cannot resolve using the manuals, ask the designated person at your site to contact Sybase Technical Support. Gather the following information prior to calling Technical Support to help resolve your problem more quickly.

- If a problem occurs while you are trying to access remote data, execute the same script against a local table. If the problem does not exist on the local table, it is specific to Component Integration Services and you should continue through this list.
- Find out what version of Component Integration Services you are using:

  ```
  select @@cis_version
  ```

- Note the SQL script that reproduces the problem. Include the script that was used to create the tables.

- Find the processing plan for your query. This is generated using **set showplan.** An example of this is:

  ```
  set showplan, noexec on
  go
  select au_lname, au_fname from authors
      where au_id = ‘A1374065371’
  go
  ```

  The output for this query will look like this:

  ```
      STEP1
      The type of query is SELECT.
      FROM TABLE
      authors
      Nested iteration
      Using Clustered Index
  ```

  The **noexec** option compiles the query, but does not execute it. No subsequent commands are executed until **noexec** is turned off.

- Obtain the event logging when executing the query by turning on trace flags 11201 – 11205. These trace flags log the following:

  - 11201 – Client connect, disconnect, and attention events
  - 11202 – Client language, cursor declare, dynamic prepare, and dynamic execute-immediate text
  - 11203 – Client rpc events
  - 11204 – Messages routed to client
  - 11205 – Interaction with remote servers

  After executing the script with the trace flags turned on, the logging is found in the error log in the *$SYBASE/install* directory. For example:

  ```
  dbcc traceon (11201,11202,11203,11204,11205)
  go
  select au_lname, au_fname from authors
      where au_id = 'A1374065371'
  go
  dbcc traceoff (11201,11202,11203,11204,11205)
  go
  ```

  The error log output is as follows (the timestamps printed at the beginning of each entry have been removed to improve legibility):

```
server  LANGUAGE, spid 1: command text:
select au_lname, au_fname from authors where au_id
= 'A1374065371'
server  SIGDISABLE, spid 1: signals disabled on
endpoint 10
server  RMT_CONNECT, spid 1: connected to server
'SYBASE', using language/charset
'us_english.iso_1', packet size 512
server  SYB_TSCN, spid 1, server SYBASE:
SELECT au_id, au_lname, au_fname FROM
pubs2.dbo.authors WHERE au_id = "A1374065371"
server  OMNIENDS, spid 1: closing cursor 'O1_16'
server  OMNICLOS, spid 1: deallocating cursor
'O1_16', type CONNECTION.
```

This tracing is global, so once the trace flags are turned on, any query that is executed will be logged; therefore, turn tracing off once you have your log. Also, clean out the error log periodically by bringing the server down, renaming the error log, and restarting the server. This creates a new error log.

Index