# New Features for Adaptive Server Version 12.5

Adaptive Server version 12.5 Beta

Document ID: 37753-01-1250-Beta

Last Revised: October 31, 2000

**Principal author:** Emeryville Technical Publications

**Document ID:** 37753

This publication pertains to Adaptive Server version 12.5 Beta of the Sybase database management software and to any subsequent version 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

## Sybase Trademarks

## Restricted Rights

# Table of Contents

# 1 New calculation for total memory in Adaptive Server

This section describes the changes in Adaptive Server version 12.5 for calculating total memory.

Configuration parameters allow various Adaptive Server resources to be configured. Some of the configurable resources consume memory. The changes in Adaptive Server version 12.5, effect the configuration parameters which direct the consumption of memory.

In earlier versions of Adaptive Server, **total memory** calculated the memory required by the parameters used to configure memory (that is, those configuration parameters displayed with **sp_configure "Memory Use"**). Adaptive Server first allocated memory to user-configurable parameters. Any remaining memory was then divided between the default data cache and the procedure cache, based on the value of the **procedure cache percent** parameter.

With Adaptive Server version 12.5, you specify each memory requirement with an absolute value, using **sp_configure**. You also specify the size of the procedure and default data caches in an absolute value. Resources can then be allocated dynamically during runtime. The way memory is dynamically allocated depends upon how you configure **allocate max shared memory** and **dynamic allocation on demand**.

Also, many of the configuration parameters that were static (you had to restart the server for them to take effect) are now dynamic; you do not have to restart the server to reconfigure the server's memory.

You set the maximum amount of memory that Adaptive Server can use with **max total_memory**. Resources are then allocated dynamically during runtime. The procedure and data cache sizes are now represented in absolute values, rather than in a percentage or a relative value.

Adaptive Server allocates many memory increases dynamically; typically, it does not decrease memory dynamically. It is important that you accurately assess the needs of your system, because you will need to restart the server if you decrease the memory configuration parameters.

For a complete description of the changes to configuration parameters, see "Configuration parameter changes" on page 1-8.

Figure 1-1 and Figure 1-2 illustrate the difference between how Adaptive Server version 12.0 and earlier servers added memory and how Adaptive Server version 12.5 adds memory:

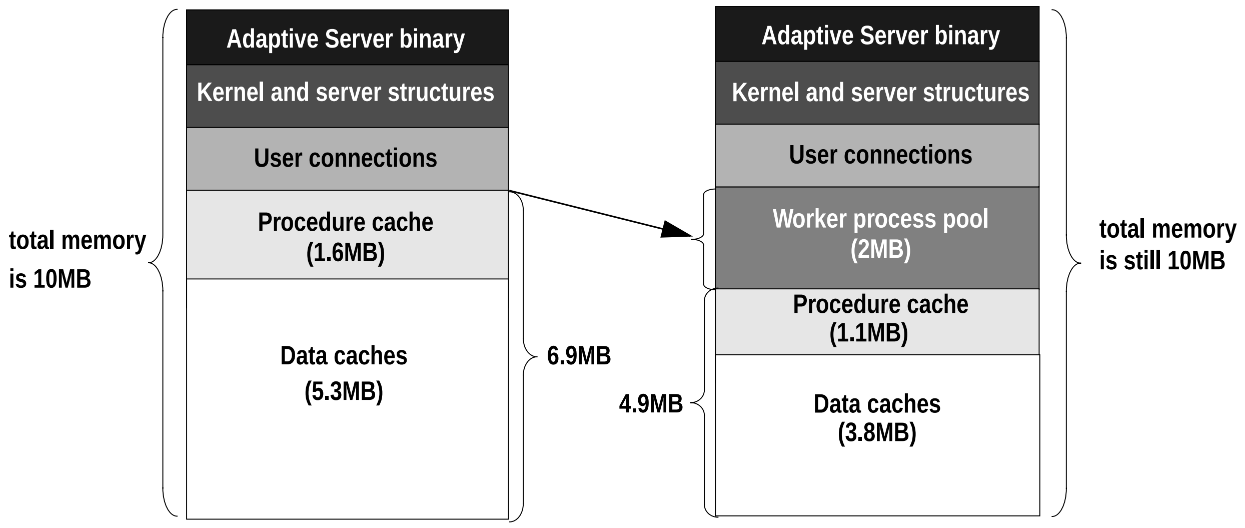

**Figure 1-1: Adding a worker process pool in Adaptive Server version 12.0**

In this example, the System Administrator adds a 2MB worker process pool. However, because the **total memory** value must remain the same, the new pool of memory reduces the procedure and data caches by.5MB and 1.5MB respectively. Figure 2 shows the same scenario on a 12.5 Adaptive Server:

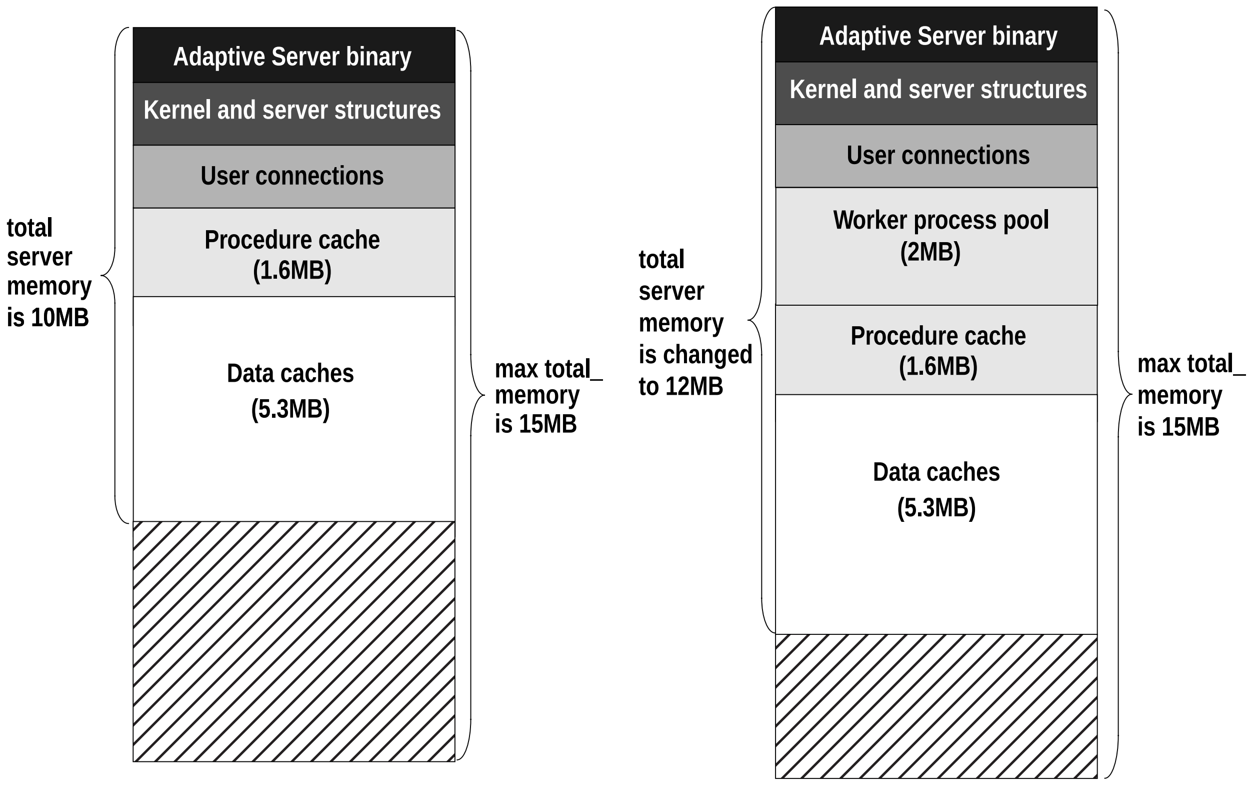
Adaptive Server binary
Kernel and server structures
User connections
Procedure cache (1.6MB)
Data caches (5.3MB)
total server memory is 10MB
max total_ memory is 15MB
Adaptive Server binary
Kernel and server structures
User connections
Worker process pool (2MB)
Procedure cache (1.6MB)
Data caches (5.3MB)
total server memory is changed to 12MB
max total_ memory is 15MB

**Figure 1-2: Adding a worker process pool in Adaptive Server version 12.5**

In version 12.5, even though the 2MB worker process pool is added to the server, the procedure and data caches maintain their originally configured sizes; 1.6MB and 5.3MB, respectively. Because **max total_memory** is 5MB larger than the **total memory** size, it also easily absorbs the added memory pool. If the new worker process pool brings the size of the server above the limit of **max total_memory**, the command to increase the worker process pool fails. If this happens, set the value of **max total_memory** to a value greater than the logical memory to resolve the problem.

For more information on logical memory, see "How does Adaptive Server allocate memory?" on page 1-6.

## Setting the maximum server memory

The sum of the values of the memory configuration parameters cannot be set beyond the value of **max total_memory.** If the amount of memory to which you set **max total_memory** is not sufficient, Adaptive

Server cannot start. **sp_configure "total memory"** displays the amount of memory for the current Adaptive Server configuration.

The **allocate max shared memory** parameter allows you to either allocate all the memory specified by **max total_memory** at start-up or to allocate only the memory required by the logical memory specification during start-up.

For example, if you set **allocate max shared memory** to 0 (the default) and **max total_memory** to 500MB, but the server configuration only requires 100MB of memory at start-up, Adaptive Server allocates the remaining 400MB only when it requires the additional memory. However, if you set **allocate max shared memory** to 1, Adaptive Server allocates the entire 500MB when it starts.

The advantage of allocating all the memory at start-up is that there is no slow time while the server is readjusting for the additional memory. However, if you do not predict memory growth properly, and **max total_memory** is set to a large value, you may be wasting physical memory. Since you cannot dynamically decrease memory configuration parameters, in many situations, it is important that you take into account other memory requirements.

## How much memory does Adaptive Server need?

The total memory Adaptive Server requires to start is the sum of all memory configuration parameters plus the *size of the procedure cache* plus the *size of the buffer cache,* where the *size of the procedure cache* and the *size of the buffer cache* are expressed in round numbers rather than in percentages.

To determine the total amount of memory Adaptive Server is using at a given moment, use **sp_configure**. For example:

```
sp_configure "total memory"

Parameter Name   Default      Memory Used   Config Value      Run Value
 -------------   ---------    -----------   -----------       ---------
total memory     33792      48148       24074                 23538
```

The run value for the total memory parameter shows the total logical memory being consumed by the current Adaptive Server configuration.  This example will provide different values for each instillation, as no two Adaptive Server's are likely to be configured in exactly the same fashion.

### Decreasing memory configuration parameters

You can dynamically decrease memory configuration parameters under these conditions:

- You have memory that has been allocated but not used, and
- The **dynamic allocation on demand** configuration parameter is set to 1.

(If the value of **dynamic allocation on demand** is 0, you can not decrease memory configuration parameters dynamically.)

When **dynamic allocation on demand** is 1, you can decrease memory configuration parameters dynamically, if there has been no actual increase in memory use. For example, if the **number of user connections** is 100, and you decide that 50 is a more appropriate configuration value, the value of **number of user connections** may be dynamically decreased as long as no more than 50 user connections are currently in use. If the number of user connections is greater than 50, you can not decrease the memory configuration parameter to 50, because by doing so, you are attempting to release memory already in use.

### Determining the procedure cache size

**procedure cache size** specifies the size of your procedure cache in 2K pages. For example:

```
sp_configure “procedure cache size”
```

```
Parameter Name          Default  Memory Used   Config Value  Run Value
----------------------  -------  -----------   ---------     --------
procedure cache size    3271     8248          20000         20000
```

The amount of memory used for the procedure cache is 8.248MB. To set the procedure cache to a different size, issue the following:

```
sp_configure “procedure cache size”, new_size
```

This example resets the procedure cache size to 10000 2k pages (20MB):

```
sp_configure “procedure cache size”, 10000
```

➤ ***Note***

The **procedure cache percent** configuration parameter has been removed from Adaptive Server 12.5.

## Determining the default data cache size

Both **sp_cacheconfig** and **sp_helpcache** display the current default data cache in MB. For example, the following shows an Adaptive Server configured with 19.86MB of default data cache:

```
sp_cacheconfig

Cache Name          Status       Type     Config Value      Run Value
------------------- ----------   -------- ------------      ----------
default data cache  Active       Default      0.00 Mb       19.86Mb
                                          ------------      --------
                                 Total        0.00Mb        19.86 Mb
========================================================================
Cache: default data cache,   Status: Active,   Type: Default
      Config Size: 0.00 Mb,   Run Size: 19.86 Mb
      Config Replacement: strict LRU,   Run Replacement: strict LRU
      Config Partition:           1,   Run Partition:           1
IO Size     Wash Size    Config Size  Run Size     APF Percent
--------    ---------    ------------ ------------ -----------
2 Kb        4066 Kb      0.00 Mb      19.86 Mb         10
```

To change the default data cache, issue **sp_cacheconfig,** and specify "default data cache." For example, to change the default data cache to 25MB, enter:

```
sp_cacheconfig "default data cache", "25MB"
```

You must restart Adaptive Server for this change to take effect.

In pre-12.5 versions of Adaptive Server, the size of the default data cache was determined by the amount of remaining memory available to the server.

In Adaptive Server 12.5, the default data cache is an absolute value. During the upgrade process, Adaptive Server sets the default data cache size to the run value of the default data cache in the configuration file.

## How does Adaptive Server allocate memory?

Memory exists in Adaptive Server as logical or physical memory:

- Logical memory – is the sum of all the configuration parameter values that you configure using **sp_configure**, and is the amount of memory that Adaptive Server requires at the present moment. This value can go up or down depending on resource needs.
- Physical memory – is the sum of all shared memory segments in Adaptive Server. That is, physical memory is the amount of memory on which Adaptive Server is running at a given

moment. You can verify this number with the global variable *@@tot_phymem*. The value of *@@tot_phymem* can only increase because Adaptive Server does not release memory. You can decrease the amount of physical memory only by changing the configuration parameters and restarting Adaptive Server.

**dynamic allocation on demand** is the memory configuration parameter that allows you to determine whether your memory resources are allocated to physical memory as soon as they are requested or only as they are needed. Setting **dynamic allocation on demand** to 1 will allocate memory configuration changes as needed, and setting it to 0 will allocate the total memory requested in the memory configuration change at the time of the memory reconfiguration.

For example, assume you have set the value of **dynamic allocation on demand** to 1. If you change **number of user connections** to 1000, the logical memory equals 1000 multiplied by the amount of memory per user. If the amount of memory per user is 112K, then the logical memory per user is 112MB (1000 x 112).

This is the maximum amount of memory that the **number of user connections** configuration parameter is allowed to use. However, if only 500 users are connected to the server, the amount of physical memory used by the **number of user connections** parameter is 56MB (500 x 112).

Now assume the value of **dynamic allocation on demand** is 0; when you change **number of user connections** to 1000, all 112MB will be converted to physical memory, because all user connection resources are configured immediately.

Optimally, you should organize Adaptive Server's memory so that the amount of physical memory is less than the amount of logical memory, which is less than the **max total_memory.** This can be achieved, in part, by setting the value of **dynamic allocation on demand** to 1, and setting the value of **allocate max shared memory** to 0.

## If you are upgrading

During the upgrade process, pre-12.5 Adaptive Server configuration values for **total memory, procedure cache percent,** and **min online engines** are used to calculate the new values for **procedure cache size** and **number of engines at startup**. Adaptive Server computes the size of the default data cache during the upgrade and writes this value to the configuration file. If the computed sizes of the data cache or procedure cache are less than the default sizes, they are reset to the default. During the upgrade, **max total_memory** is set to the value of **total memory** specified in the configuration file.

If this memory is insufficient, Adaptive Server does not start and the following error message is printed to the error log:

```
The value of the ‘max total_memory’ parameter
(%ID) defined in the configuration file is not
high enough to set the other parameter values
specified in the configuration file.  ‘max
total_memory’ should be greater than the logical
memory (%ID).
```

You should reset the value of **max total_memory** to comply with the resource requirements.

You can use the **verify** option of **sp_configure** to verify any changes you make to the configuration file without having to restart Adaptive Server. The syntax is:

```
sp_configure “configuration file”, 0, “verify”,
   “full_path_to_file”
```

See the *System Administration Guide* for more information.

## Configuration parameter changes

The following sections describe the configuration parameter changes for Adaptive Server version 12.5.

### Configuration parameters that are now dynamic

In earlier versions of Adaptive Server, the following configuration parameters were static, but now dynamically change Adaptive Server‘s use of shared memory:

- **additional network memory**
- **audit queue size**
- **cpu grace time**
- **deadlock pipe max messages**
- **default database size**
- **default fill factor percent**
- **disk i/o structures**
- **errorlog pipe max messages**
- **max cis remote connections**

- **memory per worker process**
- **number of alarms**
- **number of aux scan descriptors**
- **number of devices**
- **number of dtx participants**
- **number of java sockets**
- **number of large i/o buffers**
- **number of locks**
- **number of mailboxes**
- **number of messages**
- **number of open databases**
- **number of open indexes**
- **number of open objects**
- **number of pre-allocated extents**
- **number of user connections**
- **number of worker processes**
- **open index hash spinlock ratio**
- **open index spinlock ratio**
- **open object spinlock ratio**
- **partition groups**
- **partition spinlock ratio**
- **permission cache entries**
- **plan text pipe max messages**
- **print recovery information**
- **process wait events**
- **size of global fixed heap**
- **size of process object heap**
- **size of shared class heap**
- **size of unilib cache**
- **sql text pipe max messages**
- **statement pipe max messages**
- **tape retention in days**
- **time slice**

- **user log cache spinlock ratio**

## Changed configuration parameter

In earlier versions, **total memory** described the amount of memory in 2K units, which Adaptive Server allocated from the operating system. Because the memory allocation is dynamic for Adaptive Server version 12.5, the **total memory** parameter specifies only the logical memory required for the current configuration requirements of Adaptive Server.

### *total memory*

| Summary information | |
|---|---|
| Name in pre-11.0 release | **memory** |
| Default value | N/A |
| Range of values | N/A |
| Status | Read-only |
| Display level | Intermediate |
| Required role | System Administrator |

Displays the logical memory for the current configuration of Adaptive Server. The logical memory is the amount of memory that Adaptive Server's current configuration uses. You cannot use **total memory** to set any of the memory configuration parameters.

## New configuration parameters

The following configuration parameters have been added to Adaptive Server version 12.5.

### max total_memory

| Summary information | |
|---|---|
| Name in pre-11.0 release | N / A |
| Default value | Platform-dependent |
| Range of values | Platform-dependent minimum – 2147483647 |
| Status | Dynamic |
| Display level | Basic |
| Required role | System Administrator |

Specifies the maximum amount of physical memory that you can configure Adaptive Server to allocate. **max total_memory** must be greater than the logical total memory consumed by the current configuration of Adaptive Server.

There is no performance penalty for configuring Adaptive Server to use the maximum memory available to it on your computer. However, assess the other memory needs on your system, or Adaptive Server may not be able to acquire enough memory to start.

See Chapter 18, "Configuring Memory," for instructions on how to maximize the amount of **max total_memory** for Adaptive Server.

#### *If Adaptive Server cannot start*

Adaptive Server must have the amount of memory available that is specified by **max total_memory.** If the memory is not available, Adaptive Server will not start. If this occurs, reduce the memory requirements for Adaptive Server by manually changing the value of **max total_memory** in the server's configuration file. You may also need to reduce the values for other configuration parameters that require large amounts of memory. Then restart Adaptive Server to use the memory needed by the new values. If Adaptive Server fails to start because the total of other configuration parameter values is higher than the **max total_memory** value, see Chapter 18, "Configuring Memory," for information about configuration parameters that use memory.

### procedure cache size

| Summary information | |
|---|---|
| Name in pre-11.0 release | N/A |
| Default value | 3271 |
| Range of values | 3271 – 2147483647 |
| Status | Dynamic |
| Display level | Basic |
| Required role | System Administrator |

Specifies the size of the procedure cache in 2K pages. Adaptive Server uses the procedure cache while running stored procedures. If the server finds a copy of a procedure already in the cache, it does not need to read it from the disk. Adaptive Server also uses space in the procedure cache to compile queries while creating stored procedures.

Since the optimum value for **procedure cache size** differs from application to application, resetting it may improve Adaptive Server's performance. For example, if you run many different procedures or ad hoc queries, your application will use the procedure cache more heavily, so you may want to increase this value.

#### *If you are upgrading*

In earlier versions of Adaptive Server, the size of the procedure cache was based on a percentage of the available memory. If you are upgrading to version 12.5, the procedure cache size is set to the size of the procedure cache determined at run time during the upgrade process. Unlike pre-12.5 Adaptive Server, this parameter is dynamically configurable subject to the amount of **max total_memory** currently configured.

### number of engines at startup

| Summary information | |
|---|---|
| Name in pre-11.0 release | N/A |
| Default value | 1 |
| Range of values | 1 – number of CPUs on machine |
| Status | Static |
| Display level | Basic |
| Required role | System Administrator |

Specifies the number of engines Adaptive Server brings online at start-up. You cannot set number of engines at start-up to a number greater than the number of CPUs on your machine.

### allocate max shared memory

| Summary information | |
|---|---|
| Name in pre-11.0 release | N/A |
| Default value | 0 |
| Range of values | 0,1 |
| Status | Dynamic |
| Display level | Basic |
| Required role | System Administrator |

**allocate max shared memory** determines whether Adaptive Server allocates all the memory specified by **max total_memory** at start-up or only the amount of memory the configuration parameter indicates.

By setting **allocate max shared memory** to 0, you ensure that Adaptive Server uses only the amount of shared memory required by the current configuration. When you set **allocate max shared memory** to 0, Adaptive Server allocates only the amount of memory required by the configuration parameters at start-up, which will be a smaller value than **max total_memory**. If you change the configuration parameters (for example if you increase the **number of user connections**,) Adaptive Server then increases the amount of physical memory used by the amount of memory required to increase the value of **number of user connections**.

If you set **allocate max shared memory** to 1, Adaptive Server allocates all the memory specified by **max total_memory** at start-up. If **allocate max shared memory** is 1, and if you increase **max total_memory,** additional shared memory elements are obtained. This means that Adaptive Server will always have the memory required for any memory configuration changes you make and there is no slow time while the server readjusts for additional memory. However, if you do not predict memory growth accurately, and **max total_memory** is set to a large value, you may waste physical memory.

### dynamic allocation on demand

| Summary information | |
|---|---|
| Name in pre-11.0 release | N / A |
| Default value | 1 |
| Range of values | 0, 1 |
| Status | Dynamic |
| Display level | Basic |
| Required role | System Administrator |

Determines when memory is allocated for changes to dynamic memory configuration parameters.

If you set **dynamic allocation on demand** to 1, memory is allocated only as it is needed. That is, if you change the configuration for **number of user connections** from 100 to 200, the memory for each user is added only when the user connects to the server. Adaptive Server continues to add memory until it reaches the new maximum for user connections.

If **dynamic allocation on demand** is set to 0, all the memory required for any dynamic configuration changes is allocated immediately. That is, when you change the number of user connections from 100 to 200, the memory required for the extra 100 user connections is immediately allocated.

### Deleted configuration parameters

The following configuration parameters have been removed from Adaptive Server version 12.5:

- **procedure cache percent**
- **max cis remote servers**
- **max engine freelocks**
- **max roles enabled per user**
- **min online engines**
- **number of languages in cache**
- **freelock transfer block size**
- **engine adjust interval**

## New global variable

Adaptive Server version 12.5 adds the *@@tot_physmem* global variable. *@@tot_physmem* returns the current amount of shared memory used by Adaptive Server. The amount of shared memory is reported in number of pages.

To run *@@tot_physmem*, enter:

```
select @@tot_physmem
```

## New stored procedure

The following stored procedure has been added to Adaptive Server 12.5.

### sp_engine

#### Function

Enables you to bring an engine online or offline.

#### Syntax

**sp_engine {“online” | offline} [, *engine_id*]**

### Parameters

"online" – bring an engine online. Online must be specified within quotations as it is a reserved key word.

offline – take an engine offline.

*engine_id* – the ID of the engine you are bringing offline.

### Comments

- You can bring an engine online only if **max online engines** is greater than the current number of engines that are online, and if enough CPU is available to support the additional engine.
- You do not need to specify the *engine_id* to bring an engine online.
- You cannot take engine number 0 offline.

### Permissions

You need a System Administrator role to bring engines online or offline.

# 2 New limits for Adaptive Server version 12.5

Adaptive Server version 12.5 includes increases in the following limits:

- Page size
- Number of columns per table, column, row, and index size
- Number of arguments for stored procedures
- Length of expressions
- Number of expressions in a **select** statement
- Number of logins per server; number of users per database

See Table 2-10 on page 34 and Table 2-11 on page 35 for a summary of the new limits for Adaptive Server version 12.5.

## Varying logical page sizes

➤ ***Note***

Adaptive Server version 12.5 does not use the **buildmaster** binary to build the master device. Instead, Sybase has incorporated **buildmaster** functionality in the **dataserver** binary.

The dataserver command allows you to create master devices and databases with logical pages of size 2K, 4K, 8K, or 16K. Larger logical pages allow you to create larger rows, which can improve your performance because Adaptive Server accesses more data each time it reads a page. For example, a single 16K page can hold 8 times the amount of data as a 2K page, an 8K page holds 4 times as much data as a 2K page, and so on.

The logical page size is a server-wide setting; you cannot have databases with varying size logical pages within the same server. All tables are appropriately sized so that the row size is no greater than the current page size of the server. That is, rows cannot span multiple pages.

## Building a new master device

This section describes the process for creating a new master device using the buildmaster utility. The master device is built during two phases; the build phase and the boot phase. During the build phase the master device is built and booted, but it is then shut down again. You must then manually reboot the server for the boot phase. After this you can start, stop, and restart Adaptive Server whenever necessary without having to rebuild the master device.

The memory for the master device is measured in three types of memory:

- Logical memory – These are the pages that the database objects are built with. All databases and all database objects use the same logical page size. Logical page sizes come in sizes of 2K, 4K, 8K, and 16K.
- Virtual memory – This is the physical page allocation at the disk level, and is always done in 2K pages. All disk I/O is done in this virtual page size.
- Memory page size – The memory allocated and managed within Adaptive Server., and is always measured in 2K pages.

The syntax to create a new master device with **dataserver** is:

```
dataserver -d <master device name>
        . . .
        [-I interfaces_file_name]
        [-b master_device_size [k|K|m|M|g|G]
        [-z logical_page_size [k|K]
        [ [-forcebuild ]
        -c path_to_file
        [-n controller_number]
        [-w database_name_to_rewrite]
        -h
```

Where:

- **I** specifies the full path to the interfaces file.
- **b** indicates the size of the master device (for example, 3.2G).
- **z** specifies the logical page size.
- **forcebuild** overrides checks to make sure that **dataserver** is not overwriting an existing master device.
- **c** specifies the configuration file.
- **w** indicates the name of the master device you are rewriting.

- and **h** prints the syntax for the **dataserver** command.

➤ ***Note***

-b specifies the number virtual pages used to create the master device. The size of a virtual page is always 2K. -z specifies the logical (database) page size for the server, and is always 2K, 4K, 8K, or 16K.

If you specify a configuration file with the **-c** parameter, make sure all the parameters in this configuration file are compatible before you boot the server. If some of the configuration parameters are incompatible, the server may not boot. To avoid this, do not specify a configuration file when you build the master device. **buildmaster** uses all default settings if when you do not specify a configuration file.

To start an existing Adaptive Server, issue the **dataserver** command without the **-b** and **-z** options.

For example, to:

- Build a master device of the default logical page size (2K), creates a master device large enough to create all required system databases, enter:

```
dataserver -d /sybase/masterdb.dat -b
```

- Start Adaptive Server with an existing master device, enter:

```
dataserver -d /sybase/masterdb.dat
```

- Build a 100MB master device using the default logical page size (2K) and starts the server, enter:

```
dataserver -d /sybase/masterdb.dat -b100M
```

- Build a 100MB master device with a logical page size of size 4K, enter:

```
dataserver -d /sybase/masterdb.dat -b100M -z4K
```

- Build a master device of 102,400 virtual pages of size 2K, and create databases using a logical page size of 8K, and boot the server. If the total requested space (102,400 x 2K 200 MB) is insufficient to build all the required system databases using the specified logical page size, then an error message is reported, and the process fails.

```
dataserver -d /work1/masterdb.dat -b 102400 -z8K
```

## Upgrading to a server with larger page sizes

Adaptive Servers previous to version 12.5 used 2K logical page sizes. You can only upgrade existing installations to servers using the 2K logical page size. You cannot change an installation's page size by upgrading. That is, if your current Adaptive Server uses 2K logical pages, you can upgrade only to an Adaptive Server that uses 2K logical pages.

## Viewing the current server limits

To determine the size of your servers logical page size, run **dbcc serverlimis.** For example:

```
dbcc serverlimits
```

See "New dbcc commands" on page 2-41 for more information

## Backup Server and larger logical page sizes

You can dump and load databases and transaction logs only from Adaptive Servers that share the same logical page size. For example, you cannot load a dump from an Adaptive Server that uses 2K logical pages into an Adaptive Server that uses 16K logical pages.

➤ ***Note***

All existing dumps from previous to version 12.5 use a logical page size of 2K, so you can load these dumps into an Adaptive Server version 12.5 that uses a logical page size of 2K.

## Using **bcp** with enhanced limits

You can use **bcp** version 12.5 to bulk load and bulk unload data with the enhanced limits in version 12.5. For example, you can use the version 12.5 **bcp** to copy out data that includes wide rows and columns.

## Larger logical page sizes and buffers

Because buffers are sized in logical pages, if Adaptive Server uses larger logical pages, the buffers may also be larger. For example, an

Adaptive Server that uses 2K logical page size uses a large I/O mass of 16K (eight logical pages), but an Adaptive Server with 16K logical page size uses a large I/O mass of 128K (also eight logical pages).

## Number of columns and column size

The maximum number of columns you can create in a table is:

- 1024 for fixed-length columns in both all-pages locked (APL) and data-only locked (DOL) tables
- 254 for variable-length columns in an APL table.
- 1024 for variable-length columns in an DOL table

The maximum size of a column depends on:

- Whether the table includes any variable- or fixed-length columns.
- The logical page size of the database. For example, in a database with 2K logical pages, the maximum size of a column in an APL table can be as large as a single row, about 1962 bytes, less the row format over heads. Similarly, for a 4K page, the maximum size of a column in a APL table can be as large as 4010 bytes, less the row format over heads. See Table 2-1 for more information.
- If you attempt to create a table that is greater than the limits of the logical page size, create table issues an error message.

### Size of columns containing fixed-length data

Columns with fixed length data (for example, **char**, **binary**, and so on) have the following maximum sizes:

**Table 2-1: Maximum length of fixed length columns in APL and DOL tables**

| Locking scheme | Page size | Max. row size | Max. column length |
|---|---|---|---|
| All-pages locked (APL) tables | 2K (2048 bytes) | 1962 | 1960 bytes |
| | 4K (4096 bytes) | 4010 | 4008 bytes |
| | 8K (8192 bytes) | 8106 | 8104 bytes |
| | 16K (16384 bytes) | 16298 | 16296 bytes |

**Table 2-1: Maximum length of fixed length columns in APL and DOL tables**

| Locking scheme | Page size | Max. row size | Max. column length |
|---|---|---|---|
| Data-only locked (DOL) tables | 2K (2048 bytes) | 1964 | 1958 bytes |
| | 4K (4096 bytes) | 4012 | 4006 bytes |
| | 8K (8192 bytes) | 8108 | 8102 bytes |
| | 16K (16384 bytes) | 16300 | 16294 bytes if the table does not include any variable length columns |
| | 16K (16384 bytes) | 16300 (subject to a *max start* offset of varlen col = 8191) | 8191 - 6 - 2 = 8183 bytes if the table includes at least one variable-length column. (This size includes 6 bytes for the row overhead and 2 bytes for the row length field.) |

The maximum size of a fixed-length column in a DOL table with a 16K logical page size depends on whether the table contains variable-length columns. The maximum possible starting offset of a variable-length column is 8192. If the table has any variable length columns, the sum of the fixed-length portion of the row, plus row over heads, cannot exceed 8191 bytes, and the maximum possible size of all the fixed-length columns is restricted to 8183 bytes, when the table contains any variable-length columns.

## Variable-length columns in APL tables

Variable-length columns (for example, **varchar**, **varbinary**, and so on) in an APL table have the following minimum row overhead for each row:

- 2 bytes for the initial row overhead.
- 2 bytes for the row length.
- 2 bytes for the column-offset table at the end of the row. This is always *n + 1* bytes, where *n* is the number of variable-length columns in the row.

So, a single-column table has an overhead of at least six bytes, plus additional overhead for the adjust table. The column size is the sum of the column length, the number of bytes for the adjust table, and the six-byte overhead. This is described in Table 2-2.

**Table 2-2: Maximum size of variable length columns in an APL table**

| Page size | Max. row length | Max. column length |
|---|---|---|
| 2k (2048 bytes) | 1962 | 1948 |
| 4k (4096 bytes) | 4010 | 3988 |
| 8K (8192 bytes) | 8196 | 8158 |
| 16K (16384 bytes) | 16298 | 16228 |

### Variable-length columns that exceed the logical page size

If your table uses 2K logical pages, you can create some variable length columns whose total row length exceeds the maximum row length for a 2K page size. This allows you to create tables where some, but not all, variable length columns contain the maximum possible size. However, when you issue **create table**, you receive a warning message that says the resulting row size could exceed the maximum possible row size, causing a future insert or update to fail.

For example., on a server that uses 2K pages, if you create a table that contains a variable-length column with a length of 1975 bytes, Adaptive Server creates the table and the column. However, if you attempt to insert data of this length into this column, the data’s length exceeds the maximum length of the row, which is 1962 bytes, as listed in Table 2-2.

## Variable length columns in DOL tables

For a single, variable-length column in a DOL table, the minimum row overhead for each row is:

- 6 bytes for the initial row overhead.
- 2 bytes for the row length.
- 2 bytes for the column offset table at the end of the row. Each column offset entry is 2 bytes. There are *n* such entries, where *n* is the number of variable-length columns in the row.

The total overhead is 10 bytes. There is no adjust table for DOL rows. The actual variable-length column size is:

```
The actual column length + 10 bytes of overhead <= maximum row length
```

This is illustrated in Table 2-3:

**Table 2-3: Maximum size of variable-length columns in a DOL table**

| Page size | Max. row length | Max. column length |
|---|---|---|
| 2K (2048 bytes) | 1964 | 1954 |
| 4K (4096 bytes) | 4012 | 4002 |
| 8K (8192 bytes) | 8108 | 7998 |
| 16K (16384 bytes) | 16300 | 16290 |

DOL tables must have an offset of less than 8192 bytes. For example, for a server that uses 8K pages, the following **insert** fails because the offset adds up to more than 8192 bytes:

```
create table t1(
    c1 int not null,
    c2 varchar(5000) not null
    c3 varchar(4000) not null
    c4 varchar(10) not null

    ... more fixed length columns

    cvarlen varchar(nnn)) lock datarows
```

The offset for columns *c2*, *c3*, and *c4* is 9010, so the entire insert fails.

## Organizing columns in DOL tables by size of variable-length columns

For DOL tables that use variable-length columns, arrange the columns so the longest columns are placed towards the end of the table definition. This allows you to create tables with much larger rows than is possible if the large column appears at the beginning of the table definition. For instance, in a 16K page server, the following table definition is acceptable:

```
create table t1 (
    c1 int not null,
    c2 varchar(1000) null,
    c3 varchar(4000) null,
    c4 varchar(9000) null) lock datarows
```

However, the following table definition is not acceptable. The potential start offset for column *c2* is greater than the 8192-byte limit because of the proceeding 9000 byte *c4* column:

```
create table t2 (
    c1 int not null,
    c4 varchar(9000) null,
    c3 varchar(4000) null,
    c2 varchar(1000) null) lock datarows
```

The table is created, but Adaptive Server issues a warning message indicating the column that is causing the problem.

## Index size

Table 2-4 describes the limits for index size for APL and DOL tables:

**Table 2-4: Index row-size limits**

| Page size | User-visible Index row-size limit | Internal index row-size limit |
|---|---|---|
| 2K (2048 bytes) | 600 | 650 |
| 4L (4096 bytes) | 1250 | 1310 |
| 8K (8192 bytes) | 2600 | 2670 |
| 16K (16384 bytes) | 5300 | 5390 |

You can create tables with columns wider than the limit for the index key; however you cannot index these columns. For example, if you perform the following on a 2K page server:

```
create table t1 (
c1 int
c2 int
c3 char(700))
```

and then try to create an index on *c3*, the command fails and Adaptive Server issues an error message because column *c3* is larger than the index row-size limit (600 bytes).

## Simplified units for *disk init*, *disk reinit*, *create database*, and *alter database*

**disk init**, **disk reinit**, **create database**, and **alter database** allow you to specify unit sizes for space allocation in terms of the number of pages, kilobytes, megabytes, or gigabytes. For the following syntax, the size parameter that supports unit specifiers is in **bold face** type.

This is the **disk init** syntax:

```
disk init
name = "device_name" ,
physname = "physicalname" ,
vdevno = virtual_device_number ,
size = size_of_device
[, vstart = virtual_address
    , cntrltype = controller_number ]
[, contiguous]
[, dsync = {true|false}]
```

This is the **disk reinit** syntax:

```
disk reinit
name = "device_name",
physname = "physicalname" ,
vdevno = virtual_device_number ,
size = size_of_device
[, vstart = virtual_address
    , cntrltype = controller_number]
[, dsync = {true|false}
```

This is the syntax for **create database**:

```
create database database_name
[on {default | database_device} [= size]
    [, database_device [= size]]...]
[log on database_device [= size]
    [, database_device [= size]]...]
[with {override | default_location = "pathname"}]
[for {load | proxy_update}]
```

This is the syntax for **alter database**:

```
alter database database_name
[on {default | database_device } [= size]
    [, database_device [= size]]...]
[log on { default | database_device } [ = size ]
    [ , database_device [= size]]...]
[with override]
[for load]
[for proxy_update]
```

Where size is ‘k’ or ‘K’ (kilobytes), ‘m’ or ‘M’ (megabytes), and ‘g’ or ‘G’ (gigabytes).

The following apply to the syntax for these commands:

- If you do not include a unit specifier you do not have include quotes around *size*. However, you must use quotes if you include a unit specifier.
- Sybase recommends that you always include the unit specifier in both the **disk init** and **create database** commands to avoid confusion in the actual number of pages allocated.
- You can specify the size as a **float** integer, but the size is rounded down to the nearest whole-number multiple of logical pages.
- If you do not specify a unit specifier:
  - **disk init** and **disk reinit** use the virtual page size of 2K.
  - The **size** argument for **create database** and **alter database** is in terms of megabytes of disk space. This value is converted to the number of logical pages the master device was built with.
  - Because Adaptive Server allocates space for databases in chunks of 256 logical pages, **create database** and **alter database** round-down the size specified to the nearest multiple of allocation units.
  - The minimum size of a database depends on the logical page size used by the server, described in Table 2-4

**Table 2-5: Minimum database sizes:**

| Logical page size | Minimum database size |
|---|---|
| 2K | 2 megabytes |
| 4K | 4 megabytes |
| 8K | 8 megabytes |
| 16K | 16 megabytes |

### *Examples*

The following creates a 20MB device with a page size in units of 2K pages (note that the **size** parameter has no unit specifier):

```
disk init
name = “books_dev”,
physname = “/sybase/devices/books.dat”,
vdevno = 21,
size = 10240
```

The following specifies a 20MB device:

```
disk init
name = “comp_books_dev”,
physname = “/sybase/devices/comp_books.dat”,
vdevno = 25,
size = "20M"
```

The following creates a 2MB database on the default device:

```
create database pubs3
on default = "2M"
```

The following creates a 20MB database on the *pubs_data* and places a 5MB log on a separate device (note that the **size** parameter has no unit specifier):

```
create database pubs
on pubs_data = 20
log on pubs_log = "5M"
```

## Space allocation

The size of Adaptive Server‘s logical pages (2K, 4K, 8K, or 16K) determines the server’s space allocation. Each allocation page, OAM page, data page, index page, text page, and so on are built on a logical page. For example, if the logical page size of Adaptive Server is 8K, each of these page types are 8K in size. All of these pages consume the entire size specified by the size of the logical page.

OAM pages have a greater number of OAM entries for larger logical pages (for example, 8K) than for smaller pages (2K).

Table 2-6 describes the space allocation per different logical pages:

**Table 2-6: Space allocation for different logical page sizes**

| Item | 2K | 4K | 8K | 16K | Comment (see key below) |
|---|---|---|---|---|---|
| Size of APL page header (bytes) | 32 | 32 | 32 | 32 | 1 |
| Size of DOL page header (bytes) | 44 | 44 | 44 | 44 | 1 |
| Row-offset specifier per row (APL and DOL) at the end of the page | 2 bytes | 2 bytes | 2 bytes | 2 bytes | 1 |
| Number of logical pages in an extent | 8 | 8 | 8 | 8 | 1 |
| Size of one extent | 16K | 32K | 64k | 128K | 2 |

| Item | 2K | 4K | 8K | 16K | Comment (see key below) |
|---|---|---|---|---|---|
| Size of one in-memory buffer for a logical page | 2K | 4K | 8K | 16K | 2 |
| Size of one large mass (1 extent worth of buffers) | 16K | 32K | 64K | 128K | 2 |
| Number of extents in an allocation unit (AU) | 32 | 32 | 32 | 32 | 1 |
| Number of logical pages in an allocation unit | 256 | 256 | 256 | 256 | 1 |
| Size of an AU | 0.5MB | 1MB | 2MB | 4MB | 2 |
| Minimum size of a usable disk that can be created by **disk init** is either 1M or the size of 1 AU, whichever is larger | 1MB | 1MB | 2MB | 4MB | 2 |
| Minimum size of a single disk piece that can be allocated to a database using **create database** or **alter database** commands | Larger of 1M or 1 AU | | | | 2 |
| Minimum size of a single disk piece that can be allocated for the log | Larger of 1M or 1 AU | | | | 2 |
| Default size of database if the size is not specified by the user (and if not configured via default database size); the same size as the **model** database | 4 AUs (2M) | 4 AUs (4M) | 4 AUs (8M) | 4 AUs (16M) | 2 |
| Default size of a single disk piece that you can allocate to a database with **create database** or **alter database** | 2M | 4M | 8M | 16M | 2 |
| Number of OAM entries in one OAM page | 250 | 506 | 1018 | 2042 | 2varies with page size |
| Space tracked by one OAM page if only one extent on an AU is allocated to a given object | 3.9M | 15.8M | 63.62M | 255.2M | 2 |
| Useful space for data in one text page (bytes) | 1800 | 3600 | 7500 | 16200 | 2 |

### Comments

- 1 – unchanged from previous versions of Adaptive Server.

- 2 – Varies according to logical page size

### Space overhead requirements

Regardless of the logical page size it is configured for, Adaptive Server allocates space for objects (tables, indexes, text page chains) in extents, each of which is eight logical pages. That is, if a server is configured for 2K logical pages, it allocates one extent, 16K, for each of these objects; if a server is configured for 16K logical pages, it allocates one extent, 128K, for each of these objects.

This is also true for system tables. For small tables, space usage is higher using larger logical pages than for smaller logical pages. For example, for a server configured for 2K logical pages, *systypes*, which has about 31 short rows, and both a clustered and a non-clustered index, uses about 3 extents, or 48K of space. If you migrate the server to use 8K pages, the space required for *systypes* is still 3 extents, about 192K of space. For a server configured for 16K, *systypes* requires about 384K.

Databases, also, are affected by larger page sizes. Each database includes the system catalogs and their indexes. If you migrate from a smaller to larger logical page size, you must account for the amount of disk space each database requires. Table 2-4 on page 25 lists the minimum size for a database on each of the logical page sizes.

### Number of rows per data page

The number of rows allowed for a DOL data page is determined by:

- The page size
- A 10-byte overhead for the row ID – this overhead specifies a row-forwarding address

Table 2-7 shows the maximum number of data rows that can fit on a DOL data page:

**Table 2-7: Maximum number of data rows for a DOL data page**

| Page size | Max. # of rows |
|---|---|
| 2K | 166 |
| 4K | 337 |

| Page size | Max. # of rows |
|---|---|
| 8K | 678 |
| 16K | 1361 |

APL data pages can have a maximum of 256 rows. Because each page requires a 1-byte row number specifier, large pages with short rows incur some unused space. For example, if Adaptive Server is configured with 8K logical pages and rows that are 25 bytes long, the page will have 1275 bytes of unused space, after accounting for the row-offset table, and the page header.

## Maximum number of arguments for stored procedures

The maximum number of arguments for stored procedures is 2048.

## Maximum length of expressions, variables, and arguments in stored procedures

The maximum size for expressions, variables, and arguments passed to stored procedures is 16384 (16K) bytes, for any page size. This can be either character or binary data. You can insert variables and literals up to this maximum size without using the **writetext** command.

### If your upgraded Adaptive Server used a lower maximum length

Earlier versions of Adaptive Server had a maximum size of 255 bytes for expressions, variables, and arguments for stored procedures. Any scripts or stored procedures that you wrote for previous versions of Adaptive Server that used this old maximum may now return larger string values because of the larger maximum page sizes.

Because of the larger value, Adaptive Server may truncate the string, or the string may cause overflow if it was stored in another variable or inserted into a column or string. If columns of existing tables are modified to increase the length of character columns, you must change any stored procedures that operate data on these columns to reflect this new length.

### Clients retrieving data with enhanced limits

Adaptive Server version 12.5 can store data that has different limits than data stored in previous versions. Clients also must be able to handle the new limits the data can use. If you are using older versions of Open Client and Open Server, they cannot process the data if you perform the following:

1. Upgrade to Adaptive Server version 12.5.
2. Drop and recreate the tables with wide columns
3. Insert wide data.

## Maximum number of expressions in a select statement

Adaptive Server version 12.5 has no explicit limit on the number of expressions in a **select** statement. The number is only limited by the available system memory.

## Number of logins

Table 2-8 lists the limits for the number of logins, users, and groups for Adaptive Server:

**Table 2-8: Limits for logins, users, and groups**

| Item | Version 12.0 | Version 12.5 limit | New range |
|---|---|---|---|
| Number of logins per server (SUID) | 64K | 2 billion plus 32K | -32768 to 2 billion |
| Number of users per database | 48K | 2 billion less 1048576 | -32768 to 16383; 1048576 to 2 billion |
| Number of groups per database | 16K | 1048576 | 16392 to 1048576 |

Figure 2-1 describes this range of values:

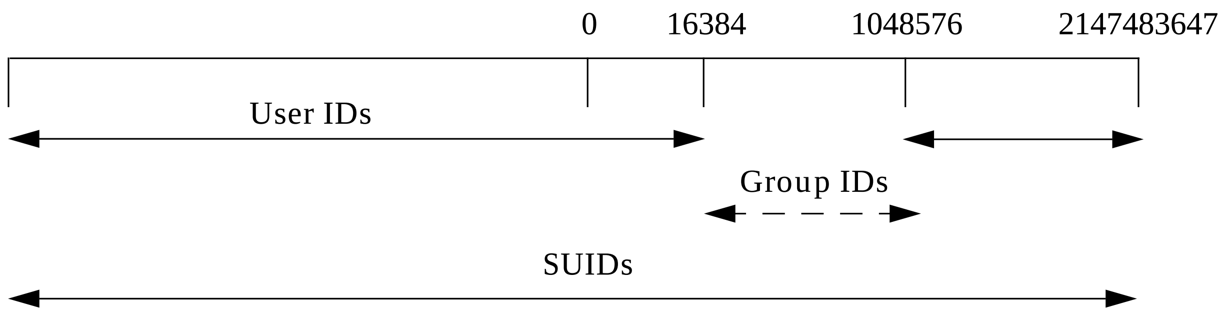

**Figure 2-1: Range for logins, users, and groups**

Although Adaptive Server can handle over 2 billion users connecting at one time, the actual number of users that can connect to Adaptive Server is limited by:

- The **number of user connections** configuration parameter.
- The number of file descriptors available from the operating system. Each user login uses one file descriptor per connection.

➤ ***Note***

Before Adaptive Server can have more than 64K logins and simultaneous connections, the operating system must be configured for more than 64K file descriptors. See your operating system documentation for information about increasing the number of file descriptors.

Table 2-9 lists the global variables for the server limits of logins, users, and groups:

**Table 2-9: Global variables for logins, users, and groups**

| Name of variable | What it displays | Value |
|---|---|---|
| *@@invaliduserid* | Invalid user ID | -1 |
| *@@minuserid* | Lowest user ID | -32768 |
| *@@guestuserid* | Guest user ID | 2 |
| *@@mingroupid* | Lowest group user ID | 16384 |
| *@@maxgroupid* | Highest group user ID | 1048576 |
| *@@maxuserid* | Highest user ID | 2147483647 |

**Table 2-9: Global variables for logins, users, and groups**

| Name of variable | What it displays | Value |
|---|---|---|
| *@@minsuid* | Lowest server user ID | -32768 |
| *@@probesuid* | Probe server user ID | 2 |
| *@@maxsuid* | Highest server user ID | 2147483647 |

To issue a global variable, enter:

```
select variable_name
```

For example:

```
select @@minuserid
```

## Summary of new limits for version Adaptive Server 12.5

Adaptive Server version 12.5 includes the following limits:

**Table 2-10: New limits for Adaptive Server version 12.5**

| Item | Allpages locked (APL) tables | | | | Data-only locked (DOL) tables | | | |
|---|---|---|---|---|---|---|---|---|
| | ---------------------------------Page Sizes ------------------------------------ | | | | | | | |
| | 2048 | 4096 | 8192 | 16384 | 2048 | 4096 | 8192 | 16384 |
| User-visible maximum row lengths | 1960 | 4008 | 8104 | 16296 | 1958 | 4006 | 8102 | 16294 |
| Maximum row lengths, including overheads | 1962 | 4010 | 8106 | 16298 | 1964 | 4012 | 8108 | 16300 |
| Fixed-length column size | 1960 | 4008 | 8104 | 16296 | 1958 | 4006 | 8102 | 16294 |
| Variable-length column size | 1948 | 3988 | 8158 | 16228 | 1954 | 4002 | 7998 | 16290 |
| User-visible size of index key | 600 | 1250 | 2600 | 5300 | 600 | 1250 | 2600 | 5300 |
| Internal size of index key | 650 | 1310 | 2670 | 5390 | 650 | 1310 | 2670 | 5390 |

Table 2-11 describes the new limits for version 12.5 that do not depend on the server’s page logical page size.

**Table 2-11: New limits for version 12.5 independent of page size**

| Item | Adaptive Server 12.0 limits | New limit |
|---|---|---|
| Number of logins per server | 65536 | 2147516415 |
| Number of users per database | 49152 | 2146484222 |
| Number of groups per database | 16384 | 1032193 |
| Number of columns per table | 250 | 1024 |
| Number of fixed length columns per table (for both APL and DOL tables) | 250 | 1024 |
| Number of variable-length columns per APL table | 250 | 254 |
| Number of variable-length columns per DOL table | 250 | 1024 |
| Number of arguments to stored procedures | 255 | 2048 |
| Number of expressions (columns) in a **select** statement | (undefined) | Depends on available system memory |
| Length of variables for *char* / *binary* datatypes | 255 | 16384 |
| Length of concatenated strings | 255 | 16384 |

## Unchanged limits in version 12.5

Table 2-12 describes some limits that have not changed for version 12.5.

**Table 2-12: Limits that have not changed for Adaptive Server version 12.5**

| Item | Adaptive Server version 12.0 limit |
|---|---|
| Number of key columns in an index key | 31 |
| Number of arguments for stored procedures | 255 |
| Maximum length of a database object | 30 |
| Number of items in an **order by** clause | 31 |
| Number of terms in a **group by** clause | 31 |

**Table 2-12: Limits that have not changed for Adaptive Server version 12.5**

| Item | Adaptive Server version 12.0 limit |
|---|---|
| Number of devices per server | 256 |
| Number of segments per database | 31 |
| Maximum engines per server | 128 |

## Changes to the create table command

**create table** for Adaptive Server version 12.5 includes the same functionality as previous versions of Adaptive Server, but is restricted to creating tables with the limits described in earlier sections (column and row length, number of columns per table, number of rows per table, and so on). The following comments apply to the 12.5 version of **create table**:

- Adaptive Server reports an error if the total size of all fixed length columns, plus the row overhead, is greater than table's locking scheme and page size allow. These limits are described in Table 2-2 and Table 2-3.
- If you create a DOL table with a variable-length column that exceeds a 8191 byte offset, you cannot add any rows to the column.
- In the following circumstances will create tables; however, you will receive errors about size limitations when you perform DML operations:
  - If the total row size for rows with variable-length columns exceeds the maximum column size.
  - If the length of a single variable-length column exceeds the maximum column size.
  - For DOL tables, if the offset of any variable-length column other than the initial column exceeds the limit of 8191 bytes.

## Changes to the *alter table* command

The following are changes to **alter table**:

- **alter table** raises an error if the number of variable-length columns in an APL table exceeds 254.

- The maximum value for the **max_rows_per_page** is 256 bytes for APL tables. **max_rows_per_page** parameter is not used for DOL tables.
- When converting a table to a different locking scheme, the data of the target table cannot violate the limits the source table requires. For example, if you attempt to convert a DOL with more than 254 variable length columns to an APL table, **alter table** fails because an APL table is restricted to having no more than 254 columns.

## Changes to *create index*

If you create columns with lengths greater than the index row-size limit, those columns cannot be indexed.

## Transact-SQL command updates

### set

You can update as many as 1024 columns in the **set** clause using literals, variables, or expressions returned from a subquery.

### select...for browse

- You cannot use the **select...for browse** option on tables containing more than 255 columns.

### compute

- If a **compute** clause includes a **group by** clause:
  - The **compute** clause cannot contain more than 255 aggregates
  - The **group by** clause cannot contain more than 255 columns.
- Columns included in a compute clause cannot be longer than 255 bytes.

### like

- The character string indicated by the **like** keyword cannot be longer than 255 bytes.

### declare cursor

- You can include as many as 1024 columns in an **update** clause of a client's **declare cursor** statement.

#### The + operator

- Returns result strings up to a length of 16384 bytes.

## Client and server compatibility for the new limits

This section describes issues for clients running against Adaptive Server version 12.5.

### **print** statements

If you substitute parameters in **print** statements, you can combine variables that include **print** statements.

### If you connect to Adaptive Server with earlier versions Sybase software

Earlier versions of Sybase software (Open Client, Open Server, Adaptive Server) could only send or receive data in packets 255 bytes long. However, Adaptive Server version 12.5 is able to send and receive data 16K long. If you connect to Adaptive Server version 12.5 with an earlier version of Open Client, Open Server, Adaptive Server, and so on, the data they receive will be truncated to 255 bytes.

## New functions

This section describes the new functions for Adaptive Server version 12.5.

## pagesize()

### Function

Returns the page size in bytes for the specified object

### Syntax

```
int=pagesize(object_name [, index_name])
   int=pagesize(dbid, object_id [, index_id])
```

### Arguments

*object_name* – The name of the object you are searching. If you do not specify a fully qualified object name, the current database is searched.

*index_name* – Indicates the name of the index used for the search.

*dbid* – The ID of the database specified by *object_name*.

*object_id* – The ID of the object indicated by *object_name*.

*index_id* – The ID of the index indicated by *index_name*.

### Example

The following checks the page size of *sysobjects*:

```
select pagesize(‘sysobjects’, ‘sysobjects’)
-----------
8192
```

### Comments

- If you do not indicate an index name of ID, the default is to use the data level of the table.
- If the specified object is not a table (for example, if the name of a view is provided), **pagesize()** returns zero.
- If the specified object does not exist, **pagesize()** returns NULL.
- Because the logical page size is a server-wide setting for Adaptive Server, all the objects you check using **pagesize()** report the same size.

### Permissions

Any user can execute **pagesize()**.

## lockscheme()

### Function

Returns the locking scheme of the specified object as a string.

### Syntax

```
varchar(11) = lockscheme(object_name)
   varchar(11) lockscheme(object_id, dbid)
```

### Arguments

*object_name* – The name of the object you are searching. If you do not specify a fully qualified object name, the current database is searched.

*dbid* – The ID of the database specified by *object_name*.

*object_id* – The ID of the object indicated by *object_name*.

### Example

The following displays the locking scheme for *sysobjects*:

```
select lockscheme(‘sysobjects’)
------------------------------
allpages
```

The following displays the locking scheme for the *authors* table (*object_id* = 32000114) in the *pubs2* database (*dbid* = 4):

```
select lockscheme(32000114, 4)
------------------------------
allpages
```

### Comments

- **lockscheme()** returns *varchar(11)* and allows NULLs.
- If the specified object is not a table, then **lockscheme()** returns the string “`not a table`.”

### Permissions

Any user can execute **lockscheme()**.

## New global variables

Adaptive Server version 12.5 includes the following global variables:

*@@maxpagesize* – returns the logical page size the server uses.

## New dbcc commands

The following section describes new dbcc commands.

### syntax

**dbcc serverlimits**

### Keywords and options

**serverlimits** – Displays the current limits for Adaptive Server about page, index, buffer, SQL, column, cache and row limits.

### Examples

**dbcc serverlimits**

```
Limits independent of page size:
================================

Server-wide, Database-specific limits

Max engines per server                          : 128
Max number of logins per server                 : 2147516416
Max number of users per database                : 2146484223
Max number of groups per database               : 1032193
Max number of user-defined roles per server     : 1024
Min database page size                          : 2048
Max database page size                          : 16384
Max length of a database-object name            : 30

Database page-specific limits

APL page header size                            : 32
DOL page header size                            : 44

Table, Index related limits

Max number of columns in a table/view           : 1024
Max number of indexes on a table                : 250
Max number of user-keys in a single index       : 31

Cache manager related limits

Default number of buffers in a named cache      : 256

General SQL related
```

```
Max number of arguments to stored procedures     : 2048
Max number of subqueries in a single statement   : 16
Max number of referential constraints per table  : 192

Limits as a function of the page size:
======================================

Item dependent on page size: 2048     4096     8192     16384
------------------------------------------------------------------------
Table-specific row-size limits

Max possible size of a log-record row on APL log page:
2014        4062     8158     16350
Physical Max size of an APL data row, incl row-overheads:
1962        4010     8106     16298
Physical Max size of a  DOL data row, incl row-overheads:
1964        4012     8108      16300
Max user-visible size of an APL data row:
1960     4008     8104     16296
Max user-visible size of a  DOL data row:1958 4006 8102 16294
Max user-visible size of a fixed-length column in an APL table:
1960     4008     8104     16296
Max user-visible size of a fixed-length column in a  DOL table:
1958     4006     8102     16294
Max user-visible size of a variable-length column in an APL table:
1948     3988     8068     16228
Max user-visible size of a variable-length column in a  DOL table:
1954     4002     8098     16290
Max number of rows per APL data page:256      256      256      256
Max number of rows per DOL data page:166      337      678      1361

Index-specific row-size limits

Max index row-size, including row-overheads 650 1300 2700 5400
Max user-visible index row-size:600      1250     2600     5300

OAM-manager related limits

Max number of OAM entries per OAM page:250      506      1018     2042

Text-manager related limits

Max text size available for user data: 1800     3600     7650     16200

Cache manager related limits

Min size of named cache (KB):512      1024     2048     4096
Default size of named cache (KB): 1024     2048     4096     8192

DBCC execution completed. If DBCC printed error messages, contact a user
with System Administrator (SA) role.
```

### Comments

**dbcc serverlimits** displays the information for all the logical page sizes to which Adaptive Server can be set, not just the logical page size it is currently set.

### Permissions

Anybody can run **dbcc serverlimits**.

# 3 Getting Started with Java

This chapter describes the Java runtime environment, how to enable Java on the server, and how to install Java classes in the database. You should read this chapter if you are installing either or both of these Adaptive Server version 12.5 beta features:

- SQLJ functions and stored procedures
- XML in the database
- Your installation must possess a valid Sybase Java site license to install and use these features.
- For more information about using Java and SQL together, refer to *Java in Adaptive Server Enterprise*.
- In this chapter, these topics are discussed:

| Name | Page |
|---|---|
| The Java Runtime Environment | page 3-45 |
| Enabling the Server for Java | page 3-47 |
| Creating Java Classes and JARs | page 3-47 |
| Installing Java Classes in the Database | page 3-48 |

## The Java Runtime Environment

The Adaptive Server runtime environment for Java requires a Java VM, which is available as part of the database server, and the Sybase runtime Java classes, or Java API. If you are running Java applications on the client, you may also require the Sybase JDBC driver, jConnect, on the client.

### Java Classes in the Database

You can use either of the following sources for Java classes:

- Sybase runtime Java classes
- User-defined classes

## Sybase Runtime Java Classes

To support Java in the database, Adaptive Server:

- Comes with its own Java VM, specifically developed for handling Java processing in the server.
- Uses its own JDBC driver that runs in the server and accesses the database.

The Sybase Java VM runs in the database environment. It interprets compiled Java instructions and runs them in the database server. The Sybase Java VM supports a subset of JDK version 1.1.8 (UNIX and Windows NT) classes and packages.

The Sybase runtime Java classes are the low-level classes installed to Java-enable a database. They are downloaded when Adaptive Server is installed and are available thereafter from *$SYBASE /$SYBASE_ASE/lib/runtime.zip* (UNIX) or *%SYBASE% \ %SYBASE_ASE% \ lib \ runtime.zip* (Windows NT). You do not need to set the CLASSPATH environment variable specifically for Java in Adaptive Server.

Sybase does not support runtime Java packages and classes that assume a screen display, deal with networking and remote communications, or handle security.

## User-Defined Java Classes

You install user-defined classes into the database using the **installjava** utility. Once installed, these classes are available from other classes in the database and from SQL as user-defined datatypes.

## JDBC Drivers

The Sybase internal JDBC driver that comes with Adaptive Server conforms to JDBC version 2.0.

If your system requires a JDBC driver on the client, you can use:

- jConnect version 4.2, which supports JDK version 1.x
- jConnect version 5.2, which supports JDK version 2.x

.

## Enabling the Server for Java

To enable the server and its databases for Java, enter this command from isql:

```
sp_configure “enable java”, 1
```

Then shutdown and reboot the server.

By default, Adaptive Server is not enabled for Java. You cannot install Java classes or perform any Java operations until the server is enabled for Java.

You can increase or decrease the amount of memory available for Java features in Adaptive Server to optimize performance using the sp_configure system procedure. Java configuration parameters are described in the *Adaptive Server System Administration Guide*.

### Disabling the Server for Java

To disable Java in the database, enter this command from isql:

```
sp_configure “enable java”, 0
```

## Creating Java Classes and JARs

The Sybase-supported classes from the JDK are installed on your system when you install Adaptive Server version 12.5 beta. This section describes the steps for creating and installing your own Java classes.

To make your Java classes (or classes from other sources) available for use in the server, follow these steps:

1. Write and save the Java code that defines the classes.
2. Compile the Java code.
3. Create Java archive (JAR) files to organize and contain your classes.
4. Install the JARs and classes in the database.

### Writing the Java Code

Use the Sun Java SDK or a development tool such as Sybase PowerJ to write the Java code for your class declarations. Save the Java code in a file with an extension of *.java*. The name and case of the file must be the same as that of the class.

➤ **Note**

Make certain that any Java API classes used by your classes are among the supported API classes.

### Compiling the Java Code

This step turns the class declaration containing Java code into a new, separate file containing byte code. The name of the new file is the same as the Java code file but has an extension of *.class*. You can run a compiled Java class in a Java runtime environment regardless of the platform on which it was compiled or the operating system on which it runs.

### Saving Classes in a JAR File

You can organize your Java classes by collecting related classes in packages and storing them in JAR files.

To install Java classes in a database, the classes or packages *must* first be saved in a JAR file, in uncompressed form. To create an uncompressed JAR file that contains Java classes, use the Java **jar cf0** command.

In this UNIX example, the **jar** command creates an uncompressed JAR file that contains all *.class* files in the **jcsPackage** directory:

```
jar cf0 jcsPackage.jar jcsPackage/*.class
```

Note that the “0” in **cf0** is “zero.”

JAR files allow you to install or remove related classes as a group.

## Installing Java Classes in the Database

To install Java classes from a client operating system file, use the **installjava** (UNIX) or **instjava** (Windows NT) utility from the command line.

Refer to *Adaptive Server Utilities Programs* for your platform for detailed information about these utilities. Both utilities perform the same tasks; for simplicity, this document uses UNIX examples.

## Using *installjava*

**installjava** copies a JAR file into the Adaptive Server system and makes the Java classes contained in the JAR available for use in the current database. The syntax is:

```
installjava
    -f file_name
    [-new | -update]
    [-j jar_name]
```

For example, to install classes in the *addr.jar* file, enter:

```
installjava -f "/home/usera/jars/addr.jar"
```

The **–f** parameter specifies an operating system file that contains a JAR. You must use the complete path name for the JAR.

This section describes retained JAR files (using **-j**) and updating installed JARs and classes (using **new** and **update**). For more information about these and the other options available with **installjava**, see the *Utility Programs* manual for your platform.

### Retaining the JAR File

When a JAR is installed in a database, the server disassembles the JAR, extracts the classes, and stores them separately. The JAR is not stored in the database unless you specify **installjava** with the **-j** parameter.

Use of **-j** determines whether the Adaptive Server system retains the JAR specified in **installjava** or uses the JAR only to extract the classes to be installed.

- If you do not specify the **-j** parameter, the Adaptive Server system does not retain any association of the classes with the JAR. This is the default option.
- If you do specify the **-j** parameter, Adaptive Server installs the classes contained in the JAR in the normal manner, and then retains the JAR and its association with the installed classes.

If you retain the JAR file:

- You can remove the JAR and all classes associated with it, all at once, with the **remove java** statement. Otherwise, you must remove each class or package of classes one at a time.
- Other systems may request that the class associated with a given Java column be downloaded with the column value. If a class

retains its association with the JAR, the Adaptive Server system can download the JAR, rather than individual classes.

## Referencing Other Java-SQL Classes

Installed classes can reference other classes in the same JAR file and classes previously installed in the same database, but they cannot references classes in other databases.

If the classes in a JAR file do reference undefined classes, an error may result:

- If an undefined class is referenced directly in SQL, it causes a syntax error for "undefined class."
- If an undefined class is referenced within a Java method that has been invoked, it throws a Java exception that may be caught in the invoked Java method or cause a general SQL exception.

The definition of a class can contain references to unsupported classes and methods as long as they are not actively referenced or invoked. Similarly, an installed class can contain a reference to a user-defined class that is not installed in the same database as long as the class is not instantiated or referenced.

# 4 Using Java Methods as SQL Functions and Stored Procedures

➤ ***Note***

To use Adaptive Server's Java capabilities, you must have an Adaptive Server Java site license

Adaptive Server 12.5.beta extends the Java capabilities provided with Adaptive Server 12.0. With Adaptive Server version 12.0, you can:

- Install and execute Java methods from both the client and the data server.
- Execute Java methods as built-in functions in the data server.
- Use Java classes as abstract datatypes (ADTs), so that you can store Java objects in the database and define columns as Java datatypes.

Adaptive Server version 12.5.beta provides additional ways to execute Java methods. With version 12.5.beta, you can enclose Java static methods in SQL wrappers and use them exactly as you would standard SQL stored procedures or user-defined functions. This new functionality:

- Allows Java methods to return values, output parameters, and result sets to the calling environment.
- Complies with Part 1 of the SQLJ standard specification.
- Allows you to use existing Java methods as SQLJ procedures and functions on the server, on the client, and on any SQLJ-compliant, third-party database.

By wrapping a Java method in a SQL name, you also take advantage of traditional SQL syntax and SQL metadata and permission capabilities.

➤ ***Note***

SQLJ stored procedures and functions extend the Java-SQL features and capabilities offered in Adaptive Server 12.0. They do not replace or change those features.

Java methods that do not return result sets or output parameters can be invoked either as Java-SQL methods (described in *Java in Adaptive Server Enterprise*) or as SQLJ routines (described in this chapter).

***To create a SQLJ stored procedure or function:***

Before you can use Adaptive Server SQLJ capabilities, you must make sure that Java has been enabled on your server. Refer to "Getting Started in Java" for instructions for enabling Java on Adaptive Server.

You perform these steps to create and execute a SQLJ stored procedure or function.

1. Create and compile the Java method. Install the method class in the database using installjava.

   Refer to "Getting Started in Java" for information on creating, compiling, and installing Java methods in Adaptive Server.

2. Using create procedure or create function, define a SQL name for the method.

3. Execute the procedure or function. The examples in this chapter use JDBC method calls. You can also call the method using Embedded SQL or ODBC.

When executing a SQLJ procedure or function call, an uncaught Java exception raises a SQL exception and displays the error message:

```
Java method terminated with exception
```

## Compliance with SQLJ specifications

Compliance with SQLJ standards ensures that Sybase SQLJ functionality ports to all third-party, standards-compliant relational databases.

Adaptive Server SQLJ stored procedures and functions comply with SQLJ Part 1 of the standard specifications for using Java with SQL. These specifications were developed by a consortium of SQL vendors and have been accepted as an official ANSI standard. They can be found on the Web at http://www.ansi.org. In this document, SQLJ refers to capabilities compliant with SQLJ Part 1 of the standard specifications.

- SQLJ Part 0: "Database Language SQL - Part 10: Object Language Bindings (SQL/OLB)," ANSI X3.135.10-1998.

➤ ***Note***

SQLJ Part 0 is Part 10 of the SQL Standard.

Specifications for embedding SQL statements in Java methods.

- SQLJ Part 1: "SQLJ—Part 1: SQL Routines using the Java Programming Language," ANSI NCITS N331.1.

  Specifications for calling Java static methods as SQL stored procedures and user-defined functions.

- SQLJ Part 2: "SQLJ—Part 2: SQL Types using the Java Programming Language," ANSI NCITS N331.2.

  Specifications for using Java classes as SQL datatypes.

Adaptive Server version 12.5.beta supports most features described in the SQLJ Part 1 specification; however, there are some differences. Unsupported features are listed in Table 4-3 on page 78 and partially supported features are listed in Table 4-4 on page 79. Sybase-defined features—those not defined by the standard but left to the implementation—are listed in Table 4-5 on page 79.

In those instances where Sybase proprietary implementation differs from the SQLJ specifications, Sybase supports the SQLJ standard. For example, Sybase SQL stored procedures support two parameter modes: in and inout. The SQLJ standard supports three parameter modes: in, out, and inout. The Sybase syntax for creating SQLJ stored procedures also supports all three parameter modes. See the syntax description for "create function" on page 82 for more information about parameter modes.

## General issues

This section describes general issues and constraints that apply to SQLJ functions and stored procedures

- Only static (class) methods can be referenced in a SQLJ function or stored procedure.
- Sybase recommends that you do not use static variables in methods referenced by SQLJ functions or stored procedures. The values returned for these variables may be unreliable.
- During the execution of a SQLJ routine, data is passed from SQL to Java and back to SQL. The data conversions must follow the rules for datatype mapping outlined in the JDBC standard and in *Java in Adaptive Server Enterprise*.

Refer to "Mapping Java and SQL datatypes" on page 73 for a discussion of datatype mapping and conversions.

- If a method referenced by a SQLJ function or stored procedure invokes SQL, returns parameters from the SQL system to the calling environment, or returns result sets from SQL to the calling environment, you must use a JDBC interface that enables object-oriented access to the relational database.

## Examples

The examples used in this chapter assume a SQL table called sales_emps with these columns:

- name – the employee's name
- id – the employee's identification number
- state – the state in which the employee is located
- sales – amount of employee's sales
- jobcode – the employee's job code

The table definition is:

```
create table sales_emps
       (name varchar(50), id char(5),
       state char(20), sales decimal (6,2),
       jobcode integer null)
```

The example Java classes and methods are:

- Routines1.region() – maps a U.S. state code to a region number. The method does not use SQL.
- Routines1.correctStates() – performs a SQL update command to correct the spelling of state codes. Old and new spellings are specified by input parameters.
- Routines2.bestTwoEmps() – determines the top two employees by their sales records and returns those values as output parameters.
- Routines3.orderedEmps() – creates a SQL result set consisting of selected employee rows ordered by values in the sales column, and returns the result set to the client.
- Routines5.job2() – Returns a string value corresponding to an integer job code value.

## SQLJ user-defined functions

The create function command specifies a SQLJ function name and signature for a Java method. You can use SQLJ functions to read and modify SQL and to return a value described by the referenced method.

The SQLJ syntax for create function is:

```
create function [owner].sql_function_name
           ([sql_parameter_name sql_datatype
               [( length)| (precision[, scale])]
           [, sql_parameter_name sql_datatype
               [( length )      |
               ( precision[, scale]) ...]])
       returns sql_datatype
           [( length)| (precision[, scale])]
       [modifies sql data]
       [returns null on null input |
           called on null input]
       [deterministic | not deterministic]
       [exportable]
       language java
       parameter style java
       external name 'java_method_name
           [([java_datatype[, java_datatype ...]])]'
```

When creating a SQLJ function:

- The SQL function signature is the name (*sql_parameter_name*) and SQL datatype (*sql_datatype*) of each function parameter.
- When creating a SQLJ function, you must include the parentheses that surround function parameter information—even if you do not include that information.

  For example:

```
create function blue ()
           language java
           parameter style java
       external name ‘javamethod’
```

- You can include the deterministic or not deterministic keywords, but Adaptive Server 12.5.beta does not use them. They are included for syntactic compatibility with the SQLJ Part 1 standard.
- The clauses returns null on null input and called on null input specify how Adaptive Server handles null arguments of a function call. returns null on null input specifies that if the value of any argument is null at runtime, the return value of the function is set to null and the

function body is not invoked. called on null input, the default, specifies that the function is invoked regardless of null argument values.

Function calls and null argument values are described in detail in "Handling nulls in the function call" on page 60.

- The modifies sql data clause specifies that the method invokes SQL operations and reads and modifies SQL data. This is the default value. You do not need to include it except for syntactic compatibility with the SQLJ Part 1 standard.
- The exportable keyword specifies that the function is to run on a remote server using Omni capabilities. Both the function and the method on which it is based must be installed on the remote server.
- The clauses language java and parameter style java specify that the referenced method is written in Java and that the parameters are Java parameters. You *must* include these phrases when creating a SQLJ function.
- The external name clause specifies that the routine is not written in SQL and identifies the Java method, class and, package name (if any).
- The Java method signature specifies the Java datatype (*java_datatype*) of each method parameter (*java_parameter_name*). The Java method signature is optional, but if one is not specified, Adaptive Server infers one from the SQL function signature.

  See "Mapping Java and SQL datatypes" on page 73 for more information.

### Writing the Java method

Before you can create a SQLJ function, you must write the Java method that references it, compile the method class, and install it in the database.

In this example, the Routines1.region maps a state code to a region number and returns that number to the user. The entire Routines1 class is shown below; the Routines1.correctStates() method is referenced in the SQLJ procedure in "Modifying SQL data" on page 63.

```
import java.lang.*;
import java.sql.*;


static String _url = "jdbc:default:connection";
```

```
public class Routines1 {

     public static int region(String s)
              throws SQLException {
          s = s.trim();
          if (s.equals(“MN”) || s.equals(“VT”) ||
              s.equals(“NH”) ) return 1;
          if (s.equals(“FL”) || s.equals(“GA”) ||
              s.equals(“AL”) ) return 2;
          if (s.equals(“CA”) || s.equals(“AZ”) ||
              s.equals(“NV”) ) return 3;
          else throw new SQLException
              (“Invalid state code”, “X2001”);
     }
     public static void correctStates
          (String oldSpelling, String newSpelling)
          throws SQLException {

          Connection conn = null;
          PreparedStatement pstmt = null;
          try {
              Class.forName
                  (”sybase.asejdbc.ASEDriver”);
              conn = DriverManager.getConnection(_url);
          }
          catch (Exception e) {
              System.err.println(e.getMessage() +
                  “:error in connection”);
          }
          try {
              pstmt = conn.prepareStatement
                  (“UPDATE sales_emps SET state = ?
                  WHERE state = ?”);
              pstmt.set.String(1, newSpelling);
              pstmt.set.String(2, oldSpelling);
              pstmt.executeUpdate();
          }
          catch (SQLException e) {
              System.err.println(“SQLException: “ +
              e.getErrorCode() + e.getMessage());
          }
     return;
     }
}
```

#### *Creating the SQLJ function*

After writing and installing the method, you can create the SQLJ function. For example:

```
create function region_of(state char(20))
        returns integer
language java parameter style java
external name 'Routines1.region'
```

The SQLJ create function statement specifies an input parameter `(state char(20))`and an integer return value. The SQL function signature is “state char(20).” Because there is no explicit Java method signature, the system infers a Java method signature for the method from the SQL function signature.

SQLJ routines rely on datatype correspondance between the parameters of the SQLJ routine and the referenced Java method. When the Java method signature is not present, the system infers one according to the correspondance rules in Table 4-1 on page 74. In this example, Adaptive Server infers a Java method signature of java.lang.String for the Routines1.region method.

#### *Calling the function*

You can call the SQLJ function as if it were a built-in function. For example:

```
select name, dbo.region_of(state) as region
    from sales_emps
where dbo.region_of(state)=3
```

➤ ***Note***

When calling a SQLJ function, you must include the function owner as well as the function name. Because only the Database Owner can create SQLJ functions, each function invocation has the form “dbo.*function_name*.”

## Handling null argument values

Java class datatypes and Java primitive datatypes handle null argument values in different ways.

- Java datatypes that are classes—such as java.lang.Integer, java.lang.String, java.lang.byte[], and java.sql.Timestamp—can hold both actual values and null reference values.

- Java primitive datatypes—such as boolean, byte, short, and int—have no representation for a null value. They can hold only non-null values.

When a Java method is invoked that causes a SQL null value to be passed as an argument to a Java parameter whose datatype is a Java class, it is passed as a Java null reference value.When a SQL null value is passed as an argument to a Java parameter of a Java primitive datatype, however, an exception is raised as the Java primitive datatype has no representation for a null value.

Typically, you will write Java methods that specify Java parameter datatypes that are classes and nulls are handled without raising an exception. If you choose to write Java functions that use Java parameters that cannot handle null values, you can either:

- Include the returns null on null input clause when you create the SQLJ function, or
- Invoke the SQLJ function using a case or other conditional expression to test for null values and call the SQLJ function only for the non-null values.

You can handle expected nulls when you create the SQLJ function or when you call it. The following sections describe both scenarios, and reference this method:

```
import java.lang.*;
import java.sql.*;


public class Routines5 {
     public static String job2(int jc)
              throws SQLException {
     if (jc==1) return “Admin”;

     else if (jc==2) return “Sales”;
     else if (jc==3) return “Clerk”;
     else return “unknown jobcode”;
     }
}
```

### Handling nulls when creating the function

If null values are expected, you can include the returns null on null input clause when you create the function. For example:

```
create function job_of22(jc integer)
    returns varchar(20)
returns null on null input
language java parameter style java
external name 'Routines5.job2'
```

You can then call job_of22 in this way:

```
select name, dbo.job_of22(jobcode)
    from sales_emps
where dbo.job_of22(jobcode) <> "Admin"
```

When the SQL system evaluates the call job_of22(jobcode) for a row of sales_emps in which the jobcode column is null, the value of the call is set to null without actually calling the Java method Routines5.job2. For rows with non-null values of the jobcode column, the call is performed normally.

Thus, when a SQLJ function created using the returns null on null input clause encounters a null argument, the result of the function call is set to null and the function is not invoked.

➤ ***Note***

If you include the returns null on null input clause when creating a SQLJ function, the returns null on null input clause applies to *all* function parameters, including nullable parameters.

If you include the called on null input clause (the default), null arguments for non-nullable parameters will generate an exception.

### Handling nulls in the function call

You can use a conditional function call to handle null values for non-nullable parameters. The following example uses a case expression:

```
select name,
    case when jobcode is not null
        then dbo.job_of2(jobcode)
        else null end
from sales_emps where
    case when jobcode is not null
        then dbo.job_of2(jobcode)
        else null end <> "Admin"
```

In this example, we assume that the function job_of2 was created using the default clause called on null input.

### Deleting a SQLJ function name

You can delete the SQLJ function name for a Java method using the drop function command. For example, enter:

```
drop function region_of
```

which deletes the region_of function name and its reference to the Routines1.region method. drop function does not affect the referenced Java method or class.

See "drop function" on page 84 for syntax and usage information.

## SQLJ stored procedures

Using Java-SQL capabilities, you can install Java classes in the database and then invoke those methods from a client or from within the SQL system. Adaptive Server version 12.5.beta lets you invoke Java static (class) methods in another way—as SQLJ stored procedures.

SQLJ stored procedures:

- Can return result sets and/or output parameters to the client
- Behave exactly as standard SQL stored procedures when executed
- Can be called from the client using ODBC, isql, or JDBC
- Can be called within the server from other stored procedures or native Adaptive Server JDBC

The end-user need not know whether the procedure being called is a SQLJ stored procedure or a Sybase Transact-SQL stored procedure. They are invoked in the same way.

The SQLJ syntax for create procedure is:

```
create procedure [owner.]sql_procedure_name
       ([[ in | out | inout ] sql_parameter_name
           sql_datatype [( length) |
           (precision[, scale])]
       [, in | out | inout ] sql_parameter_name
           sql_datatype [( length) |
           (precision[, scale]) ...])
       [modifies sql data ]
       [dynamic result sets integer]
       [deterministic | not deterministic]
       language java
       parameter style java
       external name 'java_method_name
           [([java_datatype[, java_datatype ...]])]'
```

➤ ***Note***

To comply with the SQLJ Part 1 standard, the SQLJ create procedure command syntax is different from syntax used to create Sybase Transact-SQL stored procedures.

Refer to "create procedure" on page 85 for a detailed description of each keyword and option in this command.

When creating SQLJ stored procedures:

- The SQL procedure signature is the name (*sql_parameter_name*) and SQL datatype (*sql_datatype*) of each procedure parameter.
- When creating a SQLJ stored procedure, you must include the parentheses that surround the SQL procedure signature—even if you do not include that information.

  For example:

  ```
  create procedure red ()
          language java
          parameter style java
      external name 'javamethod1'
  ```

- You must include the dynamic result sets *integer* option when result sets are to be returned to the calling environment. Use the *integer* variable to specify the maximum number of result sets expected.
- You can include the keywords deterministic or not deterministic for compatibility with the SQLJ standard. However, Adaptive Server 12.5.beta does not make use of this option.
- You must include the language java parameter style java keywords, which tell Adaptive Server that the external routine is written in Java

and the runtime conventions for arguments passed to the external routine are Java conventions.

- You can include the keywords modifies sql data to indicate that the method invokes SQL operations and reads and modifies SQL data. In Adaptive Server 12.5.beta, this is the default value.
- You can define different SQL names for the same Java method using create procedure and then use them in the same way.
- The external name clause indicates that the external routine is written in Java and identifies the Java method, class, and package name (if any).
- The Java method signature specifies the Java datatype (*java_datatype*) of each method parameter (*java_parameter_name*). The Java method signature is optional, but if one is not specified, Adaptive Server infers one from the SQL procedure signature.

  See "Mapping Java and SQL datatypes" on page 73 for detailed information.

## Modifying SQL data

You can use a SQLJ stored procedure to modify information in the database. The method referenced by the SQLJ procedure must be either:

- A method of type void, or
- A method with an int return type (the int return type is a Sybase extension of the SQLJ standard)

By enclosing the method in a SQL wrapper, you can then call the method as if it were a SQL stored procedure.

### Writing the Java method

The method Routines1.correctStates performs a SQL update statement to correct the spelling of state codes. Input parameters specify the old and new spellings. correctStates is a void method; no value is returned to the caller.

➤ ***Note***

Only the correctStates method is shown below. See "SQLJ user-defined functions" on page 55 for the complete text of the Routines1 class.

```
public static void correctStates(String oldSpelling, String
newSpelling) throws SQLException {

     Connection conn = null;
     PreparedStatement pstmt = null;
     try {
          Class.forName(”sybase.asejdbc.ASEDriver”);
          conn = DriverManager.getConnection(_url);
     }
     catch (Exception e) {
          System.err.println(e.getMessage() +
              “:error in connection”);
     }
     try {
          pstmt = conn.prepareStatement
              (“UPDATE sales_emps SET state = ?
              WHERE state = ?”);
          pstmt.set.String(1, newSpelling);
          pstmt.set.String(2, oldSpelling);
          pstmt.executeUpdate();
     }
     catch (SQLException e) {
          System.err.println(“SQLException: “ +
          e.getErrorCode() + e.getMessage());
     }
     return;
}
```

### Creating the stored procedure

Before you can call a Java method with a SQL name, you must create the SQL name for it using the SQLJ create procedure command. The modifies sql data clause is optional.

```
create procedure correct_states(old char(20),
        not_old char(20))
modifies sql data
language java parameter style java
external name 'Routines1.correctStates'
```

The correct states procedure has a SQL procedure signature of “old char(20), not_old char(20).” Because there is no Java method signature specified, the system infers a method signature of String and String from the SQL procedure signature. Note that the datatypes of the input parameters in the correctStates method are indeed String.

### Calling the stored procedure

You can execute the SQLJ procedure exactly as you would a Transact-SQL procedure. In this example, the procedure executes from isql:

```
execute correct_states 'GEO', 'GA'
```

## Using input and output parameters

Java methods do not support output parameters. When you wrap a Java method in SQL, however, you can take advantage of Sybase SQLJ capabilities that allow input, output, and input/output parameters.

When you create a SQLJ procedure, you identify the mode for each parameter as in, out, or **inout**.

- For input parameters, use the in keyword to qualify the parameter. in is the default; Adaptive Server assumes an input parameter if you do not enter a parameter mode name.
- For output parameters, use the out keyword.
- For parameters that can pass values both to and from the referenced Java method, use the inout keyword.

➤ ***Note***

You create Sybase Transact-SQL stored procedures using only the in and out keywords. The out keyword corresponds to the SQLJ inout keyword. See the create procedure reference pages in the *Adaptive Server Reference Manual* for more information.

- To create a SQLJ stored procedure that defines output parameters, you must:
- Define the output parameter(s) using either the out or inout option when you create the SQLJ stored procedure.
- Declare those parameters as Java arrays in the Java method. SQLJ uses arrays as containers for the method's output parameter values.

  For example, if you want an Integer parameter to return a value to the caller, you must specify the parameter type as Integer[] (an array of Integer) in the method.

  The array for an out or inout parameter is created implicitly by the system. It has a single element. The input value (if any) is placed in the first (and only) element of the array before the Java method is called. When the Java method returns, the first element is removed and assigned to the output variable. Typically, this element will be assigned a new value by the called method.

The following examples illustrate the use of output parameters using a Java method bestTwoEmps() and a stored procedure best2 that references that method.

### Writing the Java method

The Routines2.bestTwoEmps() method returns the name, ID, region, and sales of the two employees with the highest sales performance records. The first eight parameters are output parameters requiring a containing array. The ninth parameter is an input parameter and does not require an array.

◆ *WARNING!*

**This method is in development; it may not run on your computer.**

```
import java.lang.*;
import java.math.*;
import java.sql.*;

public class Routines2 {
     public static void bestTwoEmps(String[] n1,
                  String[] id1, int[] r1,
                  BigDecimal[] s1, String[] n2,
                  Sting[] id2, int[] r2, BigDecimal[] s2,
                  int regionParm) throws SQLException {


     n1[0] = “****”; n2[0] = “****”;
     id1[0] = ““; id2[0] = ““:
     r1[0] = 0; r2[0] = 0;
     s1[0] = new BigDecimal(0);
     s1[0] = new BigDecimal(0);
```

```
        try {
             Connection conn = DriverManager.getConnection
                 (“jdbc:default:connection”);
             java.sql.PreparedStatement stmt =
                 conn.prepareStatement(SELECT name, id,
                 region_of(state) as region, sales FROM
                 sales_emps WHERE region_of(state)>? AND
                 sales IS NOT NULL ORDER BY sales DESC”);
             stmt.setIngeger(1, regionParm)
                 ResultSet r = stmt.executeQuery();

             if(r.next()) {
                 n1[0] = r.getString(“name”);
                 id1[0] = r.getString(“id”);
                 r1[0] = r.getInt(“region”);

                 s1[0] = R.getBigDecimal(“sales”, 2);
             }
             else return;

             if(r.next()) {
                 n2[0] = r.getString(“name”);
                 id2[0] = r.getString(“id”);
                 r2[0] = r.getInt(“region”);
                 s2[0] = r.getBigDecimal(“sales”, 2);
             }
             else return;
        }
        catch (SQLException e) {
             System.err.println(“SQLException: “ +
                 e.getErrorCode() + e.getMessage());
        }
}
```

### Creating the SQLJ procedure

Create a SQL name for the bestTwoEmps method. The first eight parameters are output parameters; the ninth is an input parameter.

```
create procedure best2
    (out n1 varchar(50), out id1 varchar(5),
    out s1 decimal(6,2), out n2 varchar(50),
    out id2 varchar(50), out r2 integer,
    out s2 decimal(6,2), in region integer)
    language java parameter style java
    external name 'Routines2.bestTwoEmps'
```

The SQL procedure signature for best2 is: varchar(20), varchar(5), decimal (6,2) and so on. No Java method signature is specified. Thus, the system infers a Java method signature of java.lang.String, java.lang.String, java.math.BigDecimal and so on in accordance with the JDBC standard datatype mapping shown in Table 4-1 on page 74. However, because best2 specifies output parameters, the equivalent Java method parameter must specify an array of type java.lang.String[], java.lang.String[], java.math.BigDecimal[] and so on. Otherwise, the method will fail.

The SQL system:

1. Creates the needed arrays for the out and inout parameters when the SQLJ procedure is called.
2. Copies the contents of the parameter arrays into the out and inout target variables when returning from the SQLJ procedure.

### Calling the procedure

After the method is installed in the database and the SQLJ procedure referencing the method has been created, you can call the SQLJ procedure.

The following example uses JDBC calls. You can also use Embedded SQL, isql, or ODBC.

◆ ***WARNING!***

**This JDBC procedure call example is in development; it may not run on your computer.**

```
java.sql.CallableStatement stmt =
     conn.prepareCall(“{call
     best2(?,?,?,?,?,?,?,?,?)}”);
stmt.registerOutParameter(1,java.sql.Types.STRING);
stmt.registerOutParameter(1,java.sql.Types.STRING);
stmt.registerOutParameter(1,java.sql.Types.
     INTEGER);
stmt.registerOutParameter(1,java.sql.Types.
     DECIMAL);
stmt.registerOutParameter(1,java.sql.Types.STRING);
stmt.registerOutParameter(1,java.sql.Types.STRING);
stmt.registerOutParameter(1,java.sql.Types.
     INTEGER);
stmt.registerOutParameter(1,java.sql.Types.
     DECIMAL);
stmt.setInt(9,3);
stmt.executeUpdate();
String n1 = stmt.getString(1);
String n2 = stmt.getString(2);
int r1 = stmt.getInt(3);
BigDecimal s1 = stmt.getBigDecimal(4);
String n2 = stmt.getString(5);
String id2 = stmt.getString(6);
int r2 = stmt.getInt(7);
BigDecimal s2 = stmt.getBigDecimal(8);
```

## Returning result sets

A SQL result set is a sequence of SQL rows that is delivered to the calling environment.

When a Transact-SQL stored procedure returns one or more results sets, those result sets are implicit output from the procedure call. That is, they are not declared as explicit parameters or return values.

Java methods can return Java result set objects, but they do so as explicitly declared method values.

To return a SQL-style result set from a Java method, you must first wrap the Java method in a SQLJ stored procedure. When you call the method as a SQLJ stored procedure, the result sets, which are returned by the Java method as Java result set objects, can then be processed by the SQL caller as if they were normal SQL-style result sets.

When writing the Java method to be invoked as a SQLJ procedure that returns a SQL-style result set, you must specify an additional parameter to the method for each result set that the method can return. Each such parameter is a single-element array of the Java ResultSet class.

This section describes the basic process of writing a method, creating the SQLJ stored procedure, and calling the method. See "Returning result sets" on page 69 for detailed information about returning result sets.

### Writing the Java method

The following method, Routines3.orderedEmps, invokes SQL, includes a ResultSet parameter, and uses JDBC calls for securing a connection and opening a statement.

```
import java.lang.*;

import java.sql.*;


public class Routines3 {

     public static void orderedEmps
              (int regionParm, ResultSet[] rs) throws
              SQLException {


          Connection conn = null;
          PreparedStatement pstmt = null;

          try {
              Class.forName
                  ("sybase.asejdbc.ASEDriver");
              conn =
                  DriverManager.getConnection(_url);
          }
          catch (Exception e) {
              System.err.println(e.getMessage() +
                  ":error in connection");
          }

          try {
              java.sql.PreparedStatement
                  stmt = conn.prepareStatement
                  ("SELECT name, dbo.region_of(state) as
                  region, sales FROM sales_emps WHERE
                  dbo.region_of(state) > ? AND
                  sales IS NOT NULL
                  ORDER BY sales DESC");
              stmt.setInt(1, regionParm);
```

```
                rs[0] = stmt.executeQuery();
                return;
            }
            catch (SQLException e) {
                System.err.println("SQLException:
                " + e.getErrorCode() + e.getMessage());
            }
            return;
        }
    }
```

orderedEmps returns a single result set. You can also write methods that return multiple result sets. For each result set returned, you must:

- Include a separate ResultSet array parameter in the method signature.
- Create a Statement object for each result set.
- Assign each result set to the first element of its ResultSet array.

Adaptive Server makes sure that only one ResultSet object is opened for each Statement object. When creating Java methods that return result sets:

- Create a Statement object for each result set that is to be returned to the client.
- Do not explicitly close ResultSet and Statement objects. Adaptive Server closes them automatically.

➤ ***Note***

Adaptive Server ensures that ResultSet and Statement objects are not closed by garbage collection unless and until the affected result sets have been processed and returned to the client.

- If some rows of the result set are fetched by calls of the Java next() method, only the remaining rows of the result set are returned to the client.

### Creating the SQLJ stored procedure

When you create a SQLJ stored procedure that returns result sets, you must specify the maximum number of result sets that can be returned. In this example, the ranked_emps procedure returns a single result set.

```
create procedure ranked_emps(region integer)
```

```
dynamic result sets 1
language java parameter style java
external name 'Routines3.orderedEmps'
```

If ranked_emps generates more result sets than are specified by create procedure, a warning displays and the procedure returns only the number of result sets specified.

When you include the dynamic result sets clause with an integer value > 0, and you do not specify the method signature, Adaptive Server infers a method signature that includes a trailing java.sql.ResultSet parameter. In this instance, the system infers a method signature of int and java.sql.ResultSet[].

➤ ***Note***

Some restrictions apply to method overloading when you infer a method signature involving result sets. See "Mapping Java and SQL datatypes" on page 73 for a more information.

## Calling the procedure

After you have installed the method's class in the database and created the SQLJ stored procedure that references the method, you can call the procedure. You can write the call using any mechanism that processes SQL result sets.

For example, to call the ranked_emps procedure using JDBC, enter the following:

```
java.sql.CallableStatement stmt =
    conn.prepareCall("{call ranked_emps(?)}");
stmt.setInt(1,3);
ResultSet rs = stmt.executeQuery();
while (rs.next()) {
    String name = rs.getString(1);
    int.region = rs.getInt(2);
    BigDecimal sales = rs.get.BigDecimal(3);
    System.out.print("Name = " + name);
    System.out.print("Region = " + region);
    System.out.print("Sales = " + sales);
    System.out.printIn();
```

The ranked_emps procedure supplies only the parameter declared in the create procedure statement. The SQL system supplies an empty array of ResultSet parameter and calls the Java method, which assigns the output result set to the array parameter. When the Java method completes, the SQL system returns the result set in the output array element as a SQL result set.

### Deleting a SQLJ stored procedure name

You can delete the SQLJ stored procedure name for a Java method using the drop procedure command. For example, enter:

```
drop procedure correct_states
```

which deletes the correct_states procedure name and its reference to the Routines1.correctStates method. drop procedure does not affect the Java class and method referenced by the procedure.

## Advanced topics

The following topics present a detailed description of SQLJ topics.

### Mapping Java and SQL datatypes

When you create a stored procedure or function that references a Java method, you must make sure that SQL and Java parameters are in datatype correspondence. This means that the datatypes of input and output parameters or result sets cannot conflict when values are converted from the SQL environment to the Java environment and back again. SQLJ relies on the mapping of corresponding SQL and Java datatypes. The rules for how this mapping takes place are shown in Table 4-1 on page 74.

Each SQL parameter and its corresponding Java parameter must be mappable. SQL and Java datatypes are mappable in these ways:

- A SQL datatype and a primitive Java datatype are *simply mappable* if so specified in Table 4-1.
- A SQL datatype and a non-primitive Java datatype are *object mappable* if so specified in Table 4-1.
- A SQL abstract datatype (ADT) and a non-primitive Java datatype are *ADT mappable* if both are the same class or interface.
- A SQL datatype and a Java datatype are *output mappable* if the Java datatype is an array and the SQL datatype is simply mappable, object mappable, or ADT mappable to the Java datatype. For example, character and String[] are output mappable.

- A Java datatype is *result-set mappable* if it is an array of the result set-oriented class: java.sql.resultSet.
- A Java method is mappable to SQL if each of its parameters is mappable to SQL and its result set parameters are result-set mappable and the return type is either mappable (functions) or void or int (procedures).

Support for int return types is a Sybase extension of the SQLJ Part 1 standard.

**Table 4-1: Simply and object mappable SQL and Java datatypes**

| SQL datatype | Corresponding Java datatypes | |
|---|---|---|
| | **Simply mappable** | **Object mappable** |
| character | | String |
| nchar | | String |
| varchar | | String |
| nvarchar | | String |
| text | | String |
| numeric | | java.math.BigDecimal |
| decimal | | java.math.BigDecimal |
| money | | java.math.BigDecimal |
| smallmoney | | java.math.BigDecimal |
| bit | boolean | Boolean |
| tinyint | byte | Integer |
| smallint | short | Integer |
| integer | int | Integer |
| real | float | Float |
| float | double | Double |
| double precision | double | Double |
| binary | | byte[] |
| varbinary | | byte[] |
| datetime | | java.sql.Timestamp |
| smalldatetime | | java.sql.Timestamp |

## Implicit or explicit Java method signature

When you create a SQLJ function or stored procedure, you specify a SQL signature and, either implicitly or explicitly, a Java method signature. For example, in the create procedure statement for the correctStates stored procedure the SQL stored procedure signature is “old char(20), not_old char(20)”:

```
create procedure correct_states(old char(20),
        not_old char(20))
modifies sql data
language java parameter style java
external name ‘Routines1.correctStates’
```

The Java method signature is not explicitly specified. If it were, it would follow the method name in the create procedure statement. Adaptive Server infers the Java method signature from the SQL stored procedure signature. Table 4-1 tells us that the corresponding Java datatype for character is String. Thus the inferred Java method signature is String, String and the inferred external name clause is:

```
...
external name 'Routines1.correctStates
    (String, String)'
```

Adaptive Server looks for an existing method according to normal Java overloading conventions.

- You can either explicitly specify a Java method signature or allow Adaptive Server to infer the method signature for datatypes that are:
- Simply mappable
- ADT mappable
- Output mappable
- Result-set mappable

Typically, you will let Adaptive Server infer the Java method signature according to Table 4-1 on page 74, but you *must* explicitly specify the Java method signature for datatypes that are object mappable. Otherwise, Adaptive Server infers the primitive, simply mappable datatype.

For example, the Routines5.job2 method contains a parameter of type int. (See “Handling null argument values” on page 58.) When creating a function referencing that method, Adaptive Server will infer a Java signature of int, and you need not specify it.

However, suppose the parameter of Routines5.job2 were Java Integer, which is the object-mappable type. For example:

```
public class Routines5 {
    public static String job22(Integer jc)
            throws SQLException ...
```

Then, you would need to specify the Java method signature when you create a function that references it:

```
create function job_of22(jc integer)
...
external name 'Routines5.job2(java.lang.Integer)'
```

You can also specify an empty Java method signature. For example:

```
create function job_of22(jc integer)
...
external name 'Routines5.job2()'
```

In this case, Adaptive Server does not infer a method signature from the SQL function signature. Rather, Adaptive Server looks for an existing method Routines5.job2 with no parameters.

### Ensuring signature validity

If an installed class has been modified, Adaptive Server checks to make sure that the method signature is still valid when you invoke a SQLJ procedure or function that references that class. If you have deleted a class from the database or reinstalled another class in its place, with an invalid method signature, the SQLJ routine will fail.

## Using the command *main* method

In a Java client, you typically begin Java applications by running the Java Virtual Machine (VM) on the command main method of a class. The JDBCExamples class, for example, contains a main method. When you execute the class from the command line as in the following:

```
java JDBCExamples
```

it is the command main method that executes.

➤ ***Note***

You cannot reference a Java main method in a SQLJ create function statement.

If you reference a Java main method in a SQLJ create procedure statement, the command main method must have the Java method signature `String[]` as in:

```
public static void main(java.lang.String[]) {
...
}
```

If the Java method signature is specified in the create procedure statement, it must be specified as `(java.lang.String[])`. If the Java method signature is not specified, it is assumed to be `(java.lang.String[]).`

If the SQL procedure signature contains parameters, those parameters must be either char or varchar. At runtime, they are passed as a Java array of String.

Each argument you provide to the SQLJ procedure must be char, varchar, or a literal string because it is passed to the main method as an element of the java.lang.String array. You cannot use the dynamic result sets clause when creating a main procedure.

## Returning result sets

When you use SQLJ stored procedures to return result sets:

- The referenced method must include parameters of the java.sql.ResultSet class—one for each result set to be returned.
- The create procedure statement includes the dynamic result sets clause and an integer that specifies the maximum number of result sets that can be returned.
- The Java method signature can be implicit or explicit. It specifies the corresponding Java datatypes for the SQL stored procedure signature: simply mappable, object mappable, ADT mappable, or output mappable Java datatypes.

If the Java method signature is explicit, you must include a ResultSet[] method for each expected result set.

- Some restrictions apply:
  - If you specify an explicit Java method signature, you must include the trailing result sets in the signature.

```
create procedure ranked_emps (region integer)
            dynamic result sets 1
            language java parameter style java
            external name ‘Routines3.orderedEmps
                (int, java.sql.ResultSet[])’
```

  In this case, the parameter types are resolved using normal Java overloading conventions.

  - If you specify an implicit Java method signature, the system infers a partial Java signature for the SQL parameters.

```
create procedure ranked_emps (region integer)
            dynamic result sets 1
            language java parameter style java
            external name 'Routines3.orderedEmps'
```

In this case, the parameter types cannot be resolved using normal Java overloading conventions. The system infers a method signature of int and java.sql.ResultSet[], but it cannot distinguish between methods with signatures of int and different numbers of java.sql.ResultSet[] parameters.

## SQLJ standards and Sybase proprietary-implementation differences

This section describes differences between SQLJ Part 1 standard specifications and the Sybase proprietary implementation for SQLJ stored procedures and functions.

Table 4-2 describes Adaptive Server enhancements to the SQLJ implementation.

**Table 4-2: Sybase enhancements**

| Category | SQLJ standard | Sybase implementation |
|---|---|---|
| create procedure and create function commands | Supports only Java methods that do not return values. The methods must have void return type. | Supports Java methods that allow an integer value return. The methods referenced in create procedure can have either void or integer return types. |
| create procedure and create function commands | Supports only SQL datatypes in create procedure or create function parameter list. | Supports SQL datatypes and non-primitive Java datatypes as abstract data types (ADTs). |
| create function command | Does not support implicit SQL conversion to SQLJ routine invocations. | Supports implicit SQL conversion to SQLJ routine invocations. |
| drop procedure and drop function commands | Requires complete command name: drop procedure or drop function. | Supports complete function name and abridged names: drop proc and drop func. |

Table 4-3 describes SQLJ standard features not included in the Sybase implementation.

**Table 4-3: SQLJ features not supported**

| SQLJ category | SQLJ standard | Sybase implementation |
|---|---|---|
| create function command | Allows users to specify the same SQL name for multiple SQLJ functions. | Requires unique names for all stored procedure and functions. |

| SQLJ category | SQLJ standard | Sybase implementation |
|---|---|---|
| utilities | Supports sqlj.install_jar, sqlj.replace_jar, sqlj.remove_jar, and similar utilities to install, replace, and remove JAR files. | Supports the installjava utility and the remove java Transact-SQL command to perform similar functions. |

Table 4-4 describes the SQLJ standard features supported in part by the Sybase implementation.

**Table 4-4: SQLJ features partially supported**

| SQLJ category | SQLJ standard | Sybase implementation |
|---|---|---|
| create procedure and create function commands | Allows users to install different classes with the same name in the same database if they are in different JAR files. | Requires unique class names in the same database. |
| create procedure and create function commands | Supports the key words no sql, contains sql, reads sql data, and modifies sql data to specify the SQL operations the Java method can perform. | Supports modifies sql data only. |
| create procedure command | Supports java.sql.ResultSet and the SQL/OLB iterator declaration. | Supports java.sql.ResultSet only. |
| drop procedure and drop function commands | Supports the key word restrict, which requires the user to drop all SQL objects (tables, views, and routines) that invoke the procedure or function before dropping the procedure or function. | Does not support the restrict key word and functionality. |

Table 4-5 describes the SQLJ implementation-defined features in the Sybase implementation.

**Table 4-5: SQLJ features defined by the implementation**

| SQLJ category | SQLJ standard | Sybase implementation |
|---|---|---|
| create procedure and create function commands | Supports the deterministic \| not deterministic keywords, which specify whether or not the procedure or function always returns the same values for the out and inout parameters and the function result. | Supports only the syntax for deterministic \| not deterministic, not the functionality. |
| create procedure and create function commands | The validation of the mapping between the SQL signature and the Java method signature can be performed either when the create command is executed or when the procedure or function is invoked. The implementation defines when the validation is performed. | If the reference class has been changed, performs all validations when the create command is executed. |

| SQLJ category | SQLJ standard | Sybase implementation |
|---|---|---|
| create procedure and create function commands | Can specify the create procedure or create function commands within deployment descriptor files or as SQL DDL statements. The implementation defines which way (or ways) the commands are supported. | Supports create procedure and create function as SQL DDL statements outside of deployment descriptors. |
| Invoking SQLJ routines | When a Java method executes a SQL statement, any exception conditions are raised in the Java method as a Java exception of the Exception.SQLException subclass. The effect of the exception condition is defined by the implementation. | Follows the rules for NestedSQL. |
| Invoking SQLJ routines | The implementation defines whether a Java method called using a SQL name executes with the privileges of the user who created the procedure or function or those of the invoker of the procedure or function. | SQLJ procedures and functions inherit the security features of SQL stored procedures and Java-SQL functions. |
| drop procedure and drop function commands | Can specify the drop procedure or drop function commands within deployment descriptor files or as SQL DDL statements. The implementation defines which way (or ways) the commands are supported. | Supports create procedure and create function as SQL DDL statements outside of deployment descriptors. |

## Commands, utilities, and system stored procedures

This section provides syntax and usage information for the new and enhanced Transact-SQL statements, utilities, and system stored procedures that support SQLJ features.

New statements:

- create function Transact-SQL command
- drop function Transact-SQL command

Enhanced statements:

- create procedure Transact-SQL command
- sp_depends system procedure
- drop procedure Transact-SQL command
- sp_helprotect system procedure
- installjava utility
- remove java Transact-SQL command

- sp_helpjava system procedure
- sp_help system procedure

## create function

### Description

Creates a user-defined function by adding a SQL wrapper to a Java static method. Can return a value defined by the method.

### Syntax

```
create function [owner.]sql_function_name
           ([sql_parameter_name sql_datatype
               [( length)| (precision[, scale])]
           [, sql_parameter_name sql_datatype
               [( length )     |
               ( precision[, scale ]) ... ]])
       returns sql_datatype
           [( length)| (precision[, scale ])]
       [modifies sql data]
       [returns null on null input |
           called on null input]
       [deterministic | not deterministic]
       [exportable]
       language java
       parameter style java
       external name 'java_method_name
           [([java_datatype[, java_datatype ...]])]'
```

### Parameters

*sql_function_name* – Is the Transact-SQL name of the function. It must conform to the rules for identifiers and cannot be a variable.

*sql_parameter_name* Is the name of an argument to the function. The value of each input parameter is supplied when the function is executed. Parameters are optional; a SQLJ function need not take arguments.

Parameter names must conform to the rules for identifiers. If the value of a parameter contains non-alphanumeric characters, it must be enclosed in quotes. This includes object names qualified by a database name or owner name, since they include a period. If the value of the parameter begins with a numeric character, it also must be enclosed in quotes.

*sql_datatype [(length) | ( precision [, scale])]* – Is the Transact-SQL datatype of the parameter. See the “create procedure” in the *Adaptive Server Enterprise Reference Manual* for more information about these options.

*sql_datatype* – is the SQL procedure signature.

*returns sql_datatype* Specifies the result datatype of the function.

*modifies sql data*– Indicates that the Java method invokes SQL operations, reads, and modifies SQL data in the database. This is the default implementation.

*deterministic* | *not deterministic* – Supported for syntactic compatibility with the ANSI standard. Not implemented.

*exportable* – Specifies that the procedure is to be run on a remote server using the Adaptive Server Omni feature. Both the procedure and the method it is built on must reside on the remote server.

*language java* – Specifies that the external routine is written in Java. This is a required clause for SQLJ functions.

*parameter style java* – Specifies that the parameters passed to the external routine at runtime are Java parameters. This is a required clause for SQLJ functions.

*external* – Indicates that create function defines a SQL name for an external routine written in a programming language other than SQL.

*name* – Specifies the name of the external routine (Java method). The specified name—'*java_method_name* [ *java_datatype*[{, *java_datatype*} ...]]'—is a character-string literal and must be enclosed in single quotes.

*java_method_name* – Specifies the name of the external Java method.

*java_datatype* – Specifies a Java datatype that is mappable or result-set mappable. This is the Java method signature.

## Permissions

Only the Database Owner can execute create function. The Database Owner cannot transfer permission for create function.

## drop function

### Description

Removes a SQLJ user-defined function.

### Syntax

```
drop func[tion] [owner.]function_name
       [, [owner.]function_name] ...
```

### Parameters

*[owner.]function_name* – Is the SQL name of a SQLJ function.

### Examples

```
drop function findnum
```

Deletes the SQLJ function findnum.

### Usage

- Adaptive Server checks the existence of a function each time a user or a program executes that function.
- drop function removes only user-created functions from the current database. It does not remove system functions.

### Permissions

Only the Database Owner can execute drop function.

## create procedure

### Description

Creates a stored procedure or extended stored procedure by adding a SQL wrapper to a Java static method. Can accept one or more user-supplied parameters and return result sets and output parameters.

### Syntax

```
create procedure [owner.]sql_procedure_name
       ([[ in | out | inout ] sql_parameter_name
           sql_datatype [( length) |
           (precision[, scale])]
       [, in | out | inout ] sql_parameter_name
           sql_datatype [( length) |
           (precision[, scale]) ...])
       [modifies sql data ]
       [dynamic result sets integer]
       [deterministic | not deterministic]
       language java
       parameter style java
       external name 'java_method_name
           [([java_datatype[, java_datatype ...]])]'
```

### Parameters

*sql_procedure_name* – Is the Transact-SQL name of the procedure. It must conform to the rules for identifiers and cannot be a variable. Specify the owner's name to create another procedure of the same name owned by a different user in the current database. The default value for *owner* is the current user.

*in* | *out* | *inout* – Specifies the mode of the listed parameter. in indicates an input parameter; out indicates an output parameter; and inout indicates a parameter that is both an input and an output parameter. The default mode is in.

*sql_parameter_name* – Is the name of an argument to the procedure. The value of each input parameter is supplied when the procedure is executed. Parameters are optional; a SQLJ stored procedure need not take arguments.

Parameter names must conform to the rules for identifiers. If the value of a parameter contains non-alphanumeric characters, it must be enclosed in quotes. This includes object names qualified by a database name or owner name, since they include a period. If the value of the parameter begins with a numeric character, it also must be enclosed in quotes.

*sql_datatype [(length)* | *( precision [, scale])]* – Is the Transact-SQL datatype of the parameter. See “create procedure” in the *Adaptive Server Enterprise Reference Manual* for more information about these options.

*sql_datatype* is the SQL procedure signature.

modifies sql data – Indicates that the Java method invokes SQL operations, reads, and modifies SQL data in the database. This is the default.

*dynamic result sets integer* – Specifies that the Java method can return SQL result sets. *integer* specifies the maximum number of result sets the method can return. This value is implementation-defined.

*deterministic* | *not deterministic* – This syntax is supported for compatibility with other SQLJ-compliant vendors. Use of these keywords effects no action.

*language java* – Specifies that the external routine is written in Java. This is a required clause for SQLJ stored procedures.

*parameter style java* – Specifies that the parameters passed to the external routine at runtime are Java parameters. This is a required clause for SQLJ stored procedures.

*external* – Indicates that create procedure defines a SQL name for an external routine written in a programming language other than SQL.

*name* – Specifies the name of the external routine (Java method). The specified name—‘*java_method_name* [ *java_datatype*[{, *java_datatype*} ...]]’—is a character-string literal and must be enclosed in single quotes.

*java_method_name* – Specifies the name of the external Java method.

*java_datatype* – Specifies a Java datatype that is mappable or result-set mappable. This is the Java method signature.

## Examples

```
create procedure get_emps ()
   language java parameter style java
   external name “Example1.getEmps”
```

This example creates the SQLJ procedure get_emps, with no input or output parameters, on the method Example1.getEmps.

### Usage

### Permissions

create procedure permission defaults to the Database Owner, who can transfer it to other users. Permission to use a procedure must be granted explicitly with the grant command and may be revoked with the revoke command.

See the Transact-SQL command “create procedure” in the *Adaptive Server Enterprise Reference Manual* for detailed information about permissions and usage.

## drop procedure

### Description

Removes a Transact-SQL or SQLJ stored procedure.

### Syntax

```
drop proc[edure] [owner.]procedure_name
       [, [owner.]procedure_name] ...
```

### Parameters

*[owner.]procedure_name* – Is the name of the procedure to drop. Specify the owner's name to drop a procedure of the same name owned by a different user in the current database. The default value for owner is the current user.

### Usage

The syntax and use of drop procedure is the same for Transact-SQL and SQLJ procedures. See drop procedure in the *Adaptive Server Enterprise Reference Manual* for usage information, examples, and permissions.

## installjava

### Description

Installs JAR files in the database from a client operating system file.

### Syntax

```
installjava -f file_name
       [ -new | -update ]
       [ -j jar_name ]
```

### Usage

Changes / enhancements for Adaptive Server version 12.5.beta:

- You cannot install a new JAR file in the database if the installation of that file causes an installed class, referenced by a procedure or function, to be removed. If you install such a JAR, an exception is raised when the SQLJ routine is called.
- You cannot install a new JAR file in the database if the installation of that file contains a replacement class for an installed class, referenced by a procedure or function, that does not have a valid signature for the referenced procedure or function. If you install such a JAR, an exception is raised.

For complete syntax and usage information on this utility, see installjava in the *Adaptive Server Reference Manual*.

## remove java

### Description

Removes one or more Java-SQL classes, packages, or JARs from a database.

### Syntax

```
remove java
       class class_name [, class_name]...
       | package package_name [, package_name]...
       | jar jar_name [, jar_name]...[retain classes]
```

### Usage

Changes / enhancements for Adaptive Server version 12.5.beta:

You cannot remove a Java-SQL class if that class is directly referenced by a SQLJ stored procedure or function.

To remove a Java-SQL class from the database, you must:

- Delete all SQLJ stored procedures or functions that directly reference the class using drop procedure and / or drop function.
- Delete the Java-SQL class from the database using remove java.

For complete syntax and usage information on this Transact-SQL command, see remove java in the *Adaptive Server Reference Manual*.

## sp_depends

### Description

Displays information about database object dependencies. Lists the database objects that depend on the specified object and the database objects upon which the specified object depends.

### Syntax

**sp_depends** ***objname***

### Parameters

*objname* – The name of a table, view, Transact-SQL stored procedure, trigger, SQLJ stored procedure, SQLJ function, default (whether created using a create statement or using the create table statement), check constraint, or rule to be examined for dependencies.

You cannot specify a database name. Use owner names if the object owner is not the user running the command and not the Database Owner.

### Examples

```
sp_depends region_of
```

Lists objects referenced by the region_of SQLJ function and objects that reference the region_of SQLJ function.

### Usage

Changes and enhancements for Adaptive Server Enterprise 12.5.beta:

- SQLJ stored procedures and SQLJ functions are included as database objects for which you can list dependencies. The only dependencies of SQLJ stored procedures and SQLJ functions are Java classes.
- SQLJ stored procedures and SQLJ functions will be listed as dependencies of other database objects.
- If *objname* is a SQLJ stored procedure or SQLJ function, sp_depends lists the Java class in the routine's external name—not classes specified as the return type or as datatypes in the parameter list.
- If *objname* is a rule, default, or check constraint, sp_depends displays the table to which the rule, default, or check constraint is bound.

For complete syntax and usage information on this system procedure, see sp_depends in the *Adaptive Server Reference Manual*

# sp_help

## Description

Reports information about a database object (any object listed in sysobjects) and about system or user-defined datatypes.

## Syntax

```
sp_help [objname]
```

## Parameters

**objname**

The name of an object listed in sysobjects.

## Examples

```
sp_help region_of
```

Displays information about the SQLJ function region_of.

## Usage

Changes/enhancements for Adaptive Server version 12.5.beta provide parameter information on two new object types:

- SQLJ functions
- SQLJ procedures

For complete syntax and usage information on this system procedure, see sp_help in the *Adaptive Server Reference Manual*.

## sp_helpjava

### Description

Displays information about Java classes and associated JARs that are installed in the database.

### Syntax

```
sp_helpjava ["class" [, java_class_name [, “detail” |
   “depends” ]] | "jar"
   [, jar_name [, “depends” ] ] ]
```

### Parameters

*"class" | "jar"* – Specifies whether to display information about a class or a JAR. Both class and jar are keywords, so the quotes are required.

*java_class_name* – The name of the class about which you want information. The class must be a system class or a user-defined class that is installed in the database.

*“detail” | “depends”* – *detail* lists detailed information about the specified class.

*depends* lists all the database objects that depend on the specified class or classes in the specified JAR, including SQLJ functions, SQLJ procedures, views, Transact-SQL procedures, and tables.

Both detail and depends are keywords, so the quotes are required.

*jar_name* – The name of the JAR for which you want to see information. The JAR must be installed in the database using installjava.

### Usage

Enhancements for Adaptive Server version 12.5.beta:

- The depends keyword lists dependencies of specified class or classes if the class is listed in the external name clause or used as a datatype.

For complete syntax and usage information on this system procedure, see sp_helpjava in the *Adaptive Server Reference Manual*.

# sp_helprotect

## Description

Reports on permissions for database objects, users, groups, or roles.

## Syntax

```
sp_helprotect [name [, username [, “grant”
       [, “none” | “granted” | “enabled” | role_name
   ]]]]
```

## Parameters

*name* – Is either the name of the table, view, stored procedure, SQLJ stored procedure, SQLJ function, or the name of a user, user-defined role, or group in the current database. If you do not provide a name, sp_helprotect reports on all permissions in the database.

*username* – A user’s name in the current database.

*grant* – Displays the privileges granted to name with grant option.

*none* – Ignores roles granted to the user when determining permissions granted.

*granted* – Includes information on all roles granted to the user when determining permissions granted.

*enabled* – Includes information on all roles activated by the user when determining permissions granted.

*role_name* – Displays permission information for the specified role only, regardless of whether this role has been granted to the user.

## Usage

Enhancements for Adaptive Server version 12.5.beta:

- Lists permissions for SQLJ stored procedures.
- Advises that SQLJ function access is public:

```
Implicit grant to public for SQLJ functions.
```

For complete syntax and usage information on this system procedure, see sp_helprotect in the *Adaptive Server Reference Manual*

# 5 XML in the Database

This chapter uses examples to describe how you can use Java tools to access Extensible Markup Language (XML) documents in Adaptive Server.

➤ ***Note***

**isql** can display only the first 50 characters of a result set that is derived from XML data. However, the **isql** examples in this chapter display the entire result set for purpose of illustration. To see the entire result set for any of the examples, use **xml.XqlDriver** to run the query. **xml.XqlDriver** is explained in "Querying XML documents using xml.XqlDriver" on page 108.

## Introduction

Like Hypertext Markup Language (HTML), XML is a markup language and a subset of Standardized General Markup Language (SGML). XML, however, is more complete and disciplined, and it allows you to define your own application-oriented markup tags. These properties make XML particularly suitable for data interchange.

You can generate XML-formatted documents from data stored in Adaptive Server and, conversely, store XML documents and data extracted from them in Adaptive Server. You can also use Adaptive Server to search XML documents stored on the Web.

Adaptive Server uses the XML Query Language (XQL) to search XML documents. XQL is a path-based query language that searches the XML documents using the XML structure.

Many of the XML tools needed to generate and process XML documents are written in Java. Java in Adaptive Server provides a good base for XML-SQL applications using both universal and application-specific tools.

This chapter first provides a general discussion of XML and how you can use XML in the Adaptive Server database. It then presents a series of examples that you can use as guidelines for using XML in your Adaptive Server database.

### Source code and Javadoc

The source code for the Java classes described in this chapter is available in *$SYBASE/$SYBASE_ASE/sample/JavaSql* (UNIX) or *%SYBASE% \ Ase-12_5 \ sample \ JavaSql* (Windows NT), which also contains Javadoc-generated HTML pages with the specifications of the referenced packages, classes, and methods.

### References

This chapter presents a overview of XML. For detailed information, refer to these Web documents:

- World Wide Web Consortium (W3C), at http://www.w3.org
- W3C, Document Object Model (DOM), at http://www.w3.org/DOM/
- W3C, Extensible Markup Language (XML), at http://www.w3.org/XML/
- W3C, Extensible Stylesheet Language (XSL), at http://www.w3.org/TR/WD-xsl/
- Sun Microsystems, Inc, Java Project X Technology Release 1, at http://developer.java.sun.com/developer/earlyAccess/xml/index.html
- Megginson Technologies, SAX 1.0: The Simple API for XML, at http://www.megginson.com/SAX/

## An overview of XML

XML is a markup language and subset of SGML. It was created to provide functionality that goes beyond that of HTML for Web publishing and distributed document processing.

XML is less complex than SGML, but more complex and flexible than HTML. Although XML and HTML can usually be read by the same browsers and processors, XML has characteristics that make it better able to share documents:

- XML documents possess a strict phrase structure that makes it easy to find and access data. For example, opening tags of all elements must have a corresponding closing tag, for example, <p>A paragraph.<\p>.
- XML lets you develop and use tags that distinguish different types of data, for example, customer numbers or item numbers.
- XML lets you create an application-specific document type, which makes it possible to distinguish one kind of document from another.
- XML documents allow different views of the XML data. XML documents contain only markup and content; they do not contain formatting instructions. Formatting instructions are normally provided on the client using Extensible Style Language (XSL) specifications.

You can store XML documents in Adaptive Server using the following formats:

- As XML in a text or image column
- As XML in a char or varchar column that is less than 255 characters long
- As parsed XML in an image column

## A sample XML document

The sample Order document is designed for a purchase order application. Customers submit orders, which are identified by a date and a customer ID. Each order item has an item ID, an item name, a quantity, and a unit designation.

It might display on screen like this:

ORDER

Date: July 4, 1999

Customer ID: 123

Customer Name: Acme Alpha

**Table 5-1: Items:**

| Item ID | Item Name | Quantity |
|---|---|---|
| 987 | Coupler | 5 |
| 654 | Connector | 3 dozen |
| 579 | Clasp | 1 |

The following is one representation of this data in XML:

```
<?xml version="1.0"?>
    <Order>
 <Date>1999/07/04</Date>
 <CustomerId>123</CustomerId>
 <CustomerName>Acme Alpha</CustomerName>

   <Item>
 <ItemId> 987</ItemId>
 <ItemName>Coupler</ItemName>
 <Quantity>5</Quantity>
 </Item>

<Item>
 <ItemId>654</ItemId>
 <ItemName>Connector</ItemName>
 <Quantity unit="12">3</Quantity>
 </Item>

<Item>
 <ItemId>579</ItemId>
 <ItemName>Clasp</ItemName>
 <Quantity>1</Quantity>
 </Item>

</Order>
```

The XML document for the order data consists of these parts:

- The XML declaration, <?xml version=“1.0”?>, which identifies Order as an XML document.

XML documents are represented as character data. In each document, the character encoding (character set) is specified, either explicitly or implicitly. To explicitly specify the character set, include it in the XML declaration. For example:

```
<?xml version=”1.0” encoding=”ISO-8859-1”>
```

If you do not include the character set in the XML declaration, the default, UTF8, is used.

➤ ***Note***

When the default character sets of the client and server differ, Adaptive Server bypasses normal character set translations so that the declared character set continues to match the actual character set. See "Character sets and XML data" on page 104.

- User-created element tags such as <Order>...</Order>, <CustomerId>...</CustomerId>, <Item>....</Item>. In XML documents, all opening tags must have a corresponding closing tag.
- Text data such as "Acme Alpha," "Coupler," and "579."
- Attributes embedded in element tags such as <Quantity unit = "12">. This kind of coding allows you the flexibility to customize elements.

A document with these parts, and with the element tags strictly nested, is called a **well-formed XML document**. Note that in the example above, element tags describe the data they contain, and the document contains no formatting instructions.

## XML document types

A **document type definition** (DTD) defines the structure of a class of XML documents, making it possible to distinguish between classes. A DTD is a list of element and attribute definitions unique to a class. Once you have set up a DTD, you can reference that DTD in another document, or embed it in the XML document.

Here is another example of an XML document:

```
<?xml version="1.0"?>
 <Info>
    <OneTag>1999/07/04</OneTag>
    <AnotherTag>123</AnotherTag>
    <LastTag>Acme Alpha</LastTag>

   <Thing>
        <ThingId> 987</ThingId>
        <ThingName>Coupler</ThingName>
        <Amount>5</Amount>
        <Thing/>
```

```
     <Thing>
      <ThingId>654</ThingI
      <ThingName>Connecter</ThingNam
  </Thing>

      <Thing>
          <ThingId>579</ThingId>
          <ThingName>Clasp</ThingName>
          <Amount>1</Renew>
      </Thing>
  </Info>
```

This example, called Info, is a well-formed document and has the same structure and data as the XML Order document. Nonetheless, it would not be recognized by a processor designed for Order documents because each have different DTDs.

The DTD for XML Order documents is:

```
<!ELEMENT Order (Date, CustomerId, CustomerName,
    Item+)>
 <!ELEMENT Date (#PCDATA)>
 <!ELEMENT CustomerId (#PCDATA)>
 <!ELEMENT CustomerName (#PCDATA)>
 <!ELEMENT Item (ItemId, ItemName, Quantity)>
 <!ELEMENT ItemId (#PCDATA)>
 <!ELEMENT ItemName (#PCDATA)>
 <!ELEMENT Quantity (#PCDATA)>
 <!ATTLIST Quantity units CDATA #IMPLIED>
```

This DTD specifies that:

- An order must consist of: a date, a customer ID, a customer name, and one or more items. “+” indicates one or more items. These items are required. A question mark indicates an optional element (for example, “CustomerName?”). An asterisk indicates that an element can occur zero or more times (for example, “Item*”).
- Elements defined by “(#PCDATA)” are character text.
- The “<ATTLIST…>” definition specifies that quantity elements have a “units” attribute; the “#IMPLIED” specification indicates that the “units” attribute is optional.

The character text of XML documents is not constrained. For example, there is no way to specify that the text of a quantity element should be numeric, and thus the following would be valid:

```
<Quantity unit=”Baker’s dozen”>three</Quantity>
<Quantity unit=”six packs”>plenty</Quantity>
```

Restrictions on the text of elements are handled by applications that process XML data.

An XML's DTD must follow the <?xml version="1.0"?> instruction. You can either include the DTD within your XML document, or you can reference an external DTD.

- To reference a DTD externally, use something like this:

```
<?xml version="1.0"?>
 <!DOCTYPE Order SYSTEM  "Order.dtd”>
 <Order>
…
 </Order>
```

- Here's how an embedded DTD might look:

```
<?xml version="1.0"?>
 <!DOCTYPE Order [
 <!ELEMENT Order (Date, CustomerId, CustomerName,
    Item+)>
 <!ELEMENT Date (#PCDATA)
 <!ELEMENT CustomerId (#PCDATA)>
 <!ELEMENT CustomerName (#PCDATA)>
 <!ELEMENT Item (ItemId, ItemName, Quantity)>
 <!ELEMENT ItemId (#PCDATA)>
 <!ELEMENT ItemName (#PCDATA)>
 <!ELEMENT Quantity (#PCDATA)>
 <!ATTLIST Quantity units CDATA #IMPLIED>

    ]>
 <Order>
       <Date>1999/07/04</Date>
       <CustomerId>123</CustomerId>
       <CustomerName>Acme Alpha</CustomerName>

     <Item>
       …
    </Item>
 </Order>
```

DTDs are not required for XML documents. However, a **valid XML document** has a DTD and conforms to that DTD.

## XSL: formatting XML information

You can use XSL to format XML documents. XSL specifications (style sheets) consist of a set of rules that define the transformation of an XML document into either an HTML document or a different XML document:

- XSL specifications that transform an XML document into HTML can specify normal HTML formatting details in the output HTML.
- XSL specifications that transform an XML document into another XML document can map the input XML document to an output XML document with different element names and phrase structure.

You can create your own style sheets for the display of particular classes for particular applications. XSL is normally used with presentation applications rather than with applications for data interchange or storage.

## Character sets and XML data

If the declared character sets of your client and server differ, you must take care when declaring the character set of your XML documents.

Every XML document has a character set value. If that encoding is not declared in the XML declaration, the default value of UTF8 is assumed. The XML processor, when parsing the XML data, reads this value and handles the data accordingly. When the default character set of the client and server differ, Adaptive Server bypasses normal character set conversions to ensure that the declared character set and the actual character set remain the same.

- If you introduce an XML document into the database by providing the complete text in the values clause of an **insert** statement, Adaptive Server translates the entire SQL statement into the server's character set before processing the insertion. This is the way Adaptive Server normally translates character text, and you must make sure that the declared character set of the XML document matches that of the server.
- If you introduce an XML document into the database using **writetext** or Open Client CT-Library or Open Client DB-Library programs, Adaptive Server recognizes the XML document from the XML declaration and does *not* translate the character set to that of the server.
- If you read an XML document from the database, Adaptive Server does *not* translate the character set of the data to that of the client, thus preserving the integrity of the XML document.

## Installing XML in Adaptive Server

This section assumes you have already enabled Java in Adaptive Server. For information about enabling Java, see Getting Started With Java in this document.

**installjava** copies a JAR file into Adaptive Server and makes the Java classes in that JAR file available for use in the current database. The syntax is:

```
installjava
    -f file_name
    [-new | -update]
    [-j  jar_name]
    ...
```

Where:

- *file_name* is the name of the JAR file you are installing in the server.
- **new** informs the server this is a new file.
- **update** informs the server you are updating an existing JAR file.
- *jar_name* is the name of the JAR file from which you are retaining or extracting the classes.

For more information about **installjava**, see the *Utility Guide* for your platform.

To add support for XML in Adaptive Server, You must install the *xml.jar* and *xerces.jar* files. These JAR files are located in *$SYBASE/ASE-12_5/sample/Javasql*.

For example, to install the *xml.jar* file:

```
installjava -Usa -P -Sserver_name -f /$SYBASE/ASE
12_5/sample/Javasql/xml.jar
```

## Setting the CLASSPATH environment variable

You must set your *CLASSPATH* environment variable to include the directory that contain *xerces.jar*, *xml.zip*, and *runtime.zip*.To set the *CLASSPATH* environment variable:

```
setenv CLASSPATH .:<path_to_file>
```

## Retaining the JAR file

The **-j** option determines whether or not the JAR file is stored in the database.

- Do not specify the **-j** parameter, the Adaptive Server system does not retain any association of the classes with the JAR. This is the default option.
- Do specify the **-j** parameter, Adaptive Server installs the classes contained in the JAR in the normal manner, and then retains the JAR and its association with the installed classes.

If you retain the JAR file:

- You can use remote to remove the JAR and all classes associated with it. Otherwise, you must remove each class or package of classes individually.
- Other systems may request that the class associated with a given Java-SQL column be downloaded with the column value. If a class retains its association with the JAR, Adaptive Server can download the JAR, rather than individual classes.

## Updating installed classes

The **new** and **update** clauses of **installjava** indicate whether you want new classes to replace currently installed classes.

- If you specify **new**, you cannot install a class with the same name as an existing class.
- If you specify **update**, you can install a class with the same name as an existing class, and the newly installed class replaces the existing class.

◆ ***WARNING!***

**If you alter a class used as a column data type by reinstalling a modified version of the class, make sure that the modified class can read and use existing objects (rows) in tables using that class as a data type. Otherwise, you may be unable to access existing objects without reinstalling the class.**

Substitution of new classes for installed classes depends also on whether the classes being installed or the already installed classes are associated with a JAR. Thus:

- If you update a JAR, all classes in the existing JAR are deleted and replaced with classes in the new JAR.
- A class can only be associated with a single JAR. You cannot install a class in one JAR if a class of that same name is already installed and associated with another JAR. Similarly, you cannot install a class that is not associated with a JAR if that class is already installed and associated with a JAR.

You can, however, install a class in a retained JAR with the same name as an installed class not associated with a JAR. In this case, the class not associated with a JAR is deleted and the new class of the same name is associated with the new JAR.

If you want to reorganize your installed classes in new JARs, you may find it easier to first disassociate the affected classes from their JARs.

## XML parsers

You can analyze XML documents and extract their data using SQL character-string operations such as **substring**, **charindex**, and **patindex**. However, it is more efficient to use Java in SQL and tools written in Java such as XML parsers.

XML parsers can:

- Check that a document is well-formed and valid.
- Handle character-set issues.
- Generate a Java representation of a document's parse tree.
- Build or modify a document's parse tree.
- Generate a document's text from its parse tree.

Many XML parsers are available with a free license or are in the public domain. They normally implement two standard interfaces: the Simple API for XML (SAX) and the Document Object Model (DOM).

- *SAX* is an interface for parsing. It specifies input sources, character sets, and routines to handle external references. While parsing, it generates events so that user routines can process the document incrementally, and it returns a DOM object that is the parse tree of the document.

- *DOM* is an interface for the parse tree of an XML document. It provides facilities for stepping through and assembling a parse tree.

Applications that use the SAX and DOM interfaces are portable across XML parsers.

### Converting a raw XML document to a parsed version

It is much more efficient to use a parsed XML document for queries. Use the parse() method to convert and parse a raw text or image XML document and store the result. Use the **alter table** command to convert the raw XML document. For example:

```
alter table XMLTEXT add xmldoc IMAGE null
update XMLTEXT
set xmldoc = xml.Xql.parse(xmlcol)
```

This example converts the *xmlcol* column of the *XMLTEXT* table to parsed data and stores it in the *xmldoc* column.

## Querying XML documents

You can query XML documents from either:

- The command line – the **xml.XqlDriver** driver enables you to query XML documents from the command line. This method is helpful for developing and learning XQL queries, but is not the preferred method for querying XML documents in a production environment. See “Using XQL”, below, for a description of the preferred method.
- Adaptive Server – query XML documents stored in Adaptive Server using XQL. See “Using XQL” on page 112.

### Querying XML documents using xml.XqlDriver

**xml.XqlDriver** allows you to parse and query XML documents using XQL queries. **xml.XqlDriver** can only parse and query XML documents stored as files on the local system. You cannot use **xml.XqlDriver** to parse or query XML documents stored in a database or over the network. **xml.XqlDriver** is invoked with the **java** command.

Because **xml.XqlDriver** is easily used, it can be useful for developing XQL scripts and learning XQL. However, Sybase recommends that you use **xml.XqlDriver** only as a standalone program, and not as part of another Java program because **xml.XqlDriver** includes a **main()** method. A Java program can only include one **main()** method, and if you include **xml.XqlDriver** in another Java program that includes **main()**, the program will attempt to implement both **main()** methods, which causes an error in Java.

The syntax for **xml.XqlDriver** is:

```
java xml.XqlDriver
   -qstring[XQL_query]
   -validate [true | false]
   -infile [string]
   -outfile [string]
   -debug
   -help
```

Where:

- **qstring** specifies the XQL query.
- **validate** checks the validity of the XML documents.
- **infile** is the XML document you are querying.
- **outfile** is the operating system file in which you are storing the parsed XML document
- **debug** allows you to debug the **xml.XqlDriver** program,.
- **help** displays the **xml.XqlDriver** syntax.

For information about the XQL language, see "Using XQL" on page 112.

For example, the following query selects all the book titles from *bookstore.xml*:

```
java xml.XqlDriver -qstring
"/bookstore/book/title" -infile bookstore.xml

start DTD
NAME SPACE : http://www.placeholder-name-
here.com/schema/
NAME SPACE : http://www.placeholder-name-
here.com/schema/
NAME SPACE : http://www.placeholder-name-
here.com/schema/
Query  returned true and the  result is

<xql_result>
<title>Seven Years in Trenton</title>
<title>History of Trenton</title>
<title>Trenton Today, Trenton Tomorrow</title>
</xql_result>
```

The following example lists all the author's first names *bookstore.xml* file. XQL uses a zero-based numbering system; that is, "0" specifies the first occurrence of an element in a file.

```
java xml.XqlDriver -qstring "/bookstore/book/author/first-
name[0]" -infile bookstore.xml
start DTD
NAME SPACE : http://www.placeholder-name-here.com/schema/
NAME SPACE : http://www.placeholder-name-here.com/schema/
NAME SPACE : http://www.placeholder-name-here.com/schema/
Query  returned true and the  result is

<xql_result>
           <first-name>Joe</first-name>
           <first-name>Mary</first-name>
           <first-name>Toni</first-name>
</xql_result>
```

The following lists all the authors listed in *bookstore.xml* whose last name is "Bob":

```
java xml.XqlDriver -qstring "/bookstore/book/author[last-
name='Bob']" -infile bookstore.xml
```

```
start DTD
NAME SPACE : http://www.placeholder-name-here.com/schema/
NAME SPACE : http://www.placeholder-name-here.com/schema/
NAME SPACE : http://www.placeholder-name-here.com/schema/
Query  returned true and the  result is

<xql_result>
     <author>
     <first-name>Joe</first-name>
     <last-name>Bob</last-name>
     <award>Trenton Literary Review Honorable
Mention</award></author>
     <author>
     <first-name>Mary</first-name>
     <last-name>Bob</last-name>
     <publication>Selected Short Stories of
     <first-name>Mary</first-name>
     <last-name>Bob</last-name></publication></author>
     <author>
     <first-name>Toni</first-name>
     <last-name>Bob</last-name>
     <degree from=Trenton U>B.A.</degree>
<    degree from=Harvard>Ph.D.</degree>
     <award>Pulizer</award>
<    publication>Still in Trenton</publication>
     <publication>Trenton Forever</publication></author>
</xql_result>
```

### Validating your document

The **valid** option invokes a parser that makes sure the XML document you are querying conforms to its DTD. Your XML document must have a valid DTD before you run the **validate** option.

For example, this command makes sure the *bookstore.xml* document conforms to its DTD:

```
java xml.XqlDriver -qstring "/bookstore" -validate -infile bookstore.xml
```

### Saving result sets as parsed files using xml.XqlDriver

It is much faster to querying parsed XML files than to query plain text XML files. The **outfile** option allows you to save the XML output to a file as a parsed document. The syntax for parsing an XML file is:

```
java xml.XqlDriver -qstring [“file_name”] -infile input_file_name -
outfile new_file_name
```

For example, to save *bookstore.xml* as a parsed file named *bookstore_xml.prs*:

```
java xml.XqlDriver -qstring “/bookstore” -infile bookstore.xml -
outfile bookstore_xml.prs
```

Subsequent queries that use *bookstore_xml.prs* run much faster than queries that use *bookstore.xml*. Use the parsed file name as the input file. For example:

```
java xml.XqlDriver -qstring "/bookstore/book/title" -infile
bookstore_xml.prs
Query  returned true and the  result is

<xql_result>
<title>Seven Years in Trenton</title>
<title>History of Trenton</title>
<title>Trenton Today, Trenton Tomorrow</title>
</xql_result>
```

The result set is the same as the query that selected all book titles from *bookstore.xml*, but **xml.XqlDriver** does not start the DTD. Because the XML is already parsed, it reads the file and issues the result set without having to parse it.

## Using XQL

The XML Query Language (XQL) has been designed as a general-purpose query language for XML. XQL is a path-based query language for addressing and filtering the elements and text of XML documents, and is a natural extension to the XSL syntax. XQL provides a concise, understandable notation for pointing to specific elements and for searching for nodes with particular characteristics. XQL navigation is through elements in the XML tree.

This section does not discuss XQL in depth. The most common XQL operators include:

- Child operator, **/** – indicates hierarchy. The following example returns *<author>* elements that are children of *<front>* elements from the *xmlcol* column of the *xmlimage* table:

```
select xml.Xql.query("/bookstore/book", xmlcol)
from xmlimage

<xql_result>
               <book style=autobiography>
<title>S
```

- Descendant operator, **//** – indicates that the query searches through any number of intervening levels. That is, a search using the descendant operator finds an occurrence of an element at any level of the XML structure. The following query finds all the instances of <emph> elements that occur in an <excerpt> element:

```
select xml.Xql.query("/bookstore/book/excerpt//emph",xmlcol)
from xmlimage

<xql_result>
               <emph>I</emph>
</xql_result>
```

- Equals operator, **=** – specifies the content of an element or the value of an attribute. The following query finds all examples where last-name = Bob:

```
select xml.Xql.query("/bookstore/book/author[last-name='Bob']",
xmlcol)
from xmlimage

<xql_result>
               <author>
               <first-name>Joe</first-name>
               <last-name>Bob</last-name>
               <award>Trenton Literary Review Honorable
Mention</award></author>
               <author>
               <first-name>Mary</first-name>
               <last-name>Bob</last-name>
               <publication>Selected Short Stories of
               <first-name>Mary</first-name>
               <last-name>Bob</last-name></publication></author>
               <author>
               <first-name>Toni</first-name>
               <last-name>Bob</last-name>
               <degree from=Trenton U>B.A.</degree>
               <degree from=Harvard>Ph.D.</degree>
               <award>Pulizer</award>
               <publication>Still in Trenton</publication>
               <publication>Trenton
Forever</publication></author>
“</xql_result>
```

- Filter operator, **[]** – Filters the set of nodes to its left based on the conditions inside the brackets. This example finds any occurrences of authors whose first name is Mary that are listed in a book element:

```
select xml.Xql.query("/bookstore/book[author/first-name = 'Mary']",
xmlcol)
from xmlimage
<xql_result>
                 <book style=textbook>
                 <title>History of Trenton</title>
                 <author>
                 <first-name>Mary</first-name>
                 <last-name>Bob</last-name>
                 <publication>Selected Short Stories of
                 <first-name>Mary</first-name>
                 <last-name>Bob</last-name></publication></author>
<price>55</price></book>
```

- Subscript operator, **[*index_ordinal*]** – finds a specific instance of an element. This example finds the second book listed in the XML document. Remember that XQL is zero-based, so it begins numbering at 0:

```
select xml.Xql.query("/bookstore/book[1]", xmlcol)
from xmlimage
Query  returned true and the  result is
<xql_result>
                       <book style=textbook>
                       <title>History of Trenton</title>
                       <author>
                       <first-name>Mary</first-name>
                       <last-name>Bob</last-name>
                       <publication>Selected Short Stories of
                       <first-name>Mary</first-name>
                       <last-name>Bob</last-
name></publication></author>
                       <price>55</price></book>
</xql_result>
```

- Boolean expressions – you can use Boolean expressions within filter operators. For example, this query returns all <author> elements that contain at least one <degree> and one <award>:

```
select xml.Xql.query("/bookstore/book/author[degree and award]",
xmlcol)
from xmlimage

<xql_result>
               <author>
               <first-name>Toni</first-name>
               <last-name>Bob</last-name>
               <degree from=Trenton U>B.A.</degree>
               <degree from=Harvard>Ph.D.</degree>
               <award>Pulizer</award>
               <publication>Still in Trenton</publication>
               <publication>Trenton
Forever</publication></author>
</xql_result>
```

### Query structures that affect performance

The placement of the where clause in a query affects processing. For example, this query selects all the books whose author's first name is Mary:

```
select xml.Xql.query("/bookstore/book[author/first-name ='Mary']",
XmlFile

<xql:result ><book style="textbook">
      <title>History of Trenton</title>
      <author>
      <first-name>Mary</first-name>
      <last-name>Bob</last-name>
      <publication>
      Selected Short Stories of
      <first-name>Mary</first-name>
      <last-name>Bob</last-name>
      </publication>
      </author>
      <price>55</price>
</book></xql:result>
```

Note that, **query()** is invoked twice, once in the **where** clause and once in the **select** clause, which means the query executes twice and may be slow for large documents.

As an alternative, save the result set in an object while executing the query in the where clause and then restore the result in the select. For example:

```
                declare @result HoldString
                select @result = new HoldString()
                select @result>>get()
                from XMLDAT
                where
                    @result>>put(xml.Xql.query("/bookstore/book
                        [author/first-name='Mary'],xmlcol)) !=
convert(xml.Xql, null)>>EmptyResult
```

Sybase advises that you not store the result set in the **where** clause. The query does not always execute the **where** clause, so trying to retrieve its result in the **select** clause may generate an erroneous result set. Note that **HoldString** is an example class.

Because Adaptive Server stores each document in a column of a given row, when the query scans a set of rows in the **where** clause, more than one row may satisfy the search criteria. If this occurs, the query returns a separate XML result document for each qualified row. For example, if you create the following table:

```
create table XMLTAB ( xmlcol image)
insert XMLTAB values
      (XmlDoc.parse(<xml><A><B><C>c</C></B></A></xml>));
insert XMLTAB values
      (XmlDoc.parse(<xml><D><E><C>c</C></E></D></xml>));
```

And then execute the following query:

```
select xml.Xql.query("//C", xmlcol)
from XMLTAB
where
@xml>>query("//C",xmlcol)        != convert(xml.Xql,
null)>>EmptyResult
```

You would expect to get the following result set:

```
<xql_result>
<C>c</C>
<C>c</C>
</xql_result>
```

Instead, you get the following result set because the same row is returned twice, once from the **select** clause and once from the where clause:

## Using XQL to develop applications

You can use XQL to develop standalone applications, JDBC clients, JavaBeans, and EJBs to process XML data. The **query()** and **parse()** methods enable you to query and parse XML documents. Because you can write these applications as standalone applications, you do not have to depend on Adaptive Server to supply the result set. Instead you can query XML documents stored as operating system files or stored out on the Web.

### *Example standalone application*

The following example uses the **FileInputStream()** query to read the *bookstore.xml* file, and the **URL()** method to read a Web page named *bookstore.xml* which contains information about all the books in the bookstore:

```
String result;
FileInputStream XmlFile = new FileInputStream("bookstore.xml");
if ((result =
               Xql.query("/bookstore/book/author/first-name",
XmlFile))
               != Xql.EmptyResult )
{
          System.out.println(result);
}else{
          System.out.println("Query returned false\n");
}

URL _url = new URL("http://mybookstore/bookstore.xml");
if ((result =
               Xql.query("/bookstore/book/author/first-name",
url.openStream))
                   != Xql.EmptyResult )
{
               System.out.println(result);
}else{
               System.out.println("Query returned false\n");
}
```

### Example JDBC client

The following example uses the **Xql.query()** method to query the *xmlcol* column in the *XMLTEXT* text file:

```
String selectQuery = "select xmlcol from XMLTEXT";
Statement stmt = _con.createStatement();
ResultSet rs = (SybResultSet)stmt.executeQuery(selectQuery);
String result;

InputStream is = null;
while ((rs != null) && (rs.next()))
{
           is = rs.getAsciiStream(1);
           result = Xql.query("/bookstore/book/author", is);
           ...
}
```

The following example assumes that the parsed XML data is stored in an *image* column of the *XMLDOC* table. Although this application fetches an *image* column as a binary stream, it does not parse this during the query because it identifies the content of this binary stream as a parsed XML document. Instead, the application creates a **SybMemXmlStream** method from it and then executes the query. All this is done using the **Xql.query()** method, and does not require any input from the user:

```
String selectQuery = "select xmlcol from XMLDOC";
Statement stmt = _con.createStatement();
ResultSet rs = (SybResultSet)stmt.executeQuery(selectQuery);
InputStream is = null;
String result
while ((rs != null) && (rs.next()))
{
           is = rs.getBinaryStream(1);
           result = Xql.query("/bookstore/book/author/first-
name", is));
           ...
}
```

### Example EJB component

You can write EJB components that serve as query engines on an EJB server.

The component below includes an EJB called *XmlBean*. *XmlBean* includes the **query()** method, which allows you to query any XML document on the Web. In this component, **query()** first creates an **XmlDoc** object, then queries the document.

The remote interface looks like:

```
public interface XmlBean extends javax.ejb.EJBObject
{
               /**
           * XQL Method
           */
           public String XQL(String query, URL location) throws
java.rmi.RemoteException
;
}
```

The Bean implementation looks like:

```
public class XmlBean extends java.lang.Object implements
javax.ejb.SessionBean
{
               ....
               /**
               * XQL Method
               */
           public String XQL(String query, java.net.URL
location) throws
               java.rmi.RemoteException
{
               try {
                           String result;

                           if((result =
                               Xql.query(query,
location.openStream())) !=
                               Xql.EmptyResult)
                           {
                                   return (result);
                           }else{
                                   return (null);
                           }
                       }catch(Exception e){
                           throw new
java.rmi.RemoteException(e.getMessage());
                       }
                   }
....
}
```

And the client code looks like:

```
....
Context ctx = getInitialContext();
// make the instance of the class in Jaguar
XmlBeanHome _beanHome =
(XmlBeanHome)ctx.lookup("XmlBean");
_xmlBean = (XmlBean)_beanHome.create();
URL u = new URL("http://mywebsite/bookstore.xml");
String res=
xmlBean.XQL("/bookstore/book/author/first-name",u);
```

## Accessing XML in SQL

This chapter discusses three applications of XML in SQL:

- *Transact-SQL statements* such as **insert**, **select**, and **update** for referencing SQL columns and variables that contain XML documents. These SQL operations use Java classes and methods to manipulate the XML documents.
- *Java classes* to contain XML documents and to access and update the elements of those documents. The examples include an application-specific class for the Order document type, and a general class for arbitrary SQL result sets.
- An *XML parser*, which is used by the Java classes to analyze and manipulate XML documents.

The Java classes that are used to demonstrate XML applications are **XQL**, **JXml**, **OrderXml**, and **ResultSetXml**.

- **XQL** allows you to query any XML document. XQL also includes a query () method that enables you to query any subclass objects of **JXml**. For more information, see “Using XQL” on page 112 and “XML methods” on page 163.
- **JXml** stores and parses XML. It does not validate XML documents. It is designed as a base class for subclasses that:
  - Validate specific XML document types
  - Provide application-oriented methods

  **OrderXml** and **ResultSetXml** are two such subclasses.
- The **OrderXml** class is used to illustrate support for an application-specific XML document type. **OrderXml** validates Order documents for the Order DTD. You can use **OrderXml** methods to reference and update elements of the Order document.
- **ResultSetXml** represents SQL result sets. The **ResultSetXml** constructor validates the **ResultSet** document for the **ResultSet** DTD.

ResultSetXml methods are used to reference and update elements of the ResultSet document.

The ResultSetXml class illustrates support for a general XML document type capable of representing arbitrary SQL data.

“The OrderXml class for order documents” on page 124 and “The ResultSetXml class for result set documents” on page 144 describe these classes and their methods and parameters. For Javadoc HTML pages with detailed specifications for the classes and for source code, refer to *$SYBASE/$SYBASE_ASE/sample/JavaSql* (UNIX) or *%SYBASE% \ Ase-12_0 \ sample \ JavaSql* (Windows NT).

## Inserting XML documents

Use the parse() method to insert an XML document, which takes the XML document as the argument and returns sybase.aseutils.SybXmlStream. Adaptive Server provides implicit mapping between image and sybase.aseutils.SybXmlStream. The following is an insert statement:

```
insert XMLDAT
values (..,xml.Xql.parse("<xmldoc></xmldoc>",..))
```

Adaptive Server has an implicit mapping between image or text data and InputStream. You can pass image or text columns to parse() without doing any casting. The parse() UDF parses the document and returns sybase.ase.SybXmlStream, which Adaptive Server uses to write the data to the image column.

## Updating XML documents

To update a document, first delete the original data and then insert the new data. In a document-oriented application, the number of updates to a document or portion of a document are very infrequent compared to the number of reads. An update is similar to:

```
update XMLDAT
set xmldoc = xml.Xql.parse("<xmldoc></xmldoc>")
```

## Deleting XML documents

Deleting an XML document is similar to deleting any text column. For example, to delete a table named *XMLDAT*:

```
Delete XMLDAT
```

## Storing XML documents

To use XML documents for data interchange in Adaptive Server, you must be able to store XML documents or the data that they contain in the database. To determine how best to accomplish this, consider the following:

- *Mapping and storage*: What sort of correspondence between XML documents and SQL data is most suitable for your system?
- *Client or server considerations*: Should the mapping take place on the client or the server?
- *Accessing XML in SQL*: How do you want to access the elements of an XML document in SQL?

The rest of this section discusses each of these considerations; the remainder of the chapter discusses the classes and methods you can use with XML, including:

- A simple example to illustrate the basics of data storage and exchange of XML documents
- A generalized example that you can customize for your own XML documents

### Client or server considerations

You can execute Java methods either on a client or on a server, which is a consideration for element storage and hybrid storage. Document storage involves little or no processing of the document.

- Element storage – if you map individual elements of an XML document to SQL data, in most cases, the XML document is larger than the SQL data. It is generally more efficient to assemble and disassemble the XML document on the client and transfer only the SQL data between the client and the server.
- Hybrid storage – if you store both the complete XML document and extracted elements, then it is generally more efficient to extract the data from the server, rather than transfer it from the client.

## Mapping and storage

There are three basic ways to store XML data in Adaptive Server: **element storage**, **document storage**, or **hybrid storage**, which is a mixture of both.

- Element storage – In this method, you extract data elements from an XML document and store them as data rows and columns in Adaptive Server.

  For example, using the XML Order document, you can create SQL tables with columns for the individual elements of an order: *Date*, *CustomerId*, *CustomerName*, *ItemId*, *ItemName*, *Quantity*, and *Units*. You can then manage that data in SQL with normal SQL operations:

  - To produce an XML document for Order data contained in SQL, retrieve the data, and assemble an XML document with it.
  - To store an XML document with new Order data, extract the elements of that document, and update the SQL tables with that data.

- Document storage – In this method, you store an entire XML document in a single SQL column.

  For example, using the Order document, you can create one or more SQL tables having a column for Order documents. The data type of that column could be:

  - SQL text, or
  - A generic Java class designed for XML documents, or
  - A Java class designed specifically for XML Order documents

- Hybrid storage – this method, you store an XML document in an SQL column, and also extract some of its data elements into separate columns for faster and more convenient access.

Again, using the Order example, you can create SQL tables as you would for document storage, and then include (or later add) one or more columns to store elements extracted from the Order documents.

### Advantages and disadvantages of Storage Options

Each storage option has advantages and disadvantages. You must choose the option or options best for your operation.

- If you use element storage, all of the data from the XML document is available as normal SQL data that you can query and update using SQL operations. However, element storage has the overhead of assembling and disassembling the XML documents for interchange.
- Document storage eliminates the need for assembling and disassembling the data for interchange. However, you need to use Java methods to reference or update the elements of the XML documents while they are in SQL, which is slower and less convenient than the direct SQL access of element storage.
- Hybrid storage balances the advantages of element storage and document storage, but has the cost and complexity of redundant storage of the extracted data.

## A simple example for a specific result set

This section provides a simple example that demonstrates how you can store XML documents or the data that they contain in an Adaptive Server database.

The example in this section, the XML Order document type, is designed for a specific purchase-order application, and the Java methods created for it assume a specific set of SQL tables for storing purchase order data.

For a more generalized example, applicable to a range of SQL result sets, see “A customizable example for different result sets” on page 139.

### The *OrderXml* class for order documents

The example in this section uses the **OrderXml** class and its methods for basic operations on XML Order documents. The source code and Javadoc specifications for **OrderXml** are in *$SYBASE/$SYBASE_ASE/sample/JavaSql*.

**OrderXml** is a subclass of the **JXml** class, which is specialized for XML Order documents. The **OrderXml** constructor validates the document for the Order DTD. Methods of the **OrderXml** class support referencing and updating the elements of the Order document.

- Constructor: `OrderXml(String)`

Validates that the *String* argument contains a valid XML Order document, and then constructs an **OrderXml** object containing that document. For example, “doc” is a Java string variable containing an XML Order document, perhaps one read from a file:

```
xml.order.OrderXml  ox = new
xml.order.OrderXml(doc);
```

- Constructor: `OrderXml(date, customerId, dtdOption, server)`

  The parameters are all String.

  This method assumes a set of SQL tables containing Order data. The method uses JDBC to execute a SQL query that retrieves Order data for the given *date* and *customerId*. The method then assembles an XML Order document with the data.

  The *server* parameter identifies the Adaptive Server on which to execute the query.

  - If you invoke the method in a client environment, specify the server name.
  - If you invoke the method in Adaptive Server (in a SQL statement or in **isql**), specify either an empty string or the string “jdbc:default:connection,” which indicates that the query should be executed on the current Adaptive Server.

  The *dtdOption* parameter indicates whether you want the generated Order to contain the DTD or to reference it externally.

  For example:

```
xml.order.OrderXml ox = new OrderXml(“990704”, “123”,
               “external”, “antibes:4000?user=sa”);
```

- `void order2Sql(String ordersTableName, String server)`

  Extracts the elements of the Order document and stores them in a SQL table created by the **createOrdertable( )** method. *ordersTableName* is the name of the target table. The *server* parameter is as described for the **OrderXml** constructor. For example, if *ox* is a Java variable of type **OrderXml**:

  ```
  ox.order2Sql(“current_orders”,
  “antibes:4000?user=sa”);
  ```

  This call extracts the elements of the Order document contained in *ox*, and uses JDBC to insert the extracted elements into rows and columns of the table named *current_orders*.

- `static void createOrderTable(String ordersTableName, String server)`

Creates a SQL table with columns suitable for storing Order data: *customer_id*, *order_date*, *item_id*, *quantity*, and *unit*. *ordersTableName* is the name of the new table. The *server* parameter is as described for the **OrderXml** constructor. For example:

```
xml.order.OrderXml.createOrderTable
            (“current_orders”,
“antibes:4000?user=sa”);
```

- `String getOrderElement(String elementName)`

  *elementName* is “Date,” “CustomerId,” or “CustomerName.” The method returns the text of the element. For example, if *ox* is a Java variable of type **OrderXml**:

  ```
  String customerId =
  ox.getOrderElement(“CustomerId”);
  String customerName =
  ox.getOrderElement(“CustomerName”);
  String date = ox.getOrderElement(“Date”);
  ```

- `void setOrderElement(String elementName, String newValue)`

  *elementName* is as described for **getOrderElement( )**.The method sets that element to **newValue**. For example, if *ox* is a Java variable of type **OrderXml**:

  ```
  ox.setOrderElement(“CustomerName”, “Acme Alpha
  Consolidated”);
  ox.setOrderElement(“CustomerId”, “987a”);
  ox.setOrderElement(“Date”, “1999/07/05”);
  ```

- `String getItemElement(int itemNumber, String elementName)`

  *itemNumber* is the index of an item in the order. *elementName* is “ItemId,” “ItemName,” or “Quantity.” The method returns the text of the item. For example, if *ox* is a Java variable of type **OrderXml**:

  ```
  String itemId = ox.getItemElement(2, “ItemId”);
  String itemName = ox.getItemElement(2, “ItemName”);
  String quantity = ox.getItemElement(2, “Quantity”);
  ```

- `void setItemElement(int itemNumber, String elementName, String newValue)`

  *itemNumber* and *elementName* are as described for the **getItemElement** method. **setItemElement** sets the element to *newValue*. For example, if *ox* is a Java variable of type **OrderXml**:

```
ox.setItemElement(2, “ItemId”, “44”);
ox.setItemElement(2, “ItemName”, “cord”);
ox.setItemElement(2, “Quantity”, “3”);
```

- String getItemAttribute(int itemNumber, elementName, attributeName)

  *itemNumber* and *elementName* are described as for **getItemElement( )**. *elementName* and *attributeName* are both String. *attributeName* must be “unit.” The method returns the text of the unit attribute of the item.

➤ ***Note***

Since the Order documents currently have only one attribute, the *attributeName* parameter is unnecessary. It is included to illustrate the general case, for example, if *ox* is a Java variable of type **OrderXml**.

```
String itemId = ox.getItemAttribute(2, “unit”);
```

- void setItemAttribute (int itemNumber, elementName, attributeName, newValue)

  *itemNumber*, *elementName*, and *attributeName* are as described for **getItemAttribute( )**. *elementName*, *attributeName*, and *newValue* are String. The method sets the text of the unit attribute of the item to *newValue*. For example, if *ox* is a Java variable of type **OrderXml**:

  ```
  ox.setItemAttribute(2, “unit”, “13”);
  ```

- void appendItem(newItemId, newItemName, newQuantity, newUnit)

  The parameters are all String. The method appends a new item to the document, with the given element values. For example, if *ox* is a Java variable of type **OrderXml**:

  ```
  ox.appendItem(“77”, “spacer”, “5”, “12”);
  ```

- void deleteItem(int itemNumber)

  *itemNumber* is the index of an item in the order. The method deletes that item. For example, if *ox* is a Java variable of type **OrderXml**:

  ```
  ox.deleteItem(2);
  ```

## Creating and populating SQL tables for order data

In this section we create several tables that are designed to contain data from XML Order documents, so that we can demonstrate techniques for element, document, and hybrid data storage.

### Tables for element storage

The following SQL statements create SQL tables *customers*, *orders*, and *items*, whose columns correspond with the elements of the XML Order documents.

```
create table customers
    (customer_id varchar(5) not null unique,
    customer_name varchar(50) not null)

create table orders
    (customer_id varchar(5) not null,
    order_date datetime not null,
    item_id varchar(5) not null,
    quantity int not null,
    unit smallint default 1)

create table items
    (item_id varchar(5) unique,
    item_name varchar(20))
```

These tables need not have been specifically created to accommodate XML Order documents.

The following SQL statements populate the tables with the data in the example XML Order document (see “A sample XML document” on page 99):

```
insert into customers values("123", "Acme Alpha")

insert into orders values ("123", "1999/05/07",
    "987", 5, 1)

insert into orders values ("123", "1999/05/07",
    "654", 3, 12)

insert into orders values ("123", "1999/05/07",
    "579", 1, 1)

insert into items values ("987", "Widget")

insert into items values ("654",
    "Medium connecter")

insert into items values ("579",
    "Type 3 clasp")
```

Use **select** to retrieve the Order data from the tables:

```
select order_date as Date, c.customer_id as
CustomerId,
    customer_name as CustomerName,
    o.item_id as ItemId, i.item_name as ItemName,
    quantity as Quantity, o.unit as unit
 from customers c, orders o, items i
    where c.customer_id=o.customer_id and
o.item_id=i.item_id
```

| Date | CustomerId | CustomerName | ItemId | ItemName | Quantity | Unit |
|---|---|---|---|---|---|---|
| July 4 1999 | 123 | Acme Alpha | 987 | Coupler | 5 | 1 |
| July 4 1999 | 123 | Acme Alpha | 654 | Connector | 3 | 12 |
| July 4 1999 | 123 | Acme Alpha | 579 | Clasp | 1 | 1 |

### Tables for document and hybrid storage

The following SQL statement creates a SQL table for storing complete XML Order documents, either with or without extracted elements (for hybrid storage).

```
create table order_docs
    (id char(10) unique,
    customer_id varchar(5) null,  --  For an
        extracted “CustomerId” element
    order_doc xml.order.OrderXml)
```

## Using the element storage technique

This section describes the element storage technique for bridging XML and SQL.

- “Composing order documents from SQL data” on page 129 discusses the composition of an XML Order document from SQL data.
- “Decomposing data from an XML order into SQL” on page 131 discusses the decomposition of an XML Order document to SQL data.

### Composing order documents from SQL data

In this example, Java methods generate an XML Order document from the SQL data in the tables created in “Creating and populating SQL tables for order data” on page 128.

A constructor method of the **OrderXml** class maps the data. An call of that constructor might be:

```
new xml.order.OrderXml("990704", "123",
    “external”, "antibes:4000?user=sa");
```

This constructor method uses internal JDBC operations to:

- Execute a SQL query for the Order data
- Generate an XML Order document with the data
- Return the **OrderXml** object that contains the Order document

You can invoke the **OrderXml** constructor in the client or the Adaptive Server.

- If you invoke the **OrderXml** constructor in the client, the JDBC operations that it performs use jConnect to connect to the Adaptive Server and perform the SQL query. It then reads the result set of that query and generates the Order document on the client.
- If you invoke the **OrderXml** constructor in Adaptive Server, the JDBC operations that it performs use the native JDBC driver to connect to the current Adaptive Server and perform the SQL query. It then reads the result set and generates the Order document in Adaptive Server.

### *Generating an order on the client*

Designed to be implemented on the client, **main( )** invokes the constructor of the **OrderXml** class to generate an XML Order from the SQL data. That constructor executes a **select** for the given date and customer ID, and assembles an XML Order document from the result.

```
import java.io.*;
import util.*;
public class Sql2OrderClient {
    public static void main (String args[]) {
        try{
            xml.order.Order order =
                new xml.order.OrderXml("990704",
"123",
                "external",
"antibes:4000?user=sa");
            FileUtil.string2File("Order-
sql2Order.xml",
                order.getXmlText());
        } catch (Exception e) {
        System.out.println("Exception:");
        e.printStackTrace();
        }
    }
 }
```

### *Generating an order on the server*

Designed for the server environment, the following SQL script invokes the constructor of the OrderXml class to generate an XML Order from the SQL data:

```
declare @order xml.order.OrderXml
select @order =
    new xml.order.OrderXml('990704', '123',
    'external', '')

insert into order_docs (id, order_doc) values("3",
    @order)
```

## Decomposing data from an XML order into SQL

In this section, you extract elements from an XML Order document and store them in the rows and columns of the *Orders* tables. The examples illustrate this procedure in both server and client environments.

You decompose the elements using the Java method order2Sql() of the OrderXml class. Assume that *xmlOrder* is a Java variable of type OrderXml:

```
xmlOrder.order2Sql("orders_received",
"antibes:4000?user=sa");
```

The **order2Sql( )** call extracts the elements of the XML Order document contained in variable *xmlOrder*, and then uses JDBC operations to insert that data into the *orders_received* table. You can call this method on the client or on Adaptive Server:

- Invoked from the client, **order2Sql( )** extracts the elements of the XML Order document in the client, uses jConnect to connect to the Adaptive Server, and then uses the Transact-SQL **insert** command to place the extracted data into the table.
- Invoked from the server, **order2Sql( )** extracts the elements of the XML Order document in the Adaptive Server, uses the native JDBC driver to connect to the current Adaptive Server, and then use the Transact-SQL **insert** command to place the extracted data into the table.

### *Decomposing the XML document on the client*

Invoked from the client, the **main( )** method of the **Order2SqlClient** class creates a table named *orders_received* with columns suitable for Order data. It then extracts the elements of the XML Order contained in the file *Order.xml* into rows and columns of *orders_received*. It performs these actions with calls to the static method **OrderXml.createOrderTable( )** and the instance method **order2Sql( )**.

```
import util.*;
import xml.order.*;
import java.io.*;
import java.sql.*;
import java.util.*;

public class Order2SqlClient {
     public static void main (String args[]) {
          try{
              String xmlOrder =
                  FileUtil.file2String("order.xml");
              OrderXml.createOrderTable("orders_received",
                  "antibes:4000?user=sa");
              xmlOrder.order2Sql("orders_received",
                  "antibes:4000?user=sa");
          } catch (Exception e) {
          System.out.println("Exception:");
          e.printStackTrace();
          }
     }
 }
```

#### *Decomposing the XML document on the server*

Invoked from the server, the following SQL script invokes the **OrderXml** constructor to generate an XML Order document from the SQL tables, and then invokes the method **OX.sql2Order( )**, which extracts the Order data from the generated XML and inserts it into the *orders_received* table.

```
declare @xmlorder OrderXml
select @xmlorder = new OrderXml('19990704', '123',
     'external', '')
select  @xmlorder>>order2Sql('orders_received', '')
```

## Using the document storage technique

When using the document storage technique, you store a complete XML document in a single SQL column.This approach avoids the cost of mapping the data between SQL and XML when documents are stored and retrieved, but access to the stored elements can be slow and inconvenient.

### Storing XML order documents in SQL columns

This section provides examples of document storage from the client and from the server.

#### *Inserting an order document from a client file*

The following command-line call is representative of how you can insert XML data into Adaptive Server from a client file. It copies the contents of the *Order.xml* file (using the **–I** parameter) to the Adaptive Server and executes the SQL script (using the **–Q** parameter) using the contents of *Order.xml* as the value of the question-mark (*?*) parameter.

```
java util.FileUtil -A putstring -I "Order.xml" \
 -Q "insert into order_docs (id, order_doc)      \
   values (‘1’, new xml.order.OrderXml(?)) "   \
 –S "antibes:4000?user=sa"
```

➤ ***Note***

The constructor invocation **new xml.order.OrderXml(?)** validates the XML Order document.

#### *Inserting a generated order document on the server*

Executed on the server, the following SQL command generates an XML Order document from SQL data, and immediately inserts the generated XML document into the column of the *order_docs* table.

```
insert into order_docs (ID, order_doc)
select “2”,  new xml.order.OrderXml("990704","123",
"external", "")
```

### Accessing the elements of stored XML order documents

We have created a table named *order_docs* with a column named *order_doc*. The datatype of the *order_doc* column is **OrderXml**, which is a Java class that contains an XML Order document.

The **OrderXml** class contains several instance methods that let you reference and update elements of the XML Order document. They are described in “The OrderXml class for order documents” on page 124. This section uses these methods to update the Order document.

```
<?xml version="1.0"?>
 <!DOCTYPE Order SYSTEM  "Order.dtd">
 <Order>
    <Date>1999/07/04</Date>
    <CustomerId>123</CustomerId>
    <CustomerName>Acme Alpha</CustomerName>

    <Item>
        <ItemId> 987</ItemId>
        <ItemName>Coupler</ItemName>
        <Quantity>5</Quantity>
    </Item>

    <Item>
        <ItemId>654</ItemId>
        <ItemName>Connecter</ItemName>
        <Quantity unit="12">3</Quantity>
    </Item>

    <Item>
        <ItemId>579</ItemId>
        <ItemName>Clasp</ItemName>
        <Quantity>1</Quantity>
    </Item>
 </Order>
```

Each XML Order document has exactly one *Date*, *CustomerId*, and *CustomerName*, and zero or more *Item*s, each of which has an *ItemId*, *ItemName*, and *Quantity*.

### *Client access to order elements*

The **main( )** method of the **OrderElements** class is executed on the client. It reads the *Order.xml* file into a local variable, and constructs an *OrderXml* document from it. The method then extracts the “header” elements (*Date*, *CustomerId*, and*CustomerName*) and the elements of the first Item of the Order, prints those elements, and finally updates those elements of the Order with new values.

```
import java.io.*;
import util.*;
public class OrderElements {
     public static void main ( String[] args) {
          try{

          String xml = FileUtil.file2String("Order.xml");
          xml.order.OrderXml ox =
              new xml.order.OrderXml(xml);

          // Get the header elements
          String cname = ox.getOrderElement("CustomerName");
          String cid   = ox.getOrderElement("CustomerId");
          String date = ox.getOrderElement("Date");

          // Get the elements for item 1 (numbering from 0)
          String iName1 = ox.getItemElement(1, "ItemName");
          String iId1 = ox.getItemElement(1, "ItemId");
          String iQ1 = ox.getItemElement(1, "Quantity");
          String iU = ox.getItemAttribute(1,  "Quantity",
"unit");
          System.out.println("\nBEFORE UPDATE: ")
          System.out.println("\n   "+date+ "  "+ cname + " " +
              cid);
          System.out.println("\n   "+ iName1+" "+iId1+" "
              + iQ1 + " " + iU + "\n");

          //  Set the header elements
          ox.setOrderElement("CustomerName", "Best Bakery"
          ox.setOrderElement("CustomerId", "531");
          ox.setOrderElement("Date", "1999/07/31");

          //  Set the elements for item 1 (numbering from 0)
          ox.setItemElement(1, "ItemName", "Flange");
          ox.setItemElement(1, "ItemId", "777");
          ox.setItemElement(1, "Quantity","3");
          ox.setItemAttribute(1, "Quantity", "unit", "13");

          //Get the updated header elements
          cname = ox.getOrderElement("CustomerName");
          cid   = ox.getOrderElement("CustomerId");
          date = ox.getOrderElement("Date");
```

```
                //  Get the updated elements for item 1
                    (numbering from 0)
                iName1 = ox.getItemElement(1, "ItemName");
                iId1 = ox.getItemElement(1, "ItemId");
                iQ1 = ox.getItemElement(1, "Quantity");
                iU = ox.getItemAttribute(1,  "Quantity", "unit");

                System.out.println("\nAFTER UPDATE: ");
                System.out.println("\n   "+date+ "  "+ cname + " " +
                    cid);
                System.out.println("\n   "+ iName1+" "+iId1+" "
                    + iQ1 + " " + iU + "\n");

                //Copy the updated document to another file
                FileUtil.string2File("Order-updated.xml",
                    ox.getXmlText())

                } catch (Exception e) {
                System.out.println("Exception:");
                e.printStackTrace();
                }
           }
      }
```

After implementing the methods in OrderElements, the Order document stored in *Order-updated.xml* is:

```
<?xml version="1.0"?>
 <!DOCTYPE Order SYSTEM 'Order.dtd'>
 <Order>
    <Date>1999/07/31</Date>
    <CustomerId>531</CustomerId>
    <CustomerName>Best Bakery</CustomerName>
    <Item>
        <ItemId> 987</ItemId>
        <ItemName>Coupler</ItemName>
        <Quantity>5</Quantity>
    </Item>
    <Item>
        <ItemId>777</ItemId>
        <ItemName>Flange</ItemName>
        <Quantity unit="13">3</Quantity>
    </Item>
    <Item>
        <ItemId>579</ItemId
        <ItemName>Clasp</ItemName
        <Quantity>1</Quantity>

     </Item>
 </Order>
```

### Server access to order elements

The preceding example showed uses of get and set methods in a client environment. You can also call those methods in SQL statements in the server:

```
select order_doc>>getOrderElement("CustomerId"),
     order_doc>>getOrderElement("CustomerName"),
     order_doc>>getOrderElement("Date")
 from order_docs
```

```
select order_doc>>getItemElement(1, "ItemId"),
     order_doc>>getItemElement(1, "ItemName"),
     order_doc>>getItemElement(1, "Quantity"),
     order_doc>>getItemAttribute(1, "Quantity", "unit")
 from order_docs
```

```
update order_docs
 set order_doc = order_doc>>setItemElement(1, "ItemName",
     "Wrench")
```

```
update order_docs
 set order_doc = order_doc>>setItemElement(2, "ItemId", "967")
```

```
select order_doc>>getItemElement(1, "ItemName"),
     order_doc>>getItemElement(2, "ItemId")
 from order_docs
```

```
update order_docs
 set order_doc = order_doc>>setItemAttribute(2, "Quantity",
     "unit", "6")
```

```
select order_doc>>getItemAttribute(2, "Quantity", "unit")
 from order_docs
```

### Appending and deleting items in the XML document

The **Order** class provides methods for adding and removing items from the Order document.

You can append a new item to the Order document with the **appendItem( )** method, whose parameters specify *ItemId*, *ItemName*, *Quantity*, and *units* for the new item:

```
update order_docs
 set order_doc = order_doc>>appendItem("864",
     "Bracket", "3","12")
```

**appendItem( )** is a void method that modifies the instance. When you invoke such a method in an **update** statement, you reference it as shown, as if it were an Order-valued method that returns the updated item.

Delete an existing item from the Order document using **deleteItem( )**. The **deleteItem( )** parameter specifies the number of the item to be deleted. The numbering begins with zero, so the following command deletes the second item from the specified row.

```
update order_docs
    set order_doc = order_doc>>deleteItem(1)
 where id = “1”
```

## Using the hybrid storage technique

In the hybrid storage technique, you store the complete XML document in a SQL column and, at the same time, store elements of that document in separate columns. This technique often balances the advantages and disadvantages of element and document storage.

“Using the document storage technique” on page 133 demonstrates how to store the entire XML Order document in the single column *order_docs.order_doc*. Using document storage, you must reference and access the *CustomerId* element in this way:

```
select order_doc>>getOrderElement(“CustomerID”) from order_docs
where order_doc>>getOrderElement(“CustomerID”) > “222”
```

To access *CustomerId* more quickly and conveniently than with the method call, but without first decomposing the Order into SQL rows and columns:

1. Add a column to the *order_docs* table for the *customer_id*:

   ```
   alter table order_docs
       add  customer_id   varchar(5) null
   ```

2. Update that new column with extracted *customerId* values.

   ```
   update order_docs
    set customer_id =
       order_doc>>getOrderElement("CustomerId")
   ```

3. Now, you can reference *CustomerId* values directly:

   ```
   select customer_id from order_docs where
       customer_id > “222”
   ```

You can also define an index on the column.

➤ ***Note***

This technique does not synchronize the extracted *customer_id* column with the *CustomerId* element of the *order_doc* column if you update either value.

## A customizable example for different result sets

This section demonstrates how you can store XML documents or the data that they contain in an Adaptive Server database using the ResultSet class and its methods for handling result sets. You can customize the ResultSet class for your database application.

Contrast the ResultSet document type and the Order document type:

- The Order document type is a simplified example designed for a specific purchase-order application, and its Java methods are designed for a specific set of SQL tables for purchase order data. See "A simple example for a specific result set" on page 124.
- The ResultSet document type is designed to accommodate many kinds of SQL result sets, and the Java methods designed for it include parameters to accommodate different kinds of SQL queries.

For this example, you create and work with XML ResultSet documents that contain the same data as the XML Order documents.

First, create the *orders* table and its data:

```
create table orders
     (customer_id varchar(5) not null,
     order_date datetime not null,
     item_id varchar(5) not null,
     quantity int not null,
     unit smallint default 1)

insert into orders values ("123", "1999/05/07", "987", 5, 1)
insert into orders values ("123", "1999/05/07", "654", 3, 12)
insert into orders values ("123", "1999/05/07", "579", 1, 1)
```

Also, create the following SQL table to store complete XML ResultSet documents:

```
create table resultset_docs
    (id char(5),
    rs_doc xml.resultsets.ResultSetXml)
```

## The ResultSet Document Type

ResultSet documents consist of ResultSetMetaData followed by ResultSetData as shown in the following general form:

```
<?xml version="1.0"?>
    <!DOCTYPE ResultSet SYSTEM 'ResultSet.dtd'>
    <ResultSet>

<ResultSetMetaData>
   …
   </ResultSetMetaData>

<ResultSetData>
   …
   </ResultSetData>

</ResultSet>
```

The ResultSetMetaData portion of an XML ResultSet consists of the SQL metadata returned by the methods of the JDBC ResultSet class. The ResultSetMetaData for the example result set is:

```
<ResultSetMetaData
    getColumnCount="7">

    <ColumnMetaData
        getColumnDisplaySize="25"
        getColumnLabel="Date"
        getColumnName="Date"
        getColumnType="93"
        getPrecision="0"
        getScale="0"
        isAutoIncrement="false"
        isCurrency="false"
        isDefinitelyWritable="false"
        isNullable="false"
             isSigned="false" />

    <ColumnMetaData
        getColumnDisplaySize="5"
        getColumnLabel="CustomerId"
        getColumnName="CustomerId"
        getColumnType="12"
        getPrecision="0"
        getScale="0"
        isAutoIncrement="false"
        isCurrency="false"
        isDefinitelyWritable="false"
        isNullable="false"
             isSigned="false" />
```

```
<ColumnMetaData
    getColumnDisplaySize="50"
    getColumnLabel="CustomerName"
    getColumnName="CustomerName"
    getColumnType="12"
    getPrecision="0"
    getScale="0"
    isAutoIncrement="false"
    isCurrency="false"
    isDefinitelyWritable="false"
    isNullable="false"
         isSigned="false" />
<ColumnMetaData
    getColumnDisplaySize="5"
    getColumnLabel="ItemId"
    getColumnName="ItemId"
    getColumnType="12"
    getPrecision="0"
    getScale="0"
    isAutoIncrement="false"
    isCurrency="false"
    isDefinitelyWritable="false"
    isNullable="false"
         isSigned="false" />
<ColumnMetaData
    getColumnDisplaySize="20"
    getColumnLabel="ItemName"
    getColumnName="ItemName"
    getColumnType="12"
    getPrecision="0"
    getScale="0"
    isAutoIncrement="false"
    isCurrency="false"
    isDefinitelyWritable="false"
    isNullable="false"
         isSigned="false" />
```

```
            <ColumnMetaData
                getColumnDisplaySize="11"
                getColumnLabel="Quantity"
                getColumnName="Quantity"
                getColumnType="4"
                getPrecision="0"
                getScale="0"
                isAutoIncrement="false"
                isCurrency="false"
                isDefinitelyWritable="false"
                isNullable="false"
                     isSigned="true" />
            <ColumnMetaData
                getColumnDisplaySize="6"
                getColumnLabel="unit"
                getColumnName="unit"
                getColumnType="5"
                getPrecision="0"
                getScale="0"
                isAutoIncrement="false"
                isCurrency="false"
                isDefinitelyWritable="false"
                isNullable="false"
                     isSigned="true" />
        </ResultSetMetaData>
```

The names of the attributes of ColumnMetaData are simply the names of the methods of the JDBC ResultSetMetaData class, and the values of those attributes are the values returned by those methods.

The ResultSetData portion of an XML ResultSet document is a list of *Row* elements, each having a list of *Column* elements. The text value of a *Column* element is the value returned by the JDBC getString() method for the column. The ResultSetData for the example is:

```
<ResultSetData>
     <Row>
       <Column name="Date">1999-07-04 00:00:00.0</Column>
       <Column name="CustomerId">123</Column>
       <Column name="CustomerName">Acme Alpha</Column>
       <Column name="ItemId">987</Column>
       <Column name="ItemName">Coupler</Column>
       <Column name="Quantity">5</Column>
       <Column name="unit">1</Column>
     </Row>
     <Row>
       <Column name="Date">1999-07-04 00:00:00.0</Column>
       <Column name="CustomerId">123</Column>
```

```
        <Column name="CustomerName">Acme Alpha</Column>
        <Column name="ItemId">654</Column>
        <Column name="ItemName">Connecter</Column>
        <Column name="Quantity">3</Column>
        <Column name="unit">12</Column>
      </Row>
      <Row>
        <Column name="Date">1999-07-04 00:00:00.0</Column>
        <Column name="CustomerId">123</Column>
        <Column name="CustomerName">Acme Alpha</Column>
        <Column name="ItemId">579</Column>
        <Column name="ItemName">Clasp</Column>
        <Column name="Quantity">1</Column>
        <Column name="unit">1</Column>
      </Row>
    </ResultSetData>
  </ResultSet>
```

### The XML DTD for the ResultSetXml document type

The DTD for the XML ResultSet document type is:

```
<!ELEMENT ResultSet (ResultSetMetaData ,
     ResultSetData)>
 <!ELEMENT ResultSetMetaData (ColumnMetaData)+>
    <!ATTLIST ResultSetMetaData getColumnCount CDATA
     #IMPLIED>
 <!ELEMENT ColumnMetaData  EMPTY>
 <!ATTLIST ColumnMetaData
    getCatalogName CDATA #IMPLIED
    getColumnDisplaySize CDATA #IMPLIED
    getColumnLabel CDATA #IMPLIED
    getColumnName CDATA #IMPLIED
    getColumnType CDATA #REQUIRED
    getColumnTypeName CDATA #IMPLIED
    getPrecision CDATA #IMPLIED
    getScale CDATA #IMPLIED
    getSchemaName CDATA #IMPLIED
    getTablename CDATA #IMPLIED
    isAutoIncrement (true|false) #IMPLIED
    isCaseSensitive (true|false) #IMPLIED
    isCurrency (true|false) #IMPLIED
    isDefinitelyWritable (true|false) #IMPLIED
    isNullable (true|false) #IMPLIED
    isReadOnly (true|false) #IMPLIED
    isSearchable (true|false) #IMPLIED
    isSigned (true|false) #IMPLIED
    isWritable (true|false) #IMPLIED
    >
<!ELEMENT ResultSetData (Row)*>
<!ELEMENT Row (Column)+>
<!ELEMENT Column (#PCDATA)>
<!ATTLIST Column
    null (true | false) "false"
    name CDATA #IMPLIED
```

## The *ResultSetXml* class for result set documents

This section describes the **ResultSetXml** class that supports the ResultSet DTD.

The **ResultSetXml** class is similar to the **OrderXml** class. It is a subclass of the **JXml** class, which validates a document with the XML **ResultSet** DTD, and also provides methods for accessing and updating the elements of the contained XML **ResultSet** document.

- Constructor: ResultSetXml(String)

  Validates that the argument contains a valid XML **ResultSet** document and constructs a *ResultSetXml* object containing that document. For example, if *doc* is a Java String variable containing an XML **ResultSet** document, read from a file:

```
xml.resultset.ResultSetXml rsx =
    new xml.resultset.ResultSetXml(doc);
```

- Constructor: ResultSetXml(query, cdataColumns, colNames, dtdOption, server)

  The parameters are all String.

  The query parameter is any SQL query that returns a result set.

  The server parameter identifies the Adaptive Server on which to execute the query.

  - If you invoke the method in a client environment, specify the server name.

- If you invoke the method in a Adaptive Server (in a SQL statement or isql), specify either an empty string or the string "jdbc:default:connection," indicating that the query should be executed on the current Adaptive Server.

The method connects to the server, executes the query, retrieves the SQL result set, and constructs a ResultSetXml object with that result set.

The *cdataColumns* parameter indicates which columns should be XML CDATA sections. The *colNames* parameter indicates whether the resulting XML should specify "name" attributes in the "Column" elements. The dtd Option indicates whether the resulting XML should include the XML DTD for the ResultSet document type in-line, or reference it externally.

For example:

```
xml.resultset.ResultSetXml  rsx =
    new xml.resultset.ResultSetXml
    ("select 1 as 'a', 2 as 'b', 3 ", "none",
"yes",
    "external", "antibes:4000?user=sa");
```

This constructor call connects to the server specified in the last argument, evaluates the SQL query given in the first argument, and returns an XML ResultSet containing the data from the result set of the query. This simple SQL query does not reference a table. If the constructor is called in the Adaptive Server, then the server parameter should be an empty string or *jdbc:default:connection*, to indicate a connection to the current server.

- `String toSqlScript(resultTableName, columnPrefix, writeOption, goOption)`

  The parameters are all String.

  The method returns a SQL script with a create statement and a list of insert statements that re-create the result set data.

  The *resultTableName* parameter is the table name for the create and insert statements. (SQL result sets do not specify a table name because they could be derived from joins or unions.) The *columnPrefix* parameter is the prefix to use in generated column names, which are needed for unnamed columns in the result set. The *writeOption* parameter indicates whether the script is to include the create statement, the insert statements, or both. The *goOption* parameter indicates whether the script is to include the

**go** commands, which are required in **isql** and not supported in JDBC.

For example, if *rsx* is a Java variable of type **ResultSetXml**:

```
rsx>>toSqlScript("systypes_copy", "column_", "both", "yes")
```

- `String getColumn(int rowNumber, int columnNumber)`

  *rowNumber* is the index of a row in the result set; *columnNumber* is the index of a column of the result set. The method returns the text of the specified column.

  For example, if *rsx* is a Java variable of type **ResultSetXml**:

  ```
  select rsx>>getColumn(3, 4)
  ```

- `String getColumn(int rowNumber, String columnName)`

  *rowNumber* is the index of a row in the result set; *columnName* is the name of a column of the result set. The method returns the text of the specified column.

  For example, if *rsx* is a Java variable of type **ResultSetXml**:

  ```
  select rsx>>getColumn(3, "name")
  ```

- `void setColumn(int rowNumber, int columnNumber, newValue)`

  *rowNumber* and *columnNumber* are as described for **getColumn( )**. The method sets the text of the specified column to *newValue*.

  For example, if *rsx* is a Java variable of type **ResultSetXml**:

  ```
  select rsx = rsx>>setColumn(3, 4, "new value")
  ```

- `void setColumn(int rowNumber, String columnName, newValue)`

  *rowNumber* and *columnName* are as described for **getColumn( )**. The method sets the text of the specified column to *newValue*.

  For example, if *rsx* is a Java variable of type **ResultSetXml**:

  ```
  select rsx = rsx>>setColumn(3, "name", "new value")
  ```

- `Boolean allString(int columnNumber, String compOp, String comparand)`

  *columnNumber* is the index of a column of the result set. *compOp* is a SQL comparison operator (<, >, =, !=, <=, >=). *comparand* is a comparison value. The method returns a value indicating whether the specified comparison is true for all rows of the result set.

For example, if *rsx* is a Java variable of type ResultSetXml:

```
if rsx>>allString(3, “<”, “compare value”)…
```

This condition is true if in the result set represented by *rsx*, for all rows, the value of column 3 is less than “compare value.” This is a String comparison. Similar methods can be used for other data types.

- `Boolean someString(int columnNumber, String compOp, String comparand)`

  *columnNumber* is the index of a column of the result set. *compOp* is a SQL comparison operator (<, >, =, !=, <=, >=). *comparand* is a comparison value. The method returns a value indicating whether the specified comparison is true for some row of the result set.

  For example, if *rsx* is a Java variable of type ResultSetXml:

  ```
  if rsx>>someString(3, “<”, “compare value”) …
  ```

  This condition is true if in the result set represented by *rsx*, for some row, the value of column 3 is less than “compare value.”

## Using the element storage technique

This section uses the *orders* table to illustrate mapping between SQL data and XML ResultSet documents.

- In “Composing a ResultSet XML document from the SQL data” on page 147, we generate an XML ResultSet document from the SQL data. We assume that we are the originator of the XML ResultSet document. We use the resulting XML ResultSet document to describe the ResultSet DTD.
- In “Decomposing the XML ResultSet to SQL data” on page 149, we re-generate SQL data from the XML ResultSet document. We assume we are the *recipient* of the XML ResultSet document.

## Composing a ResultSet XML document from the SQL data

You can use Java methods to evaluate a given query and generate an XML result set with the query’s data. This example uses a constructor method of the ResultSetXml class. For example:

```
new xml.resultset.ResultSetXml
    (“select 1 as ‘a’, 2 as ‘b’, 3 ”, “none”,
    “yes”, “external”, “antibes:4000?user=sa”);
```

The method uses internal JDBC operations to execute the argument query, and then constructs the XML ResultSet for the query’s data.

We can invoke this constructor in a client or in the Adaptive Server:

- If you invoke the constructor in a client, specify a server parameter that identifies the Adaptive Server to be used when evaluating the query. The query is evaluated in the Adaptive Server, but the XML document is assembled in the client.
- If you invoke the constructor in the Adaptive Server, specify a null value or *jdbc:default:connection* for the server. The query is evaluated in the current server and the XML document is assembled there.

### *Generating a ResultSet in the client*

The main( ) method of the OrderResultSetClient class is invoked in a client environment. main( ) invokes the constructor of the ResultSetXml class to generate an XML ResultSet. The constructor executes the query, retrieves its metadata and data using JDBC ResultSet methods, and assembles an XML ResultSet document with the data.

```
import java.io.*;
import util.*;
public class OrderResultSetClient {
    public static void main (String[] args) {
       try{
           String orderQuery = "select order_date as Date,
                   c.customer_id as CustomerId, "
               + "customer_name as CustomerName, "
               + "o.item_id as ItemId, i.item_name as ItemName, "
               + "quantity as Quantity, o.unit as unit "
               + "from customers c, orders o, items i "
               + "where c.customer_id=o.customer_id and
                   o.item_id=i.item_id " ;
           xml.resultset.ResultSetXml rsx
                   = new xml.resultset.ResultSetXml(orderQuery,
                   "none", "yes", "external",
                   "antibes:4000?user=sa");
          FileUtil.string2File("OrderResultSet.xml",
                   rsx.getXmlText());
```

```
            } catch (Exception e) {
                System.out.println("Exception:");
                e.printStackTrace();
              }
          }
      }
```

### *Generating a ResultSet in Adaptive Server*

The following SQL script invokes the constructor of the ResultSetXml class in a server environment:

```
declare @rsx xml.resultset.ResultSetXml
select @rsx = new xml.resultset.ResultSetXml
    (“select 1 as ‘a’, 2 as ‘b’, 3 ”, “none”,
“yes”, “external”, “”);
insert into resultset_docs values (“1”, @rsx)
```

## Decomposing the XML ResultSet to SQL data

In this section, you decompose an existing **ResultSet** document to SQL data.

- In “Decomposing data from an XML order into SQL” on page 131, you invoke the **order2Sql( )** method of the **OrderXml** class to decompose an XML Order document into SQL data. **order2Sql( )** directly inserts the extracted data into a SQL table.
- In this example, the **toSqlScript( )** method of the **ResultSetXml** class decomposes an XML **ResultSet** document into SQL data. Instead of directly inserting extracted data into a SQL table, however, **toSqlScript( )** returns a SQL script with generated **insert** statements.

The two approaches are equivalent.

### *Decomposing the XML ResultSet Document in the client*

The **main( )** method of **ResultSetXml** is executed in a client environment. It copies the file *OrderResultSet.xml*, constructs a *ResultSetXml* object containing the contents of that file, and invokes the **toSqlScript( )** method of that object to generate a SQL script that re-creates the data of the result set. The method stores the SQL script in the file *order-resultset-copy.sql*.

```
import java.io.*;
import util.*;
public class ResultSet2Sql{
    public static void main (String[] args) {
       try{
          String xml =
FileUtil.file2String("OrderResultSet.xml");
               xml.resultset.ResultSetXml rsx
               = new xml.resultset.ResultSetXml(xml);
          String sqlScript
               = rsx.toSqlScript("orderresultset_copy", "col_",
                   "both", "no");
          FileUtil.string2File("order-resultset-copy.sql",
               sqlScript);
               util.ExecSql.statement(sqlScript,
               “antibes:4000?user=sa”);
       } catch (Exception e) {
          System.out.println("Exception:");
          e.printStackTrace();
       }
    }
 }
```

This is the SQL script generated by ResultSet2Sql.

```
set quoted_identifier on
 create table orderresultset_copy (
    Date datetime not null ,
    CustomerId varchar (5) not null ,
    CustomerName varchar (50) not null ,
    ItemId varchar (5) not null ,
    ItemName varchar (20) not null ,
    Quantity integer not null ,
    unit smallint not null
 )
insert into orderresultset_copy values (
    '1999-07-04 00:00:00.0',   '123',
      'Acme Alpha',    '987', 'Widget', 5,  1 )
 insert into orderresultset_copy values (
    '1999-07-04 00:00:00.0',  '123',
      'Acme Alpha',  '654',
      'Medium connecter',  3,  12 )
 insert into orderresultset_copy values (
      '1999-07-04 00:00:00.0',  '123',
      'Acme Alpha',  '579',  'Type 3 clasp',  1,  1 )
```

The SQL script includes the **set quoted_identifier on** command for those cases where the generated SQL uses quoted identifiers.

#### *Decomposing the XML ResultSet Document in Adaptive Server*

The following SQL script invokes the **toSqlScript()** method in Adaptive Server and then creates and populates a table with a copy of the result set data.

```
declare @rsx xml.resultset.ResultSetXml
select @rsx = rs_doc from resultset_docs where id=1
select @script =
@rsx>>toSqlScript(“resultset_copy”, “column_”,
“both”, “no”)
declare @I integer
select @I = util.ExecSql.statement(@script, “”)
```

## Using the document storage technique

This section shows examples of storing XML **ResultSet** documents in single SQL columns and techniques for referencing and updating the column elements.

### Storing an XML ResultSet document in a SQL column

The following SQL script generates an XML **ResultSet** document and stores it in a table:

```
declare @query java.lang.StringBuffer
select @query = new java.lang.StringBuffer()
 -- The following “appends” build up a SQL select statement in
      the @query variable
 -- We use a StringBuffer, and the append method, so that the
      @query can be as long as needed.
select @query>>append("select order_date as Date,
      c.customer_id as CustomerId, ")
select @query>>append("customer_name as CustomerName, ")
select @query>>append("o.item_id as ItemId, i.item_name as
      ItemName, ")
select @query>>append("quantity as Quantity, o.unit as unit " )
select @query>>append("from customers c, orders o, items i ")
select @query>>append("where c.customer_id=o.customer_id and
      o.item_id=i.item_id  ")

declare @rsx xml.resultset.ResultSetXml

select  @rsx = new xml.resultset.ResultSetXml
      (@query>>toString(), 'none', 'yes',  'external' , '')

insert into resultset_docs values("1",  @rsx)
```

### Accessing the columns of stored ResultSet documents

In “Storing an XML ResultSet document in a SQL column” on page 151 you inserted a complete XML ResultSet document into the *rs_doc* column of the *resultset_docs* table. In this section, use methods of the ResultSetXml class to reference and update a stored ResultSet.

#### *A client-side call*

The main( ) method of the ResultSetElements class is executed in a client environment. It copies the file *OrderResultSet.xml*, constructs a ResultSetXml document from it, and then accesses and updates the columns of the ResultSet.

```
import java.io.*;
import util.*;
public class ResultSetElements {
    public static void main ( String[] args) {
       try{
           String xml =
               FileUtil.file2String("OrderResultSet.xml");
           xml.resultset.ResultSetXml rsx
                 = new xml.resultset.ResultSetXml(xml);

           // Get the columns containing customer and date info
           String cname = rsx.getColumn(0, "CustomerName");
           String cid   = rsx.getColumn(0, "CustomerId");
           String date = rsx.getColumn(0, "Date");

           // Get the elements for item 1 (numbering from 0)
           String iName1 = rsx.getColumn(1, "ItemName");
           String iId1 = rsx.getColumn(1, "ItemId");
           String iQ1 = rsx.getColumn(1, "Quantity");
           String iU = rsx.getColumn(1,  "unit");

           System.out.println("\nBEFORE UPDATE: ");
           System.out.println("\n   "+date+ "  "+ cname + " " +
               cid);
           System.out.println("\n   "+ iName1+" "+iId1+" "
               + iQ1 + " " + iU + "\n");

           //  Set the elements for item 1 (numbering from 0)
           rsx.setColumn(1, "ItemName", "Flange");
           rsx.setColumn(1, "ItemId", "777");
           rsx.setColumn(1, "Quantity","3");
           rsx.setColumn(1,  "unit", "13");
```

```
            //  Get the updated elements for item 1 (numbering
                from 0) iName1 = rsx.getColumn(1, "ItemName");
            iId1 = rsx.getColumn(1, "ItemId");
            iQ1 = rsx.getColumn(1, "Quantity");
            iU = rsx.getColumn(1,  "unit");

            System.out.println("\nAFTER UPDATE: ");
            System.out.println("\n   "+date+ "  "+ cname + " " +
                cid);
            System.out.println("\n   "+ iName1+" "+iId1+" "
                + iQ1 + " " + iU + "\n");

           //  Copy the updated document to another file
            FileUtil.string2File("Order-updated.xml",
                rsx.getXmlText());

            } catch (Exception e) {
            System.out.println("Exception:");
            e.printStackTrace();
            }
        }
    }
```

The **FileUtil.string2File( )** method stores the updated **ResultSet** in the file *Order-updated.xml*. The **ResultSetMetaData** of the updated document is unchanged. The updated **ResultSetData** of the document is as follows with new values in the second item.

```
<ResultSetData>
     <Row>
       <Column name="Date">1999-07-04 00:00:00.0</Column>
       <Column name="CustomerId">123</Column>
       <Column name="CustomerName">Acme Alpha</Column>
       <Column name="ItemId">987</Column>
       <Column name="ItemName">Widget</Column>
       <Column name="Quantity">5</Column>
       <Column name="unit">1</Column>
     </Row>
     <Row>
       <Column name="Date">1999-07-04 00:00:00.0</Column>
       <Column name="CustomerId">123</Column>
       <Column name="CustomerName">Acme Alpha</Column>
       <Column name="ItemId">777</Column>
       <Column name="ItemName">Flange</Column>
       <Column name="Quantity">3</Column>
       <Column name="unit">13</Column>
     </Row>
     <Row>
```

```
            <Column name="Date">1999-07-04 00:00:00.0</Column>
            <Column name="CustomerId">123</Column>
            <Column name="CustomerName">Acme Alpha</Column>
            <Column name="ItemId">579</Column>
            <Column name="ItemName">Type 3 clasp</Column>
            <Column name="Quantity">1</Column>
            <Column name="unit">1</Column>
          </Row>
        </ResultSetData>
      </ResultSet>
```

### A server-side script

Using the SQL script in "Storing an XML ResultSet document in a SQL column" on page 151, you stored complete XML **ResultSet** documents in the *rs_doc* column of the *resultset_docs* table. The following SQL commands, executed in a server environment, reference and update the columns contained in those documents.

You can select columns by name or by number:

- Select the columns of row 1, specifying columns by name:

```
select rs_doc>>getColumn(1, "Date"),
    rs_doc>>getColumn(1, "CustomerId"),
    rs_doc>>getColumn(1, "CustomerName"),
    rs_doc>>getColumn(1, "ItemId"),
    rs_doc>>getColumn(1, "ItemName"),
    rs_doc>>getColumn(1, "Quantity"),
    rs_doc>>getColumn(1,  "unit")
  from resultset_docs
```

- Select the columns of row 1, specifying columns by number:

```
select rs_doc>>getColumn(1, 0),
    rs_doc>>getColumn(1, 1),
    rs_doc>>getColumn(1, 2),
    rs_doc>>getColumn(1, 3),
    rs_doc>>getColumn(1, 4),
    rs_doc>>getColumn(1, 5),
    rs_doc>>getColumn(1,  6)
 from resultset_docs
```

Specify some nonexisting columns and rows. Those references return null values.

```
Select rs_doc>>getcolumn(1, "itemid"),
    rs_doc>>getcolumn(1, "xxx"),
    rs_doc>>getcolumn(1, "Quantity"),
    rs_doc>>getcolumn(99,  "unit"),
    rs_doc>>getColumn(1, 876)
 from resultset_docs
```

Update columns in the stored ResultSet document:

```
update resultset_docs
 set rs_doc = rs_doc>>setColumn(1, "ItemName", "Wrench")
 where id=”1”

update resultset_docs
 set rs_doc = rs_doc>>setColumn(1, "ItemId", "967")
 where id=”1”

update resultset_docs
 set rs_doc = rs_doc>>setColumn(1,  "unit", "6")
 where id=”1”

select rs_doc>>getColumn(1, "ItemName"),
     rs_doc>>getColumn(1, "ItemId"),
     rs_doc>>getColumn(1,  "unit")
 from resultset_docs
 where id=”1”
```

## Quantified comparisons in stored ResultSet documents

**ResultSetXml** contains two methods, **allString( )** and **someString( )**, for quantified searches on columns of a *ResultSetXML* document. To illustrate these two methods, first create some example rows in the *order_results* table.

The *order_results* table has been initialized with one row, whose *id* = “1” and whose *rs_doc* column contains the original Order used in all examples.

The following statements copy that row twice, assigning *id* values of “2” and “3” to the new rows. The *order_results* table now has three rows, with *id* column values of “1,” “2,” and “3” and the original Order.

```
insert into resultset_docs(id, rs_doc) select "2", rs_doc
 from resultset_docs where id="1"

insert into resultset_docs (id, rs_doc) select "3", rs_doc
 from resultset_docs where id="1"
```

The following statements modify the row with an *id* column value of “1” so that all three items have an *ItemId* of “100”:

```
update resultset_docs
 set rs_doc = rs_doc>>setColumn(0, "ItemId", "100")
 where id="1"
```

```
update resultset_docs
 set rs_doc = rs_doc>>setColumn(1, "ItemId", "110")
 where id="1"
```

```
update resultset_docs
 set rs_doc = rs_doc>>setColumn(2, "ItemId", "120")
 where id="1"
```

The following **update** statement modifies the row with *id* = “3” so that its second item (from 0) has an *ItemId* of “999”:

```
update resultset_docs
 set rs_doc = rs_doc>>setColumn(2, "ItemId", "999")
 where id="3"
```

The following **select** statement displays the *id* column and the three *ItemId* values for each row:

```
select id, rs_doc>>getColumn(0, "ItemId"),
    rs_doc>>getColumn(1, "ItemId"),
    rs_doc>>getColumn(2, "ItemId")
 from resultset_docs
```

The results of the **select** are:

```
1     100      110         120
2     987      654         579
3     987      654         999
```

Note the following:

- The row with *id* of “2” is the original Order data.
- The row with *id* of “1” has been modified so that *all ItemIds* for that row are less than “200.”
- The row with *id* of “3” has been modified so that *some ItemId* for that row is greater than or equal to “9999,”

The following expresses these quantifications with the **allString( )** and **someString( )** methods:

```
select id, rs_doc>>allString(3, "<", "200") as
“ALL test”
 from resultset_docs
```

```
select id, rs_doc>>someString(3, ">=", "999") as
“SOME test”
 from resultset_docs
```

```
select id as “id for ALL test” from resultset_docs
 where rs_doc>>allString(3, "<",
"200")>>booleanValue() = 1
```

```
select id as “id for SOME test” from resultset_docs
 where rs_doc>>someString(3, ">=",
"999")>>booleanValue() = 1
```

The first two statements show the quantifier in the **select** list and give these results:

| ID | “all” test | “some” test |
|---|---|---|
| 1 | true | false |
| 2 | false | false |
| 3 | false | true |

The last two statements show the quantifier in the **where** clause and give these results:

- ID for “all” test = “3”
- ID for “some” test = “1”

In the examples with the quantifier method in the **where** clause, note that:

- The **where** clause examples compare the method results with an integer value of 1. SQL does not support the Boolean data type as a function value, but instead treats Boolean as equivalent to integer values 1 and 0, for true and false.
- The **where** clause examples use the **booleanValue( )** method. The **allString( )** and **someString( )** methods return type java.lang.Boolean, which is not compatible with SQL integer. The Java **booleanValue( )** method returns the simple Boolean value from the Boolean object, which is compatible with SQL integer. This behavior is a result of merging the SQL and Java type systems.

The quantifier methods return java.lang.Boolean instead of simply Java boolean so that they can return null when the column is out of range, which is consistent with the SQL treatment of out-of-range conditions.

The following statements show quantifier references that specify column 33, which does not exist in the data:

```
select id, rs_doc>>allString(33, "<", "200") as
“ALL test”
 from resultset_docs
```

```
select id as “id for ALL test” from resultset_docs
 where rs_doc>>allString(33, "<",
"200")>>booleanValue() = 1
```

| Id | “all” test |
|---|---|
| 1 | NULL |
| 2 | NULL |
| 3 | NULL |

The ID for the “all” test = (empty).

## Using the hybrid storage technique

For faster and easier access to the *CustomerID* element, add a new *customer_id* column to the *resultset_docs* table, and populate it with extracted *CustomerID* elements:

```
alter table resultset_docs
    add  customer_id   varchar(5) null

update resultset_docs
    set customer_id = rs_doc>>getColumn(1,
"CustomerId")
```

## XML ResultSet documents: invalid XML characters

This section describes two techniques for dealing with XML markup characters in the result set.

- When data values contain XML markup characters, you can enclose these values in a CDATA section.
- When column names are quoted identifiers that contain XML markup characters, you can substitute the quotes and markup characters with XML entity symbols.

Each technique is described in the following sections.

### Using CDATA sections

The *cdata* parameter of the **ResultSetXml** constructor indicates which (if any) columns of the SQL result set contain character data to be bracketed as CDATA sections in the output XML. The *cdata* parameter can be “all,” “none,” or a string of zero or one characters, where a “1” in the I-th position indicates that the I-th column should be bracketed as a CDATA section.

For example, create the table *cdata* in which data values in columns 2, 3, and 4 contain XML markup characters that must be bracketed as CDATA section in the output:

```
create table cdata (
    id int,
    a varchar(250),
    b varchar(250),
    c varchar(250)
    )
 go
```

```
insert into cdata values (
    1,
    "<p>some samples:</p><ol><li>first</li>
        <li>second</li></ol>",
    "x > y || w  & z",
    "x > y || w & z"
    )
```

The following SQL statement generates an XML **ResultSet** document for this table, specifying a value “0111” for the *cdata* parameter.

```
insert into resultsets (id, rs)
    values ("2", new xml.resultset.ResultSetXml(
        "select * from cdata", '0111', 'yes',
'external', ''))
```

This SQL statement generates a SQL script for that XML **ResultSet**:

```
update resultsets
    set script =
        rs>>toSqlScript("markup_col_names",
        "col_", "both", "yes")
    where id="2"
```

The following utility calls retrieve the XML **ResultSet** and its SQL script:

```
java util.FileUtil -S "$SERVER" -A getstring -O cdata.xml \
     -Q "select new util.StringWrap(rs>>getXmlText()) from
          resultsets where id='2'"
```

```
java util.FileUtil -S "$SERVER" -A getstring -O cdata.script\
     -Q "select new util.StringWrap(script) from resultsets
          where id='2'"
```

This is the XML **ResultSet**:

```
<?xml version="1.0"?>
<!DOCTYPE ResultSet SYSTEM 'ResultSet.dtd'>
```

```
<ResultSet>
   <ResultSetMetaData getColumnCount="4">
     <ColumnMetaData getColumnDisplaySize="11" getColumnLabel="id"
getColumnName="id" getColumnType="4" getPrecision="0" getScale="0"
isAutoIncrement="false" isCurrency="false"
isDefinitelyWritable="false" isNullable="false" isSigned="true" />
     <ColumnMetaData getColumnDisplaySize="250" getColumnLabel="a"
getColumnName="a" getColumnType="12" getPrecision="0" getScale="0"
isAutoIncrement="false" isCurrency="false"
isDefinitelyWritable="false" isNullable="false" isSigned="false" />
     <ColumnMetaData getColumnDisplaySize="250" getColumnLabel="b"
getColumnName="b" getColumnType="12" getPrecision="0" getScale="0"
isAutoIncrement="false" isCurrency="false"
isDefinitelyWritable="false" isNullable="false" isSigned="false" />
     <ColumnMetaData getColumnDisplaySize="250" getColumnLabel="c"
getColumnName="c" getColumnType="12" getPrecision="0" getScale="0"
isAutoIncrement="false" isCurrency="false"
isDefinitelyWritable="false" isNullable="false" isSigned="false" />
   </ResultSetMetaData>
   <ResultSetData>
     <Row>
       <Column name="id">1</Column>
       <Column name="a">
            <![CDATA[<p>some samples:
                 </p><ol><li>first</li><li>second</li></ol>]]>
       </Column>
       <Column name="b">
            <![CDATA[x > y || w  & z]]>
       </Column>
       <Column name="c">
            <![CDATA[x > y || w & z]]>
       </Column>
     </Row>
   </ResultSetData>
 </ResultSet>
```

This is the SQL script:

```
set quoted_identifier on
 create table markup_col_names (
    id integer not null ,
    a varchar (250) not null ,
    b varchar (250) not null ,
    c varchar (250) not null
 )
 insert into markup_col_names values (
    1,
    '<p>some samples:</p><ol><li>first</li><li>second</li></ol>',
    'x > y || w  & z',
    'x > y || w & z'
    )
```

### Column names

The XML generated for a SQL result set specifies the column names of the result set in the **ResultSetMetaData** section and in the **ResultSetData** section.

The following SQL **select** specifies a result set:

```
select 1 as “A>2”, 2 as “B  & 3”, 3 as “A<<b”, 4 as
     ”D “”or”” e”
```

The result set has a single row, whose column values are 1, 2, 3, and 4. The names of those columns are quoted identifiers that contain XML markup characters.

Since the **ResultSetXml** document for such a result set specifies the column names in XML attributes, the quotation marks and XML markup characters in those names must be replaced with XML entity symbols.

This problem cannot be handled with CDATA sections, since you cannot use CDATA sections in attribute values.

The following is a SQL script that generates the **ResultSetXml** document for the result set, then generates the SQL script for that **ResultSetXml** document.

Store the generated **ResultSetXml** document in the following table:

```
create table resultsets
 (id char(5) unique,
 rs xml.resultsets.ResultSetXml null,
 script java.lang.String null)
```

The following SQL statement generates the XML **ResultSet** document and stores it into the *resultsets* table:

```
insert into resultsets (id, rs)
    values ("1", new xml.resultsets.ResultSetXml(
        "select 1 as ""A > 2"", 2 as ""b  & 3"",
              3 as ""a<<b"", 4 as ""d """"or""""
e"" ",
                'none', 'yes', 'external', '' ))
```

This SQL statement generates the SQL script for the XML **ResultSet**:

```
update resultsets
   set script =
rs>>toSqlScript("markup_col_names", "col_",
    "create", "yes")
   where id="1"
```

The following utility calls retrieve the XML ResultSet and its SQL script into client files *cdata.xml* and *cdata.script*.

```
java util.FileUtil -S "$SERVER" -A getstring -O cdata.xml \
      -Q "select new util.StringWrap(rs>>getXmlText()) from
            resultsets where id='2'"
```

```
java util.FileUtil -S "$SERVER" -A getstring -O cdata.script\
      -Q "select new util.StringWrap(script) from resultsets
            where id='2'"
```

The XML ResultSet document for the CDATA example is:

```
<?xml version="1.0"?>

<!DOCTYPE ResultSet SYSTEM 'ResultSet.dtd'>

<ResultSet>
   <ResultSetMetaData getColumnCount="4">
     <ColumnMetaData getColumnDisplaySize="11" getColumnLabel="A > 2"
getColumnName="A > 2" getColumnType="4" getPrecision="0" getScale="0"
isAutoIncrement="false" isCurrency="false"
isDefinitelyWritable="false" isNullable="false" isSigned="true" />
     <ColumnMetaData getColumnDisplaySize="11" getColumnLabel="b & 3"
getColumnName="b & 3" getColumnType="4" getPrecision="0" getScale="0"
isAutoIncrement="false" isCurrency="false"
isDefinitelyWritable="false" isNullable="false" isSigned="true" />
     <ColumnMetaData getColumnDisplaySize="11" getColumnLabel="a<<b"
getColumnName="a<<b" getColumnType="4" getPrecision="0" getScale="0"
isAutoIncrement="false" isCurrency="false"
isDefinitelyWritable="false" isNullable="false" isSigned="true" />
     <ColumnMetaData getColumnDisplaySize="11" getColumnLabel="d 'or' e"
getColumnName="d 'or' e" getColumnType="4" getPrecision="0" getScale="0"
isAutoIncrement="false" isCurrency="false"
isDefinitelyWritable="false" isNullable="false" isSigned="true" />
   </ResultSetMetaData>
   <ResultSetData>
     <Row>
       <Column name="A > 2">1</Column>
       <Column name="b & 3">2</Column>
       <Column name="a<<b">3</Column>
       <Column name="d 'or' e">4</Column>
     </Row>
   </ResultSetData>
 </ResultSet>
```

The following is the output SQL script for the CDATA example:

```
set quoted_identifier on
 create table markup_col_names (
    "A > 2" integer not null ,
    "b  & 3" integer not null ,
    "a<<b" integer not null ,
    "d ""or"" e" integer not null
```

## XML methods

This section describes the XML methods that are provided with Adaptive Server.

## parse(String xmlDoc)

### Description

Takes a Java string as an argument and returns **SybXmlStream**. You can use this to query a document using XQL.

### Syntax

```
parse(java_string xml_document)
```

Where:

- *String* is a Java string.
- *xml_document* is the XML document in which the string is located.

### Usage

- **parse()** returns *SybXmlStream*.
- If a DTD is supplied, the parser also validates the command.
- The parser does not:
  - Parse any external DTDs
  - Perform any external links (for example, XLinks)
  - Navigate through IDREFs

## parse(InputStream xml_document)

### Description

Takes an **InputStream** as an argument and returns **SybXmlStream**. You can use this to query a document using XQL.

### Syntax

**parse(*Input_Stream xml_document*)**

Where:

- *InputStream* is an input stream.
- *xml_document* is the XML document from which the input stream originates.

### Usage

- **parse()** returns **SybXmlStream**.
- If a DTD is supplied, the parser also validates the command.
- The parser does not:
  - Parse any external DTDs
  - Perform any external links (for example, XLinks)
  - Navigate through IDREFs

## query(String query, String xmlDoc)

### Description

Queries an XML document. Uses the XML document as the input argument.

### Syntax

**`query(String query, String xmlDoc)`**

Where:

- *String query* is an XQL query encapsulated in **Java.lang.string.**
- *String xmldoc* is a **Java.io.InputStream** which can be formed by files, URLs, and so on.

### Examples

The following returns the result as a Java string.

```
String result= Xql.query("/bookstore/book/author",
"<xml>...</xml>");
```

### Usage

Returns a Java string, which is a well-formed XML document.

## query(String query, InputStream xmlDoc)

### Description

Queries an XML document using an input stream as the second argument.

### Syntax

**`query(String query, InputStream xmlDoc)`**

Where:

- *String query* is an XQL query encapsulated in **Java.lang.string.**
- *InputStream xmlDoc* is a **Java.IO.InputStream** which can be formed from XML documents, URLs, and so on.

### Examples

The following queries an XML document stored in Adaptive Server.

```
FileInputStream xmlStream = new FileInputStream("doc.xml");
String result = Xql.query("/bookstore/book/author", xmlStream);
```

The following example queries an XML document on the Web using a URL as the search argument:

```
URL xmlURL = new URL("http://mywebsite/doc.xml");
String result = Xql.query("/bookstore/book/author",
xmlURL.openStream());
```

### Usage

Returns a Java string. The result of the query is encapsulated in a string as a well-formed XML document.

## query(String query, SybXmlStream xmlDoc)

### Description

Queries the XML document using a parsed XML document as the second argument.

### Syntax

**`query(String query, SybXmlStream xmlDoc)`**

Where:

- *String query* is the string you are searching for.
- *SybXmlStream* is a result provided by an earlier **parse()**. This is a Sybase-proprietary format that provides faster performance.

### Examples

The following example creates an XML stream by first parsing it and then using the result to execute an XQL query.

```
SybXmlStream xmlStream = Xql.parse("<xml>..</xml>);
String result = Xql.query("/bookstore/book/author",xmlStream);
```

### Usage

Returns a Java string where the result of the query is a encapsulated in a well-formed XML document.

# query(String query, JXml jxml)

## Description

Queries an XML document stored in a JXML format.

## Syntax

**`query(String query, JXml jxml)`**

Where:

- *String query* is the string for which you are searching.
- *JXml jxml* is an object created using the Adaptive Server object code.

## Examples

The following example creates a JXML object and then runs a query by extracting the XML document from this object.

```
JXml xDoc = new JXml("<xml>...</xml>");;
String result = Xql.query("/bookstore/book/author",             xDoc);
```

## Usage

Allows you to execute a query on an JXML document using XQL.

## SybXmlStream

### Description

Defines a method.

### Syntax

```
SybXmlStream {method}
```

### Examples

The following is the Java implementation of **SybXmlStream**.

```
public interface SybXmlStream {
/*
** seek method
** @param  pos - position in the stream where the pointer should
set to
** @return none
*/
void seek(long pos) throws IOException;
}
```

### Usage

- Used for storing XML data, which will be queried at a later time.
- **SybXmlStream** is a wrapper around streaming interfaces.
- **SybXmlStream** requires the implementation to include the **seek()** method

## SybMemXmlStream

### Description

Holds the parsed XML document in main memory.

### Syntax

**SybMemXmlStream**

### Examples

The following is the Java implementation of **SybMemXmlStream**:

```
 public class SybMemXmlStream extends ByteArrayInputStream
implements SybXmlStream {
..
}
```

### Usage

- The **parse()** method returns an instance of **SybMemXmlStream** after parsing an XML document.
- **SybmemXmlStream** is an implementation of **SybXmlStream** that retains the whole stream in main memory.

## SybFileXmlStream

### Description

Allows you to query a file in which you have stored a parsed XML document.

### Syntax

**SybFileXmlStream {*file_name*}**

Where *file_name* is the name of the file in which you stored the parsed XML document.

### Examples

1. **The following is the Java implementation of SybFileXmlStream:**

```
public class SybFileXmlStream extends InputStream
                                            implements SybXmlStream{
...
private RandomAccessFile raFile;
}
```

2. **In the following, a member of the RandomAccessFile reads a file and positions the data stream:**

```
SybXmlStream xis = Xql.parse("<xml>..</xml>");
FileOutputStream ofs = new FileOutputStream("xml.data");
((SybMemXmlStream)xis).writeToFile(ofs);

SybXmlStream is = new SybFileXmlStream("xml.data");
String result =
Xql.query("/bookstore/book/author",                      is);
```

### Usage

Is a wrapper around **RandomAccessFile**, which provides the **seek()** method on a file. The following are the steps for using **SybFileXmlStream**:

- Parse the XML document
- Store the parsed XML in a file.
- Access the class with **SybFileXmlStream**.

## result

### Description

The result set returned by the **query()** method as a Java string. The result string is a well-formed XML document.

### Syntax

```
result = string
```

Where *string* is a well-formed XML document.

### Examples

```
String result ;
If ( (result = Xql.query("/bookstore/book/author",<xml>..</xml))
                                                                   ! =
Xql.EmptyResult)
{
                              return result;
}
```

### Usage

- The result set is returned as a Java string.
- If the result set is empty, **result ()** returns <xql_result></xql_result>.
- **result()** includes a static variable, *xql.EmptyResult*, which contains an empty XML result document and can be used for testing the empty result returned from an XQL method. For example:

```
String result ;
If ( (result = Xql.query("/bookstore/book/author",<xml>..</xml))
! = Xql.EmptyResult)
{
return result;
}
```

# 6 Unicode Enhancements

Sybase believes that Unicode, with its ability to represent nearly all of the worlds written languages in a single character set, will pave the way for multilingual interoperability in the 21st century. With this in mind, we continue to enhance our Unicode features in ASE.

## New Datatypes Added

In this release, two new datatypes using the UTF-16 encoding of the Unicode character set have been added. The new *unichar* and *univarchar* datatypes are independent of the existing *char* and *varchar* dataypes, but mirror their behavior. *Unichar* is a fixed-width, non-nullable data type (like *char*) and *univarchar* is a variable-width, nullable data type (like *varchar*). The set of built-in string functions that operate on *char* and *varchar,* will also operate on *unichar* and *univarchar.*

Note however, that unlike the existing *char* and *varchar*, the new *unichar* and *univarchar* only store UTF-16 characters and have no connection to the default character set ID or default sort order ID ASE configuration option. To use these new data types, the default character set for the server *must* be set to UTF-8.

The main advantage of these new datatypes is efficiency. The UTF-16 character types are approximately 33% more space efficient than UTF-8 for Asian characters.

Each *unichar/univarchar* character requires two bytes of storage; a *unichar/univarchar* column consists of 16-bit Unicode values. The following command creates a table with one *unichar* column for 10 Unicode values requiring 20 bytes of storage:

```
Create table unitbl (unicol unichar (10))
```

The length of a *unchar/univarchar* column is limited by the size of a data page in ASE, just as in *char/varchar* columns.

## New Configuration Options

Three new configuration options have been added for this new feature:

- enable surrogate processing

- enable unicode normalization
- default unicode sort order

The following paragraphs contain information on these options

## Enable Surrogate Processing

Unicode surrogate pairs use the storage of two 16-bit Uncode values (four bytes). Keep this in mind when declaring columns intended to store Unicode surrogate pairs. By default, ASE automatically manages surrogates without splitting the pair. To disable surrogate handling in ASE, use the following command:

**Sp_configure “enable surrogate processing”,0**

Once disabled, all Unicode pairs are ignored and surrogate pairs may be split during processing.

## Enable Unicode Normalization

By default, all Unicode data are normalized before being stored into a data page. The following command can be used to disable Unicode normalization in ASE:

**SP_configure “enable unicode normalization”, 0**

If normalization is disabled, it cannot be re-enabled. This prevents normalized and non-normalized data from mixing.

➤ ***Note***

We strongly advise you not to change this option once *unichar*/*univarchar* data has been entered into ASE.

## Default Unicode Sortorder

Collating *unichar* and *univarchar* data is controlled by a new configuration option, which defines the default Unicode sort order. The default Unicode sort order option also affects the indices of *unichar/univarchar* columns. This is analogous to the default sort order ID option affecting the indices for *char/varchar* columns.

To define the default Unicode sort order, the command is:

**sp_configure, default unicode sortorder”, 0, “sort name”**

Valid “sort names” are as follows:

**Table 6-1:**

| Name | Description |
|---|---|
| defaultML | default Unicode ML ordering |
| binary | Default binary ordering |
| thaidict | Thai dictionary ordering |
| scandict | Scandinavian dictionary ordering |
| scannocp | Scandinavian case insensitive ordering |
| dict | English / French / German dictionary ordering |
| nocase | English / French / German case insensitive ordering |
| noaccent | English / French / German accent insensitive ordering |
| espdict | Spanish dictionary ordering |
| espnocs | Spanish case insensitive ordering |
| espnoac | Spanish accent insensitive ordering |
| rusdict | Russian dictionary ordering |
| rusnocs | Russian case insensitive ordering |
| cyrdict | Cyrillic dictionary ordering |
| cyrnocs | Cyrillic case insensitive ordering |
| elldict | Greek dictionary ordering |
| hundict | Hungarian dictionary ordering |
| hunnoac | Hungarian accent insensitive ordering |
| hunnocs | Hungarian case insensitive ordering |
| turkdict | Turkish dictionary ordering |
| turknoac | Turkish accent insensitive ordering |
| turknocs | Turkish case insensitive ordering |
| sjisbin | Japanese SJIS binary ordering |
| iso14651 | ISO 14651 ordering |
| eucjisbin | Japanese eucjis |
| gb2312bin | Chinese gb2312 |

**Table 6-1:**

| Name | Description |
|---|---|
| cp932msbin | Japanese cp932 |
| b165bin | Chinese b165 |
| euckcsbin | Korean euckcs |
| utf8bin | matches unicode UTF-8 binary sort order |

## Functions Supporting the New Datatypes

The existing built-in functions that support the new *unichar/univarchar* datatypes include the following:

- ascii
- charindex
- char_length (surrogate pairs are counted as two characters)
- col_length
- compare
- convert
- count
- datalength
- difference
- isnull
- lower
- ltrim (blanks are defined to be U+ 0020 only)
- max
- min
- patindex
- replicate
- reverse
- right
- rtrim (blanks are defined to be U+0020 only)
- soundex (only works for simple Roman characters)
- sortkey

- stuff
- substring
- upper

### New Functions Added

Four new functions have been added to support the new *unichar* and *univarchar* datatypes:

- uchar_expr **to_unichar (integer_expr)**

Analogous to the *char* built-in function for *char/varchar* data. This function returns a *unichar* expression having the value of the integer expression. A single Unicode value is returned for an integer expression in the range of 0x0000 to 0xFFFF. If the integer expression is in the surrogate range (0xD800 to 0xDFFF), an exception is raised.

- int **uscalar (uchar_expr)**

Analogous to the ascii built-in function for *char/varchar* data. This function returns the Unicode scalar value for the first Unicode character in an expression. If the first character is not the high order half of a surrogate pair, the value will be in the range of 0x0000 to 0xFFFF. If the first character is the high order half of a surrogate pair, then a second value must be a low-order half. The return value in this case will be in the range of 0x10000 to 0x10FFFF. If this function is called on an unmatched surrogate half, an exception is raised.

- int **uhighsurr (uchar_expr, start)**

Returns one if the Unicode value at position “start” is the high half of a surrogate pair. Otherwise, returns zero.

- int **ulowsurr(uchar_expr, start)**

Returns one if the Unicode value at position “start” is the low half of a surrogate pair. Otherwise, returns zero.

## String Concatenation

If either “*str1*” or “*str2*” is a *unichar/univarchar* datatype, the result is a *unichar/univarchar* value.

## Comparison Operators

The following comparison operators have been updated to handle *unichar/univarchar* exactly like *char/varchar.* Note especially that blanks (U+0020) are ignored in comparisons, just as they are for *char/varchar.*

0=(equal to)

0>(greater than)

0<(less than)

0>+(greater than or equal to)

0<+(less than or equal to)

0<>(not equal to)

0!=(not equal to)

0!>(not greater than)

0!<(not less than)

## Relational Expressions

All relational expressions involving at least one expression of type *unichar/univarchar* will be based on the default Unicode sort order. If one expression is of type *unichar/univarchar* and the other is of type *char/varchar,* the latter will be implicitly converted to *unichar/univarchar.*

# 7 Unicode Enhancements

## Using bcp While Converting Data

You can copy data encoded in a different character set than the server, into the server. To do this, bcp first converts the data into the character set of the server, then copies the data into a table.

For example, assume the data in datafile.dat is encoded in CP 1252 and the server character set is CP 850. As the data is copied into the db1..table1 table, it is converted from CP 1252 to CP 850. The -J flag is used to indicate the character set of the datafile.dat file:

**bcp db1..table1 in datafile.dat -c -Jcp1252 -Usa -P -Sserver**

The bcp copy is successful if the data length does not change during the conversion.

### Copying Data That Changes Length

Note that bcp cannot correctly convert and copy data into a table if the conversion results in a change of data length. If this occurs, the process is aborted and this message appears:

```
bcp insert operation is disabled when LONGCHAR
capability is turned on and data size is changing
between client and server character sets, since
bcp does not support LONGCHAR yet.
```

Suppose you are copying data from server1, whose character set is Shift-JIS, a two-byte character encoding, into server2 whose character set is UTF-8. When Shift-JIS data is converted to UTF-8, it becomes three bytes. In this situation the bcp copy would fail.

To work around this bcp limitation, do the following:

1. Copy the data out of server1 using the-J flag to specify the character set of server2, as follows:

   **bcp old_database..old_table out datafile.dat -c -Jutf8 -Usa -P -Sserver1**

   bcp converts the data from the character set of server1 (Shift-JIS) to the character set of server2 (UTF-8) into the datafile.dat file.

2. Copy the data into server2, as follows:

   bcp new_database..new_table in datafile.dat -c -Usa -P -Sserver2

Now when you copy the data into server2, it is already encoded in the server's default character set and no data conversion is necessary. The bcp copy will succeed.

# 8 Using union operators in select statements

This section describes the changes in Adaptive Server version 12.5 for enabling **union** operators in **select** statements.

In earlier versions of Adaptive Server, **union** operators in **select** statements that define views were not supported. Adaptive Server version 12.5 removes this restriction.

You might use this feature, for example, to split a large table into subtables. The data can be partitioned between the subtables based on ranges of data values in one of the columns. You can then define a view that uses **union all** to combine selects of all the subtables into a single result set. You can then issue **select** statements on the view containing the union.

```
create view year1998Sales
as
select * from Feb1998Sales
union all
select * from Apr1998Sales
union all
select * from May1998Sales
(More to come)
select * from June998Sales
union all
select * from Jul1998Sales
union all
select * from Aug1998Sales
union all
select * from Sep1998Sales
union all
...
```

In this example the view *year1998Sales* consists of 12 subtables recording sales for 1998, one for each month. It is easier to maintain and update the subtables independently than to update the entire table. Fewer rows are accessed and users can still access other subtables which are not being updated.

You can create views that reference Oracle, DB2, and Informix tables, for example, as well as local tables:

```
create view partitioned _view as
    select * from db2_table
    union
    select * from oracle_table
    union
    select * from informix_table
    union
    select * from local_table
go
select * frompartitioned_view
go
```

A **union** operator within a **create view** statement can also be used on proxy (remote) tables.

## Limitations

- The limit on the number of tables in a union view is 256.
- You cannot update or insert into a view whose **select** statement contains the **union** operator.
- You cannot **delete** from a view whose **select** statement contains the **union** operator.
- The **select** statement that contains the union view cannot include **order by** or **compute** clauses, or the keyword **into.**

## Error Messages

197 - cannot use check option in union view.

4427 - union view cannot be updated.

# 9 Component Integration Services

Release 12.5 of Adaptive Server includes many enhancements to Component Integration Services.

Component Integration Services is fully compatible with the new features of Adaptive Servicer Enterprise documented in this book.

- distributed query optimization
- transaction management
- extended data access
- supportability
- login name/password mapping to remote systems
- XNL - Extensible New Limits
- Unicode support - new data types for support of Unicode character set
- LDAP - Directory Services
- SSL - Secure communications
- Union in Views

## Enhancements to CIS

The Adaptive Server 12.5 release includes these new Component Integration Services (CIS) features:

- Distributed Query Optimization Enhancements
- Enhanced Data Access via File System
- Distributed Transaction Management Enhancements
- Administration and Diagnostic Enhancements
- Cascading Proxy Support
- Enhanced Mapping of External Logins
- Enhanced Mapping of External Logins
- Quoted Identifier Support
- Enhanced Full Text Search Capabilities
- Enhancements to Proxy Table Support for Remote Procedures
- Proxy Database Support

- New Global Variables and Set Commands

## File System Access

The 12.5 release provides new features that enable access to the file system through the SQL language. With these new features, it is possible to create proxy tables that are mapped to file system directories, or to individual files.

### Directory Access

A new class of proxy tables is allowed in the 12.5 release that enables SQL access to file system directories and their underlying files. The supported syntax is:

```
create proxy_table <table_name>
   external directory at "directory pathname[;R]"
```

The directory pathname must reference a file system directory visible to and searchable by the Adaptive Server process. A proxy table is created which maps column names to attributes of files that exist within the directory. If the ';R' (indicating "recursion") extension is added to the end of pathname, CIS extracts file information from all directories subordinate to the pathname. The following table contains a description of the proxy table columns that are created when this command successfully completes:

**Table 9-1: Proxy Table columns**

| Column Name | Datatype | Description |
|---|---|---|
| id | numeric(24) | Identity value consisting of values from st_dev and st_ino (See stat(2)). These two values are converted first to a single string (format: "%d%014ld"), and the string is then converted to a numeric value. This column is also designated a 'primary key', to facilitate usage with the Full Text Search server. |
| filename | varchar(n) | the name of the file within the directory specified in at 'pathname', or within directories subordinate to pathname. While the length of pathname is limited to 255 bytes, the total length (n) of filename is system dependent, and specified by the definition of MAXNAMLEN. For Solaris systems, this value is 512 bytes; for most other systems this will be 255 bytes. |

| Column Name | Datatype | Description |
|---|---|---|
| size | int | for regular files, specifies the number of bytes in the file. For block special or character special, this is not defined. |
| filetype | varchar(4) | the file type - legal values are: FIFO, for pipe files; DIR for directories; CHRS for character special files; BLKS for block special files; REG for ordinary files; UNKN for all other file types. Links are automatically expanded, and will not appear as a separate file type. |
| access | char(10) | access permissions, presented in a more or less 'standard' Unix format: "drwxrwxrwx" |
| uid | varchar(n) | The name of the file owner. The value of n is specified by the system definition L_cuserid, which is 9 on all systems except Compaq Tru64, where it is 64. |
| gid | varchar(n) | The name of the owning group. The value of n is specified by the system definition L_cuserid, which is 9 on all systems except Compaq Tru64, where it is 64. |
| atime | datetime | Date/time file data was last accessed |
| mtime | datetime | Date/time when file was last modified |
| ctime | datetime | Date/time when file status was last changed |
| content | image | The actual physical content of the file. |

A proxy table that maps to a file system directory can support the following SQL commands:

- **select** - File attributes and content can be obtained from the proxy table using the select command. Built-in functions that are designed to handle text values are fully supported for the content column. (i.e. textptr, textvalid, patindex, pattern).
- **insert** - A new file or files can be created using the **insert** command. The only column values that have meaning are filename and content; the rest of the columns should be left out of the **insert** statement. If they are not left out, they are ignored.
- **delete** - files may be removed by the use of the **delete** command.
- **update** - Only the name of a file may be changed using the **update** command;
- **readtext** - the contents of a file may be retrieved using the **readtext** command;
- **writetext** - the contents of a file may be modified using the **writetext** command;

No other SQL commands will operate on tables of this type.

Regular file content is available only if the Adaptive Server process has sufficient privileges to access and read the file, and if the file type indicates an 'ordinary file.' In all other cases, the content column will be null. For example:

```
select filename, size, content
from directory_table
where filename like ‘%html’
```

returns the name, size and content of regular files with a suffix of '.html', if the Adaptive Server process has access privileges to the file. Otherwise, the content column will be NULL.

The **create proxy_table** command fails if the pathname referenced by directory pathname is not a directory, or is not searchable by the Adaptive Server process.

If traceflag 11206 is turned ON, then messages are written to the errorlog that contain information about the contents of the directories and the query processing steps needed to obtain that information.

## Recursion through Subordinate Directories

If the "pathname" specified in the **create proxy_table** statement contains the ;R extension, CIS traverses all directories subordinate to pathname, returning information for the contents of each subordinate directory. When this is done, the filename returned by a query contains the complete name of the file relative to the pathname. In other words, all subordinate directory names appear in the filename. For example, if pathname species "/ work;R":

```
create proxy_table d1 external directory at "/work;R"
   select filename, filetype from d1
```

returns values for files in subordinate directories as follows:

Table 9-2: Values for Files

| Filename | Filetype |
|---|---|
| dir1 | DIR |
| dir1/file1.c | REG |
| dir1/file2.c | REG |
| dir2 | DIR |
| dir2/file1.c | REG |

## File Access

Another new class of proxy tables are allowed in the 12.5 release that enables SQL access to individual files within a file system.  The supported syntax is:

```
create proxy_table <table_name
external file at " pathname"
```

When this command is used, a proxy table with one column (named 'record', type varchar(255)) will be created.  It is assumed in this case that the contents of the file are readable characters, and individual records within the file are separated by the newline (\n) character.

It is also possible to specify your own column names and datatypes, using the **create [existing] table** command:

```
create existing table fname (
    column1 int  null,
    column2 datetime null,
    column3 varchar(1024)  null
    etc. etc.
) external file at "pathname"
```

Columns may be any datatype except text, image, or a Java ADT.  The use of the **existing** keyword is optional, and has no effect on the processing of the statement.  In all cases (**create table**, **create existing table**, **create proxy_table**), if the file referenced by pathname does not exist, it is created.  If it does exist, its contents are not overwritten.

When a proxy table is mapped to a file, some assumptions about the file and its contents are made:

1. The file is a regular file (i.e. not a directory, block special, or character special file);
2. The Adaptive Server server process has at least read access to the file.  If the file is to be created, the server process must have write access to the directory in which the file is to be created;
3. The contents of an existing file are in human-readable form;
4. Records within the file are delimited by a newline character;
5. The maximum supported record size is 32767 bytes;
6. Individual columns, except for the last one, are delimited by a single tab character;
7. There is a correspondence between tab-delimited values within each record of the file and the columns within the proxy table.

With proxy tables mapped to files, it is possible to:

1. Back-up database tables to the file system using either **select/into** or **insert/select**.  When an **insert** statement is processed, each column is converted to characters in the default character set of

the server. The results of the conversion are buffered, and all columns (except for the last) are delimited by a single tab. The last column is terminated by a newline. The buffer is then written to the file, representing a single row of data.

2. Provide a SQL alternative to using **bcp in** and **bcp out**. The use of a **select/into** statement can easily back-up a table to a file, or copy a file's contents into a table.
3. Query file content with the **select** statement, qualifying rows as needed with search arguments or functions. For example, it is possible to read the individual records within the Adaptive Server errorlog file:

```
create proxy_table errorlog
external file at
"/usr/sybase/ase12_5/install/errorlog"
select record from errorlog  where record like
"%server%"
```

The query will return all rows from the file that match the **like** pattern. If the rows are longer than 255 bytes, they will be truncated. It is possible to specify longer rows:

```
create existing table errorlog
(
record  varchar(512) null
)
external file at
"/usr/Sybase/ase12_0/install/errorlog"
```

In this case, records up to 512 bytes in length will be returned. Again, since the proxy table contains only one column, the actual length of each column will be determined by the presence of a newline character.

Only the **select** and **insert** data access statements are supported for file access. **update**, **delete** and **truncate** will result in errors if the file proxy is the target of these commands.

*Important*: When an **insert** statement is processed, the file contents are overwritten. There is no ability to append contents to the end of a file with the **insert** statement.

Traceflag 11206 is also used to log message to the errorlog. These messages contain information about the stages of query processing that are involved with file access.

## Enhanced Mapping of External Logins

Users of Adaptive Server that invoke CIS, knowingly or unknowingly, will require login names / passwords to remote servers. By default, the username / password pair used by CIS to connect to a remote server will be the same username / password used by the client to connect to Adaptive Server.

This default mapping is frequently insufficient, and since its first release CIS has supported a one-to-one mapping of Adaptive Server login names and passwords to remote server login names and passwords. For example, using the stored procedure sp_addexternlogin, it is possible to map Adaptive Server user *steve*, password *sybase* to DB2 login name *login1*, password *password1*:

```
sp_addexternlogin DB2, steve, login1, password1
```

In the 12.5 release, it is possible to provide a many-to-one mapping so that all Adaptive Server users who need a connection to DB2 can be assigned the same name and password:

```
sp_addexternlogin DB2, NULL, login2, password2
```

One-to-one mapping has precedence, so that if user *steve* has an external login for DB2, that would be used rather than the many-to-one mapping.

In addition to this, it is possible to assign external logins to Adaptive Server roles. With this capability, anyone with a particular role can be assigned a corresponding login name / password for any given remote server:

```
sp_addexternlogin DB2, null, login3, password3,
rolename
```

The use of the fifth argument to this procedure, containing the role name, identifies the name of a role, rather than the name of a user. Whenever a user with this role active requires a connection to DB2, the appropriate login name / password for the role will be used to establish the connection. When establishing a connection to a remote server for a user that has more than one role active, each role is searched for an external login mapping, and the first mapping found is used to establish the login. This is the same order as displayed by the stored procedure sp_activeroles.

The general syntax for sp_addexternlogin is:

```
sp_addexternlogin
        <servername>,
        <loginname>,
        <external_loginname>,
        <external_password>
        [, <rolename>]
```

<rolename> is optional; if specified then the loginname parameter is ignored.

Precedence for these capabilities are as follows:

- If one-to-one mapping is defined, it will be used - this has the highest precedence.
- If no one-to-one mapping is defined, then if a role is active and a mapping for it can be found, the role mapping will be used to establish a remote connection;
- If neither of the above are true, then many-to-one mapping is used if defined.
- If none of the above is true, then the Adaptive Server login name and password are used to make the connection.

If role mapping is done, and a user's role is changed (via **set role**), then any connections made to remote servers that used role mapping is disconnected. This cannot be done if a transaction is pending, therefore the **set role** command is acceptable if a transaction is active and remote connections are present that used role mapping.

The stored procedure **sp_helpexternlogin** has been updated to allow viewing the various types of extern logins that have been added using **sp_addexternlogin**. The syntax for **sp_helpexternlogin** is:

```
sp_helpexternlogin [<servername> [,<loginname>
[,<rolename>]]]
```

All three parameters are optional, and any of the parameters can be NULL.

The stored procedure **sp_dropexternlogin** has also been modified to accept a third argument, <rolename>. If <role name> is specified then the second argument, <login name>, is ignored.

## Union in Views

New syntax to support the inclusion of the **union** operator within a view has been added to the 12.5 release. Note that the resulting view is not updatable, meaning that **insert**, **delete** and **update** operations are not allowed on views containing the **union** operator.

Component Integration Services supports **union** in views when proxy tables are referenced on either side of the **union** operator by forwarding as much syntax as possible to a remote site. This makes it possible to create a 'virtual table' consisting of separate tables in Oracle and DB2, for example.

This feature is internal to Adaptive Server / CIS, and does not directly affect remote servers. However, when a statement is executed involving a view of this type, and all tables referenced by the view reside on the same remote server, the previously defined **union** capability will be examined to determine whether the **union** operator can be sent to the remote server.

## New Limits for Adaptive Server version 12.5

Limits on length of char, varchar, binary and varbinary datatypes - In the 12.5 release as in prior releases of Adaptive Server, a row cannot span page boundaries, therefore column size has been limited by row size. However, in the 12.5 release of Adaptive Server, configuration allows page sizes of 2k, 4k, 8k or 16k bytes. Also, the arbitrary limit of 255 bytes for char / binary columns has been removed. The 12.5 release supports extended sizes of char, varchar, binary and varbinary data types. The new limit depends on the page size of the server. For various page sizes, the new limits are as follows:

**Table 9-3: New Limits**

| Pagesize | Max. Column Size |
|---|---|
| 2048 | 1900 |
| 4096 | 4000 |
| 8192 | 8000 |
| 16384 | 16000 |

Note that these sizes are still approximate. The basic rule specifies that the limit will be the maximum size that still allows a single row to fit on a page. These limits will also vary depending on the locking scheme specified when the table is created. It is assumed that the bulk of proxy tables will be created with the default locking scheme, which is all page locking.

- Limits on length of Transact-SQL variables and parameters - the size of char, varchar, binary and varbinary variables will be extended to equal the maximum size of columns of the same datatype for a given server. This will allow variables to be passed to stored procedures (or RPCs) whose length exceeds the current limit of 255 bytes.

- Limits on number of columns per table- the old limit of 250 will be removed, and up to 1024 columns per table will be allowed, as long as the columns can still fit on a page. Note that there is a limit of 254 variable length columns (null columns are also considered variable length).
- Limits on the width of an index - the total width of an index within Adaptive Server can be larger than in prior releases, depending on server page size. In the following table, maximum index width is shown according to pagesize:

**Table 9-4: Maximum Index Width**

| Pagesize | Index Width |
|---|---|
| 2048 | 600 |
| 4096 | 1250 |
| 8192 | 2600 |
| 16384 | 5300 |

- Limits on the number of columns per index - the current limit of 31 columns per index will be unchanged in the Everest release.

What these changes mean to CIS and remote servers CIS connects to is described in the following sections.

## Remote Server Capabilities

When communicating with a remote server, CIS needs to know the maximum length of a char / varchar column that can be supported by the DBMS.

For connections to servers in classes Adaptive Servernterprise, ASAnywhere, ASIQ, sql_server and db2, the maximum size is determined based on known attributes of these servers (according to version).

For servers in class direct_connect and sds, this information is provided by an addition to the result set returned by the **sp_capabilities** RPC. A new capability is specified to allow the Direct Connect to indicate the maximum length of columns supported by the DBMS for which the Direct Connect is configured.

Additionally, it is necessary for the Direct Connect to know about the maximum length of char columns that can be supported by CIS. For this reason, changes to the existing RPC **sp_thread_props** are required:

```
sp_thread_props "maximum Adaptive Server column
length", n
```

This RPC is sent to a Direct Connect after CIS has established a connection for the first time. The value of *n* will be an integer indicating the maximum column size, in bytes, allowed by Adaptive Server / CIS.

### create new proxy table

The **create table** command allows columns of datatype char, varchar, binary and varbinary to be specified with extended lengths, as described above. These datatypes and lengths are forwarded to the remote server on which the table is to be created.

### create existing proxy table

The **create existing table** command also allows columns to be specified with a length of greater than 255 bytes. This allows CIS to treat columns in remote databases as char, varchar, binary or varbinary that previously had to be treated as text or image columns.

There is still an opportunity for column size mismatch errors. For example, in the case where the remote database contains a table with a column length of 5000 bytes, and the Adaptive Server processing the **create existing table** command only supports columns up to 1900 bytes, a size mismatch error would occur. In this case, it is necessary to re-specify the column as a text or image column.

In the case where the proxy table column size exceeds that of the corresponding column in the remote table, a size mismatch error is detected and the command is aborted.

### create proxy_table

The **create proxy_table** command imports metadata from a remote server and converts column information into an internal **create existing table** command, with a column list derived from the imported metadata. When obtaining the column metadata, conversion from the remote DBMS type to internal Adaptive Server types is required.

If the size of remote columns (char, varchar, binary or varbinary datatypes) exceeds 255 bytes but is still less than or equal to the maximum Adaptive Server column size, then equivalent Adaptive Server datatypes are used for the proxy table. However, if the size of a remote column exceeds the column size supported by Adaptive Server, then CIS will convert the corresponding proxy table column to text or image (as is the case with the current in-market implementation).

## alter proxy table

If this command operates on a proxy table, it is first processed locally, then forwarded to the remote server for execution. If the remote execution fails, the local changes are backed out and the command is aborted.

The remote server must processes the command appropriately, or raise an error. If an error is produced, the CIS side of the command is aborted and rolled back.

## select, insert, delete, update

CIS handle large column values when proxy tables are involved in DML operations. CIS handles DML using one of several strategies:

- TDS Language commands - if the entire SQL statement can be forwarded to a remote server, then CIS does so using TDS Language commands generated by Ct-Library - **ct_command** (CS_LANG_CMD).

  The text of the language buffer may contain data for long char or binary columns that exceeds 255 bytes, and remote servers must handle parsing of these command buffers.

- TDS Dynamic commands - if CIS cannot forward the entire SQL statement to a remote server (i.e. CIS is forced to provide functional compensation for the statement), then an **insert**, **update** or **delete** may be handled by using TDS Dynamic commands, with parameters as needed, using the Ct-Library function **ct_dynamic** (CS_PREPARE_CMD, CS_EXECUTE_CMD, CS_DEALLOC_CMD).

  The parameters for the dynamic command may be CS_LONGCHAR_TYPE or CS_LONGBINARY_TYPE.

- TDS Cursor commands - Ct-Library cursor operations can be used to handle proxy table operations for **select**, **update** and **delete** if functional compensation has to be performed. For example, if updating a proxy table and there are multiple tables in the **from** clause, CIS may have to fetch rows from multiple data sources, and for each qualifying row, apply the **update** to the target table. In this case, CIS uses **ct_cursor** ({CS_DECLARE_CMD, CS_OPEN_CMD, CS_CURSOR_UPDATE_CMD, CS_CLOSE_CMD, CS_DEALLOC_CMD}).

  After a cursor is prepared, parameters are specified. These parameters may now include those of type CS_LONGCHAR or CS_LONGBINARY.
- Bulk insert commands - when performing a **select/into** operation, if the target server supports the bulk interface (only true of remote Adaptive Server's), then the remote server must be prepared to handle char / binary values > 255 (via CS_LONGCHAR, CS_LONGBINARY values).

  Columns from remote servers may be returned to CIS as type CS_LONGCHAR_TYPE or CS_LONGBINARY_TYPE.

### RPC Handling

RPCs sent to remote servers can contain parameters of types CS_LONGCHAR and CS_LONGBINARY. The CIS command **cis_rpc_handling** supports these new types.

Note that sending long parameters to pre-12.5 servers will not be allowed, as prior versions of Adaptive Server do not support CS_LONGCHAR or CS_LONGBINARY data. CIS examines TDS capabilities for the remote server prior to sending the RPC, and if the remote server cannot accept these datatypes, an error will be produced.

## LDAP Directory Services

The LDAP directory services means that it is no longer necessary to use an interfaces file in both the client and the server. The 12.5 release supports LDAP services for obtaining server information, and so does Component Integration Services. When a connection to a remote server is attempted, CIS instructs Open Client software to reference either the interfaces file or an LDAP server.

CIS uses LDAP services only when the configuration file (*libtcl.cfg*) specifies it.

Note: When an LDAP Server is specified in *libtcl.cfg* then server information becomes accessible from the LDAP Server only and Adaptive Server / CIS will ignore any (traditional) interfaces file.

## Row-Level Access Control

CIS users can employ the features of row-level access control because access rules can be bound to columns on proxy tables.

When queries against proxy tables are processed, the access rule is added to the query tree during query normalization, thus making its existence transparent to downstream query processing. Therefore, CIS users can forward additional predicates to remote servers to restrict the amount of data transferred, according to the expression defined by the access rule.

CIS can function as an row-level access control hub to the entire enterprise through the use of access rules bound to columns on proxy tables.

# 10 Compressed Archive Support in Adaptive Server

This section describes the new compression feature in the **dump** command.

## Dumping databases and transaction logs using compress option

The **dump** command includes a **compress** option that allows you to compress databases and transaction logs using Backup Server.

The partial syntax for **dump database ... compress** and **dump transaction ... compress** commands is:

```
dump database database_name
     to "compress::[compression_level::]archive_name"
     …[stripe on ::[compression_level::]archive_name"] …
```

```
dump transaction database_name
     to "compress::[compression_level::]archive_name"
     …[stripe on ::[compression_level::]archive_name"]…
```

Where *database_name* is the database you are loading into, and **compress::***compression_level* is a number between 0 and 9, with 0 indicating no compression, and 9 providing the highest level of compression. If you do not specify *compression_level*, the default is 6. *archive_name* is the full path to the archive file of the database or transaction log you are compressing. If you do not include a full path for your dump file, Adaptive Server creates a dump file in the directory in which you started Adaptive Server.

Use the **stripe on** clause to use multiple dump devices for a single dump. See Chapter 27, “Backing Up and Restoring User Databases” in the *Adaptive Server Enterprise System Administration Guide* for more information about the **stripe on** clause.

➤ ***Note***

The **compress** option works only with local archives; you cannot use the **servername** option.

#### Example

```
dump database pubs2 to
     "compress::4::/opt/bin/Sybase/dumps/dmp090100.dmp"
```

```
Backup Server session id is:  9.  Use this value when executing
the 'sp_volchanged' system stored procedure after fulfilling any
volume change request from the Backup Server.
Backup Server: 4.132.1.1: Attempting to open byte stream device:
'compress::4::/opt/bin/Sybase/dumps/dmp090100.dmp::00'
Backup Server: 6.28.1.1: Dumpfile name 'pubs2002580BD27  '
section number 1 mounted on byte stream
'compress::4::/opt/bin/Sybase/dumps/dmp090100.dmp::00'
Backup Server: 4.58.1.1: Database pubs2: 394 kilobytes DUMPed.
Backup Server: 4.58.1.1: Database pubs2: 614 kilobytes DUMPed.
Backup Server: 3.43.1.1: Dump phase number 1 completed.
Backup Server: 3.43.1.1: Dump phase number 2 completed.
Backup Server: 3.43.1.1: Dump phase number 3 completed.
Backup Server: 4.58.1.1: Database pubs2: 622 kilobytes DUMPed.
Backup Server: 3.42.1.1: DUMP is complete (database pubs2).
```

The *compression_level* must be a number between 0 and 9. The **compress** option does not recognize numbers outside this range, and treats them as part of the file name while it compresses your files using the default compression level. For example, if you enter:

```
dump database pubs2 to “compress::99::pubs2.cmp”
```

the command creates a file called *99::pubs2.cmp*, which is compressed with the default compression level of 6.

In general, the higher the compression numbers, the smaller your archives are compressed into. However, the compression result depends on the actual content of your files.

Table 10-1 shows the compression levels for the *pubs2* database. These numbers are for reference only; the numbers for your site may differ depending on OS level and configuration.

**Table 10-1: Compression levels and compressed file sizes for pubs2**

| Compression level | Compressed file size |
|---|---|
| No compression / Level 0 | 630Kb |
| Level 1 | 128Kb |
| Level 2 | 124Kb |
| Level 3 | 121Kb |
| Level 4 | 116Kb |
| Level 5 | 113Kb |
| Level 6 / default | 112Kb |
| Level 7 | 111Kb |

**Table 10-1: Compression levels and compressed file sizes for pubs2 (continued)**

| Compression level | Compressed file size |
|---|---|
| Level 8 | 110Kb |
| Level 9 | 109Kb |

The higher the compression level, the more CPU-intensive the process is.

For example, you may not want to use a level-9 compression when archiving your files. Instead, consider the trade-off between processing effort and archive size. The default compression level (6) provides optimal CPU usage, producing an archive that is 60% to 80% smaller than a regular uncompressed archive. Sybase recommends that you initially use the default compression level, then increase or decrease the level based on your performance requirements.

For complete information about **dump database** and **dump transaction**, see the *Adaptive Server Enterprise Reference Manual*.

## Loading databases and transaction logs dumped with compress option

If you use **dump ... compress** to dump a database or transaction log, you must load this dump using the **load ... compress** option.

The partial syntax for **load database .. compress** and **load transaction .. compress** is:

```
load database database_name
   from "compress::archive_name"
   …[stripe on "compress::archive_name"]…
load transaction database_name
   from "compress::archive_name"
   …[stripe on "compress::archive_name"]…
```

Where *database_name* is the database you archived, and **compress::** invokes the decompression of the archived database or transaction log. *archive_name* is the full path to the archived database or transaction log that you are loading. If you did not include a full path when you created your dump file, Adaptive Server created a dump file in the directory in which you started Adaptive Server.

Use the **stripe on** clause if you compressed the database or transaction log using multiple dump. See Chapter 27, “Backing Up and

Restoring User Databases" in the *Adaptive Server Enterprise System Administration Guide* for more information about the **stripe on** clause.

➤ ***Note***

Do not use the *compression_level* variable for the **load** command.

**Example**

```
load database pubs2 from
     "compress::/opt/bin/Sybase/dumps/dmp090100.dmp"
```

```
Backup Server session id is:  19.  Use this value when executing
the 'sp_volchanged' system stored procedure after fulfilling any
volume change request from the Backup Server.
Backup Server: 4.132.1.1: Attempting to open byte stream device:
'compress::/opt/bin/Sybase/dumps/dmp090100.dmp::00'
Backup Server: 6.28.1.1: Dumpfile name 'pubs2002620A951  '
section number 1 mounted on byte stream
'compress::/opt/bin/Sybase/dumps/dmp090100.dmp::00'
Backup Server: 4.58.1.1: Database pubs2: 1382 kilobytes LOADed.
Backup Server: 4.58.1.1: Database pubs2: 3078 kilobytes LOADed.
Backup Server: 4.58.1.1: Database pubs2: 3086 kilobytes LOADed.
Backup Server: 3.42.1.1: LOAD is complete (database pubs2).
Use the ONLINE DATABASE command to bring this database online;
SQL Server will not bring it online automatically.
```

For complete information about **load database** and **load transaction**, see the *Adaptive Server Enterprise Reference Manual*.

# 11 System Table Changes

This section lists changes to system tables.

Table 11-1 lists new column names, the tables they are in, their datatype, and a brief description of what they are.

**Table 11-1: New columns for Adaptive Server version 12.5**

| Table name | Column name | Datatype | Description |
|---|---|---|---|
| *syscolumns* | *accessrule* | *intn* | Lets the table owner specify which rows the users can access. See "Row-Level Access Locking" for more information. |
| *sysconstraints* | *spare1* | *tinyint* | Not for customer use. |
| *syslogins* | *procid* | *int* | Stores the login trigger registered with the **login script** option in **sp_modifylogin**. |
| *sysprocesses* | *loggedindatetime* | *datetimn* | Shows the time and date when the client connected to Adaptive Server. See "Row-Level Access Locking" for more information. |
| | *ipaddr* | *varchar?* | IP address of the client where the login is made. See "Row-Level Access Locking" for more information. |
| *sysservers* | *srvcost* | *intn* | Provides the network cost in milliseconds for accessing a server over a network. Used only by the Adaptive Server query optimizer for evaluating the cost of a query when accessing a proxy table, the default is set to 1,000 ms. |
| *systypes* | *accessrule* | *intn* | Lets the table owner specify which rows users can access. See "Row-Level Access Locking" for more information. |

Table 11-2 lists column names in Adaptive Server version 12.5 that have changed their status from previous versions.

**Table 11-2: Changed status for existing columns**

| Table name | Column name | Old datatype | New datatype |
|---|---|---|---|
| *sysalternates* | *altsuid* | *smallint* | *int* |
| | *suid* | *smallint* | *int* |

**Table 11-2: Changed status for existing columns (continued)**

| Table name | Column name | Old datatype | New datatype |
|---|---|---|---|
| *syscolumns* | *colid* | *tinyint* | *smallint* |
| | *length* | *tinyint* | *int* |
| *syscomments* | *colid* | *tinyint* | *smallint* |
| | *colid2* | *tinyint* | *smallint* |
| *sysconfigures* | *status* | *smallint* | *int* |
| *sysconstraints* | *colid* | *tinyint* | *smallint* |
| *syscurconfigs* | *status* | *smallint* | *int* |
| *sysdatabases* | *suid* | *smallint* | *int* |
| *sysloginroles* | *srid* | *smallint* | *int* |
| | *suid* | *smallint* | *int* |
| *syslogins* | *suid* | *smallint* | *int* |
| *sysobjects* | *uid* | *smallint* | *int* |
| *sysprocedures* | *sequence* | *smallint* | *int* |
| *sysprocesses* | *gid* | *smallint* | *int* |
| | *suid* | *smallint* | *int* |
| | *uid* | *smallint* | *int* |
| *sysprotects* | *uid* | *smallint* | *int* |
| *sysqueryplans* | *uid* | *smallint* | int |
| *sysreferences* | *fokey1 ... 16* | *tinyint* | *smallint* |
| | *refkey1 ... 16* | *tinyint* | *smallint* |
| *sysremotelogins* | *suid* | *smallint* | *int* |
| *sysroles* | *id* | *smallint* | *int* |
| | *lrid* | *smallint* | *int* |
| *syssrvroles* | *srid* | *smallint* | *int* |
| *systypes* | *uid* | *smallint* | *int* |
| *systypes* | *uid* | *smallint* | *int* |
| | *length* | *tinyint* | *int* |
| *sysusermessages* | *uid* | *smallint* | *int* |

**Table 11-2: Changed status for existing columns (continued)**

| Table name | Column name | Old datatype | New datatype |
|---|---|---|---|
| *sysusers* | *gid* | *smallint* | *int* |
| | *uid* | *smallint* | *int* |
| | *suid* | *smallint* | *int* |

Table 11-3 describes changes to the *sysobjects* table. See "sysobjects" in the system tables chapter of the *Adaptive Server Enterprise Reference Manual* for more information on the columns used in *sysobjects*.

**Table 11-3: Changes in the sysobjects table**

| Column name | Datatype | Description |
|---|---|---|
| *type* | *char*(2) | F = SQLJ function |
| | | P = Transact-SQL or SQLJ procedure |
| *sysstat2* | int | Decimal value: 33554432<br>Hex bit representation: 0x2000000<br>Object represents a SQLJ stored procedure |

# 12 Row-level Access Control

Database owners and table owners can restrict access to a table's data rows by defining access rules and binding those rules to the table. Access to data can be further controlled by setting application contexts and creating login triggers.

These features can be grouped under the concept of row-level access control (RLAC). RLAC enables the database owner or table owner to control which rows within a table that users can access based on the their identification or profile and the privileges the user has from the application level. Adaptive Server enforces RLAC for all data manipulation languages (DMLs), which prevents users from bypassing the access control to get to the data.

These concepts are discussed in this chapter:

## Access rules

Domain rules let table owners control the values that users can enter into a particular column that is using a base datatype or any column that is using a user-defined datatype. Rules are enforced during inserts and updates.

Adaptive Server 12.5 enables row-level protection through access rules. Access rules are enforced on **select**, **update**, and **delete** operations. Adaptive Server enforces the access rules on all columns that are read in a query, even if the columns are not included in the select list. In other words, for a given query, Adaptive Server enforces the domain rule on the table that is updated and the access rule on the tables that are read.

Using access rules is similar to using views or an ad hoc query with specific **where** clauses, and does not cause performance degradation. The query is compiled and optimized after the access rules are attached. Therefore, if there are indexes on the columns that have access rules, the queries may perform better.

## Syntax for access rules

The **access** option has been added to the **create rule** syntax to allow creation of access rules. For example, a table owner creates and populates table *T* (*username* char(30), *title* char(20), *classified_data* char(1024):

```
AA, "Administrative Assistant","Memo to President"
AA, "Administrative Assistant","Tracking Stock Movements"
VP1, "Vice President", "Meeting Schedule"
VP2, "Vice President", "Meeting Schedule"
```

The table owner creates a default and a domain rule on the *username* column. The domain rule ensures that the column is updated with correct values. If the default and domain rule are not created, there is a potential security problem in which the user can insert a row into the table with arbitrary data that will not be qualified by the access rule.

The table owner then creates an access rule and binds it to the *username* column using **sp_bindrule**.

```
create default uname_default
as suser_name()
go

sp_bindefault uname_default, "T.username"
go

*/
create accessrule uname_acc_rule
as @username = suser_name()
go

sp_bindrule uname_acc_rule, "T.username"
go
```

Now, when users issue this query:

```
select * from T
```

Adaptive Server processes the access rule that is bound to the *username* column on *T* and attaches it to the query tree. The tree is then optimized and an execution plan is generated and executed as if the user had executed the query with the filter clause given in the access rule. In other words, Adaptive Server attaches the access rule and executes the query as:

```
select * from T where T.username = suser_name().
```

The result of an Administrative Assistant executing the select query is:

```
AA, "Administrative Assistant","Memo to President"
AA, "Administrative Assistant","Tracking Stock Movements"
```

The “where T.username = suser_name()” part of the query is enforced by the server. The user cannot bypass the access rule.

## Extended access rule syntax

Each access rule is bound to one column. To handle evaluation of multiple access rules, there is an extended syntax:

**create and access rule *rule_name***

**create or access rule *rule_name***

## Access rules using Java function and application contexts

The application developer can write flexible access rules using Java and application contexts, described in “Access rules using Java user-defined functions” on page 12-210 and “Application contexts” on page 12-213. For example, you can write a rule that is hierarchical. If table *T* contains all the employees’ schedules, then the President can see all employees’ schedules. Each VP can see their own schedules and their direct reports’ work schedules, but not the President’s schedule.

Access rules can be bound to a user-defined datatype defined through **sp_addtype**. Adaptive Server enforces the access rule on user tables that use these user-defined datatypes. This relieves the database owner and table owner from the task of binding access rules to columns in their normalized schema. For example, there can be a user-defined datatype named *username* for which the base type is *varchar*(30). The database owner or table owner can create an access rule and bind it to the *username* datatype. The owners can then use the *username* datatype in any tables that their application will use. Adaptive Server enforces the access rule on the tables that have columns of the *username* datatype.

## Example scenarios

For this example, assume there is one domain rule for the *region* column and an access rule for the *custid* column, which is not used in this query. For updates, *customer_table* is read, and then updated. Adaptive Server enforces the access rule while reading *customer_table* on *custid* column, and, after updating, enforces the domain rule on the *region* column.

```
update customer_table
    set region = `northwest'
    where region = `north'
```

In this next example, assume there are domain rules on *orders_table* and access rules on *old_orders_table*. Adaptive Server enforces the domain rule on *orders_table* because *orders_table* is updated, and the access rule on the *old_orders_table* because *old_orders_table* is read.

```
insert into orders_table
select *
    from old_orders_table
```

## Access rules using Java user-defined functions

Access rules can use user-defined functions written in Java that use JDBC to look up data in additional tables. Using Java functions, you can, for example, write sophisticated rules that use the profile of the application, the user logged in to use the application, and the roles that the user currently has for the application.

The following Java class uses **GetSecVal** method to demonstrate how you can use Java methods as user-defined functions inside access rules.

```
import java.sql.*;
import java.util.*;

public class sec_class {
static String _url = "jdbc:sybase:asejdbc";
public static int GetSecVal(int c1)
{
try
{
PreparedStatement pstmt;
ResultSet rs = null;
Connection con = null;
int pno_val;

pstmt = null;

Class.forName("sybase.asejdbc.ASEDriver");
con = DriverManager.getConnection(_url);

if (con == null)
{
return (-1);
}

pstmt = con.prepareStatement("select classification from sec_tab where
id = ?");

if (pstmt == null)
{
return (-1);
}

pstmt.setInt(1, c1);

rs = pstmt.executeQuery();

rs.next();

pno_val = rs.getInt(1);

rs.close();

pstmt.close();

con.close();

return (pno_val);

}
catch (SQLException sqe)
{
return(sqe.getErrorCode());
}
catch (ClassNotFoundException e)
{

System.out.println("Unexpected exception : " + e.toString());
System.out.println("\nThis error usually indicates that " + "your Java
CLASSPATH environment has not been set properly.");
e.printStackTrace();
return (-1);
}
catch (Exception e)
{
System.out.println("Unexpected exception : " + e.toString());
e.printStackTrace();
return (-1);
}
}
}
```

*(from Shell)*

```
javac sec_class.java
```

```
jar cufo sec_class. jar sec_class.class
installjava -Usa -Password -f/work/work/FGAC/sec_class.jar -
-D testdb
```

```
(from isql)
```

```
create table sec_tab (id int, classification int)
go
insert into sec_tab values (1,10)
insert into sec_tab values (2,9)
insert into sec_tab values (3,7)
insert into sec_tab values (4,7)
insert into sec_tab values (5,4)
insert into sec_tab values (6,4)
insert into sec_tab values (7,4)
go

sp_addtype class_level, int
go

create table sec_data (c1 varchar(30),
c2 varchar(30),
c3 varchar(30),
clevel class_level)
go

declare @v1 int
select @v1 = 5
while @v1 > 0
begin
insert into sec_data values('8', 'aaaaaaaaaa', 'aaaaaaaaaa', 8)
insert into sec_data values('7', 'aaaaaaaaaa', 'aaaaaaaaaa', 7)
insert into sec_data values('5', 'aaaaaaaaaa', 'aaaaaaaaaa', 5)
insert into sec_data values('5', 'aaaaaaaaaa', 'aaaaaaaaaa', 5)
insert into sec_data values('2', 'aaaaaaaaaa', 'aaaaaaaaaa', 2)
insert into sec_data values('3', 'aaaaaaaaaa', 'aaaaaaaaaa', 3)
select @v1 = @v1 -1
end
go

create access rule clevel as
@clevel <= sec_class.GetSecVal(suser_id())
go

create default clevel_def as sec_class.GetSecVal(suser_id())
go

sp_bindefault clevel_def, class_level
```

```
go

sp_bindrule clevel, class_level
go

grant all on sec_data to public
go

grant all on sec_tab to public
go
```

## Application contexts

Applications on a database server should be programmed to limit access to the data based on their users and user profiles.

The application developer is responsible for coding the application appropriately. For example, a human resources application is programmed to know which users are allowed to update salary information.

Application contexts allow users to define, store, and retrieve profiles of the user—the roles they are authorized to use and groups to which he or she belongs—and the application currently used by the user. Application contexts can be used to store and retrieve arbitrary client data, and can use Adaptive Server to store client information.

Application contexts are specific to a session. They are not persistent across sessions; however, they are available across nested levels of statement execution, unlike local variables.

An application context consists of a context name, an attribute name, and an attribute value. Users define the context name and the attributes and values for each context. Sybase provides a variety of attributes in the system application context, sys_sessions. For details, see “sys_session system application context” on page 12-218. You can also create your own application contexts as described in “Creating and maintaining application contexts” on page 12-214.

### Setting permissions for using application context functions

Application contexts are set, retrieved, and removed using functions. Therefore, any user who is logged in can reset the profiles of the session. Although execution of a function is audited, security may be compromised before the problem is noticed. You can restrict access to functions using grant and revoke privileges. Only the application context functions perform data access control checks on the user. Granting or revoking privileges for other functions does not have any effect in Adaptive Server.

Application context function execution is treated as a **select** DML. The owner of the function is the System Administrator of the server. Only users with sa_role can grant or revoke privileges on the functions. Only the **select** privilege is checked as part of server-enforced data access control checks done by the functions. By default, privileges on the functions are revoked to PUBLIC. This matches current defaults for table-level privileges.

You can grant and revoke privileges to users, roles, and groups in a given database for objects in that database. The only exceptions are **create database**, **set session authorization**, and **connect**. A user granted these privileges should be a valid user in the *master* database. For other privileges, the user should be valid in the database where the object is located.

However, functions do not have an object ID and they do not have a home database. Therefore, in each database, the database owner must grant to the appropriate user the select privilege for the functions. Adaptive Server finds the user's default database and checks the permissions against this database. With this approach, only the owner of the users' default database needs to grant the select privilege. If other databases should be restricted, the database owner of those databases must explicitly revoke permission for the user in those databases.

A System Administrator can execute the following commands to grant or revoke select privileges on specific application context functions:

```
grant select on set_appcontext to user_role
grant select on set_appcontext to joe_user
revoke select on set_appcontext from joe_user
```

## Creating and maintaining application contexts

The following functions are available for creating and maintaining application contexts:

- **set_appcontext**
- **get_appcontext**
- **list_appcontext**

- **rm_appcontext**

## set_appcontext

**set_appcontext** is used to set a context name, attribute name, and attribute value for the user session. The options are:

```
set_appcontext ("context_name", "attribute_name",
   "attribute_value")
```

Where *context_name, attribute_name* have datatypes of *char*(30) and *attribute_value* has a datatype of *char*(2048). This function returns 0 for success and -1 for failure. **set_appcontext** cannot override values of an existing application context. If you want to assign new values to a context, remove the context, then re-create it with the new values. If the values being set already exist in the session, the function returns -1. Attributes are saved as *char* datatype. If the rule must use the attribute value to compare against other datatypes, the rule should convert the *char* data to the appropriate datatype.

### *Examples*

1. **select set_appcontext("CONTEXT1","ATTR1", "VALUE1")**

   ```
   ------------
   0
   ```

   This example creates an application context named CONTEXT1. The attribute is named ATTR1 with a value of VALUE1.

2. **select set_appcontext("CONTEXT1", "ATTR2", convert(char(20), @numericvar)**

   ```
   -----------
   0
   ```

   This example shows **set_appcontext** with a datatype conversion included in the value.

3. **select set_appcontext("CONTEXT1", "ATTR1", "VALUE1")**

   ```
   ----------
   -1
   ```

   This example shows the result of attempting to override an existing application context. This context was created in example 1. The context must be removed, then recreated with the new values.

4. **select set_appcontext("CONTEXT1", "ATTR2", "VALUE1")**

```
Select permission denied on function
set_appcontext, database dbid
```

This example shows the result of a user without appropriate permissions attempting to set the application context.

## get_appcontext

**get_appcontext** returns the value of the attribute in a given context. The options are:

*get_appcontext (“context_name”, “attribute_name”)*

Where *context_name*, *attribute_name* have datatypes of *char*(30). If the attribute does not exist in the application context, **get_appcontext** returns null. The attribute value is returned as a *char* datatype. If the rule must use the attribute value to compare against other datatypes, then the rule should convert the *char* data to the appropriate datatype.

### *Examples*

1. **`select get_appcontext("CONTEXT1,"ATTR1")`**

   ```
   ----------------
   VALUE1
   ```

   This example shows a return of VALUE1 for the value of ATTR1.

2. **`select get_appcontext("CONTEXT2", "ATTR1")`**

   ```
   --------------
   NULL
   ```

   ATTR1 does not exist in CONTEXT2.

3. **`select get_appcontext("CONTEXT1","ATTR2","VALUE1")`**

   ```
   Select permission denied on built-in
   get_appcontext, database dbid
   -------
   -1
   ```

   This example shows the result of a user, without appropriate permissions, attempting to get the application context.

## list_appcontext

**list_appcontext** returns all the attributes in all the contexts for the current session. The option is:

**`list_appcontex`** ***(“context_name”)***

Where *context_name* has a datatype of char(30). It returns 0 for success or -1 for failure. Since the function cannot return multiple result sets, the results are printed in the same way that **show plan** prints the plan using **ex_callprint**. The client receives these as messages against a result set.

#### *Examples*

1. **`select list_appcontext([`*`context_name`*`])`**

   ```
   Context Name: (CONTEXT1)
   Attribute Name: (ATTR1) Value: (VALUE1)
   Attribute Name: (ATTR2) Value: (VALUE2)
   Context Name: (CONTEXT2)
   Attribute Name: (ATTR1) Value: (VALUE1)
   ```

2. **`select list_appcontext()`**

   ```
   Select permission denied on built-in
   list_appcontext, database DBID
   ----------
   -1
   ```

   This example shows the result of a user without appropriate permissions attempting to list the application contexts.

### rm_appcontext

**rm_appcontext** removes a specific application context, or all application contexts. If the removal is successful, the function returns 0. If not, it returns -1. The options are:

**`rm_appcontext (`*`“context_name”, “attribute_name”`*`)`**

Where *context_name* and *attribute_name* have datatypes of *char*(30).

#### *Examples*

1. **`select rm_appcontext("Context1","*")`**

   ```
   ------------
   0
   ```

2. **`select rm_appcontext("*","*")`**

   ```
   -------------
   0
   ```

3. **`select rm_appcontext("CONTEXT1", "ATTR1")`**

   ```
   --------------
   0
   ```

4. **`select rm_appcontext("NON_EXISTING_CTX", "ATTR")`**

   ```
   --------------
   -1
   ```

5. **`select rm_appcontext("CONTEXT1","ATTR2")`**

   ```
   Select permission denied on built-in
   get_appcontext, database dbid
   ------------
   -1
   ```

   This example shows the result of a user without appropriate permissions attempting to remove the application contexts.

## sys_session system application context

**sys_session** is a system application context function. It is predefined and cannot be modified using **set_appcontext**. It returns the following values:

**Table 12-1: sys_session attributes and values**

| Attribute | Value |
|---|---|
| username | login name |
| hostname | host name the client has connected from |
| applname | Name of the application as set by the client |
| session_userid | User ID of the user in the current database |
| session_groupid | Group ID of the user in the current database |
| current_userid | User ID (after setting setuser) |
| current_groupid | Group ID (after setting setuser) |
| current_dbid | ID of the current database the user is in |
| current_dbname | Name of the current database the user is in |
| spid | Server process ID |
| proxy_name | Proxy name set using the **set session authorization** or **set proxy** commands |
| proxy_suserid | The server user id of the proxy |
| clientname | Client name set by the middle-tier application, using the **set clientname** command |
| clientapplname | Client application name set by the middle-tier application, using the **set clientapplname** command |

**Table 12-1: sys_session attributes and values**

| Attribute | Value |
|---|---|
| clienthostname | Client hostname set by the middle-tier application, using the **set clienthostname** command |
| language | Current language the client is using by default or after using the **set language** command (**@@language**) |
| character_set | Character set the client is using (**@@client_csname**) |
| lasterror | Error code returned by the last statement executed (**@@error**) |
| rowcount | Rowcount set by **set rowcount** command (**@@rowcount**) |
| servername | Name of the server (**@@servername**) |
| transaction_count | **@@trancount** |
| nesting_level | **@@nestlevel** |
| dateformat | Date format expected by the client, set using the **set dateformat** command |
| roles | Currently enabled roles |
| is_showplan_on | Returns YES if **set showplan** is on, NO if it is off |
| is_noexec_on | Returns YES if **set noexec** is on, NO if it is off |
| is_ansi_null_on | Returns YES if **set ansinull** is on, NO if it is off |

## Login triggers and scripts

A login trigger is useful for setting the application context for the user who is logged in. The login trigger is a stored procedure that is fired as the last step in the login process.

The System Security Officer, System Administrator or database administrator can register a login trigger to users in the server. Users can register a login trigger with their own login.

The login trigger can be registered through **sp_modifylogin**. The syntax is:

```
sp_modifylogin "login name", "login script",
   "sproc_name"
```

Run this procedure from the user's default database. The login trigger should be available in this default database. Adaptive Server searches the *sysobjects* table in the user's default database to find the login trigger object. If there are any errors executing the login trigger, they are printed in the server error log. Adaptive Server executes the login trigger as a background task. Only the database administrator and the user can use the **login script** option in **sp_modifylogin**.

To provide a secure environment, the database administrator should:

1. Revoke **select** privilege on the **set_appcontext** function.
2. Create a login trigger and register the login trigger to the user.
3. Provide execute privilege to the login trigger to be executed by a user.

   The login trigger should look up a table that has the application names, the users using these applications in the server, and their appropriate contexts, then set the context attribute values accordingly.

All rules should do either or both of the following:

- Use the application context environment for the user profile.
- Use the Java functions or any other server functions that are read-only.

## Example scenario of row-level access control

This section describes an example scenario of row-level access control.

This example assumes that a company has one department with a director, two managers managing two groups in the department, and two engineers working for each of the two managers. The example creates access rules such that when the department director logs in and queries the HR table, she can see her own record and all the records of her direct reports and their direct reports. Similarly, when the managers log in they can see their own records and those of the engineers reporting to them.

The *hr_emp* table is created as follows:

```
/*
** Employee table
*/
create table hr_emp
(u_empid int,
u_name varchar(30),
u_title varchar(30),
u_mgrname varchar(30),
u_dept varchar(30),
)

insert dst_emp values (suser_id("Director"), "D1",
"Director","The Boss", "DST")
insert dst_emp values (suser_id("Manager1"), "M1",
"Manager-1","D1", "QP1")
insert dst_emp values (suser_id("Engineer11"), "E11", "Emp1.1",
"M1","QP1")
insert dst_emp values (suser_id("Manager2"), "M2",
"Manager-2","D1", "QP2")
insert dst_emp values (suser_id("Engineer21"), "E21", "Emp2.1",
"M2","QP2")
```

The following profile is defined for the director, managers, and engineers.

**Table 12-2: Context names and attributes for the sample application**

| User | Context | Attribute | Value |
|---|---|---|---|
| Director | DST | DEPT | ALL |
| Director | DST | EMPID | ALL |
| Manager1 | DST | DEPT | QP1 |
| Manager1 | DST | EMPID | ALL |
| Engineer11 | DST | DEPT | QP1 |
| Engineer11 | DST | EMPID | E11 |
| Manager2 | DST | DEPT | QP2 |
| Manager2 | DST | EMPID | ALL |
| Engineer21 | DST | DEPT | QP2 |
| Engineer21 | DST | EMPID | E12 |

The profile is populated in *app_context_tbl* in *hr_db*.

```
create table app_context_tbl (u_name varchar(30), appname
varchar(30), attr varchar(30), value varchar(30))
go

/* director */
insert app_context_tbl values ("Director","DST","DEPT","All")
insert app_context_tbl values ("Director","DST","EMPID","All")
go

/* manager group 1*/
insert app_context_tbl values ("Manager1","DST","DEPT","QP1")
insert app_context_tbl values ("Manager1","DST","EMPID","All")
go

/* engineer group 1 */
insert app_context_tbl values ("Engineer11","DST","DEPT","QP1")
go

/* manager group 2 */
insert app_context_tbl values ("Manager2","DST","DEPT","QP2")
insert app_context_tbl values ("Manager2","DST","EMPID","All")
go

/* engineer group 2 */
insert app_context_tbl values ("Engineer21","DST","DEPT","QP2")
go
```

The following stored procedure is registered as the login trigger for each user. The procedure is owned by the System Administrator because the default privilege on **set_appcontext** has been revoked for all other users.

```
create proc loginproc
    as
        declare @appname varchar(20)
        declare @attr varchar(20)
        declare @value varchar(20)
        declare @retval int
        declare apctx cursor for
select appname, attr, value
    from hr_db.dbo.app_context_tbl
    where u_name = suser_name()
open apctx
fetch apctx into @appname, @attr, @value
    while (@@sqlstatus = 0)

begin
    select @retval = set_appcontext(rtrim(@appname),
        rtrim(@attr), rtrim(@value))
    fetch apctx into @appname, @attr, @value
end
go

grant execute on loginproc to public
go

/* make sure we are in the user's default database */
use hr_db
go

sp_modifylogin "Director", "login script", "loginproc"
go

sp_modifylogin "Manager1","login script","loginproc"
go

sp_modifylogin "Manager2","login script","loginproc"
go

sp_modifylogin "Engineer11","login script","loginproc"
go

sp_modifylogin "Engineer21", "login script", "loginproc"
go
```

To create the access rules and bind them to the *hr_emp* table columns.

```
/*
** Ind.contributor should satisfy the following conditions
** (1) Empid should match
** (2) Department should match
**
** Manager should satisfy the following conditions
** (1) Department should match
**
** No conditions for Director
*/
create access rule emp_access
as
@empid = suser_id() or get_appcontext("DST", "EMPID") = "All"
go
```

```
sp_bindrule emp_access, "hr_emp.u_empid"
go
create access rule dept_access
as
@deptid = get_appcontext("DST", "DEPT") or
get_appcontext("DST","DEPT") = "All"
go
sp_bindrule dept_access, "hr_emp.u_dept"
go
```

Adaptive Server can now enforce the access rules and return rows that satisfy the access rules when users log in and query the *hr_emp* table.

```
isql -UEngineer11 -Ppassword
1> select * from hr_emp
2> go
```

```
u_empid     u_name         u_title  u_mgrname
u_dept
----------- ------------- -------- -------------
---------
6           E11             Emp1.1   Manager1
QP1
```

```
isql -UManager2 -Ppassword
1>select * from hr_emp
2>go
```

```
u_empid     u_name         u_title  u_mgrname
u_dept
----------- ------------- -------- -------------
---------
6 M2 Manager-2 D1 DST
7 E21 Emp2.1 Manager2 QP1
```

```
isql -UDirector -Ppassword
1>select * from hr_emp
2>go
```

```
u_empid     u_name         u_title  u_mgrname
u_dept
----------- ------------- -------- -------------
---------
3 D1 Director The Boss DST
4 M1 Manager-1 D1 QP1
5 M2 Manager-2 D1 QP2
6 E11 Emp1.1 Manager1 QP1
7 E12 Emp1.2 Manager2 QP2
```

# 13 New Features in Open Client / Open Server 12.5

This section describes the new features of Open Client / Open Server version 12.5.

- unichar16 – a user-defined format that stores Unicode-encoded characters in two-byte format.
- New limits that support wider tables, and larger page sizes.
- Implicit Cursors – an optimized use of the Client Library fetch row command.

To enable any of these features, the cs_context structure must be set with version CS_VERSION_125, using cs_ctx_alloc.

```
retcode = cs_ctx_alloc (CS_VERSION_125, context);
The version parameter specifies which features are
enabled.
```

**Table 13-1: cs_cxt_alloc parameters**

| Value of version | Indicates | Features |
|---|---|---|
| CS_VERSION_110 | 11.1 behavior | Unicode character set support.<br>Use of external configuration files to control Client Library property settings. |
| CS_VERSION_120 | 12.0 behavior | High-availability failover functionality.<br>Bulk row inserts.<br>New SORTMERGE property. |
| CS_VERSION_125 | 12.5 behavior | Two-byte UTF-16 encoded character support.<br>Extended page size and column size support.<br>Implicit cursors for high performance single-row fetches. |

## unichar datatype

Open Client / Open Server 12.5 unichar supports two-byte characters, supporting multilingual client applications, and reducing the overhead associated with character-set conversions.

Designed the same as the Open Client / Open Server CS_CHAR datatype, CS_UNICHAR is a shared, C-programming datatype that can be used anywhere the CS_CHAR datatype is used. The CS_UNICHAR datatype stores character data in Unicode UCS Transformational Format 16-bit (UTF-16), which is two-byte characters.

The Open Client / Open Server CS_UNICHAR datatype corresponds to the Adaptive Server 12.5 UNICHAR fixed-width and

UNIVARCHAR variable-width datatypes, which store two-byte characters in the Adaptive Server database.

As a standalone, Open Client 12.5 applications can use this new functionality to convert other datatypes to and from CS_UNICHAR at the client site, even if the server does not have the capability to process two-byte characters.

## New datatypes and capabilities

To send and receive two-byte characters, the client specifies its preferred byte order during the login phase of the connection. Any necessary byte swapping is performed on the server site.

The Open Client ct_capability() parameters:

- CS_DATA_UCHAR – is a request sent to the server to determine whether the server supports two-byte characters.
- CS_DATA_NOUCHAR – is a parameter sent from the client to tell the server not to support unichar for this specific connection.

To access two-byte character data, Open Client / Open Server implements:

- CS_UNICHAR– a datatype.
- CS_UNICHAR_TYPE – a datatype constant to identify the data's datatype.

Setting the CS_DATAFMT parameter's datatype to CS_UNICHAR_TYPE allows you to use existing API calls, such as ct_bind, ct_describe, ct_param, and so on.

CS_UNICHAR uses the format bitmask field of CS_DATAFMT to describe the destination format.

For example, in the Client Library sample program, rpc.c, the BuildRpcCommand() function contains the section of code that describes the datatype:

```
...
strcpy (datafmt.name, “@charparam”);
datafmt.namelen =CS_NULLTERM;
datafmt.datatype = CS_CHAR_TYPE;
datafmt.maxlength = CS_MAX_CHAR;
datafmt.locale = NULL;
...
```

In this example, the character type is defined as `datafmt.datatype = CS_CHAR_TYPE`. Use an ASCII text editor to edit the datafmt.datatype field to:

```
...
strcpy (datafmt.name, “@charparam”);
datafmt.namelen =CS_NULLTERM;
datafmt.datatype = CS_UNICHAR_TYPE;
datafmt.maxlength = CS_MAX_CHAR;
datafmt.status = CS_RETURN;
datafmt.locale = NULL;
...
```

Since CS_UNICHAR is a UTF-16 encoded Unicode character datatype that is stored in two bytes, the maximum length of CS_UNICHAR string parameter sent to the server is restricted to one-half the length of CS_CHAR, which is stored in one-byte format.

Table 13-2 lists the CS_DATAFMT bitmask fields.

**Table 13-2: CS_DATAFMT structure**

| Bitmask field | Description |
|---|---|
| CS_FMT_NULLTERM | The data is two-byte Unicode null-terminated (0x0000). |
| CS_FMT_PADBLANK | The data is padded with two-byte Unicode blanks to the full length of the destination variable (0x0020). |
| CS_FMT_PADNULL | The data is padded with two-byte Unicode nulls to the full length of the destination variable (0x0000). |
| CS_FMT_UNUSED | No format information is provided. |

## *isql* and *bcp* utilities

Both the isql and the bcp utilities automatically support unichar data if the server supports two-byte character data.

If the client’s default character set is UTF-8, isql displays two-byte character data, and bcp saves two-byte character data in the UTF-8 format. Otherwise, the data is displayed or saved, respectively, in two-byte Unicode data in binary format.

Use `isql -Jutf8` to set the client character set for isql. Use `bcp -Jutf8` to set the client character set for the bcp utility.

## Limitations

The sever to which the Open Client / Open Server is connecting must support two-byte Unicode datatypes, and use UTF-8 as the default character set.

If the server does not support two-byte Unicode datatypes, the server returns an error message: `“Type not found. Unichar/univarchar is not supported.”`

CS_UNICHAR does not support the conversion from UTF-8 to UTF-16 byte format for CS_BOUNDARY and CS_SENSITIVITY. All other datatype formats are convertible.

CS_UNICHAR does not provide C programming operations on UTF-16 encoded Unicode data such as Unicode character strings. For full support for Unicode character strings, you must use the Sybase product, Unilab. See the *Unilib Reference Manual* at at http://sybooks.sybase.com. The reference manual is part of the Sybase Unicode Developers Kit 2.0.

## New limits in version 12.5

Open Client / Open Server 12.5 allow client applications to send and receive wider tables and larger page sizes that are supported in Adaptive Server 12.5.

Larger pages – support for 2K, 4K, 8K, or 16K logical page databases.

Wide tables – for columns in excess of 255 characters, and more than 255 columns per table.

Client-Library applications compiled on earlier versions of Client-Library must be recompiled with the Open Client / Open Server 12.5 libraries to enable the new limits.

### Page size

Adaptive Server Enterprise 12.5 supports logical page sizes of 2K, 4K, 8K, and 16K. Open Client / Open Server uses the Bulk-Library (blklib) routines to populate these pages. Until the release of 12.5, blklib only supported the Adaptive Server page size of 2K.

Table 13-3 lists the new bulk library constants and their values.

**Table 13-3: Page size values**

| blk_pagesize | blk_maxdatarow | blk_maxcolsize | blk_maxcolno | blk_boundary |
|---|---|---|---|---|
| 2K | 1962 | 1960 | 1962 | 1960 |
| 4K | 4010 | 4008 | 4010 | 4008 |
| 8K | 8106 | 8104 | 8106 | 8104 |
| 16K | 16298 | 16296 | 16298 | 16298 |

Increased page size limits allow for increased number of columns, depending upon the type of table. The limits are:

- 1024 for fixed-length columns in both all-pages locking (APL) and data-only locking (DOL) tables
- 254 for variable-length columns in an APL table
- 1024 for variable-length columns in an DOL table

No changes have been made to the existing blklib APIs, nor have any new APIs been added to accommodate the larger page size support in Adaptive Server 12.5.

### Compatibility

Support for large page size is automatically enabled if:

- The client is set to CS_VERSION_125 or higher,
- It is linked with Open Client Server 125 library, and
- The Adaptive Server to which it is connected has the capabilities to handle wide tables:

```
select @@version
```

If Open Client / Open Server 12.5 blklib is linked to a version 12.5 bcp application that communicates with a pre-12.5 Adaptive Server, the bcp utility assumes that Adaptive Server has the 2K page size.

If the blklib is linked to a bcp application that was built with a version of the utility earlier than 12.5 (the version string is not set to CS_VERSION_125), it cannot support the copy of large pages.

## Wide tables

Adaptive Server Enterprise 12.5 supports tables with more than 255 columns and column sizes in excess of 255 characters or 255 binary data. To accommodate the expanded table limits in Adaptive Server, Open Client / Open Server 12.5 sends and receives wider tables and tables with more columns.

### Capability

To support wide tables, the client sends a login packet to the server along with a capability packet. Possible ct_capability parameters include:

- CS_WIDETABLE – a request capability that a client sends to the server indicating the client has the capability to receive larger data table formats.

- CS_NOWIDETABLE – a response capability that a client sends to the server to have the server disable wide table support for this particular connection.

If the version of the application is set to CS_VERSION_125, the Client-Library always sends CS_WIDETABLE capability to the server; the application does not have control of the request capability. However, the application can set CS_NOWIDETABLE response capability before the connection is established to specifically request the server not to enable wide table capabilities.

The syntax of ct_capability is:

```
CS_RETCODE ct_capability (connection, action, type,capability, value)
   CS_CONNECTION              *connection;
   CS_INT                      action;
   CS_INT                      type;
   CS_INT                      capability;
   CS_VOID                     *value;
```

where the values of *type* are CS_WIDETABLES or CS_NOWIDETABLES.

If you do not want to enable wide table support, you can send the server a CS_NOWIDETABLE command to disable this feature.

```
...
CS_BOOL     boolv = CS_TRUE
...
 retcode = ct_capability ( *conn_ptr, CS_SET, CS_CAP_RESPONSE,
     CS_NOWIDETABLES, &boolv);
...
```

ct_dynamic() with CS_CURSOR_DECLARE supports the flags CS_PREPARE, CS_EXECUTE, and CS_EXEC_IMMEDIATE to prepare and execute dynamic SQL statements that reference the 1024-column limit of Adaptive Server 12.5.

ct_param() can be used to pass as many as 1024 arguments to a dynamic SQL statement.

### Changes in application program

If the column data you are retrieving is in excess of CS_MAX_CHAR (256 characters or 256 binary data), you must edit the CS_DTATFMT structure field datafmt.maxlength definition to the maximum length, in bytes, of the data that you are retrieving. Otherwise you get a truncation error.

If you expect wider columns in the client program, change the column array size in the application program.

For example, if the application expects a column that is 300 characters wide, then the column should mention CS_CHAR col1[300] at an appropriate place. Assign an appropriate length-of-character datatype, to the maxlength parameter of the CS_DATAFMT structure for RPC applications if the column is more than 255 bytes. The following is recommended for the CS_DATAFMT parameter:

```
datafmt.datatype = CS_CHAR_TYPE
datafmt.maxlength = sizeof(col1)
```

The following example is a small ctlib program using the pubs2 database.

1. Alter the *authors* table and a column "comment" declare as a *varchar(500)*:

   ```
   1>alter table authors add comment varchar(500) null
   2>go
   ```

2. Update the new column within the table:

```
1>update authors set comment = replicate (substring(state,1,1), 500)
2>go
/* This SQL command will update the comment column with a replicate of
500 times the first letter of the state for each row. */
```

3. Modify the *example.h* file to set the "new limits" capabilities.

   ```
   #define EX_CTLIB_VERSION CS_VERSION_125
   ```

4. Update the *exutils.h* file and reset the MAX_CHAR_BUF to 16384 (16K).

5. Recompile and link ctlib using 12.5 headers and libraries.

6. Execute and test on a Adaptive Server version 12.5 *X*k page size server.

   If you set CS_VERSION_125, you see the following (only displays the last 2 rows):

```
Heather                 McBadden
95688        CCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCC
CCCCCCCCCCCCCC
Anne                    Ringer
```

```
84152       UUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUU
UUUUUUUUUUUUUU
```

7. Update the *example.h* file and reset ctlib to CS_VERSION_120. Recompile and link using *OCS-12_5* headers and libraries.

➤ ***Note***

If you execute the same program without setting CS_VERSION_125 first, you retrieve only the first 255 bytes of the comment column and cannot retrieve wide columns, even if you are using version 12.5 of Adaptive Server and OCS-12.5 libraries.

Open Client message:

```
Message number: LAYER = (1) ORIGIN = (4) SEVERITY = (1) NUMBER = (132)
Message String: ct_fetch(): user api layer: internal common library
error: The bind of result set item 4 resulted in truncation.
Error on row 21.
Heather              McBadden
95688       CCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCC
```

Open Client message:

```
Message number: LAYER = (1) ORIGIN = (4) SEVERITY = (1) NUMBER = (132)
Message String: ct_fetch(): user api layer: internal common library
error: The bind of result set item 4 resulted in truncation.
Error on row 22.
Anne                 Ringer
84152       UUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUU
```

## Wide-table compatibility

Wide-table support is activated automatically if:

- The client is set to CS_VERSION_125,
- It is linked with Open Client Server 12.5 library, and
- the Adaptive Server to which it is connected has the capabilities to handle wide tables.

If the Client-Library application's version string is not set to CS_VERSION_125, and it is linked to an Open Client / Open Server 12.5, the application does not support the extended limits and there is no behavioral change.

If the Open Client / Open Server version 12.5 connects to a pre-12.5 Adaptive Server, the server returns a capability bit of 0, indicating that it does not support wide tables; the connection is still made but there are no behavioral changes.

Likewise, if a pre-12.5 version of Open Client / Open Server connects to an Adaptive Server 12.5, the new limits are not enabled. However, if the Adaptive Server determines that it must send a wide-table format to an older client, the data is truncated and sent.

➤ ***Note***

Adaptive Server 11.0.x and SQL Server return a mask length of 0 for any mask length in excess of 7 bytes. If the connection request receives a capability mask of 0, you see this error message:

```
ct_connect(): protocol specific layer: external error: "This
server does not accept new larger cpability mask, the original
cap mask will be used."
```

and the extended limits are not enabled.

## Implicit Cursors

Implicit cursors are designed for Open Client applications that perform multiple single-row fetches. Implicit cursors reduce the number of context switches performed by the server, decreasing network traffic and increasing the performance of client applications performing multiple, single-row fetches.

There is no external configuration required to implement implicit cursors. The cursor in a client application sends the cursor declare statement:

```
ct_cursor (cmd, CS_CURSOR_DECLARE, cur_name,
   CS_NULLTERM, QUERY_TEXT, CS_NULLTER,
   CS_IMPLICIT_CURSOR);
```

When the server receives a CS_CURSOR_OPEN command as an implicit cursor type, the server fetches the rows and sends them to the client, repeating the process until it sends all the rows—without any further requests from the client. When the server encounters the last row in a fetch, it closes the cursor and sends cursor closed information to the client.

### Compatibility

If an Open Client / Open Server application version string is set to CS_VERSION_125, it sends a declaring cursor of type CS_IMPLICIT_CURSOR. If the receiving Adaptive Server version string is VERSION_125, the implicit cursor option is transparently enabled.

If either the receiving Adaptive Server or the requesting Open Client application does not have the version string set to VERSION_125, the implicit cursor functionality is not enabled, and there is no behavioral change in the application.

### Limitations

When an implicit cursor issues a fetch command, the server retrieves all rows that match the fetch statement criteria and places the result set in a cache. For this reason, any insertions of new rows that match the criteria are not included in the active result set unless the connection has not yet received the cursor closed information.

The result set of a ct_fetch with an implicit cursor type is read-only.