GemStone®

GemConnect

Version 2.0

January 2007

GemStone™

INTELLECTUAL PROPERTY OWNERSHIP

This documentation is furnished for informational use only and is subject to change without notice. GemStone Systems, Inc. assumes no responsibility or liability for any errors or inaccuracies that may appear in this documentation.

This documentation, or any part of it, may not be reproduced, displayed, photocopied, transmitted, or otherwise copied in any form or by any means now known or later developed, such as electronic, optical, or mechanical means, without express written authorization from GemStone Systems, Inc.

Warning: This computer program and its documentation are protected by copyright law and international treaties. Any unauthorized copying or distribution of this program, its documentation, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted under the maximum extent possible under the law.

The software installed in accordance with this documentation is copyrighted and licensed by GemStone Systems, Inc. under separate license agreement. This software may only be used pursuant to the terms and conditions of such license agreement. Any other use may be a violation of law.

Use, duplication, or disclosure by the Government is subject to restrictions set forth in the Commercial Software - Restricted Rights clause at 52.227-19 of the Federal Acquisitions Regulations (48 CFR 52.227-19) except that the government agency shall not have the right to disclose this software to support service contractors or their subcontractors without the prior written consent of GemStone Systems, Inc.

This software is provided by GemStone Systems, Inc. and contributors “as is” and any expressed or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall GemStone Systems, Inc. or any contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

COPYRIGHTS

PATENTS

GemStone is covered by U.S. Patent Number 6,256,637 “Transactional virtual machine architecture”, Patent Number 6,360,219 “Object queues with concurrent updating”, and Patent Number 6,567,905 “Generational Garbage Collector”. GemStone may also be covered by one or more pending United States patent applications.

TRADEMARKS

GemStone, **GemBuilder**, **GemConnect**, and the GemStone logos are trademarks or registered trademarks of GemStone Systems, Inc. in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Sun, **Sun Microsystems**, **Solaris**, and **SunOS** are trademarks or registered trademarks of Sun Microsystems, Inc. All **SPARC** trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. **SPARCstation** is licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

HP and **HP-UX** are registered trademarks of Hewlett Packard Company.

Intel and **Pentium** are registered trademarks of Intel Corporation in the United States and other countries.

Microsoft, **MS**, **Windows**, **Windows 2000** and **Windows XP** are registered trademarks of Microsoft Corporation in the United States and other countries.

Linux is a registered trademark of Linus Torvalds and others.

Red Hat and all Red Hat-based trademarks and logos are trademarks or registered trademarks of Red Hat, Inc. in the United States and other countries.

AIX and **POWER4** are trademarks or registered trademarks of International Business Machines Corporation.

Other company or product names mentioned herein may be trademarks or registered trademarks of their respective owners. Trademark specifications are subject to change without notice. All terms mentioned in this documentation that are known to be trademarks or service marks have been appropriately capitalized to the best of our knowledge; however, GemStone cannot attest to the accuracy of all trademark information. Use of a term in this documentation should not be regarded as affecting the validity of any trademark or service mark.

GemStone Systems, Inc.
1260 NW Waterhouse Avenue, Suite 200
Beaverton, OR 97006

Preface

About This Manual

GemConnect provides a way for GemStone Smalltalk programmers to load and update relational data Oracle databases into GemStone.

This manual tells you how GemConnect works and how applications can use it to access relational data. The manual provides reference information on GemConnect classes and sample code demonstrating GemConnect functions.

Assumptions

To make use of the information in this manual, you must be familiar with the operation of the GemStone system, as well as the relational database system you are using GemConnect to access. You should also be an experienced GemStone Smalltalk programmer. (See the *GemStone Programming Guide* for information about GemStone Smalltalk.)

How This Manual is Organized

- Chapter 1 provides an overview of GemConnect.
- Chapter 2 discusses how GemConnect works in the context of GemStone and a three-tier client/server architecture.
- Chapter 3 discusses how to set up your connection to the Oracle database.
- Chapter 4 discusses how to use GemConnect to read from and update your Oracle database.
- Appendix A explains how to use the C language source module included with the product to customize GemConnect for your system.
- Appendix B lists and discusses GemConnect errors you may encounter.

Installing GemConnect

This manual assumes that you have a correctly installed GemConnect product, as well as a fully configured and operational GemStone object server, GemBuilder, and an Oracle relational database.

For information on installing the software, see the *GemConnect Installation Guide* for your operating system platform.

Typographical Conventions

This document uses the following typographical conventions:

- Command lines you type in are shown in **bold** type. For example:

  % **env | grep GEM**

- UNIX commands and Topaz commands are also shown in **bold** type. For example:

  copyextent

- Smalltalk methods and instance variables, UNIX file names and paths, and screen dialogue examples are shown in `monospace` type. For example:

  `markForCollection`

- Placeholders that are meant to be replaced with real values are shown in *italic* type. For example:

  StoneName.conf

Technical Support

GemStone provides several sources for product information and support. The product-specific manuals and online help provide extensive documentation, and should always be your first source of information. GemStone Technical Support engineers will refer you to these documents when applicable.

GemStone Web Site: http://support.gemstone.com

GemStone's Technical Support website provides a variety of resources to help you use GemStone products. Use of this site requires an account, but registration is free of charge. To get an account, just complete the Registration Form, found in the same location. You'll be able to access the site as soon as you submit the web form.

The following types of information are provided at this web site:

Help Request allows designated support contacts to submit new requests for technical assistance and to review or update previous requests.

Documentation for GemConnect is provided in PDF format. This is the same documentation that is included with your GemConnect product.

Release Notes and **Install Guides** for your product software are provided in PDF format in the Documentation section.

Downloads and **Patches** provide code fixes and enhancements that have been developed after product release. Most code fixes and enhancements listed on the GemStone Web site are available for direct downloading.

Bugnotes, in the Learning Center section, identify performance issues or error conditions that you may encounter when using a GemStone product. A bugnote describes the cause of the condition, and, when possible, provides an alternative means of accomplishing the task. In addition, bugnotes identify whether or not a fix is available, either by upgrading to another version of the product, or by applying a patch. Bugnotes are updated regularly.

TechTips, also in the Learning Center section, provide information and instructions for topics that usually relate to more effective or efficient use of GemStone products. Some Tips may contain code that can be downloaded for use at your site.

Community Links provide customer forums for discussion of GemStone product issues.

Technical information on the GemStone Web site is reviewed and updated regularly. We recommend that you check this site on a regular basis to obtain the

latest technical information for GemStone products. We also welcome suggestions and ideas for improving and expanding our site to better serve you.

You may need to contact Technical Support directly for the following reasons:

- Your technical question is not answered in the documentation.
- You receive an error message that directs you to contact GemStone Technical Support.
- You want to report a bug.
- You want to submit a feature request.

Questions concerning product availability, pricing, keyfiles, or future features should be directed to your GemStone account manager.

When contacting GemStone Technical Support, please be prepared to provide the following information:

- Your name, company name, and GemStone/S license number
- The GemStone product and version you are using
- The hardware platform and operating system you are using
- A description of the problem or request
- Exact error message(s) received, if any

Your GemStone support agreement may identify specific individuals who are responsible for submitting all support requests to GemStone. If so, please submit your information through those individuals. All responses will be sent to authorized contacts only.

For non-emergency requests, the support website is the preferred way to contact Technical Support. Only designated support contacts may submit help requests via the support website. If you are a designated support contact for your company, or the designated contacts have changed, please contact us to update the appropriate user accounts.

Email: support@gemstone.com

Telephone: (800) 243-4772 or (503) 533-3503

Requests for technical assistance may also be submitted by email or by telephone. We recommend you use telephone contact only for more serious requests that require immediate evaluation, such as a production system that is non-operational. In these cases, please also submit your request via the web or email, including pertinent details such error messages and relevant log files.

If you are reporting an emergency by telephone, select the option to transfer your call to the technical support administrator, who will take down your customer information and immediately contact an engineer.

Non-emergency requests received by telephone will be placed in the normal support queue for evaluation and response.

24x7 Emergency Technical Support

GemStone/S offers, at an additional charge, 24x7 emergency technical support. This support entitles customers to contact us 24 hours a day, 7 days a week, 365 days a year, if they encounter problems that cause their production application to go down, or that have the potential to bring their production application down. Contact your GemStone/S account manager for more details.

Contents

Chapter 1. Product Overview

Chapter 2. GemConnect in the Three-Tier Architecture

Chapter 3. Connecting to Oracle

Chapter 4. Oracle Database Operations

Appendix A. Using the C Source Module

Appendix B. GemConnect Errors

Index

Chapter

1 Product Overview

GemConnect is a set of GemStone Smalltalk classes, methods, and primitives that provides an interface between GemStone and Oracle. With GemConnect, a programmer can load data from the Oracle relational database into objects on the GemStone server, making the relational data available as objects to Smalltalk client applications. In addition, changes to these objects can be written back to the relational database.

Centralizing the mapping between relational data and objects in a shared object server reduces the complexity and increases the performance of your overall system in the following ways:

- Client applications interact with server objects, without having to perform object-to-relational data mapping.
- Client applications need only communicate with the object server, without individual connections to one or more relational databases.
- Objects containing relational data can be cached, eliminating the need to convert the relational data repeatedly.

1.1 What Can GemConnect Do for You?

GemConnect is most useful to you in either of the following situations:

- You support GemStone Smalltalk applications (or you are developing new ones) that need to read and/or update data from a relational database.
- You wish to partition your existing client/server applications and use GemStone to create middle-tier services that read and/or update data in a relational database. This three-tier client-server model is described briefly in this chapter and in more detail in Chapter 2.

GemConnect allows you to connect to a relational database, submit queries and updates to the data, read data from a query into GemStone object form, be notified of changes to the data in GemStone, and flush those changes back to the relational database.

GemConnect also allows you to collect updates to data from multiple concurrent GemStone sessions and post them to the relational database in a batch.

What GemConnect Won't Do

Each Smalltalk vendor has a different interface that it uses to access relational databases. Because GemConnect cannot reasonably include all the possible Smalltalk interfaces, it does not facilitate transparent porting of client-side Smalltalk programs to GemStone Smalltalk.

GemConnect also does not convert foreign key references from the relational database into direct object references in GemStone.

1.2 GemConnect in Context

GemStone and GemConnect are most useful in the development of a three-tier client/server architecture for business applications. For a more complete discussion of the three-tier architecture and GemStone's place in it, see the white paper *Integrating Business Objects Systems with Relational Databases*, published by GemStone Systems, Inc.

The following paragraphs briefly describe the two- and three-tier architectures for client/server operations.

The Two-Tier Approach

In the two-tier approach to client/server applications, the *client* is responsible for user interface and business application processing, and the *server* stores and manages data. This model becomes more inefficient as applications grow larger and become more complex.

The Three-Tier Approach

In the three-tier approach, the *client* handles user interface processing, but the application logic of the system is partitioned between the client and an *object server* like GemStone. The object server provides tools for building reusable business objects that bundle data and processing. These business objects can be shared by many users. The object server also manages access to the existing *relational databases* that store the business's data.

GemConnect and the Three-Tier Approach

GemStone's object application server and GemConnect together make it possible to access the relational data currently stored in your Oracle database and make it available to these reusable business objects. You will be able to run larger and more complex applications far more efficiently than on a typical two-tier client/server system.

The three-tier approach is discussed in more detail in the next chapter, with emphasis on GemConnect's place in the architecture.

Chapter

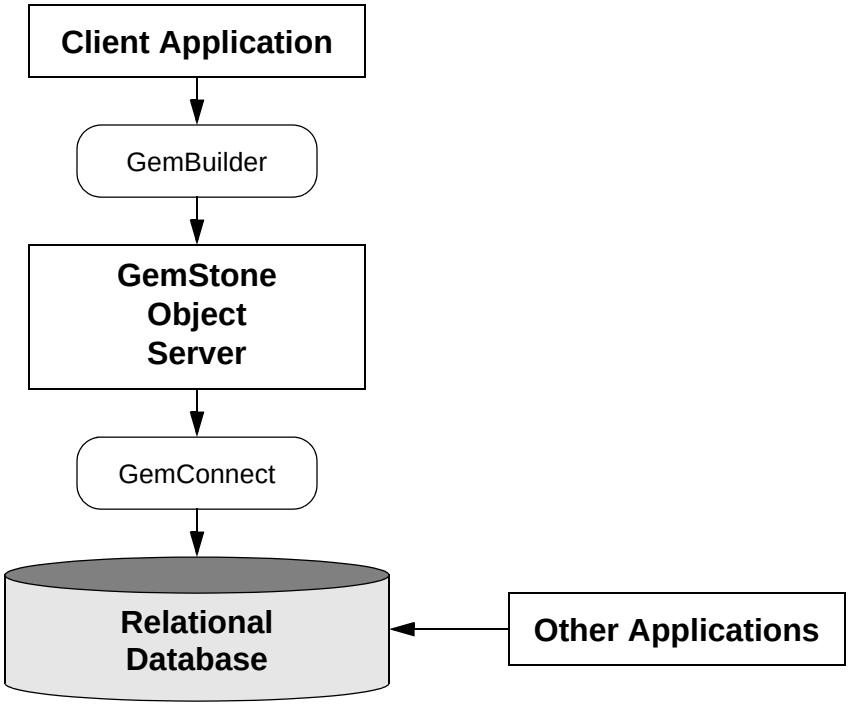
2 GemConnect in the Three-Tier Architecture

The GemStone three-tier client/server architecture includes client applications and relational database servers, with the GemStone object server providing a layer between the applications and the relational databases that organizes and manages data in business objects.

- GemBuilder provides the interface between a client Smalltalk, C, or C++ application and the GemStone object server.
- GemConnect provides an interface between data stored in the relational database and data in the business objects stored in the GemStone object server.

This chapter places GemConnect in the context of the three-tier architecture. Figure 2.1 shows a block diagram of the layers in the three-tier architecture and the interfaces between them.

Figure 2.1 The GemStone Three-Tier Architecture



2.1 Using Tuple Objects to Read and Change Data

Before we get into the details of how GemConnect works with applications, you need to understand a key concept, the tuple object. The *tuple object* is the form in which relational data is carried and manipulated by GemConnect. A tuple object is an instance of a *tuple class*. A tuple class is instantiated in GemConnect to represent one row of data from a database table.

GemStone creates tuple objects to hold and manipulate the data that comes out of the relational database. Client applications that access relational data through GemStone use these tuple objects, which are passed from GemStone to the application by GemBuilder replication mechanisms. Often, this means that a client application has class definitions that correspond to the tuple objects.

Your application can implement an automatic update mechanism by sending a message to a tuple object (or a collection of them) and allowing GemConnect to propagate changes back to the relational database, using the tuple object's ability to generate SQL statements.

2.2 Client Applications and GemBuilder

As shown in Figure 2.1, GemBuilder is the interface between a client Smalltalk, C, or Java application and the GemStone object server. GemBuilder classes establish relationships between client objects and GemStone objects, mapping transparently between the client and GemStone.

GemBuilder also includes *forwarders*, which are client Smalltalk objects whose data and behavior actually reside in GemStone. Forwarders are useful for objects that, for performance reasons, may be too large or complex to replicate on both the client and the GemStone object server. You can use forwarders with GemConnect to process tuple objects on the GemStone server and return other results.

2.3 GemStone Object Server

Business objects contain data and processing rules specific to your enterprise's business situations. The GemStone object server provides an environment in which object-oriented client applications can interact with business objects that persist and can be shared. GemStone provides interfaces for the client applications, manages these business objects, and executes their processing logic.

2.4 GemConnect

GemConnect provides an interface between the relational database and the GemStone object server. GemConnect's functionality includes:

- User actions, classes, and methods to manage connecting to and interacting with the relational database.
- Mapping of relational results into objects. Default or customized column-to-instance-variable mappings are possible, as is automatic data type mapping from relational data to and from GemStone objects.
- Notification of changes to tuple objects.
- Generation of SQL statements to act on relational data.

- Support for updating the relational database immediately or in batch mode.
- A C language source module that allows you to extend GemConnect's user-action library.

2.5 Relational Database

For businesses with large amounts of data stored in relational form, the great advantage of the three-tier architecture is that the relational database is still accessed and updated as before. Applications can read data directly from it and write data to it directly, while other applications which have been partitioned to use GemStone business objects can read and write relational data through GemBuilder and GemConnect.

2.6 How the Architecture Works

GemConnect and the three-tier architecture function together in the following ways.

Reading Data

When a client application obtains data from the relational database, the following steps take place:

1. The client Smalltalk application requests data from a GemStone business object via GemBuilder.
2. Through GemConnect, GemStone creates a connection to the relational database and executes an SQL query.
3. The relational database produces a result set.
4. Through GemConnect, GemStone maps the relational data to an object model, creating tuple objects. The instance variables of the tuple objects contain data from the rows in the result set.
5. GemStone passes the tuple objects back to the application.

Writing Changes

Once an application makes modifications to the tuple object, changes are written back to the relational database and GemStone in the following sequence:

1. The client application sends the modified tuple objects to GemStone and makes a request to commit the changed data.
2. GemStone, through GemConnect, creates, then executes (either immediately or later, in a batch) the SQL statements to write the changed data to the relational database.
3. GemConnect registers connections to be committed to the relational database.
4. GemStone commits its transaction, then the relational database commits its transactions.

Once all registered connections have committed, a message is returned to GemStone that the commit process has succeeded.

Chapter

3 Connecting to Oracle

Before you can read and write to your relational database, you must establish a connection to that database. This chapter discusses how to set up your connection to the Oracle database.

3.1 Setup

Correct operation of GemConnect assumes that, in addition to your client application program, the following system components are available to you:

- A fully configured and operational GemStone object server, with the GemConnect product installed in it.
- An Oracle relational database.

Note that Oracle requires certain environment variables to be set correctly before the language interfaces will operate. If the variables are not set correctly, the interface will not be able to connect to the relational server, resulting in GemConnect connection failures.

- For linked GemBuilder sessions, the client application process must be configured correctly to run Oracle applications.

- For RPC sessions, the account under which the Gem process is run must perform this setup in the appropriate startup file.

For example, if you were running the Gem process on a UNIX computer and connecting to Oracle, the `.profile` file on the UNIX system might contain the following:

```
ORACLE=/pub/oracle
export ORACLE
LD_LIBRARY_PATH=$ORACLE/lib
export LD_LIBRARY_PATH
```

If you intend to access an Oracle database that include strings stored in UTF8 or UTF16 format, you may also need commands similar to the following:

```
NLS_LANG=AMERICAN_AMERICA.UTF8
export NLS_LANG
```

See the Oracle documentation for further details about setting necessary paths and environment variables.

3.2 Connecting to the Relational Database

To connect to a relational database, you use a parameters object and a connection object.

- The *parameters object* sets the characteristics of the connection.
- The *connection object* makes the actual connection to the relational database. It is also used to send update, query, and commit requests to the relational database.

Using Parameters Objects

The parameters object configures the connection object. Keeping the connection configuration information in a parameters object, rather than in the connection object itself, allows the information to be collected at the GemBuilder level and transported down to GemStone.

You can create a parameters object as you need one, or create and store parameters objects as persistent objects in GemStone, either for private or shared use. Persistent parameters objects allow you to store connection information in GemStone and reuse it.

Parameters objects contain a dictionary for parameters not included in the default GemConnect classes. You can add additional connection configuration information to this dictionary. The `GsRdbParameters` class provides messages to access and update the information (`otherAt:` and `otherAt:put:`).

The default parameters include:

- `server` — the name of the Oracle database server.
- `userName` — the Oracle username to use for this connection.
- `password` — the password for the designated Oracle username.
- `textLimit` — The maximum amount of information fetched in Oracle LONG and LONGRAW columns. The default limit is the maximum, 65532 bytes. Values larger than this will be ignored, and the default used.
- `autoCommit` — a Boolean indicating if the connection should use Oracle's autoCommit mode (a commit is performed automatically after each write operation).

To create and configure the parameters object, use the `GsOracleParameters` subclass. Example 3.1 is a code fragment that shows how to create a Oracle parameters object.

Example 3.1

```
params := GsOracleParameters new.
params server: 'server-name';
      userName: 'user-name' ;
      password: 'password'.
```

Using Connection Objects

The connection object controls the interaction with the database, usually by means of SQL commands. When the connection object is sent the `connect` message, it connects to the database, using the information contained in the parameters object. The connection object is an instance of a `GsOracleConnection`.

In Example 3.2, a connection is established using the parameters object created earlier. The connection object is then sent the `connect` message to open the connection to the relational database.

Example 3.2

```
conn := GsOracleConnection newWithParameters: params.
conn connect.
```

Although a connection object can exist without being connected to the relational database, connection objects are meant to be created and used during one GemStone session, then discarded, rather than stored as persistent objects. Connections are specific to a GemStone session. For example, you cannot open a connection in one GemStone session and then attempt to read / write to it from another GemStone session. You cannot use a connection from multiple GemStone sessions.

Accessing UTF8/UTF16 data in Oracle

Instances of GsOracleConnection may automatically convert Oracle data stored in UTF8 or UTF16 format into standard GemStone Strings and DoubleByteStrings. This conversion is controlled by the instance variable `charConversion`. Example 3.3 shows an example of setting the connection to convert data from #UTF8. For this to work correctly, both the Oracle database and the environment must be set up to use the UTF8 character set.

Example 3.3

```
conn charConversion: #UTF8
```

Valid inputs to the `charConversion:` method are: nil, #UTF8, or #UTLF16. If the data cannot be converted, the error #typeConversionError is raised.

Note

The environment variable $NLS_LANG must be set to an appropriate UTF character set (for example, AMERICAN_AMERICA.UTF8) in order to use this feature.

Connection Cache

GemConnect maintains a cache in which to name and store connection objects during a GemStone session. When you establish a new connection to the relational database and name that connection, GemConnect places the connection into the connection cache under that name. Example 3.4 shows how to name a connection and store it in the connection cache.

Example 3.4

```
conn cacheWithName: #NamedConnection
```

You can access a named connection stored in the connection cache with the connection class method `connectionWithName:`.

Example 3.5

```
GsOracleConnection connectionWithName: #NamedConnection
```

If you try to use an existing name for a new connection, GemConnect returns an error.

You can remove a connection from the connection cache by using the connection method removeFromCache:

Example 3.6

```
conn removeFromCache
```

When the GemStone session ends, the connection cache is emptied.

Managing Connections

Within a GemStone session, you can use the class method `lastConnection` to access the last instance of `GsOracleConnection` that was successfully connected to the relational database (whether or not it was stored in the connection cache). Using `lastConnection`, GemStone can use the same interface throughout the session to interact with the database, without having to store the connection as a global variable or in the connection cache. This is useful if you're only establishing one connection at a time.

You could get the same effect by naming the connection and placing it in the cache, but that would require you to check the cache each time before being able to use the connection.

You can gather all the connection instances for a particular session with the `GsOracleConnection >> allConnections` method:

Example 3.7

```
GsOracleConnection allConnections.
```

3.3 Disconnecting from the Relational Database

Connections to a relational database consume system resources. GemStone does not free connection resources automatically until you log out of the GemStone session. When you're finished with a connection object, send a `disconnect` message to the object. This message disconnects the object from the relational database and frees the resources allocated to the connection. Example 3.8 disconnects the connection we established earlier.

Example 3.8

```
conn disconnect.
```

To disconnect all the connection objects at once, use the `GsOracleConnection >> allConnections` method and send `disconnect` to each of the connections returned. This provides a simple way to disconnect all sessions at once.

Once a connection is freed, any instance of read stream (`GsRdbReadStream`) or write stream (`GsRdbWriteStream`) associated with it is invalid. For more about read streams, see Chapter 4, "Oracle Database Operations".

3.4 Checking the status of a connection

You can check if a connection is connected to an Oracle repository or not by using the method `connected`. For example:

Example 3.9

```
conn connected ifFalse: [conn connect].
```

Chapter

4 Oracle Database Operations

Once you have an open connection to the relational database, the connection instance can read, write, and perform other operations on the Oracle database. This chapter discusses how to use GemConnect to read data from and update your Oracle database, and includes the following topics:

Before you attempt to use GemConnect, we suggest that you become familiar with the information in this chapter.

4.1 Executing SQL Statements in Oracle

Reading data from Oracle into GemStone objects, or writing changes to GemStone objects out to corresponding Oracle tables, requires information on the mapping

between objects and table rows. Other types of Oracle operations that can be done in SQL statements can be executed by statements such as this:

Example 4.1

```
conn executeNoResults: <sql statement>
```

For INSERT, DELETE, and UPDATE SQL statements, you can get back the number of rows modified by the SQL statement by using:

Example 4.2

```
numRowsModified := conn executeReturnRowsAffected: <sql statement>
```

You should not use these methods to execute SELECT statements, or to perform COMMIT or ROLLBACK operations.

4.2 Reading Information from the Relational Database

To read information from the relational database, the connection instance can execute SQL statements to generate a result set. You use a read stream to read the data from the result set into GemConnect tuple objects (see page 2-2).

A *read stream* is an instance of the `GsRdbReadStream` class, which is a non-positionable stream over the result of an SQL query to the relational database. Read streams are created as a result of executing a SELECT query. Methods that return read streams are the `GsOracleConnection >> openCursorOn:*` and `execute:*` methods.

This section describes several ways to use read streams to get data from a relational database into a form that your applications can use.

Creating an OrderedCollection

When the result set of an SQL query is returned through the read stream, GemConnect's default behavior is to create an `OrderedCollection` object that contains the column values for each row of the result set.

Example 4.3 shows how to create an `OrderedCollection` object that contains all the results of an SQL `select` statement.

Example 4.3

```
resultStream := conn openCursorOn: 'select * from Books'.
anOrderedCollection := resultStream upToEnd.
```

Creating a Tuple Class During a GemConnect Session

You may find it more useful to create tuple objects to hold the relational data returned by the SQL query in instance variables. The most common way to generate tuple objects from the result set is through the `GsRdbReadStream>> createTupleClassNamed:inDictionary:` method. This method creates a tuple class with the name you supply, installs it in the current read stream, places it in the symbol dictionary you specify, then returns it. See Example 4.4.

Example 4.4

```
resultStream createTupleClassNamed:#aName
      inDictionary:symbolDict
```

Alternatively, you can create a tuple class through the `GsOracleConnection` class, with the `execute:tupleClassName:inDictionary:` message.

Instance Variable Names

The new tuple class's instance variable names are translations of the current result set's column names, if the database returned them. (If no names were returned, the instance variables are called `iv1`, `iv2`, `iv3`, etc.)

The following rules apply when column names are translated to GemStone instance variable names.

1. A column name that begins with a character that is not a letter or an underscore is translated to an instance variable name beginning with a lowercase letter "g." (`1Name` becomes `g1Name`.)
2. A column name containing characters other than letters, numbers, or underscore is translated to an instance variable name with underscores substituted for those characters. (`id#` becomes `id_`.)

   Rule 1 is applied before Rule 2. That is, a column name that *begins with* a character that is not a letter, number, or underscore has a "g" inserted before the character is translated to an underscore. (`$price` becomes `g_price`.)

You can change this translation between column names and instance variables in the public C source module, using the `GsPublicIVsAndConstraints` function.

rdbPostLoad

Each tuple object is also sent a message named `rdbPostLoad` after it is loaded with data from the relational database. By default, the `rdbPostLoad` message does nothing. You can reimplement `rdbPostLoad` to perform such functions as notifying a tuple object of its changes or time-stamping the object.

Converting Relational Data Types to Classes

The data types in each row retrieved from the relational database are mapped to GemStone fundamental data type classes. The new tuple class uses these mappings to convert relational data types to classes, as specified in Table 4.1. (If you have altered the public C source module to change the data type mapping, the new tuple class uses those mappings instead.)

Types not listed in Table 4.1 are converted to instances of `Object`.

Table 4.1 Type Conversion from Oracle to GemStone Classes

Oracle Type	Mapped to GemStone Class
float	Number
float(p)	Number
number	Number (Value returned is an Integer if it contains no decimal part. Otherwise, it is a Float.)
number(p)	SmallInteger or Integer (SmallInteger if precision ≤ 8.)
number(p, s)	SmallInteger, Integer, or ScaledDecimal (If scale ≤ 0 and precision ≤ 8, SmallInteger, with scale subtracted from precision. If scale ≤ 0 and precision > 8, Integer, with scale subtracted from precision. Otherwise, ScaledDecimal.)
date	DateTime
rowid	String (Oracle rowid to string conversion is used)
long	String (Size limited by `GsRdbParameters textlimit`)

Table 4.1 Type Conversion from Oracle to GemStone Classes

Oracle Type	Mapped to GemStone Class
char	String
varchar2	String
raw	ByteArray
longraw	ByteArray (Size limited by `GsRdbParameters textlimit`) (Oracle limitations currently make it impossible to fetch this type of column from GemConnect.)

NOTE

Dates returned from Oracle data that are out of range in GemStone (i.e., dates before January 1, 1901 or after December 31, 1000000) will be converted to `nil` inside a GemConnect tuple object.

Creating a Tuple Class Outside a GemConnect Session

You can create a tuple class before starting a GemConnect session, using GemStone and GemBuilder, then use the tuple class from within GemConnect to map the relational data. See Example 4.5.

Example 4.5

```
Object subclass:       #Book
   instVarNames: # ('id' 'pub' 'date' 'title')
   classVars:          # ( )
   classInstVars:      # ( )
   poolDictionaries:   # [ ]
   inDictionary:        UserGlobals
   constraints:        #[ ]
   instancesInvariant:  false
   isModifiable:        false
```

Example 4.6 shows how to execute an SQL statement and use the tuple class to map the relational data.

Example 4.6

```
resultStream :=
      conn openCursorOn: 'select * from Book'
            tupleClass: Book
```

Relational Data Structure in Tuple Classes

By default, GemConnect places data returned from the first result column into the first instance variable of the tuple object, the second result column into the second instance variable, and so on. However, you might wish to use a different mapping than the default between the column names and the instance variables.

If you reimplement the class method `rdbColumnMapping` in a tuple object's class, GemConnect sends messages to tuple objects to load instance variables with relational column information according to the mapping that you specify. The `rdbColumnMapping` method returns a collection of the correspondences between the name of each column in the relational data and the methods available to access and update the data in that column.

Example 4.7 specifies the mapping between column names and instance variables. Each entry contains (in order) the column name, instance variable name, accessing message, and updating message.

Example 4.7

```
rdbColumnMapping
^#( #( #id    #id   #id   #id:)
    #( #pub   #pub  #pub  #pub:)
    #( #date  #date #date #date:)
    #( #title  #title  #titleCode #settitleFromCode:)
   )
```

If the accessor and the updater have the same name as the instance variable, you need not put them in the column map.

Managing Read Streams

Read streams allocate large memory blocks to hold data being retrieved from Oracle. When you are finished using a read stream, send it the message `free` to

release these memory blocks. If your session continues to create new read streams without freeing them, you may run out of memory.

A way to free all streams opened on a connection is to iterate over the results of sending the message `GsOracleConnection >> allStreams` to your connection. This returns both read and write streams.

4.3 Writing to the Relational Database

You can use GemConnect to generate SQL statements with which to access or update data. SQL strings are not generated automatically. You must explicitly call the SQL generation messages in the GemConnect classes with the appropriate parameters.

Generating the SQL Update Strings

GemConnect provides three SQL generation messages.

To generate a SQL insert string for a tuple object being inserted into a collection, send this message to the tuple object:

`generateSQLInsert:` *theConnection*

To generate a SQL delete string for a tuple object being deleted from a collection, send this message to the tuple object:

`generateSQLDelete:` *theConnection*

To generate a SQL update string for a particular change, send this message to the tuple object:

`generateSQLUpdate:` *theConnection* `instVarName:` *theChangedInstVar* `newValue:` *theNewValue*

GemConnect provides a single method to post updates to the relational database, regardless of how you generated the updates. The `GsOracleConnection` class message `executeNoResults:` accepts the SQL string generated from the change notification message and sends it to the relational database for execution. The message returns only a Boolean result, since there is generally no need for the data to be processed further by GemStone after the update is made.

Tracking the Relational Data

In order to ensure that the correct data in the relational database is updated, the relational data mapped into GemConnect must have three types of information

associated with it: column mapping, the name of the relational database table from which the data came, and primary key information.

Using this information, GemConnect can be sure that when it generates SQL update strings, they will operate on the correct data. There are two ways to associate this identifying information with the relational data.

1. You can use class methods to associate the information that identifies the relational data with the tuple object. In this approach, the tuple object must reimplement three methods with which to retrieve the necessary information:
   - `rdbTableName` — Returns the name of the relational table into which the changed data in the tuple object will be loaded.
   - `rdbColumnMapping` — Returns an array of column name-to-instance variable pairs. Each element in the array corresponds to a column to be updated in the table.
   - `rdbPrimaryKeyMaps` — Returns the column mapping entries that correspond to the primary key for each instance. This mapping identifies the row to update in the relational database table.

   The SQL generation methods use this information to identify where the changed data belongs in the relational database.
2. Alternatively, you can specify the identifying information to the lower-level versions of the SQL generation methods.
   - `generateSQLInsert:table:columns:`
   - `generateSQLDelete:table:keys:`
   - `generateSQLUpdate:instVarName:newValue:table:columns:keys:`

Change Notification

Since GemStone or your application may alter the data read from the relational database, you need to know which tuple objects are about to be modified so that you can update the relational database.

By default, GemConnect does not notify you when tuple objects are changed, since change notification can have a negative impact on system performance.

You can, however, implement the `rdbPostLoad` message for each tuple class that holds the relational data so that GemConnect will notify you when an instance variable in a tuple object is about to change. To get notification of changes, use the message `notifyChange: true`. See Example 4.8.

Example 4.8

```
rdbPostLoad
      self notifyChange: true.
```

NOTE

Some GemStone server product versions do not support Change Notification. When using those GemStone versions, any attempt to use Change Notification will return an error.

To stop notification of changes, use the message `notifyChange: false`.

To determine whether a tuple object is having its changes tracked, send it the message `notifyChange`.

NOTE

If an object held by one of the instance variables in a tuple object changes, the tuple object is not notified.

In addition to being notified of changes to tuple objects, you can also be notified of insertions and deletions that affect `UnorderedCollection` and `SequenceableCollection` objects. To enable this notification, use the `notifyChange:` message.

Tracking How an Object Has Changed

Each time an instance variable of a tuple object is about to be modified, detailed information about the change is passed to the object by the message `aboutToChange:newValue:`. You can use this information to write the changes out to the relational database.

Similarly, detailed change information is sent to nonsequenceable (unordered) collection objects by the `aboutToAdd:` and `aboutToRemove:` messages.

Detailed change information is sent to sequenceable collection objects by the `aboutToInsert:index:` and `aboutToDelete:index:` messages.

For operations such as `at:put:` or `copyFrom:to:into:startingAt:` on sequenceable collection objects, when change notification is enabled, the message `aboutToChange:newValue:` is sent for each element of the collection affected by the change. In this case, the first argument is the offset into the collection (or unnamed instance variable) where the change will take place. The second argument is the new value that will be stored at that offset.

The following classes will notify you only if instance variables change:

- `AbstractDictionary` (and all its subclasses)
- `RcCounter`
- `RcQueue`

Invariant, special, and byte-indexable classes cannot be configured to notify you of changes.

Change Notification Sequence

Change notification takes place in the following sequence:

1. The client application makes changes to data originally loaded from the relational database. At some point, these changes are flushed to the GemStone tuple object.
2. If change notification is turned on for the tuple object, the GemStone object manager sends an `aboutToChange:newValue:` message before the instance variable in the tuple object changes. The application must reimplement the method that handles this message.
3. Since the change notification includes information describing how the tuple object is about to be changed, the application may use the change notification to generate SQL statements with which to update the relational database.
   - For a discussion of the `generateSQL*` messages, see "Generating the SQL Update Strings" on page 4-7.
4. The changes may be written directly to the relational database, either in the `aboutToChange:newValue:` message or immediately after it returns, or they may be queued for later updates. (If the changes are written through immediately, the data in the relational database is changed before the changes take effect in GemStone.)
   - For a discussion of these two approaches — immediate write-through and queued updates, see "Writing Changes" on page 4-11.

Example 4.9 writes a change through to the relational database.

Example 4.9

```
aboutToChange: offset newValue: newInstVar
| conn |
conn := GsOracleConnection connectionWithName:
 #OracleConnection.
conn executeNoResults:
 (self generateSQLUpdate: conn
   instVarName: (self class allInstVarNames at: offset)
   newValue: newInstVar).
```

In Example 4.10, the change is written to a queue for later update.

Example 4.10

```
aboutToChange: offset newValue: newInstVar
| conn |
conn := GsOracleConnection connectionWithName:
 #OracleConnection.
conn queue: RcQueue new.
conn queue add:
 (self generateSQLUpdate: conn
   instVarName: (self class allInstVarNames at: offset)
   newValue: newInstVar).
```

Writing Changes

Whenever data is changed, and change notification is turned on (page 4-8), you're notified of changes made to tuple objects. At that point, you have several options in terms of handling the changes and sending them to the relational database. You can write changes immediately or queue updates and make them all at once. Once you've updated the database, you can commit those changes.

Immediate Write-Through

You can write the changed data through to the relational database immediately. As discussed earlier (beginning on page 4-9), the `aboutToChange:newValue:` message returns information about how instance variables in the tuple objects were changed. GemConnect uses this information to generate SQL statements through the connection object to update the relational database. (The process for

updating from collections is similar, except that GemConnect uses information from the `aboutToAdd:`, `aboutToRemove:`, `aboutToInsert:index` and `aboutToDelete:index` messages.)

Queueing Updates

If you were to update the relational database immediately every time a tuple object was modified, performance would be very slow. GemConnect allows you to collect updates from a single GemStone session or even from multiple concurrent sessions and post all the updates to the database in one batch. There are several ways of accomplishing this.

The GemConnect connection objects provide an instance variable for holding a queue object. If you want to write the updates made in one session out to the relational database, do the following:

1. Create an `RcQueue` or `RcIdentityBag` (or other collection) instance when you create the connection object.
2. Collect the updated information and add it to the queue.
3. Before you close the connection, send all the accumulated updates to the relational database.

The information in the queue can take many forms.

- The objects in the queue might be the actual SQL update strings generated by the `aboutToChange:newValue:` method.
- Alternatively, the queue objects might be tuple objects loaded from the database that contain instance variable information.

To accumulate update information for multiple concurrent sessions, you can create a persistent instance of `RcQueue` or `RcIdentityBag`. See “Making Queues Persistent” on page 4-27.

Queueing Update Messages

One approach to queueing is to generate the SQL statements from the change notification information and queue them for later execution. (For information about writing changes to a queue, see Example 4.10 on page 4-11.)

Example 4.11 shows how to flush queued messages.

Example 4.11

```
conn queue do: [:sqlString |
  conn executeNoResults: sqlString
   ].
```

Queueing Changed Objects

The preferred way to write changed data to the relational database is to queue the change information needed to generate the SQL updates. There are two variations of this process.

1. For cases in which the objects are very large or do not change much, we recommend that you queue the object with its change information. When the update queue is flushed, GemConnect generates the SQL statements necessary to update the relational database.

   Example 4.12 shows how to queue the entire changed object, along with the information about what has changed.

Example 4.12

```
aboutToChange: offset newValue: newInstVar
| chgivs conn |
conn := GsOracleConnection connectionWithName:
      #GsOracleConnection
chgivs := conn queue at:self ifAbsent:[nil].
(chgivs isNil) ifTrue: [
  chgivs:= conn queue at:self put:(IdentitySet new)
  ]
chgivs add:(self class allInstVarNames at: offset).
```

NOTE

In this approach, if you change a primary key name in the object, you must be certain to include that information with the queued object. Otherwise, GemConnect will not be able to locate the appropriate location in the relational table to update.

Example 4.13 shows how to flush the queue of changed objects.

Example 4.13

```
conn queue doKeysAndValues: [:tuple :chgs |
      colsNvals := tuple getNamesAndValuesFor: (tuple
            rdbColumnMapping) scope: chgs.
      keysNvals := tuple getNamesAndValuesFor: (tuple
            rdbPrimaryKeyMaps).
      sql := conn class generateSQLUpdateForTable: (tuple
                  rdbTableName)
            columns:colsNvals
            keys: keysNvals.
      conn executeNoResults: sql].
```

2. Another approach to queueing changed objects is to set up `rdbPostLoad` to copy the entire tuple object when it is made available to the application, making sure that change notification is turned on for the object. When you receive notification that the object is about to be changed, you queue the changed object for a later update. At update time, you compare the original with the changed version of the tuple object, and then generate the SQL statements to update the relational database.

   Examples 4.14 through 4.17 demonstrate this approach.

Example 4.14

```
"Preserving state of object"
rdbPostLoad
  self notifyChange: true.
  self tagAt: 1 put: self copy.
```

Example 4.15

```
"Add object to queue when changed"
aboutToChange: offset newValue: newValue
  |conn|
conn := GsOracleConnection connectionWithName:
 #OracleConnection.
conn queue add: self
```

Example 4.16

```
"Flush the queue"
conn queue do: [:obj |
  conn executeNoResults: (obj generateSQLUpdate: conn)
   ].
```

Example 4.17

```
"Generate the SQL statements"
generateSQLUpdate: conn
|sql original ivs scope|

original := self tagAt: 1.
original isNil ifTrue: [
"If no original values, this is a new object. Generate
insert."
  sql := self generateSQLInsert: conn
  ]
ifFalse: [
"If there are original values, generate update"
  scope := OrderedCollection new.
  ivs := self class allInstVarNames.
  1 to: (ivs size) do: [:i |
     (original instVarAt: i) = (self instVarAt: i)
            ifFalse: [scope add: (ivs at: i) ].
     ].
  sql := self generateSQLUpdate: conn scope: scope.
  ].
^sql
```

Both of these approaches show the clear advantage of queueing update objects (rather than update messages). If you were to queue update messages, you would queue (and later execute) one SQL statement for each change made to each tuple object. By queueing objects, you generate all the updates for a single object with one SQL statement.

4.4 Using WriteStreams for Batch Operations

If your application generates a number of similar SQL statements, such as a group of SQL inserts to the same table, or a group of updates changing the same column, you can improve performance by using a write stream.

In GemConnect, a *write stream* is an instance of `GsRdbWriteStream` that allows you to add a tuple (or a collection of tuples) containing changes to be written to the relational database. Using write streams, you can perform SQL updates, inserts, and deletes without having to worry about constructing SQL statements.

`GsRdbWriteStream` is a subclass of `GsRdbReadStream`.

Creating a Write Stream

Methods on class `GsOracleConnection` allow you to create instances of `GsRdbWriteStream` that perform either Oracle INSERT, DELETE, or UPDATE operations. A given write stream can only perform one type of operation, and only for a single tuple class.

As described earlier, GemConnect uses three types of information to ensure that the correct data in the Oracle database is updated: column mapping, the name of the relational database table from which the data came, and primary key information. This information allows GemConnect to determine the mapping between GemStone tuple classes and the associated Oracle database tables.

You can use `GsOracleConnection` methods to override the default table name, column mapping, or primary key mapping when useful.

Column Mapping

As discussed earlier in this chapter (page 4-6), column mapping establishes the correspondences between the name of each column in the relational data and the methods available to access and update the data in that column. Each entry in the column map contains (in order) the column name, instance variable name, accessing message, and updating message.

When you use write streams to perform SQL INSERT and UPDATE operations, the entries in the column maps must be ordered in the same sequence in which they appear in the Oracle table.

For example, if you have an Oracle table with rows in this order:

```
('ID', 'LAST_NAME', 'FIRST_NAME', 'ADDRESS')
```

then your column map information must order them as shown in Example 4.18.

Example 4.18

```
#((ID id id id:)
(LAST_NAME lastName lastName lastName:)
(FIRST_NAME firstName firstName firstName:)
(ADDRESS address address address:))
```

You can also supply a tuple class of Array, in which case you only supply abbreviated column map information with two elements:

Example 4.19

```
#((ID id)
(LAST_NAME lastName)
(FIRST_NAME firstName)
(ADDRESS address))
```

When you pass an Array to `nextPut:`, the elements in the array are mapped first-to-last as specified by the column mapping.

Primary Key Mapping

Primary key mapping is used to associate column mapping entries with the primary key for each instance. Key mapping identifies the row to update in the relational database table.

For SQL DELETE and UPDATE operations, the entries in the key maps must be ordered in the same sequence in which they appear in the Oracle table. In addition, for UPDATE operations, all entries in the key map must also be present in the column map.

Performing INSERT Operations

Inserting data into the Oracle database is a two-part process:

1. First, you create the write stream by sending the message `openInsertCursorOn:` (or one of its variants) to the connection object.
2. Once you have created the write stream, you can add tuple objects to it. To add a single entry, send the message `nextPut:` to the write stream, with the tuple object as the argument.

To add a collection of tuple objects iteratively to the stream, use the message `nextPutAll:` instead. See Example 4.20.

Example 4.20

```
writeStream := conn openInsertCursorOn: TupleClass.
writeStream nextPutAll: TupleInstanceCollection.
conn commitTransaction.  "This also makes sure that
writeStream is flushed"
```

If you need to override the default column mapping or database table name, you include the `columnMapping:` and `tableName:` keywords:

`openInsertCursorOn:` *theTupleClass* `columnMapping:` *columnMap*

or

`openInsertCursorOn:` *theTupleClass* `columnMapping:` *columnMap* `tableName:` *theTableName*

Performing DELETE Operations

Example 4.21 uses the `openDeleteCursorOn:` method to delete the contents of TupleInstanceCollection.

Example 4.21

```
writeStream := conn openDeleteCursorOn: TupleClass.
writeStream nextPutAll: TupleInstanceCollection.
conn commitTransaction.
```

You can also delete data using arrays. Example 4.22 uses the `keyMapping:` keyword to override the default key mapping "ID" for ID = 1001, 1002, 1003.

Example 4.22

```
writeStream := conn openDeleteCursorOn: Array keyMapping:
#((ID id)).
writestream nextPut: #(1001).
writestream nextPut: #(1002).
writestream nextPut: #(1003).
conn commitTransaction.
```

Alternatively, you could use `nextPutAll:` to perform the same operation:

Example 4.23

```
writeStream := conn openDeleteCursorOn: Array keyMapping:
#((ID id)).
writestream nextPutAll: #((1001)(1002)(1003)).
conn commitTransaction.
```

If you need to override the database table name, you can send this message:

`openDeleteCursorOn:` *theTupleClass* `keyMapping:` *keyMap* `tableName:` *theTableName*

Performing UPDATE Operations

Example 4.24 uses the `openUpdateCursorOn:` message to update the column "ADDRESS", using "ID" as the key, for the specified collection of tuple objects.

Example 4.24

```
writeStream := conn
  openUpdateCursorOn: TupleClass
  columnMapping: #((ID id id id:)(ADDRESS address address address:))
  keyMapping: #((ID id id id:)).
writeStream nextPutAll: TupleInstanceCollection.
conn commitTransaction.
```

Example 4.25 is a variation that uses Arrays in lieu of TupleClass. (Because you don't need to specify accessor and updater messages for Array, the column map information is simplified.)

Example 4.25

```
writeStream := conn
  openUpdateCursorOn: Array
  columnMapping: #((ID id)(ADDRESS address))
  keyMapping: #((ID id id:)).
writeStream nextPut: #(101 '123 West 12th Ave').
writeStream nextPut: #(102 '707 North Elm St').
conn commitTransaction.
```

As with INSERT and DELETE, you can include the `tableName:` keyword to override the database table name:

`openUpdateCursorOn:` *theTupleClass* `columnMapping:` *columnMap* `keyMapping:` *keyMap* `tableName:` *theTableName*

Note that for `openUpdateCursorOn:` messages, the columns specified in the keyMap must also be included in the columnMap.

Buffering Behavior

Each `GsRdbWriteStream` includes a buffer for holding entries waiting to be written to the Oracle database, sized according to the connection's batch size at the time the stream is created. (See "Batch Size" on page 4-21.)

When you commit a transaction (by sending the message `commitTransaction` to the GsOracleConnection), GemConnect automatically flushes the contents of the write stream buffer to the Oracle database. Flushes also happen automatically when the buffer is filled.

Similarly, when you roll back a transaction (by sending the message `rollbackTransaction` to the GsOracleConnection), GemConnect clears the contents of the write stream buffer without writing them to the Oracle database.

If you manually perform an Oracle commit or rollback, GemConnect does not automatically flush or clear the write stream buffer. Instead, you must perform these operations explicitly, so as to avoid either losing data, or writing data to Oracle that you did not intend. See Example 4.26 and Example 4.27, respectively.

Example 4.26

```
writestream flush.
conn executeNoResults: 'commit'.
```

Example 4.27

```
writestream clear.
conn executeNoResults: 'rollback'.
```

Dependency Lists

During a commit, instances of `GsRdbWriteStream` are flushed in the reverse order of when they were created — newest streams are flushed first. You can override this order by using the stream's `dependencyList`. Any streams listed in the `dependencyList` will be flushed prior to the flush of the primary stream.

To add a stream to the stream's dependency list:

Example 4.28

```
wrStream addDependency: anotherWriteStream.
```

Similarly, to remove a stream from the stream's dependency list:

Example 4.29

```
wrStream removeDependency: anotherWriteStream.
```

Batch Size

By default, the write stream processes a maximum of 20 rows at a time. You can send the message `GsOracleConnection >> batchSize:` to define a different batch size. Once you have specified a batch size, this value is used for all subsequently generated read streams and write streams.

Example 4.30

```
conn batchSize "Retrieve the current setting of
batchSize"

conn batchSize: 20  "Set the batchSize to 20"
```

As a result of buffering, when GemConnect finally does the write to Oracle, there may be multiple errors generated for different objects. Recovering from these multiple errors may require your application to maintain internal information about these past operations so that they can be replayed in case of a flush error. In some cases, you might choose to forego the performance improvement that buffering provides and to override buffering altogether, so that the Oracle write occurs immediately after each object is put into the writeStream via a `nextPut:` or `nextPutAll:` call.

To override buffering, set the batchSize to 1. For example:

Example 4.31

```
conn batchSize: 1  "Override buffering"
```

Managing Write Streams

Write streams allocate large memory blocks to hold data being buffered to write to Oracle. When you are finished using a write stream, send it the message `free` to release these memory blocks. If your session continues to create new write streams without freeing them, you may run out of memory.

Because `free` also performs a `flush` (page 4-20), you should `clear` the buffer first if you don't want the contents flushed to the Oracle database.

A way to free all streams opened on a connection is to iterate over the results of sending the message `GsOracleConnection >> allStreams` to your connection. This returns both read and write streams.

4.5 Committing Changes

Once you've written changes to the relational database, you must commit them for the changes to persist. In order to maintain consistency in your data, you must commit changes to both the relational database and the GemStone objects.

Oracle Transaction Control

You use instances of `GsOracleConnection` to commit or roll back transactions in the relational database. `GsOracleConnection` includes the following methods for controlling transactions in the relational database manually:

- `commitTransaction` — Commits the current transaction to the relational database.
- `rollbackTransaction` — Rolls back the current transaction in the relational database (similar to Gemstone's ABORT operation).

Synchronizing GemStone and Relational Database Transactions

GemConnect also provides a level of synchronization between a GemStone transaction and relational database transactions.

While this level of synchronization does not provide a true two-phase synchronized commit mechanism, it does enforce consistency in the way GemStone and relational database transactions are committed, as well as create a structure that will subsequently accommodate a true two-phase commit process.

Commit List

The synchronized commit process provided by GemConnect begins with the commit list. Connection instances are registered to allow their transactions to be associated with and committed as part of the local GemStone Session.

To register a connection, use the `GsOracleConnection >> addToCommitList` method, which adds a given connection to the commit list. (To unregister a connection, use `GsOracleConnection >> removeFromCommitList`.)

You synchronize transactions by executing the following GemStone methods:

- `System >> commitTransaction`
- `System >> abortTransaction`

(The `System Class >> beginTransaction` method does not support synchronizing transactions.)

NOTE

The `commitTransaction` *mechanism uses the GemStone* `System Class >> continueTransaction` *method to provide voting for GemStone sessions. This method has side effects that may affect your data's consistency temporarily. For details, see the class and method comments in the GemStone image.*

Commit Sequence

When a commit is requested, the following sequence of events takes place:

1. The commit list in the local GemStone session is checked for any registered connections.
2. If there are none, the local GemStone transaction is committed normally. If there are registered connections, the local GemStone session is asked to vote to commit.
3. If the local GemStone session votes affirmatively, the registered connections are each asked in turn to vote to commit.

   NOTE

   GemConnect cannot ask for a true vote to commit from a relational database connection. Therefore, any vote by a GemConnect connection registered in the commit list will always return `true`*, as a vote to commit.*

   If the GemStone session or any registered connection returns a negative vote to commit, the `System commitTransaction` call returns `false` and the commit process is aborted. Otherwise, if the GemStone session and all registered connections vote to commit, the commit process continues.
4. The local GemStone session is committed.

   If the GemStone commit is unsuccessful, the `System commitTransaction` call returns `false` and the commit process is aborted.
5. Assuming this commit is successful, each registered connection in the commit list is committed in turn.

   If any registered connection fails to commit, the `System commitTransaction` call returns `false` and the commit process is aborted.

   If the GemStone transaction and all registered connections commit successfully, the `System commitTransaction` call returns `true`.

Note that the window of time between voting and committing makes it possible for one connection to commit while another does not. At this time, there is no way to resynchronize connections if such a failure takes place.

Checking Results

To determine which registered connections voted to commit, you can send the `voteResults` message to the commit list object `System _commitCoordinator`. The `voteResults` message returns an array containing a value for each connection committed to the list.

`0` — The connection was read-only and voted affirmative.

`1` — The connection was read / write and voted affirmative.

`2` — The connection voted negative

`nil` — The connection did not vote because a connection or connections before it in the commit list voted negative.

To determine the commit results for registered connections, use the `commitResults` method. This message returns an array containing either a Boolean value or `nil`.

`true` — The connection committed successfully.

`false` — The connection failed to commit.

`nil` — The connection did not commit because a connection or connections before it in the commit list failed to commit.

Oracle Names

By default, names in Oracle are case-insensitive. This means that “book,” “Book,” and “BOOK” are all interpreted as the same name. Oracle can force a name to be treated as case-sensitive by enclosing the name in double quotes. GemConnect supports this behavior.

GemConnect enforces the relational column naming rules required by Oracle. Certain column names must be surrounded by double quotes (`"6column_name"`) to be interpreted correctly, both by Oracle and by GemConnect.

The following types of column names must be double-quoted in Oracle:

- Any column name that does not begin with a letter.
- Any column name that includes characters other than letters, numbers, underscore, number sign (#), or dollar sign ($).

- Any column name the Oracle user wants designated as case-sensitive. (All other column names are treated as case-insensitive and are returned with all letters capitalized.)

In addition, any column name created in Oracle with double quotes, regardless of whether it contains any of the characters that must be treated specially, must always be referred to with the double quotes surrounding it, as it is now considered to be case-sensitive.

GemConnect and Oracle Column Names

GemConnect treats Oracle column names that must be double-quoted in the following ways:

- For queries, column names that are expected to be quoted must be double-quoted or Oracle will return an error. If the results of a query are stored in a tuple class with a column map, the map must include double-quoted column names where necessary. When using column maps, GemConnect matches double-quoted column names to instance variable names in a case-sensitive manner and unquoted column names in a case-insensitive manner. The double quotes are stripped from the column names when this matching takes place.

  For example, the Oracle column name `"10_GaLLons"` matches a GemConnect instance variable named `g10_GaLLons`, but the Oracle column name `GaLLons` matches a GemConnect instance variable named `GALLONS`.

Example 4.32

```
rdbColumnMapping
^(#(GaLLons,
    GALLONS))
```

- For generating SQL statements from GemConnect, any column maps or primary key maps with column names that require double-quoting must include double quotes around those names. If the double quotes are not present where expected, the SQL command is invalid and Oracle will return an error.

4.6 Multi-Session Operation

Thus far, all of the discussion in this chapter has focused on how GemConnect operates in a single session. GemConnect can also operate effectively in a situation where you have more than one GemConnect session running at a time, sharing business objects.

To ensure consistency across the GemConnect sessions, you want any queues that you use to collect updates to the relational database to become persistent objects, so that multiple users may share them. You may also wish to make some parameters objects persistent, to make certain connections easier to reproduce.

Making Queues Persistent

To accumulate update information for multiple concurrent sessions, do the following:

1. Create a persistent instance of `RcQueue` or `RcIdentityBag` and name it in a shared symbol list dictionary.
2. Write the information needed to generate SQL updates or the SQL updates themselves to the `RcQueue` or `RcIdentityBag` in `aboutToChange:newValue:` methods or elsewhere.
3. Have the queue processed periodically by a scheduling task like UNIX's **cron**.

Making Tuple Classes Persistent

You can make a tuple class persistent by placing it in a global variable in GemStone. For details, see the *GemStone Programming Guide.*

Making Parameters Objects Persistent

You can make the parameters objects that you use to set the characteristics for a connection available to other connections if you make them persistent. To make a parameters object persistent, you need to place it in a global dictionary. See Example 4.33.

Example 4.33

```
params := GsOracleParameters new
params userName: 'user-name'; password: 'password';
      server: 'server-name'.
UserGlobals at:#params put: params.
```

Appendix

A Using the C Source Module

GemConnect is not a complete collection of the relational database interfaces that every Smalltalk applications program might need. Since your program may need access to functions that GemConnect does not provide, we have provided a means for you to extend and customize GemConnect for your own purposes.

The C language source module is supplied for Oracle, the relational database management system that this version of GemConnect supports. The C language module includes public functions that you can reimplement to provide additional functionality to GemConnect. You need only have an appropriate C compiler to recompile the source code after you're through changing it. See the documentation for your version of the GemStone / S or GemStone / S 64 Bit server to determine the correct compiler to use.

A.1 Using the Public C Source Module

The install process places a C source file named `gsorapublic.c` into the `make` directory of the GemConnect tree.

You can use the source module to add new primitives to your GemConnect system, to change the default data type mappings that GemConnect uses, and to change the process by which GemConnect logs into the relational database. Each

source file includes some sample primitives to guide you in developing other relational database interfaces. You can also extend GemConnect in other ways.

The `make` directory also includes a makefile to recompile the C module and rebuild the shared user-action library for your operating system platform. Once you edit the source file and recompile it, you need to copy the new user action library to the GemStone user action library directory (`$GEMSTONE/ualib`) before the new functionality will be available to GemConnect.

NOTE

You will have to preserve any site-specific changes to the C source module and remake and rebind the user-action library whenever you install a new version of GemConnect.

Edit Source File

Each source file includes a set of sample primitives as an aid to developing your own. You may reimplement them to add functionality to GemConnect. Table A.1 lists these public functions and describes when they are called. For details about what the functions do by default, see comments in the source file itself.

Table A.1 Public Source File Functions

Function Name	When Called
`GsPublicInitialize`	During user-action initialization, after the relational database API is configured.
`GsPublicShutdown`	During `GciUserActionShutdown` after connections have been deallocated, but before database contexts have been freed.
`GsPublicConnectAlloc`	After a connection is allocated, but before it is opened.
`GsPublicConnected`	After a connection has been opened.
`GsPublicDisconnect`	Before a connection is closed.
`GsPublicReadStreamAlloc`	After a read / write stream is allocated, but before it is opened.
`GsPublicReadStreamInit`	After a read / write stream has been initialized and its command executed.

Table A.1 Public Source File Functions (Continued)

Function Name	When Called
GsPublicReadStreamDrop	Before a read / write stream is deallocated.
GsPublicSetClassAndConversion	To fill in column-mapping conversions.
GsPublicConvertToObject	To convert a column datum into an object.
GsPublicIVsAndConstraints	To retrieve instance-variable and constraint information from a column map.

Changing Data Type Conversions

You can alter the default data type conversions by editing the appropriate file, then rebuilding and reinstalling the shared user action library.

To do this, reimplement the GsPublicSetClassAndConversion and GsPublicConvertToObject methods, use the appropriate makefile to recompile the shared library, then copy it to the appropriate directory. For details, see “Run Make” on page A-5 and “Install Into GemStone ualib Directory” on page A-6.

Tailoring the Relational Database Login Process

To change how GemConnect logs into the relational database, reimplement the C source module functions GsPublicConnectAlloc and GsPublicInitialize. (Use the otherAt: parameters from GsRdbParameters to specify new parameters, if necessary.)

A.2 Rebuilding the GemConnect Library

This section describes the steps necessary to rebuild the GemConnect shared library after you've changed the public C source module.

Assuming you have the correct environment variables set and you have an ANSI-compliant C compiler, rebuilding the shared library is as simple as running the `make` process.

Environment Variables

Before you run the `make` process, be sure that you have the GEMSTONE environment variable set, as well as the environment variables to point to the relational database software (ORACLE_HOME).

NOTE

This example shows settings for GemStone and GemConnect running on a SPARCstation with the Solaris operating system.

Step 1. Set the GEMSTONE environment variable. (In this example, *installDir* is the pathname of the directory where GemStone is installed, starting with a slash.)

C shell:

```
% setenv GEMSTONE installDir/GemStone6.1.5-sparc.Solaris
```

Bourne or Korn shell:

```
$ GEMSTONE=installDir/GemStone6.1.5-sparc.Solaris
$ export GEMSTONE
```

Step 2. Set the GEMCONNECT environment variable. (This example assumes that GemConnect is installed in the same top-level directory as GemStone.)

C shell:

```
% setenv GEMCONNECT installDir/GemConnect2.0+oracle10.0-sparc.Solaris
```

Bourne or Korn shell:

```
$ GEMCONNECT=installDir/GemConnect2.0+oracle10.0-sparc.Solaris
$ export GEMCONNECT
```

Step 3. Set the ORACLE_HOME environment variable to point to the directory containing the relational software (in this example, `/pub/oracle`).

C shell:

```
% setenv ORACLE_HOME /pub/oracle
```

Bourne or Korn shell:

```
$ ORACLE_HOME=/pub/oracle
$ export ORACLE_HOME
```

Step 4. Set the library path variables so that GemStone can reach the relational database software.

IMPORTANT NOTE

If you are running GemStone/S 6.x in an Oracle installation that supports both 32-bit and 64-bit Oracle, replace `$ORACLE_HOME/lib` *with* `$ORACLE_HOME/lib32`.

C shell:

```
% setenv LD_LIBRARY_PATH $ORACLE_HOME/lib
```

Bourne or Korn shell:

```
$ LD_LIBRARY_PATH=$ORACLE_HOME/lib
$ export LD_LIBRARY_PATH
```

NOTE:

There may be additional environment parameters that need to be set for your platform. See the comment fields in the makefile for details.

Run Make

In the `make` directory, run the makefile to rebuild your GemConnect shared library.

Step 5. Change to the `make` directory.

```
% cd $GEMCONNECT/make
```

Step 6. Execute the makefile. The correct makefile to execute depends on the server version and platform you are using.

For GemStone/S:

```
% make MakeFile32
```

For GemStone/S 64 Bit version 1.1.x:

```
% make MakeFile641
```

For GemStone/S 64 Bit version 2.1.x:

```
% make MakeFile642
```

Compiler Notes for Solaris

You must have the ANSI C compiler to rebuild the API. The C compiler is distributed with Solaris is not ANSI-compliant.

Install Into GemStone ualib Directory

Once you've rebuilt the library, you must copy it to the shared user-action library directory.

Step 7. Copy the new shared user action library to the GemStone `ualib` shared library directory.

Windows:

```
> cp oraapi20-32.dll %GEMSTONE%/ualib
```

UNIX (GemStone/S 6.x):

```
% cp liboraapi20-32.so $GEMSTONE/ualib
```

UNIX (GemStone/S 64 Bit v1.x):

```
% cp liboraapi20-641.so $GEMSTONE/ualib
```

UNIX (GemStone/S 64 Bit v2.x):

```
% cp liboraapi20-642.so $GEMSTONE/ualib
```

A.3 Adding a User Action

This example shows you how to implement a new user action.

Declaring the User Action

You must create the new user action, adding it to the `gsorapublic.c` source file. Example A.1 demonstrates this. This sample code is included in the `gsorapublic.c` source file shipped with GemConnect.

Example A.1

```
/*============================================================
 * Name - GsColumnInfo
 *
 * Returns an array of arrays, each with the following
 * information about a column in the result set of a
 * particular read stream:
 *
 *      1) column name
 *      2) data type (see cstypes.h for integer values)
 *      3) max-length
 *      4) status (see cstypes.h for bit values)
 *      5) precision
 *      6) scale
 *      7) class of field when translated to an object
 *         (actual datum may be a subclass of this class)
 *
 * This user action may be called from Smalltalk by
 * invoking:
 *
 *        values := System userAction: #GsOraColumnInfo
 *                      with: stream connection with: stream.
 *
 *============================================================
 */

OopType GsColumnInfo ARGS2(
    OopType, connectOop,
    OopType, readStreamOop)
{
    int i;
```

```
    GsConnection *conn;
    GsReadStream *stream;
    GsCmap  *cmap;
    OopType elements[6];
    OopType element;
    OopType result;
    char *name;

    /* check to see if we have a valid connection object, if
not return nil */
    for (conn=AllConnections; conn != NULL; conn=conn->next)
{
        if (conn->object == connectOop)
            break;
    }
    if (!conn)
        return OOP_NIL;

    /* check to see if we have a valid read stream object,
if not return nil */
    for (stream=conn->readStreams; stream != NULL; stream=
stream->next) {
        if (stream->object == readStreamOop)
            break;
    }
    if (!stream)
        return OOP_NIL;

    /* create the array we will be returning the column
information in */
    result = GciNewOop(OOP_CLASS_ARRAY);

    for (i=0; i<stream->numColumns; i++) {
        cmap = &(stream->columns[i]);
        name = &(cmap->columnName[0]);
        elements[0] = GciNewByteObj(OOP_CLASS_STRING,
(ByteType *)name,
                                    strlen(name));
        elements[1] = GciLongToOop(cmap->datafmt.datatype);
        elements[2] = GciLongToOop(cmap->datafmt.maxlength);
        elements[3] = GciLongToOop(cmap->datafmt.status);
        elements[4] = GciLongToOop(cmap->datafmt.precision);
```

```
            elements[5] = GciLongToOop(cmap->datafmt.scale);
            elements[6] = cmap->instVarClass;
            element = GciNewOop(OOP_CLASS_ARRAY);
            GciStoreOops(element, 1, elements, 7);
            GciStoreOop(result, i+1, element);
        }

        return result;
    }
```

Implementing the User Action

Now you must register the user action so that GemStone can see it.

Example A.2

```
    static GciUserActionSType allActions[] = {
      { "GsOraColumnInfo",                  2, (UA)GsColumnInfo
    },
      { "GsOraInstVarsAndConstraints",      2,
    (UA)GsPublicIVsAndConstraints },
      };
```

Calling the User Action

Implement a GemStone method on `GsRdbReadStream`.

Example A.3

```
method: GsRdbReadStream
columnInfo

   ^System userAction: (connection _rscolumnInfo)
          with: connection with: self
%
```

Then implement a GemStone method on a connection subclass.

Example A.4

```
method: GsOracleConnection
_rscolumnInfo

   ^#GsOraColumnInfo
%
```

Once you've finished implementing the user action, you need to:

- Recompile the public source file or files that you've edited by running the make process again (see "Run Make" on page A-5).
- Copy the new shared library into the `$GEMSTONE/ualib` directory (see "Install Into GemStone ualib Directory" on page A-6).

Now your new user action is available through GemConnect.

Appendix

B GemConnect Errors

This appendix contains a summary of GemStone system problems that might affect GemConnect operation, as well as a list of the GemConnect error messages.

For further information about GemStone restart and diagnostic procedures, see the *GemStone System Administrator's Guide*. If errors appear to originate with Oracle, you may also need to consult the system documentation for your relational database management system.

B.1 Troubleshooting

When GemConnect will not operate, it may be difficult to find the reason, especially if GemStone and your Oracle server are running on different machines. Problems on other machines may not be visible from your workstation.

Failure to Start

If you cannot log in to GemStone, one of the following situations may have occurred:

- The limit on the number of simultaneous GemStone sessions has been reached.
- The Stone process is not running.

- The `netldi` process is not running.
- A network connection has failed.
- GemStone user account is set up incorrectly.

If you cannot log in to Oracle, one of these situations may have occurred:

- Your GemStone repository has not been upgraded to include the GemConnect classes.
- The Oracle relational database server is not installed on the machine GemStone is running on.
- The Oracle relational database server is not running.
- User accounts on the relational database have not been configured correctly.
- The `$ORACLE_HOME` environment variables have not been set or they point to the wrong place.

B.2 GemConnect Error Messages

GemConnect may produce any of the error messages listed in Table B.1. Error messages that do not appear in this list come from another source, either from GemStone or from your relational database system.

GemConnect errors are returned in the following form:

errorname — errorText arguments

Table B.1 GemConnect Errors

Error Message	Arguments	Explanation
`aggColumnName` — Column names of the form `agg`*XX* are not allowed in results.	*connection identifier, column name*	GemConnect uses the form `agg`*XX* as a default for instance variable names (where *XX* is an integer identifier), and thus cannot return a relational column with a name of that form.

Table B.1 GemConnect Errors (Continued)

Error Message	Arguments	Explanation
`badColumnMap` — A column map entry with too few or non-character collection elements was encountered.	*column map entry*	The specified column mapping scheme does not contain enough elements. Each column map entry must have at least two values: column name and instance variable name. These values must be of type `CharacterCollection`.
`badConnCacheName` — There is already a connection cached with this name.	*connection name*	A connection with this name already resides in the named connection cache.
`columnAllocError` — A buffer could not be allocated to bind the values for a column in the result set.	*connection identifier, column name*	A memory allocation (`malloc`) failure occurred when GemConnect tried to allocate memory for a binding buffer.
`columnBindingError` — Could not bind all columns in columnMap with relational table.	*connection identifier, query*	
`failedToInitialize` — The specified relational database interface could not be initialized.	*none*	GemConnect could not initialize the Oracle database's client interface.

Table B.1 GemConnect Errors (Continued)

Error Message	Arguments	Explanation
`flushError` — Problem writing to relational database during write stream flush.	*connection identifier, stream, details*	One or more Oracle errors occurred during the flush of the associated write stream. Note that because of buffering, the `#nextPut:` or `#nextPutAll:` operation that provided the object causing this error may have occurred some time earlier.
`internalError` — GemConnect internal error.	*connection identifier, stream, details*	This is an unexpected logic error from within GemConnect. Contact GemStone Technical Support with background information on this error plus the contents of the *details* array, especially the last entry containing the diagnostic string.
`invalidColumnMap` — A column map was passed which contained elements which were not of the expected class Array.	*connection identifier, column map*	The column map was not an Array or elements of it were not in an Array.
`invalidConnection` — This relational database connection is not valid.	*connection identifier*	The connection to Oracle was not established or it has been disconnected.
`invalidSql` — Invalid SQL statement for cursored execution.	*connection identifier, query*	`openCursorOn:` may not be used with fully qualified non-SELECT SQL statements.
`invalidStream` — This relational database result stream is not valid.	*stream identifier*	The stream to Oracle did not open successfully or it has been closed.

Table B.1 GemConnect Errors (Continued)

Error Message	Arguments	Explanation
`invalidTableName` — The table does not exist in this relational database.	*table name, connection identifier*	No table with the given name exists in Oracle
`invalidTupleInstance` — A tuple was passed to a write stream that is not the expected class.	*connection identifier, stream, tuple*	
`noChangeNotification` — Object change notification is not supported on this version of GemStone.	none	The server product and version to which GemConnect is connected does not support object change notification.
`noColumnMap` — There is no column map available to perform the requested action with this tuple class.	*class name*	The tuple class did not specify a column mapping scheme for the method to use.
`noPrimaryKeyMap` — There is no primary key map available to perform the requested action with this tuple class.	*class name*	The tuple class did not specify a primary key mapping scheme for the method to use.
`noTableName` — There is no table name available to perform the requested action with this tuple class.	*class name*	The tuple class did not specify a table name for the method to operate on.
`oracleError` — An unexpected error was encountered during Oracle processing.	*connection identifier, write stream identifier, details*	GemConnect caught an unspecified error generated by Oracle. The *details* are in the format described in Table B.2.

Table B.1 GemConnect Errors (Continued)

Error Message	Arguments	Explanation
`queryError` — An error was encountered while performing a relational query.	*connection identifier, write stream identifier, details*	GemConnect caught an error returned from Oracle while trying to execute a query. The details are in the format described in Table B.2.
`readError` — An error was encountered while reading relational data.	*read stream identifier*	GemConnect caught an error returned from Oracle while reading data from a result set.
`streamAtEnd` — End of stream was encountered while reading relational data.	*read stream identifier*	GemConnect caught an error returned from Oracle when the RDBMS reached the end of a result set during a read using a stream.
`typeConversionError` — Cannot convert GS Object for Oracle column.	*connection identifier, stream, details*	A GemStone object being converted for an Oracle column is incompatible with the Oracle column DataType.
`unmappedColumnInResult` — A column map was given, but one or more result columns were not represented in it.	*column name*	A tuple class specified a column map for data coming into it, but the SQL query results contained a column for which a mapping was not specified.

GsOracleConnection Class >> messages

When an error occurs, you can obtain additional information about the related Oracle error(s) by sending:

```
GsOracleConnection messages
```

This message returns an array of error fields, as listed in Table B.2.

Table B.2 Oracle error results

1	Oracle Call Interface (OCI) return code	Usually -1
2	Number of errors	Usually 1, unless there are multiple errors on a writeStream flush
3	Oracle error code	
4	Oracle error message	
5	GemStone object	The object written to a writeStream that generated this error
6..N	When multiple errors are returned simultaneously, the array includes elements 3–5 (Oracle error code, Oracle error message, GemStone object) for each subsequent error.	
last	Internal diagnostic message	A string containing internal diagnostic information, for use by GemStone Technical Support in tracking down unexpected errors.

Index

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

W