Adaptive Server™ Specialty Data Store™ Developer's Kit

SDK Release 11.5
Document ID: 30508-01-1150-01
Last Revised: June 13, 1997

Principal author: Jim Cluett and Lori Johnson

Document ID: 30508-01-1150

This publication pertains to SDK Release 11.5 of the Sybase database management software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

Document Orders

To order additional documents, U.S. and Canadian customers should call Customer Fulfillment at (800) 685-8225, fax (617) 229-9845.

Customers in other countries with a U.S. license agreement may contact Customer Fulfillment via the above fax number. All other international customers should contact their Sybase subsidiary or local distributor.

Upgrades are provided only at regularly scheduled software release dates.

Sybase Trademarks

Restricted Rights

Table of Contents

About This Book

1. Introduction

2. Interface Topics

3. SQL Commands

4. Text and Image Handling

5. System and Catalog RPCs

Index

List of Tables

About This Book

This book describes the interface between the Adaptive Server™ *sds* server class and an implementation of an Open Server™ application. Although this book refers to Adaptive Server throughout, the rules to build a Specialty Data Store™ that interfaces with OmniConnect™ are the same.

Audience

Read this book if you are responsible for developing and providing Specialty Data Stores. This book is written for experienced Open Server application developers.

How to Use This Book

This book is divided into the following chapters:

- Chapter 1, "Introduction," defines a Specialty Data Store and provides an overview of Open Server event handling.
- Chapter 2, "Interface Topics," describes various interface topics, including Specialty Data Store requirements for Open Server event handling, error handling, and datatypes.
- Chapter 3, "Transact-SQL Commands," describes the syntax of the subset of Transact-SQL™ and DB2 SQL generated within Adaptive Server and transmitted to a Specialty Data Store.
- Chapter 4, "Text and Image Handling," describes the mechanism used by Adaptive Server to handle *text* and *image* datatypes when interacting with a Specialty Data Store.
- Chapter 5, "RPCs," contains reference pages for system administration, text and image and catalog RPCs, that are generated by Adaptive Server.

Related Documents

The following manuals provide additional background. If you are not familiar with Open Server and Adaptive Server, first study the following manuals:

- *SYBASE Open Server Server-Library/C Reference Manual*

- *SYBASE Open Client Client-Library/C Reference Manual*
- *Open Client and Open Server Common Libraries Reference Manual*
- *SYBASE Adaptive Server Reference Manual, Volumes 1 and 2.*
- *Component Integration Services User's Guide for Sybase Adaptive Server and OmniConnect*

Other Sources of Information

Electronic Information Sources

For the most up-to-date information on Sybase® products, including information on availability, certifications, bugs, and troubleshooting, refer to Sybase's electronic services:

- AnswerBase™, Sybase's CD-ROM knowledge base
- Sybase OpenLine and PrivateLine, the Sybase forums on CompuServe
- The Support*Plus*℠ Online Services and SyBooks-on-the-Web World Wide Web pages

Support*Plus* Online Services and SyBooks-on-the-Web

To get to Support*Plus* Online Services and Sybooks-on-the-Web:

1. Connect to the Sybase home page: *www.sybase.com*.
2. Follow links to Services and Support.
3. Follow links to Sybase Enterprise Technical Support.
4. Follow links to Support*Plus* Online Services or SyBooks-on-the-Web.

SyBooks-on-the-Web is accessible to the public.

Support*Plus* Online Services is intended for customer use only. Therefore, you must register to access Support*Plus* Online Services.

Conventions

Formatting SQL Statements

SQL is a free-form language: there are no rules about the number of words you can put on a line or where you must break a line. However, for readability, all examples and syntax statements in this manual are formatted so that each clause of a statement begins on a new line. Clauses that have more than one part extend to additional lines, which are indented.

SQL Syntax Conventions

The conventions for syntax statements in this manual are as follows:

Table 1: Syntax statement conventions

Key	Definition
command	Command names, command option names, utility names, utility flags, and other keywords are in **`bold Courier`** in syntax statements, and in **bold Helvetica** in paragraph text.
variable	Variables, or words that stand for values that you fill in, are in italics.
`{ }`	Curly braces indicate that you choose at least one of the enclosed options. Do not include braces in your option.
`[ ]`	Brackets mean choosing one or more of the enclosed options is optional. Do not include brackets in your option.
`( )`	Parentheses are to be typed as part of the command.
`|`	The vertical bar means you may select only one of the options shown.
`,`	The comma means you may choose as many of the options shown as you like, separating your choices with commas to be typed as part of the command.

- Syntax statements (displaying the syntax and all options for a command) are printed like this:

  `sp_dropdevice [`*`device_name`*`]`

  or, for a command with more options:

```
select column_name
    from table_name
    where search_conditions
```

In syntax statements, keywords (commands) are in normal font and identifiers are in lowercase: normal font for keywords, italics for user-supplied words.

- Examples showing the use of Transact-SQL commands are printed like this:

```
select * from publishers
```

- Examples of output from the computer are printed like this:

```
pub_id  pub_name                city         state
------- --------------------    ------------ -----
0736    New Age Books           Boston       MA
0877    Binnet & Hardley        Washington   DC
1389    Algodata Infosystems    Berkeley     CA

(3 rows affected)
```

Case

You can disregard case when you type keywords:

```
SELECT is the same as Select is the same as select
```

Adaptive Server's sensitivity to the case (upper or lower) of database objects, such as table names, and data depends on the sort order installed on your Adaptive Server. Case sensitivity can be changed for single-byte character sets by reconfiguring Adaptive Server's sort order. (See the *System Administration Guide* for more information.)

Obligatory Options {You Must Choose At Least One}

- **Curly Braces and Vertical Bars**: Choose **one and only one** option.

```
{die_on_your_feet | live_on_your_knees |
live_on_your_feet}
```

- **Curly Braces and Commas**: Choose one or more options. If you choose more than one, separate your choices with commas.

```
{cash, check, credit}
```

Optional Options [You Don't Have to Choose Any]

- **One Item in Square Brackets:** You don't have to choose it.

  ```
  [anchovies]
  ```

- **Square Brackets and Vertical Bars:** Choose n**one or only one**.

  ```
  [beans | rice | sweet_potatoes]
  ```

- **Square Brackets and Commas**: Choose **none, one, or more than one** option. If you choose more than one, separate your choices with commas.

  ```
  [extra_cheese, avocados, sour_cream]
  ```

Ellipsis: Do It Again (and Again)...

An ellipsis (...) means that you can **repeat** the last unit as many times as you like. In this syntax statement, **buy** is a required keyword:

```
buy thing = price [cash | check | credit]
    [, thing = price [cash | check | credit]]...
```

You must buy at least one thing and give its price. You may choose a method of payment: one of the items enclosed in square brackets. You may also choose to buy additional things: as many of them as you like. For each thing you buy, give its name, its price, and (optionally) a method of payment.

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

What is a Specialty Data Store?

A Specialty Data Store allows Adaptive Server clients to access external data as if it were stored locally in the Adaptive Server. A Specialty Data Store is an Open Server application designed to interface with the Adaptive Server.

The Adaptive Server allows data from Specialty Data Stores to be joined with data from the local server, a remote server, or with other Specialty Data Stores. When a client sends a SQL statement to the Adaptive Server, the server will parse the SQL and send each Specialty Data Store the appropriate portion of the original statement. The Adaptive Server then creates a result set based on the results from all servers involved in the query. A Specialty Data Store is only concerned about access to the data it manages.

The Specialty Data Store interface defines the behavior expected of a Specialty Data Store. This includes a subset of SQL and a set of remote procedure calls that will be generated by Adaptive Server.

What is the Specialty Data Store Developer's Kit?

The Specialty Data Store Developer's Kit (SDK) contains the following items to assist developers in creating a Specialty Data Store:

- This manual.
- A working sample Specialty Data Store program. This Specialty Data Store allows a Unix or Windows NT file system to be accessed through the Adaptive Server. Using this Specialty Data Store you can:
  - Search a file system for content
  - Alter a file's content using Transact-SQL
  - Import files into the Adaptive Server

  This example implements all of the major interfaces required for a Specialty Data Store, and is structured so that it can be used as the framework for a Specialty Data Store you design.

One important piece of this example program is a generalized Transact-SQL parser that can be used as is, or modified for your specific requirements.

Specialty Data Store Example

In this example, the Specialty Data Store allows Adaptive Server clients to access spreadsheet files. The Specialty Data Store presents each spreadsheet to the Adaptive Server as if it were a Sybase table. The columns and rows of the spreadsheet are mapped to a virtual table that the Specialty Data Store presents to the Adaptive Server. The Specialty Data Store maps each cell of the spreadsheet to an Adaptive Server datatype. For example, a cell containing text will be mapped to the *char* datatype. Cells containing decimal values will be mapped to the *decimal* datatype.

The Adaptive Server client will see the spreadsheets in the Adaptive Server system catalogs along with all of the tables actually stored in the Adaptive Server. The client can perform most SQL operations on the spreadsheet tables, and the spreadsheet tables can be used in join operations between tables stored within the Adaptive Server, or tables stored in other Specialty Data Stores.

Figure 1-1 shows how a Specialty Data Store allows access to spreadsheets:

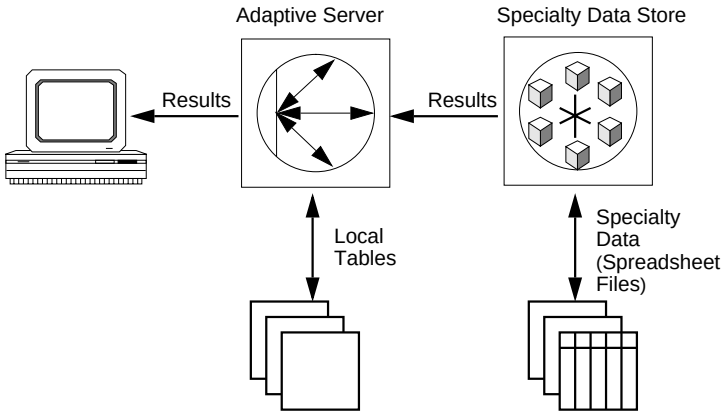

Figure 1-1: Specialty Data Store providing access to spreadsheet files

Designing a Model for your Specialty Data Store

The basic function of a Specialty Data Store is to present to the Adaptive Server a view of externally stored data in the form of a relational table. The Specialty Data Store creates the view of a virtual table on top of externally stored data in the Adaptive Server. The first step in developing your Specialty Data Store is to create this model. Once the design of the model is complete, the implementation of SQL operations such as **select** will become clearer and easier.

When the Specialty Data Store creates a virtual table view of data, it can create additional information about the data and map that into the table view. For example, the virtual view of a text document might contain a column with actual text data, and columns with calculated data such as verb count and noun count. Using calculated columns is a way to incorporate special data search and calculation functions provided by an external data storage mechanism.

Another example of using calculated columns is a Specialty Data Store used to implement access to a text storage and search system. The text storage system has a function called **relevance()** that searches a document for words that are relevant to a word. The **relevance()** function returns a floating point value as a score.

A virtual table called *text_table* might be created for the documents that looks like:

Column	Datatype
data	*text*
relevance	*char(200)*
relevance_rating	*float*

An Adaptive Server user could then enter a query such as:

```
select data, relevance_rating from text_table
    where relevance = “bananas”
    and relevance_rating > 3
```

This query would select text segments where the *relevance_rating* to “bananas” was greater than 3.

The *relevance* column is used to pass an input value to the **relevance()** function provided by the text storage system. The *relevance* column would not actually return anything if it were included in the select list of a **select** statement. It exists as a means of passing a search parameter in a **where** clause. The calculated result of the **relevance()** function would be returned in the *relevance_rating* column.

When designing a model for your Specialty Data Store, consider the capabilities of the Specialty Data Store interface and what makes sense for your particular data store. With the simplest set of capabilities, a Specialty Data Store has to provide the Adaptive Server with the means of scanning tables with cursors. Only simple **and** conditions are required to be processed by the Specialty Data Store.

If a Specialty Data Store does not provide complex capabilities such as **group by**, the Adaptive Server will compensate and perform the calculation itself after retrieving the qualifying rows for the query. The best performance is seen when the Adaptive Server is able to off-load as much of the query as possible to the Specialty Data Store.

Consider the following query:

```
select sum(a) from t1
```

If the Specialty Data Store supports the **aggregate** capability, a cursor is opened to the Specialty Data Store containing the complete statement:

```
select sum(a) from t1
```

The Specialty Data Store processes the cursor and returns a single row in the result set. If, however, the **aggregate** capability is not supported, the Adaptive Server performs the **sum** calculation itself.

A cursor is opened with the following text:

```
select a from t1
```

Each row of the table is returned to the Adaptive Server and the **sum** is calculated. Although this method yields the correct answer, it significantly increases network overhead.

The designer of the Specialty Data Store should consider the types of queries a client is likely to perform when deciding on the capability level of the Specialty Data Store. In some cases it makes sense to treat specialty data as read-only data. This makes development of a Specialty Data Store significantly easier since **insert**, **update**, and **delete** processing does not need to be implemented. If you choose this, the Specialty Data Store should be defined to the Adaptive Server as a read-only server. This can be done with the **sp_server_option** procedure in the Adaptive Server.

The Specialty Data Store designer should also consider whether data definition language (DDL) should be supported. Some Specialty Data Stores will never create new tables and indexes. The Adaptive Server is simply accessing an existing remote database.

Sample Specialty Data Store

The sample Specialty Data Store (*filesds*) included in the SDK contains the source code for a Specialty Data Store that allows access to file systems. The model of the *filesds* Specialty Data Store maps the specialty data as follows:

- File system directories are presented as tables
- Each row in the table represents a file in the directory
- There are columns for the file name, information on the owner, size, and a *text* column for the contents of the file

The contents of the file can be accessed and updated using **readtext** and **writetext** commands. Files can be created and deleted using SQL **insert** and **delete** statements. Content of files can be searched using a **select** statement with a **like** clause. Files can also be imported into the Adaptive Server using a **select into** command.

The sample Specialty Data Store implements all of the major interfaces and procedures that are required, and provides a framework for the Specialty Data Store that you are developing.

Using the *filesds* Specialty Data Store

The *filesds* sample can be used to access files in your file system from the Adaptive Server after installing it and configuring the Adaptive Server. It is distributed in source form and must be compiled and linked with Open Server. The sample is located in the following directories:

- *sdsdk/sample/filesds* - the code for the *filesds* Specialty Data Store
- *sdsdk/sample/parser* - the code for the Specialty Data Store parser

Installing the *filesds* Specialty Data Store

To install the *filesds* sample, follow these steps:

1. Install Open Server and setup a server name and address in the Sybase interfaces file.
2. Make sure the SYBASE environment variable points to the Open Server installation directory. It is used by the makefile to find include files and libraries.
3. Go to the *sdsdk/sample/parser* directory. The parser uses generated .c and .h files from the UNIX LEX and YACC

programs. A version of these generated files is already present that will work for Windows NT users. The generated files that are supplied were built using AIX.

4. Run the platform-specific shell to rebuild the parser library. For example, on UNIX platforms use:

   `build.`*`unix_platform`*

   or, on a Windows NT platform use:

   `nmake -f make.nt`

5. Build the sample Specialty Data Store. Go to the *sdsdk/sample/filesds* directory. Makefiles are available for each platform. Run the makefile for your platform. For example, on Sun Solaris use:

   `make -f make.sun`

6. Copy the sample configuration file to the *$SYBASE* directory. Rename it based on the name of your server. For example, for a server named "FOO", rename the file to *FOO.cfg*. Open the *FOO.cfg* file and edit the section in brackets changing [FILESDS] to [FOO].

7. Edit the **textdirectory** configuration parameter to point to the topmost directory where the Specialty Data Store should search for files. This directory should contain one or more subdirectories. The subdirectories should contain files you are interested in accessing through the Adaptive Server. The following is an example directory structure:

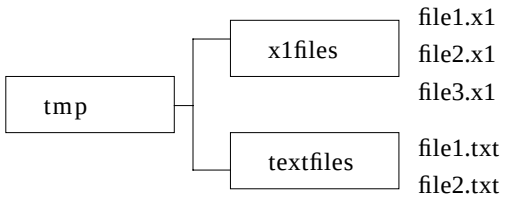

Figure 1-2: Example directory structure

The setting for this directory structure is:

textdirectory = /tmp

Using the *filesds* Specialty Data Store to Access Data

Once *filesds* is installed, Adaptive Server can be configured to allow access to files in your file system. The following steps take you through configuring Adaptive Server to access files in the example directory structure shown in Figure 1-2 on the server *FOO*.

1. Start *filesds*. On UNIX, use the command:

   ```
   textsds -SFOO &
   ```

   On Windows NT, use the command:

   ```
   start textsds -SFOO
   ```

2. Add the server *FOO* to the Adaptive Server using the following command:

   ```
   sp_addserver FOO, sds
   ```

3. You should now be able to see a list of tables available through the Specialty Data Store by issuing the command:

   ```
   FOO...sp_tables
   ```

   This produces a list of tables. Each of these tables is a subdirectory underneath the *tmp* directory configured in **textdirectory.**

4. Create proxy tables for the files in your directory structure. For example, create a proxy table for the *x1files* subdirectory by issuing the command:

   ```
   sp_addobjectdef xlfiles, "FOO...xlfiles"
   ```

   This creates a mapping between the Adaptive Server table called *xlfiles* and the table managed by the Specialty Data Store also called *xlfiles*.

5. Create the *xlfiles* table using:

   ```
   create existing table xlfiles
   (
       filename    varchar(30),
       owner       varchar(16),
       size        int,
       content     image
   )
   ```

A table now exists in Adaptive Server that maps to your file system directory called *xlfiles*. The sample *filesds* is written to support only this specific table format. You can now access and manipulate data in *xlfiles*. The following are examples of this.

1. `select filename, size from xlfiles`

   This lists all of the files with their sizes in the *xlfiles* directory.

2. `select filename, textptr(content) from xlfiles`

   This lists filenames and text pointers that can be used with **readtext** and **writetext** to manipulate the file.

3.
```
select filename, content from xlfiles
    where size < 1500
```

   This returns files that are less than 1500 bytes. **isql** does not support selecting more than 1500 bytes of *text* and *image* data. Data over 1500 bytes should be accessed with **readtext** and **writetext** when using **isql**.

Sample Program Information

The sample *filesds* is provided as a sample that implements most of the Specialty Data Store interface. The primary goal is to provide a framework for building your Specialty Data Store. The following sections provide a description of the files included with the sample program.

Sample Program Code

The code is separated into components that are generic to almost all Specialty Data Stores, and the code is implementation specific.

You must modify the following two files:

- *procs.c* - this code implements the catalog stored procedures. In the *filesds* Specialty Data Store, the code looks through the contents of directories to present a file system as if it were a relational database.
- *access.c* - this code implements a query. In this implementation, the parser output is looked at and file I/O routines are called to search directories, update files, etc. These functions are called from cursor, dynamic, language, and bulk events.

Sample Data Structures

The file *sds.h* contains the definition of the following structures used in the sample:

- CLIENT structure - this structure contains information associated with each client or Open Server SRVPROC. For this Open Server

application, a client is always an Adaptive Server connection. It contains information such as the login and password of the client, a list of declared cursor and dynamic statements, and general state information. The structure exists from the time the Adaptive Server first connects until the time it disconnects.

- CURSOR structure - this structure is allocated each time a cursor is declared. It contains the cursor's statement, any parameters for the open cursor, parsing information, and the result set of the cursor.
- PARSBLK structure - this structure is used by the parser for storing its output. It has allocated memory attached to it. A PARSBLK structure is contained in each CLIENT structure and each CURSOR structure. The PARSBLK structure in the CLIENT structure is used for parsing language events.

Sample Modules

The following is a description of the sample source code:

- *main.c* - this contains the program's main entry point. This is mainly Open Server initialization code.
- *connect.c* - this contains the event handler for the connect event. You will want to modify this to perform required security checks. The client's login and password are available.
- *attention.c* - this is the Open Server attention event handler. It sets a status in the CLIENT structure indicating an attention has occurred. This status should be checked periodically when returning a large result set.
- *bulk.c* - this is the Open Server bulk event handler. Bulk events are also used by **writetext** operations. When a **writetext** is issued to an Open Server, a language event will occur that contains the **writetext** command. Immediately after the language event, a bulk event will occur with the data associated with the **writetext** command. The code in the bulk event can then read the data either in whole or in parts.
- *client.c* - these are functions used to maintain the CLIENT structure. It includes enqueue and dequeue functions.
- *config.c* - this contains code for reading a configuration file.
- *cursors.c* - this contains the Open Server cursor event. The code in this file contains the basic functions to respond to cursor events. Functions outside this module are called to actually process the

queries in the cursors. Most Specialty Data Stores can use the cursor code in this file as is. The access functions it calls will need to be tailored to your Specialty Data Store.

- *dynamic.c* - this contains the Open Server dynamic event handler. Dynamic events are used by the Adaptive Server for **insert**, **update**, and **delete** statements. This code is also a basic framework which calls other functions for the actual query processing.
- *lang.c* - this contains the Open Server language event handler. All DDL and transaction commands will come to this event.
- *rpc.c* - this contains the Open Server remote procedure call event handler. This function looks at the RPC name and calls an appropriate handling function.
- *procs.c* - this contains the functions for handling the RPC's that a Specialty Data Store is required to support. Much of this code is specific to the *filesds* Specialty Data Store.
- *error.c* - this is the Open Server error handler. It logs error messages to the log file. There is also a function to send an error message to an Adaptive Server client. This probably will not need to be altered for your application.
- *disc.c* - this is the Open Server disconnect event handler. This function is called whenever an Adaptive Server client closes a connection to the Specialty Data Store. This function must cleanup all resources acquired during the connection.
- *access.c* - this contains query processing functions for the Specialty Data Store. This code is very specific for the *filesds* Specialty Data Store, however, you may want to copy the basic framework.
- *fileio.c* and *fileiont.c* - these contain functions for manipulating files. They are probably only useful for this sample.
- *globals.c* - this contains various global variables used by the *filesds* Specialty Data Store.
- *hash.c* - this contains functions for mapping files to text pointers. When a Specialty Data Store manufactures a text pointer value, it must be capable of using the value to calculate the location of the original row and column. These functions perform this mapping as well as save the hashed values in a lookup file.

Sample Configuration File

The sample configuration file is named *FILESDS.cfg*. Use this as the framework for your configuration file by copying the sample configuration file to the $SYBASE directory. Rename it based on the name of your server. For example, for a server named “FOO”, rename the file to *FOO.cfg*.

Sample Parser

Since the Adaptive Server sends requests to a Specialty Data Store in SQL format, you must have a parser to process requests. A sample parser is included in the s*dsdk/sample/parser* directory. This parser is based on the UNIX YACC and LEX utilities.

Processing for subqueries, **union**, and expressions are the only major capabilities not supported in the sample parser. Table 1-1 lists the capabilities of this parser as they relate to the **sp_capabilities** RPC. You can choose not to implement all of the supported capabilities; however, if you implement capabilities that are not supported, changes must be made to the parser.

Table 1-1: Sample parser support of capabilities

ID	Capability Name	Value
101	sql syntax	1 = Transact-SQL
102	join handling	1 = no outer joins
103	aggregate handling	1 = ANSI 89 support level
104	**and** predicates	1 = supported
105	**or** predicates	1 = supported
106	**like** predicates	1 = ANSI-style supported
107	bulk insert handling	0 = not supported
108	text / image handling	2 = text with textptr
109	transaction handling	1 = local transactions supported
110	text pattern handling	0 =not supported
111	order by	1 = supported
112	group by	1 = ANSI SQL compatible
113	net password encryption	0 = not supported
114	object name case sensitivity	0 = case insensitive

Table 1-1: Sample parser support of capabilities

ID	Capability Name	Value
115	distinct	1 = supported
117	union	0 = not supported
118	string functions	0 = not supported
119	expression handling	0 = not supported
120	truncate blanks	0 = do not truncate trailing blanks
121	language handling	1 = all queries supported except those containing dates
122	date functions	0 = not supported
123	math functions	0 = not supported
124	convert function	0 = not supported
125	T-SQL delete / update	0 = multiple tables not supported
126	insert select	0 = not supported
127	subquery support	0 = not supported

The following describes the function of each of the parser files.

- *parser.h* - this contains the parser structures and definitions that are visible to the caller of the parser. This file must be included by the caller of the function **sqlpars()**.
- *parser.l* - this contains the parser's LEX code.
- *parser.y* - this contains the parser's YACC code.
- *sqlpars.c* - this contains the main entry point into the parser, **sqlpars()**.
- *zpars.c* - this contains routines that are called by *parser.l* and *parser.y*. The routines in this file populate the user visible parser structures.
- *zpars.h* - this contains internal parser definitions and should **not** be included by the caller of **sqlpars()**.

The caller of the parser must include the file *parser.h* and declare a PARSBLK structure. When **sqlpars()** is called, the caller passes the address of the PARSBLK structure and the address of a Transact-SQL statement. If the parser succeeds, the PARSBLK structure contains information about the parsed statement. Refer to the *parser.h* file for the layout of the PARSBLK structure.

Building a Specialty Data Store

Building a Specialty Data Store requires several steps that should be implemented in stages so that they can be tested as they are developed. Your Specialty Data Store is going to consist of an Open Server application with a number of event handlers (refer to "Specialty Data Store Components" on page 1-15 for an overview of the event handlers). Sybase recommends building these handlers in the following order. For a more detailed description of the RPCs and commands used in these milestones, refer to Chapter 3, "SQL Commands," and Chapter 5, "System and Catalog RPCs."

Milestone 1: *connect to* Command

Passthrough connections can be made between the client and the Specialty Data Store using the **connect to** command in the Adaptive Server when this milestone is completed. In passthrough mode, requests made by an Adaptive Server client are sent directly to the Specialty Data Store without processing by Adaptive Server. The Specialty Data Store then processes the passthrough queries.

To reach this milestone:

1. Implement the **sp_capabilities** RPC. This RPC provides the Adaptive Server with a list of capabilities the Specialty Data Store supports.
2. Implement the **sp_thread_props** RPC. Adaptive Server uses this RPC to notify the Specialty Data Store of changes in the condition of a connection. The thread property *passthrough_mode* is used to notify the Specialty Data Store that the connection is a passthrough connection.
3. Implement a minimal language handler. When an Adaptive Server client enters passthrough mode, all commands are sent to the Specialty Data Store as language events.

Milestone 2: Table Definition

When passthrough connections are established, you have an interesting gateway, but the power of the Specialty Data Store is in providing a view of external data to the Adaptive Server. Completing this milestone enables a user to create tables in the Adaptive Server which serve as proxies for remote data. Once proxy

tables are created, they work as an alias for the external table and can be used as if they were regular local tables.

To reach this milestone:

1. Implement the **sp_tables** RPC. Adaptive Server uses this to verify the existence of a remote table. It also verifies that the table name uniquely identifies a table.
2. Implement the **sp_columns** RPC. This procedure is used to verify the proxy table definition in the Adaptive Server is correct. The result set sent by the Specialty Data Store describes each column of the table.
3. Implement the **sp_statistics** RPC. **sp_statistics** describes the indexes that exist on the table. If your Specialty Data Store will not support indexes, this procedure should not return a result set.
4. Execute a **create existing table** command to define a proxy table to the Adaptive Server.

Milestone 3: Read-Only Access

Now that proxy tables can be defined, implement the following interface items so that an Adaptive Server client can perform **select** statements on proxy tables managed by the Specialty Data Store:

1. Your minimal language event should only return "done" for all transaction requests and DDL statements at this time. Additional support can be added later if desired.
2. Implement the cursor event. Assuming the most basic capabilities, the Adaptive Server will send **select** requests as cursor events. Refer to the Specialty Data Store sample program as a guide in implementing this.

Milestone 4: Insert, Update, and Delete

After you can successfully read data stored in the Specialty Data Store, implement the following to enable you to **insert**, **update**, and **delete** data:

1. Implement the dynamic event handler. This event will be used whenever **insert**, **update**, and **delete** statements can be passed entirely to the Specialty Data Store for executing.
2. Positioned updates and deletes through cursors should be a small extension to your cursor event handling. Cursor updates

and deletes are select cursors, followed by one or more fetches, followed by a cursor update or delete event.

Milestone 5: Text and Image Handling

The next step is to add support of *text* and *image* datatype handling. See Chapter 4, "Text and Image Handling" for a complete description of the RPCs and functions that you can choose to implement.

Milestone 6: Data Definition Language and Transaction Management

If you choose to support data definition language (DDL) and transaction management, you must provide extensions to your language event handling to handle DDL and transaction commands.

Specialty Data Store Components

An Open Server application is mainly composed of a set of event handlers. Each time the Adaptive Server makes a request, an Open Server event is generated. Refer to the *Open Server Server-Library/C Reference Manual* for a detailed description of these events. The Open Server events that a Specialty Data Store must be prepared to process are described in the following sections.

SRV_ATTENTION

An attention event is raised when Adaptive Server cancels a request for data. This is usually due to a cancel request that was received from the Adaptive Server's client.

The Specialty Data Store must stop results processing and cancel all processing associated with the connection when it receives SRV_ATTENTION.

SRV_BULK

The bulk event is used during *text* and *image* handling and during bulk insert events. When Adaptive Server sends a **writetext** command to the Specialty Data Store, the data associated with the command is

sent through a bulk event. See Chapter 4, “Text and Image Handling” for more details on *text* and *image* handling.

SRV_CONNECT

The connect event is raised when Adaptive Server connects to the Specialty Data Store.

The Specialty Data Store should validate the login request and make a connection to the corresponding, underlying data source, if appropriate. One connection is made on behalf of each client that references a remote object. The connection is maintained until the transaction is complete. The 11.5 version of Adaptive Server will leave the connection active until the Adaptive Server client disconnects.

SRV_CURSOR

Adaptive Server uses the cursor event to process **select**, **update**, **delete**, and possibly **readtext**. There are several cursor commands that can be sent to a Specialty Data Store. These commands are described in the following table:

Table 1-2: Cursor commands sent to a Specialty Data Store

Command	Description
declare	Associates a cursor name with the body of the cursor.
open	Executes the body of the cursor, and generates a cursor result set.
information	Reports the status of the cursor, or sets the cursor row fetch count.
fetch	Fetches rows from the cursor result set.
update or **delete**	Updates or deletes the contents of the current cursor row.
close	Makes the cursor result set unavailable. Re-opening a cursor regenerates the cursor result set.
deallocate	Renders the cursor non-existent. A cursor that has been deallocated cannot be re-opened.

None of these commands contain embedded data values as part of the command string. Instead, all data values are passed to the Specialty Data Store as cursor parameters when the cursor is opened.

➤ ***Note***

Adaptive Server attempts to send most SQL requests as language events if the **language** capability is set through **sp_capabilities**. Adaptive Server may also send **insert**, **update**, and **delete** statements as dynamic statements. Adaptive Server will do this when it determines that a statement can be completely passed off to a Specialty Data Store, and the rows do not need to be viewed by the server.

SRV_DISCONNECT

A disconnect event is generated when the Adaptive Server disconnects from the Specialty Data Store. Adaptive Server closes all connections on behalf of a client when that client disconnects from the server.

It is the responsibility of the Specialty Data Store to terminate connections to foreign databases, and to free resources such as memory that have been allocated for connection.

SRV_DYNAMIC

Adaptive Server generates a dynamic event in order to send a parameterized statement to the Specialty Data Store. Dynamic events do not generate a result set. There are several dynamic commands that can be sent. These commands are described in the following table:

Table 1-3: Dynamic SQL commands sent to a Specialty Data Store

Command	Description
prepare	Prepare a statement for execution
describe input	Request input parameter formats for the current prepared statement
describe output	Request column formats for the current prepared statement
execute	Execute a prepared statement
execute immediate	Execute an unprepared statement, which has no parameters and does not return results
deallocate	Deallocate a prepared statement

Adaptive Server uses dynamic SQL requests to pass **insert**, **update** and **delete** commands to a Specialty Data Store. These statements never contain embedded data values as part of the command string. Instead data values are sent as parameters when the statement is executed.

SRV_LANGUAGE

Adaptive Server sends language requests to a Specialty Data Store to process a number of data definition statements. The syntax of these commands is described in Chapter 3, "SQL Commands." Language events are also used for **begin transaction**, **commit transaction**, **prepare transaction**, **rollback transaction**, **readtext**, and **writetext**. Depending on how the language capability is set, additional SQL may be available.

SRV_RPC

The RPC event is raised in response to a remote procedure call (RPC) request from the Adaptive Server.

Adaptive Server issues RPCs for several reasons:

- To determine the existence and structure of a table and its indexes. The RPCs **sp_tables**, **sp_columns**, and **sp_statistics** are used for this purpose.
- To obtain information about the capabilities of the Specialty Data Store.
- To request *text* and *image* information from the Specialty Data Store
- When the **execute** command is processed and the RPC name contains a remote server name.

A description of each RPC that can be generated by Adaptive Server is described in Chapter 5, "System and Catalog RPCs."

Debugging a Specialty Data Store

The following are tips for debugging a Specialty Data Store:

- Use Adaptive Server trace flag 11205 to see the events being sent from Adaptive Server. To set the trace flag, use:

  ```
  dbcc traceon(11205)
  ```

- Add logging statements to the top of each Open Server event handler.
- You can enable Open Server network logging in the sample program through the *network_tracing* configuration parameter. This information is low level but may be helpful.

2 Interface Topics

Overview

This chapter describes the following interaction between Adaptive Server and the Specialty Data Store:

- Adaptive Server Configuration
- Specialty Data Store Connect Handling
- Specialty Data Store Capabilities
- Specialty Data Store Language Handling
- Specialty Data Store RPC Handling
- Specialty Data Store Cursor Handling
- Specialty Data Store Bulk Copy Handling
- Specialty Data Store Thread Properties
- *text* and *image* Handling
- Transaction Management
- Passthrough and Stand-alone Modes
- Datatypes
- Error Handling

Adaptive Server Configuration

Before Adaptive Server can interact with a Specialty Data Store, Adaptive Server must be configured. First configure the remote servers definition, then configure the remote table definition. See the *Component Integration Services User's Guide* for more information on configuring the Adaptive Server.

Remote Server Definition

The Adaptive Server 11.5 supports a server class called *sds*. This access method generates the requests described in this manual. These requests and their expected responses define the nature of the Adaptive Server Specialty Data Store interface. It is assigned when the remote server is configured.

Remote servers are configured within Adaptive Server by using the system procedure, **sp_addserver.** The parameter for *server_class* must be included.

The following server classes are supported by Adaptive Server:

- *local* - indicates that the server name is the local server. This name appears in the *@@servername* global variable.
- *sql_server* - the remote server is a SQL Server. Adaptive Server determines whether the SQL Server is a System10 or later, or pre-System10 version, and use the appropriate access method. Note that the System10 interface uses cursors and dynamic SQL events, rather than language events, for all DML statements.
- *db2* - the remote server is an Open Server application serving as a gateway to a DB2 (or compatible) RDBMS
- *generic* - the remote server is an Open Server application that conforms to the interface specification for a Generic Access Module.
- *direct_connect*- the remote server is a Sybase DirectConnect™.
- ***sds*** - the remote server is an Open Server application that conforms to the interface specification for a Specialty Data Store, as described in this book.

➤ ***Note***

Only the *sds* interface is documented in this book.

The syntax for the system procedure **sp_addserver** is:

sp_addserver *server_name,server_class* [,*network_name*]

where:

- *server_name* - the name used to identify the server. It must be unique.
- *server_class* - one of the supported server classes defined above. If this is set to *local*, then *netname* is ignored.
- *network_name* - the server name contained within the interfaces file. This name may be the same as *server_name*, or it may differ.

For more information on **sp_addserver**, see the *Adaptive Server Reference Manual*.

Logging in to Remote Servers

Once the remote server is configured within Adaptive Server, login information needs to be considered. By default, Adaptive Server uses the name and password of its clients whenever it needs to connect to a remote server on behalf of those clients. However, this default can be overridden by the use of the system procedure **sp_addexternlogin**. This procedure allows a System Administrator to define the name and password to be used by Adaptive Server when connecting to a remote server on behalf of a particular user. For more information on **sp_addexternlogin**, see the *Adaptive Server Reference Manual*.

Adaptive Server stores all passwords in encrypted form. If a Specialty Data Store has been configured with password security (see "Specialty Data Store Capabilities" on page 2-8), and the server option *net password encryption* has been set by the Adaptive Server System Administrator, then Adaptive Server transmits password information to that Specialty Data Store in encrypted form.

The Adaptive Server extension to Transact-SQL, **connect to *server_name***, lets you verify that the Adaptive Server configuration is correct. This command establishes a passthrough mode connection to the remote server. This passthrough mode remains in effect until you issue a **disconnect** command.

Remote Table Definition

After the remote server is configured, objects in that remote server cannot be accessed by Adaptive Server as tables until the remote location and type are mapped to a local Adaptive Server object, and the table is defined.

There are two methods of defining the storage location of remote objects. The first method defines a location for all objects in a database, while the second defines the location of individual objects. Only one of these methods is required.

Defining the Storage Location of Individual Objects

Use the system procedure **sp_addobjectdef** to define individual object storage locations. This procedure allows the user to associate a remote object name with a local Adaptive Server table name. The remote object may or may not exist at this time. The syntax for **sp_addobjectdef** is as follows:

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

where:

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

  ```
  dbname.owner.object
  ```

  where:

  - *dbname* is the local database name (optional).
  - *owner* is the local owner name (optional).

  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. When *object_type* is *table*, *view*, or *rpc*, *object_loc* takes the form:

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

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).
- *owner* is the name of the remote server user that owns the remote object (optional).
- *object* is the name of the remote *table*, *view*, or *rpc*.

- *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.

For more information on **sp_addobjectdef**, see the *Adaptive Server Reference Manual*.

Defining the Storage Location For All Database Objects

Use the system procedure **sp_defaultloc** to define the storage location for all objects in a given database. The remote objects may or may not already exist. The syntax for **sp_defaultloc** is:

```
sp_defaultloc database_name,"storage_location"
    [, object type]
```

where:

- *database_name* is the name of the database within Adaptive Server to which the default location is to be applied.
- *storage_location* defines the location in a remote server or directory with which Adaptive Server tables are associated. The syntax of the *storage_location* parameter is:

  `server_name.database_name.owner"`
- *object_type* is the type of remote object. It can be *table*, *view*, or *rpc*. This parameter is optional; the default is *table.*

For more information, see the *Adaptive Server Reference Manual*.

create [existing] table

Once the storage location is defined, the table can be created as a new or existing object. If the table does not currently exist, use the **create table** syntax. If it already exists, use the **create existing table** syntax. If the object type is *rpc*, the object must be defined using **create existing table**.

When a **create table** or **create existing table** statement is received by Adaptive Server, and the object type is either *table* or *view*, the existence of the remote object is checked by means of the catalog RPC **sp_tables**. If the object exists, then its column and index attributes are obtained using the RPCs **sp_columns** and **sp_statistics**, respectively. Column attributes are compared with those defined for the object in the **create existing table** command. Column name, type, length and null property are checked. Index attributes are added to Adaptive Server's *sysindexes* system table.

Once the object is created, either as a new or existing object, the remote object can be queried by referencing its local name.

Refer to the reference pages in Chapter 3, "SQL Commands," regarding **create table** and **create index**, and to the references pages in Chapter 5, "System and Catalog RPCs," for the catalog RPCs used by Adaptive Server.

Specialty Data Store Connect Handling

Each Specialty Data Store responds in some manner to a connect request issued by Adaptive Server. Each Adaptive Server thread may have different thread properties, which must be processed by the Specialty Data Store connect handler.

Connection Properties

A number of connection properties are of particular interest to the Specialty Data Store during connection processing. The following properties are available to the Specialty Data Store developer by means of **srv_thread_props():**

- User name – the name of the user issuing the connect request. The Specialty Data Store should use this, in conjunction with the user password, to validate the login with the remote DBMS. The thread property is SRV_T_USER.
- Password – the user password; may or may not be encrypted (see "Non-negotiated Logins" on page 2-6). The thread property is SRV_T_PWD.
- Language – the language used by the Adaptive Server. The Specialty Data Store sets this language based on what it has received from the client. The thread property is SRV_T_LOCALE. It must be combined with **cs_locale**(..., **CS_GET**, **CS_SYB_LANG**...).
- Character set – the character set the Adaptive Server uses. The value sent will be the default character set of the Specialty Data Store. The thread property is SRV_T_LOCALE. It must be combined with **cs_locale**(..., **CS_GET**, **CS_SYB_CHARSET**...).
- Application name – the Adaptive Server sets this name to **omniserver**. The thread property is SRV_T_APPLNAME.
- Remote server name - the name of the remote server that the Adaptive Server is connecting to. When setting up the Sybase interfaces file, you can associate multiple server names with the same network port. The client system views each entry as a separate server, but in fact each name can be routed to the same server. This flexibility can be used to have different servers with different behavior, yet still only have one server program running. The Specialty Data Store can look at the name of the server the Adaptive Server is connecting to and adjust its behavior accordingly. This name is available through the thread property SRV_T_RMTSERVER.

Non-negotiated Logins

By default, Adaptive Server does not perform negotiated logins; login requests are transmitted via Client-Library CT_SRV_T_RMTSERVER calls and no security information is negotiated. However, if the server to which Adaptive Server is

connecting is configured to require *net password encryption*, a negotiated login is attempted.

Negotiated Logins

Adaptive Server may send passwords to the Specialty Data Store in encrypted form if:

- The Specialty Data Store has indicated that it can support them by means of the *security* capability (see “sp_capabilities” on page 5-3).
- The Adaptive Server has been configured with *net password encryption* for the server involved.

If both of these conditions are true, the Specialty Data Store must check to determine the type of negotiated login that is being requested, and respond accordingly. The following negotiated logins are possible:

- ENCRYPT – indicates that the client wishes to pass an encrypted password. This must be supported by the Specialty Data Store if the *security* capability indicates that password encryption is supported.
- CHALLENGE – indicates that the client wishes to negotiate via a challenge / response exchange. This type of negotiation cannot be initiated by Adaptive Server.
- SECLABEL – indicates that the client sends security labels. This is not used by Adaptive Server.
- APPDEFINED – indicates that an application-defined login handshake is in use. At this time, there is no negotiation defined between the Adaptive Server and Specialty Data Store.

➤ ***Note***

The standard Sybase client-server encryption algorithm is used to handle password encryption. The algorithm used is a one-way encryption algorithm; therefore, if the Specialty Data Store needs to convey password information to a third party DBMS, then the Specialty Data Store is responsible for the mapping between the user name and the remote password.

Specialty Data Store Capabilities

The first time Adaptive Server establishes a connection to a Specialty Data Store, it issues an RPC named **sp_capabilities** and expects a set of results in return. This result set must describe certain capabilities of the Specialty Data Store so that Adaptive Server can adjust its interaction with that Specialty Data Store to take advantage of available features. For more information on **sp_capabilities**, refer to "sp_capabilities" on page 5-3.

The capabilities of interest to Adaptive Server are:

SQL Syntax (101)

This indicates the syntax of SQL that the Adaptive Server will generate. Sybase recommends that Transact-SQL be used since this is the only dialect supported by the SDK parser. The possible values are:

- 1 = Sybase Transact-SQL syntax
- 2 = IBM DB2 syntax

Join Handling (102)

This indicates the Specialty Data Store capability for joining two or more tables in a single query. The possible values are:

- 0 = cannot handle joins of any kind
- 1 = can handle normal joins, but not outer joins
- 2 = can handle all joins
- 3 = Oracle join behavior

Aggregate Handling (103)

This indicates the Specialty Data Store capability for handling row aggregates. The aggregate functions are **sum()**, **min()**, **max()**, **count()** and **avg()**. The possible values are:

- 0 = aggregates not supported.
- 1 = ANSI SQL aggregate handling. **count(*colname*)** is not supported (only **count(*)** is supported). Additionally, aggregates and non-

aggregates cannot appear on the target list together unless the query contains a **group by.**

- 2 = Transact-SQL aggregate behavior.

and Predicates (104)

This indicates the Specialty Data Store can process search conditions connected via **and** operators. Specialty Data Stores **must** handle the **and** clause. The only value supported is:

- 1 = supported

or Predicates (105)

This indicates the Specialty Data Store can process search conditions connected via **or** operators. The possible values are:

- 0 = not supported
- 1 = supported

like Predicates (106)

Describes the level of support for the **like** predicate. Three possible values are supported:

- 0 = **like** is unsupported
- 1 = compatible with ANSI SQL
- 2 = compatible with Sybase Transact-SQL

If the Specialty Data Store can process the **like** pattern search condition, the pattern can contain the following special characters, according to the support level indicated:

Table 2-1: Pattern matching characters supported by like

Character	Description
%	zero or more of any characters (levels 1 and 2)
_	any one character (levels 1 and 2)
[ ]	range of characters (level 2 only)
^	characters not in range (level 2 only)

bulk insert Handling (107)

This indicates whether the Specialty Data Store can handle bulk insert requests. The possible values are:

- 0 = not supported
- 1 = supported

➤ ***Note***

If *text* or *image* is supported with text pointers, then the SRV_BULK event is also raised during *writetext* requests.

text and *image* Handling (108)

Indicates the level of support for *text* and *image* data (refer to Chapter 4, "Text and Image Handling" for a complete discussion of *text* and *image* handling). The possible values are:

- 0 = not supported
- 1 = *text* and *image* is supported, but text pointers are not supported
- 2 = *text* and *image* and text pointers are supported

Transaction Handling (109)

This indicates the Specialty Data Store capability for handling transactions. Two levels of support are possible:

- 0 = no transaction support
- 1 = minimum transaction support (**begin**, **prepare**, **commit**, and **rollback**)

Text Pattern Handling (110)

Indicates whether the **patindex** function is supported. The possible values are:

- 0 = not supported
- 1 = supported

order by (111)

This indicates the Specialty Data Store capability for handling the **order by** clause. The possible values are:

- 0 = not supported
- 1 = supported

group by (112)

This indicates the Specialty Data Store capability for handling the **group by** clause. The possible values are:

- 0 = not supported
- 1 = ANSI SQL compatible
- 2 = Transact-SQL **group by** (**group by** "all", aggregates and non-aggregates in the target list)

Net Password Encryption (113)

This indicates the Specialty Data Store capability for handling encrypted passwords. The possible values are:

- 0 = not supported
- 1 = supported

Object Name Case Sensitivity (114)

This indicates whether object names are case sensitive. The possible values are:

- 0 = object names are case insensitive
- 1 = object names are case sensitive

distinct Handling (115)

This indicates the Specialty Data Store capability for handling the **distinct** keyword. The possible values are:

- 0 = **distinct** not supported
- 1 = **distinct** supported

union Support (117)

This indicates whether the Specialty Data Store can handle a **union** statement in a **select**. The possible values are:

- 0 = **union** not supported
- 1 = **union** supported

String Functions (118)

This indicates the list of string functions supported by a Specialty Data Store. The possible values are:

- 0 = no string functions are supported
- 1 = only the **substr()** function is supported
- 2 = **substr()**, **lower()**, **ltrim()**, **rtrim()**, and **upper()** functions are supported
- 3 = all Transact-SQL string functions are supported

Expression Handling (119)

This indicates the expression handling capability of the Specialty Data Store. Expressions are statements consisting of column names, constants, operators, and parenthesis. Constants can appear as parameter markers depending on the language handling capability (121). Expressions can be nested. The possible values are:

- 0 = minimal expressions are supported. Only column names are supported in this mode. The select list of a **select** statement may also contain the constant value of "1". For example:

  ```
  select 1 from t1 where a = b
  ```

- 1 = ANSI SQL compatible expressions only. These include column names, constants, +, -, /, *, and the functions defined by the math function (123) and string function (118) capabilities. For example:

  ```
  select a + 3 from t1 where a > (b - (c - 2))
  ```

- 2 = Transact-SQL expressions (This includes ANSI SQL compatible expressions, modulo, bit operators, and other Transact-SQL extensions).

Truncate Blanks (120)

Normally *char* parameters can contain trailing blanks that pad the data to its defined length. This option truncates trailing blanks on parameters. The possible values are:

- 0 = do not truncate trailing blanks
- 1 = truncate trailing blanks

Language Handling (121)

This indicates the level of SQL parsing supported by the Specialty Data Store. If this capability is set, the Adaptive Server assumes that it does not need to use cursors and dynamic statements for all requests. It is more efficient to send SQL requests as language events instead of cursor or dynamic events. Unlike cursor and dynamic events, datatype values are not parameterized. Instead, they are expanded to the normal Transact-SQL syntax of a datatype. The possible values are:

- 0 = do not use language events for other than transaction control, **readtext**, **writetext**, and DDL statements.
- 1 = able to parse SQL as long as *date* datatypes are not present. Language events will be used when the query does not involve *date* datatypes.
- 2 = able to parse all SQL with any datatype. Use language events whenever possible.

Date Functions (122)

This indicates the list of date functions supported by a Specialty Data Store. The possible values are:

- 0 = no date functions are supported
- 1= all Transact-SQL date functions are supported

Math Functions (123)

This indicates the list of math functions supported by a Specialty Data Store. The possible values are:

- 0 = no math functions are supported

- 1= **abs**, **cos**, **exp**, **floor**, **power**, **round**, **sign**, **sin**, **sqrt**, and **tan** functions are supported
- 2= all Transact-SQL math functions are supported

convert Function (124)

This indicates support for the **convert()** function. The possible values are:

- 0 = **convert()** function is not supported
- 1= **convert()** function is supported

Transact-SQL *deletelupdate* (125)

This indicates support for the Transact-SQL capability of having multiple tables in an **update** or **delete from** clause. The possible values are:

- 0 = Multiple tables are not supported
- 1=Multiple tables are allowed in **from** clause

insert select (126)

This indicates the capability to handle a **select** clause within an **insert** statement. The possible values are:

- 0 = **insert select** is not supported.
- 1 = ANSI SQL is supported. No **group by** or **order by** is supported.
- 2 = Transact-SQL is supported.

Subquery Support (127)

This indicates whether the Specialty Data Store can handle a subquery in a SQL statement. The possible values are:

- 0 = subqueries are not supported
- 1 = ANSI SQL is supported
- 2 = Transact-SQL is supported, except **group by** and **order by** in a subquery is not supported

Specialty Data Store Language Handling

Specialty Data Stores are required to handle a core set of SQL requests as language events. In some cases, these requests do not make sense for a particular data store. In this case, the Specialty Data Store can reply with an error message.

The following SQL commands are sent to a Specialty Data Store as language events:

- **alter table**
- **begin transaction**
- **commit transaction**
- **create index**
- **create table**
- **drop index**
- **drop table**
- **insert bulk**
- **prepare transaction**
- **rollback transaction**
- **truncate table**
- **writetext bulk**

The syntax for each command is described in Chapter 3, “SQL Commands”.

Specialty Data Store RPC Handling

A Specialty Data Store must be capable of handling RPC requests from the Adaptive Server. In addition, the Specialty Data Store may provide support for user-generated requests issued through the Adaptive Server in response to the **execute** command.

There are four categories of RPCs that the Adaptive Server issues:

- Catalog RPCs
- *text* and *image* handling RPCs
- Administrative RPCs
- User-generated RPCs that are specific to your Specialty Data Store implementation

Catalog RPCs

Catalog RPCs are defined to enable a DBMS-independent manner of accessing system catalogs. The RPCs enable a common interface to the dictionary of the underlying DBMS supported by the Specialty Data Store, without forcing the client program (Adaptive Server) to know anything about the means of doing so.

The required catalog RPCs are:

- **sp_columns**
- **sp_statistics**
- **sp_tables**

These RPCs are described in Chapter 5, “System and Catalog RPCs”.

text and *image* Handling RPCs

If a Specialty Data Store indicates that it supports *text* and *image* datatypes, then it must also support the following RPCs to enable Adaptive Server to extract information regarding the *text* or *image* column:

- **sp_textvalid**
- **sp_patindex**
- **sp_datalength**
- **sp_char_length**

These RPCs are described in Chapter 4, “Text and Image Handling” and Chapter 5, “System and Catalog RPCs”.

Administrative RPCs

The administrative RPCs are:

- **sp_capabilities**
- **sp_thread_props**

These RPCs are described in Chapter 5, “System and Catalog RPCs”.

User-generated RPCs

An Adaptive Server user can request the execution of a remote procedure by means of the **execute** command. If the RPC name is of the form:

SERVER...rpc_name

then Adaptive Server issues an RPC to the named server, passing any arguments that are provided. The datatype of the arguments can be any supported Adaptive Server datatype, including *text* and *image*.

Specialty Data Store Cursor Handling

The following SQL commands are issued from the Adaptive Server to a Specialty Data Store by means of cursors:

- **select**
- **update**
- **delete**
- **readtext**

In each case, the cursor commands used are **declare**, **open**, **fetch** (**select** only) and **close**. **deallocate** is also used to terminate the cursor.

Data is sent by parameters. For example, the **where** clause of a **select** statement contains host variable identifiers in the command text, and Adaptive Server sends parameters for each host variable. The parameters are sent in the native Open Server datatype, so that the Specialty Data Store does not have to interpret the type from a character string.

An example of this is:

```
select c3 from table_1 where c1 = @p1 and c2 = @p2
```

In this example, *@p1* and *@p2* are parameters, and the Specialty Data Store should expect to receive two cursor parameters at the time the cursor is opened.

Similarly, the values of the **set** clause of an **update** statement are passed as parameters:

```
update t1 set c1 = @p1, c2 = @p2 where [current of
    cursor name]
```

Each cursor can be re-used, with different parameter values, after it is closed and before it is deallocated. Thus, a pre-compiled cursor

that has been declared can be opened as many times as necessary without forcing cursor re-compilation.

➤ ***Note***

The parameters are indicated in Transact-SQL syntax by the "@" character, while in DB2 syntax, each parameter is indicated by a "?" with no following characters.

Specialty Data Store Dynamic Event Handling

insert, **update** and **delete** commands are sent from Adaptive Server to a Specialty Data Store by means of SRV_DYNAMIC events. In each case, the dynamic statement is first prepared, and input parameters are described. The statement is then executed at least once with actual data values for each parameter marker. A request to execute a prepared statement can be issued many times, each time with separate data values.

When Adaptive Server is finished with the command, it deallocates the statement.

In each case, Adaptive Server issues appropriate calls to Client-Library to make the requests, and the Specialty Data Store must provide appropriate responses in its SRV_DYNAMIC event handler.

The syntax of the dynamic commands sent to a Specialty Data Store during the **prepare** phase is described in Chapter 3, "SQL Commands".

Specialty Data Store Bulk Copy Handling

There are two categories of bulk requests transmitted from Adaptive Server to a Specialty Data Store using the SRV_BULK event:

- **bulk insert** into table
- *text* and *image* **write** operations

bulk insert Into Table

If the Specialty Data Store has indicated that it can support bulk inserts, Adaptive Server generates **bulk insert** commands in the following cases:

- The target table of Adaptive Server's **select into** is owned by a Specialty Data Store
- Adaptive Server receives a **bulk copy** in request to a table that is mapped to a Specialty Data Store

Adaptive Server uses standard Client-Library **bulk copy** functions to initiate the bulk transfer to the Specialty Data Store. This requires that the Specialty Data Store respond to the bulk request as described in the following sections.

Bulk Copy Initialization

The **bulk copy** request is initialized by Adaptive Server by a call to the **blk_init()** function. This function sends a language request to the Specialty Data Store in the form:

insert bulk *table_name*

The Specialty Data Store must respond to this in its SRV_LANGUAGE event handler as follows:

- Record the bulk type internally by calling **srv_thread_props()** with *cmd* set to CS_SET, *property* set to SRV_T_BULKTYPE, and *bufp* pointing to a value of SRV_BULKLOAD.
- Allocate a bulk descriptor structure, CS_BLKDESC, by calling **blk_alloc()**.
- Initialize the CS_BLKDESC structure. This would normally be done by a call to **blk_init()**, in which a connection to an Adaptive Server is expected. In the case of a Specialty Data Store, the initialization must be performed by the Specialty Data Store application logic.
- Initialize the Specialty Data Store half of the exchange with a call to **blk_srvinit()**.

Bulk Transfer

Once the proper response to a bulk initialization request has been made, the Specialty Data Store receives bulk copy rows from Adaptive Server in the form of SRV_BULK events, and must respond to these events as follows:

- Allocate a row buffer using **blk_rowalloc()**
- Call **blk_getrow()** until there are no more rows

- For each row, **blk_colval()** provides the data value of a given column in the row
- For each row, an appropriate **insert** command must be issued to transfer the row to the target DBMS
- The function **blk_rowdrop()** must be called to free the row buffer when processing is complete.
- The function **blk_drop()** should be called to deallocate the CS_BLKDESC structure

➤ ***Note***

Refer to *Open Client and Open Server Common Libraries Reference Manual* for a complete description of the functions required to implement this behavior.

Bulk Copy Events for *text* and *image* Data

If the Specialty Data Store indicates that it fully supports *text* and *image* data, Adaptive Server generates bulk events to send *text* and *image* data. Adaptive Server uses the standard Client-Library function **ct_send_data()** to send *text* and *image* data to a Specialty Data Store. This function causes both language and bulk events.

For more information, see Chapter 4, "Text and Image Handling," and the *Open Client and Open Server Common Libraries Reference Manual*.

Specialty Data Store Thread Properties

All requests to modify thread-specific options are transmitted from Adaptive Server to the Specialty Data Store by means of the **sp_thread_props** RPC. In particular, the following options can be set by Adaptive Server using this RPC, when one of its clients has specified a change:

- *rowcount* – this limits the number of rows returned from a Specialty Data Store to Adaptive Server
- *textsize* – this limits the size of *text* or *image* data returned from the Specialty Data Store in a single fetch

- *passthrough mode* – informs the Specialty Data Store that its client wishes to enter or exit passthrough mode (no SQL transformation is to be performed)

update and *delete* Handling

The query optimizer of Adaptive Server examines each **update** and **delete** statement it receives, and determines whether the entire statement can be reconstructed for sending to a remote server, or whether it must first examine the data involved before the **update** or **delete** can occur.

If the entire statement can be reconstructed for transmission to a remote server, a dynamic **update** or **delete** occurs. The remote server receives a complete statement in which the **set** and **where** clauses are parameterized.

If the Adaptive Server must first examine the rows involved in an **update** or **delete**, a cursor is opened which issues a **select ... for update**. The Adaptive Server will then generate a cursor **update** or **delete** event for the row made current by the cursor.

In both cases, host variables, or parameter markers, are used to define the right side of expressions in the **set** and **where** clauses. For example, in the case of dynamic **update** or **delete**, the following syntax is used:

```
update t1 set c1 = ? where c2 = ?
delete t1 where c1 = ?
```

In the case of cursor updates, the following syntax is passed to the Specialty Data Store:

```
update t1 set c1 = @p1
```

The clause **where current of cursor...** is assumed, and not transmitted by Adaptive Server.

Parameters

When a table is created in the Adaptive Server through the **create existing table** command, the server sends an **sp_columns** request for a description of the data to the Specialty Data Store. As part of this result set, the Specialty Data Store sends back a *remote datatype* column. The value in the column is stored in the Adaptive Server's system catalogs. Whenever the Adaptive Server sends a parameter that represents a value for that column, it puts the *remote datatype*

value into the user-defined datatype field of the parameter. If no *remote datatype* was returned, or the table was created through a **create table** command, the *remote datatype* value will be zero.

This feature allows the Specialty Data Store to deal with cases in which there is an ambiguous mapping of remote datatypes to Sybase datatypes. The Specialty Data Store can look at the user-defined datatype field to get the datatype that the column was originally derived from.

Here is an example of how this might be used with a Specialty Data Store for accessing DB2. DB2 has a *date*, *time*, and *timestamp* datatype, while Sybase uses the *datetime* datatype for expressing both date and time. When the Specialty Data Store describes the column with **sp_columns**, it returns a *remote datatype* value that indicates whether the column was originally *date*, *time*, or *timestamp*. When a parameter is received from the Adaptive Server that references this column, the user defined datatype field tells the Specialty Data Store the original datatype. The Specialty Data Store can then read the parameter and format a DB2 SQL statement with the correct format for the DB2 datatype.

The values for *remote datatype* are defined entirely by the Specialty Data Store. The Adaptive Server simply stores the value and resends it in the user-defined datatype field of a parameter.

Transaction Management

Adaptive Server manages transactions for its clients, if the transactions involve work done in Specialty Data Stores that support transactions. In order to do so effectively, the following commands, sent as language events, must be supported by the Specialty Data Store if the Specialty Data Store indicates that it supports transaction management:

- **begin transaction**
- **commit transaction**
- **prepare transaction**
- **rollback transaction**

While release 11.5 of Adaptive Server does not support a full two-phase commit capability, it attempts to narrow the window of opportunity for failure when trying to commit a distributed unit of work.

When Adaptive Server attempts to commit work performed by all servers involved in a particular transaction, it first broadcasts a **prepare transaction** command to all servers involved in the transaction. The purpose of this is to verify that all servers are still active. If all servers respond positively, then the **commit transaction** command is broadcast to each server serially.

If the **prepare** command results in a failure return status, then all servers involved in the transactions are issued a **rollback** command.

Passthrough Mode

Passthrough mode is requested by the Adaptive Server in response to a client request to establish passthrough operation to a Specialty Data Store.

If an Adaptive Server client wishes to enter into a passthrough dialog with the Specialty Data Store, Adaptive Server sends the following RPC command to the Specialty Data Store:

```
sp_thread_props “passthru mode”, 1
```

When the Adaptive Server client wishes to leave passthrough mode, Adaptive Server issues the RPC:

```
sp_thread_props “passthru mode”, 0
```

While in passthrough mode, the Specialty Data Store can provide responses to incoming language commands in whatever manner is appropriate for the DBMS supported by the Specialty Data Store.

This procedure is called so the Specialty Data Store can change its behavior because of the knowledge the SQL is being generated by an end user instead of by the Adaptive Server. For instance, a security check may need to be performed.

Datatypes

Datatype compatibility must be considered when:

- Adaptive Server passes datatypes to the Specialty Data Store during a **create table** or **alter table** statement
- Adaptive Server queries the Specialty Data Store to determine the datatypes of existing columns in a remote table during a **create existing table** statement
- Parameters are used to pass a data value from a Specialty Data Store to Adaptive Server, as part of a DML statement

create table or *alter table*

Each time a user defines a column with the **create table** or **alter table** statement, a datatype for the column must be provided. Adaptive Server reconstructs the **create table** and **alter table** statements using the syntax described in Chapter 3, "SQL Commands," and passes commands to the Specialty Data Store. The Specialty Data Store must be able to transform the commands into a form that the underlying DBMS recognizes.

Adaptive Server uses the datatype of the Adaptive Server column when it reconstructs these commands; thus the commands will contain all Adaptive Server datatypes.

create existing table

When a **create existing table** command is processed, Adaptive Server ensures that the columns of the existing remote table are compatible with the columns being defined for the table within Adaptive Server. The following checks are made for each column:

- The datatypes of the remote table's columns must be compatible with those of the table being defined within Adaptive Server
- The column length defined for columns of types *char*, *varchar*, *binary*, and *varbinary* must match those of corresponding columns of the remote table
- Scale and precision of columns of type *numeric* or *decimal* must match those of the corresponding column in the remote table
- If the remote table column allows NULL values, then the Adaptive Server definition must allow NULL values
- If the remote table column does not allow NULL values, then the Adaptive Server definition must not allow NULL values
- The count of columns in the remote table must match the Adaptive Server's definition

The datatypes of the columns in the existing table are queried via the catalog RPC **sp_columns**.

DML Statements

Adaptive Server passes data values as parameters to either a cursor or dynamic SQL statement. The parameters are in the datatype native to the Adaptive Server, and must be converted by the

Specialty Data Store into formats appropriate for the underlying DBMS.

Adaptive Server only transmits data values as part of the text of a SQL command when the language capability indicates it can be handled. Datatype information is passed in the CS_DATAFMT structure associated with the parameter. The following fields of the structure contains datatype information:

- *datatype* – the CS_LIB type representing the data. For example, CS_INT_TYPE.
- *usertype* – the native DBMS datatype. This type is passed to Adaptive Server during a **create existing table** command as part of the result set from **sp_columns** (see "sp_columns" on page 5-7). Adaptive Server returns this type in the *usertype* field to assist the Specialty Data Store in type conversions.

Result Rows

When a Specialty Data Store returns result rows as part of processing a **select** command, it has more flexibility in the datatypes it uses. The Adaptive Server must be able to convert the datatype of the result column into the datatype described by **sp_columns**. These conversion rules are determined by the capabilities of the **cs_convert** function documented in the Open Server manual set.

For example, a Specialty Data Store might describe a column by **sp_columns** as being a *float* datatype. When a **select** is processed by the Specialty Data Store, it can choose to return that column as a *char* datatype as long as the string could be converted to a *float*, i.e., "123.456".

For the best performance, it is recommended that result rows be returned in the format that they are described by in **sp_columns**. This eliminates a conversion step.

Error Handling and Messaging

Adaptive Server does not attempt to interpret or transform an error message when it receives one from a remote server. All error or informational messages received from a remote sever are passed on to the Adaptive Server client that caused the message to be generated.

Whether or not Adaptive Server treats the results of a command as an error depends on the done status it receives from the remote server, and not the error message itself. If the done packet contains an error status, Adaptive Server treats the results as though an error occurred, and issues a backout or rollback as appropriate for the command.

3 SQL Commands

Adaptive Server sends SQL commands to a Specialty Data Store when a client requests data access to a table that has been mapped to a remote location. The Adaptive Server issues an **sp_capabilities** RPC to the Specialty Data Store the first time a connection is made. The syntax of the SQL statements the Adaptive Server generates and transmits to the Specialty Data Store is determined by its response to the **sp_capabilities** *sql_syntax* parameter.

In this chapter, SQL commands, clauses, and other syntactical elements are presented in alphabetical order. These commands are a subset of Transact-SQL and DB2 SQL that the Adaptive Server uses when interacting with a Specialty Data Store.

➤ ***Note***

This chapter defines the minimum level of SQL that will be sent. The results of **sp_capabilities** can cause more complex SQL such as expressions and subqueries to be generated.

The following table lists commands discussed in this chapter:

Table 3-1: SQL commands

Command	Description
alter table	Add new columns to an existing table.
begin transaction	Begin a new transaction.
commit transaction	Commit work performed for this transaction.
create index	Create a new index on a table.
create table	Create new tables.
delete (cursor)	Remove rows from a table.
delete (dynamic)	Remove rows from a table.
drop index	Remove an index from a table.
drop table	Remove a table.
insert	Add new rows to a table or view.
insert bulk	Begin or resume **bulk insert** operation.
prepare transaction	Prepare to commit a transaction.

Table 3-1: SQL commands (continued)

Command	Description
readtext	Transmit *text* and *image* data to client.
rollback transaction	Roll back or abort the current transaction.
select	Retrieve rows from database objects.
truncate table	Truncate the table by removing all rows. This statement is not logged, and is not part of any transaction.
update (cursor)	Positional update: change data in a row made current by a cursor.
update (dynamic)	Dynamic update: change data in an existing row(s).
writetext	Write *text* data to Open Server.
writetext bulk	Begin transmission of *text* and *image* data to Open Server.

alter table

Function

Adds new columns to an existing table.

Transact-SQL Syntax

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

DB2 SQL Syntax

```
ALTER TABLE [owner.]table_name
   ADD column_name datatype
       {[, next_column]}...
```

Keywords and Options

table_name – is the name of the table you want to change.

column_name – is the name of a column you want to add to the table.

datatype – is any of the system datatypes except *bit*. If the syntax is DB2, then datatypes are expressed as DB2 datatypes.

null – specifies that a null value should be assigned as the default if the user does not supply a value.

next_column – indicates that you can include additional column definitions (separated by commas) using the same syntax described for a column definition.

Examples

1.
```
alter table publishers
add manager_name varchar(40) null
```

   Add the *manager_name* column to the *publishers* table. For each existing row in the table, assign a null value to the new column.

Comments

- The **alter table** command is sent from Adaptive Server to a Specialty Data Store as a language event.

- Adaptive Server sends the name of the remote table that was provided when the proxy table was defined via the **sp_addobjectdef** system procedure.

begin transaction

Function

Marks the starting point of a transaction.

Transact-SQL Syntax

begin transaction [*transaction_name*]

DB2 SQL Syntax

(No equivalent; **BEGIN TRANSACTION** is used)

Keywords and Options

transaction_name – is the name assigned to the transaction. It must conform to the rules for identifiers.

Examples

```
1. begin transaction
```

Comments

- The **begin transaction** command is sent from Adaptive Server to a Specialty Data Store as a language event.
- In Adaptive Server 11.5, *transaction_name* is not used; it is reserved for use in a later release.
- There is no DB2 equivalent of this command. If the Specialty Data Store requires DB2 syntax, the command is sent as **BEGIN TRANSACTION**.

commit transaction

Function

Commits the work performed by the current transaction.

Transact-SQL Syntax

```
commit transaction [transaction_name]
```

DB2 SQL Syntax

```
COMMIT WORK
```

Keywords and Options

transaction_name – is the name assigned to the transaction. It must conform to the rules for identifiers.

Examples

```
1. commit transaction
```

Comments

- The **commit transaction** command is sent from Adaptive Server to a Specialty Data Store as a language event.
- If a transaction is not currently active, **commit transaction** and **rollback transaction** statements have no effect.
- In Adaptive Server 11.5, *transaction_name* is not used; it is reserved for use in a later release.

create index

Function

Creates indexes on column(s) in a table.

Transact-SQL Syntax

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

DB2 SQL Syntax

```
CREATE [UNIQUE] INDEX index_name
   ON [owner.]table_name
   (column_name [, column_name]...)
```

Keywords and Options

unique – prohibits duplicate index values.

clustered – indicates the physical order of rows on this table is the same as the indexed order of the rows. Only one clustered index is permitted per table.

nonclustered – indicates that there is a level of indirection between the index structure and the data itself. You can have up to 249 nonclustered indexes per table.

index_name – is the name of the index. Index names must be unique within a table, but need not be unique within a database.

table_name – is the name of the table in which the indexed column or columns are located.

column_name – is the column or columns to be included in the index. Composite indexes are based on the combined values of up to 16 columns. The sum of the maximum lengths of all the columns used in a composite index cannot exceed 256 bytes.

fillfactor – specifies how full the DBMS makes each page when it is creating a new index on existing data. The **fillfactor** percentage is relevant only at the time the index is created. As the data changes,

the pages are not maintained at any particular level of fullness. The default is 0. If the **fillfactor** is set to 100, the DBMS creates indexes whose pages are 100% full.

ignore_dup_key – responds to a duplicate key entry into a table that has a unique index. An attempted **insert** of a duplicate key is ignored, and the **insert** is cancelled with an informational message.

sorted_data – speeds creation of an index when the data in the table is in sorted order. If **sorted_data** is specified but data is not in sorted order, an error message displays and the command is aborted.

on *segment_name* – specifies that the index is to be created on the named segment.

Examples

```
1. create index au_id_ind
   on authors (au_id)

2. create unique clustered index au_id_ind
   on authors (au_id)

3. create index ind1
   on titleauthor (au_id, title_id)

4. create nonclustered index zip_ind
   on authors (zip) with fillfactor = 25
```

Comments

- The **create index** command is sent from Adaptive Server to a Specialty Data Store as a language event.
- Columns of *bit*, *text*, and *image* datatypes cannot be indexed.
- A table can have a maximum of 249 nonclustered indexes, whether or not it also has a clustered index.
- You cannot create an index on a view.
- If your Specialty Data Store supports multiple clustered indexes, Adaptive Server will treat the first as a clustered index. Subsequent clustered indexes will be treated as non-clustered indexes.

create table

Function

Creates new tables.

Transact-SQL Syntax

```
create table [database.[owner].]table_name
  (column_name datatype {null | not null}
  [{, next_column }...])
  [on segment_name]
```

DB2 SQL Syntax

```
CREATE TABLE [owner.]table_name
   (column_name datatype [NOT NULL]
   [{, next_column }...])
   [IN { DATABASE database_name |
       database_name.tablespace_name }]
```

Keywords and Options

table_name – is the name of the new table. It conforms to the rules for identifiers and is unique within the database and to the owner.

column_name – is the name of the column in the table. It conforms to the rules for identifiers, and is unique in the table.

datatype – is the datatype of the column. Only system datatypes are used. Certain datatypes expect a length, *n*, in parentheses:

```
datatype(n)
```

null | not null – specifies a null value if a user does not provide a value during an insertion and no default exists (for **null**), or that a user must provide a non-null value if no default exists (for **not null**). Adaptive Server always specifies **null** or **not null**.

next_column – indicates that you can include additional column definitions (separated by commas) using the same syntax described for a column definition.

on *segment_name* – specifies the name of the segment on which to place the table.

IN – specifies a DB2 database name or a database and tablespace name combination.

Examples

1.
```
create table titles
(title_idtid 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 bitnot null)
```

   Creates the *titles* table.

Comments

- The **create table** command is sent from Adaptive Server to a Specialty Data Store as a language event.
- Adaptive Server sends the name of the remote table that was provided when the proxy table was defined via the **sp_addobjectdef** system procedure.
- If *segment_name* is not provided in the command, Adaptive Server looks for a semicolon (;) in the storage location name in the table definition created by **sp_addobjectdef** (see "Defining the Storage Location of Individual Objects" on page 2-3). Everything that follows the semicolon is passed to the Specialty Data Store as *segment_name*.

delete (cursor)

Function

Removes a row from a table. The row affected must have been made current by a cursor.

Transact-SQL Syntax

delete [[*database*.]*owner*.]{*table_name* | *view_name*}

DB2 SQL Syntax

DELETE FROM [*owner*.]{*table_name* | *view_name*}

Keywords and Options

table_name or *view_name* – is the name of the table or view the rows will be deleted from.

Comments

- Adaptive Server issues a cursor **delete** request if it must examine any column's data in order to fulfill the client request. This is true if:
  - There is more than one table involved in the **delete** statement
  - The **delete** statement contains a **where** clause with built-in functions
- The **cursor delete** command is passed from Adaptive Server to a Specialty Data Store as a series of cursor commands:
  - **declare**
  - **open**
  - **fetch**
  - **delete**
  - **close**
  - **deallocate**
- No **where** clause is constructed; Adaptive Server assumes that the Specialty Data Store appends the equivalent of:

  where current of cursor *cursor_name*
- The cursor can be used multiple times before it is deallocated.

- When the Specialty Data Store calls **srv_senddone** to mark the completion of the **delete** command, it must indicate the number of rows affected by the **delete** command. Normally this row count is 1.
- Adaptive Server sends the name of the remote table that was provided when the proxy table was defined via the **sp_addobjectdef** system procedure.

delete (dynamic)

Function

Removes row(s) from a table.

Transact-SQL Syntax

```
delete [[database.]owner.]{table_name | view_name}
   [where column_name relop expression
       [ and column_name relop expression...] ]
```

DB2 SQL Syntax

```
DELETE FROM [owner.]{table_name | view_name}
   [WHERE column_name relop expression
       [ AND column_name relop expression ...] ]
```

Keywords and Options

table_name or *view_name* – is the name of the table or view the rows will be deleted from.

column_name – is the name of the column used in the comparison.

relop – is the keyword **like**, **is null**, or **is not null**, or the comparison operator =, <>, >, <,<=, or >=.

expression – depending on how the *expression handling* capability is set, *expression* can consist of column names, constants, operators, parenthesis, and subqueries.

Comments

- Adaptive Server issues a dynamic **delete** request if it does not have to examine any of the column's data in order to fulfill the client request. This is true if:
  - There is only one table involved in the **delete** statement
  - The **delete** statement contains no built-in functions in its **where** clause
- The dynamic **delete** command is passed from Adaptive Server to a Specialty Data Store as a series of dynamic requests:
  - **prepare** (define parameter formats)
  - **execute** (with parameter data)

  - **deallocate**

- The **where** clause is optional; it is provided by Adaptive Server if the original **delete** command contained a **where** clause.
- The prepared statement can be executed multiple times before it is deallocated.
- When the Specialty Data Store calls **srv_senddone** to mark the completion of the **delete** command, the number of rows affected must be indicated.
- Adaptive Server sends the name of the remote table that was provided when the proxy table was defined via the **sp_addobjectdef** system procedure.

drop index

Function

Removes an index from a table in the current database

Transact-SQL Syntax

drop index *table_name.index_name*

DB2 SQL Syntax

DROP INDEX *index_name*

Keywords and Options

table_name – is the table in which the indexed column is located. The table must be in the current database.

index_name – is the name of the index to be dropped.

Examples

```
1. drop index authors.au_id_ind
```

Comments

- The **drop index** command is sent from Adaptive Server to a Specialty Data Store as a language event.
- To view indexes that exist on a table, use:

  sp_helpindex *table_name*

drop table

Function

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

Transact-SQL Syntax

drop table [[*database*.]*owner*.]*table_name*

DB2 SQL Syntax

DROP TABLE [*owner*.]*table_name*

Keywords and Options

table_name – is the name of the table to be dropped.

Examples

```
1. drop table authors
```

Comments

- The **drop table** command is passed from Adaptive Server to a Specialty Data Store as a language event.
- Adaptive Server sends the name of the remote table that was provided when the proxy table was defined via the **sp_addobjectdef** system procedure.

insert (dynamic)

Function

Adds a new row to a table or view.

Transact-SQL Syntax

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

DB2 SQL Syntax

```
INSERT INTO [owner.]{table_name|view_name}
   [(column_list)]
   VALUES (? [, ?]...)
```

Keywords and Options

table_name, *view_name* – is the table or view in which the new rows will be inserted.

column_list – is a list of one or more columns to which data is to be added. The columns can be in any order, but the incoming data (whether in a **values** clause or a **select** clause) are in the same order.

values – is a keyword that introduces a list of expressions.

? – specifies parameters that Adaptive Server will pass at the time the **insert** command should be executed.

Examples

```
2. insert titles
   (title_id, title, type, pub_id, notes, pubdate,
       contract)
   values (?, ?, ?, ?, ?, ?, ?)
```

Comments

- The **insert** command is passed from Adaptive Server to a Specialty Data Store as a series of dynamic SQL commands:
  - **prepare**
  - **execute**
  - **close**
  - **deallocate**

- The **values** list is passed as dynamic SQL parameters.
- Adaptive Server sends the name of the remote table that was provided when the proxy table was defined via the **sp_addobjectdef** system procedure.

insert bulk

Function

Begins or resumes bulk insert operation.

Transact-SQL Syntax

```
insert bulk [owner.]{table_name|view_name}
   [with nodescribe]
```

DB2 SQL Syntax

```
INSERT BULK [owner.]{table_name|view_name}
   [WITH NODESCRIBE]
```

Keywords and Options

bulk – indicates that bulk copy rows are sent to the Specialty Data Store by means of the SRV_BULK event. The Specialty Data Store must initialize a CS_BLKDESC data structure and return it to Adaptive Server via the **blk_srvinit()** function.

table_name, view_name – is the table or view the bulk copy rows will be sent to.

with nodescribe – indicates that the previous batch of bulk insert operations should be committed to the database.

Comments

- The **insert bulk** command is passed from Adaptive Server to a Specialty Data Store as a language event.
- If the command includes the **with nodescribe** option, the CS_BLKDESC has been allocated and described and need not be described to the client again. This command is sent to the Specialty Data Store as a result of a **blk_done** (BLK_COMMIT) request from the client (Adaptive Server), indicating that the previous batch of rows should be committed to the database.
- **INSERT BULK** is not proper DB2 syntax; however, if the Specialty Data Store indicates that it supports bulk insert capabilities, then it must support this syntax regardless of the value of the *sql syntax* capability.

prepare transaction

Function

Used to see if a server is prepared to commit a transaction.

Transact-SQL Syntax

```
prepare transaction
```

DB2 SQL Syntax

(No equivalent; **`PREPARE TRANSACTION`** is used)

Comments

- The **prepare transaction** command is passed from Adaptive Server to a Specialty Data Store as a language event.
- There is no DB2 equivalent of this command. If the Specialty Data Store requires DB2 syntax, the command is sent using the Transact-SQL form.
- Adaptive Server expects the Specialty Data Store to acknowledge the receipt of the **prepare transaction** command by returning a "done" status value in **srv_senddone()**.
- If the Specialty Data Store returns an error status, Adaptive Server attempts to issue a **rollback** command to each server involved in the transaction.

readtext

Function

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

Transact-SQL Syntax

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

DB2 SQL Syntax

(No equivalent; Transact-SQL form is sent)

Keywords and Options

table_name.column_name – the name of the *text* or *image* column must include the table name. The database name and owner name are optional.

text_pointer – a *varbinary(16)* value that stores the pointer to the *text* or *image* data.

offset – specifies the number of bytes or characters to skip before starting to read *text* or *image* data.

size – specifies the number of bytes or characters of data to read.

using – specifies whether **readtext** interprets the *offset* and *size* parameters as a number of bytes (**bytes**) or as a number of characters (**chars** and **characters** are synonymous).

Comments

- There is no DB2 equivalent of this command. If the Specialty Data Store requires DB2 syntax, the command is sent using the Transact-SQL form.
- Adaptive Server sends this command only if the Specialty Data Store indicates that it supports *text* and *image* data.

rollback transaction

Function

Rolls a user-defined transaction back to the beginning of the transaction.

Transact-SQL Syntax

```
rollback transaction [transaction_name]
```

DB2 SQL Syntax

```
ROLLBACK WORK
```

Keywords and Options

transaction_name– is the name assigned to the transaction. It must conform to the rules for identifiers.

Examples

```
1. rollback transaction
```

Comments

- The **rollback transaction** command is passed from Adaptive Server to a Specialty Data Store as a language event.
- In Adaptive Server 11.5, *transaction_name* is not used; it is reserved for use in a later release.

select

Function

Retrieves rows from database objects.

Transact-SQL Syntax

```
select select_list
   [from [[database.]owner.]{table_name |view_name}
   [,[[database.]owner.]{table_name|view_name}]... ]
   [where column_name relop expression
       [and column_name relop expression ...] ]
   [for update of column_name_list]
```

DB2 SQL Syntax

```
SELECT select_list
   FROM [owner.]{table_name |view_name}
       [,[owner.]{table_name |view_name}]... ]
   [WHERE column_name relop expression
       [AND column_name relop expression ...] ]
   [FOR UPDATE OF column_name_list]
```

Keywords and Options

select_list – is one or more of the following items:

- A list of column names in the order in which they should be returned.
- The constant value of "1". For example, Adaptive Server sends a **select 1...** statement to process a **count** if the Specialty Data Store does not support the **count** aggregate.
- Optional capabilities the Specialty Data Store supports such as aggregates and expressions.

table_name, *view_name* – lists tables and views used in the **select** statement. If there is more than one table or view in the list, their names are separated by commas.

Table names and view names can be given correlation names. This is done by providing the table or view name, then a space, then the correlation name, like this:

```
select pub_name, au_lname, au_fname
from publishers t1, authors t2
```

column_name– is the name of the column used in the comparison.

relop– is the keyword **like**, **is null**, or **is not null**, or the comparison operator =, <>, >, <,<=, or >=.

expression – depending on how the *expression handling* capability is set, *expression* can consist of column names, constants, operators, parenthesis, and subqueries.

Examples

```
1. select pub_id, pub_name, city, state from
   publishers for read only
2. select pub_name, pub_id
   from publishers
3. select 1 from publishers
   where city = @p1
4. select type, price
   from titles where price > @p1 for update of price
5. select au_id, TEXTPTR(copy)
   from blurbs where au_id = @p1
```

Comments

- Depending on the capabilities of the Specialty Data Store, the **select** statement may also contain **group by**, **order by**, or **union**.
- The **select** command is passed from Adaptive Server to a Specialty Data Store as a series of cursor commands:
  - **declare**
  - **open**
  - **fetch**
  - **close**
  - **deallocate**
- Each cursor can be reused multiple times after having been passed a new set of parameters prior to being opened.
- If the or predicate capability (105) is supported, **or** predicates can be used in the **where** clause search conditions.

- Data values in the **where** clause search conditions are passed as cursor parameters using the datatype associated with the column.
- Cursor parameters are indicated with a "@" when Transact-SQL syntax is used, and with a "?" when DB2 syntax is used.
- Cursors can be opened with the following options:
  - **read only** – indicates that the cursor is a read only cursor and no updates are applied to rows made current by it
  - **update** – indicates that the cursor is an updateable cursor, and rows made current by it can be deleted or updated
- Adaptive Server sends the name of the remote table that was provided when the proxy table was defined via the **sp_addobjectdef** system procedure.

truncate table

Function

Removes all rows from a table.

Transact-SQL Syntax

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

DB2 SQL Syntax

DELETE FROM [*owner*.]*table_name*

Keywords and Options

table_name – is the name of the table to be truncated.

Examples

```
1. truncate table authors
```

Comments

- The **truncate table** command is passed from Adaptive Server to a Specialty Data Store as a language event.
- Adaptive Server sends the name of the remote table that was provided when the proxy table was defined via the **sp_addobjectdef** system procedure.
- The **truncate table** operation is not logged; thus, a trigger cannot be fired as a result of a truncate operation.
- If DB2 syntax is used, the Specialty Data Store must make sure that the statement's work is committed since Adaptive Server does not follow this statement with a **commit work** command.

update (cursor)

Function

Changes data in a row made current by a cursor, either by adding data or by modifying existing data.

Transact-SQL Syntax

```
update [[database.]owner.]{table_name | view_name}
   set column_name1 = @p1
       [, column_name2 = @p2]...
```

DB2 SQL Syntax

```
UPDATE [owner.]{table_name | view_name}
   SET column_name1 = ?
       [, column_name2= ?]...
```

Keywords and Options

table_name, *view_name* – is the table or view where the data will be updated.

set – specifies the column name and assigns the new value. The value is passed as a cursor parameter.

Examples

1.
```
update authors
set au_lname = @p1
```

The row made current by the cursor *authors_cursor* is modified; the column *au_lname* is set to the value of the parameter *@p1*.

Comments

- Adaptive Server issues a cursor **update** request if it must examine any column's data to fulfill the client request. This is true if:
  - There is more than one table involved in the **update** statement
  - The **update** statement contains built-in functions in its **where** clause
  - A column name is referenced in the **set** expression
- The **update** command is passed from Adaptive Server to a Specialty Data Store as a series of cursor commands:
  - **declare**

  - **open**
  - **fetch**
  - **update**
  - **close**
  - **deallocate**

- A **where** clause is not constructed. It is assumed that the Specialty Data Store appends the equivalent of:

  `where current of cursor` ***`cursor_name`***

- The cursor can be re-used multiple times before it is deallocated.
- When the Specialty Data Store calls **srv_senddone** to mark the completion of the **update** command, the number of rows affected must be indicated.
- Adaptive Server sends the name of the remote table that was provided when the proxy table was defined via the **sp_addobjectdef** system procedure.

update (dynamic)

Function

Changes data in existing rows of the referenced table.

Transact-SQL Syntax

```
update [[database.]owner.]{table_name | view_name}
   set column_name1 = ?
       [, column_name2 = ?]...
   [ where column_name relop expression
       [ AND column_name relop expression ...] ]
```

DB2 SQL Syntax

```
UPDATE [owner.]{table_name | view_name}
   SET column_name1 = ?
       [, column_name2 = ?]...
   [ where column_name relop expression
       [ AND column_name relop expression ...] ]
```

Keywords and Options

table_name, view_name – is the table or view where the data will be updated.

set – specifies the column name and assigns the new value. The value is passed as a parameter.

column_name – is the name of the column used in the comparison.

relop – is the keyword **like**, **is null**, or **is not null**, or the comparison operator =, <>, >, <,<=, or >=.

expression – depending on how the *expression handling* capability is set, *expression* can consist of column names, constants, operators, parenthesis, and subqueries.

Examples

1.
```
update authors
set au_lname = ?
where au_id = ?
```

The *au_lname* column is set to the value of *parameter 1* where the value of au_id is equal to the value of *parameter 2*.

Comments

- Adaptive Server issues a dynamic **update** request if it does not have to examine any column's data in order to fulfill the client request. This is true if:
  - There is only one table involved in the **update** statement
  - The **update** statement does not contain built-in functions in its **where** clause
  - A column name is not referenced in the **set** expression
- The **update** command is passed from Adaptive Server to a Specialty Data Store as a series of dynamic requests:
  - **prepare** (define parameter formats)
  - **execute** (with parameter data)
  - **deallocate**
- When the Specialty Data Store calls **srv_senddone** to mark the completion of the **update** command, the number of rows affected must be indicated.
- Adaptive Server sends the name of the remote table that was provided when the proxy table was defined via the **sp_addobjectdef** system procedure.

writetext bulk

Function

Permits insert and updating of a *text* or *image* column.

Transact-SQL Syntax

```
writetext bulk
   [[database.]owner.]table_name.column_name
   text_pointer [timestamp=ts_value] [with log]
```

DB2 SQL Syntax

(No equivalent; Transact-SQL form is sent)

Keywords and Options

bulk - indicates that a bulk copy operation is to follow as a SRV_BULK event.

table_name.column_name – the name of the *text* or *image* column must include the table name. The database name and owner name are optional.

text_pointer – a *varbinary(16)* value that stores the pointer to the *text* or *image* data.

timestamp – supplies a hexadecimal value that can be used as the timestamp for the given *text* or *image* data. This value comes from the client that originated the writing of *text* or *image* data. The client obtains the value from an initial query on the column. This implies that the Specialty Data Store can supply a timestamp value during the initial query, and use it during a **writetext bulk** to ensure a matching value.

with log – a flag that indicates if the requested *text* or *image* operation should be logged. This flag originates for the initial client request. Typically, Specialty Data Stores do not log *text* or *image* operations, even if this flag is supplied.

Comments

- When sending *text* or *image* data, Adaptive Server uses the Client-Library function **ct_send_data()**. This function causes both language and bulk events.
- The **writetext bulk** command is received within the language event.

- Receipt of the **writetext bulk** command indicates that the Specialty Data Store is about to receive a stream of *text* or *image* bytes. These bytes are received within a bulk event.
- For more information on the **writetext bulk** command, refer to Chapter 4, "Text and Image Handling," and the *Open Client and Open Server Common Libraries Reference Manual*.

4 Text and Image Handling

Handling of *text* and *image* datatypes within the Adaptive Server is completely different and more complex than the handling of other datatypes. *text* and *image* data is unique because the values can be large. Due to the size, applications frequently request portions of a *text* or *image* column.

Special processing to more efficiently handle *text* and *image* datatypes is required for handling each of the following actions:

- Writing
- Reading
- Deleting
- Updating
- Pattern matching with the **like** keyword
- Processing the **char_length** function
- Processing the **datalength** function
- Processing the **textvalid** function

This chapter discusses the interaction between the Adaptive Server and Specialty Data Stores in dealing with these functions. Other issues discussed in this chapter include:

- Using text pointers with *text* and *image* data
- Using text timestamps with *text* and *image* data
- Transferring *text* and *image* data between Adaptive Server and a Specialty Data Store

Supporting Text Pointers

Text pointers allow applications to manipulate a large value without sending the entire object through the network. A text pointer is a 16-byte handle to a *text* or *image* column in a row. Instead of returning the entire object in a query, the text pointer value is returned instead. An application can then use the text pointer value in a **readtext** or **writetext** command to fetch or update the actual column value.

Specialty Data Stores have the option of supporting text pointers. If text pointers are supported, overall performance is improved since the Adaptive Server can rely on the Specialty Data Store to perform

many text and image functions. Additionally, the entire *text* or *image* value never has to cross the network unless the client requests the complete value.

For best performance, Specialty Data Stores should produce and return text pointers for a given column in a given row. Conversely, when a Specialty Data Store is given a text pointer, it must be able to access the data the text pointer references.

If the Specialty Data Store does not support text pointers, the Adaptive Server will always request the complete value of a *text* or *image* column to be returned in a result set. The Adaptive Server stores these values and manufactures its own text pointer to give to a client. This generates a great deal of overhead in the Adaptive Server and the network.

Specialty Data Store Support of Text Pointers

Adaptive Server requests a *text* and *image* column's text pointer via a **select** request using the **textptr()** function. The Specialty Data Store returns a text pointer value that is a 16-byte *binary* value. The only requirement is that the Specialty Data Store must be able to use that text pointer value to access the column if it is returned to the Specialty Data Store during a **readtext** or **writetext** command.

For example:

```
select au_fname, au_lname, textptr(copy)
    from blurbs
    where au_id = "409-56-7008"
```

When a Specialty Data Store receives this request, it must return the following for the rows with *au_id* "409-56-7008" in the *blurbs* table:

- *au_fname* and *au_lname* columns.
- A text pointer value that points to the data in the *text* column *copy*. The text pointer value can later be used to fetch the actual data.

For more information on selecting *text* and *image* data, see "Selecting Data When Text Pointers Are Supported" on page 4-5.

Supporting Text Timestamps

Text timestamps mark the time of a *text* or *image* column's last modification. Adaptive Server does not specifically request a text timestamp, but acquires the timestamp via Client-Library when the

data is fetched. The timestamp is supplied in the CS_IODESC of the fetched value.

Adaptive Server performs no special processing on text timestamps except to:

- Pass the supplied timestamp value from a client to a Specialty Data Store when performing **writetext** or Open Client functions
- Pass the supplied timestamp value from a Specialty Data Store to a client when performing a **select** or **readtext**

Specialty Data Stores have the option of supporting text timestamps.

Specifying Text and Image Capabilities

Specialty Data Stores indicate which text and image capabilities they support through the **sp_capabilities** RPC (see "sp_capabilities" on page 5-3). These capabilities are discussed in the following sections.

Text and Image Handling

The *text/image handling* capability indicates if the Specialty Data Store:

- Does not support *text* and *image* datatypes
- Supports *text* and *image* datatypes but not text pointers
- Supports *text* and *image* datatypes and text pointers

If *text* and *image* datatypes are not supported, Adaptive Server does not allow mapping of *text* and *image* data columns to a table on that Specialty Data Store.

If *text* and *image* datatypes are supported, but text pointers are not, Adaptive Server processes text and image operations by first storing the *text* and *image* data internally. Also, Adaptive Server imposes the following limitations on Adaptive Server client operations:

- **writetext** is not supported
- **readtext** is not supported
- **textptr()** is not supported
- **insert** and **update** via Open Client text and image functions is not supported
- Data length is limited to 450 bytes for **insert** and **update**

Text Pattern Handling

The *text pattern handling* capability indicates if a Specialty Data Store can perform pattern matching on *text* data. If a Specialty Data Store does not support this feature, Adaptive Server retrieves the data and stores it internally during pattern matching operations.

Inserting *text* and *image* Data

When Adaptive Server inserts a row containing one or more *text* or *image* columns, the interactions between Adaptive Server and the Specialty Data Store depend on the action initiating the insert.

Data Inserted Based on the *insert* Command

When Adaptive Server inserts data because of an **insert** command, Adaptive Server sends the Specialty Data Store an **insert** command. Adaptive Server processes the insert via a parameterized dynamic request, with each input value being passed as a parameter. The length of the data using this insert method is limited to approximately 450 bytes.

Data Inserted Based on the *writetext* Command

When Adaptive Server inserts data because of a **writetext** command, Adaptive Server processes the insert using Client-Library's **ct_send_data()** function. The length of the data is not limited.

➤ ***Note***

Adaptive Server performs inserts based on **writetext** only if the Specialty Data Store supports text pointers.

The interactions that occur are as follows:

1. The data for the *text* or *image* column is issued by Adaptive Server using the following Client-Library calls:
   - **ct_command**
   - **ct_data_info**
   - **ct_send_data** (called multiple times in a loop)
   - **ct_send**

- **ct_results**

2. The Specialty Data Store must handle the data as described in the *Open Client and Open Server Common Libraries Reference Manual*. The handling includes the following:
   - Language event: receipt of **writetext bulk** command
   - Bulk event: obtain data via **srv_text_info** and **srv_get_text** (called multiple times in a loop)

Selecting *text* and *image* Data

Adaptive Server selects a *text* or *image* column based on a client's **select** or **readtext** command. If the Specialty Data Store does not support text pointers, Adaptive Server also selects *text* or *image* data to perform the following:

- Pattern matching
- **char_length** functions
- **datalength** functions

Data is selected differently depending on whether the Specialty Data Store supports text pointers.

Selecting Data When Text Pointers Are Supported

When Adaptive Server processes a select operation when text pointers are supported, the following interactions occur:

1. Adaptive Server requests a *text* or *image* column's text pointer via a **select** request using the **textptr()** function. Adaptive Server performs this automatically when the client issues a **select** command. The client must request the text pointer if a **readtext** command is going to be issued. The following is an example of such a request:

   ```
   select au_id, textptr(copy) from blurbs
       where au_id = "409-56-7008"
   ```

2. The Specialty Data Store must return valid text pointer values for the requested row(s). The value is a *binary(16)* datatype.
3. Adaptive Server issues a **readtext** command to the Specialty Data Store. The command has the following syntax (see "readtext" on page 3-21 for more information):

```
readtext
[[database.]owner.]table_name.column_name
text_pointer offset size
[using chars]
```

If **size** is equal to zero, the entire *text* or *image* column is being selected.

4. The Specialty Data Store must return the requested *text* or *image* data via the following Open Server calls:

   srv_text_info
   srv_send_text (called multiple times in a loop)

5. Adaptive Server retrieves the data via the following Client-Library calls:

   ct_fetch
   ct_data_info
   ct_get_data (called multiple times in a loop)

Selecting Data When Text Pointers Are Not Supported

When a Specialty Data Store does not support text pointers, Adaptive Server selects the entire *text* or *image* data before processing the following:

- **select** commands
- Pattern matching
- **char_length** functions
- **datalength** functions

When Adaptive Server processes the select operation, the following interactions occur:

1. Adaptive Server requests a *text* or *image* column's data with a **select** request. The following is an example of such a request:

   ```
   select au_id, copy from blurbs
       where au_id = "409-56-7008"
   ```

2. The Specialty Data Store must return the requested row(s) supplying the entire *text* or *image* data.

3. Adaptive Server retrieves the *text* or *image* data via the following Client-Library calls and stores the data internally:

   ct_fetch
   ct_data_info
   ct_get_data (called multiple times in a loop)

Updating *text* and *image* Data

When Adaptive Server is updating a row with one or more *text* or *image* columns, the interactions that occur depend on the action initiating the update.

Data Updated Based on the *update* Command

When Adaptive Server is updating data because of an **update** command, Adaptive Server issues the Specialty Data Store an **update** command. Adaptive Server processes the update using a parameterized dynamic request, with each new value being passed as a parameter. The length of the data is limited to approximately 450 bytes.

Data Updated Based on the *writetext* Command

When Adaptive Server is updating data because of a **writetext** command, Adaptive Server does not perform the update via a dynamic request, but uses Client-Library's **ct_send_data()** function. The length of the data is not limited.

➤ ***Note***

Adaptive Server performs updates based on **writetext** only if the Specialty Data Store supports text pointers.

Pattern Matching on *text* Data

Adaptive Server processes pattern matching on *text* data when processing the **pattern** function and the **like** predicate. Adaptive Server processes pattern matching differently depending on whether the Specialty Data Store supports pattern matching.

Pattern Matching When Pattern Matching Is Supported

When the Specialty Data Store supports pattern matching, Adaptive Server issues an **sp_patindex** RPC within a cursor event to pass the

pattern matching request to a Specialty Data Store. **sp_patindex** takes the following parameters:

Table 4-1: sp_patindex parameters

Parameter Name	Datatype	Description
database	*varchar(30)*	Input
owner	*varchar(30)*	Input
table	*varchar(30)*	Input
column	*varchar(30)*	Input
txtptr	*binary(16)*	Input; text pointer of row / column being processed
ret_value	*integer*	Output; contains the starting position of the first occurrence of *pattern*. "0" should be returned if the pattern is not found.
pattern	*varchar(255)*	Input; pattern being searched for. The pattern can contain Sybase pattern matching characters ("%_").
use_bytes	*integer*	Input; value will be greater than zero if the pattern matching should be done on a byte basis. Otherwise, matching should be done on a character basis.

For more information on **sp_patindex**, see "sp_patindex" on page 5-11.

When Pattern Matching Is Not Supported

When the Specialty Data Store does not support pattern matching, Adaptive Server requests the data from the Specialty Data Store, stores the data as *text* data, and processes the pattern match internally.

Processing the *char_length* Function

Adaptive Server processes the **char_length** function differently depending on whether the Specialty Data Store supports text pointers.

Processing *char_length* When Text Pointers Are Supported

When the Specialty Data Store supports text pointers, Adaptive Server relies on the Specialty Data Store to perform the **char_length** function on *text* data. This reduces the amount of data transferred between Adaptive Server and the Specialty Data Store.

Adaptive Server issues an **sp_char_length** RPC within a cursor event to pass the **char_length** function to a Specialty Data Store. **sp_char_length** takes the following parameters:

Table 4-2: sp_char_length parameters

Parameter Name	Datatype	Description
database	*varchar(30)*	Input
owner	*varchar(30)*	Input
table	*varchar(30)*	Input
column	*varchar(30)*	Input
txtptr	*binary(16)*	Output; text pointer of row / column being processed
ret_value	*integer*	Output; contains the character length

For more information on **sp_char_length**, see “sp_char_length” on page 5-6.

Processing *char_length* When Text Pointers Are Not Supported

When the Specialty Data Store does not support text pointers, Adaptive Server requests the data from the Specialty Data Store, stores the data as *text* data, and processes the **char_length** function internally.

Processing the *datalength* Function

Adaptive Server processes the **datalength** function differently depending on whether the Specialty Data Store supports text pointers.

Processing *datalength* When Text Pointers Are Supported

When the Specialty Data Store supports text pointers, Adaptive Server relies on the Specialty Data Store to perform the **datalength** function on *text* and *image* data. This reduces the amount of data transferred between Adaptive Server and the Specialty Data Store.

Adaptive Server issues an **sp_datalength** RPC within a cursor event to pass the **datalength** function to a Specialty Data Store. **sp_datalength** takes the following parameters:

Table 4-3: sp_datalength parameters

Parameter Name	Datatype	Description
database	*varchar(30)*	Input
owner	*varchar(30)*	Input
table	*varchar(30)*	Input
column	*varchar(30)*	Input
txtptr	*binary(16)*	Input; text pointer of row / column being processed
ret_value	*integer*	Output; contains the data length

For more information on **sp_datalength**, see “sp_datalength” on page 5-10.

Processing *datalength* When Text Pointers Are Not Supported

When the Specialty Data Store does not support text pointers, Adaptive Server requests the data from the Specialty Data Store, stores the data as *text* data, and processes the **datalength** function internally.

Processing the *textvalid* Function

Adaptive Server always relies on the Specialty Data Store to process the **textvalid** function. If the Specialty Data Store supports text pointers, Adaptive Server passes these requests to the Specialty Data Store; otherwise, Adaptive Server indicates the text pointer is valid.

When Adaptive Server can pass a **textvalid** function to a Specialty Data Store, it passes the function by issuing an **sp_textvalid** RPC within a cursor event. **sp_textvalid** takes the following parameters:

Table 4-4: sp_textvalid parameters

Parameter Name	Datatype	Description
database	*varchar(30)*	Input
owner	*varchar(30)*	Input
table	*varchar(30)*	Input
column	*varchar(30)*	Input
txtptr	*binary(16)*	Input; text pointer of row / column being processed
ret_value	*integer*	Output; set to "1" if the text pointer is valid; otherwise, set to "0"

For more information on **sp_textvalid**, see "sp_textvalid" on page 5-17.

5 System and Catalog RPCs

Overview

This chapter consists of reference pages, presented in alphabetical order, which discuss specific RPCs issued by Adaptive Server to Specialty Data Stores.

The following table lists the RPCs discussed in this chapter.

Table 5-1: Command RPCs

RPC	Description
sp_capabilities	Show the capabilities of the Specialty Data Store
sp_char_length	Determine the number of characters in text referenced by **textptr**
sp_columns	Show names and attributes of all columns associated with the table or view
sp_datalength	Determine the length of text data referenced by **textptr**, in bytes
sp_patindex	Determine the character position in text field of the beginning of the supplied pattern
sp_statistics	Show index attributes for a given table
sp_tables	Show attributes of a given table, or all tables
sp_textvalid	Determine whether or not a supplied textptr is still valid
sp_thread_props	Show / modify attributes of Adaptive Server's connection to the Specialty Data Store

Introduction to Catalog Procedures

This section lists the catalog RPCs alphabetically. These RPCs also exist in the Adaptive Server, as a uniform catalog interface for accessing database gateways. Catalog RPCs return data dictionary information in table form.

The Specialty Data Store must implement four of these RPCs: **sp_capabilities**, **sp_columns**, **sp_statistics**, and **sp_tables**. The others may need to be implemented to support capabilities specified thru **sp_capabilities**.

Syntax and Optional Parameters

The syntax is provided in expanded form because many of these procedures have more optional parameters, and in many cases it is more convenient to supply the parameters in the form:

```
@parametername = value
```

than to supply all of the parameters. The parameter names in the syntax statements match the parameter names defined by the procedures.

For example, here is the syntax for **sp_columns**:

```
sp_columns table_name [, table_owner]
   [, table_qualifier] [, column_name]
```

If you need to use **sp_columns** to find information about a particular column, you can use:

```
sp_columns @table_name = publishers,
    @column_name = "pub_id"
```

This provides the same information as the command with all of the parameters specified:

```
sp_columns publishers, "dbo", "pubs2", "pub_id"
```

You can also use "null" as a placeholder:

```
sp_columns publishers, null, null, "pub_id"
```

➤ ***Note***

Adaptive Server sends RPC parameters by position only, and never uses names to identify parameters.

sp_capabilities

Function

Used to determine the capabilities of the Specialty Data Store.

Syntax

```
sp_capabilities
```

Parameters

None.

Comments

- **sp_capabilities** is mandatory for all implementations of a Specialty Data Store. It is used by Adaptive Server to determine the functional capabilities of the Specialty Data Store.
- If the RPC returns a status of 0, Adaptive Server assumes the RPC completed successfully.
- If the RPC fails to return a status or returns a status other than 0, Adaptive Server assumes the RPC did not complete successfully.
- The only reason an error is generated as a result of executing this RPC is to describe an operating system or an Open Server error.
- The result set must contain sufficient information to allow Adaptive Server to successfully interact with the Specialty Data Store during normal query processing. The following table lists the format of the result set (refer to "Specialty Data Store Capabilities" on page 2-8 for a more detailed explanation of the capabilities):

Table 5-2: sp_capabilities result set

ID	Capability Name	Value
101	sql syntax	1 = Transact-SQL; 2 = DB2
102	join handling	0 = unsupported; 1 = no outer join; 2 = Transact-SQL support level; 3= Oracle support level
103	aggregate handling	0 = unsupported; 1 = ANSI SQL support level; 2 = Transact-SQL support level
104	and predicates	0 = unsupported; 1 = supported
105	or predicates	0 = unsupported; 1 = supported

Table 5-2: sp_capabilities result set

ID	Capability Name	Value
106	like predicates	0 = unsupported; 1 = ANSI-style supported; 2 = Transact-SQL-style supported
107	bulk insert handling	0 = unsupported; 1 = supported
108	text/image handling	0 = unsupported; 1 = text without textptr; 2 = text with textptr
109	transaction handling	0 = unsupported; 1 = local transactions supported
110	text pattern handling	0 = unsupported; 1 = supported
111	order by	0 = unsupported; 1 = supported
112	group by	0 = unsupported; 1 = ANSI SQL compatible; 2 = Transact-SQL compatible
113	net password encryption	0 = unsupported; 1 = supported
114	object name case sensitivity	0 = case insensitive; 1 = case sensitive
115	distinct	0 = unsupported; 1 = supported
117	union	0 = unsupported; 1 = supported
118	string functions	0 = unsupported; 1 = **substr()** function supported; 2 = **substr()**, **lower()**, **ltrim()**, **rtrim()**, and **upper()** functions supported; 3 = Transact-SQL string functions supported
119	expression handling	0 = unsupported; 1 = ANSI SQL expressions; 2 = Transact-SQL expressions
120	truncate blanks	0 = do not truncate trailing blanks; 1 = truncate trailing blanks
121	language handling	0 = transaction control, **readtext**, **writetext** and DDL statements supported; 1 = all queries supported except those containing dates; 2 = all queries supported
122	date functions	0 = unsupported; 1 = Transact-SQL date functions supported
123	math functions	0 = unsupported; 1 = **abs**, **cos**, **exp**, **floor**, **power**, **round**, **sign**, **sin**, **sqrt**, **tan** supported; 2 = Transact-SQL math functions supported
124	convert function	0 = unsupported; 1 = supported
125	T-SQL delete/update	0 = multiple tables not supported; 1 = multiple tables supported

Table 5-2: sp_capabilities result set

ID	Capability Name	Value
126	insert select	0 = unsupported; 1 = ANSI SQL **insert select**; 2 = Transact-SQL **insert select**
127	subquery support	0 = unsupported; 1 = ANSI SQL subquery support; 2 = Transact-SQL subquery support

sp_char_length

Function

Determines the number of characters in the *text* data associated with the **textptr**. The result may be different than that returned by **sp_datalength**, if multi-byte characters are in use.

Syntax

```
sp_char_length [table_qualifier, ][owner, ]table,
   column, txtptr, ret_value
```

Parameters

table_qualifier – varchar(30) – the database portion or first-part of a three-part name. Adaptive Server supports three-part naming in various DBMS products, which means tables can be directly identified and queried using a three-part name. Support for this parameter is mandatory.

owner– varchar(30) – the name of the object owner.

table – varchar(30) – the name of the object containing the *text* column.

column – varchar(30) – the name of the *text* column within *table.*

txtptr – binary(16) – a Specialty Data Store-specific handle for identifying *text* data.

ret_value – int – the output parameter in which character count is placed.

Comments

- This RPC will only be issued if the Specialty Data Store supports *text* and *image* text pointers.
- **sp_char_length** is used for *text* and *image* processing to handle the **char_length**() function. See “Processing the char_length Function” on page 4-8 for more information.
- The character length should be returned in the *ret_value* parameter.
- If the RPC returns a status of 0, Adaptive Server assumes that the RPC completed successfully.
- If the RPC returns a status other than 0, Adaptive Server assumes the RPC did not complete successfully.

sp_columns

Function

Returns column information for a single object that can be queried in the current DBMS environment. The returned columns belong to either a table or a view.

Syntax

```
sp_columns table [, owner] [, table_qualifier]
   [, column_name]
```

Parameters

table – varchar(30) – the name of the object containing the columns.

owner– varchar(30) – the name of the object owner. If the parameter is not specified, **sp_columns** should use the default rules of the underlying DBMS to determine which table's columns to return. Support of this parameter is mandatory.

table_qualifier – varchar(30) – the database portion or first-part of a three-part name. Adaptive Server supports three-part naming in various DBMS products, which means tables can be directly identified and queried using a three-part name. Support for this parameter is mandatory.

column_name – varchar(30) – the name of a column in the table. Support for this parameter is optional.

Comments

- If the specified table does not exist, an empty result set should be returned.
- The following table shows the results set:

Table 5-3: Results set for sp_columns

Column	Datatype	Description
table_qualifier	*varchar(30)*	This column supports three-part naming in various DBMS products, which means tables can be directly identified and queried using a unique three-part name. This column represents the database portion of the three-part name.
table_owner	*varchar(30)*	This column represents the owner portion of the three-part name.
table_name	*varchar(30)*	This column represents the table or view name portion of the three-part name. This field cannot be NULL.
column_name	*varchar(30)*	This field cannot be NULL.
data_type	*smallint*	Integer code for ODBC datatype (see "ODBC Datatypes" on page 5-19). The native datatype name is returned in the *type_name* column. This field cannot be NULL.
type_name	*varchar(30)*	String representing a datatype. The underlying DBMS presents this datatype name.
precision	*int*	Number of significant digits.
length	*int*	Length in bytes of a datatype. This field cannot be NULL.
scale	*smallint*	Number of digits to the right of the decimal point.
radix	*smallint*	Base for numeric types.
nullable	*smallint*	The value "1" means NULL is possible; "0" means NOT NULL.
remarks	*varchar(254)*	
ss_data_type	*smallint*	A number representing the Adaptive Server datatype that this column is being mapped to (see "Adaptive Server Datatypes" on page 5-20). When a **create existing table** statement is executed, this column is compared to the column definition in the **create existing table** statement and must match.

Table 5-3: Results set for sp_columns (continued)

Column	Datatype	Description
colid	*tinyint*	A number uniquely defining the column within the table. It should be ascending based on the order of columns in the **create table** statement.
remote_data_type	*int*	A number representing the underlying DBMS's datatype. The meaning of this value is defined by the Specialty Data Store only. It is only stored by the Adaptive Server.

sp_datalength

Function

Obtains the length of *text* data, in bytes, referenced by the text pointer.

Syntax

```
sp_datalength [table_qualifier, ][owner, ] table,
   column, txtptr, ret_value
```

Parameters

table_qualifier – varchar(30) – the database portion or first-part of a three-part name. Adaptive Server supports three-part naming in various DBMS products, which means tables can be directly identified and queried using a three-part name. Support for this parameter is mandatory.

owner – varchar(30) – the name of the object owner.

table – varchar(30) – the name of the object containing the *text* column.

column – varchar(30) – the name of the *text* column within *table.*

txtptr – binary(16) – a Specialty Data Store-specific handle for identifying *text* data.

ret_value – int – the output parameter in which the character count is placed.

Comments

- This RPC will only be issued if the Specialty Data Store supports *text* and *image* text pointers.
- **sp_datalength** is used for *text* and *image* processing to handle the **datalength()** function. See “Processing the datalength Function” on page 4-9 for more information.
- The data length should be returned in the *ret_value* parameter.
- If the RPC returns a status of 0, Adaptive Server assumes that the RPC completed successfully.
- If the RPC returns a status other than 0, Adaptive Server assumes that the RPC did not complete successfully.

sp_patindex

Function

Finds the character position in a *text* column that contains the supplied pattern.

Syntax

```
sp_patindex [table_qualifier, ][owner, ]table,
   column, pattern, ret_value
```

Parameters

table_qualifier – varchar(30) – the database portion or first-part of a three-part name. Adaptive Server supports three-part naming in various DBMS products, which means tables can be directly identified and queried using a three-part name. Support for this parameter is mandatory.

owner – varchar(30) – the name of the object owner.

table – varchar(30) – the name of the object containing the *text* column.

column – varchar(30) – the name of the *text* column within *table.*

ret_value – int – the output parameter containing the character offset of the start of the pattern.

pattern – varchar(255) – the pattern on which to search. The pattern can contain Sybase pattern matching characters.

use_bytes – int – if supplied as "0", **sp_patindex** should return the offset in characters; otherwise, the offset should be returned in bytes.

Comments

- This RPC will only be issued if the Specialty Data Store supports *text* and *image* text pointers and text pattern handling.
- Used for *text* and *image* processing to handle the **patindex()** function and the **like** clause. See "Pattern Matching on text Data" on page 4-7 for more information.
- An integer value representing the starting position of the first occurrence of *pattern* in the specified column's data should be returned in the *ret_value* parameter. "0" should be returned if *pattern* is not found.

- If the RPC returns a status of 0, Adaptive Server assumes that the RPC completed successfully.
- If the RPC returns a status other than 0, Adaptive Server assumes that the RPC did not complete successfully.

sp_statistics

Function

Returns a list of all indexes on a single table, determined by the *table_qualifier, table_owner,* and *table_name* parameters.

Syntax

```
sp_statistics table [, owner]
   [, table_qualifier] [, index_name] [, is_unique]
```

Parameters

table – varchar(30) – the name of the object containing the index.

owner – varchar(30) – the name of the object owner. If the parameter is not specified, **sp_statistics** should use the default rules of the underlying DBMS to determine which table's index to return. Support of this parameter is mandatory.

table_qualifier – varchar(30) – the database portion or first-part of a three-part name. Adaptive Server supports three-part naming in various DBMS products, which means tables can be directly identified and queried using a three-part name. Support for this parameter is mandatory.

index_name – the index name. Support for this parameter is optional.

is_unique – the indexes to be returned. Enter "y" if only unique indexes are to be returned. Support for this parameter is optional.

Comments

- If the specified table does not exist, an empty result set should be returned.
- The following table shows the results set:

Table 5-4: Results set for sp_statistics

Column	Datatype	Description
table_qualifier	*varchar(32)*	This column supports three-part naming in various DBMS products, which means tables can be directly identified and queried using a unique three-part name. This column represents the database portion of the three-part name.

Table 5-4: Results set for sp_statistics (continued)

Column	Datatype	Description
table_owner	*varchar(32)*	This column represents the owner portion of the three-part name.
table_name	*varchar(32)*	This column represents the table or view name portion of the three-part name. This field cannot be NULL.
non_unique	*smallint*	The value ”0” means unique, and “1” means not unique. This field cannot be NULL.
index_qualifier	*varchar(30)*	
index_name	*varchar(32)*	This field cannot be NULL.
type	*smallint*	The value “0” means statistics for a table, “1” means clustered, “2” means hashed, and “3” means other. This field cannot be NULL.
seq_in_index	*smallint*	This field cannot be NULL.
column_name	*varchar(32)*	This field cannot be NULL.
collation	*char(1)*	The value “A” means ascending, “D” means descending, and “NULL” means not applicable. Adaptive Server ignores descending indexes.
cardinality	*int*	Number of rows in the table or unique values in the index.
pages	*int*	Number of pages needed to store the index or table.

- The indexes in the results set should appear in ascending order by the columns *non_unique*, *type*, *index_name*, and *seq_in_index*.
- The index type *clustered* refers to an index in which the data in the table is stored in the order of the index. This corresponds to Adaptive Server clustered indexes.

sp_tables

Function

Returns a list of objects that can be queried in the current DBMS environment, that is, any object that can appear in a **from** clause.

Syntax

```
sp_tables [table] [, owner]
   [, table_qualifier][, table_type]
```

Parameters

table – varchar(30) – the name of the specific object.

owner – varchar(30) – the name of the object owner.

table_qualifier – varchar(30) – the database portion or first-part of a three-part name. Adaptive Server supports three-part naming in various DBMS products, which means tables can be directly identified and queried using a three-part name. Support for this parameter is mandatory.

table_type – a list of values, separated by commas, giving information about all tables of the table type(s) specified, including the following:

```
"'TABLE', 'SYSTEM TABLE', 'VIEW'"
```

For example:

```
sp_tables @table_type = "'TABLE', 'VIEW'"
```

This procedure returns information about all tables in the current database of the type *TABLE* and *VIEW* and excludes information about system tables.

Support for this parameter is optional.

➤ ***Note***

Single quotation marks must surround each table type, and double quotation marks must enclose the entire parameter. Table types must be entered in uppercase.

Comments

- If the specified table does not exist, an empty result set should be returned.
- The following table shows the results set:

Table 5-5: Results set for sp_tables

Column	Datatype	Description
table_qualifier	*varchar(30)*	This column supports three-part naming in various DBMS products, which means tables can be directly identified and queried using a unique three-part name. This column represents the database portion of the three-part name.
table_owner	*varchar(30)*	This column represents the owner portion of the three-part name.
table_name	*varchar(30)*	This column represents the table or view name portion of the three-part name. This field cannot be NULL.
table_type	*varchar(32)*	One of the following: “TABLE”, “VIEW”, “SYSTEM TABLE”, “SYNONYM”, “ALIAS” (Adaptive Server does not support “SYNONYM” and “ALIAS”). This field cannot be NULL.
remarks	*varchar(254)*	

sp_textvalid

Function

Determines whether a given text pointer is valid.

Syntax

```
sp_textvalid [table_qualifier, ][owner, ]table,
   column, txtptr, ret_value
```

Parameters

table_qualifier – varchar(30) – the database portion or first-part of a three-part name. Adaptive Server supports three-part naming in various DBMS products, which means tables can be directly identified and queried using a three-part name. Support for this parameter is mandatory.

owner – varchar(30) – the name of the object owner.

table – varchar(30) – the name of the object containing the *text* column.

column – varchar(30) – the name of the *text* column within *table.*

txtptr – binary(16) – a Specialty Data Store-specific handle for identifying *text* data.

ret_value – int – the output parameter in which "1" is returned for a valid text pointer and "0" is returned for an invalid text pointer.

Comments

- **sp_textvalid** is only issued if the Specialty Data Store supports *text* and *image* text pointers.
- **sp_textvalid** is used for *text* and *image* processing to handle the **textvalid()** function. See "Processing the textvalid Function" on page 4-10 for more information.
- The validity of the text pointer should be returned in the *ret_value* parameter – "1" for valid, "0" for invalid.
- If the RPC returns a status of "0", Adaptive Server assumes that the RPC completed successfully.
- If the RPC returns a status other than "0", Adaptive Server assumes that the RPC did not complete successfully.

sp_thread_props

Function

Enables a client to show or set various thread properties supported by the Specialty Data Store.

Syntax

sp_thread_props [*property_name* [, *property_value*]]

Parameters

property_name – the name of the thread property to be set or shown.

property_value – the value to which the thread property is to be set.

Comments

- In Adaptive Server 11.5, this RPC is used to set or reset the passthrough mode property only.
- If no parameters are given, a list of all properties and their current values is returned.
- If only *property_name* is given, only the value of that property is shown.
- If *property_value* is given, *property_name* must also be given.
- The *passthrough mode* property can be set to “1” (entering passthrough mode) or “0” (exiting passthrough mode).
- If no parameters are given, or only *property_name* is given, a single result set consisting of property name(s) and value(s) is returned.
- If the RPC returns a status of “0”, Adaptive Server assumes that the RPC completed successfully.
- If the RPC returns a status other than “0”, Adaptive Server assumes that the RPC did not complete successfully.

ODBC Datatypes

Following are the datatype code numbers and matching datatype names that are returned in the *data_type* column for **sp_columns.** The source for the description is the Open Database Connectivity API.

Table 5-6: ODBC datatype codes

Name	Type
char	1
decimal	3
double precision	8
float	6
integer	4
numeric	2
real	7
smallint	5
varchar	12

Table 5-7: ODBC extended datatype codes

Name	Type
binary (bit datatype)	-2
bigint	-5
bit	-7
date	9
long varbinary	-4
long varchar	-1
time	10
timestamp	11
tinyint	-6
varbinary (bit varying datatype)	-3

Adaptive Server Datatypes

Following are the datatype code numbers and matching datatype names that are returned in the *ss_data_type* column for **sp_columns**.

Table 5-8: Adaptive Server datatype codes

Name	Type
binary	0x2D
bit	0x32
char	0x2F
small datetime	0x3A
datetime	0x3D
decimal	0x6A
float	0x3E
real	0x3B
image	0x22
tinyint	0x30
smallint	0x34
int	0x38
smallmoney	0x7A
money	0x3C
numeric	0x6C
text	0x23
varbinary	0x25
varchar	0x27

Index

D

E

F

G

H

I

R

S

T

U

V

W

Y

Z