User Guide

March 2005

Version 9.0

This manual describes the Dharma Software Development Kit (SDK). It describes implementing ODBC, JDBC and .NET access to proprietary data and considerations for creating a release kit to distribute the completed implementation.

March 2005

Dharma Systems welcomes your comments on this document and the software it describes. Send comments to:

Documentation Comments

Dharma Systems, Inc.

Brookline Business Center.

#55, Route 13

Brookline, NH 03033

Phone: 603-732-4001

Fax: 603-732-4003

Electronic Mail: support@dharma.com

Web Page: http://www.dharma.com

Contents

3 Implementation Strategy

Introduction

PURPOSE OF THIS MANUAL

This manual describes the Dharma Software Development Kit (SDK). It describes implementing ODBC, JDBC and .NET access to proprietary data and considerations for creating a release kit to distribute the completed implementation.

This manual complements the material in the *Dharma SDK SQL ReferenceManual, Dharma SDK ODBC Driver Guide, Dharma SDK JDBC Driver Guide, Dharma SDK ISQL Reference Manual* and *Dharma SDK .NET Data Provider Guide* that contain instructions and reference material for administrators and programmers.

AUDIENCE

This manual is intended for a variety of audiences, including any reader who needs to understand and assess the benefits of the Dharma SDK. In addition, this document is also intended for programmers implementing the storage interfaces to a proprietary storage system

STRUCTURE

This manual contains the following chapters:

Chapter 1	Introduces the features of the Dharma SDK and describes how it works.
Chapter 2	Describes installing the Dharma SDK development components and setting up the supplied sample implementation of the storage interface routines.
Chapter 3	Discusses different approaches to mapping proprietary to relational tables and details a series of stages for implementing the storage interfaces.
Chapter 4	Lists the files that must be included on release kits for distributing a completed Dharma SDK implementation to other systems.
Chapter 5	Provides detailed reference material on the C storage interfaces to proprietary storage systems.
Chapter 6	Provides detailed refernce material on Java Stubs Storage interface to proprietary storage systems..
Appendix A	Contains reference information on utilities used to configure the Dharma Server.
Appendix B	Details the structure of the system catalog tables.

Appendix C	Describes storing and returning values using the internal Dharma SDK storage format for SQL NUMERIC values.
Appendix D	Contains a glossary of terms you should know.

CONVENTIONS

The Dharma SDK supports both UNIX and Microsoft Windows environments. Symbols in the left margin indicate material that is applicable to a specific environment.

Indicates steps specific to UNIX.

Indicates steps specific to Windows.

RELATED DOCUMENTATION

Dharma SDK SQL Reference Manual	This manual describes syntax and semantics of SQL language statements and elements for the Dharma SDK .
Dharma SDK User Guide	This manual describes the Dharma Software Development Kit (SDK).It describes implementing JDBC, ODBC and .NET access to proprietary data and considerations for creating a release kit to distribute the completed implementation.
Dharma SDK ISQL Reference Manual	This manual provides reference material for the ISQL interactive tool provided in the Dharma SDK environment. It also includes a tutorial describing how to use the ISQL utility.
Dharma SDK ODBC Driver Guide	This manual describes Dharma SDK support for ODBC (Open Database Connectivity) interface and how to configure the Dharma SDK ODBC Driver.
Dharma SDK JDBC Driver Guide	Describes Dharma SDK support for the JDBC interface and how to configure the Dharma SDK JDBC Driver.
Dharma SDK .NET Data Provider Guide	This guide gives an overview of the .NET Data Provider. It describes how to set up and use the .NET Data Provider to access a Dharma SDK database from .NET applications.
Microsoft ODBC Programmer's Reference, Version 3.0	Describes the ODBC interface, its features, and how applications use it.

Chapter 1

Introduction

1.1 OVERVIEW

The Dharma SDK (Software Development Kit) is the fastest and easiest route to providing industry-standard Open Database Connectivity (ODBC), Java Database Connectivity(JDBC) and .NET access to any proprietary database. With the Dharma SDK, you're one step away from opening your data to the following benefits:

- Plug-and-play interoperability with a vast selection of Windows and Web tools
- Accessibility from any client platform to any server operating system
- Technology that reduces development and testing time, and minimizes deployment and support issues

The Dharma SDK gives you these benefits at a fraction of the cost of implementing ODBC/JDBC/.NET support using other methods.

1.2 CLIENT/SERVER AND DESKTOP CONFIGURATIONS

Dharma offers the Dharma SDK in Desktop and Client/Server configurations:

- The Dharma SDK Client/Server configuration provides network access to your proprietary data. The Dharma SDK Server library runs on the UNIX or Windows 2000/XP server hosting the proprietary storage system. The ODBC tool and the Dharma SDK ODBC Driver, or the JDBC application and the Dharma JDBC Driver run on Windows 2000/XP or UNIX clients. The .NET application and the Dharma SDK .NET Data Provider run on the Windows 2000/XP client.
- The Dharma SDK Desktop configuration is supported only on Windows 2000/XP and implements a “single-tier” ODBC architecture where the ODBC tool, the Dharma SDK software and the proprietary data all reside on the same Windows computer.

Dharma's technology provides simplified development and seamless access from ODBC/JDBC and Web tools to data in your proprietary storage system.

To help get you started, both configurations include a complete sample implementation, with source code, that you can adapt to your specific requirements.

1.2.1 Client/Server Configuration

The Dharma SDK Client/Server configuration is distributed as a ready-to-link server library and client side drivers for ODBC, JDBC and .NET

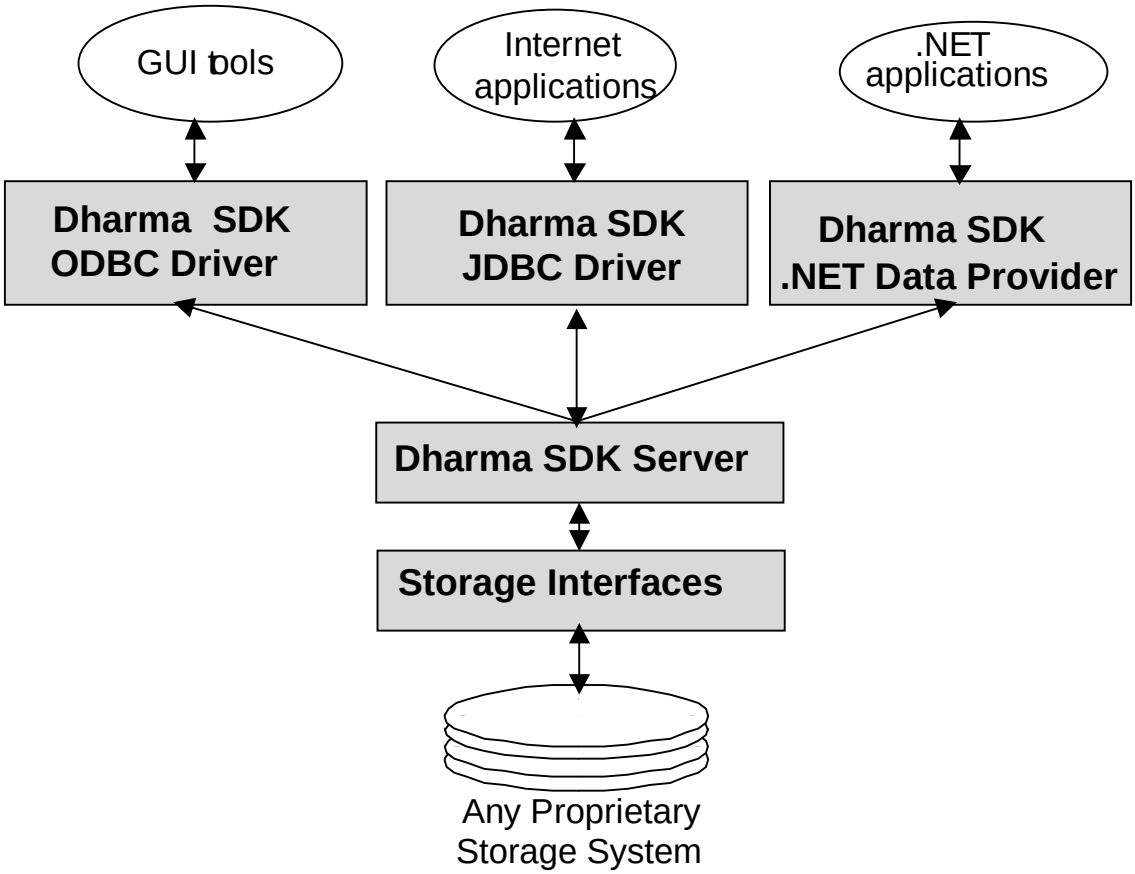
- The Dharma SDK ODBC Driver processes ODBC function calls from ODBC applications/tools that request data from the proprietary storage system.

Similarly the Dharma SDK JDBC Driver processes JDBC API calls from JDBC applications. The Dharma SDK .NET Data Provider services requests from .NET applications. The Dharma SDK Drivers connects to the Dharma SDK Server, translates the standard SQL statements into syntax the data source can process, and returns data to the application. The Dharma SDK ODBC Driver runs on Windows 2000/XP/NT/98 clients as well as UNIX clients. The Dharma SDK JDBC Driver runs on UNIX and Windows2000/XP/NT clients. The Dharma SDK .NET Data Provider runs on Windows 2000 and XP clients.

- The Dharma Server library runs on the server hosting the proprietary storage system. You link it with storage interfaces to your proprietary storage system.

The following figure shows the components in the Dharma SDK Client/Server.

Figure 1-1: Components in the Client/Server Dharma SDK



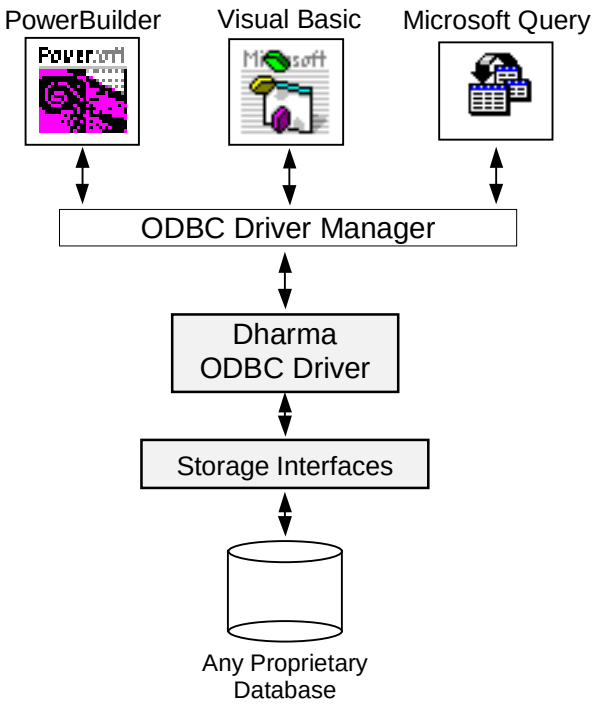
1.2.2 Desktop configuration

The Dharma SDK Desktop configuration is distributed as a ready-to-link ODBC driver library. JDBC and .NET clients are not supported in this configuration.You link

it with storage interfaces to your proprietary storage system to provide access from Windows, XP and Web tool directly to your data.

The following figure shows the components in Dharma SDK Desktop.

Figure 1-2: Components in the Dharma Desktop SDK



1.3 STORAGE INTERFACES

The Dharma SDK supports two types of Storage Interfaces. C Storage stubs for interfacing with Proprietary databases written in C/C++ language and Java Storage stubs for interfacing with Proprietary databases written in Java language.

1.4 SQL FEATURE SUPPORT

The Dharma SDK includes the following SQL features:

- An extensive set of scalar functions
- Table and column privileges
- Queries that use outer joins and set operators

- SQL support for subqueries, derived tables, views, and case expressions
- Performance optimizations for access to very large databases.

1.5 BENEFITS

Tools Interoperability

The Dharma SDK ODBC Driver is compatible with Version 3.52 of the Microsoft ODBC standard. It supports all Core and Level 1 API functions, and all Level 2 functions required by Windows and Web tools.

The Dharma SDK JDBC Driver is JDBC 1.3 standard compliant.The Dharma SDK JDBC Driver is a type 4 Driver.

Dharma extensively tests the Dharma SDK to insure that it provides seamless compatibility with the latest versions of all popular Windows-based tools, including PowerBuilder, Microsoft Access, Microsoft Visual Basic, and Crystal Reports. Leading Web servers that have been tested with the Dharma SDK include Netscape Communication Server and Microsoft Internet Information Server.

In addition, Dharma maintains marketing relationships with key client/server vendors and tests pre-release versions of their products to insure continued compatibility.

Fast Development

With Dharma's SDK, instead of months – or years – of development investment, you can implement robust ODBC/JDBC/.NET access to proprietary storage systems in weeks. All without any changes to the underlying storage system.

Unlike other access solutions, which require that you provide 20 percent of the coding effort, Dharma's technology dramatically decreases the need for code development. Implementers are freed from issues such as:

- Accessing and managing of metadata
- Choosing the best method for retrieving user data
- Implementing additional sorting and buffering mechanisms
- Implementing update logic if only read access is required

With the Dharma SDK, typical storage template implementations require only 500 to 1,000 lines of C or Java source code. Compared to other technologies, the minimal coding required avoids extensive testing and support burdens.

1.6 IMPLEMENTING ACCESS TO PROPRIETARY DATA

Extending ODBC/JDBC/.NET access to your proprietary data is a simple process:

1. Implement C or Java storage interface templates (also called stubs in this manual) to access user data in the proprietary system.
2. Link the implemented storage interfaces with the Dharma SDK library to create a DLL or executable in case of C stubs. For Java stubs, compile the Java files to create new classes.
3. Enter metadata that describes the proprietary data layout

To get started, read Chapter 2 to find out how to install Dharma SDK development components and use the supplied sample implementation. Then, see Chapter 3 for specifics on implementing storage interfaces to access your own proprietary storage system.

Chapter 2

Getting Started

2.1 INTRODUCTION

This chapter describes how to get started using the Dharma SDK. This includes installing the development component and setting up and accessing the supplied sample implementation of the Dharma SDK. The following sections describe these steps for the Desktop and Client/Server configurations.

Other chapters describe how to proceed with your own implementation for your proprietary storage system. For example,

- Chapter 3 describes how to implement the storage interfaces. In particular, section 3.4 describes how to build a Dharma SDK Server executable image from your implementation and set up access to it.This section describes how to build the classes for the Java storage stubs as well.
- Chapter 4 describes considerations for creating a release kit to install your completed implementation of the Dharma SDK Server on systems with the proprietary storage system.

2.2 REQUIRED SOFTWARE

This section summarizes the following:

- The operating systems that the Dharma SDK supports
- How to install development components from the CD-ROM
- The compilers used to build completed implementations of the storage interfaces
- On Sun Solaris, HP-UX and Linux, use the Unix ODBC Driver manager which is available for free download from the following URL http://www.unixodbc.org (unixODBC-2.2.10.tar.gz)

The following table summarizes the supported operating systems and compilers.This table lists the contents of Dharma SDK Components in the CD-ROM as well.

Table 2-1: Summary of Supported Operating Systems and Compilers

Operating System	To Install Dharma SDK Components from CD-ROM	Compiler
Microsoft Windows 2000, Windows XP (Desktop)	Run the setup program in the following subdirectory appropriate to your license **Desktop - DHPRODT90**	Microsoft Visual C++ Version 6.0
Microsoft Windows 2000 (Client/Server)	Run the setup program in the following subdirectories appropriate to your license **Server – DHPROSVR90** **ODBC Driver – DHODBC90** **JDBC Driver – DHJDBC90** **.NET provider – DHDOTNET90**	Microsoft Visual C++ Version 6.0
Sun Solaris Version 8.0/9.0 (Client/Server only)	Untar the file in the _SOL subdirectory appropriate to your license: **Server - DHPROSVR90.SOL** ODBC Driver - DHODBC90.SOL JDBC Driver - **DHJDBC90.SOL**	Sun *cc Workshop compilers* 5.0
IBM AIX Version 4.3.1 (Client/Server only)	Untar the file in the _AIX subdirectory appropriate to your license: **Server – DHPROSVR90.AIX** **ODBC Driver – DHODBC90.AIX** **JDBC Driver - DHJDBC90.AIX**	C compiler included with C Set++ for AIX Version 3.1.4. RTE level 3.6.4.
SCO OpenServer Version 5.0.2 (Client/Server only)	Untar the file in the _SCO subdirectory appropriate to your license: **Server – DHPROSVR90.SCO** **ODBC Driver – DHODBC90.SCO** **JDBC Driver - DHJDBC90.SCO**	Optimizing C Compiler (from OpenServer Development System Version 5.0.2).
HP-UX Version 11.0 (Client/Server only)	Untar the file in the _HP subdirectory appropriate to your license: **Server – DHPROSVR90.HP** **ODBC Driver – DHODBC90.HP** **JDBC Driver - DHJDBC90.HP**	HP Native C Compiler.
Red Hat Linux Version 9.0 (Client/Server only)	Untar the file in the _LIN subdirectory appropriate to your license: **Server – DHPROSVR90.LIN** **ODBC Driver – DHODBC90.LIN** **JDBC Driver - DHJDBC90.LIN**	GNU Project C and C++ Compiler Version 3.3.1

SDK for Java product requires sun JDK version 1.4.2 to build the Java storage stubs. C/C++ information given in the above table is not applicable for SDK for Java.

2.3 DESKTOP

As described in section 1.2.1, the Desktop configuration combines the Dharma SDK ODBC Driver and the Dharma SDK Server in a single executable file that provides access to a proprietary storage system on the same Windows system. The Dharma Desktop SDK runs on Windows XP and Windows 2000.

2.3.1 Installing Development Components

The Dharma SDK Desktop configuration is distributed on CD-ROM. To install the development components:

1. Insert the CD-ROM and execute the *setup.exe* program.
2. Answer the queries from setup, including where to create the directory for the development components. Examples in this section use the *%TPEROOT%* directory.

The installation creates the directory structure shown in the following figure. Table 2–2 includes descriptions of the components in each directory.

Figure 2-1: Dharma SDK Desktop Directories and Files for C stubs

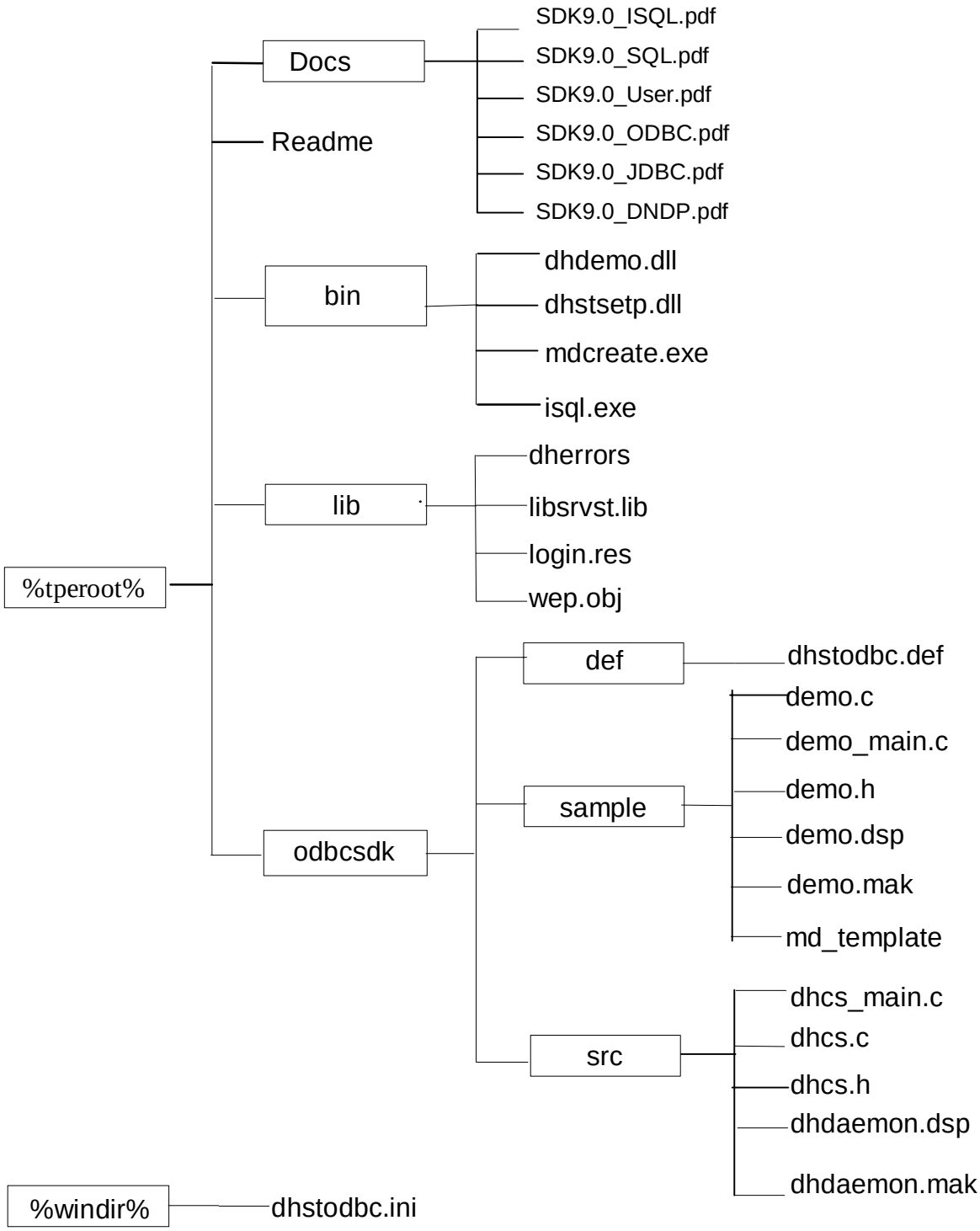


Table 2-2: Summary of Dharma SDK Desktop Development Components for C stubs.

File	Description
Docs/SDK9.0_User.pdf	Dharma SDK User Guide in .PDF format.
Docs/SDK9.0_SQL.pdf	Dharma SDK SQL Reference in .PDF format.
Docs/SDK9.0_ODBC.pdf	Dharma SDK ODBC Driver Guide in .PDF format.
Docs/SDK9.0_JDBC.pdf	Dharma SDK JDBC Driver Guide in .PDF format.
Docs/SDK9.0_DNDP.pdf	Dharma SDK .NET Data Provider Guide in .PDF format.
Docs/SDK9.0_ISQL.pdf	Dharma SDK ISQL Reference in .PDF format
Readme	Online version of installation instructions, including any additional notes not included in the printed documentation
bin\dhdemo.dll	ODBC sample DLL, pre-built from files in sample directory (copy the file to *dhstodbc.dll* to use the sample implementation)
bin\dhstsetp.dll	Setup DLL for adding ODBC data sources
bin\mdcreate.exe	Utility to create a data dictionary
bin\isql.exe	Utility for loading metadata and executing simple SQL queries
lib\dherrors	Dharma error mapping file
lib\libsrvst.lib	Dharma SQL engine library (links with implemented storage interfaces)
lib\login.res	Object file to link with implemented storage interfaces
lib\wep.obj	Object file to link with implemented storage interfaces
odbcsdk\def\dhstodbc.def	Definitions file of interfaces exported from the DLL
odbcsdk\sample\demo.c	Source file for sample implementation
odbcsdk\sample\demo_main.c	Source file for sample main() function implementation
odbcsdk\sample\demo.h	Header file for sample implementation
odbcsdk\sample\demo.dsp	Visual C++ project file for building the ODBC driver sample implementation
odbcsdk\sample\demo.mak	Makefile for building the ODBC driver from the sample implementation
odbcsdk\sample\md_template	Template script for loading metadata
odbcsdk\src\dhcs.c	Source file for stubs
odbcsdk\src\dhcs_main.c	Source file main() function in stubs
odbcsdk\src\dhcs.h	Header file for stubs
odbcsdk\src\dhdaemon.dsp	Visual C++ project file for building the ODBC driver for the proprietary storage system
odbcsdk\src\dhdaemon.mak	Makefile for building the ODBC driver for the proprietary storage system

Table 2-2: Summary of Dharma SDK Desktop Development Components for C stubs.

File	Description
%windir%\dhstodbc.ini	Initialization file containing environment variables (placed in the directory specified by the *windir* environment variable)

Figure 2-2: Dharma SDK desktop directories and files for Java stubs

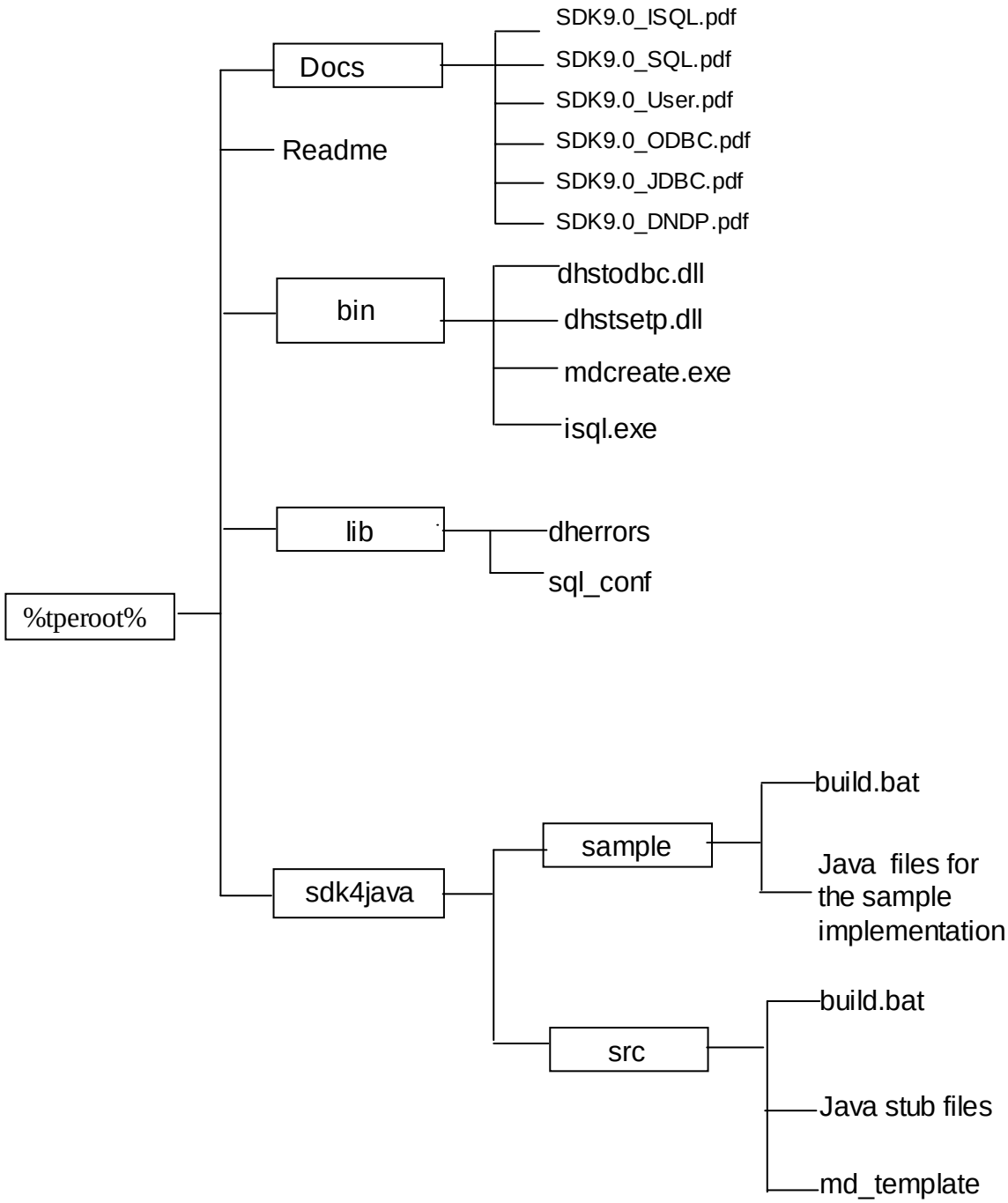


Table 2-3: Summary of Dharma SDK Desktop Development Components for Java stubs.

File	Description
Docs/SDK9.0_User.pdf	Dharma SDK User Guide in .PDF format.
Docs/SDK9.0_SQL.pdf	Dharma SDK SQL Reference in .PDF format.
Docs/SDK9.0_ODBC.pdf	Dharma SDK ODBC Driver Guide in .PDF format.
Docs/SDK9.0_JDBC.pdf	Dharma SDK JDBC Driver Guide in .PDF format.
Docs/SDK9.0_DNDP.pdf	Dharma SDK .NET Data Provider Guide in .PDF format.
Docs/SDK9.0_ISQL.pdf	Dharma SDK ISQL Reference in .PDF format
Readme	Online version of installation instructions, including any additional notes not included in the printed documentation
bin\dhstodbc.dll	ODBC DLL
bin\dhstsetp.dll	Setup DLL for adding ODBC data sources
bin\mdcreate.exe	Utility to create a data dictionary
bin\isql.exe	Utility for loading metadata and executing simple SQL queries
lib\dherrors	Dharma error mapping file
lib\sql_conf	Configuration file used by the isql
sdk4java\src\build.bat	Batch file that is used to build the classes
sdk4java\src*java	Java files that customers are expected to fill in.
sdk4java\src\md_template	Template script for loading metadata
sdk4java\sample\build.bat	Batch file that is used to build the classes.
sdk4java\sample*java	Source files for sample implementation.

2.3.2 Renaming the Desktop Sample Implementation

2.3.2.1 SDK for C stubs

The installation procedure creates an already-built version of the sample Dharma SDK implementation in the file *bin\dhdemo.dll*.

To use the sample implementation, you must copy or rename the file to *dhstodbc.dll*.

Note that if you rename the file, it will be overwritten when you build the Dharma SDK DLL from your implementation (see section 3.4.1.1). If that happens, and you want to rebuild the sample implementation, execute the makefile *odbcsdk\sample\demo.mak*. Open and build the *demo.mak* file in Microsoft Visual C++ to create the Dharma SDK DLL for the sample implementation.

2.3.2.2 SDK for Java stubs

SDK for Java provides classes for a sample java storage system. These class files are present in %TPEROOT%\classes directory. Building the classes after the stub implementation will overwrite these classes. Hence, you may want to backup the existing files in classes directory to use the sampple implementation later. To rebuild the sample java storage system classes, execute build.bat file provided in the %TPEROOT%\sdk4java\sample directory.

2.3.3 Loading Metadata

Metadata defines SQL tables and indexes that map the structure of data in a proprietary storage system to standard relational forms. The Dharma SDK includes utilities to create a data dictionary for your proprietary storage system (the *mdcreate* utility) and load metadata into it (the *isql* utility).

The executable %TPEROOT%\bin\mdcreate.exe is a utility to create a data dictionary that stores metadata. Invoke the *mdcreate* utility and supply a name that will be used for the data dictionary and for access to the sample implementation. For example:

```
%TPEROOT%\bin\mdcreate demo_db
```

The *mdcreate* utility creates a subdirectory called *dbname.dbs* under the

%TPEROOT% directory and populates the directory with the necessary files. For instance, the preceding example creates the directory %TPEROOT%\demo_db.dbs.

The executable *%TPEROOT%\bin\isql* is a tool for loading metadata. It accepts a script with special SQL CREATE TABLE and CREATE INDEX statements that insert metadata for existing tables.

The sample implementation includes a script that loads the metadata for several tables. (As part of the implementation process, you create such a script for existing tables in your proprietary storage system. See section 3.3.1.1).

To load the metadata for the sample, invoke *isql* to execute the script file *%TPEROOT%\odbcsdk\sample\md_template for C* and %TPEROOT%\sdk4jdbc\sample\md_template for SDK for Java. The following example shows invoking *md_template* to create metadata for a database called *demo_db*:

For SDK for C

isql -s %TPEROOT%\odbcsdk\sample\md_template demo_db

For SDK for Java

```
isql -s %TPEROOT%\sdk4java\sample\md_template demo_db


        Dharma/isql Version 09.00.0000
Dharma Systems Inc              (C) 1988-2005.
Dharma Systems Pvt Ltd          (C) 1988-2005.
```

The *isql* command has other options for additional flexibility. See the *isql* reference section in Appendix A and *Dharma SDK ISQL Reference Manual* for a more detailed description of the *isql* command.

2.3.4 Adding Names of ODBC Data Sources

Use the Microsoft ODBC Administrator utility to add names of specific data sources you want to access.

1. Invoke the Microsoft ODBC Administrator from Windows (by default, from the Control Panel program group). The Administrator's Data Sources dialog box appears.
2. Click on the System DSN tab. A list of existing system data sources appears.
3. Click on the Add… button. The Create New Data Source dialog box appears.
4. In the list box, double-click on the Dharma SDK Desktop driver. The Dharma ODBC Setup dialog box appears.
5. Enter information in the following text boxes:
   - **Data Source Name:** — the name of the ODBC data source for use in ODBC connect calls and by the ODBC Administrator.
   - **Description:** — An optional descriptive string.
   - **Database:** — The database name you specified when you invoked the *mdcreate* utility to create the data dictionary (see section 2.3.3).
   - **User ID:** — The user name for the process.
   - **Password:** — The password for the process.
   - **Data Dir:** — The location of the data dictionary directory. Leave this field blank unless the *mdcreate* command used the -d argument (see Appendix A). (If it did, specify the same value here as that used in the -d argument.)

You must supply the name of the data source. If you omit the database name, user name, or password, the driver prompts the ODBC application user for that information when it connects to the data source.

The ODBC Administrator utility updates the ODBC Driver Manager registry entry with the information supplied in the dialog box.

2.4 CLIENT/SERVER

With Dharma SDK Client/Server, you need to complete steps on both the server system and the client system. In addition, you need to execute *makefiles* to build a Dharma SDK Server executable image for the supplied sample implementation.

The following sections describe these steps:

- Installing the Dharma SDK development components
- Setting the TPEROOT variable to refer to the installation directory
- Renaming the Dharma Server for the supplied sample implementation

- Setting up ODBC access to the sample Dharma SDK Server:
  - Configuring and starting the *dhdaemon* Dharma Server process on the server system
  - Loading data definitions into the data dictionary on the server system
  - Installing and configuring the ODBC Driver on client systems (required on all systems that will access the server)

The steps to complete these tasks are similar for both UNIX and Windows platforms. Icons in the left margin indicate where there are differences.

Indicates steps specific to UNIX.

Indicates steps specific to Windows.

2.4.1 Installing Development Components

For both UNIX and Windows, the Dharma SDK is distributed on CD-ROM.

For UNIX, the CD-ROM is formatted in ISO 9660 format. To install the development components on UNIX, follow these steps:

1. Log in as *root*.
2. Create an account with the user name *dharma* and log in as *dharma*.
3. Mount the CD-ROM, specifying an appropriate mountpoint in the *mount* command (for instance, */cdrom*). Here are *mount* commands for mounting the CD-ROM on various UNIX platforms:
   - Sun Solaris: Automatically mounted
   - IBM AIX: `mount -v'cdrfs' -r'' /dev/cd0 /cdrom`
   - SCO OpenServer: `mount -f ISO9660 -r /dev/cd0 /cdrom`
   - HP-UX:
     ```
     ioscan -fnC disk  # returns CD-ROM device name
              mount -o cdcase -r /dev/dsk/c0t2d0 /cdrom
     ```
   - Linux: `mount /mnt/cdrom4`
4. Extract the contents of the distribution media with a *tar* command. The name of the tar file to extract depends on your operating system and license. See Figure 2-1 for the correct file name.

Here is a typical command that extracts files from the CD-ROM. Substitute the appropriate mountpoint and tar file name for your environment:

```
$ cd /vol6/sdkdir
$ tar -xvf /cdrom/DHPROSVR90.SOL
```

The tar command creates the directory structure and files shown in Figure 2-3. Table 2-4 gives brief descriptions of the files.

Windows

1. Run the *setup.exe* file on the CD-ROM.
2. Answer the queries from setup, including where to create the directory for the development components. Examples in this section use the *%TPEROOT%* directory.

The installation creates a parallel directory structure and file names to those on UNIX. See Figure 2-3 and Table 2-4.

Figure 2-3: Dharma SDK Client/Server Directories and Files for C stubs.

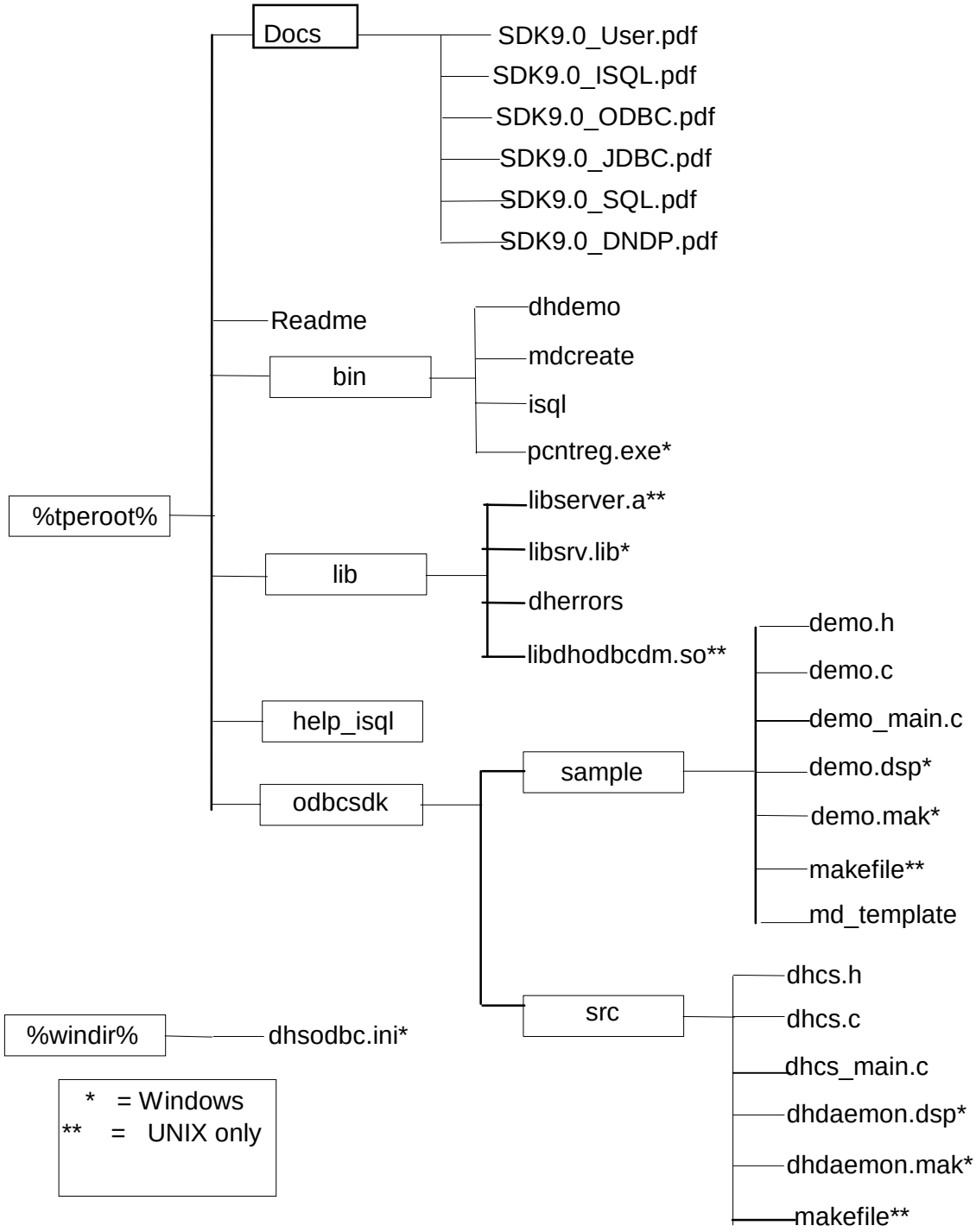


Table 2-4: Summary of Dharma SDK Client/Server Development Components for C stubs.

File	Description
Docs/osdkdgv80.pdf	Dharma SDK Guide in .PDF format.
Docs/osdkrmv80.pdf	Dharma SDK ODBC and SQL Reference in .PDF format.
Readme	Online version of installation instructions, including any additional notes not included in the printed documentation
bin/dhdemo	ODBC sample executable, pre-built from files in the *sample* directory (copy the file to *dhdaemon* to use the sample implementation)
bin/mdcreate	Utility to create a data dictionary
bin/isql	Utility for loading metadata and executing simple SQL queries on it
bin/pcntreg.exe	Utility to add and delete entries for the Dharma SDK in the Windows 2000 registry
lib/libserver.a lib\libsrv.lib	Dharma SQL engine library (links with implemented storage interfaces)
lib/dherrors	Dharma error mapping file
help_isql	Dharma ISQL online help
odbcsdk/sample/demo.h	Header file for sample implementation
odbcsdk/sample/demo_main.c	Source file for sample main() function implementation
odbcsdk/sample/demo.c	Source file for sample implementation
odbcsdk/sample/demo.dsp	Visual C++ project file for building the ODBC driver sample implementation on Windows 2000
odbcsdk/sample/demo.mak	Makefile for building the ODBC driver from the sample implementation on Windows 2000
odbcsdk/sample/makefile	Makefile for building the Dharma SDK Server from the sample implementation on UNIX
odbcsdk/sample/md_template	Template script for loading metadata
odbcsdk/src/dhcs.h	Header file for stubs
odbcsdk/src/dhcs_main.c	Source file for main() function in stubs
odbcsdk/src/dhcs.c	Source file for stubs
odbcsdk/src/dhdaemon.dsp	Visual C++ project file for building the ODBC driver for the proprietary storage system on Windows 2000
odbcsdk/src/dhdaemon.mak	Makefile for building the ODBC driver for the proprietary storage system on Windows 2000

Table 2-4: Summary of Dharma SDK Client/Server Development Components for C stubs.

File	Description
odbcsdk/src/makefile	Makefile for building the Dharma SDK Server for the proprietary storage system on UNIX
%windir%\dhsodbc.ini	Initialization file containing environment variables (placed in the directory specified by the windir environment variable)

Figure 2-4: **Dharma SDK Client/Server Directories and Files for Java stubs.**

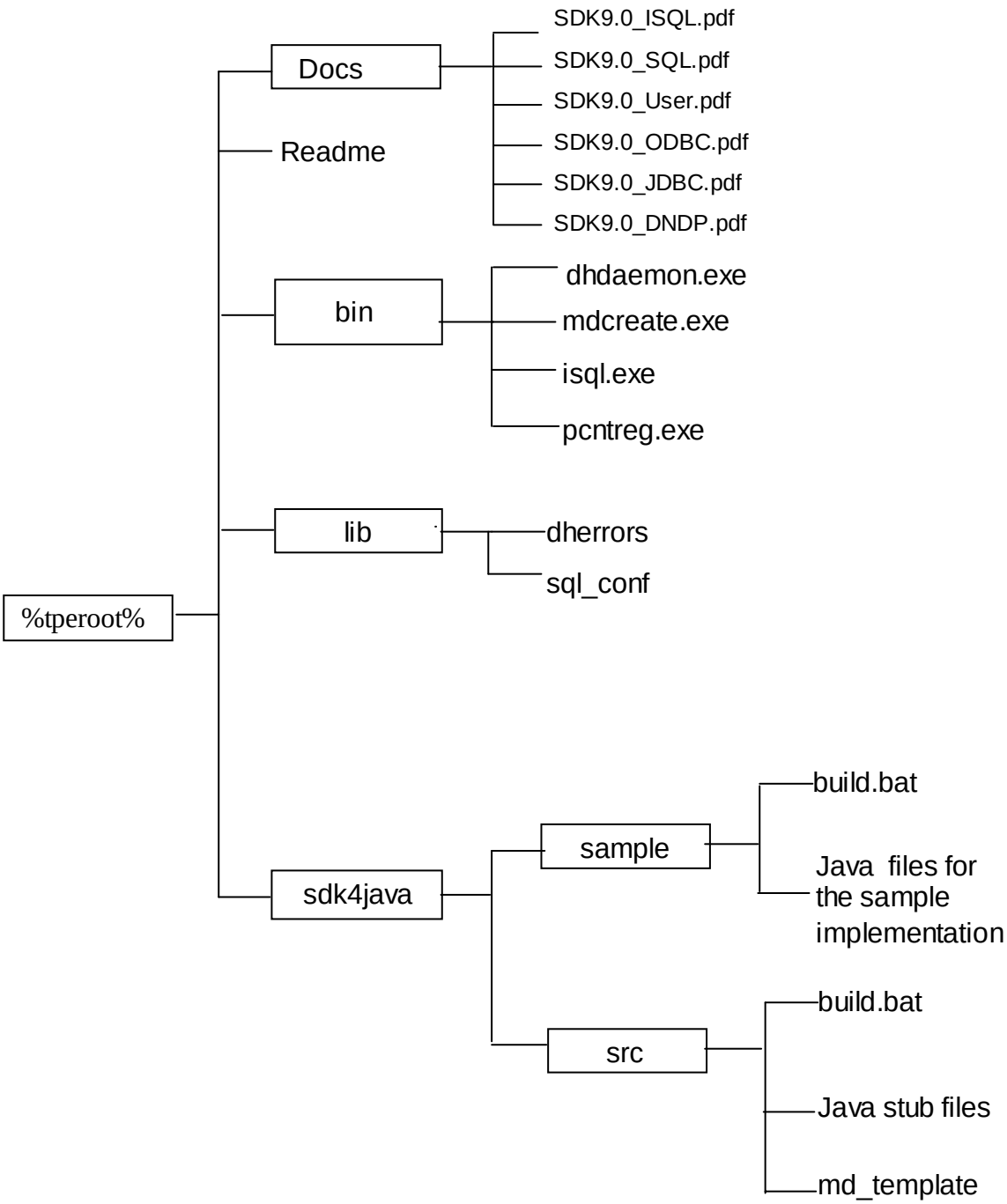


Table 2-5: Summary of Dharma SDK Client/Server Development Components for Java stubs.

File	Description
Docs/SDK9.0_User.pdf	Dharma SDK User Guide in .PDF format.
Docs/SDK9.0_SQL.pdf	Dharma SDK SQL Reference in .PDF format.
Docs/SDK9.0_ODBC.pdf	Dharma SDK ODBC Driver Guide in .PDF format.
Docs/SDK9.0_JDBC.pdf	Dharma SDK JDBC Driver Guide in .PDF format.
Docs/SDK9.0_DNDP.pdf	Dharma SDK .NET Data Provider Guide in .PDF format.
Docs/SDK9.0_ISQL.pdf	Dharma SDK ISQL Reference in .PDF format
Readme	Online version of installation instructions, including any additional notes not included in the printed documentation
bin\dhdaemon.exe	Dharma server
bin\mdcreate.exe	Utility to create a data dictionary
bin\isql.exe	Utility for loading metadata and executing simple SQL queries
lib\dherrors	Dharma error mapping file
lib\sql_conf	Configuration file used by the isql utility.
sdk4java\src\build.bat	Batch file that is used to build the classes
sdk4java\src*java	Java files that customers are expected to fill in.
sdk4java\src\md_template	Template script for loading metadata
sdk4java\sample\build.bat	Batch file that is used to build the classes.
sdk4java\sample*java	Source files for sample implementation.

2.4.2 Setting the TPEROOT Variable on the Server System

The TPEROOT environment variable specifies the main directory created during the Dharma SDK installation. TPEROOT must be set before you can run the Dharma SDK Server.

On UNIX, you must set TPEROOT interactively or in a script:

```
setenv TPEROOT /vol6/sdkdir
```

On Windows, when you install the Dharma SDK development components, the installation creates an initialization file, *%windir%\dhsodbc.ini*, that sets TPEROOT to the directory you specified during the installation. You should not have to change the file.

2.4.3 Renaming the Client/Server Sample Implementation

2.4.3.1 SDK for C stubs

The installation procedure creates an already-built version of the sample Dharma SDK Server implementation in the *bin/dhdemo* executable. You need to copy or rename the file to use the sample implementation.

On UNIX, copy or rename the *dhdemo* executable file to *dhdaemon*.

Note that if you rename the file, it will be overwritten when you build the Dharma SDK Server from your implementation (see section 3.4.2.2). If that happens, and you want to use the sample implementation, execute the makefile *odbcsdk/sample/makefile* to rebuild the *bin/dhdemo* executable file. Refer to Table 2-1 to make sure you have the compiler to build the sample implementation.

Example 2-1: Rebuilding the Dharma SDK Server for the Sample Implementation

```
$ cd $TPEROOT/odbcsdk/sample
$ make
```

On Windows, copy or rename the *dhdemo.exe* executable file to *dhdaemon.exe*.

Note that if you rename the file, it will be overwritten when you build the Dharma SDK Server from your implementation (see section 3.4.2.2). If that happens, and you want to use the sample implementation, execute the makefile *odbcsdk\sample\demo.mak* to rebuild the *bin\dhdemo.exe* executable file. Open and build the *demo.mak* file in Microsoft Visual C++ to create the Dharma SDK Server for the sample implementation.

2.4.3.2 SDK for Java stubs

SDK for Java provides classes for a sample java storage system. These class files are present in %TPEROOT%\classes directory. Building the classes after the stub implementation will overwrite these classes. Hence, you may want to backup the existing-files in classes directory to use the sampple implementation later. To rebuild the sample java storage system classes, execute build.bat file provided in the %TPEROOT%\sdk4java\sample directory.

2.4.4 Starting the *dhdaemon* Dharma SDK Server Process

The following sections describe the steps you need to complete to start the Dharma SDK Server:

- Edit the network services file to associate the *sqlnw* service name with a port number
- Start the *dhdaemon* Dharma SDK Server process

For clients to access the proprietary storage system through the Dharma SDK Server, they need to install the Dharma SDK ODBC Driver and add a data source that corresponds to the Dharma SDK Server. Section 2.3.4 describes those steps.

2.4.4.1 Edit the *Services* File to Add the *sqlnw* Service Name

The network services file (typically, the file */etc/services*) must associate a service name for the Dharma network with a port number. Log in as *root* to modify the services file.

Use the service name *sqlnw* for the Dharma network. Edit the network services file and add an entry similar to the one shown in the following example. Choose port numbers that will not conflict with other network applications.

Example 2-2: Server-Side Services File Entry for *sqlnw*

```
sqlnw           1990/tcp
```

Applications that connect to databases over the network must specify the same port number for the service name used in starting the *dhdaemon* process. Since the Dharma SDK ODBC Driver expects a port number of 1990 by default, use that number to avoid having to modify the *services* file on clients.

Windows

The details of adding the service name are the same for Windows. However, the path for the *services* file is likely to be different. A typical path for the services file on Windows is:

```
%windir%\system32\drivers\etc\SERVICES
```

2.4.4.2 UNIX Server Systems: Start the *Dhdaemon* Process

Unix

The Dharma SDK Server process *dhdaemon* must be running for ODBC clients to access the proprietary storage system. Issue the *dhdaemon start* command as shown in the following example to start the process:

Example 2-3: Starting the *dhdaemon* Process for the Sample Implementation

```
$ dhdaemon start


                Dharma/dhdaemon Version 09.00.0000
        Dharma Systems Inc              (C) 1988-2005.
         Dharma Systems Pvt Ltd            (C) 1988-2005.
Daemon started: PID 25457
```

The *dhdaemon* command has other options for additional flexibility. See the *dhdaemon* reference section in Appendix A for more details.

2.4.4.3 Windows Server Systems: Start the *Dhdaemon* Service

Windows

In Windows, the *dhdaemon* executable runs as a listener process. The installation automatically registers the *dhdaemon* image as the *Dhdaemon 9.00.00* service in the Windows registry. (You can use the *pcntreg* utility to remove or re-register the *dhdaemon* executable as a service. See the *pcntreg* reference section in Appendix A for more details.)

Follow these steps to start the *Dhdaemon* service:

1. Invoke the Windows Control Panel and select *Services*. In the list that appears, select the entry for *Dhdaemon*.

2. Click the Start button.

Note: The *dhdaemon* command has the options that can be entered in the Startup Parameteres: edit box. See the *dhdaemon* section in Appendix A for more details.

2.4.5 Loading Metadata

Metadata defines SQL tables and indexes that map the structure of data in a proprietary storage system to standard relational forms. The Dharma SDK includes utilities to create a data dictionary for your proprietary storage system (the *mdcreate* utility) and load metadata into it (the *isql* utility).

2.4.5.1 Creating the Data Dictionary with *mdcreate*

The executable *$TPEROOT/bin/mdcreate* is a utility to create a data dictionary that accepts metadata.

Log in as *dharma* before creating the data dictionary. Invoke the *mdcreate* utility and supply a name that will be used for the data dictionary and for access to the proprietary storage system. Example 2–4 shows invoking *mdcreate* to create a database called *demo_db* for use with the sample implementation:

Example 2-4: Using *mdcreate* to Create the *demo_db* Sample Database

```
$ $TPEROOT/bin/mdcreate demo_db
          Dharma/mdcreate Version 09.00.0000
        Dharma Systems Inc              (C) 1988-2005.
        Dharma Systems Pvt Ltd          (C) 1988-2005.
$
```

The *mdcreate* utility creates a subdirectory called *dbname.dbs* under the *$TPEROOT* directory and populates the directory with the necessary files. For instance, the previous example creates the directory *$TPEROOT/demo_db.dbs*.

Windows

On Windows, the executable for *mdcreate* is in the bin subdirectory under the directory specified during installation. The *mdcreate* utility creates a subdirectory called *dbname.dbs* under the installation directory and populates the directory with the necessary files.

2.4.5.2 Loading Metadata With *isql*

The executable *$TPEROOT/bin/isql* is a tool for loading metadata. It accepts a script with special SQL CREATE TABLE and CREATE INDEX statements that insert metadata for existing tables.

The sample implementation includes a script that loads the metadata for several tables in the sample. (As part of the implementation process, you create such a script for existing tables in your proprietary storage system. See section 3.3.1.1.)

To load the metadata for the sample, invoke *isql* to execute the script file *$TPEROOT/odbcsdk/sample/md_template*. Invoke *isql* on the server after the *dhdaemon* service is started. Log in as *dharma* before invoking *isql*.

The following example shows how to invoke *md_template* to create metadata for a database called *demo_db*.

Example 2-5: Using *isql* to Load Metadata

For SDK for C

```
$ isql -s $TPEROOT/odbcsdk/sample/md_template demo_db
```

For SDK for Java

```
$ isql -s $TPEROOT/sdk4java/sample/md_template demo_db

              Dharma/isql Version 09.00.0000c
        Dharma Systems Inc              (C) 1988-2005.
        Dharma Systems Pvt Ltd        (C) 1988-2005.
/vol6/sdkdir/bin/dhdaemon.exe <SQL SERVER 24211> -d demo_db -h
23907608 sqlnw
--
--
--    Template file for loading metadata for tables that already
--    exist in the underlying storage system.
--
--
CREATE TABLE  test1(
    int_col INTEGER,
    char_col CHAR(32),
    date_col DATE
    )
    STORAGE_ATTRIBUTES 'METADATA_ONLY'
.
.
.
```

The *isql* command has other options for additional flexibility. See the *isql* reference section in Appendix A for a more detailed description of the *isql* command.

Windows

On Windows, the executable for *isql* is in the bin subdirectory under the directory specified during installation.

2.5 DHARMA SDK ODBC DRIVER

2.5.1 Introduction

This chapter contains the following information:

- Summarizes support for ODBC.
- Describes building the Dharma SDK ODBC Driver.

- Shows how to edit network configuration files and add ODBC data sources
- DescribeshowtheODBCdriverobtainstheusernameandpasswordtopassto the Dharma SDK data source for validation

2.5.2 Installing and Configuring the Dharma SDK ODBC Driver

Client systems that access the Dharma SDK Server must first install the Dharma SDK ODBC Driver and configure their systems. This section describes:

- Installing the Dharma SDK ODBC Driver
- Editing network configuration files
- Adding an ODBC data source so applications can connect to the Dharma SDK Server

The Dharma SDK ODBC Driver runs on Microsoft Windows.

2.5.2.1 Installing the Dharma SDK ODBC Driver

Windows

To install the Dharma SDK ODBC Driver, follow these steps:

1. Run the *setup* file in the ODBC_driver directory of the CD-ROM
2. Answer the queries from setup.

Unix

To install the ODBC driver on UNIX client machine, follow these steps:

1. Log in as *roo*t.
2. Create an account with the user name *dharma* and log in as *dharm*a.
3. Mount the CD-ROM, specifying an appropriate mount point in the *mount* command(for instance, */cdro*m).
4. Extract the ODBC driver with a *tar* command. The name ofthe tar file to extract depends on your operating system and license. See Figure 2-1 for the correct file name.

The Dharma SDK ODBC Driver is distributed as a shared object dhodbcdm.so or a static library libclient.a, based on the platform. The dhodbcdm.so is to be linked with the driver manager while libclient.a is to be linked directly with the ODBC application.

2.5.2.2 Editing Network Configuration Files

Once the Dharma SDK ODBC Driver is installed on a client system, you need to supply information about what systems the driver will connect to. To do this, you may need to edit two network configuration files, the *services* and *hosts* files.

- This step is only necessary if the server-side services file specified a port number other than 1990. Edit the services file (typically called *services.txt* and located in

the main directory for your TCP package). Add an entry identical to that added in the server-side services file, as shown in the following example.

Example 2-6: Client-Side Services File Entry for *sqlnw*

```
sqlnw           1990/tcp
```

Be sure to use the same port number as that specified in the server-side *services* file (see section 2.4.4.1).

- Edit the *hosts* file (typically called h*osts.txt* and located in the main directory for your TCP package). Add the addresses and names of any hosts you wish to access with ODBC.

2.5.2.3 Adding the ODBC Data Sources for the Dharma SDK Server

Use the ODBC Administrator utility to add the names of any Dharma SDK Server data sources the Dharma SDK ODBC Driver will connect to:

1. Invoke the Microsoft ODBC Administrator from Windows (by default, from the Control Panel program group). The Administrator's Data Sources dialog box appears.
2. Click on the System DSN tab. A list of existing system data sources appears.
3. Click on the Add… button. The Create New Data Source dialog box appears.
4. In the Installed ODBC Drivers list box, double-click on the Dharma SDK driver. The Dharma ODBC Setup dialog box appears.
5. Enter information in the following text boxes:-
   - **Data Source Name:** — A local name for the Dharma SDK Server data source for use in ODBC connect calls and by the ODBC Administrator.
   - **Description:** — An optional descriptive string.
   - **Host:** — The name of the system where the Dharma SDK Server data source resides·
   - **Database:** — The database for the process to connect to on the host system Use the same name you specified when you invoked the *mdcreate* utility to create the data dictionary (see section 2.4.5.1).
   - **User ID:** — The user name for the process·
   - **Password:** — The password for the process·
   - **Service:** — The service name used by the server. Leave this field blank unless the *dhdaemon* server process was started using the command line and the command specified the *-s* argument (see Appendix A). (If it did, specify the same value here as that used in the *-s* argument.)

You must supply the name of the data source. If you omit the host name, database name, user name, or password, the driver prompts the ODBC application user for that information when it connects to the data source.

The ODBC Administrator utility updates the ODBC Driver manager registry entry with the information supplied in the dialog box.

Unix

The Dharma SDK ODBC library allows you to link and run an ODBC application on UNIX. Use the UnixODBC Driver manager on Solaris and Linux which is available as a free download from http://www.unixodbc.org

To add data sources, edit an initialization file. The ODBC Driver Manager installation creates an initialization file called *odbc.ini* that resides in the top-level directory. You can use any text editor to edit this file. However, you can also use any initialization file as defined by the ODBCINI environment variable.

1. Add the following entry in the [ODBC Data Sources] section.

   ```
   Demo_db=Dharma Dharma  SDK ODBC Driver
   ```

2. Create a new section named for the DSN name you create, and enter information in the following text boxes:

- **Driver:** Absolute path to the Dharma SDK ODBC Driver, The Dharma SDK ODBC Driver resides in the *\lib* directory of the installation root.
- **Description:** An optional descriptive string
- **Host:** The name of the system where the Dharma SDK Server data source resides
- **Database:** The database for the process to connect to on the host system. Use the same name you specified when you invoked the *mdcreate* utility to create the data dictionary (see section 2.3.3).
- **User ID:** The user name for the process.
- **Password:** The password for the process.
- **Service:** The service name used by the server. Leave this field blank unless the *dhdaemon* server process was started using the command line and the command specified the *-s* argument (see Appendix A). If the *-s* argument was used, specify the same value here as that used in the *-s* argument.)

The following example shows how an initialization file might look after creating data sources for the *demo_db* database associated with the Dharma SDK ODBC Driver:

Host=isis

Database= demo_db

User ID=dharma

Password=dummy

Service=sqlnw

[ODBC]

Trace=0

TraceFile=/space/dhsdk9/odbc_trace.out

TraceDll=/opt/odbc/lib/odbctrac.so

InstallDir=/opt/odbc

2.5.2.4 Handling of User Name and Password

When an ODBC application issues an SQLConnnect call, the Dharma SDK ODBC driver passes any user name or password information to the host system. On the host system, Dharma SDK passes the user name and password to the underlying storage system for validation.

The ODBC driver gets the user name and password information as follows:

1. By reading the input arguments passed from the application's call to the SQLConnect or SQLDriverConnect functions
2. If no values are supplied to the function calls, the driver uses the values stored in the registry for the data source

The driver passes any user name and password obtained from these sources to the server. If none of these sources provide the user name and password, the driver passes empty quoted strings. If the underlying storage system returns an authentication error, the driver displays a dialog box for the user of the client application to supply information.

2.6 DHARMA SDK JDBC DRIVER

2.6.1 Installing the JDBC Driver

To install the Dharma SDK JDBC Driver, follow these steps:

1. Run the *setup* file in the **DHJDBC90** directory of the CD-ROM
2. Answer the queries from setup.

The JDBC driver is supplied as a jar file, *DharmaDriver.jar*, that allows you to link and run a JDBC application on UNIX. Extract the JDBC driver with a tar command from the tar file in **DHJDBC90 from the CD-ROM**.

See the *Dharma SDK JDBC Driver Guide* for details on setting up the driver for access by client applications.DharmaDriver.jar file should be set in the CLASSPATH to run the JDBC application.

To test if the driver installed successfully, invoke it through a sample Java program. You must supply a JDBC URL as part of the java invocation command line. The URL for the Dharma JDBC Driver is of the form:

jdbc:dharma:T:host_name:db_name:port_number

See the *JDBC Driver Guide* for more detail on Java URLs and connecting to databases. Supply the name of a host system and database specific to your environment. For example:

```
java dharma.jdbc.DharmaTest jdbc:dharma:T:labnt:testdb:4006
```

The *port_number* component of the URL is optional if the services file on the server uses the default port number of 1990.

2.7 DHARMA SDK .NET DATA PROVIDER

2.7.1 Introduction

This section describes installing the Dharma SDK .NET Data Provider on Microsoft Windows. Refer to the *Dharma SDK .NET Data Provider Guide* for more information on using the .NET Data Provider.

2.7.2 Required Software

The prerequisites for using the Dharma SDK .NET Data Provider DLL are listed below.

1. Microsoft VC++ .NET version 7.0.9466.
2. Microsoft .NET Framework SDK version 1.1

2.7.3 Installing from a Distribution Kit on any Client System

To install the Dharma SDK .NET Data Provider, follow these steps:

1. Run the setup program present in the **DHDOTNET90** directory of the distribution kit.
2. Answer the queries from setup.

Chapter 3

Implementation Strategy

3.1 INTRODUCTION

This chapter describes the overall strategy you use to implement the Dharma SDK storage interfaces. Because of the large variations in how different proprietary storage systems store and access data, this section does not focus on any particular implementation.

The Dharma SDK includes a complete sample implementation (*$TPEROOT/odbcsdk/sample* for the C SDK, *$TPEROOT/sdk4java/sample* for the Java SDK). You can adapt the sample files as appropriate for implementing the interfaces.

This chapter describes:

Some approaches for mapping proprietary data and proprietary access methods to relational data and relational indexes

- A suggested series of incremental development stages to implement the storage interfaces
- For each of the stages, details on the interfaces you need to implement
- Building an ODBC executable after each development stage
- Setting runtime variables that specify attributes of Dharma SDK behavior

3.2 PHILOSOPHY

Note If data in your proprietary storage system is already in relational format, you can skip this section.

Proprietary storage systems typically store data in a form that does not match standard relational format. Relational format requires that data be laid out in tabular format, with a clean separation between data and indexes. Proprietary storage systems, on the other hand, probably use a more complex format to store data. Also, proprietary storage systems often intersperse data and access methods.

Deciding on a philosophy for mapping proprietary data and access methods to SQL tables and indexes is an important part of the implementation of the Dharma SDK. This section discusses different approaches you can consider. Before you implement the storage interfaces, settle on a particular approach and use it as a guide during development.

The rest of this section discusses how you can map two common proprietary formats of data to the relational model.

3.2.1 Two Common Proprietary Formats

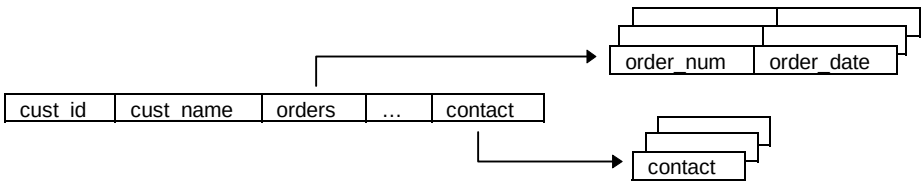
One common format for proprietary data is records with repeating fields. In such a record, there can be one or more occurrences of values in the repeating fields for each value of a non-repeating field. The following example shows the structure of a record with repeating fields. In it, the c*ust_id* and *cust_name* fields are not repeating. The *order_num* and *order_date* fields form a compound repeating field, where there is a corresponding *order_date* value for each value in *order_num*. The compound repeating field holds information about all of a customer's orders. In addition, there is a separate repeating field, *contacts*, that contains the names of multiple contacts at the customer.

Example 3-1: Proprietary Data: Records With Repeating Fields

cust_id	cust_name	order_num	order_date			…	contact	
1	a	o1	d1			…	c1	c2
2	b	o3	d3	o4	d4	…	c3	

Another common format for proprietary data is records that include navigational elements as part of records. In such a hierarchical record, part of the data contained in the record is a pointer to multiple occurrences of one or more fields. The following example shows a hierarchical record representing the repeating fields shown in Example 3-1.

Example 3-2: Proprietary Data: Hierarchical Records



3.2.2 Mapping Proprietary Data to a Relational View

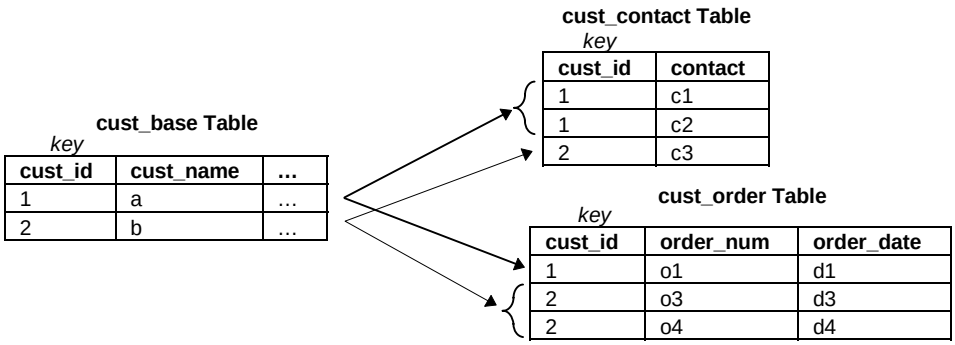
One way to map proprietary data with repeating fields or built-in navigational elements to standard relational tables is to split data into separate tables.

In this approach, split the data so that the repeating fields (or pointers in a hierarchical record) appear to reside in their own, separate tables. There is a "parent" table that contains only non-repeating data. There is also a "child" table for each repeating field or pointer in a hierarchical record.

Note Splitting data in this manner does not mean that the actual data in the proprietary storage system is restructured. It provides a logical relational view of the data so client applications can issue standard relational queries against it.

The parent table contains a key field (or set of fields) that uniquely identifies a row in the table. The child table includes this key field to identify a set of child records that corresponds to the parent record. The following example shows how the relational tables might appear using this approach.

Example 3-3: Splitting Proprietary Data Records Into Separate Tables



Once you identify key columns, you need to create indexes for them.

3.2.3 Mapping Proprietary Access Methods to Relational Indexes

Typically, a proprietary storage system has built-in mechanisms to quickly access the key fields of a record. You can easily map these access methods to relational indexes on the parent tables.

In addition, you will likely need to define additional indexes on child tables. These indexes may be "virtual" indexes that do not physically exist or indexes that correspond to existing navigational elements.

For instance, the repeating-field example in Example 3-1 was split into a parent table and two child tables, as shown in Example 3-3. If there is an existing mechanism to access the *cust_id* field in the proprietary data, you can create virtual indexes on the *cust_id* column in the child tables. This is because the child rows are physically part of the parent row and accessing the parent row through its index effectively accesses the child rows as well.

Even though the data is presented relationally as residing in separate tables, they are part of the same record in the proprietary data format. This means relational joins between the parent and child table can use virtual indexes on the child tables to give "pre-joined" performance. Your implementation of the index storage interfaces must recognize that the virtual indexes indicate that access to a child table is available through the existing access mechanism on the parent table.

The hierarchical-record example in Example 3-2 also split into the parent and child records shown in Example 3-3. In this case, too, you can create indexes on the *cust_id* column in the child tables, since there are existing navigational elements (the pointers) to the child tables from the parent table. The navigational elements, in combination with the physical index, effectively provide indexed access to the child tables.

3.2.4 Developing an Algorithm for Accessing Data

Whatever relational view you ultimately choose to represent data in your proprietary storage system, you need to implement storage interfaces that take standard relational constructs and use them to retrieve the correct data.

The implementation must translate references to simple relational tables to the corresponding fields in the proprietary storage system data. In addition, the stub imple-

mentation must have a mechanism for detecting which of those fields are repeating (or pointers in a hierarchical record).

3.3 STAGES OF IMPLEMENTATION

You can simplify implementation of the Dharma SDK storage interfaces by dividing development into stages. Each implementation stage provides an increasing level of access or functionality to the proprietary storage system.

By building the dhdaemon executable for your implementation at each stage, you can verify incremental completion of the functionality for that stage.Compile the java files at each stage to verify the functionality of of your Java stub implementation.

The following table describes the implementation stages and lists which storage interfaces you need to implement for each stage. For read access, only stages 1 through 3 are required.

Table 3-1: Implementation Stages for Developing the Storage Stubs

Stage 1	**Metadata Access**	
Stage 1 maps data in the proprietary storage system to standard relational tables and loads the resulting table definitions into the system catalog. Stage 1 also verifies your software build environment by linking your storage system code with the storage interfaces and Dharma SDK library to create an dhdaemon executable for the first time for C stubs. Java files are compiled and your software build environment is verified for Java stubs. After stage 1 implementation, you can issue queries on system tables to retrieve data on tables in the proprietary storage system.		
Storage Interfaces to Implement for Stage 1		
C stubs	**Java Stubs**	**Description**
dhcs_rss_init	StorageEnvironment.createStorageEnvironment, StorageEnvironment.createStorageManagerHandle	Initializes a connection to the proprietary storage system and performs any required user authentication.
dhcs_add_table	StorageManagerHandle.createTable	For stage 1 implementation, generates a table identifier that corresponds to an existing table name. Additional implementation required for stage 5 support.
dhcs_rss_cleanup	StorageEnvironment.close, StorageManagerHandle.close	Closes the proprietary storage system and performs any required cleanup.
Stage 2	**Read Access**	
Stage 2 provides read access to data in the proprietary storage system but does not take advantage of indexes or other performance-enhancing access methods that may be available in the proprietary storage system.		
Storage Interfaces to Implement for Stage 2		
C stubs	**Java Stubs**	**Description**
dhcs_alloc_tid	RecordID constructor	Allocates memory to store a tuple identifier and initializes the tuple identifier.
dhcs_free_tid		Frees memory from a tuple identifier.

Table 3-1: Implementation Stages for Developing the Storage Stubs

dhcs_assign_tid	RecordID.setRecordID(RecordID)	Copies the value for a tuple identifier.
dhcs_compare_tid	RecordID.compareRecordID	Compares two tuple identifiers and returns a value indicating equality or relative size.
dhcs_char_to_tid	RecordID.setRecordID(String)	Converts a character string to a tuple identifier.
dhcs_tid_to_char	RecordID.getRecordID	Converts a tuple identifier to a character string.
dhcs_tpl_scan_open	StorageManagerHandle.getTableScanHandle	Opens a table for scanning when no indexes are available.
dhcs_tpl_scan_fetch	TableScanHandle.getNextRecord	Fetches the next record from a table.
dhcs_tpl_scan_close	TableScanHandle.close	Closes a table that was opened for scanning.
dhcs_get_error_msg	DharmaStorageException	Returns the error message for any error code generated by the storage manager. As provided, the interface generates the Not yet implemented message whenever it is called. Subsequent stages require continued implementation as the storage manager generates additional error codes.
dhcs_tpl_open	StorageManagerHandle.getTableHandle	Opens a table by allocating memory for a table handle.
dhcs_tpl_close	TableHandle.close	Closes a table by deallocating the table handle.
Stage 3	**Indexed Access**	
Stage 3 implements complete read access to data in the proprietary storage system. This stage requires mapping existing indexes and proprietary access methods to standard relational indexes For environments that do not require write access or access to long data types, stage 3 is the final implementation stage.		
Storage Interfaces to Implement for Stage 3		
C stubs	**Java Stubs**	**Description**
dhcs_rss_get_info	StorageManagerHandle.getStorageManagerInfo	In stage 3, returns details on how a storage manager supports indexed access. In stage 4, indicates how the storage manager processes updates to indexes.
dhcs_create_index	StorageManagerHandle.createIndex	For stage 3 implementation, only generates an index identifier that corresponds to an existing index name. Additional implementation required for stage 5 support.
dhcs_ix_scan_open	StorageManagerHandle.getIndexScanHandle	Opens an index for scanning.
dhcs_ix_scan_fetch	IndexScanHandle.getNextRecord	Fetches the next record in an index scan.

Table 3-1: Implementation Stages for Developing the Storage Stubs

dhcs_ix_scan_close	IndexScanHandle.close	Closes an index which was opened for scanning.
dhcs_tpl_fetch	TableHandle.getRecord	Fetches a specific record from a table.
dhcs_get_error_msg	DharmaStorageException	Continued implementation: Returns the error message for error codes generated by the storage manager.
Stage 4	**Write Access**	
Stage 4 provides the ability to insert, update, and delete data in the proprietary storage system. Stage 4 implementation is optional.		
Storage Interfaces to Implement for Stage 4		
C stubs	**Java Stubs**	**Description**
dhcs_rss_get_info	StorageManagerHandle.getStorageManagerInfo	(Initial implementation in stage 3.) In stage 4, indicates how the storage manager processes updates to indexes.
dhcs_tpl_insert	TableHandle.insert	Inserts a record into a table.
dhcs_tpl_delete	TableHandle.delete	Deletes a record from a table.
dhcs_tpl_update	TableHandle.update	Updates values in an existing table record.
dhcs_ix_open	StorageManagerHandle.getIndexHandle	Opens an index for updating.
dhcs_ix_close	IndexHandle.close	Closes an index after updating.
dhcs_ix_insert	IndexHandle.insert	Inserts a record into an index.
dhcs_ix_delete	IndexHandle.delete	Deletes a record from an index.
dhcs_begin_trans	StorageEnvironment.beginTransaction	Starts a transaction.
dhcs_commit_trans	StorageEnvironment.commitTransaction	Commits a transaction.
dhcs_abort_trans	StorageEnvironment.rollbackTransaction	Aborts, or rolls back, a transaction.
dhcs_get_error_msg	DharmaStorageException	Continued implementation: Returns the error message for error codes generated by the storage manager.
Stage 5	**Data Definition**	
Stage 5 implements the ability to create new tables and indexes in the proprietary storage system. Stage 5 implementation is optional.		
Storage Interfaces to Implement for Stage 5		
C stubs	**Java Stubs**	**Description**
dhcs_add_table	StorageManagerHandle.createTable	(Initial implementation in stage 1.) For stage 5 implementation, creates a new table in the proprietary storage system.

Table 3-1: Implementation Stages for Developing the Storage Stubs

dhcs_drop_table	StorageManagerHandle.dropTable	Deletes a table from the proprietary storage system.
dhcs_create_index	StorageManagerHandle.creatIndex	(Initial implementation in stage 3.) For stage 5 implementation, creates a new index in the proprietary storage system.
dhcs_drop_index	StorageManagerHandle.dropIndex	Deletes an index from the proprietary storage system.
dhcs_get_error_msg	DharmaStorageException	Continued implementation: Returns the error message for error codes generated by the storage manager.
Stage 6		**Long Data Type Support**
Stage 6 provides access to unstructured character and binary data in columns defined as LONG VARCHAR or LONG VARBINARY. The characteristics of such long data-type data (or simply "long data") are completely dependent on the implementation. Stage 6 implementation can be limited to long data access, or include the ability to insert and delete long data in the proprietary storage system. Stage 6 implementation is optional.		
Storage Interfaces to Implement for Stage 6		
C stubs	**Java Stubs**	**Description**
dhcs_get_data	not currently implemented	Retrieves a segment of a long field value.
dhcs_put_data	not currently implemented	Stores a segment of a long field value.
dhcs_put_hdl	not currently implemented	Copies data from one long-field handle to another.
dhcs_get_error_msg	not currently implemented	Continued implementation: Returns the error message for error codes generated by the storage manager.
Stage 7		**Dynamic Metadata Support**
Stage 7 allows implementations to dynamically provide details about tables and indexes that reside in the proprietary storage system. Stage 7 implementation is optional.		
Storage Interfaces to Implement for Stage 7		
C stubs	**Java Stubs**	**Description**
dhcs_get_metainfo	not currently implemented	Returns the number of tables that the user can access, and whether subsequent calls to *dhcs_get_tblinfo* will return detail on those tables sorted by table name.
dhcs_get_tblinfo	not currently implemented	Returns detail on a table.
dhcs_get_idxinfo	not currently implemented	Returns detail on an index.
dhcs_get_colinfo	not currently implemented	Returns detail on columns in a table.
dhcs_get_error_msg	not currently implemented	Continued implementation: Returns the error message for error codes generated by the storage manager.

3.3.1 Stage 1: Metadata Access

Stage 1 loads and accesses metadata.

As mentioned in Chapter 2, metadata are definitions for SQL tables and indexes that map the structure of data in the proprietary storage system to standard relational forms.

The first step in stage 1 is to write an SQL script that expresses the mapping of proprietary data to relational tables. The *isql* utility uses this script to actually create the tables and indexes that correspond to the structure of data in the proprietary storage system. As you implement the storage interfaces, the script provides a detailed description of the relational tables that client applications will access.

Stage 1 implements the following functionality:

- Connection to the proprietary storage system and optional security authentication
- Generation of table identifiers that correspond to names of existing tables in the proprietary storage system to support metadata loading
- Disconnection from the proprietary storage system

3.3.1.1 Creating *md_script*, the SQL Script to Load Metadata

The script you write to load metadata serves two purposes:

- You use it during development to create the metadata against which you implement storage interfaces
- It becomes part of the release kit installed on other systems with the proprietary storage system

The Dharma SDK development components include a sample script you can adapt to create your own script that loads metadata for the proprietary storage system. The template is in the file *md_template ($TPEROOT/odbcsdk/sample/md_template, $TPEROOT/sdk4java/sample/md_template).*

Create your script in a file called *md_script*. (The file name is important only because the instructions for creating a release kit in Chapter 4 specify that file name.)

The script contains CREATE TABLE and INDEX statements with the STORAGE_ATTRIBUTES 'METADATA_ONLY' clause. This clause directs the SQL engine to insert metadata into the data dictionary without requiring the proprietary storage system to create an empty table or index. The table or index name used in the CREATE statement must be the same as an existing table or index in the proprietary storage system.

The following example shows an excerpt from the sample script that illustrates the CREATE TABLE and CREATE INDEX syntax.

Example 3-4: Script Template for Loading Metadata

```
$ more odbcsdk/sample/md_template
--
--
```

```
--    Template file for loading metadata for tables that already
--    exist in the underlying storage system.
--
--
CREATE TABLE  test1(
    col1 integer,
    col2 char(32),
    col3 date
    )
    STORAGE_ATTRIBUTES 'METADATA_ONLY';

CREATE INDEX test1_idx ON test1(col1)
    STORAGE_ATTRIBUTES 'METADATA_ONLY';
```

3.3.1.2 Initializing Connections to the Proprietary Storage System

When the SQL engine starts, it calls an initialization interface (*dhcs_rss_init, StorageEnvironment.createStorageEnvironment)* . This interface is called only when the SQL engine starts, and it is the only function called when it starts. The specific processing done by the routine depends on the requirements of the proprietary storage system. In general, the routine must:

- Initialize a connection to the proprietary storage system. This may include opening files, loading data structures, or other steps specific to the particular proprietary storage system.
- Perform any desired authentication based on the user name and password arguments the SQL engine passes to it. The SQL engine does no authentication of its own.

3.3.1.3 Partially Implementing *dhcs_add_table* to Return Table Identifiers

As discussed previously, you need to choose an approach for representing the data in the proprietary storage system as a series of relational tables. Once you do that, you use the *isql* utility to load metadata in the system tables.

The *isql* utility processes the SQL script file. To support processing this script through *isql*, you partially implement the interface to add tables (*dhcs_add_table, StorageManagerHandle.createTable)*. (If you choose to implement stage 5, you then provide complete support for table creation.)

When the SQL engine calls the stubs (*dhcs_add_table, StorageManagerHandle.createTable)* to load metadata, it sets the metadata-only flag to TRUE and provides the name of an existing table in the proprietary storage system. The implementation must generate a unique table identifier that the SQL engine associates with the table name. The value of the identifier must be from 1000 to 32767.

The implementation must also keep track of table identifiers and their corresponding table names. The SQL engine passes only the identifier, not the name, in subsequent

calls. It is the implementation's responsibility to associate the identifier with the correct table.

See section 5.2.1 for details on the *dhcs_add_table* interface. See “createTable” on page 46.for details on the *StorageManagerHandle.createTable* method.

3.3.1.4 Closing Connections With *dhcs_rss_cleanup*

The SQL engine calls the *dhcs_rss_cleanup*, StorageEnvironment.close interface when an application issues the *SQLDisconnect* call. The specific steps associated with this call depend on your proprietary storage system, but could include deallocating memory and closing files.

3.3.1.5 Testing Stage 1 Implementation

To test stage 1 implementation, you need to first do the following:

- Link your storage system code with the storage interfaces and Dharma SDK library to create an dhdaemon executable for the first time.
- Execute the SQL script using *isql* (see Appendix A).

Once you do that, you can issue queries against the system tables (see Appendix B for details of the system tables). Example 3-2 shows invoking *isql* to do a simple query on one of the system tables to confirm completion of stage 1 implementation. The query accesses the *systables* system table to retrieve some details on any tables that are not system tables (WHERE TBLTYPE <> 'S'). The results show that metadata on a single table, *test1*, was loaded into the Dharma SDK.

Example 3-5: Stage 1 Completion: Confirming Access to Metadata

```
$ isql proprietary_db
              Dharma/isql Version 09.00.0000
           Dharma Systems Inc              (C) 1988-2005.
           Dharma Systems Pvt Ltd          (C) 1988-2005.


/dharma/bin/dhdaemon.exe <SQL SERVER 28999> -d proprietary_db -
h 415136 sqlnw_ks
> SELECT TBL, ID, CREATOR, OWNER, TBLTYPE
FROM DHARMA.SYSTABLES WHERE TBLTYPE <> 'S';
tbl     id      creator owner   tbltype
---     --      ------- -----   -------

test1,  1000,   dharma, dharma, T

Tuple selected = 1

>
```

3.3.2 Stage 2: Read Access

Stage 2 provides simple read access to data in the proprietary storage system. Stage 2 implements the following additional functionality:

- Generation and manipulation of tuple identifiers that point to specific rows of data in the proprietary storage system
- Sequential scan of rows in tables in the proprietary storage system
- Generation of implementation-specific error messages
- Opening and closing tables

The following sections provide some more detail on how you approach implementing these routines. Figure 3-1and Figure 3-2 show how the SQL engine makes a series of calls to these routines for two types of SQL SELECT statements:

- A simple statement that refers to one table
- A more complex statement that joins data from two tables

Figure 3-1: Calls to Retrieve Data in Stage 2: Simple SELECT

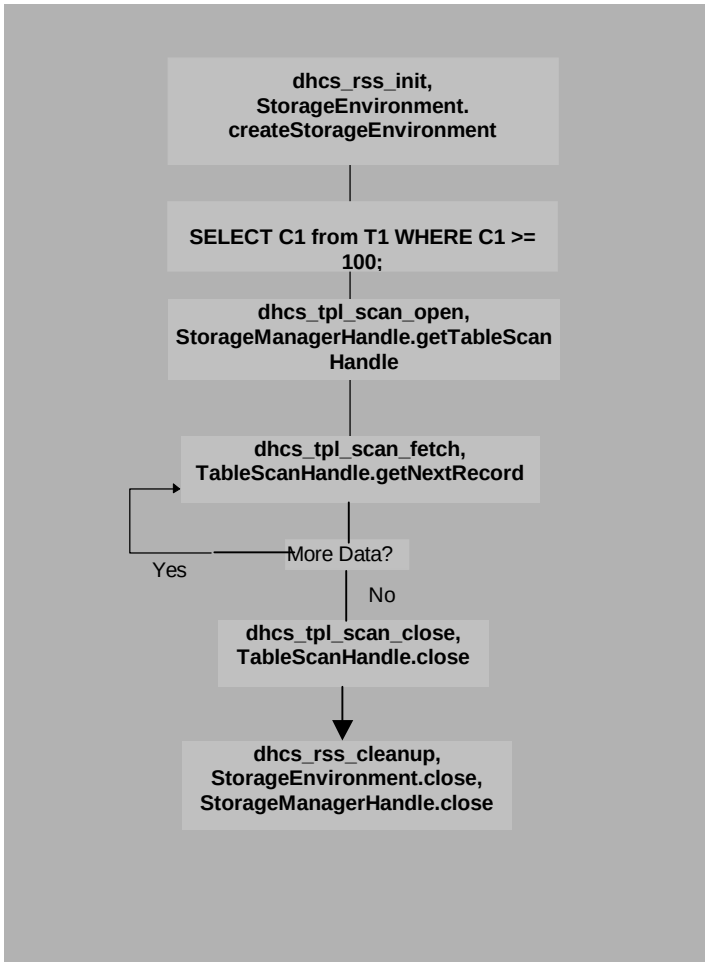
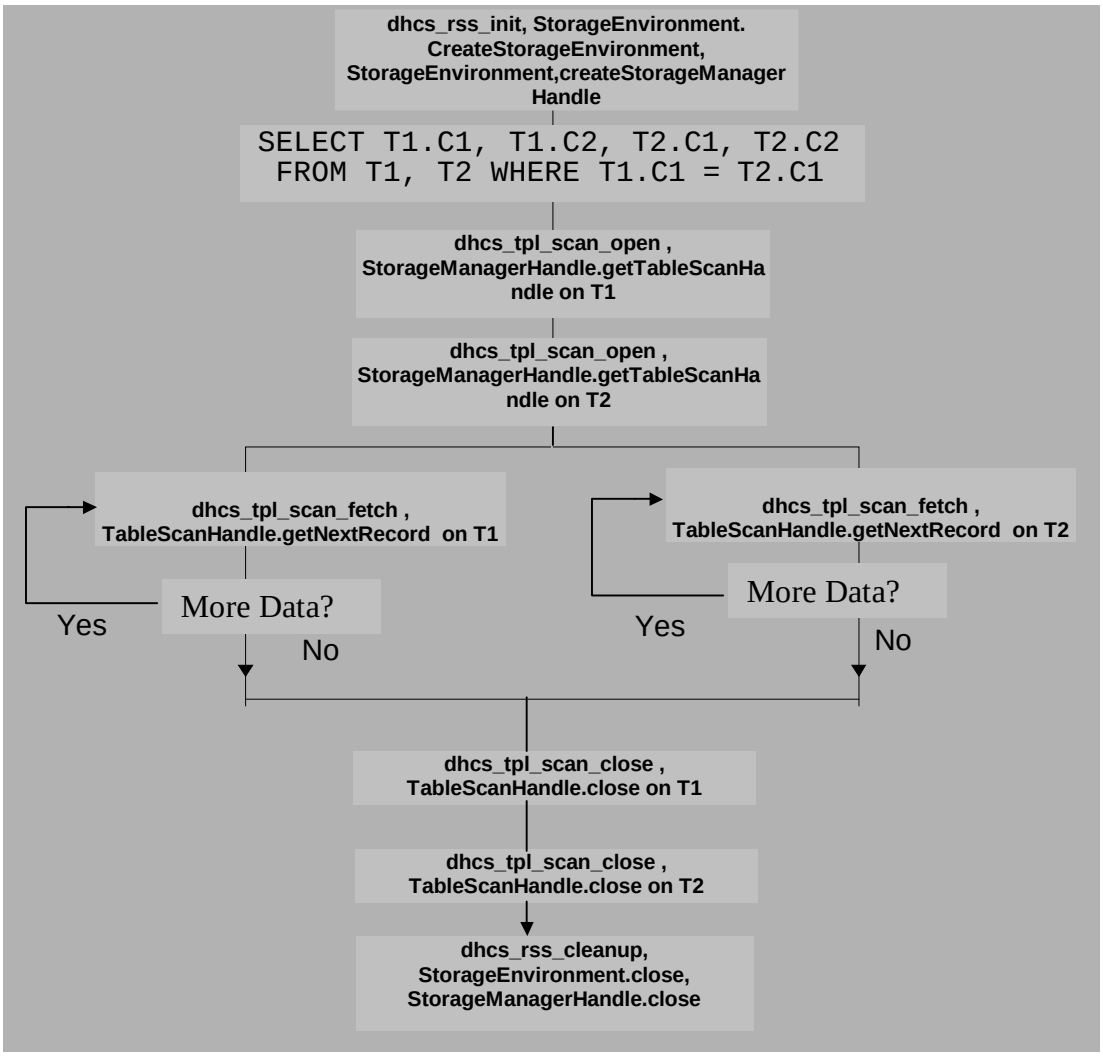


Figure 3-2: Calls to Retrieve Data in Stage 2: Two-Table Join



3.3.2.1 Implementing the Tuple Identifier Interfaces

A tuple identifier, or *tid* (**note**: In the Java implementation, tuple identifiers are implemented through the *RecordID* class*)*, uniquely identifies a record in the proprietary storage system. The implementation must return a tuple identifier when it retrieves data from the proprietary storage system.

Chapter 5 describes the tuple identifier interfaces (Chapter 6 describes the RecordID class). These interfaces are utility functions that the implementation itself as well as the SQL engine call routinely.

The format of a tuple identifier is specific to the proprietary storage system. In general, the implementation must provide the following functionality through the interfaces:

- Allocate and free memory to store *tid* values
- Copy and compare *tid* values

- Convert *tid* values to and from character strings (the maximum allowable length of the character-string version of *tids* is 255)

The sample illustrates implementation of these interfaces with a *tid* of type char.

3.3.2.2 Retrieving Data Through Table Scans

To retrieve records when no indexes are available, the SQL engine opens a table scan, fetches records from it and then closes the scan. It retrieves each record and compares it with any conditions specified in the SQL statement until the proprietary storage system indicates no more data is available.

When it opens a scan (*dhcs_tpl_scan_open, StorageManagerHandle.getTableScanHandle),* the SQL engine passes the table identifier for the table of interest. The implementation needs to open files or load the required data structures that correspond to the table. The routine must return a handle for the table scan and point to the first record in the table (in the java implementation, the class object returned is the handle).

The SQL engine uses this scan handle (or class) on each call to fetch a record (*dhcs_tpl_scan_fetch, TableScanHandle.getNextRecord)* (see Chapters 5 & 6). In addition, it may pass a pointer to a list of field values, a pointer to a tuple identifier, or both. The elements of the list indicate which table columns are of interest.

To fetch a record, implementations should:

- If there are no more records in the table, return SQL_NOT_FOUND.
- Check whether the list of field values is empty (a null pointer, or an empty list). If the list is not empty, supply the field values for the table columns specified in the list. Before writing data to the passed field-values structures, the implementation must convert values to host format. The header file *$TPEROOT/odbcsdk/src/dhcs.h* specifies the host format for all the SQL data types.
- Check whether the pointer to the *tid* is null. If it is not, supply the *tid* for the current record.
- Advance the scan to the next record.

The SQL engine makes an explicit call to close the scan (*dhcs_tpl_scan_close, TableScanHandle.close)*. In a C implementation it passes the scan handle to indicate which scan the implementation should close. The implementation frees the scan handle and performs any other operations appropriate to the proprietary storage system. In a Java implementation, the class member function is called and appropriate action is taken.

3.3.2.3 Returning Implementation-Specific Error Messages

The storage interface routine (*dhcs_get_error_mesg, DharmaStorageExceptiongetErrorMessage)* provides a mechanism for implementations to generate error messages specific to the proprietary storage system.

The SQL engine calls error message handler when it receives an error code generated by the storage manager. The storage manager can return such error codes during execution of any routine, through *dhcs_status_t* or *DharmaStorageException*. See Chapters 5 & 6 for details.

3.3.2.4 Opening and Closing Tables

The SQL engine uses table handles for internal operations. To complete stage 2, implement the storage interfaces to open and close table handles (*dhcs_tpl_open* and *dhcs_tpl_close, StorageManagerHandle.getTableHandle, TableHandle.close)* that allocate and free table handles:

- When the SQL engine opens a table handle it passes the table identifier generated during table creation (*dhcs_tpl_add_table, StorageManagerHandle.createTable)*. The routine must identify and initialize the corresponding table in the proprietary storage system and return a handle for the table.
- The SQL engine makes an explicit call to closes a table handle (*dhcs_tpl_close, TableHandle.close)*. In the C implementation the SQL engine passes the table handle generated by the corresponding call to *dhcs_tpl_open*. The close routine frees the handle.

3.3.2.5 Testing Stage 2 Implementation

To test stage 2 implementation, you can use the *isql* utility. Issue queries such as the following on tables in the proprietary database:

SELECT * FROM T1;	This statement initiates a table scan on T1.
SELECT COUNT(*) FROM T1;	This statement checks implementation of the tuple identifier interfaces.

3.3.3 Stage 3: Indexed Access

Stage 3 implements indexed access to data in the proprietary storage system.

This stage requires mapping existing indexes and proprietary access methods to standard relational indexes.

The SQL engine asks implementations provide details on the properties of their indexes (*dhcs_rss_get_info, StorageManagerHandle.getStorageManagerInfo)* . Among other characteristics, responses to these queries indicate which comparison operations (such as equal or greater than) the proprietary storage system can support through indexed retrieval.

In addition to providing for responses to requests about index properties, stage 3 implements the following additional functionality:

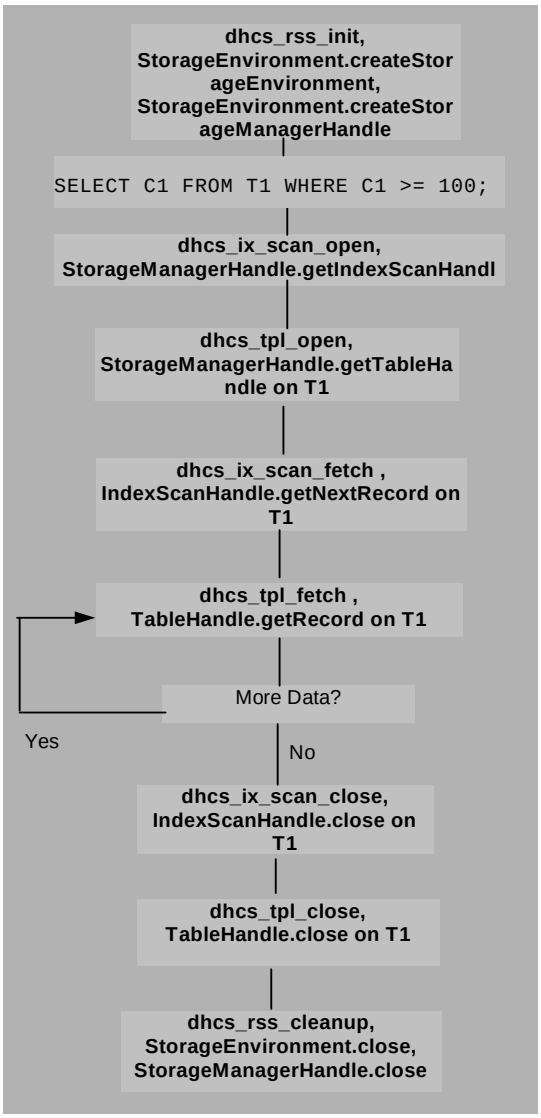
- Generation of index identifiers that correspond to names of existing indexes in the proprietary storage system (*dhcs_create_index, StorageManagerHandle.createIndex*)
- Use of index scans instead of table scans (*dhcs_ix_scan_open*, *dhcs_ix_scan_fetch*, and *dhcs_ix_scan_close*, *StorageManagerHandle.getIndexScanHandle, IndexScanHandl.getNextRecord, IndexScanHandle.close*). Index scans retrieve the tuple identifiers for rows that satisfy query criteria. If an index for a table exists, a SELECT statement that specifies any of the supported comparison operators results in calls to index scan interfaces instead of the table scan interfaces.

- Support for non-scan table retrieval (*dhcs_tpl_open*, *dhcs_tpl_fetch*, and *dhcs_tpl_close*, *StorageMangerHandle.getTableHandle*, *TableHandle.getRecord*, *TableHandle.close*). Instead of time-consuming table scans, the SQL engine passes the tuple identifiers obtained from the index scan to these interfaces, which directly retrieve the rows.

The following sections provide some more detail on how you approach implementing these routines. If you do not require write access to your proprietary storage system, stage 3 is the final implementation stage.

The following figure shows how the SQL engine makes a series of calls after indexed access has been implemented.

Figure 3-3: Calls to Retrieve Data in Stage 3



3.3.3.1 Responding to Index Property Information Calls

The SQL engine makes calls (*dhcs_rss_get_info, StorageManagerHandle.getStorageManagerInfo)* to obtain details on the properties of an implementation's index support. It makes these calls repeatedly with different info type arguments. Most info types require a Boolean response (true or false); when requesting supported comparison operators (IX_PUSH_DOWN_RESTRICTS) a list is returned.

The following table summarizes the info types. See the *Info Type Values* discussion in section 5.8.3 for details. **Note:** For the Java implementation, the info types do not have the DHCS_ prefix.

Table 3-2: Info Type Properties Describing Index Support

Info Type Argument	Meaning
DHCS_IX_ALL_COMPONENTS	Specific to multi-component indexes: When performing an index scan, whether search values must be provided for all components.
DHCS_IX_COMPUTE_AGGR	Whether the storage manager supports the SQL MIN and MAX aggregate functions.
DHCS_IX_FETCH_ALL_FIELDS	Whether the storage system is able to return all of the fields of the record, and not just the index component fields, in response to an index scan (*dhcs_ix_scan_fetch, IndexScanHandle.getNextRecord)*. The SQL engine takes advantage of this property to avoid calls to the table interface to retrieve records (*tpl_fetch, TableHandle.getRecord)*. See the discussion in Chapters 5 & 6 for these interfaces.
DHCS_IX_PUSH_DOWN_RESTRICTS	Comparison operators which the storage manager can process through index scans (as opposed to being processed internally by the SQL engine). Refer to Table 5-2: on page 5-37 (C stubs) or Table 6-2 on page 53(Java stubs).
DHCS_IX_SCAN_ALLOWED	Whether indexes of the specified type support index scans.
DHCS_IX_SORT_ORDER	Whether a scan on the index returns records in the order of the index key.
DHCS_IX_TID_SORTED	Whether the index returns records sorted by tuple identifier.
DHCS_IX_UPD_REQUIRED	(Not applicable to stage 3.) Whether the SQL engine must update indexes through separate calls to the index interfaces (*dhcs_ix_insert* or *dhcs_ix_delete, IndexHandle.insert, IndexHandle.delete*) after an insert, update, or delete operation on a table.

3.3.3.2 Partially Implementing index creation to Return Index Identifiers

The process for implementing an abbreviated version of the index creation routine (*dhcs_create_index, StorageManagerHandle.createIndex*) is parallel to implementing support for loading table metadata (*dhcs_add_table, StorageManagerHandle.createTable*), as described in Chapter 5:

- Change the *md_script* to add CREATE INDEX statements that will load metadata for SQL indexes that correspond to the structure of indexes or other navigational elements in the proprietary storage system (see Chapter 5).
- Partially implement the index creation routine (*dhcs_create_index, StorageManagerHandle.createIndex)* to return index identifiers. The routine needs to generate an index identifier with a value from 1000 to 32767. As with table identifiers, the implementation must also keep track of index identifiers and their corresponding index names.

See Chapter 5 for details on the *dhcs_create_index* interface. See Chapter 6 for details on the *StorageManagerHandle.createIndex* interface.

3.3.3.3 Retrieving Data Through Index Scans

When it processes a SELECT statement that specifies any of the supported comparison operators, the SQL engine checks the *sysindexes* catalog table to see if there are indexes for the table. If there is an index, the engine calls the index interface to open an index scan (*dhcs_ix_scan_open*, *StorageManagerHandle.getIndexScanHandle*).

Implementation of index scan handle opening (*dhcs_ix_scan_open*, *StorageManagerHandle.getIndexScanHandle*) is parallel to opening table scans (*dhcs_tpl_scan_open*, *StorageManagerHandle.getTableScanHandle*) in the following ways:

- The SQL engine passes the index identifier for the index of interest (as well as the table identifier).
- The implementation needs to open files or load the required data structures that correspond to the index. (**Note:** The implementation should make use of any available storage systems caching mechanisms.)
- The implementation generates a handle for the index scan.

However, for index scan opens, the SQL engine also passes an operator argument and any comparison values that the implementation must process. This operator/comparison-value combination corresponds to an SQL predicate and indicates a condition that is true or false about a row or group of rows (such as WHERE C1 >= 100).

The SQL engine queries storage systems to see which comparison operators they support for a given index type (*dhcs_rss_get_info* with the info type argument DHCS_IX_PUSH_DOWN_RESTRICTS (Refer to Table 5-2: on page 5-37), *StorageManagerHandle.getStorageManagerInfo* with the *infoType* argument IX_PUSH_DOWN_RESTRICTS(Refer to Table 6-2: on page 6-53)). Implementations must at least support a forward index scan (DHCS_IXOP_FIRST, IXOP_FIRST). After opening an index scan, the implementation must position the scan to the first record in the index that satisfies the operator argument based on the supplied comparison values. (**Note:** This can be done in the scan opening routine or in the fetch routine prior to the first fetch. The details are up to the implementation.)

If the query requires field values that are not available through the index, the SQL engine opens the table (*dhcs_tpl_open*, *StorageManagerHandle.getTableHandle*). When it opens the table, the SQL engine passes the table identifier generated when the table was created (*dhcs_tpl_add_table*, *StorageManagerHandle.createTable*). The routine must identify and initialize the corresponding table in the proprietary storage system and return a handle for the table. The SQL engine passes the table handle to calls during subsequent fetch operations for the C stubs or uses the class object to call the appropriate methods for the Java stubs.

When it calls index fetch (*dhcs_ix_scan_fetch*, *IndexScanHandle.getNextRecord*), the SQL engine supplies the same operator and comparison values as it did to its call to open the scan (*dhcs_ix_scan_open*, *StorageManagerHandle.getIndexScanHandle)*. The implementation supplies the values for the requested fields, supplies the tuple identifier for the record, and advances the scan to the next index record that satisfies

the predicate criteria. If no more records meet the criteria, the implementation returns SQL_NOT_FOUND.

Before calling index fetch again, the SQL engine may fetch the record from the table (*dhcs_tpl_fetch, TableHandle.getRecord*) if it needs column values not available through the index. These routines implement a non-scan operation, meaning they operate on a single row of a table that is specified through a tuple identifier obtained through an index scan and passed to the routine by the SQL engine. The implementation retrieves the appropriate row from the table, based on the supplied tuple identifier, and fills in the requested values.

When the query returns no more data, the SQL engine explicitly closes open handles (*dhcs_ix_scan_close* and *dhcs_tpl_close, IndexScanHandle.close and TableHandle.close*) .

3.3.3.4 Testing Stage 3 Implementation

To test stage 3 implementation, you can use the *isql* utility. Issue queries such as the following on tables in the proprietary database:

SELECT * FROM T1 WHERE C1 = 'whatever';	Presuming there is an index on the *c1* column and that the proprietary storage system supports the = comparison operator, this statement initiates an index scan on the index and a non-scan fetch on *T1*.
SELECT T1.C1, T1.C2, T2.C1, T2.C2 FROM T1, T2 WHERE T1.C1 = T2.C1;	Presuming there are indexes on the *t1.c1* and *t2.c1* columns, and that the proprietary storage system supports the = comparison operator, this statement initiates scans on virtual indexes.

3.3.4 Stage 4: Write Access

Stage 4 provides the ability to insert, update, and delete data in the proprietary storage system. Stage 4 implementation is optional, and adds the following functionality:

- The ability to add new records (*dhcs_tpl_insert* and *dhcs_ix_insert, TableHandle.insert and IndexHandle.insert*), delete records (*dhcs_tpl_delete* and *dhcs_ix_delete, TableHandle.delete and IndexHandle.delete*), and modify existing records (*dhcs_tpl_update*, *dhcs_ix_insert*, and *dhcs_ix_delete, TableHandle.update, IndexHandle.insert and IndexHandle.delete*)
- Transaction management (*dhcs_begin_trans*, *dhcs_commit_trans*, and *dhcs_abort_trans, StorageEnvironment.beginTransaction, StorageEnvironment.commitTransaction* and *StorageEnvironment.rollbackTransaction*)

The specific routines the SQL engine calls during update operations depends on whether the proprietary storage system requires the SQL engine to manage updates to indexes when corresponding table rows are updated. The SQL engine queries the implementation to determine its requirements (*dhcs_rss_get_info* with the DHCS_IX_UPD_REQUIRED info type argument, *StorageManagerHandle.getStorageManagerInfo* with the IX_UPD_REQUIRED *infoType* argument).

If implementations return TRUE, the SQL engine explicitly calls separate routines (*dhcs_ix_open, dhcs_ix_delete*, *dhcs_ix_insert*, and *dhcs_ix_close, StorageManager-*

Handle.getIndexHandle, *IndexHandle.delete*, *IndexHandle.insert*, *IndexHandle.close*) to update the appropriate index row when a table row is updated. If implementations return FALSE, the SQL engine never calls those routines and presumes the proprietary storage system updates indexes.

The next two figures show the series of calls the SQL engine makes to process two typical update situations:

- A simple INSERT statement that specifies values directly
- An INSERT statement that retrieves values from another table (where both tables have indexes)

Figure 3-4: Calls to Store Data in Stage 4: Simple INSERT Statement

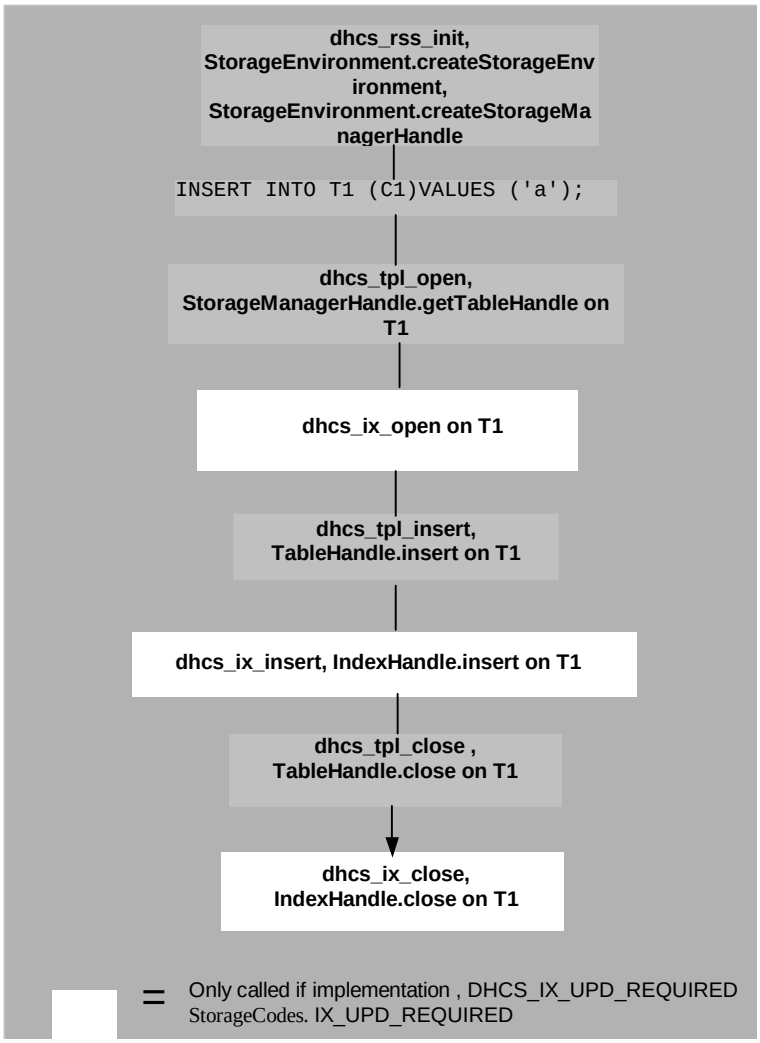
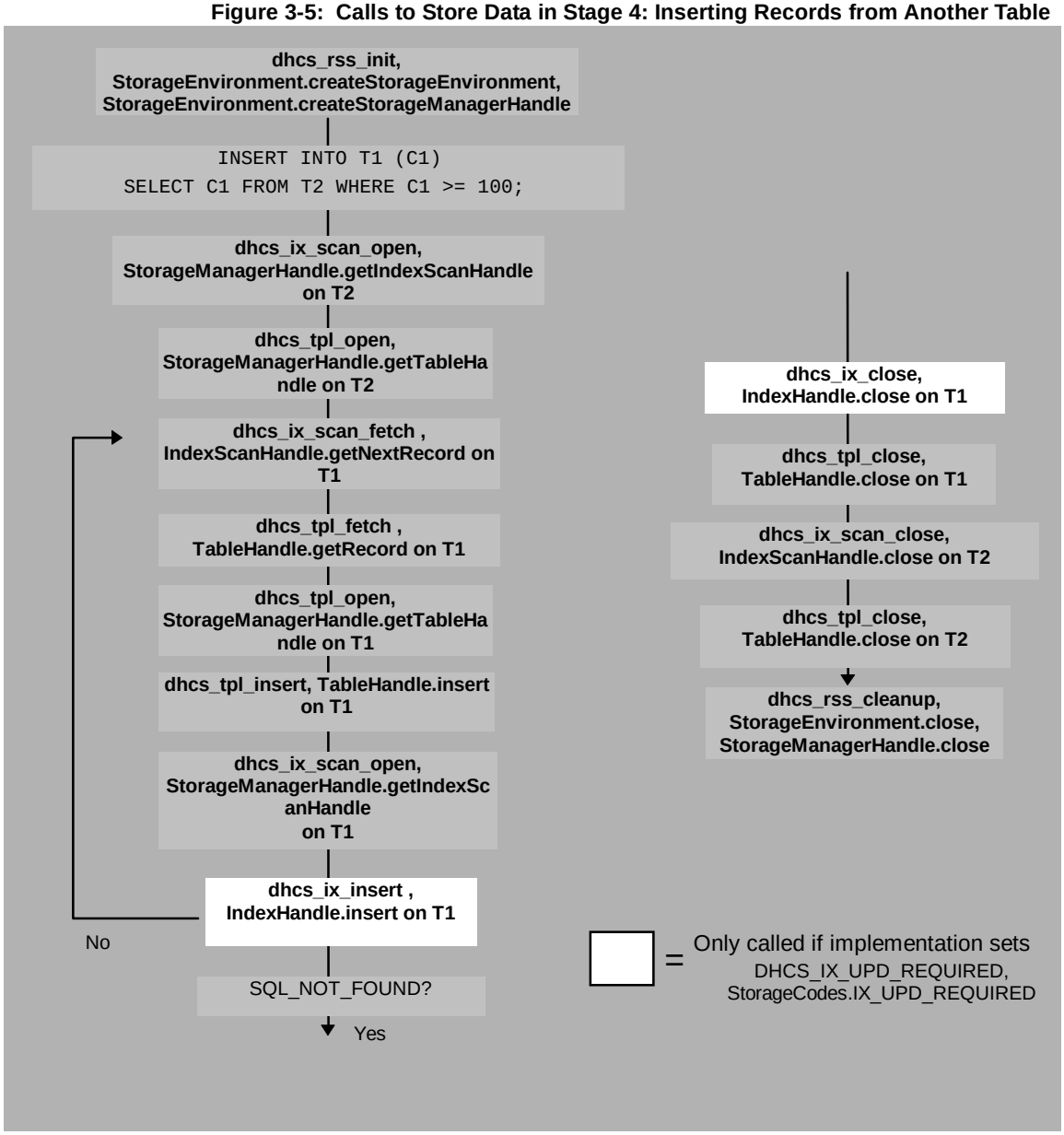


Figure 3-5: Calls to Store Data in Stage 4: Inserting Records from Another Table



3.3.4.1 Adding, Modifying, and Deleting Records

As shown in Figure 3-4, the SQL engine first inserts a records into the table (*dhcs_tpl_insert, TableHandle.insert*). It passes a list of field values and, for the C stubs, a table handle. The implementation stores the record in the table and generates a tuple identifier.

If the implementation requires explicit index updation, the SQL engine next inserts an index record (*dhcs_ix_insert, IndexHandle.insert*) and passes the tuple identifier just generated and values for the index components. The implementation stores the values as a new index record.

The SQL engine follows a similar process for UPDATE statements to modify values in an existing record. It updates the table (*dhcs_tpl_update, TableHandle.update),*

passing the tuple identifier for the existing record, along with the values to be modified. There is no corresponding update routine for indexes, however. Instead, if the implementation requires explicit index updation, the SQL engine makes two calls to delete the old entry (*dhcs_ix_delete, IndexHandle.delete*) and create the new one (*dhcs_ix_insert, IndexHandle.insert*).

When deleting records, the SQL engine first deletes the record from the table (*dhcs_tpl_delete, TableHandle.delete*). It passes the tuple identifier for the record, and for the C stubs, the table handle. If the implementation requires explicit index updataion, the SQL engine then deletes the index entry (*dhcs_ix_delete, IndexHandle.delete*).

3.3.4.2 Managing Transactions

A transaction is a group of operations whose changes can be made permanent or undone only as a unit. Once you implement the ability to change data in the proprietary storage system, you may also need to implement transaction management to protect against data corruption.

Note Based on the requirements of your proprietary storage system, you may decide you do not need to implement transaction management. If so, you can skip this section. As supplied, the transaction storage interfaces print a warning message and return a success status code. If you do not wish to implement transaction support, simply remove the warning message.

The SQL engine does not manage transactions, but only calls the appropriate storage interface when directed to do so by an client application. For instance, the SQL engine does not guarantee that one user's changes will not conflict with another user's. It is up to the implementation to enforce whatever concurrency control and consistency level it requires, given the capabilities of the underlying proprietary storage system.

Applications issue ODBC calls that result in the SQL engine calling one of the following storage interfaces:

- *dhcs_begin_trans/StorageEnvironment.beginTransaction* signals the beginning of a series of operations that must be managed as a single transaction
- *dhcs_commit_trans/StorageEnvironment.commitTransaction* ends a transaction and specifies that results of the operations within it be made permanent
- *dhcs_abort_trans/StorageEnvironment.rollbackTransaction* ends a transaction and specifies that results of the operations within it be undone

These routines do not include an identifier to distinguish between different users' transactions. However, each connection to the database generates a new process, which implementations can map to their mechanism for managing transactions.

3.3.4.3 Testing Stage 4 Implementation

To test stage 4 implementation, you can use the *isql* utility. Issue queries such as the following on tables in the proprietary database:

INSERT INTO T1 (C1) VALUES ('whatever');	Checks insert execution. If there is an index on c1 and the implementation set DHCS_IX_UPDATE_REQUIRED, also initiates an insert operation on the index.
DELETE FROM T1 WHERE C1 = 'whatever';	Checks delete execution. If there is an index on c1 and the implementation set DHCS_IX_UPDATE_REQUIRED, also initiates a delete operation on the index.
UPDATE T1 SET C1 = 'new value';	Checks update execution. If there is an index on c1 and the implementation set DHCS_IX_UPDATE_REQUIRED, also initiates a delete and insert operation on the index.

3.3.5 Stage 5: Data Definition

Stage 5 implements the ability to create new tables and indexes, and delete existing tables and indexes in the proprietary storage system. Stage 5 implementation is optional and requires that your proprietary storage system has a mechanism for creating new database objects.

Stage 5 involves full implementation of table creation (*dhcs_add_table*, *StorageManagerHandle.createTable*) and index creation (*dhcs_create_index*, *StorageManagerHandle.createIndex*). As implemented in stages 2 and 3, those routines return a table or index identifier that corresponds to an existing object in the proprietary storage system. For stage 5, implementations must extract the metadata the SQL engine passes to the routines:

- For table creation in the C stubs (*dhcs_add_table*) , a pointer to the *dhcs_fld_list_t* data structure, which contains details of the table column definitions
- For table creation in the Java stubs (*StorageManagerHandle.createTable*), a *TableFields* array which contains details of the table column definitions
- For index creation in the C stubs (*dhcs_create_index)*, a pointer to the *dhcs_keydesc_t* data structure, which contains details of the index keys and the sort order for the index
- For index creation in the Java stubs (*StorageManagerHandle.createIndex*), an *IndexFields* array which contains details of the index keys and the sort order for the index

3.3.5.1 Testing Stage 5 Implementation

To test stage 5 implementation, you can use the *isql* utility. Issue queries such as the following on tables in the proprietary database:

CREATE TABLE NEW (C1 INT, C2 CHAR(10));	Checks table creation.

CREATE INDEX NEW_IX ON NEW (C1 ASC);	Checks index creation.

3.3.6 Stage 6: Long Data Type Support

Note: The Java stub implementation does not currently support long data types.

Stage 6 provides access to unstructured character and binary data in columns defined with the SQL LONG VARCHAR or LONG VARBINARY data type.

Data in such columns can be of any length and of any format. For instance, long data-type columns can store large amounts of text, long strings of binary data (such as executable images or input from data-collection devices), or graphics files.

Because of the arbitrary length and structure of such long data-type data (or simply "long data"), the Dharma SDK provides storage interfaces to retrieve or store it in segments. These interfaces are modeled after the Microsoft ODBC *SQLGetData* and *SQLPutData* functions. An ODBC application calls these ODBC functions, and the SQL engine in turns calls *dhcs_get_data* or *dhcs_put_data*:

- To retrieve long data, it loops through calls to *dhcs_get_data*. Each call to *dhcs_get_data* retrieves a segment of a long field value.
- To store long data, it loops through calls to *dhcs_put_data*. Each call to *dhcs_put_data* stores a segment of a long field value.
- To copy data from one long data-type column to another, it calls *dhcs_put_hdl*.

For all these interfaces, the SQL engine passes a field handle. Field handles are character strings that identify storage for data in long data-type columns. The structure and contents of field handles are up to the implementation.

Implementations create field handles when the SQL engine calls the *dhcs_tpl_insert* routine. (This is in contrast to conventional data-type columns, for which the SQL engine passes actual values to the insert routine.) Similarly, for fetch routines, implementations return field handles instead of the actual long data values. The SQL engine then passes the field handle to the appropriate long-data storage interface, looping until the fetch or insert is complete.

The following sections describe this process in more detail.

3.3.6.1 Retrieving Long Data

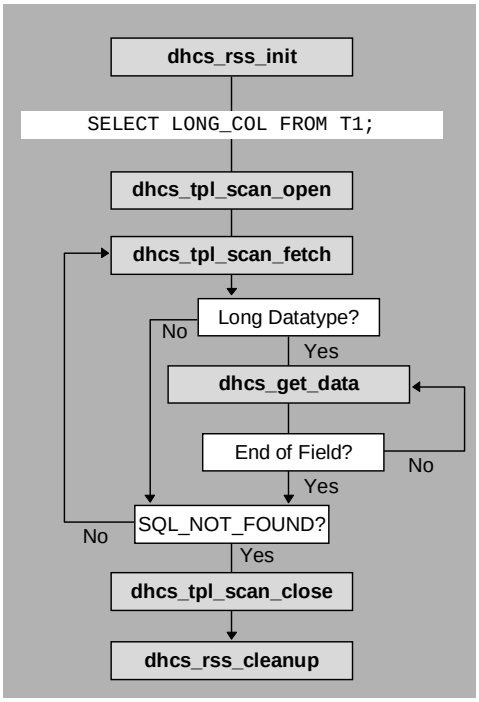
To retrieve a table row that includes long data, the SQL engine calls *dhcs_tpl_scan_fetch* or *dhcs_ix_scan_fetch*, the same as it does for conventional data-type columns.

However, instead of returning the actual data for a long column, the implementation returns a field handle that identifies storage for the data in the field. When the SQL engine calls *dhcs_get_data* it passes this field handle. The implementation retrieves a segment of the field value and stores it in a buffer. It also indicates the length of the data remaining to be retrieved.

If the ODBC application calls *SQLGetData* multiple times, the SQL engine calls *dhcs_get_data* multiple times as well. Chapter 5 describes the arguments to *dhcs_get_data* and how they interact.

The following figure shows the series of calls the SQL engine makes to retrieve long data.

Figure 3-6: Calls to Retrieve Data in Stage 6



3.3.6.2 Storing Long Data

The process for storing long data is similar to retrieving long data values:

- The SQL engine calls *dhcs_tpl_insert*.
- Instead of passing data values for long columns, the SQL engine expects *dhcs_tpl_insert* to return a field handle identifying storage to receive the long data. (The implementation should initialize this storage, since there is no guarantee that the client application will actually request the SQL engine to call *dhcs_put_data*.)
- The SQL engine passes the field handle in a call to *dhcs_put_data*, along with a segment of the data for the field. The implementation stores the segment.
- If the client application requests, the SQL engine calls *dhcs_put_data* again until the entire field value is stored.

There is no mechanism for updating long data. The SQL engine generates an error if it encounters an SQL UPDATE statement that specifies a LONG VARCHAR or LONG VARBINARY column.

The SQL engine uses a different process for INSERT … SELECT statements that copy data from one long column to another. Instead of looping through calls to

dhcs_get_data to retrieve the data, and then looping through calls to *dhcs_put_data* to store it, the SQL engine uses the *dhcs_put_hdl* routine to simply copy the field handle for the data. See Chapter 5 for details.

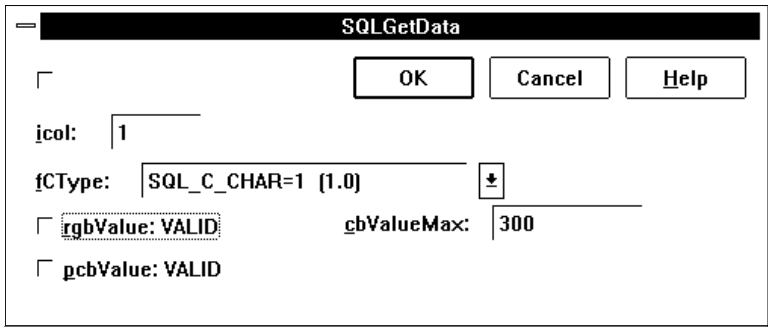
3.3.6.3 Creating Indexes on Long Data-Type Columns

Indexes on long columns are a special case. The only index operator allowed on a long column is CONTAINS. When an index is created on a long field, the SQL engine checks that CONTAINS is the only operator supported by that index type, and generates an error if that is not the case. See the CONTAINS notes on page 5-1 for more detail.

3.3.6.4 Testing Stage 6 Implementation

To test stage 6 implementation, you need an ODBC application that supports long data types. (Specifically, you need to use an ODBC tool that issues *SQLGetData* and *SQLPutData* calls.) The ODBC Test application supplied as part of Microsoft's ODBC SDK is one way to issue SQLGetData and SQLPutData calls. The following figure shows the dialog box from the ODBC Test application for issuing the *SQLGet-Data* call.

Figure 3-7: Testing Long Data Type Support



3.3.7 Stage 7: Dynamic Metadata Support

Note: The Java stub implementation does not currently support dynamic metadata.

Stage 7 allows implementations to dynamically provide details about tables and indexes that reside in the proprietary storage system. Stage 7 implementation is optional.

By default, implementations use SQL scripts and the *isql* utility to load definitions of metadata into the Dharma SDK data dictionary. The Dharma SDK then maintains the metadata internally, separate from the proprietary storage system. This default is appropriate for environments where details about application tables are known ahead of time.

However, in some environments, applications frequently add or change tables and indexes by means other than SQL and the Dharma SDK. In such an environment, the default approach of loading metadata through a static SQL script makes it difficult to keep the Dharma SDK metadata synchronized with the dynamically-created tables.

To accommodate these dynamic environments, the Dharma SDK includes routines for implementations to provide detail on user tables and indexes that reside in the proprietary storage system. If an implementation indicates support for dynamic metadata, the SQL engine calls the routines each time a user connects to the storage system. The implementation responds to the calls by returning metadata for tables that the user is allowed to access.

Note A consequence of supporting dynamic metadata is that implementations should ignore calls to data definition routines-*dhcs_add_table*, *dhcs_create_index*, *dhcs_drop_table*, and *dhcs_drop_index*-that set the *meta_data_only* argument to TRUE.

The SQL engine stores the detail provided through these routines in memory-resident versions of the *systables*, *sysindexes*, *syscolumns*, and *systabauth* data dictionary tables. Once the memory-resident tables contain the necessary metadata information, the SQL engine processes queries as usual.

For instance, if a user creates a table, the SQL engine still calls the *dhcs_add_table* routine and updates the memory-resident metadata information to reflect the new table or index. When the connection is closed, however, the data is gone and subsequent connections will show the new table only if the information about it is returned by the dynamic metadata routines called at connection time.

3.3.7.1 Indicating Support for Dynamic Metadata

Storage managers indicate support for dynamic metadata through the DH_DYNAMIC_METADATA variable. If this runtime variable is set to Y, the SQL engine relies on the storage manager to provide details on user tables and indexes.

To set DH_DYNAMIC_METADATA:

- On Windows XP and Windows 2000, change the *%windir%/dh*odbc.ini* initialization file and add the variable.
- On UNIX, set the environment variable for the user *dharma*.

In client/server configurations, set the DH_DYNAMIC_METADATA environment variable on the server system before starting the server process.

3.3.7.2 Providing Detail on User Tables and Indexes

To support dynamic metadata, storage managers implement routines that the SQL engine calls to obtain details of user tables and indexes in the storage system.

When a user first connects to the storage system, before the first transaction begins, the SQL engine calls routines to get information about tables that the user can access:

- *dhcs_get_metainfo* returns the number of tables that the user can access, and whether subsequent calls to *dhcs_get_tblinfo* will return detail on those tables sorted by table name. The SQL engine calls *dhcs_get_metainfo* once for each user connection. Implementation of *dhcs_get_metainfo* is optional. See Chapter 5 for details.
- d*hcs_get_tblinfo* returns detail on a single table. This routine passes the structure *dhcs_tblinfo_t*, which the implementation fills in with the name, owner, and iden-

tifier for the table. The SQL engine calls *dhcs_get_tblinfo* repeatedly until the implementation returns SQL_NOT_FOUND to indicate there are no more tables accessible by the user. The SQL engine uses the information supplied through calls to *dhcs_get_tblinfo* to load a memory-resident version of the *systables* system catalog table. See Chapter 5 for details.

Note that it is the responsibility of the implementation to determine which tables are accessible by the user connected to the storage system, and to return metadata for those tables only.

To improve performance, the SQL engine limits the information it retrieves when a user first connects to the storage system. It postpones retrieving detail about the columns in a table and any indexes associated with the table until an SQL statement first refers to that table. At that point, the SQL engine calls the following routines to get full detail about the columns and indexes:

- *dhcs_get_colinfo* returns detail on all the columns in the table. The routine passes arguments that identify the table of interest, and an array of structures which the implementation fills in with details of the table's columns. The SQL engine loads the column information into a memory-resident version of the *syscolumns* system catalog table.
- *dhcs_get_idxinfo* returns detail on a single index. The routine passes arguments that identify the table of interest, and the structure *dhcs_idxinfo_t,* which the implementation fills in with details on a single index. The SQL engine calls *dhcs_get_idxinfo* repeatedly until the implementation returns SQL_NOT_FOUND to indicate there are no more indexes for the table. The SQL engine loads the index information into a memory-resident version of the *sysindexes* system catalog table.

The dynamic metadata routines only support metadata for tables, columns, and indexes. It is not possible for implementations to provide metadata about other objects, such as constraints on user tables.

3.3.7.3 Testing Stage 7 Implementation

To test stage 7 implementation, you can use the *isql* utility. Issue queries on system catalog tables to confirm that the metadata was correctly returned by the dynamic metadata routines:

SELECT TBL FROM SYSTABLES WHERE TBLTYPE='T';	Checks that the *dhcs_get_tblinfo* routine supplied metadata for all user tables.
SELECT DISTINCT IDXNAME FROM SYSINDEXES I, SYSTABLES T WHERE I.TBL = T.TBL AND T.TBLTYPE='T';	Checks that the *dhcs_get_idxinfo* routine supplied metadata for all user indexes.
SELECT SC.TBL, SC.COL, FROM SYSCOLUMNS SC, SYSTABLES ST WHERE SC.TBL = ST.TBL AND ST.TBLTYPE='T';	Checks that the *dhcs_get_colinfo* routine supplied metadata for all the columns in user tables.

3.4 BUILDING AND CONFIGURING THE DHARMA SDK SERVER

As you progress in your implementation, you will routinely want to build the Dharma SDK Server executable that uses your proprietary storage system.

The steps to do this parallel those described in Chapter 2 for building the Dharma SDK Server for the sample implementation. The steps are different for the Dharma SDK Desktop and Dharma SDK Client/Server configurations.

This section describes building the Dharma SDK Server for the Desktop and Client/Server configurations of the Dharma SDK.

3.4.1 Desktop

Before you build the Dharma SDK Server, first install the development components as described in Chapter 2.

Once you do that, complete the following steps:

- Build the Desktop Dharma SDK DLL.
- Create and load the data dictionary (only required once).

3.4.1.1 Building the Desktop Dharma SDK DLL

C Stubs

Build the Dharma SDK DLL by executing the *dhdaemon.mak* makefile in *%TPEROOT%\odbcsdk\src*. You may need to modify the makefile to link with additional libraries and object modules specific to your implementation. As provided, the *dhdaemon.mak* file in *%TPEROOT%\odbcsdk\src* generates an executable named *dhstodbc.dll* in the *%TPEROOT%\bin directory.*

Open and build the *dhdaemon.mak* file in Microsoft Visual C++ to create the Desktop Dharma SDK DLL.

Java Stubs

Build the class files for the Java stubs by executing the build.bat batch file in %TPEROOT%\sdk4java\src. You may need to modify the batch file to use classes specific to your implementation. The build.bat file in %TPEROOT%\sdk4java\src generates the classes for the Java storage stubs and places in %TPEROOT%\classes\jrss\stubs directory.

3.4.1.2 Creating and Loading the Data Dictionary

The executable *%TPEROOT%\bin\mdcreate* is a utility to create a data dictionary that accepts metadata. Invoke the *mdcreate* utility and supply a name that will be used for the data dictionary and for access to the sample implementation. For example:

```
%TPEROOT%\bin\mdcreate proprietary_db
```

The *mdcreate* utility creates a subdirectory called *<dbname>.dbs* under the *%TPEROOT%* directory and populates the directory with the necessary files. For instance, the preceding example creates the directory

%TPEROOT%\proprietary_db.dbs.

The executable *%TPEROOT%\bin\isql* is a tool for loading metadata and for interactive SQL queries. It accepts a script with special SQL CREATE TABLE and CREATE INDEX statements that insert metadata for existing tables.

Note If your implementation supports dynamic metadata, you do not need to load metadata using *isql*. Instead, the SQL engine loads metadata automatically each time an application connects to the database. See section 3.3.7 for details on dynamic metadata support.

You can create a SQL script file with CREATE TABLE and INDEX statements specific to your database. To load the metadata for your proprietary storage system, invoke *isql* to execute the script file. The following example shows invoking the SQL commands in a file called *md_script* to create metadata for a database called *proprietary_db*.

```
isql -s %TPEROOT%\odbcsdk\md_script proprietary_db


          Dharma/isql Version 09.00.0000
     Dharma Systems Inc           (C) 1988-2005.
     Dharma Systems Pvt Ltd       (C) 1988-2005.


Password for dharma to access proprietary_db:
```

The isql command has other options for additional flexibility. See the *isql* reference section in Appendix A for a more detailed description of the *isql* command.

3.4.2 Client/Server

Before you build the Dharma SDK Server, first install the development components and the Dharma SDK ODBC Driver as described in Chapter 2.

Once you do that, complete the following steps:

- Stop the *dhdaemon* Dharma SDK Server process if it is running.
- Build a new version of the *dhdaemon* Dharma SDK Server executable.
- Restart the *dhdaemon* server to use the new executable.
- Create and load the data dictionary (only required once).

The following sections describe these steps in more detail.

3.4.2.1 Stopping the *dhdaemon* Process

On UNIX, use the *dhdaemon* stop command, as shown in the following example, to stop the server.

Example 3-8: Stopping the *dhdaemon* Dharma SDK Server Process

```
$ dhdaemon stop
                  Dharma/dhdaemon Version 09.00.0000
          Dharma Systems Inc           (C) 1988-2005.
```

```
                Dharma Systems Pvt Ltd         (C) 1988-2005.

Daemon version:           Feb 10 2005 17:02:43
      Running since:      02/11/2005 17:43:25   on bhima
Working directory:        /vol6/sdkdir
SQL-Server version:       /vol6/sdkdir/bin/dhdaemon
Nr of servers started:    0
              running:    0
              crashed:    0

dhdaemon stopped: PID 11292
```

If the Dharma SDK Server is already stopped, the *dhdaemon* stop command generates a *Connection refused* message:

```
$ dhdaemon -s sqlnw_ks stop

                Dharma/dhdaemon Version 09.00.0000
        Dharma Systems Inc              (C) 1988-2005.
        Dharma Systems Pvt Ltd          (C) 1988-2005.

Daemon:connect failed: Connection refused
Daemon:connect failed: Connection refused
```

Windows

On Windows, stop the *dhdaemon* service through the Windows Control Panel:

- Invoke the Windows Control Panel and select Services. In the list of services that appears, locate the entry for the *Dhdaemon* service.
- If the entry for *Dhdaemon* indicates the service is running, select it and click the Stop button.

3.4.2.2 Building the Client/Server Dharma SDK Server Executable

C Stubs

Unix

On UNIX, build the Dharma SDK Server by executing the makefile *$TPEROOT/odbcsdk/src/makefile*. You may need to modify the makefile to link with additional libraries and object modules specific to your implementation. The makefile in *$TPEROOT/odbcsdk/src* generates the *dhdaemon* Dharma SDK Server in the *$TPEROOT/bin* directory.

Log in as *dharma* before building the Dharma SDK Server. The following example shows building the *dhdaemon* executable from completed routine templates.

Example 3-9: Building the *dhdaemon* Dharma SDK Server for a Proprietary Storage System

```
$ cd $TPEROOT/odbcsdk/src
```

```
$ make
```

Java Stubs

On UNIX, build the Dharma SDK Server by executing the script *$TPEROOT/sdk4java/src/build.sh*. You may need to modify the script to link with additional libraries and object modules specific to your implementation. The script in *$TPEROOT/odbcsdk/src* generates the *dhdaemon* Dharma SDK Server in the *$TPEROOT/bin* directory.

Log in as *dharma* before building the Dharma SDK Server. The following example shows building the *dhdaemon* executable from completed routine templates.

Example 3-10: Building the *dhdaemon* Dharma SDK Server for a Proprietary Storage System

```
$ cd $TPEROOT/sdk4java/src
$ sh build.sh
```

C Stubs

Windows

On Windows, build the Dharma SDK Server by executing the *dhdaemon.mak* makefile in *%TPEROOT%\odbcsdk\src*. You may need to modify the makefile to link with additional libraries and object modules specific to your implementation. The *dhdaemon.mak* file in *%TPEROOT%\odbcsdk\src* generates the *dhdaemon* Dharma SDK Server in the *%TPEROOT%\bin* directory.

Open and build the *dhdaemon.mak* file in Microsoft Visual C++ to create the dhdaemon Dharma SDK Server.

Java Stubs

On Windows, build the Dharma SDK Server by executing the *build.bat* batch file in *%TPEROOT%\sdk4java\src*. You may need to modify the batch file to link with additional libraries and object modules specific to your implementation. The *build.bat* file in *%TPEROOT%\sdk4java\src* generates the *dhdaemon* Dharma SDK Server in the *%TPEROOT%\bin* directory.

3.4.2.3 Restarting the *dhdaemon* Service

Unix

On UNIX, use the *dhdaemon start* command, as shown in the following example, to start the server.

Example 3-11: Restarting the dhdaemon Process for the Proprietary Storage System

```
$ dhdaemon start
             Dharma/dhdaemon Version 09.00.0000
       Dharma Systems Inc          (C) 1988-2005.
       Dharma Systems Pvt Ltd      (C) 1988-2005.
Daemon started: PID 2669
```

Windows

On Windows:

- Invoke the Windows Control Panel and select *Services*. In the list that appears, select the entry for the *Dhdaemon* service.
- Click the Start button.

3.4.2.4 Creating the Data Dictionary

The executable *$TPEROOT/bin/mdcreate* is a utility to create a data dictionary that accepts metadata.

Log in as *dharma* before creating the data dictionary. Invoke the *mdcreate* utility and supply a name that will be used for the data dictionary and for access to the proprietary storage system.

The *mdcreate* utility creates a subdirectory called *<dbname>.dbs* under the *$TPEROOT* directory and populates the directory with the necessary files. The following example shows invoking *mdcreate* to create a database called *proprietary_db*, resulting in the directory *$TPEROOT/proprietary_db.dbs*.

Example 3-12: Using *mdcreate* to Create a Database

```
$ dharma/bin/mdcreate proprietary_db


            Dharma/mdcreate Version 09.00.0000
        Dharma Systems Inc           (C) 1988-2005.
        Dharma Systems Pvt Ltd       (C) 1988-2005.
$
```

3.4.2.5 Loading Metadata for the Proprietary Storage System

The executable *$TPEROOT/bin/isql* is a tool for loading metadata as well as executing interactive SQL queries. It accepts a script with special SQL CREATE TABLE and CREATE INDEX statements that insert metadata for existing tables.

Note If your implementation supports dynamic metadata, you do not need to load metadata using *isql*. Instead, the SQL engine loads metadata automatically each time an application connects to the database. See section 3.3.7 0 for details on dynamic metadata support.

You invoke *isql* on the server after the *dhdaemon* service is started. Log in as *dharma* before invoking *isql*.

You can create a SQL script file with CREATE TABLE and INDEX statements specific to your database. To load the metadata for your proprietary storage system, invoke *isql* to execute the script file. The following example shows invoking the SQL commands in a file called *md_script* to create metadata for a database called *proprietary_db*.

Example 3-13: Using *isql* to Load Metadata

```
$ isql -s $TPEROOT/md_script proprietary_db


             Dharma/isql Version 09.00.0000
       Dharma Systems Inc           (C) 1988-2005.
       Dharma Systems Pvt Ltd       (C) 1988-2005.


/vol6/sdkdir/bin/dhdaemon.exe <SQL SERVER 26517> -d
proprietary_db -h 415136 sqlnw
```

```
> > > Server 26517 done: Thu Nov 19 14:55:34 1998
```

The *isql* command has other options for additional flexibility. See the *isql* reference section in Appendix A for a more detailed description of the *isql* command.

3.5 SETTING DHARMA SDK RUNTIME VARIABLES

This section describes runtime variables that specify various characteristics of Dharma SDK behavior.

On Windows, initialization files set the variables. Edit one of the following files to change the default settings:

- Desktop: *%windir%/dhstodbc.ini*
- Client/server: *%windir%/dhsodbc.in*i

Add a line to the initialization file to set the desired environment variable. For instance:

```
TPE_DATADIR=E:\data\
```

On UNIX, set an environment variable for the user *dharma* at the command line or embed it in a script. For instance:

```
$ setenv TPE_DATADIR /usr/data/
```

3.5.1 Specifying the Main Dharma SDK Directory with TPEROOT

The TPEROOT variable specifies the main directory for the Dharma SDK installation.

TPEROOT must be set to build or run the Dharma SDK Server. There is no default for TPEROOT. When you install the Dharma SDK development components on Windows, the installation creates the appropriate initialization file for your configuration and sets TPEROOT to the directory you specified during the installation. On UNIX, you must set TPEROOT interactively or in a script.

Similarly, when you create a release kit to install your implementation of the Dharma SDK Server executable, the installation should make sure TPEROOT specifies the main directory for the installation.

TPEROOT does not have to be set on client systems.

The TPEROOT variable can hold the path in UNC notation. The UNC path notation will be in the following form:

- \\<machine-name\<share-point>\<path>
- TPEROOT=\\SmartStation\SmartC-Share\dharma

3.5.2 Specifying Location of the Data Dictionary with TPE_DATADIR

The TPE_DATADIR variable specifies the location for the data dictionary:

- The *mdcreate* utility creates a subdirectory to contain data dictionary files in the directory path specified by TPE_DATADIR.

- The Dharma SDK executable uses the path specified by TPE_DATADIR to access the data dictionary.

In the Desktop configuration, the *-d* argument to the *mdcreate* and *isql* commands overrides the value of TPE_DATADIR. If TPE_DATADIR is not set and *mdcreate* does not specify *-d*, the default is the path specified by TPEROOT.

3.5.3 Indicating Support for Dynamic Metadata with DH_DYNAMIC_METADATA

Storage managers indicate support for dynamic metadata through the DH_DYNAMIC_METADATA variable. If this environment variable is set to Y, the SQL engine relies on the storage manager to provide details on user tables and indexes.

3.5.4 Thread Safety of Dharma SDK ODBC Driver

By default, the Dharma SDK ODBC Driver is THREAD SAFE. However, this may not always be desired as it involves overhead on the performance of the system. Single-threaded ODBC applications do not require Thread Safety as only one thread is involved.

To disable the Thread Safety feature, the following runtime flag is used:

```
DH_DISABLE_ODBC_THREAD_SAFETY
```

This flag must be set in *dhs*odbc.ini* on Windows. In UNIX, it should set in environment for user *dharma*.

The default is Thread Safety enabled. To disable the Thread Safety feature, set this variable to the following:

```
DH_DISABLE_ODBC_THREAD_SAFETY = Y
```

3.5.5 Controlling Log File Output with TPESQLDBG

The Dharma SDK Server creates a log file called *sql_server.log* in the directory that contains the data dictionary (the directory is the *dbname.dbs* directory under the *$TPEROOT* directory).

The TPESQLDBG variable controls what logging information the SQL engine writes to *sql_server.log*. It has the format:

```
TPESQLDBG=xxxxxxxxxx
```

Specify Y or N for each occurrence of x to enable (Y) or disable (N) each of several categories of logging. For example, TPESQLDBG=YNNYNNNNNN enables the first and fourth categories of logging. The following table shows the logging category that each position in the TPESQLDBG variable enables:

Table 3-3: TPESQLDBG Logging Values

Position	Logging Category and Recommended Value
1	SQL: set to Y to log details of how the SQL engine processes SQL statements passed to it by applications. Details include: - Original SQL statement passed by the application - Decomposition of the statement by the engine parser - Optimization strategy chosen by the engine optimizer
2	Cache: logs the size of the binary trees created during processing. Generally, leave set to N (disabled).
3	Data dictionary manager: logs internal details of the internal logic used during processing. Generally, leave set to N (disabled).
4	Execution manager: set to Y to log details of runtime operations performed by the SQL engine execution manager.
5	Optimizer: set to Y to log details of runtime operations performed by the SQL engine optimizer.
6	Remote operations: set to Y to log details of remote operations.
7	Display cost: set to Y to log cost assessments calculated for each node in the SQL tree.
8	Heap manager handles: Set to Y to log summary information about the heap and parameter handles maintained by the heap manager.
9	Heap manager items: set to Y to log details of parameter and heap handle items.
10	Primitive heap manager: set to Y to log debug information about the primitive heap manager.

Generally, the only categories you ever need to enable are 1 and 4. TPESQLDBG provides useful information for debugging problems. For best performance, however, and to limit the size of the log file, do not set TPESQLDBG, or disable all TPESQLDBG logging categories:

```
TPESQLDBG=NNNNNNNNNN
```

3.5.6 Setting Default Date Format With TPE_DFLT_DATE

The TPE_DFLT_DATE variable allows users to change the default date format used by SQL. Changing the default date format affects:

- The default output format of date values.
- How SQL interprets date literals in queries and INSERT statements.

Set the TPE_DFLT_DATE variable to one of the following values to change the default date format:

- US_DFLT_DATE
- UK_DFLT_DATE

- ISO_DFLT_DATE

Changing the value of TPE_DFLT_DATE changes how SQL interprets character strings inserted as DATE values or compared to DATE columns. For example, setting TPE_DFLT_DATE to UK_DFLT_DATE allows users to supply date literals in British format (dd/mm/yyyy). The value of TPE_DFLT_DATE also determines the default output format of date data.

The following table details the different formats each value of TPE_DFLT_DATE supports. The boldface entries indicate the default output format for each setting.

Table 3-4: Date Formats Supported for Different Values of TPE_DFLT_DATE

US_DFLT_DATE	UK_DFLT_DATE	ISO_DFLT_DATE
	dd-mm-yyyy	
	dd/mm/yyyy	
	dd-mm-yy	
	dd/mm/yy	
mm-dd-yyyy		mm-dd-yyyy
mm/dd/yyyy		mm/dd/yyyy
mm-dd-yy		
mm/dd/yy		
yyyy-mm-dd	yyyy-mm-dd	yyyy-mm-dd
yyyy/mm/dd	yyyy/mm/dd	**yyyy/mm/dd**
dd-mon-yyyy	dd-mon-yyyy	dd-mon-yyyy
dd/mon/yyyy	dd/mon/yyyy	dd/mon/yyyy
dd-mon-yy	dd-mon-yy	
dd/mon/yy	dd/mon/yy	

Note You must change the value of the TPE_DFLT_DATE variable before starting the *dhdaemon* process. Once *dhdaemon* starts, changing TPE_DFLT_DATE will not affect the default date format.

The following example shows an interactive session on UNIX that uses the default input format through *isql*, changes the format to UK, then uses that input format.

Example 3-14: Using Different Date Input Formats

```
$ printenv TPE_DFLT_DATE
$ dhdaemon start -q

dhdaemon started: PID 17294
$ isql newff
> -- Insert with default input format:
> insert into dtest values ('5/7/1956');
```

```
1 record inserted.
> select to_char(c1, 'Month ddth') from dtest;
TO_CHAR(C1,MONTH DDTH)
----------------------
May        7th
1 record selected
> quit
$ dhdaemon stop
dhdaemon stopped: PID 17294
$ setenv TPE_DFLT_DATE uk_dflt_date
$ printenv TPE_DFLT_DATE
uk_dflt_date
$ dhdaemon start -q

dhdaemon started: PID 17305
$ isql newff
> -- Insert using UK-style date format:
> insert into dtest values ('5/7/1956');
1 record inserted.
> select to_char(c1, 'Month, ddth') from dtest;
TO_CHAR(C1,MONTH, DDTH)
-----------------------
May      ,  7th
July     ,  5th
2 records selected
>
```

3.5.7 Controlling Interpretation of Years in Date Literals With DH_Y2K_CUTOFF

By default, SQL generates an invalid date string error if the year component of date literals is specified as anything but 4 digits. The DH_Y2K_CUTOFF runtime variable changes this default behavior to allow 1- and 2-digit year specifications and control how SQL interprets them. (3-digit year specifications always generate an error.)

The following table lists the different values for DH_Y2K_CUTOFF and how SQL interprets them.

Table 3-5: Values of DH_Y2K_CUTOFF Runtime Variable

Value	Interpretation of 1- or 2-Digit Year in Date Literals
Not set	
Set to no value	
Less than zero Greater than 100	1- or 2-digit years not allowed

Table 3-5: Values of DH_Y2K_CUTOFF Runtime Variable

Value	Interpretation of 1- or 2-Digit Year in Date Literals
0	20th century: Adds 1900 to value (for instance, 99 denotes 1999).
100	21st century: Adds 2000 to value (for instance, 99 denotes 2099).
Greater that zero and less than 100	Depends on value of literal:
.	If value is less than DH_Y2K_CUTOFF, 21st century
.	If value is greater than or equal to DH_Y2K_CUTOFF, 20th century

Note Third-party tools have varying behavior regarding years represented as less than 4 digits. You may want to consider that behavior in choosing whether and how to use DH_Y2K_CUTOFF. For instance, current Microsoft Access behavior is equivalent to setting DH_Y2K_CUTOFF to 30. Microsoft Query behavior is similar to setting DH_Y2K_CUTOFF to 0.

The following example shows excerpts of interactive SQL sessions on UNIX that illustrate how changing the value of DH_Y2K_CUTOFF affects SQL's interpretation of the same 2-digit year in a date literal.

Note You must change the value of the DH_Y2K_CUTOFF variable before starting the *dhdaemon* process. Once *dhdaemon* starts, changing DH_Y2K_CUTOFF will not affect the previously-set behavior.

Example 3-15: Interpretation of 1- or 2-Digit Year in Date Literals

```
> -- DH_Y2K_CUTOFF not set:
> insert into dtest values('5/7/56');
error(-20230): Invalid date string
... Exit, set DH_Y2K_CUTOFF, stop and restart dhdaemon, re-invoke isql
...
> ! printenv DH_Y2K_CUTOFF
0
> insert into dtest values('5/7/56');
1 record inserted.
> select * from dtest;
C1
--
05/07/1956
1 record selected
... Exit, set DH_Y2K_CUTOFF, stop and restart dhdaemon, re-invoke isql
...
> ! printenv DH_Y2K_CUTOFF
100
> insert into dtest values('5/7/56');
1 record inserted.
```

```
> select * from dtest;
C1
--
05/07/2056
1 record selected
... Exit, set DH_Y2K_CUTOFF, stop and restart dhdaemon, re-invoke isql
> ! printenv DH_Y2K_CUTOFF
50
>  insert into dtest values('5/7/56');
1 record inserted.
> select * from dtest;
C1
--
05/07/1956
1 record selected
... Exit, set DH_Y2K_CUTOFF, stop and restart dhdaemon, re-invoke isql
> ! printenv DH_Y2K_CUTOFF
60
> insert into dtest values('5/7/56');
1 record inserted.
>  select * from dtest;
C1
--
05/07/2056
1 record selected
```

Chapter 4

Creating a Release Kit for Distributing the Dharma SDK Server

4.1 INTRODUCTION

Once you complete your implementation, you need to create a release kit to install the Dharma SDK on systems proprietary storage system. This chapter lists the files you need to include in the release kit.

4.2 DESKTOP

The release kit must include the files listed in the following table:

Table 4-1: Files Required for Dharma SDK Server Desktop Release Kit

File	Description
bin\dhstodbc.dll	Dharma SDK ODBC Driver DLL (built from implemented storage interfaces).
bin\dhstsetp.dll	Setup DLL for configuring ODBC data sources.
bin\mdcreate.exe	Utility to create a data dictionary.
bin\isql.exe	Utility for loading metadata and executing simple SQL queries.
lib\dherrors	Dharma error mapping file. The release kit must copy this file to the *%TPEROOT%\lib*\ directory.
lib\sql_conf	Configuration file used by isql tool
md_script	Completed SQL script to load metadata corresponding to data in the proprietary storage system.
*classes	This directory contains class files for the java storage system

Table 4-1: Files Required for Dharma SDK Server Desktop Release Kit

File	Description
%windir%\system32\msvcrt.dll %windir%\system32\mtxdm.dll %windir%\system32\odbc16gt.dll %windir%\system32\odbc32.dll %windir%\system32\odbc32gt.dll %windir%\system32\odbccad32.exe %windir%\system32\odbccp32.cpl %windir%\system32\odbccp32.dll %windir%\system32\odbccr32.dll %windir%\system32\odbccu32.dll %windir%\system32\odbcinst.cnt %windir%\system32\odbcinst.hlp %windir%\system32\odbcinst.dll %windir%\system32\odbctrac.dll	Files required to run ODBC. The installation of the Desktop development components creates these files if they do not already exist.
%windir%\dhstodbc.ini	Initialization file containing environment variables. This file must at least set the TPEROOT environment variable.

*This is applicable only for SDK for Java.

4.3 CLIENT/SERVER

The release kit must include the files listed in the following table:

Table 4-2: Files Required for Dharma SDK Server Client/Server Release Kit

File	Description
bin/dhdaemon	Executable for Dharma SDK Server.
bin/mdcreate	Utility to create a data dictionary.
bin/isql	Utility for loading metadata.
bin/pcntreg.exe	Utility to add and delete entries for the Dharma SDK in the Windows registry
lib/dherrors	Dharma error mapping file. The release kit must copy this file to the lib subdirectory under the directory pointed to by the TPEROOT environment variable..
lib/sql_conf	Configuration file used by isql tool
md_script	Completed SQL script to load metadata corresponding to data in the proprietary storage system.
*classes	This directory contains class files for the java storage system
%windir%\dhsodbc.ini	Windows only: Initialization file containing environment variables. This file must at least set the TPEROOT environment variable.

*This is applicable only for SDK for Java.

The method for building the files into a kit varies between UNIX and Windows.

On UNIX, create a *tar* file.

On Windows, create a *setup.exe* as a self-extracting executable.

4.4 PROVIDING JAR FILE FOR SDK FOR JAVA

In case you wish to provide a jar file instead of class files for SDK for Java, follow the instructions given below.

1. Create a jar file from the classes directory under the installation directory

E.g: jar cvf StorageStubs.jar dharma

2.Supply this jar file in the install kit. The name of the jar file should be set in the CLASSPATH

Chapter 5

Storage Interface Reference

5.1 COMMON DATA STRUCTURES

The file *$TPEROOT/odbcsdk/src/dhcs.h* defines a number of data structures that are used as common arguments across several different storage interfaces. Table 5–1 lists the major common data structures, and the following sections describe them in more detail.

See the *dhcs.h* header file for definitions of other structures and types referred to in these structures.

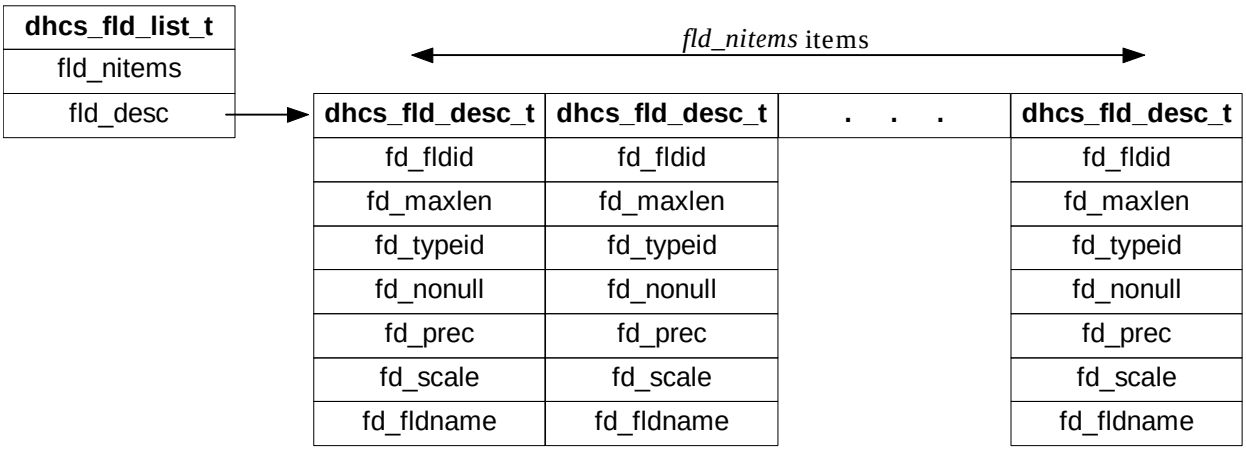
Table 5-1: Major Common Data Structures Defined in *dhcs.h*

Structure	Purpose
dhcs_fld_list_t	Contains a list of table fields passed to *dhcs_add_table*.
dhcs_fld_desc_t	Pointed to by *dhcs_fld_list_t*. Also passed directly to *dhcs_get_colinfo*. Contains the table field id and other details describing individual fields.
dhcs_keydesc_t	Contains a list of index keys, passed to *dhcs_create_index*.
dhcs_kfld_desc_t	Pointed to by *dhcs_keydesc_t*. Contains the index key id, its corresponding table field id, and other details about the key.
dhcs_fldl_val_t	Contains a list of field values, passed to the following routines: *dhcs_tpl_insert* *dhcs_tpl_update* *dhcs_tpl_fetch* *dhcs_tpl_scan_fetch* *dhcs_ix_insert* *dhcs_ix_delete* *dhcs_ix_scan_open* *dhcs_ix_scan_fetch*
dhcs_fv_item_t	Pointed to by d*hcs_fldl_val_t*. Contains the table field id (for *dhcs_tpl* routines) or index key id (for *dhcs_ix* routines) and a pointer to the value of the table field or index key.
dhcs_data_t	Pointed to by *dhcs_fv_item_t*. Contains field values (or, for long data types, field handles) and data type information.

5.1.1 Table Field Lists: *dhcs_fld_list_t and dhcs_fld_desc_t*

The following figure shows the structures that make up a table field list.

Figure 5-1: Table Field Lists: *dhcs_fld_list_t* and *dhcs_fld_desc_t*



5.1.1.1 dhcs_fld_list_t

The *dhcs_fld_list_t* structure contains a list of fields that describe the columns in an SQL table that an ODBC application is creating. The SQL engine passes a structure of type *dhcs_fld_list_t* when it calls the *dhcs_add_table* routine.

Definition

```
typedef struct  {
    short              fld_nitems ;
    dhcs_fld_desc_t    *fld_desc ;
} dhcs_fld_list_t ;
```

Field Descriptions

fld_nitems	An integer that specifies the number of fields in the list (in other words, the number of columns in the table being created).
fld_desc	An array of structures of type *dhcs_fld_desc_t*. Each element of the array contains detail on a table column.

5.1.1.2 dhcs_desc_t

The *dhcs_desc_t* structure contains detail on a single column in an SQL table. Storage managers must process the *dhcs_desc_t* structure in the following cases:

- When the SQL engine calls *dhcs_add_table* routine, the *dhcs_fld_list_t* structure includes a pointer to an array of *dhcs_desc_t* structures. In this case, the *dhcs_desc_t* structure is filled in with values, and the storage manager creates a column according to the detail in the *dhcs_desc_t* structure.
- When the SQL engine calls the *dhcs_get_colinfo* routine, an element in the info argument is a structure of type *dhcs_desc_t*. In this case, the *dhcs_desc_t* structure is empty, and the implementation fills it in with the appropriate values.

Definition

```
typedef struct {
    dhcs_field_t      fd_fldid ;
    unsigned short    fd_maxlen;
    dhcs_typeid_t     fd_typeid;
    dh_boolean        fd_nonull;
    short             fd_prec;
    short             fd_scale;
    char              *fd_fldname;
} dhcs_fld_desc_t ;
```

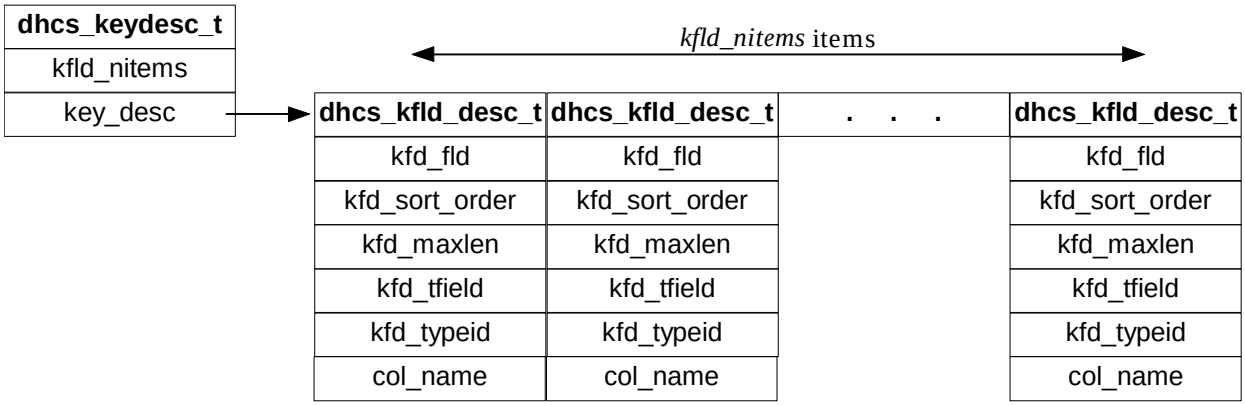
Field Descriptions

fd_fldid	A short integer that identifies the column. The value in *fd_fldid* can be any value that is appropriate for the storage system (column sequence or byte offset, for example). The value for *fd_fldid* must uniquely identify the column within the table. Implementations must keep track of column identifiers and their corresponding names. The SQL engine passes only the identifier, not the name, in subsequent calls. It is the implementation's responsibility to associate the identifier with the correct column.
fd_maxlen	A short integer that specifies:· - For fixed-length data types, the fixed length - For variable-length data types, the maximum length
fd_typeid	A short integer that specifies the SQL data type.
fd_nonull	A Boolean value that specifies whether the column allows null values. A value of TRUE indicates that the column does not allow null values.
fd_prec	A short integer that specifies the maximum number of digits for numeric types.
fd_scale	A short integer that specifies the number of digits to the right of the decimal point for numeric types.
fd_fldname	A null terminated character string that contains the column name.

5.1.2 Index Key Lists: *dhcs_keydesc_t and dhcs_kfld_desc_t*

The following figure shows the fields in the structures that make up an index key list.

Figure 5-2: Index Key Lists: *dhcs_keydesc_t* and *dhcs_kfld_desc_t*



5.1.2.1 dhcs_keydesc_t

The *dhcs_keydesc_t* structure contains a list of fields that describe the keys in an SQL index that an ODBC application is creating. The SQL engine passes a structure of type *dhcs_keydesc_t* when it calls the *dhcs_create_index* routine.

Definition

```
typedef struct  {
    short               kfld_nitems ;
    dhcs_kfld_desc_t    *kfld_desc ;
} dhcs_keydesc_t ;
```

Field Descriptions

kfld_nitems	An integer that specifies the number of fields in the list (in other words, the number of keys in the index being created).
key_desc	An array of structures of type *dhcs_kfld_desc_t*. Each element of the array contains detail on an index key column.

5.1.2.2 dhcs_kfld_desc_t

The *dhcs_kfld_desc_t* structure contains detail on an index key column.

Definition

```
typedef struct {
    dhcs_field_t        kfd_field;
    unsigned char       kfd_sort_order;
    unsigned short      kfd_maxlen;
    dhcs_field_t        kfd_tfield;
    dhcs_typeid_t       kfd_typeid;
    char                col_name[DHCS_MAX_FLDLEN_P1];
} dhcs_kfld_desc_t ;
```

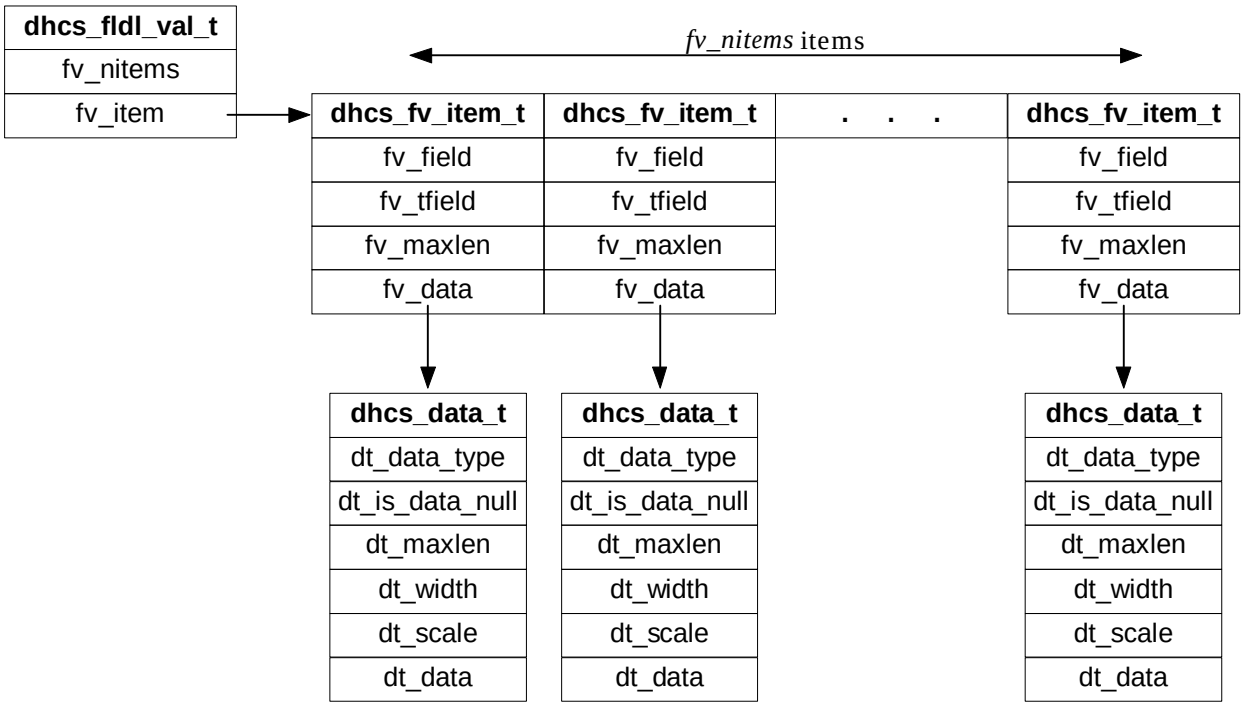
Field Descriptions

kfd_field	A short integer that identifies the index key field. Unlike the table field identifiers passed to *dhcs_add_table*, storage systems should not modify the index key field identifier value in *kfd_field*.
kfd_sort_order	A character that indicates the sort order specified in the SQL CREATE INDEX statement.
kfd_maxlen	A short integer that specifies the maximum length for data in the index key. This will be the same value as for the table field that corresponds to the index key.
kfd_tfield	The field identifier for the table field that corresponds to the index key.
kfd_typeid	A long integer that specifies the SQL data type. This will be the same value as for the table field that corresponds to the index key.
col_name	A null terminated character string that contains the column name. This will be the same value as for the table field that corresponds to the index key.

5.1.3 Field Value Lists: *dhcs_fldl_val_t* and Associated Structures

The following figure shows the fields in the structures that make up a field value list.

Figure 5-3: Field Value Lists: *dhcs_fldl_val_t* and Associated Structures



5.1.3.1 dhcs_fldl_val_t

The *dhcs_fldl_val_t* structure contains a list of structures that contain or will receive data values:

- The SQL engine passes a field value list that includes data values when it supplies values to be inserted or updated in a record (*dhcs_tpl_insert*, *dhcs_tpl_update*, *dhcs_ix_insert*, and *dhcs_ix_delete*) or to specify search criteria for retrieving records (*dhcs_ix_scan_open* and *dhcs_ix_scan_fetch*). In these cases, the field value list is strictly an input parameter.
- The SQL engine passes a field value list with empty data value fields for the storage system to fill in with values retrieved from the appropriate table or index record (*dhcs_tpl_fetch*, *dhcs_tpl_scan_fetch*, and *dhcs_ix_scan_fetch*). In these cases, the field value list is both an input and output parameter. On input, each element of the list includes the table or index field identifier of interest, as well as other detail about the field. On output, the storage system supplies values in the data value fields.

Definition

```
typedef struct {
    short            fv_nitems ;
    dhcs_fv_item_t   *fv_item ;
} dhcs_fldl_val_t ;
```

Field Descriptions

fv_nitems	Number of field values in the list.
fv_item	Pointer to an array of field values (structures of type *dhcs_fv_item_t*). Each element of the array contains identifiers, detail about the field, and the actual data.

5.1.3.2 dhcs_fv_item_t

The *dhcs_fv_item_t* structure contains field identifiers for one of the fields in a field value list, and a pointer to another structure that contains or will receive the value for that field.

Definition

```
typedef struct {
    dhcs_field_t          fv_field;
    dhcs_field_t          fv_tfield;
    unsigned short        fv_maxlen;
    dhcs_data_t           *fv_data;
} dhcs_fv_item_t ;
```

Field Descriptions

fv_field	Contains the table field identifier (for *dhcs_tpl* routines) or index key identifier (for *dhcs_ix* routines).

fv_tfield Used only during *dhcs_ix_scan_fetch* routines, *fv_tfield* contains table field identifiers for all the field values needed to satisfy a particular query.

For table fields that are also index keys, then, *fv_tfield* contains the table field identifier that corresponds to the index key identifier in *fv_field*. If a query requires a field value that is not also an index key, the SQL engine sets *fv_field* to SQL_INVAL_FLDID to indicate there is no index key for the field

If the storage system returned a value of TRUE when the SQL engine called *dhcs_rss_get_info* with an *info_type* of DHCS_IX_FETCH_ALL_FIELDS, the storage system fetches values for those fields as well as the index component fields when it processes *dhcs_ix_scan_fetch*.

fv_maxlen A short integer that specifies:·

- For fixed-length data types, the defined length
- For variable-length data types, the maximum length

fv_data Pointer to a structure that contains field values (or, for long data types, field handles) and data type information.

5.1.3.3 dhcs_data_t

The *dhcs_data_t* structure contains a field value (or, for long data types, field handles) and data type information for one of the fields in a field value list.

Definition

```
typedef struct {
    dhcs_typeid_t        dt_data_type;
    dh_boolean           dt_is_data_null;
    unsigned short       dt_maxlen;
    unsigned short       dt_data_len;
    short                dt_width;
    short                dt_scale;
    void                 *dt_data;
} dhcs_data_t ;
```

Field Descriptions

fv_nitems Number of field values in the list.

dt_data_type A long integer that specifies the SQL data type.

dt_is_data_null A Boolean value that specifies whether the column contains a null value. A value of TRUE indicates that the column is null.

dt_maxlen A short integer that specifies:·

- For fixed-length data types, the defined length
- For variable-length data types, the maximum length

dt_data_len A short integer that specifies, for variable-length data types only, the actual length of the data.

dt_width	A short integer that specifies the maximum number of digits for numeric types.
dt_scale	A short integer that specifies the number of digits to the right of the decimal point for numeric types.
dt_data	A pointer to storage for the data value (or, for long data types, the field handle).

5.2 TABLE INTERFACES

5.2.1 dhcs_add_table

Adds a table to a storage manager, or generates an identifier for an existing table.

Syntax

```
extern dhcs_status_t
dhcs_add_table (
    dhcs_fld_list_t     *fld_list,
    char                *table_name
    dh_boolean           meta_data_only,
    dhcs_fld_list_t     *primary_key_list,
    dhcs_tableid_t      *table_id,
    char                *owner,
    void                *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

INOUT fld_list

A list of field descriptions for the columns in the table.

Field definition information includes the field name, the field identifier, the field type, and a flag that indicates whether null values are allowed. Additionally, depending on the data type of the field, the length or the maximum length of the data type may be provided. Before returning from *dhcs_add_table*, the storage manager can change the identifiers in *fld_list* to any value that is appropriate for the storage system. See section 5.1 for details on the data structures that make up a field list.

Note The SQL engine uses the sequence of field identifiers to determine column order when SQL SELECT and INSERT statements do not explicitly specify column names. Because of this, when storage managers modify field identifiers, they must exercise care to avoid changing the values in a way that alters their original sequence. Consider the following table definition:

```
CREATE TABLE test (col1 INT, col2 INT)
```

If the storage manager changes the sequence of field identifiers (for instance, if it assigns a field identifier of 2 to *col1* and a field identifier of 1 to *col2*), then the statement INSERT INTO test VALUES (1,2) will store the value 1 into *col2* and 2 into *col1*.

Implementations must keep track of field identifiers and their corresponding field names. The SQL engine passes only the identifier, not the name, in subsequent calls. It is the implementation's responsibility to associate the identifier with the correct field.

IN table_name

The name of the table that is being created. *table_name* will contain the name as specified in the CREATE TABLE statement.

If the CREATE TABLE statement also specified 'METADATA_ONLY' in the STORAGE_ATTRIBUTES clause, *table_name* will contain the name of an existing table in the proprietary storage system.

IN meta_data_only

A flag that indicates the SQL engine is inserting metadata into the system catalog tables for a table that already exists in the proprietary storage system. The storage manager should not create a new table, but instead return a table id for the SQL engine to associate with the existing table name.

The SQL engine sets this flag to TRUE when the CREATE TABLE statement specified 'METADATA_ONLY' in the STORAGE_ATTRIBUTES clause. Unless they support dynamic metadata (see section 5.5), implementations use this mechanism to load metadata for existing tables. If implementations do support dynamic metadata, they should ignore calls that set the *meta_data_only* flag.

IN primary_key_list

A list of primary key fields.

primary_key_list will be a subset of the fields specified in the *fld_list* argument. This list will be empty unless primary key fields were specified with the CREATE TABLE statement. A primary key is characterized by the constraint that no two records in a table may have the same primary key value, and that no fields of the primary key may have a null value.

Storage systems that support primary keys can use this information to create the primary key for the table. Storage systems that do not support primary keys can ignore this information.

In addition to passing down the primary key list, the SQL engine will automatically create a unique index on the primary key fields. Creating this index allows storage systems that do not directly support primary keys to support them indirectly via the index.

To create the primary-key index, the SQL engine calls *dhcs_create_index* (see section 5.3.1) with the unique argument set to TRUE, and the *ix_type* argument set to B. The SQL engine generates a unique name for the index, prefixed with SYS_, and passes it as the *index_name* argument. The components of the index will be the fields that make up the primary key in the order that they appear in the table, and the sort order for each index component is ascending.

OUT table_id

The table id for the table that was created or generated for an existing table.

The table id is a unique identifier that will be used on subsequent calls to identify the table. The SQL engine stores this id in the SYSTABLES catalog table along with the table name. The SQL engine reserves table identifiers below 1000 and above 32767. Implementations must generate table identifiers within those values.

Implementations must keep track of table identifiers and their corresponding table names. The SQL engine passes only the identifier, not the name, in subsequent calls. It is the implementation's responsibility to associate the identifier with the correct table.

IN owner

A character string that specifies the user issuing the CREATE TABLE statement. Implementations can ignore this argument.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

5.2.2 dhcs_drop_table

Deletes a table from the proprietary storage system.

Syntax

```
extern dhcs_status_t
dhcs_drop_table (
    dhcs_tableid_t   tableid,
    dh_boolean       meta_data_only,
    void             *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN tableid

The id for the table that is being dropped.

IN meta_data_only

A flag that indicates the SQL engine is only deleting metadata from the system catalog tables for the specified table. The SQL engine sets this flag to TRUE when the DROP TABLE statement specified 'METADATA_ONLY' in the STORAGE_ATTRIBUTES clause. Unless they support dynamic metadata, implementations use this mechanism to unload metadata for tables that have been deleted in the underlying storage system

through means other than the Dharma SDK. If implementations do support dynamic metadata, they should ignore calls that set the *meta_data_only* flag.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

dhcs_drop_table is called as a direct result of the DROP TABLE statement. Tableid serves to identify the table to be dropped. By calling *dhcs_drop_table*, the SQL engine is informing the storage system that the table is no longer needed, and that it effectively may be destroyed.

5.2.3 dhcs_tpl_close

Closes a table that was opened for non-scan operations.

Syntax

```
extern dhcs_status_t
dhcs_tpl_close (
    void    *tpl_hdl,
    void    *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN tpl_hdl

A handle for the table, as returned by *dhcs_tpl_open*.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

Closes a table that was opened within a storage manager.

5.2.4 dhcs_tpl_delete

Deletes a record from a table.

Syntax

```
extern dhcs_status_t
dhcs_tpl_delete (
    void    *tpl_hdl,
    void    *tid,
    void    *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.
SQL_NOT_FOUND	If the tid does not identify a valid record.

Arguments

IN tpl_hdl

A handle for the table, as returned by *dhcs_tpl_open*.

IN tid

The tid for the record to be deleted.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

dhcs_tpl_delete deletes a record from a table. The tid argument identifies the record to be deleted. Once a record is deleted, the tid assigned to it may be assigned to a new record.

After calling *dhcs_tpl_delete*, the SQL engine will call *dhcs_rss_get_info* with the DHCS_IX_UPD_REQUIRED flag:

- If TRUE is returned, then the SQL engine will update any corresponding indexes appropriately.
- If FALSE is returned, then the SQL engine assumes that the storage system will update the corresponding indexes during the execution of *dhcs_tpl_delete*.

5.2.5 dhcs_tpl_fetch

Fetches a specific record from a table.

Syntax

```
extern dhcs_status_t
dhcs_tpl_fetch (
    void                    *tpl_hdl,
    void                    *tid,
    dhcs_tpl_fetch_hint_t   fetch_hint,
    dhcs_fldl_val_t         *field_values,
    void*conn_hdl

) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.
SQL_NOT_FOUND	If the tid does not identify a valid record.

Arguments

IN tpl_hdl

A handle for the table, as returned by *dhcs_tpl_open*.

IN tid

The tid that identifies the record to be fetched

IN fetch_hint

Indicates if the record is being fetched in the context of a SQL statement which only performs read operations or if it is being executed in the context of a SQL statement that could perform writes. *fetch_hint* will be one of the following values:

DHCS_TPL_FH_READ	The record being fetched is not a candidate for being updated in the context of the current SQL statement.
DHCS_TPL_FH_WRITE	The record being fetched is a candidate for being updated in the context of the current SQL statement.

INOUT field_values

A field value list in which the storage system returns field values fetched for the record.

If any of the values are for columns defined with LONG VARCHAR or LONG VARBINARY data types, then the field values for those columns do not contain actual data. For such columns, on output, the storage manager supplies a field handle that identifies storage for the data.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the

dhcs_rss_init routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

dhcs_tpl_fetch fetches a record from a table. Tid identifies the record within the table that is to be fetched.

field_values is a pointer to a list of field items. *field_values* may contain field items for all of the fields within the record, or it may contain field items for a subset of the fields.

Each field item is a structure of type *dhcs_fv_item_t*. Each field-item structure contains a field id that identifies a field within the record whose value is to be returned. Each field-item structure also points to another structure, of type *dhcs_data_t*, to store the field value.

Using the field id, the storage system should extract the appropriate field value (or, for long data types, the field handle) from the retrieved record and store it in the *dhcs_data_t* structure.

fetch_hint indicates whether the record that is being fetched is a candidate for being updated in the context of the current SQL statement. A storage system may wish to use this information when making concurrency control decisions (locking) relative to the record being fetched.

Note that *fetch_hint* is in fact just a hint. It is strictly relative to the current SQL statement. Even if *fetch_hint* is set to DHCS_TPL_FH_READ, it does not imply that the record being fetched was not already updated earlier in the transaction, or that it will not be updated at some future point during the execution of the transaction.

5.2.6 dhcs_tpl_insert

Inserts a record into a table.

Syntax

```
extern dhcs_status_t
dhcs_tpl_insert (
    void              *tpl_hdl,
    dhcs_fldl_val_t   *field_values,
    void              * tid,
    void              *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN tpl_hdl

A handle for the table, as returned by *dhcs_tpl_open*.

INOUT field_values

The list of field values for the record that is to be inserted into the table. A value exists for each field in the table.

If any of the values are for columns defined with LONG VARCHAR or LONG VARBINARY data types, then the field values for those columns do not contain actual data. For such columns, the *data_t* component of *field_values* is both an input and output argument. On output, the storage manager supplies a field handle that identifies storage for the data. (The implementation should initialize this storage, since there is no guarantee that the ODBC application will actually request the SQL engine to call *dhcs_put_data* to store data.)

INOUT tid

The tuple identifier (tid) assigned to the record that the storage system inserted.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

dhcs_tpl_insert is used to insert a record into a table. *field_values* contains the list of field values (or, for long data types, storage for field handles) for the record to be inserted. There is one field value for each field that makes up the table. The fields are ordered in the list by their field id.

The storage system must assign a tuple identifier (tid) to the record that is inserted. This tid is returned via the tid argument. The tid argument is an INOUT argument. On input it will contain a NULL_TID value that is appropriate for the storage system that is being accessed. The SQL engine allocated the NULL_TID by calling *dhcs_alloc_tid*. On output, the tid must contain a tid value that can be used by the SQL engine to relocate the record that was inserted. The SQL engine may use the tid on subsequent calls to other functions to identify the record that was inserted.

Note that the SQL engine imposes no requirement on a storage manager relative to the order of records within a table. The storage manager determines a new record is inserted into the table relative to other, already-existing records.

After calling *dhcs_tpl_insert*, the SQL engine will call *dhcs_rss_get_info* with the DHCS_IX_UPD_REQUIRED flag:

- If TRUE is returned, then the SQL engine will update any corresponding indexes appropriately by calling *dhcs_ix_insert*.
- If FALSE is returned, then the SQL engine assumes that the storage system will update the corresponding indexes during the execution of *dhcs_tpl_insert*. If

inserting the record into the associated indexes would result in a duplicate index key value for a unique index, then the record should not be stored in the table or the index, and an error returned.

5.2.7 dhcs_tpl_open

Opens a table for non-scan operations.

Syntax

```
extern dhcs_status_t
dhcs_tpl_open (
    dhcs_tableid_t   tableid,
    void             **tpl_hdl,
    void             *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN tableid

The identifier for the table that is being opened for scanning. The SQL engine obtains tableid from the SYSTABLES system catalog table.

OUT tpl_hdl

A handle for the table. The format of the handle is specific to the storage system. The SQL engine passes the handle in subsequent calls to *dhcs_tpl_insert*, *dhcs_tpl_delete*, *dhcs_tpl_update*, and *dhcs_tpl_close*.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

The SQL engine calls *dhcs_tpl_open* to open a table for non-scan operations. In response, the storage manager makes sure the table is open and supplies a handle that the SQL engine passes to subsequent routines.

Although the SQL engine presumes that the table specified by tableid is open after calling *dhcs_tpl_open*, the storage manager should not automatically open files or load data structures each time the SQL engine calls this function. This is because previous SQL statements may have resulted in calls to functions that already opened the table. Instead, the storage manager should use whatever file-caching mechanism

exists in the underlying storage system to check if the table is already open, and open it only if necessary.

5.2.8 dhcs_tpl_scan_close

Closes a table that was opened for scanning.

Syntax

```
extern dhcs_status_t
dhcs_tpl_scan_close (
    void     *tpl_scan_hdl,
    void     *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN tpl_scan_hdl

A handle for the scan, as returned by *dhcs_tpl_scan_open*.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

Closes a table that was opened for scanning within a storage manager.

5.2.9 dhcs_tpl_scan_fetch

Fetches the next record from a table.

Syntax

```
extern dhcs_status_t
dhcs_tpl_scan_fetch (
    void      *tpl_scan_hdl,
    dhcs_fldl_val_t     *fld_values,
    void     *tid,
    void     *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.
SQL_NOT_FOUND	When no more records exist in the scan.

Arguments

IN tpl_scan_hdl

A handle for the scan, as returned by *dhcs_tpl_scan_open*.

INOUT field_values

A field value list in which the storage system returns field values fetched for the record.

If any of the values are for columns defined with LONG VARCHAR or LONG VARBINARY data types, then the field values for those columns do not contain actual data. For such columns, on output, the storage manager supplies a field handle that identifies storage for the data.

OUT tid

A pointer to a tuple identifier for the record that was fetched.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

dhcs_tpl_scan_fetch fetches the next record from a table scan. When a table scan is opened, the scan is positioned before the first record of the table. Each call to *tpl_scan_fetch*, results in the scan being moved to the next record of the table, and the field values from the record being returned.

field_values is a pointer to a list of field items. Each field item identifies a field within the retrieved record whose value is to be returned, and provides a location to store the field value (or, for long data types, the field handle). If *field_values* is NULL, it indicates that no field values are to be returned for the record. If *field_values* is non-NULL, then a value must be returned for each field for which a field item is specified. *field_values* may contain a field item for each field in the record, or it may contain field items for a subset of the fields.

Tid provides a location to return the tid for the retrieved record. If tid is a NULL pointer, then it indicates that the tid value for the record is not to be returned. If tid, is non-NULL pointer, then a value must be returned for the tid.

5.2.10 dhcs_tpl_scan_open

Opens a table for scanning when no indexes are available.

Syntax

```
extern dhcs_status_t
dhcs_tpl_scan_open (
    dhcs_tableid_t          tableid,
    dhcs_tpl_fetch_hint_t   fetch_hint,
    void                    **tpl_scan_hdl,
    void                    *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN tableid

The identifier for the table that is being opened for scanning. The SQL engine obtains tableid from the SYSTABLES system catalog table.

IN fetch_hint

Indicates if the scan is being performed in the context of an update statement:

DHCS_TPL_FH_READ	The table is being scanned in the context of a read statement
DHCS_TPL_FH_WRITE	The table is being scanned and selected records may be updated

fetch_hint indicates whether the scan is in the context of an update statement. It indicates that the SQL engine may ultimately update a selected record via the *dhcs_tpl_update* or delete a selected record via the *dhcs_tpl_delete* member functions. This flag may be used by storage systems whose concurrency control policy (locking policy) needs to differentiate or wishes to differentiate between reading a record and reading a record for update.

OUT tpl_scan_hdl

A handle for the scan. The format of the handle is specific to the storage system. The SQL engine passes the handle in subsequent calls to *dhcs_tpl_scan_fetch* and *dhcs_tpl_scan_close*.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

The SQL engine calls *dhcs_tpl_scan_open* to open a table for scanning. In response, the storage manager makes sure the table is open and supplies a scan handle that the SQL engine passes to subsequent scan routines.

Although the SQL engine presumes that the table specified by tableid is open after calling *dhcs_tpl_scan_open*, the storage manager should not automatically open files or load data structures each time the SQL engine calls this function. This is because previous SQL statements may have resulted in calls to functions that already opened the table. Instead, the storage manager should use whatever file-caching mechanism exists in the underlying storage system to check if the table is already open, and open it only if necessary.

5.2.11 dhcs_tpl_update

Updates values in an existing table record.

Syntax

```
extern dhcs_status_t
dhcs_tpl_update (
    void                *tpl_hdl,
    void                *tid,
    dhcs_fldl_val_t     *update_field_values,
    void                *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.
SQL_NOT_FOUND	If the tid does not identify a valid record.

Arguments

IN tpl_hdl

A handle for the table, as returned by *dhcs_tpl_open*.

IN tid

The tid that identifies the record to be updated.

IN update_field_values

The list of field values for the record that is to be updated. A value exists for each field in the table that is to be updated.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK

Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

dhcs_tpl_update updates a record in a table. Tid identifies the record within the table that is to be updated. *update_field_values* contains a list of field values items. Each field value item identifies a field to be updated and the new value for the field. Note that only fields to be updated are contained in the list.

For each field that is to be updated the value of that field is replaced by the value that was extracted from *update_field_values*.

After calling *dhcs_tpl_delete*, the SQL engine will call *dhcs_rss_get_info* with the DHCS_IX_UPD_REQUIRED flag:

- If TRUE is returned, then the SQL engine will update any corresponding indexes appropriately.
- If FALSE is returned, then the SQL engine assumes that the storage system will update the corresponding indexes during the execution of *dhcs_tpl_update*.

5.3 INDEX INTERFACES

5.3.1 dhcs_create_index

Creates an index for a table within a storage manager, or generates an identifier for an existing index.

Syntax

```
extern dhcs_status_t
dhcs_create_index (
    dhcs_tableid_t     tableid,
    dh_boolean         unique,
    dh_boolean         meta_data_only,
    char               ix_type,
    dhcs_keydesc_t     *keydesc,
    char               *index_name,
    dhcs_indexid_t     *indexid,
    void               *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN tableid

The table for which the index is being created.

IN unique

A flag that indicates whether records in the index must be unique. If TRUE, the index is unique. If FALSE, then the index allows duplicate records.

IN meta_data_only

A flag that indicates the SQL engine is inserting metadata into the system catalog tables for an index that already exists in the proprietary storage system. The storage manager should not create a new index, but instead return an index id for the SQL engine to associate with the existing index name.

The SQL engine sets this flag to TRUE when the CREATE INDEX statement specified 'METADATA_ONLY' in the STORAGE_ATTRIBUTES clause. Unless they support dynamic metadata (see section 3.3.7), implementations use this mechanism to load metadata for existing indexes. If implementations do support dynamic metadata, they should ignore calls that set the *meta_data_only* flag.

IN ix_type

A flag that indicates the type of index. The SQL engine passes the TYPE argument specified in an application's SQL CREATE INDEX statement. If the CREATE INDEX statement did not include the TYPE argument, *ix_type* is set to B.

The *ix_type* argument does not imply any particular indexing technique. It is an arbitrary flag that allows the storage manager to indicate differing support for multiple types of indexes. The SQL engine calls *dhcs_rss_get_info* for each index type, and the storage manager can respond with different index properties for each type. (For instance, that different index types support different comparison operators.)

Note If the data type of the index key column is LONG VARCHAR or LONG VARBINARY, the SQL engine generates an error if the index type supports any operators other than DHCS_IXOP_CONTAINS and DHCS_IXOP_NOTCNTNS. This restriction means that SQL CREATE INDEX statements that specify long data-type columns as index keys must also specify the TYPE argument.

The SQL engine also passes the *ix_type* value when it opens the index through the *dhcs_ix_scan_open* or *dhcs_ix_open* routines.

IN keydesc

A description of the index key fields. Field information includes a key-field identifier, the corresponding field identifier in the table, data type, maximum length, and sort order.

IN index_name

The name of the index that is being created. *index_name* will contain the name as specified in the CREATE INDEX statement.

If the CREATE INDEX statement also specified 'METADATA_ONLY' in the STORAGE_ATTRIBUTES clause, *index_name* will contain the name of an existing index in the proprietary storage system.

OUT indexid

The id assigned by the storage system for the created index.

The index id is a unique identifier that will be used on subsequent calls to identify the index. The SQL engine stores this id in the SYSINDEXES catalog table along with the index name. The SQL engine reserves index identifiers below 1000 and above 32767. Implementations must generate index identifiers within those values.

Implementations must keep track of index identifiers and their corresponding index names. The SQL engine passes only the identifier, not the name, in subsequent calls. It is the implementation's responsibility to associate the identifier with the correct index.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

Creates an index for a table. Tableid identifies the table for which the index is being created.

Keydesc provides the descriptive information that is needed to create the index, including the number of components in the index and the sort order for records in the index.

This routine returns an indexid. The indexid is a number generated by the storage system that will be used on subsequent calls to identify the index. The SQL engine will store this id in the *sysindexes* catalog table along with index name.

5.3.2 dhcs_drop_index

Deletes an index from the proprietary storage system.

Syntax

```
extern dhcs_status_t
dhcs_drop_index (
    dhcs_tableid_t     tableid,
    dhcs_indexid_t     indexid,
    dh_boolean         meta_data_only,
    void               *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN tableid

The id of the table for the index that is being dropped.

IN indexid

The id of the index that is being dropped.

IN meta_data_only

A flag that indicates the SQL engine is only deleting metadata from the system catalog tables for the specified index. The SQL engine sets this flag to TRUE when the DROP INDEX statement specified 'METADATA_ONLY' in the STORAGE_ATTRIBUTES clause. Unless they support dynamic metadata (see section 3.3.7), implementations use this mechanism to unload metadata for indexes that have been deleted in the underlying storage system through means other than the Dharma SDK. If implementations do support dynamic metadata, they should ignore calls that set the *meta_data_only* flag.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

dhcs_drop_index is called as a direct result of the drop index statement. Tableid and indexid taken together serve to identify the index to be dropped. When the index is dropped, the SQL engine removes all knowledge of the index from the catalog tables. By calling *dhcs_drop_index*, the SQL engine is informing the storage system that the index is no longer needed, and that it effectively may be destroyed.

5.3.3 dhcs_ix_close

Closes an index after updating.

Syntax

```
extern dhcs_status_t
dhcs_ix_close (
    void     *ix_hdl,
    void     *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN ix_hdl

A handle for the index, as returned by *dhcs_ix_open.*

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

Closes an index within a storage manager.

5.3.4 dhcs_ix_delete

Deletes a record from an index.

Syntax

```
extern dhcs_status_t
dhcs_ix_delete (
    void                *ix_hdl,
    dhcs_fldl_val_t     *index_values,
    void                *tid,
    void                *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN ix_hdl

A handle for the index, as returned by *dhcs_ix_open.*

IN index_values

The list of index key component values for the record that is to be deleted from the index. A value exists for each component in the index.

If any of the values are for columns defined with LONG VARCHAR or LONG VARBINARY data types, then the field values for those columns do not contain actual data. Instead, the *data_t* component of *field_values* contains a field handle that identifies storage for the data.

IN tid

The tid of the record within the table associated with this index that the index key component values correspond to.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

dhcs_ix_delete is used by the SQL engine to delete an index record from an index. *index_values* contains the list of index key component values (or, for long data types, field handles) for the record to be deleted. Tid identifies the record within the table associated with this index that the index key component values correspond to. The index key component values and the tid values taken together form an index record.

Within *index_values* there is one component value for each index key component that makes up the index. Each component value is represented as a field item. The field items are ordered within *index_values* by their component id.

The record to be deleted from the index is the one whose component key values match the ones provided in *index_values*, and whose tid value matches the value provided by tid.

Before calling *dhcs_ix_delete*, the SQL engine will call *dhcs_rss_get_info* with the DHCS_IX_UPD_REQUIRED flag. If TRUE is returned, then the SQL engine will execute the *ix_delete*.

If FALSE is returned, then the SQL engine will not call *dhcs_ix_delete*. Instead it will assume that the storage system will update the corresponding indexes during the execution of the *dhcs_tpl_delete* function.

5.3.5 dhcs_ix_insert

Inserts a record into an index.

Syntax

```
extern dhcs_status_t
dhcs_ix_insert (
    void                *ix_hdl,
    dhcs_fldl_val_t     *index_values,
    void                *tid,
    void                *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN ix_hdl

A handle for the index, as returned by *dhcs_ix_open.*

IN index_values

The list of index key component values for the record that is to be inserted into the index. A value exists for each component in the index.

If any of the values are for columns defined with LONG VARCHAR or LONG VARBINARY data types, then the field values for those columns do not contain actual data. Instead, the *data_t* component of *field_values* contains a field handle that identifies storage for the data.

IN tid

The tuple identifier of the table record for which this index entry is being inserted.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the

dhcs_rss_init routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

dhcs_ix_insert is used by the SQL engine to insert an index record into an index. *index_values* contains the list of values (or, for long data types, field handles), one for each component of the index.

Tid identifies the record within the table associated with this index that the index key component values correspond to. The index key component values and the tid values taken together form an index record.

When inserting the index record into the index, the record must logically be stored according to the criteria that was established when the index was created. If duplicate records are not allowed, the storage system must compare the key component values of the index record to the key component values of records already contained within the index. If a record exists with the same values, then the storage system should return an error.

Note In the case where the storage system determines a duplicate record exists, the storage system is also responsible for removing the table record already inserted during execution of the *dhcs_tpl_insert* routine. The SQL engine does not call *dhcs_tpl_delete* to enforce the constraint against duplicate records. The storage system should remove the table record during its processing of the *dhcs_abort_trans* routine.

Within *index_values* there is one component value for each index key component that makes up the index. Each component value is represented as a field item. The field items are ordered within *index_values* by their component id.

The details of how an index record is stored within an index is storage manager specific, but it must be stored in such a way that the index component key values, along with the associated tid, can be retrieved as a unit via the *dhcs_ix_scan_fetch* function.

Before calling *ix_insert*, the SQL engine will call *dhcs_rss_get_info* with the DHCS_IX_UPD_REQUIRED flag.

- If TRUE is returned, then the SQL engine will execute *dhcs_ix_insert* after it executes *dhcs_tpl_insert*.
- If FALSE is returned, then the SQL engine will not call *dhcs_ix_insert*. Instead it assumes that the storage system will update the corresponding indexes during the execution of insert and update functions.

5.3.6 dhcs_ix_open

Opens an index for updating.

Syntax

```
extern dhcs_status_t
dhcs_ix_open (
```

```
    dhcs_tableid_t     tableid,
    dhcs_indexid_t     indexid,
    char               ix_type,
    void               **ix_hdl,
    void               *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN tableid

The identifier for the table that corresponds to the index that is being opened.

IN indexid

The identifier for the index that is being opened.

IN ix_type

A flag that indicates the type of index. The SQL engine passes the same value here as it passed to the *dhcs_create_index* function for this index. See section 5.3.1 for details.

OUT ix_hdl

A handle for the index. The format of the handle is specific to the storage system. The SQL engine passes the handle in subsequent calls to *dhcs_ix_insert*, *dhcs_ix_delete*, and *dhcs_ix_close*.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

The SQL engine calls *dhcs_ix_open* to open an index for update operations. In response, the storage manager makes sure the index is open and supplies a handle that the SQL engine passes to subsequent index update routines.

The tableid and indexid arguments taken in combination identify the particular index to be opened. The SQL engine obtains indexid and tableid from the SYSINDEXES catalog table.

Although the SQL engine presumes that the index specified by indexid is open after calling *dhcs_tpl_open*, the storage manager should not automatically open files or load data structures each time the SQL engine calls this function. This is because previous SQL statements may have resulted in calls to functions that already opened the

index. Instead, the storage manager should use whatever file-caching mechanism exists in the underlying storage system to check if the index is already open, and open it only if necessary.

5.3.7 dhcs_ix_scan_close

Closes an index which was opened for scanning.

Syntax

```
extern dhcs_status_t
dhcs_ix_scan_close (
    void     *ix_scan_hdl,
    void     *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN ix_scan_hdl

A handle for the index scan, as returned by *dhcs_ix_scan_open*.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

Closes an index that was opened for scanning within a storage manager.

5.3.8 dhcs_ix_scan_fetch

Fetches the next record in an index scan.

Syntax

```
extern dhcs_status_t
dhcs_ix_scan_fetch (
    void           *ix_scan_hdl,
    dhcs_ix_oper_t    operator,
    dhcs_fldl_val_t   *index_search_vals,
    dhcs_fldl_val_t   *field_values,
    void              *tid,
```

```
    void              *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.
SQL_NOT_FOUND	When no more records exist.

Arguments

IN ix_scan_hdl

A handle for the index scan, as returned by *dhcs_ix_scan_open*.

IN operator

Indicates the type of scan to perform.

The SQL engine supplies the same value here as on the corresponding call to *dhcs_ix_scan_open*. It is up to the storage manager to decide whether to process the operator value during execution of *dhcs_ix_scan_open* or *dhcs_ix_scan_fetch*. See Table 5-2: on page 5-37 for a list of the valid operators and their meanings.

IN index_search_vals

The list of values to use for comparison when searching for an index record. The SQL engine supplies the same values here as on the corresponding call to *dhcs_ix_scan_open*. (The search values for CONTAINS predicates are a special case. See the CONTAINS notes on page 5–51 for more detail.)

INOUT field_values

A field value list in which the storage system returns field values fetched for the index record that meets the criteria specified by operator and *index_search_vals*.

If any of the values are for columns defined with LONG VARCHAR or LONG VARBINARY data types, then the field values for those columns do not contain actual data. For such columns, on output, the storage manager supplies a field handle that identifies storage for the data.

INOUT tid

A pointer to a location to store the tid for the record that was fetched.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

ix_scan_fetch fetches the next record from an index based on the operator and comparison values stored in *index_search_vals*.

When an index scan is opened, the scan is positioned before the first record of the index that matches the comparison values based on the operator. With each call to *dhcs_ix_scan_fetch*, the storage manager:

- Returns values to non-null members of the *field_values* list
- Returns the tid for the record, if its input value is not null
- Moves the scan to the next record of the index that matches the comparison criteria

field_values is a pointer to a list of field items. Each field item identifies a field within the retrieved index record whose value is to be returned, and provides a location to store the field value (or, for long data types, field handles). If *field_values* is NULL, it indicates that no field values are to be returned for the index record. If *field_values* is non-NULL, then a value must be returned for each field for which a field item is specified.

The field items are ordered within the *field_values* structure by their index key id. The index key id identifies the index key field to be retrieved, and the *dhcs_data_t* structure provides a location for storing the retrieved value. Using the index field id, the storage system should extract the appropriate field value from the retrieved index record and store it in the *dhcs_data_t* structure.

The SQL engine may set the index key id to SQL_INVAL_FLDID rather than to a valid index key id. This means the storage system indicated it supports the fetch all fields feature by returning TRUE when the SQL engine called *dhcs_rss_get_info* with an *info_type* of DHCS_IX_FETCH_ALL_FIELDS. In that case, the field item represents a field which is not part of the index, but is a field within the table that the index being scanned is associated with. (See the following discussion.)

Tid provides a location to return the tid for the retrieved record. If tid is NULL, it indicates that the tid value for the record is not to be returned. If tid is non-NULL, then a value must be returned for the tid.

Fetching All Fields Through Index Scans: DHCS_IX_FETCH_ALL_FIELDS

A storage system typically returns a subset of the index key component fields and a tuple identifier (tid) in response to a *dhcs_ix_scan_fetch* call. If the SQL engine needs field values beyond those that make up the index key, then it specifies the appropriate tid when calling *dhcs_tpl_fetch* to get the remaining field values for the row.

However, many storage systems, hierarchical systems in particular, have direct access to all the field values of a row when performing a *dhcs_ix_scan_fetch* call. For cases where the SQL engine needs field values beyond the fields that make up the index key, a significant performance advantage is possible if all the field values that are needed are returned in response to a *dhcs_ix_scan_fetch* rather than just the index key fields. The performance gain occurs because the *dhcs_tpl_fetch* call is eliminated.

The SQL engine determines support for obtaining field values in this manner through the DHCS_IX_FETCH_ALL_FIELDS property. A storage system indicates support by returning TRUE for the DHCS_IX_FETCH_ALL_FIELDS info type of *dhcs_rss_get_info*.

The SQL engine identifies all the fields that it needs, whether they are index keys or not, in the *field_values* argument of the *dhcs_ix_scan_fetch* call. *field_values* is a structure of type *dhcs_fldl_val_t*, itself a list of structures of type *dhcs_fv_item_t* (see page 5-6). Each *dhcs_fv_item_t* structure represents a field value to be returned. The list contains two parts. The first part identifies index key fields (and their corresponding table fields) and the second part identifies the additional table fields that are not index key fields:

- In the first part, the *fv_field* element of the *dhcs_fv_item_t* structure contains the index key id. The *fv_tfield* element contains the table field id that corresponds to the index key id in *fv_field*.
- In the second part, the *fv_field* element is set to SQL_INVAL_FLDID to indicate there is no index key for this field. The *fv_tfield* element contains the table field id.

The index and table fields to be retrieved can thus be identified by comparing the *fv_field* element to SQL_INVAL_FLDID. Note that the second part of the list could be empty if the query refers only to the index key fields.

To process the *field_values* list, the stub implementation must loop through each element of the list:

- Use the *fv_field* value to identify desired index key fields and store their values in the *dhcs_data_t* structure.
- Use the *fv_tfield* value to identify desired table fields and store their values in the *dhcs_data_t* structure.

The following examples show how values in the data structures used by *dhcs_ix_scan_fetch* would appear after some specific SQL statements:

Example 5-4: Eliminating Tuple Scans Using *DHCS_IX_FETCH_ALL_FIELDS*

```
create table t1(c1 int, c2 int, c3 int, c4 int, c5 int, c6 int)
create index t1_idx on t1(c1, c2 , c3)
insert into t1 values(10, 20, 30, 40, 50, 60)
commit work
select * from t1 where c1 = 10

dhcs_ix_scan_fetch(
index_values :
    fv_field = 0, fv_tfield = 0, data = 10
field_values :
    fv_field = 0, fv_tfield = 0, data = 10
    fv_field = 1, fv_tfield = 1, data = 20
    fv_field = 2, fv_tfield = 2, data = 30
    fv_field = 65535, fv_tfield = 3, data = 40
    fv_field = 65535, fv_tfield = 4, data = 50
    fv_field = 65535, fv_tfield = 5, data = 60
ixoper = 0  (DHCS_IXOP_EQ)
   ...
)
```

```
C1          C2          C3          C4          C5          C6

--          --          --          --          --          --

10          20          30          40          50          60
1 record selected
select c1, c2, c5, c6 from t1 where c1 = 10

dhcs_ix_scan_fetch(
index_values :
    fv_field = 0, fv_tfield = 0,  data = 10
field_values :
    fv_field = 0, fv_tfield = 0,  data = 10
    fv_field = 1, fv_tfield = 1,  data = 20
    fv_field = 65535, fv_tfield = 4, data = 50
    fv_field = 65535, fv_tfield = 5, data = 60
ixoper = 0  (DHCS_IXOP_EQ)
  ...
)

C1          C2          C5          C6
--          --          --          --
10          20          50          60

1 record selected
select c2, c3 from t1

dhcs_ix_scan_fetch(
index_values :
    fv_field = 0, fv_tfield = 0, data = NULL
    fv_field = 1, fv_tfield = 1, data = NULL
    fv_field = 2, fv_tfield = 2, data = NULL
field_values :
    fv_field = 1, fv_tfield = 1, data = 20
    fv_field = 2, fv_tfield = 2, data = 30
ixoper = 6  (DHCS_IXOP_FIRST)
  ...
)


C2          C3
--          --
20          30

1 record selected
```

5.3.9 dhcs_ix_scan_open

Opens an index for scanning.

Syntax

```
extern dhcs_status_t
dhcs_ix_scan_open (
    dhcs_tableid_t          tableid,
    dhcs_indexid_t          indexid,
    char                    ix_type,
    dhcs_ix_oper_t          operator,
    short                   num_field_values,
    dhcs_fldl_val_t         *index_search_vals,
    short                   index_scan_hint,
    dhcs_tpl_fetch_hint_t   fetch_hint,
    void                    **ix_scan_hdl,
    void                    *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN tableid

The identifier for the table that corresponds to the index that is being opened.

IN indexid

The identifier for the index that is being opened.

IN ix_type

A flag that indicates the type of index. The SQL engine passes the same value here as it passed to the *dhcs_create_index* function for this index. See page 5-23 for details.

IN operator

A comparison operator that indicates the type of scan to perform. The operators specify a condition that is true or false about a given row or group of rows. They correspond to SQL predicates. The operator is one of the list returned by the storage manager in response to the DHCS_IX_PUSH_DOWN_RESTRICTS info type argument of *dhcs_rss_get_info*. Table 5–2 lists the possible values for the index operators. See *Index Operator Notes* on page 5-39 for more detail.

The SQL engine also passes the operator when it calls *dhcs_ix_scan_fetch*. The storage manager can process it during execution of either routine.

Table 5-2: Index Scan Comparison Operators

Operator	Type of Scan	Number of Comparison Values
DHCS_IXOP_EQ	Equal	One for each index component used
DHCS_IXOP_GT	Greater than	One for each index component used
DHCS_IXOP_GE	Greater than or equal	One for each index component used
DHCS_IXOP_LE	Less than or equal	One for each index component used
DHCS_IXOP_LT	Less than	One for each index component used
DHCS_IXOP_NE	Not equal	One for each index component used
DHCS_IXOP_BET	Inclusive between	Two for each index component used
DHCS_IXOP_BET_IE	Low-end inclusive between	Two for each index component used
DHCS_IXOP_BET_EI	High-end inclusive between	Two for each index component used
DHCS_IXOP_BET_EE	Exclusive between	Two for each index component used
DHCS_IXOP_NOTBET	Not between (inclusive)	Two for each index component used
DHCS_IXOP_FIRST	Start at first record	None
DHCS_IXOP_LAST	Return last record	None
DHCS_IXOP_IN	Equal to any of a list of one or more values	One for each index component used and each value in the list
DHCS_IXOP_NOTIN	Not equal to any of a list of one or more values	One for each index component used and each value in the list
DHCS_IXOP_CONTAINS	Storage-manager defined	One for each index component used
DHCS_IXOP_NOTCNTNS	Storage-manager defined	One for each index component used

IN num_field_values

The number of index components used in the predicate for which the scan is to return records. This varies from zero (for DHCS_IXOP_FIRST or DHCS_IXOP_LAST) up to the number of components in the index.

The implication of this number depends on the operator. For instance, a *num_field_values* of 3 means:

- For basic predicates (DHCS_IXOP_EQ, DHCS_IXOP_GT, DHCS_IXOP_GE, DHCS_IXOP_LE, DHCS_IXOP_LT, and DHCS_IXOP_NE), there are 3 values

in *index_search_vals*. A predicate for a DHCS_IXOP_EQ operator would be of the form:

A = index_search_val1 AND B = index_search_val2 AND C + = index_search_val3

- For between operators (DHCS_IXOP_BET, DHCS_IXOP_BET_IE, DHCS_IXOP_BET_EI, DHCS_IXOP_BET_EE, and DHCS_IXOP_NOTBET), there are 6 values in *index_search_vals*, and the predicate is of the form:

  A BETWEEN index_search_val1 AND index_search_val2 AND

  B BETWEEN index_search_val3 AND index_search_val4 AND

  C BETWEEN index_search_val5 AND index_search_val6

- For DHCS_IXOP_IN and DHCS_IXOP_NOTIN, that there are 3 sets of values (for these operators, *num_field_values* does not imply the number of values in the sets) and the predicate is of the form:

  A IN (index_search_val1 , index_search_val2 , ...) AND

  B IN (index_search_valx, ...) AND

  C IN (index_search_valy, ...)

- For DHCS_IXOP_CONTAINS and DHCS_IXOP_NOTCNTNS, there are 3 values in *index_search_vals*. A predicate for a DHCS_IXOP_CONTAINS operator would be of the form:

  A CONTAINS 'index_search_val1' AND

  B CONTAINS 'index_search_val2' AND

  C CONTAINS 'index_search_val3'

Although the number of fields represented in the predicate can be derived, *index_search_vals*, *num_field_values* supplies it directly.

IN index_search_vals

The list of values to use for comparison when searching for an index record. The SQL engine passes the same list when it calls *dhcs_ix_scan_fetch*. The storage manager can process the values during execution of either routine. (The search values for CONTAINS predicates are a special case.

IN index_scan_hint

Indicates if fixed length keys are used.

IN fetch_hint

Indicates whether the scan is being performed in the context of an update statement:

DHCS_TPL_FH_READ	The table is being scanned in the context of a read statement
DHCS_TPL_FH_WRITE	The table is being scanned and selected records may be updated

fetch_hint indicates that a selected index record may be updated via the *dhcs_ix_insert*, *dhcs_ix_delete*, *dhcs_tpl_update*, or the *dhcs_tpl_delete* functions. This flag may be used by certain storage managers whose concurrency control policy

(locking policy) needs to differentiate or wishes to differentiate between reading a record and reading a record for update.

OUT ix_scan_hdl

A handle for the index scan. The format of the handle is specific to the storage system. The SQL engine passes the handle in subsequent calls to *dhcs_ix_scan_fetch* and *dhcs_ix_scan_close*.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See page 5-87 for more detail.

Description

The SQL engine calls *dhcs_ix_scan_open* to open an index for scanning. In response, the storage manager makes sure the index is open and supplies a scan handle that the SQL engine passes to subsequent index scan routines.

The *tableid* and *indexid* arguments taken in combination identify the particular index to be opened. The SQL engine obtains *indexid* and *tableid* from the SYSINDEXES catalog table.

Although the SQL engine presumes that the index specified by indexid is open after calling *dhcs_tpl_scan_open*, the storage manager should not automatically open files or load data structures each time the SQL engine calls this function. This is because previous SQL statements may have resulted in calls to functions that already opened the index. Instead, the storage manager should use whatever file-caching mechanism exists in the underlying storage system to check if the index is already open, and open it only if necessary.

Index Operator Notes

The operator argument describes the type of index scan to perform by indicating the comparison criteria for selecting records from the index.

Implementations indicate support for various comparison operators by including them in the array of values they return in response to the DHCS_IX_PUSH_DOWN_RESTRICTS info type argument of *dhcs_rss_get_info*.

Implementations must at least support the DHCS_IXOP_FIRST operator. If the storage manager does not support a particular operator, the SQL engine processes such predicates internally (or, for DHCS_IXOP_CONTAINS and DHCS_IXOP_NOTCNTNS, generates an error). If the storage manager indicates it does not support processing of any but the DHCS_IXOP_FIRST index operator, the SQL engine requests that the storage manager return all records by passing the DHCS_IXOP_FIRST operator.

The SQL engine "pushes-down" processing of supported predicates to the storage manager. The objective of pushing down such index predicates is to reduce the over-

all cost of executing an SQL statement by allowing the SQL engine optimizer to consider options not otherwise available.

For operator values that supply comparison values, *index_search_vals* contains the values to be compared as well as their field ids and data types. Note that the values of operator and *index_search_vals* the SQL engine provides in *dhcs_ix_scan_open* are also provided on each call to *dhcs_ix_scan_fetch*. It is up to the storage manager to decide whether to process the operator value during execution of *dhcs_ix_scan_open* or *dhcs_ix_scan_fetch.*

The following discussion gives some more detail on the individual operators.

DHCS_IXOP_EQ, DHCS_IXOP_GT, DHCS_IXOP_GE, DHCS_IXOP_LE, DHCS_IXOP_LT, and DHCS_IXOP_NE

For these operators, the number of comparison values provided will be from one (1) up to the number of components in the index. All index records whose components values match the comparison values according to the operator that is provided should be returned via *ix_scan_fetch*.

DHCS_IXOP_BET, DHCS_IXOP_BET_IE, DHCS_IXOP_BET_EI, DHCS_IXOP_BET_EE, and DHCS_IXOP_NOTBET

For these operators, there are two comparison values for each index component. Each pair of comparison values indicates the upper and lower bounds of a range. So, the number of comparison values the SQL engine supplies in *index_search_vals* is twice the value passed in the *num_field_values* input argument.

When the SQL engine calls *dhcs_ix_scan_fetch* with one of the range operators, the storage manager should return all index records whose components meet the criteria detailed in the following table:

Table 5-3: BETWEEN Range Operators

Operator	Returns
DHCS_IXOP_BET	Records whose components are greater than or equal to the lower bound of the range, and less than or equal to the upper bound of the range.
DHCS_IXOP_BET_IE	Records whose components are greater than or equal to the lower bound of the range, and less than the upper bound of the range.
DHCS_IXOP_BET_EI	Records whose components are greater than the lower bound of the range, and less than or equal to the upper bound of the range.
DHCS_IXOP_BET_EE	Records whose components are greater than the lower bound of the range, and less than the upper bound of the range.
DHCS_IXOP_NOTBET	Records whose components are less than the lower bound of the range, and greater than the upper bound of the range.

DHCS_IXOP_FIRST and DHCS_IXOP_LAST

For operators DHCS_IXOP_FIRST and DHCS_IXOP_LAST, there are no comparison values:

- DHCS_IXOP_FIRST indicates that the index scan will start with the first record of the index. The storage system should iterate through all other records on successive calls to *dhcs_ix_scan_fetch*.
- DHCS_IXOP_LAST indicates that the index scan need only return the last record in the index. The storage system will not need to iterate through the index backwards.

Note that which record is first or last is dependent on the sort order of the fields within the index. See *dhcs_ix_insert* (section 5.3.5) for a more detailed description of how records are ordered within an index.

DHCS_IXOP_IN

For the DHCS_IXOP_IN operator, there is a set of comparison values for each index component. The storage manager must determine how many comparison values there are for each index component by examining the *index_search_vals* argument.

With DHCS_IXOP_IN, the storage manager should return all index records whose components have values in the cross-product of the sets of comparison values, as shown in the following table:

Table 5-4: Rows Returned for DHCS_IXOP_IN

Component 1 comparison values	Component 2 comparison values	Component 3 comparison values	Rows Returned
a, b	1, 2	x, y	a 1 x a 1 y a 2 x a 2 y b 1 x b 1 y b 2 x b 2 y

If a storage manager does not support DHCS_IXOP_IN (as indicated in the storage manager response to *dhcs_rss_get_info*), the SQL engine checks whether the storage manager supports DHCS_IXOP_EQ. If it does, the SQL engine translates an IN predicate to a series of calls to *dhcs_ix_scan_open* using DHCS_IXOP_EQ. If it does not, the SQL engine processes the predicate internally.

DHCS_IXOP_NOTIN

The DHCS_IXOP_NOTIN operator is similar to DHCS_IXOP_IN, with the following differences:

- With DHCS_IXOP_NOTIN, the storage manager should return all index records whose components do not have values in the cross product of the sets of comparison values.

- If a storage manager does not support DHCS_IXOP_NOTIN (as indicated in the storage manager response to *dhcs_rss_get_info*), the SQL engine processes the predicate internally, without first checking for support of DHCS_IXOP_NE.

DHCS_IXOP_CONTAINS and DHCS_IXOP_NOTCNTNS

The semantics of the these operators are defined by the storage manager, and indicates support for the SQL CONTAINS predicate. The SQL CONTAINS predicate is an extension that allows storage managers to provide implementation-defined search capabilities on character and binary data. The SQL syntax for a CONTAINS predicate is

column_name [NOT] CONTAINS 'string'

The SQL engine restricts the data type of *column_name* to CHARACTER, VARCHAR, LONG VARCHAR, BINARY, VARBINARY, or LONG VARBINARY. The format of the quoted string argument and the semantics of the CONTAINS predicate is determined by the storage manager. The following example shows one possible format for a CONTAINS predicate:

```
WHERE C1 CONTAINS 'ODBC , ORACLE +100'
```

With this format, the +100 syntax might indicate that the two search keywords must occur within 100 lines (or words, or pages) of each other.

The CONTAINS predicate is the only predicate that can operate on data in LONG data-type columns. The arbitrary size and unstructured format of data in such columns require special consideration. Note the following:

- Unlike the other comparison operators, the SQL engine will not process the DHCS_IXOP_CONTAINS and DHCS_IXOP_NOTCNTNS operators internally if the storage manager does not support them. Instead, the SQL engine generates an error when it encounters an SQL CONTAINS predicate.
- Indexes which support DHCS_IXOP_CONTAINS or DHCS_IXOP_NOTCNTNS can not support any other operators. The SQL engine generates an error if an index type which supports either DHCS_IXOP_CONTAINS or DHCS_IXOP_NOTCNTNS also supports other operators.
- Indexes which support DHCS_IXOP_CONTAINS and DHCS_IXOP_NOTCNTNS will perform better if they also support the DHCS_IX_TID_SORTED property.
- For LONG data-type columns, the SQL engine passes search values as a pointer to a character or binary string, not a field handle. The data type id of the character string is DHCS_LVC or DHCS_LVB. This is the only case where the SQL engine sets the data type id to a long type, but passes a value instead of a field handle to the *dhcs_data_t* component of a field value list.

For DHCS_IXOP_CONTAINS and DHCS_IXOP_NOTCNTNS, the number of comparison values provided will be from one (1) up to the number of components in the index.

5.4 LONG DATA TYPES INTERFACES

5.4.1 dhcs_get_data

Retrieves a segment of a long field value.

Syntax

```
extern dhcs_status_t
dhcs_get_data(
    dhcs_fld_hdl      * fld_hdl,
    char              * buf,
    long              buf_len,
    long              * len,
    long              offset,
    dh_boolean        * is_null,
    void              *conn_hdl
);
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN fld_hdl

The field handle for the long data-type field to be retrieved. Previous calls to the *dhcs_tpl_scan_fetch*, *dhcs_tpl_fetch*, or *dhcs_ix_scan_fetch* routines generate the field handles passed to *dhcs_get_data*. The field handle includes details on where the long data resides (such as a pointer to a file or disk location). However, specifics about the contents and structure of a field handle are defined by the storage manager.

OUT buf

The field segment retrieved.

IN buf_len

Buffer length for buf. *buf_len* specifies the maximum amount of data that can be retrieved in this call to *dhcs_get_data*.

OUT len

Length in bytes of data in the field, starting at offset. The total length of the data in the field is the sum of len plus offset.

IN offset

An offset, in bytes, that indicates where to start retrieval of the field segment. The total length of the data in the field is the sum of len plus offset.

OUT is_null

A flag indicating whether the field is null.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

The function *dhcs_get_data* retrieves a segment of a long field value. The SQL engine uses it to retrieve character, binary, or storage-system-specific data in columns defined as LONG VARCHAR or LONG VARBINARY.

ODBC applications that retrieve long field values specify the maximum size of data they can accept in a single call. The SQL engine passes this value to *dhcs_get_data* as *buf_len*. If the data in the long field is greater than *buf_len*, the SQL engine will call *dhcs_get_data* multiple times to retrieve the entire field value.

The storage manager retrieves data from the field starting at offset. The SQL engine initially sets offset to zero. On subsequent calls to *dhcs_get_data*, it increments offset:

- For LONG VARBINARY data, the SQL engine simply increments offset by *buf_len*.
- For LONG VARCHAR data, the storage manager must return the segment as a null-terminated string. The SQL engine takes this into account and increments offset by *buf_len -1*.

The storage manager indicates the length of data in the field through the len argument. It subtracts the value of offset from the total length of the data in the field and returns the resulting value in len. Thus, the sum of offset plus len is always the total length of the data in the field.

When the storage manager sets len to a value less than the *buf_len*, it signals that the current field segment is the last one. The SQL engine calls *dhcs_get_data* until the storage manager sets len to a value less than *buf_len*.

For instance, consider a long data value that is a total of 90 bytes long. Table 5–5 shows values for the various arguments to *dhcs_get_data* over a series of calls to retrieve the entire field for a 20-byte buffer length.

Table 5-5: Argument Values to *dhcs_get_data* Over a Series of Calls

		Binary		Character	
	buf_len	**offset**	**len**	**offset**	**len**
Call 1	20	0	90	0	90
Call 2	20	20	70	19	71
Call 3	20	40	50	38	52
Call 4	20	60	30	57	33
Call 5	20	80	10	76	14

5.4.2 dhcs_put_data

Stores a segment of a long field value.

Syntax

```
extern dhcs_status_t
dhcs_put_data(
    dhcs_fld_hdl      * fld_hdl,
    char              * buf,
    long              buf_len,
    long              offset,
    dh_boolean        * is_null,
    void              *conn_hdl
);
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN fld_hdl

The field handle for the long data-type field to be stored. A previous call to the *dhcs_tpl_insert* routine generated the field handle passed to *dhcs_put_data*. The field handle includes details on where to store the data (such as a pointer to a file or disk location). However, specifics about the contents and structure of a field handle are defined by the storage manager.

IN buf

The segment to be stored in the long field.

IN buf_len

Buffer length for buf. *buf_len* specifies the amount of data to be stored in this call to *dhcs_put_data*.

IN offset

An offset, in bytes, that indicates where to start storage of the field segment.

IN is_null

A flag indicating whether the field is null.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections.

Description

The function *dhcs_put_data* stores a segment of a long field value. The SQL engine uses it to store character, binary, or storage-system-specific data in columns defined as LONG VARCHAR or LONG VARBINARY.

ODBC applications that store long field values specify the maximum size of data they will pass in a single call. The SQL engine passes this value to *dhcs_put_data* as *buf_len*. The ODBC application specifies the offset at which to store the data, which is passed to *dhcs_put_data* as the offset argument. The ODBC application may store the data in multiple segments, in which case there will be multiple calls to *dhcs_put_data*.

5.4.3 dhcs_put_hdl

Copies data from one long-field handle to another.

Syntax

```
extern dhcs_status_t
dhcs_put_hdl(
    dhcs_fld_hdl   * dest_hdl,
    dhcs_fld_hdl   * src_hdl,
    void           *conn_hdl
);
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN dest_hdl

The destination field handle to copy data to.

IN src_hdl

The source field handle to retrieve data from.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

The function *dhcs_put_hdl* provides a shortcut for the SQL engine to process SQL INSERT statements that copy data in columns defined as LONG VARCHAR or LONG VARBINARY.

The SQL engine calls *dhcs_put_hdl* when it encounters an INSERT statement such as the following.

```
INSERT INTO T1 (C1) SELECT C2 FROM T2;
```

If C1 and C2 contain long data, the SQL engine calls *dhcs_put_hdl* instead of iterating through calls to *dhcs_get_data* and *dhcs_put_data*. To process such a case, the SQL engine makes the following calls:

- *dhcs_tpl_scan_fetch*, *dhcs_tpl_fetch*, or *dhcs_ix_scan_fetch*, which return the field handle the SQL engine passes as the *src_hdl* argument to *dhcs_put_hdl*. This field handle includes details on where the long data resides (such as a pointer to a file or disk location).
- *dhcs_tpl_insert*, which returns the field handle the SQL engine passes as the *dest_hdl* argument to *dhcs_put_hdl*. This field handle includes details on where to store the data.
- *dhcs_put_hdl* with arguments derived from the preceding calls.

The storage manager takes whatever steps are necessary to copy the data.

5.5 DYNAMIC METADATA INTERFACES

5.5.1 dhcs_get_colinfo

Retrieves detail on a table column from the storage manager. The SQL engine calls this routine only when storage managers indicate they support dynamic metadata (see section 5.5).

Syntax

```
extern dhcs_status_t
dhcs_get_colinfo (
    char            *table_name,
    char            *owner_name,
    dhcs_tableid_t  table_id,
    dhcs_colinfo_t  **info,
    long            * no_cols,
    void            *conn_hdl
);
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN table_name

A null-terminated character string that contains the table name for which the implementation should return column information. This value will be one of the values that the implementation supplied in the info.*table_name* output argument in response to a call to *dhcs_get_tblinfo.*

IN owner_name

A null-terminated character string that contains the owner of the table for which the implementation should return column information. This value will be one of the values that the implementation supplied in the info.*owner_name* output argument in response to a call to *dhcs_get_tblinfo*.

IN table_id

The identifier of the table for which the implementation should return column information. This value will be one of the values that the implementation supplied in the *info.id* output argument in response to a call to *dhcs_get_tblinfo*.

OUT info

An array of structures of type *dhcs_colinfo_t*. The SQL engine allocates and passes an array of 500 structures. The implementation supplies details for a single column in an element of the array.

The *dhcs_colinfo_t* structure definition and field descriptions are as follows:

```
typedef struct {
    dhcs_fld_desc_t     fld_info ;
    dhcs_dflt_type_t    dflt_type ;
    char                dflt_value[DHCS_MAX_DFLT_LEN_P1] ;
} dhcs_colinfo_t ;
```

fld_info	A structure of type *dhcs_desc_t* in which the implementation returns details of the column's definition. See page 5-2 for details of the *dhcs_desc_t* structure.
dflt_type	An enumerated type that represents different possible default values for the column. (A default value is the value that SQL stores in a column if an update operation for a row does not specify a value for the column.) Implementations either leave this field empty (in which case, the SQL engine uses a default value of NULL), or supply one of the following values: **DHCS_DFLT_LITERAL**: An integer, numeric or string constant. If *dflt_type* specifies DHCS_DFLT_LITERAL, the implementation should specify the string value in the *dflt_value* field of info. **DHCS_DFLT_USER**: The name of the user issuing the INSERT or UPDATE statement on the table. Valid only for columns defined with character data types. **DHCS_DFLT_NULL**: A null value. **DHCS_DFLT_UID**: The user id of the user executing the INSERT or UPDATE statement on the table. **DHCS_DFLT_SYSDATE**: The current date. Valid only for columns defined with DATE data types. **DHCS_DFLT_SYSTIME**: The current time. Valid only for columns defined with TIME data types. **DHCS_DFLT_SYSTIMESTAMP**: The current date and time. Valid only for columns defined with TIMESTAMP data types.
dflt_value	A null-terminated character string that contains a literal default value. Only applicable if *dflt_type* is set to DHCS_DFLT_LITERAL. The maximum length of the character string is 255 characters.

OUT no_cols

The number of columns in the table. This value indicates how many elements in the info array will be filled in.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

If a storage manager indicates it supports dynamic metadata, the SQL engine relies on the storage manager to provide details on the structure of tables and indexes, instead of storing those details in the static system catalog.

The SQL engine uses the information supplied through calls to *dhcs_get_colinfo* to load a memory-resident version of the syscolumns system catalog table. The SQL engine calls *dhcs_get_colinfo* when an SQL statement first accesses a particular table.

When it calls *dhcs_get_colinfo*, the SQL engine supplies input arguments that identify the table of interest. The combination of the *table_name* and *owner_name* arguments uniquely identifies a table in the storage system. Implementations can use that combination or the *table_id* argument to identify the table, whichever is more convenient. In response, the storage manager supplies details on the columns in the table in the info argument, up to the maximum of 500 columns in a table.

5.5.2 dhcs_get_idxinfo

Retrieves detail on an index from the proprietary storage system. The SQL engine calls this routine only when storage managers indicate they support dynamic metadata (see section 5.5).

Syntax

```
extern dhcs_status_t
dhcs_get_idxinfo (
    char            *table_name,
    char            *owner_name,
    dhcs_tableid_t  table_id,
    dhcs_idxinfo_t  *info,
    void            *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.
SQL_NOT_FOUND	After returning details on the last index for the table.

Arguments

IN table_name

A null-terminated character string that contains the table name for which the implementation should return index information. This value will be one of the values that the implementation supplied in the info.*table_name* output argument in response to a call to *dhcs_get_tblinfo*.

IN owner_name

A null-terminated character string that contains the owner of the table for which the implementation should return index information. This value will be one of the values that the implementation supplied in the info.*owner_name* output argument in response to a call to *dhcs_get_tblinfo*.

IN table_id

The identifier of the table for which the implementation should return index information. This value will be one of the values that the implementation supplied in the *info.id* output argument in response to a call to *dhcs_get_tblinfo*.

OUT info

A structure of type *dhcs_idxinfo_t*. When the SQL engine calls *dhcs_get_idxinfo*, the structure is empty. Implementations fill in the fields of the structure with details about the index. The *dhcs_idxinfo_t* structure definition and field descriptions are as follows:

```
typedef struct {
    dhcs_indexid_t        id;
    char                  index_name[DHCS_MAX_IDLEN_P1];
    char                  index_owner[DHCS_MAX_IDLEN_P1];
    dh_boolean            unique;
    char                  ix_type;
    int                   no_cols;
    dhcs_idxkey_info_t    idxkey_info[DHCS_MAX_IDXFIELDS ] ;
} dhcs_idxinfo_t;
```

id — A long integer identifier assigned by the storage system that uniquely identifies the index. The SQL engine passes *id* on subsequent calls to identify the index. The SQL engine reserves index identifiers below 1000 and above 32767. Implementations must generate index identifiers within those values. Implementations must keep track of index identifiers and their corresponding index names. The SQL engine passes only the identifier, not the name, in subsequent calls. It is the implementation's responsibility to associate the identifier with the correct index.

index_name — A null-terminated character string that contains the index name.

index_owner — A null-terminated character string that contains the owner of the index.

unique — A flag that indicates whether records in the index must be unique. If TRUE, the index is unique. If FALSE, then the index allows duplicate records.

ix_type — A single-character flag that indicates the type of index. The *ix_type* argument does not imply any particular indexing technique, but is an arbitrary flag that allows the storage manager to indicate differing support for multiple types of indexes. See the discussion of the *ix_type* argument of *dhcs_create_index* on page 5-23 for details.

no_cols — The number of index-key columns in the index. This value indicates how many elements in the *idxkey_info* array will be filled in.

idxkey_info An array of structures of type *dhcs_idxkey_info_t*. The SQL engine allocates and passes an array with DHCS_MAX_IDXFIELDS number of structures. The implementation supplies details for a single index key in an element of the array. The storage manager must supply information on the index keys in the same order as they exist in the proprietary storage system.

The *dhcs_idxkey_info_t* structure definition and field descriptions are as follows:

```
typedef struct {
char        sort_order  ;
char        col_name[DHCS_MAX_IDLEN_P1];
} dhcs_idxkey_info_t;
```

sort_order
A character that indicates the sort order for the index key. A value of A indicates ascending and a value of D indicates descending
order.col_name
The index key column name, supplied as null terminated character string.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

If a storage manager indicates it supports dynamic metadata, the SQL engine relies on the storage manager to provide details on the structure of tables and indexes, instead of storing those details in the static system catalog.

The SQL engine uses the information supplied through calls to *dhcs_get_idxinfo* to load a memory-resident version of the *sysindexes* system catalog table. The SQL engine calls *dhcs_get_idxinfo* when an SQL statement first accesses a particular table.

The SQL engine loops through calls to *dhcs_get_idxinfo* for each index defined on the table, and the storage manager supplies index details in the info argument. The storage manager indicates there are no more indexes for the table by returning SQL_NOT_FOUND.

When it calls *dhcs_get_idxinfo*, the SQL engine supplies input arguments that identify the table of interest. The combination of the *table_name* and *owner_name* arguments uniquely identifies a table in the storage system. Implementations can use that combination or the *table_id* argument to identify the table, whichever is more convenient.

5.5.3 dhcs_get_metainfo

Retrieves summary information about tables in the proprietary storage system. The SQL engine calls this routine only when storage managers indicate they support dynamic metadata (see section 5.5).

Syntax

```
extern dhcs_status_t
dhcs_get_metainfo (
    unsigned long     *num_tbl,
    dh_boolean        *tbl_sorted,
    void              *conn_hdl
);
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

INOUT num_tbl

The number of tables in the storage system for which subsequent calls to *dhcs_get_tblinfo* will return detail. This is the number of tables to which the user currently connected has access. The SQL engine initializes *num_tbl* to a default value and uses this default value if the storage manager does not change it in response to *dhcs_get_metainfo*.

OUT tbl_sorted

A Boolean value that indicates whether the implementation supplies responses to the SQL engine's calls to *dhcs_get_tblinfo* sorted by table name. A value of TRUE means the responses are sorted by table name.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

If a storage manager indicates it supports dynamic metadata, the SQL engine relies on the storage manager to provide details on the structure of tables and indexes, instead of storing those details in the static system catalog.

The SQL engine calls *dhcs_get_metainfo* when a user connects to the database, before it calls other dynamic metadata routines. In response, the storage manager supplies information that the SQL engine can use to improve performance of the other routines:

- The *num_tbl* argument specifies the number of tables for which subsequent calls to *dhcs_get_tblinfo* will return detail. The SQL engine uses this information to allocate memory for the memory-resident version of the systables system catalog table.

- If a storage manager does not supply a value for *num_tbl*, the SQL engine allocates memory for a default number of tables. If necessary, the SQL engine dynamically extends this default allocation during its calls to *dhcs_get_tblinfo*. This dynamic allocation slows performance of those calls, however.
- The *tbl_sorted* argument indicates whether the SQL engine must sort the storage manager's responses to *dhcs_get_tblinfo* before loading a memory-resident index for the systables system catalog table. The key for that index is the table name. If storage managers indicate that responses are sorted by table name, it loads the index without sorting, which improves overall performance of dynamic metadata loading.

Implementation of *dhcs_get_metainfo* is optional. If storage managers do not implement it, the SQL engine uses a default value for the number of tables, and assumes that responses to *dhcs_get_tblinfo* are not sorted.

5.5.4 dhcs_get_tblinfo

Retrieves detail on a table from the proprietary storage system. The SQL engine calls this routine only when storage managers indicate they support dynamic metadata (see section 5.5).

Syntax

```
extern dhcs_status_t
dhcs_get_tblinfo (
    dhcs_tblinfo_t*info,
    void*conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion
SQL_NOT_FOUND	After returning details on the last table in the storage system that is accessible by the currently-connected user.

Arguments

OUT info

A structure of type *dhcs_tblinfo_t*. When the SQL engine calls *dhcs_get_tblinfo*, the structure is empty. Implementations fill in the fields of the structure with details about the table. The *dhcs_tblinfo_t* structure definition and field descriptions are as follows:

```
typedef struct {
    dhcs_tableid_t       id;
    char                 table_name[DHCS_MAX_IDLEN_P1];
    char                 table_owner[DHCS_MAX_IDLEN_P1];
    dh_boolean           read_only ;
```

```
} dhcs_tblinfo_t;
```

id	A long integer identifier assigned by the storage system that uniquely identifies the table. The SQL engine passes id on subsequent calls to identify the index. The SQL engine reserves table identifiers below 1000 and above 32767. Implementations must generate table identifiers within those values. Implementations must keep track of table identifiers and their corresponding table names. The SQL engine passes only the identifier, not the name, in subsequent calls. It is the implementation's responsibility to associate the identifier with the correct table.
table_name	A null-terminated character string that contains the table name.
table_owner	A null-terminated character string that contains the owner of the table.
read_only	A Boolean value that indicates whether the user connected to the database has read-only access to the table. A value of TRUE means the user has read-only access. If *read_only* is set to TRUE, the SQL engine will not allow the user to issue INSERT, UPDATE, and DELETE statements on the table.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

If a storage manager indicates it supports dynamic metadata, the SQL engine relies on the storage manager to provide details on the structure of tables and indexes, instead of storing those details in the static system catalog.

The SQL engine uses the information supplied through calls to *dhcs_get_tblinfo* to load a memory-resident version of the systables system catalog table.

The SQL engine calls *dhcs_get_tblinfo* when a user connects to the database. The SQL engine loops through calls to *dhcs_get_tblinfo* for each table in the database, and the storage manager supplies table details in the info argument. The storage manager indicates there are no more tables for which to supply detail by returning SQL_NOT_FOUND.

It is the responsibility of the implementation to determine which table are accessible by the user connected to the storage system, and to return metadata for those tables only.

The metadata for each table that the SQL engine retrieves through *dhcs_get_tblinfo* does not include detail on individual columns of the table. The SQL engine retrieves details on the columns later, through a call to *dhcs_get_colinfo*, if and when an SQL statement first refers to the table.

5.6 TUPLE IDENTIFIER INTERFACES

5.6.1 dhcs_alloc_tid

Allocates space for an empty tuple identifier for a storage manager and initializes the tuple identifier values.

Syntax

```
extern dhcs_status_t
dhcs_alloc_tid (
    void     **tid_hdl,
    void     *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

OUT tid_hdl

A handle for the tuple identifier (tid).

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

The function *dhcs_alloc_tid* allocates memory for a tid handle. The tid handle specifies a storage-manager-specific structure that contains the value or values that make up a tid. This interface is a utility function that the storage manager itself as well as the SQL engine calls routinely.

In addition to allocating memory for the tid handle, *dhcs_alloc_tid* must also initialize the allocated tid to a unique invalid value. This value should compare as equal to itself, but not equal to any valid tid.

For instance, the sample implementation provided with the Dharma SDK implements tids as the char data type; its implementation of *dhcs_alloc_tid* initializes the value of an allocated tid to 0xFF.

5.6.2 dhcs_assign_tid

Copies the value for a tuple identifier (tid).

Syntax

```
extern dhcs_status_t
dhcs_assign_tid (
    void     *from_tid,
    void     *to_tid,
    void     *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN from_tid

A tid whose value is to be copied.

OUT to_tid

The updated tid value.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

dhcs_assign_tid is the mechanism to copy tid values.

5.6.3 dhcs_char_to_tid

Converts a character string to a tid.

Syntax

```
extern dhcs_status_t
dhcs_char_to_tid (
    short    len,
    char     *in_buf,
    void     *tid,
    void     *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN len

Length of the input buffer.

IN in_buf

The character string to be converted to a tid. The maximum allowable size of *in_buf* is 255.

OUT tid

The resultant tid.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

The *dhcs_char_to_tid* routine converts a tid from its string form to its internal storage manager form. The tid is stored in *in_buf* as a NULL terminated string. The size of the string is indicated by len.

5.6.4 dhcs_compare_tid

Compares two tids and returns a value indicating equality or relative size.

Syntax

```
extern short
dhcs_compare_tid (
    void     *tid1,
    void     *tid2,
    void     *conn_hdl
) ;
```

Returns

short

1	tid1 is greater than tid2.
0	The tid values are equal
-1	tid1 is less than tid2.

Arguments

IN tid1

One of the tids to be compared.

IN tid2

The tid to be compared with tid1.

Description

dhcs_compare_tid compares two tid values for equality and relative size. Two tids are equal if they are both the NULL_TID for that storage manager, or if they both point to the same row of some table.

Note that since the format of a tid is specific to the particular storage manager, the mechanism by which the storage manager determines that the tids are equal is also specific to the storage manager.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

5.6.5 dhcs_free_tid

Frees the space for a tuple identifier (tid) that was created within a storage manager.

Syntax

```
extern dhcs_status_t
dhcs_free_tid (
    void     *tid,
    void     *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN tid_hdl

The handle for the tuple identifier (tid).

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK

Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

5.6.6 dhcs_tid_to_char

Converts a tid to a character string.

Syntax

```
extern dhcs_status_t
dhcs_tid_to_char (
    short    len,
    void     *tid,
    char     *out_buf,
    void     *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN len

Length of output buffer.

IN tid

Tuple id to convert.

OUT out_buf

The resultant character string. The maximum allowable size of *out_buf* is 255.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

The *dhcs_tid_to_char* routine converts a tid from its internal storage manager form to a string form. The buffer to store the string is indicated by *out_buf*. The size of the string is indicated by len. Note that since the format of a tid is specific to the particular storage manager, the character string format is also storage manager specific. The only requirement is that the character string format be such that it can be converted back to its internal form using the *dhcs_char_to_tid* routine.

5.7 TRANSACTION INTERFACES

5.7.1 dhcs_abort_trans

Aborts, or rolls back, a transaction.

Syntax

```
extern dhcs_status_t
dhcs_abort_trans (
    void    *conn_hdl
) ;
```

Returns

STATUS_OK	Successful completion.

Arguments

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

Terminates the current transaction begun by the last call to *dhcs_begin_trans*. The storage system must undo all changes made to tables and indexes during the transaction.

5.7.2 dhcs_begin_trans

Starts a transaction.

Syntax

```
extern dhcs_status_t
dhcs_begin_trans (
    void*conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

Begins a transaction. The transaction that is started is the current transaction within the storage environment. All operations that are executed once the transaction is begun, until either *dhcs_commit_trans* or *dhcs_abort_trans* is executed, are considered to be part of this current transaction. A storage system must take whatever actions are appropriate to ensure the transaction properties of atomicity, isolation, consistency, and durability. The SQL engine does not enforce these properties.

5.7.3 dhcs_commit_trans

Commits a transaction.

Syntax

```
extern dhcs_status_t
dhcs_commit_trans (
    void    *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.
DHCS_TRANSACTION_ROLLBACK	If the storage system has decided to rollback the transaction.

Arguments

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

Terminates the current transaction begun by the last call to *dhcs_begin_trans.*

The storage system must make permanent any changes to tables and indexes made during the transaction, and make the changes visible so that they may be accessed by current and subsequent transactions according to the concurrency control policies implemented by the storage manager.

5.8 MISCELLANEOUS FUNCTIONS

5.8.1 dhcs_get_error_mesg

Returns the error message for any error code generated by the storage manager.

Syntax

```
extern dhcs_status_t
dhcs_get_error_mesg (
    long              errcode,
    unsigned short    msgbuf_len,
    char              *msgbuf,
    void              *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.
-1	Otherwise

Arguments

IN errcode

The error code returned by the storage manager through *dhcs_status_t* during execution of a routine. The code must be between -1000 and -9999.

IN msgbuf_len

Length of the error message buffer.

OUT msgbuf

Pointer to the buffer containing the error message.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

The SQL engine calls *dhcs_get_error_mesg* when it receives an error code generated by the storage manager. The storage manager can return such error codes during execution of any routine, through *dhcs_status_t*.

dhcs_get_error_mesg provides a mechanism for the SQL engine to retrieve message text associated with the error code from the storage manager. The source file

$TPEROOT/odbcsdk/src/demo.c implements the *dhcs_get_error_mesg* routine. Implementations add #define directives to the file to associate a mnemonic string with an error return code. They also add entries to the *dhcs_error_table* structure that associate an actual error message with the mnemonic code. The following example shows excerpts from the sample implementation's version:

Example 5-5: Adding Error Messages

```
/*
 * DHCS error returns.  The range is between -1000 and -9999.
 */


#define         DHCS_ERR_NOTYET -1001L
#define         DH_DEMO_MAX_TABLES_EXCEEDED -1002L
#define         DH_DEMO_MAX_INDEXES_EXCEEDED -1003L
.
.
.
/*
 * Error table listing the error codes and the error messages.
 */


    static      dhcs_err_entry_t dhcs_error_table [] = {
{ DHCS_ERR_NOTYET, "Not yet implemented" },
{ DH_DEMO_MAX_TABLES_EXCEEDED, "Maximum number of tables
allowed exceeded" },
{ DH_DEMO_MAX_INDEXES_EXCEEDED, "Maximum number of indexes
allowed exceeded" },
.
.
.
```

5.8.2 dhcs_rss_cleanup

Syntax

```
extern dhcs_status_t
dhcs_rss_cleanup (
    void     *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

The *dhcs_rss_cleanup* routine is used to close a database and clean up the storage environment. The specific functions performed by this routine are implementation-dependent.

5.8.3 dhcs_rss_get_info

Returns details on how a storage manager supports various types of indexes.

Syntax

```
extern dhcs_status_t
dhcs_rss_get_info(
    dhcs_rss_info_t     info_type,
    void                *input_buffer,
    unsigned short      out_buffer_len,
    void                *out_buffer,
    unsigned short      *out_buffer_size,
    void                *conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN info_type

The type of information which is being requested. See the *Info Type Values* discussion on page 5-67 for details on valid *info_type* values.

IN input_buffer

A one-character flag that specifies the type of index the SQL engine is requesting information about. CREATE INDEX statements specify the index type in the optional TYPE argument, and the SQL engine calls *dhcs_rss_get_info* for details on the properties of each index type. (If the CREATE INDEX statement omits the TYPE argument, the SQL engine sets the index type to B.)

The index type does not imply any particular indexing technique. It is an arbitrary flag that allows the storage manager to indicate different properties for multiple types of indexes. The SQL engine calls *dhcs_rss_get_info* for each index type, and the storage manager can respond with different index properties for each type. (For instance, that different index types support different comparison operators.)

Note The SQL engine does not supply an index type when it calls *dhcs_rss_get_info* with the DHCS_IX_UPD_REQUIRED *info_type* value. In that case, *input_buffer* is null, and the SQL engine assumes that the response is true for all index types.

IN out_buffer_len

The length of the output buffer.

INOUT out_buffer

A buffer allocated by the SQL engine in which the storage manager is to return the requested information. Depending on the info type, the storage manager either returns a Boolean value or an array of unsigned bytes in *out_buffer*:

Boolean	A character. If set to 1 it indicates TRUE. If set to 0 it indicates FALSE. With the exception of DHCS_IX_PUSH_DOWN_RESTRICTS, the storage manager returns a Boolean value for all *info_type* values.
Array	For the DHCS_IX_PUSH_DOWN_RESTRICTS *info_type*, *out_buffer* is an array of unsigned bytes representing the operators for which the storage system supports push-down processing.

OUT out_buffer_size

The total number of bytes that are available to be returned by the storage manager. If *out_buffer_size* is less than *out_buffer_len*, then it indicates the number of bytes within *out_buffer* that were actually used. If *out_buffer_size* is greater than *out_buffer_len*, then *out_buffer* is assumed to be completely full, and *out_buffer_size* indicates the actual number of bytes that would have been returned had *out_buffer* been large enough.

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

The SQL engine calls *dhcs_rss_get_info* to get details on the properties of different types of indexes supported by a storage manager. The *info_type* argument specifies the property of interest, and the *input_buffer* argument specifies the index type.

The output buffer is allocated by the SQL engine. Its size is indicated by *out_buffer_len*. For most *info_type* values, the output buffer is a Boolean that indicates support or lack of support for the particular property specified by *info_type*. The

storage manager indicates the number of bytes available to be returned by setting *out_buffer_size*.

Info Type Values

DHCS_IX_ALL_COMPONENTS

Description: Specific to indexes that include multiple table columns (multiple-component indexes). When performing an index scan, whether search values must be provided for all components. If TRUE is returned in response to DHCS_IX_ALL_COMPONENTS, then the storage manager is indicating that when the SQL engine is performing an index scan, within the *index_search_vals* list, a comparison value must be provided for all components of the index.

Input Parameter: *index_type*

Output Type: Boolean

DHCS_IX_COMPUTE_AGGR

Description: Whether the storage manager supports the SQL MIN and MAX aggregate functions. (In other words, if the storage manager returns TRUE to DHCS_IX_SORT_ORDER, and returns DHCS_IXOP_FIRST, and DHCS_IXOP_LAST in response to DHCS_IX_PUSH_DOWN_RESTRICTS, it should also return TRUE to DHCS_IX_COMPUTE_AGGR.)

Input Parameter: *index_type*

Output Type: Boolean

DHCS_IX_FETCH_ALL_FIELDS

Description: If TRUE is returned in response to DHCS_IX_FETCH_ALL_FIELDS, then the storage manager is indicating that in response to a call to *dhcs_ix_scan_fetch*, the storage system is able to return all of the fields of the record, and not just the index component fields. The SQL engine takes advantage of this property to avoid *tpl_fetch* calls.

Input Parameter: *index_type*

Output Type: Boolean

DHCS_IX_PUSH_DOWN_RESTRICTS

Description: Comparison operators which the storage manager can process during index scans for the specified type of index. The return value is an array of unsigned bytes indicating which operators are supported by the storage manager. The SQL engine will only push down processing of operators that are contained within the list that the storage manager returns. The SQL engine uses this list as the basis for the operator input argument to *dhcs_ix_scan_fetch* (page 5-31) and *dhcs_ix_scan_open*. See Table 5-2: on page 5-37 for a list of the valid values. If the storage manager indicates it does not support processing of an index comparison operator, the SQL engine processes the operator internally.

Input Parameter: *index_type*

Output Type: Array of unsigned bytes

DHCS_IX_SCAN_ALLOWED

Description: Whether indexes of the specified type support index scans. Storage managers return FALSE for indexes that are inherently non-scan-oriented, such as hash indexes.

Input Parameter: *index_type*

Output Type: Boolean

DHCS_IX_SORT_ORDER

Description: Whether indexes of the specified type are sorted. In other words, whether a scan on the index returns records in the order of the index key.

Input Parameter: *index_type*

Output Type: Boolean

DHCS_IX_TID_SORTED

Description: Whether indexes of the specified type return records sorted by tuple identifier. Ordinarily, indexes only guarantee to return records that meet the provided comparison criteria and the records are not sorted by tuple identifier. However, if the storage manager sets DHCS_IX_TID_SORTED to TRUE, the SQL engine can significantly optimize processing of compound predicates that specify multiple indexes on the same table (including specifying the same index multiple times).

For instance, the following search condition benefits from returning records that are sorted by tuple identifier:

```
WHERE C1 CONTAINS 'ODBC' AND C1 CONTAINS 'SQL'
```

In this case, the SQL engine first retrieves the tuple identifiers returned by the first predicate, then retrieves the tuple identifiers returned by the second predicate, and performs an intersect operation on the two sets. This intersect operation is much more efficient if the SQL engine can assume the sets are returned in tuple identifier order.

Typically, to support DHCS_IX_TID_SORTED, storage managers need to perform special processing at run time. Or, if a table is loaded with rows in index key order (resulting in the index key and tuple identifier sort order being the same), storage managers can support DHCS_IX_TID_SORTED for that index key.

Input Parameter: *index_type*

Output Type: Boolean

DHCS_IX_UPD_REQUIRED

Description:	Whether the SQL engine must update indexes after an insert, update, or delete operation on a table. If the storage manager sets DHCS_IX_UPD_REQUIRED to TRUE, it indicates that the SQL engine must directly manage the updating of indexes in addition to tables. When an SQL INSERT, UPDATE, or DELETE statement is executed on some table, in addition to calling *dhcs_tpl_insert*, *dhcs_tpl_update*, or *dhcs_tpl_delete* on the table, the SQL engine will execute *dhcs_ix_insert*, or *dhcs_ix_delete* on the corresponding indexes. If FALSE is returned, then the SQL engine will assume that the index will be updated indirectly by the storage manager as a side effect of the execution of *dhcs_tpl_insert*, *dhcs_tpl_update*, or *dhcs_tpl_delete*.
Input Parameter:	None
Output Type:	Boolean

5.8.4 dhcs_rss_init

Opens a database and initializes the storage environment for a user.

Syntax

```
extern dhcs_status_t
dhcs_rss_init (
    const char     *database,
    const char     *userid,
    const char     *passwd
    void           **conn_hdl
) ;
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN database

The name of the database to be opened. This name is created by the *mdcreate* utility (see Appendix A) and used thereafter to refer to a particular proprietary storage system.

IN userid

The name of the user as provided on the connect call.

IN passwd

The password of the user as provided on the connect call.

OUT conn_hdl

An implementation-specific handle that identifies the user connection.

The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration. In that environment, *conn_hdl* provides a mechanism for a storage manager to identify multiple user connections. (In the Client/Server configuration of the Dharma SDK, each connection creates a separate instance of the SQL engine by spawning a separate process. This mechanism is not available in the Desktop Configuration, where all connections go through a single DLL.)

In the Desktop configuration, storage managers can optionally supply a value in *conn_hdl* when the SQL engine calls *dhcs_rss_init*. The data type of the argument and details of how the storage manager uses it to distinguish between different user connections is up to the storage manager. The SQL engine passes any value supplied in response to *dhcs_rss_init* as the *conn_hdl* input argument to all subsequent storage interface calls for the duration of the connection. If the storage manager does not return a value in response to *dhcs_rss_init*, the SQL engine passes a null pointer on subsequent calls.

Description

The *dhcs_rss_init* routine is used to initialize a connection by opening a database and initializing the storage manager environment. *dhcs_rss_init* is only called when the SQL engine starts, and it is the only function called at startup.

Implementations must perform whatever specific functions are required to initialize a connection to the proprietary storage system.

Note that the database name, user name, and password arguments are opaque strings to the SQL engine. No attempt is made by the SQL engine to verify the format or validity of any of these strings. The storage environment should authenticate the database name, user name, and password according to the specific requirements of the storage environment.

5.8.5 dhcs_rss_initcall

Indicates to the storage manager that a new SQL request is to be executed.

Syntax

```
extern dhcs_status_t
dhcs_rss_initcall(
        void * conn_hdl
);
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN conn_hdl

An implementation-specific handle that identifies the user connection. The SQL engine supplies the same value here as the storage manager supplied in response to the *dhcs_rss_init* routine. The *conn_hdl* argument is relevant only in the Dharma SDK Desktop configuration, where storage managers use it to distinguish between multiple user connections. See section 5.8.4 for more detail.

Description

Indicates to the storage manager that a new SQL request is about to be executed. The function performed by this routine is implementation-specific based on the requirements of the storage environment.

5.9 UTILITY FUNCTIONS

Unlike the storage interface functions, the following utility functions are already implemented. Storage managers call the functions for data conversion and comparison.

5.9.1 dhcs_compare_data

Compares two values of the same data type and generates a value indicating equality or relative size.

Syntax

```
extern dhcs_status_t dhcs_compare_data (
    int     data_type,
    int     len1,
    void    *ptr1,
    int     len2,
    void    *ptr2,
    short   *result
);
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN data_type
The data type of the values to be compared. Note that *dhcs_compare_data* does not support the long data types DHCS_LVC and DHCS_LVB.

len1
Length of the input buffer for the first value.

ptr1
Pointer to the input buffer for the first value.

len2
Length of the input buffer for the second value.

ptr2
Pointer to the input buffer for the second value.

OUT result
The result of the comparison:

1	Value 1 is greater than value 2.
0	The values are equal.
-1	Value 1 is less than value 2.

5.9.2 dhcs_conv_data

Converts data from one host type to another.

Syntax

```
extern dhcs_status_t
dhcs_conv_data (
long     input_type,
long     input_len,
void     *input_ptr,
long     output_type,
long     output_len,
void     *output_ptr
);
```

Returns

dhcs_status_t

STATUS_OK	Successful completion.

Arguments

IN input_type

Input data type to be converted. Table 5–6 lists valid values. Note that *dhcs_conv_data* does not support the long data types DHCS_LVC and DHCS_LVB.

Table 5-6: Type Names for Data Type Conversion

DHCS_BIGINT	DHCS_BINARY	DHCS_BIT
DHCS_CHAR	DHCS_DATE	DHCS_DOUBLE
DHCS_INTEGER	DHCS_MONEY	DHCS_NUMERIC
DHCS_REAL	DHCS_SMALLFLOAT	DHCS_SMALLINT
DHCS_TIME	DHCS_TIMESTAMP	DHCS_TINYINT

IN input_len

Length of input buffer containing the data to be converted.

IN input_ptr

Pointer to the input buffer containing the data to be converted.

IN output_type

Desired data type to convert the input data to. Table 5–6 lists valid values.

IN output_len

Length of the output buffer to contain the converted data.

OUT output_ptr

Pointer to the output buffer that contains the converted data.

Chapter 6

Java Stubs Storage Interface Reference

6.1 COMMON CLASSES

The Dharma SDK provides a number of classes that are used as common arguments across several different storage interfaces. Table 6–1 lists the common classes, and the following sections describe them in more detail.

Table 6-1: Common classes

Class	Purpose
DharmaRecord	Holds a record.
RecordID	A unique identifier for a record within a table.
DharmaArray	A container class extended by TableFields, IndexFields and FieldValues.
FieldValue	Holds the data value of a field.
FieldValues	Holds an array of elements of FieldValue.
TableField	Holds information describing a field in a table.
TableFields	Holds an array of elements of TableField
IndexField	Holds information describing a field in an index.
IndexFields	Holds an array of elements of IndexField.

6.1.1 DharmaRecord

This class holds a record. It is used by TableHandle.getRecord(), TableScanHandle.getNextRecord() and IndexScanHandle.getNextRecord() to return a record from the storage system to the SQL Engine.

Definition

```
public class DharmaRecord
```

Members

RecordID recordID

Record id for this record.

boolean isNull[]

An array of flags to indicate whether a column is holding a null value.

Object record[]

An array of Objects to hold the field values.

Methods

6.1.1.1 DharmaRecord

Constructs a DharmaRecord.

```
public
DharmaRecord(int fieldCount) throws DharmaStorageException;
```

Returns

A DharmaRecord object. Throws DharmaStorageException if there is an error.

Arguments

IN int fieldCount

A count of the number of fields in the record.

Description

Creates a *DharmaRecord* of the given size. *DharmaRecord*s are used to return rows from table or index operations.

6.1.1.2 setFieldValue

Sets the value for a field in a DharmaRecord.

```
public void
setFieldValue (int fieldIndex,
            Object fieldValue) throws DharmaStorageException;
```

Returns

None. Throws DharmaStorageException if there is an error.

Arguments

IN int fieldIndex

Index of the field in the record. Indexes start at 0.

IN Object fieldValue

Data for the field.

Description

Stores the Object representing the field value in the corresponding entry in the *DharmaRecord*.

6.1.1.3 getFieldValue

Gets the value for a field from a DharmaRecord.

```
public Object
getFieldValue (int fieldIndex) throws DharmaStorageException;
```

Returns

Value of the field as an Object. Throws DharmaStorageException if there is an error.

Arguments

IN int fieldIndex

Index of the field in the record. Indexes start at 0.

Description

Gets the Object representing a field value from the corresponding entry in the *DharmaRecord*.

6.1.1.4 setNull

Marks a field in a DharmaRecord as NULL.

```
public void
setNull (int fieldIndex) throws DharmaStorageException;
```

Returns

None. Throws DharmaStorageException if there is an error.

Arguments

IN int fieldIndex

Index of the field in the record. Indexes start at 0.

Description

Marks the corresponding field in the *DharmaRecord* as having a NULL value.

6.1.1.5 isNull

Returns whether or not a field in a DharmaRecord is NULL.

```
public boolean
isNull (int fieldIndex) throws DharmaStorageException;
```

Returns

Boolean, true if the field is NULL, false otherwise. Throws DharmaStorageException if there is an error.

Arguments

IN int fieldIndex

Index of the field in the record. Indexes start at 0.

Description

Returns a boolean value indicating whether the corresponding field in the *DharmaRecord* has a NULL value. A return value of TRUE indicates that the field is NULL.

6.1.1.6 setRecordID

Sets the record ID that the DharmaRecord corresponds to.

```
public void
setRecordID (RecordID rId) throws DharmaStorageException
```

Returns

None. Throws DharmaStorageException if there is an error.

Arguments

IN RecordID rID

The Record ID to associate the DharmaRecord with.

Description

Sets the *RecordID* member of the *DharmaRecord*. The *RecordID* provides a unique identifier of the row in the associated table that the *DharmaRecord* contains values for.

6.1.1.7 getRecordID

Returns the record ID that the DharmaRecord corresponds to.

```
public RecordID
getRecordID () throws DharmaStorageException;
```

Returns

The Record ID the DharmaRecord is associated with. Throws DharmaStorageException if there is an erdror.

Arguments

None

Description

Gets the *RecordID* member of the *DharmaRecord*. The *RecordID* provides a unique identifier of the row in the associated table that the *DharmaRecord* contains values for.

6.1.2 RecordID

A RecordID is a unique identifier for a record within a table. The implementation provided by Dharma uses a long integer value to uniquely identify a record. If the storage system needs to represent records using a more sophisticated mechanism, the RecordID class implementation provided by Dharma can be modified as required.

Definition

```
public class RecordID
```

Members

long recordID

Record Identifier is represented as a long number.

boolean isSet

Flag denotes if the RecordID has been set.

Methods

6.1.2.1 RecordID

Constructs a RecordID.

```
public
RecordID (long rID) throws DharmaStorageException;
```

Returns

A RecordID object. Throws DharmaStorageException if there is an error.

Arguments

IN long rID

A long value representing the Record ID.

Description

Constructor for a *RecordID*. The *RecordID* provides a unique identifier of a row in a table. By default the *RecordID* class uses a long to store the identifier. If the storage system needs to represent records using a more sophisticated mechanism, the RecordID class implementation provided by Dharma can be modified as required.

6.1.2.2 RecordID

Default constructor.

```
public
RecordID () throws DharmaStorageException;
```

Returns

A RecordID object. Throws DharmaStorageException if there is an error.

Arguments

None

Description

Default constructor for a *RecordID*.

6.1.2.3 setRecordID

Sets the RecordID.

```
public void
setRecordID (RecordID rID) throws DharmaStorageException
```

Returns

None. Throws DharmaStorageException if there is an error.

Arguments

IN RecordID rID

The *RecordID* to set the value to.

Description

Sets the *RecordID* value.

6.1.2.4 setRecordID

Sets the RecordID.

```
public void
setRecordID (long l) throws DharmaStorageException;
```

Returns

None. Throws DharmaStorageException if there is an error.

Arguments

IN long l

A long value representing the Record ID.

Description

Sets the *RecordID* value. By default the *RecordID* class uses a long to store the identifier. If the storage system needs to represent records using a more sophisticated mechanism, the RecordID class implementation provided by Dharma can be modified as required and this method would be changed to reflect the new internal representation. This method is used only internally within the RecordID class.

6.1.2.5 setRecordID

Sets the RecordID.

```
public void
setRecordID (String s) throws DharmaStorageException
```

Returns

None. Throws DharmaStorageException if there is an error.

Arguments

IN String s

A String representing the Record ID.

Description

Sets the *RecordID* value.

6.1.2.6 getString

Returns a string representation of the RecordID.

```
public String
getString() throws DharmaStorageException;
```

Returns

A String value representing the Record ID.. Throws DharmaStorageException if there is an error.

Arguments

None

Description

The RecordID.getString method converts a RecordId from its internal storage manager form to a string form. Note that because the format of the RecordID is specific to the particular storage manager, the character string format is also storage manager specific. The only requirement is that the character string format be such that it can be converted back to its internal form using the RecordID.setRecordID(String s) method. Being able to convert the RecordID to a string representation allows client tools to retrieve the RecordID and use it in queries.

6.1.2.7 getLong

Returns a long representation of the RecordID.

```
public long
getLong () throws DharmaStorageException;
```

Returns

A long value representing the Record ID. Throws DharmaStorageException if there is an error.

Arguments

None

Description

The RecordID.getLong method converts a RecordID from its internal storage manager form to a long form. By default the RecordID's internal representation is a long. If the storage system needs to represent records using a more sophisticated mechanism, then the RecordID class can be modified appropriately and this method would be changed to reflect the new internal representation. This method is used only internally within the RecordID class.

6.1.2.8 compareRecordID

Compares RecordIDs.

```
public int
compareRecordID (RecordID cRecordID) throws
                DharmaStorageException;
```

Returns

An int value representing the result of the comparison.

- Returns 0 if this RecordID and cRecordID are equal.
- Returns 1 if this RecordID is greater than cRecordID.
- Returns -1 if this RecordID is less than cRecordID.

Arguments

IN RecordID cRecordId

RecordID to compare this RecordID with.

Description

The RecordID.compareRecordID method compares two RecordID values for equality and relative size. Two RecordIDs are equal if they are both the null RecordID (isRecordIDSet returns FALSE) for that storage manager, or if they both point to the same row of some table. Note that since the format of a RecordID is specific to the particular storage manager, the mechanism by which the storage manager determines that the RecordIDs are equal is also specific to the storage manager.

6.1.2.9 isRecordIDSet

Returns a boolean value indicating if the RecordID has been set.

```
public boolean
isRecordIDSet () throws DharmaStorageException;
```

Returns

A boolean value indicating if the RecordID has been set. A value of TRUE indicates that the value has been set.

Arguments

None

Description

This method indicates if the RecordID contains a valid RecordID value or not.

6.1.3 DharmaArray

This is a container class. Classes TableFields, IndexFields and FieldValues extend this class to store TableField, IndexField and FieldValue. This class uses an array to hold the Objects.

Definition

```
public class DharmaArray
```

Members

int size

Size of array.

Object dataArray[]

Array of objects.

Methods

6.1.3.1 DharmaArray

Constructs a DharmaArray.

```
public
DharmaArray (int sz, Object dataarray[])
             throws DharmaStorageException
```

Returns

A DharmaArray object. Throws DharmaStorageException if there is an error.

Arguments

IN int sz

Size of array to create

IN Object dataarray[]

Array of Objects to store in DharmaArray

Description

This constructs a *DharmaArray* of the specified size and places the Objects passed in in the *dataarray* argument into the *DharmaArray.*

6.1.3.2 getNthElement

Gets the nth element from the array. Index starts from 0.

```
public Object
getNthElement (int index) throws DharmaStorageException;
```

Returns

Object representing the nth element in the DharmaArray. Throws DharmaStorage-Exception if there is an error.

Arguments

IN int index

Index of Object in array to return.

Description

This method returns Nth Object in the *DharmaArray* where N is specified by the *index* argument

6.1.3.3 getSize

Gets the size of the array. Index starts from 0.

```
public int
getSize () throws DharmaStorageException;
```

Returns

Int representing the size of the DharmaArray. Throws DharmaStorageException if there is an error.

Arguments

None

Description

This method returns the size of the *DharmaArray*

6.1.4 FieldValue

The FieldValue class represents a field in an index or table.

Definition

```
public class FieldValue
```

Members

short m_fieldID;

Contains the table field identifier (for Table operationss) or index key identifier (for Index operations)

short m_tableFieldID;

This field is used only during index scan operations. It contains the table field identifier for the field value needed to satisfy a particular query. For table fields that are also index keys this field contains the table field identifier that corresponds to the index key identifier in the index. If a query requires a field value that is not an index key, the SQL engine sets fieldID to StorageCodes.INVAL_FLDID to indicate there is no index key that corresponds to the field

If the storage system returned a value of TRUE when the SQL engine called GetStorageManagerInfo with an infoType of Storage-Codes.IX_FETCH_ALL_FIELDS, the storage system fetches values for all fields, not just those that are index component fields, when it processes index scans.

Object m_data

An Object containing the field data.

short m_typeID

The data type of the field.

boolean m_isNull

A Boolean value that specifies whether the column contains a null value. A value of TRUE indicates that the column is null.

short m_maxLength

An integer value that specifies:

- For fixed-length data types, the defined length

- For variable-length data types, the maximum length

short m_dataLength

The actual length of the data (for variable-length data types only).

short m_width

The maximum number of digits for numeric types.

short m_scale

The number of digits to the right of the decimal point for numeric types.

Methods

6.1.4.1 FieldValue

Constructs a *FieldValue*.

```
public
FieldValue (short fieldID, Object val,
            short type) throws DharmaStorageException;
```

Returns

A *FieldValue* object. Throws *DharmaStorageException* if there is an error.

Arguments

IN short fieldID

ID of the field.

IN Object val

Data for the field.

IN short type

Type of the field.

IN boolean isnull

Specifies whether column is null or not.

Description

This method constructs a *FieldValue* object.

6.1.4.2 FieldValue

Constructs a FieldValue.

```
public
FieldValue (short fieldID, Object val)
            throws DharmaStorageException;
```

Returns

A *FieldValue* object. Throws *DharmaStorageException* if there is an error.

Arguments

IN short fieldID

ID of the field.

IN Object val

Data for the field.

Description

This method constructs a *FieldValue* object.

6.1.4.3 FieldValue

Default constructor. Constructs a *FieldValue* Object.

```
public
FieldValue () throws DharmaStorageException;
```

Returns

A *FieldValue* object. Throws *DharmaStorageException* if there is an error.

Arguments

None

Description

This method constructs a FieldValue object.

6.1.4.4 getFieldID

Returns the field id.

```
public short
getFieldID() throws DharmaStorageException;
```

Returns

A short value representing the field id. Throws *DharmaStorageException* if there is an error.

Arguments

None

Description

This method returns the field id associated with the *FieldValue* object. This id identifies the field within the table or index that the *FieldValue* is associated with. When a storage manager returns TRUE for the *StorageManager.getStorageManagerInfo* method with the StorageCodes.IX_FETCH_ALL_FIELDS flag, if the field to be retrieved is not included among the index components, this value will be set to StorageCodes.INVAL_FLDID.

6.1.4.5 setFieldID

Sets the field id.

```
public void
setFieldID(short fieldID) throws DharmaStorageException;
```

Returns

None. Throws *DharmaStorageException* if there is an error.

Arguments

IN short fieldID

Value to set field id to.

Description

This method sets the field id associated with the *FieldValue* object. This id identifies the field within the table or index that the *FieldValue* is associated with.

6.1.4.6 getTableFieldID

Returns the table field id.

```
public short
getTableFieldID() throws DharmaStorageException;
```

Returns

A short value representing the table field id. Throws *DharmaStorageException* if there is an error.

Arguments

None

Description

This method gets the table field id associated with the *FieldValue* object. This id identifies the field within the table that corresponds to the index in use. This value is used only during *IndexScanHandle.getNextRecord*

6.1.4.7 setTableFieldID

Sets the table field id.

```
public void
setTableFieldID(short tableFieldID) throws
DharmaStorageException;
```

Returns

None. Throws *DharmaStorageException* if there is an error.

Arguments

IN short tableFieldID

Value to set table field id to.

Description

This method sets the table field id associated with the *FieldValue* object. This id identifies the field within the table that corresponds to the index in use. This value is used only during *IndexScanHandle.getNextRecord*

6.1.4.8 getMaxLength

Returns the maximum field length.

```
public short
getMaxLength() throws DharmaStorageException;
```

Returns

A short value representing the maximum field length. Throws *DharmaStorageException* if there is an error.

Arguments

None

Description

This method returns an integer value that specifies:

- For fixed-length data types, the defined length

- For variable-length data types, the maximum length

6.1.4.9 setMaxLength

Sets the maximum field length.

```
public void
setMaxLength(short maxLength) throws DharmaStorageException;
```

Returns

None. Throws *DharmaStorageException* if there is an error.

Arguments

IN short maxLength

Value to set the maximum length to.

Description

This method sets the maximum length of the field to an integer value that specifies:

- For fixed-length data types, the defined length

- For variable-length data types, the maximum length

6.1.4.10 getDataLength

Returns the data length.

```
public short
getDataLength() throws DharmaStorageException;
```

Returns

A short value representing the data length. Throws *DharmaStorageException* if there is an error.

Arguments

None

Description

This method gets the actual length of the data (for variable-length data types only).

6.1.4.11 setDataLength

Sets the data length.

```
public void
setDataLength(short dataLength) throws
DharmaStorageException
```

Returns

None. Throws *DharmaStorageException* if there is an error.

Arguments

IN short len

Description

This method sets the actual length of the data (for variable-length data types only).

6.1.4.12 getWidth

Returns the width.

```
public short
getWidth() throws DharmaStorageException
```

Returns

A short value representing the width. Throws *DharmaStorageException* if there is an error.

Arguments

None

Description

This method gets the width of the field which is the maximum number of digits for numeric types.

6.1.4.13 setWidth

Sets the width.

```
public void
setWidth(short width) throws DharmaStorageException
```

Returns

None. Throws *DharmaStorageException* if there is an error.

Arguments

IN short width

Value to set the width to.

Description

This method sets the width of the field which is the maximum number of digits for numeric types.

6.1.4.14 getScale

Returns the scale.

```
public short
getScale() throws DharmaStorageException
```

Returns

A short value representing the scale. Throws *DharmaStorageException* if there is an error.

Arguments

None

Description

This method gets the scale of the field which is the number of digits to the right of the decimal point for numeric types.

6.1.4.15 setScale

Sets the scale.

```
public void
setScale(short scale) throws DharmaStorageException
```

Returns

None. Throws *DharmaStorageException* if there is an error.

Arguments

IN short scale

Value to set the scale to.

Description

This method sets the scale of the field which is the number of digits to the right of the decimal point for numeric types.

6.1.4.16 getData

Returns the data as an Object.

```
public Object
getData() throws DharmaStorageException;
```

Returns

An Object representing the field data. Throws *DharmaStorageException* if there is an error.

Arguments

None

Description

This method gets the Object that contains the data of the field.

6.1.4.17 setData

Sets the data Object.

```
public void
setData(Object val) throws DharmaStorageException;
```

Returns

None. Throws *DharmaStorageException* if there is an error.

Arguments

IN Object val

Object to set the data to.

Description

This method sets the Object that contains the data of the field.

6.1.4.18 getTypeID

Returns the field data type.

```
public short
getTypeID() throws DharmaStorageException;
```

Returns

An short value representing the field data type. Throws *DharmaStorageException* if there is an error.

Arguments

None

Description

This method gets the SQL data type of the field.

6.1.4.19 setTypeID

Sets the field data type.

```
public void
setTypeID(short type) throws DharmaStorageException;
```

Returns

None. Throws *DharmaStorageException* if there is an error.

Arguments

IN short type

Value to set the type id to.

Description

This method sets the SQL data type of the field.

6.1.4.20 isNull

Returns a value indicating if the field data is NULL.

```
public boolean
isNull() throws DharmaStorageException;
```

Returns

An boolean value indicating if the field data is NULL. A value of TRUE indicates that the field data is NULL. Throws *DharmaStorageException* if there is an error.

Arguments

None

Description

This method returns a boolean value indicating if the data of the field is NULL. A value of TRUE indicates that the data is NULL.

6.1.4.21 setNull

Sets the field data value to NULL.

```
public void
setNull() throws DharmaStorageException;
```

Returns

None. Throws *DharmaStorageException* if there is an error.

Arguments

None

Description

This method sets a boolean value indicating if the data of the field is NULL. A value of TRUE indicates that the data is NULL.

6.1.5 FieldValues

This class holds an array of *FieldValue*. The SQL Engine passes a *FieldValues* object to the storage system when it supplies values to be inserted or updated or to specify the search criteria for retrieving records.

Definition

```
public class FieldValues extends DharmaArray
```

Members

None

Methods

6.1.5.1 FieldValues

Constructs a *FieldValues* array.

```
public
FieldValues (int fieldCount, Object fieldvalues[])
             throws DharmaStorageException
```

Returns

A *FieldValues* object. Throws *DharmaStorageException* if there is an error.

Arguments

IN int fieldCount

A count of the number of fields in the array

IN Object fieldValues

The array of field objects.

Description

This method constructs a *FieldValues* array of the specified size and loads the array with the Objects provided in the *fieldvalues* argument.

6.1.5.2 getNth

Gets the Nth element. Index starts from 0.

```
public FieldValue
getNth(int index) throws DharmaStorageException
```

Returns

A *FieldValue* object. Throws *DharmaStorageException* if there is an error.

Arguments

IN int index

The index of the fieldValue in the array.

Description

This method gets the Nth Object in the *FieldValues* array.

6.1.6 TableField

This class holds information describing a field in a table.

Definition

```
public class TableField
```

Members

short m_fieldID

The id of the table field.

short m_dataType

The data type of the field.

String m_fieldName

The name of the field.

boolean m_isNullable

Flag to indicate if the field can be NULL. A value of TRUE indicates the field can be NULL.

short m_maxLength

Specifies :

- For fixed-length data types, the fixed length.

- For variable-length data types, the maximum length.

short m_width

Specifies the maximum number of digits for numeric types.

short m_scale

Specifies the number of digits to the right of the decimal point for numeric types.

Methods

6.1.6.1 TableField

Constructs a *TableField*.

```
public
TableField(short fieldID, short type, String fieldName)
           throws DharmaStorageException
```

Returns

A TableField object. Throws *DharmaStorageException* if there is an error.

Arguments

IN short fieldID

Id of the field.

IN short type

Data type of the field.

IN String fieldName

Name of the field

Description

Constructs a *TableField* Object with the specified id, type and field name.

6.1.6.2 getFieldName

Returns the name of the field.

```
public String
getFieldName () throws DharmaStorageException
```

Returns

A String Object contaning the name of the field. Throws *DharmaStorageException* if there is an error.

Arguments

None

Description

Gets the name of the field as a String.

6.1.6.3 setFieldName

Sets the name of the field.

```
public void
setFieldName(String fieldName) throws DharmaStorageException
```

Returns

None, throws *DharmaStorageException* if there is an error.

Arguments

IN String fieldName

Name for the field.

Description

Sets the name of the field to the passed String.

6.1.6.4 getFieldID

Returns the id of the field.

```
public short
getFieldID () throws DharmaStorageException
```

Returns

A short value representing the field id.

Arguments

None

Description

Gets the field id.

6.1.6.5 setFieldID

Sets the id of the field.

```
public void
setFieldID(short fieldID) throws DharmaStorageException
```

Returns

None

Arguments

IN short fieldID

ID for the field.

Description

Sets the field id.

6.1.6.6 isNullable

Returns a boolean indicating if the field can contain a NULL value.

```
public boolean
isNullable () throws DharmaStorageException
```

Returns

A boolean indicating if the field can contain NULL values. TRUE indicates that NULL values are allowed.

Arguments

None

Description

Returns a boolean indicating if the field can contain a NULL value. A value of TRUE indicates that the field value can be NULL.

6.1.6.7 setNullable

Sets the field as allowed to contain NULL values.

```
public void
setNullable() throws DharmaStorageException
```

Returns

None

Arguments

None

Description

Sets a flag indicating that the field can contain a NULL value.

6.1.6.8 setNotNullable

Sets the field as not allowed to contain NULL values.

```
public void
setNotNullable() throws DharmaStorageException
```

Returns

None

Arguments

None

Description

Sets a flag indicating that the field can not contain a NULL value.

6.1.6.9 getMaxLength

Returns the maximum length of the field.

```
public short
getMaxLength () throws DharmaStorageException
```

Returns

A short value representing the length.

Arguments

None

Description

Returns a short value representing the maximum length of the field for variable length fields. For fixed length fields, returns the fixed length.

6.1.6.10 setMaxLength

Sets the maximum length of the field.

```
public void
setMaxLength(short maxLength) throws DharmaStorageException
```

Returns

None

Arguments

IN short maxLength

The maximum length of the field.

Description

Sets a short value representing the maximum length of the field for variable length fields. For fixed length fields the value represents the fixed length.

6.1.6.11 getWidth

Returns the width of the field.

```
public short
getWidth () throws DharmaStorageException
```

Returns

A short value representing the width.

Arguments

None

Description

Returns a short value representing the maximum number of digits for numeric types.

6.1.6.12 setWidth

Sets the maximum number of digits for numeric types.

```
public void
setWidth(short width) throws DharmaStorageException
```

Returns

None

Arguments

IN short wd

Width for the field.

Description

Sets a short value representing the maximum number of digits for numeric types.

6.1.6.13 getScale

Returns the scale of the field

```
public short
getScale () throws DharmaStorageException
```

Returns

A short value representing the scale.

Arguments

None

Description

Returns a short value representing the number of digits to the right of the decimal point for numeric types.

6.1.6.14 setScale

Sets the number of digits to the right of the decimal point for numeric type.

```
public void
setScale(short scale) throws DharmaStorageException
```

Returns

None

Arguments

IN short sc

Scale for the field.

Description

Sets a short value representing the number of digits to the right of the decimal point for numeric types.

6.1.7 TableFields

This class holds an array of TableField. The SQL Engine passes a TableField object to the storage system when it calls StorageManagerHandle.createTable() to create a table.

Definition

```
public class TableFields extends DharmaArray
```

Members

None

Methods

6.1.7.1 TableFields

Constructs a *TableFields* array.

```
public
TableFields (int fieldCount,Object tablefields[])
                throws DharmaStorageException
```

Returns

A *TableFields* object. Throws *DharmaStorageException* if there is an error.

Arguments

IN int fieldCount

A count of the number of fields in the array

IN Object tablefields

The array of *TableField* objects

Description

Constructs a *TableFields* array of size *fieldCount* and loads the array with the objects in the *tablefields* argument.

6.1.7.2 getNth

Returns the Nth element of the array as a *TableField* Object. Index starts from 0.

```
public TableField
getNth (int index) throws DharmaStorageException
```

Returns

A *TableField* object. Throws *DharmaStorageException* if there is an error.

Arguments

IN int index

The element of the array to return.

Description

Returns the Nth element of the *TableFields* array as a *TableField* object.

6.1.8 IndexField

This class holds information describing a field in an index.

Definition

```
public class IndexField
```

Members

short m_fieldID

The id of the index field.

short m_sortOrder

Ascending or descending index sort order.

short m_tableFieldID

The id of the corresponding field in the table.

short m_typeID

The data type of the field.

String m_fieldName

The name of the field.

Methods

6.1.8.1 IndexField

Constructs an *IndexField.*

```
IndexField (short fieldID, short typeID, char sortOrder,
            short tableFieldID, String fieldName)
            throws DharmaStorageException
```

Returns

A *IndexField* object. Throws *DharmaStorageException* if there is an error.

Arguments

IN short fieldID

Id of the field.

IN short typeID

Data type of the field.

IN char sortOrder

Ascending or descending index sort order

IN short tableFieldID

The id of the corresponding field in the table.

IN String fieldName

Name of the field.

Description

Constructs an *IndexField* object.

6.1.8.2 getFieldID

Returns the id of the field.

```
public short
getFieldID () throws DharmaStorageException
```

Returns

A short value representing the field id.

Arguments

None

Description

Returns a short value representing the id of the index field.

6.1.8.3 setFieldID

Sets the id of the field.

```
public void
setFieldID(short fieldID) throws DharmaStorageException
```

Returns

None

Arguments

IN short fieldID

ID for the field.

Description

Sets the field id.

6.1.8.4 getTypeID

Returns the data type of the field.

```
public short
getTypeID () throws DharmaStorageException
```

Returns

A short value representing the data type of the field.

Arguments

None

Description

Returns a short value representing the data type of the index field.

6.1.8.5 setTypeID

Sets the type id of the field.

```
public void
setTypeID(short typeID) throws DharmaStorageException
```

Returns

None

Arguments

IN short typeID

Data type for the field.

Description

Sets the data type of the index field.

6.1.8.6 getSortOrder

Returns the sort order of the field.

```
public int
getSortOrder () throws DharmaStorageException
```

Returns

An integer value representing the sort order of the field.

Arguments

None

Description

Returns a char value representing the sort order of the index field as specified in the index create statement. StorageCodes.ASCENDING signifies an ascending sort order, StorageCodes.DESCENDING signifies a descending sort order.

6.1.8.7 setSortOrder

Sets the sort order of the field.

```
public void
setShortOrder(int sortOrder) throws DharmaStorageException
```

Returns

None

Arguments

IN short sortOrder

Ascending or descending sort order for the field.

Description

Sets a char value representing the sort order of the index field as specified in the index create statement. StorageCodes.ASCENDING signifies an ascending sort order, StorageCodes.DESCENDING signifies a descending sort order.

6.1.8.8 getTableFieldID

Returns the id of the corresponding table field.

```
public short
getTableFieldID () throws DharmaStorageException
```

Returns

A short value representing the id of the corresponding table field.

Arguments

None

Description

Gets a short value representing the id of the field in the table corresponding to the index field.

6.1.8.9 setTableFieldID

Sets the id of the corresponding table field.

```
public void
setTableFieldID(short tableFieldID)
                throws DharmaStorageException
```

Returns

None

Arguments

IN short tableFieldID

Description

Sets a short value representing the id of the field in the table corresponding to the index field.

6.1.8.10 getFieldName

Returns the name field.

```
public String
getTableFieldID () throws DharmaStorageException
```

Returns

A String object containing the name of the field.

Arguments

None

Description

Gets a String value representing the name of the field in the index.

6.1.8.11 setFieldName

Sets the name of the field.

```
public void
setFieldName (String fieldName)
              throws DharmaStorageException
```

Returns

None

Arguments

IN String fieldName

Name of the field.

Description

Sets a String value representing the name of the field in the index.

6.1.9 IndexFields

This class holds an array of IndexField. The SQL Engine passes an IndexField object to the storage system when it calls StorageManagerHandle.createIndex() to create an index.

Definition

```
public class IndexFields extends DharmaArray
```

Members

None

Methods

6.1.9.1 IndexFields

Constructs an *IndexFields* array.

```
public
IndexFields (int fieldCount,Object indexfields[])
               throws DharmaStorageException
```

Returns

A *IndexFields* object. Throws *DharmaStorageException* if there is an error.

Arguments

IN int fieldCount

A count of the number of fields in the array

IN Object indexfields

The array of IndexField objects

Description

Constructs a *IndexFields* array of size *fieldCount* and loads the array with the Objects passed in the *indexfields* argument.

6.1.9.2 getNth

Returns the Nth element of the array as an *IndexField* Object. Index starts from 0.

```
public IndexField
getNth (int index) throws DharmaStorageException
```

Returns

An *IndexField* object. Throws *DharmaStorageException* if there is an error.

Arguments

IN int index

The element of the array to return.

Description

Gets a Nth element of the *IndexFields* array as specified by the *index* argument and returns it as an *IndexField* object.

6.2 TABLE INTERFACES

6.2.1 TableHandle

The *TableHandle* class is the equivalent of *tpl_hdl_t* in the C++ stubs. It is created using the *getTableHandle* method in the *StorageManagerHandle* class. The *TableHandle* class is used to manipulate a table. It provides the basic functionality of opening, and closing a table, as well as the basic operations of inserting a record, fetching a record, updating a record, and deleting a record.

Definition

```
public class TableHandle
```

Members

None

Methods

6.2.1.1 insert

Inserts a record into a table.

```
public RecordID
insert(FieldValues fieldValues) throws DharmaStorageException
```

Returns

The *RecordID* of the inserted record. Throws *DharmaStorageException* if there is an error.

Arguments

IN FieldValues fieldValues

The values for the fields in the record to be inserted

Description

TableHandle.insert is used to insert a record into a table. *fieldValues* contains the list of field values for the record to be inserted. There is one field value for each field that makes up the table. The fields are ordered in the list by their field id.

The storage system must assign a record identifier (*recordID*) to the record that is inserted. This *recordID* is returned from *TableHandle.insert*. On output, the *recordID* must contain a *recordID* value that can be used by the SQL engine to relocate the record that was inserted. The SQL engine may use the *recordID* on subsequent calls to other functions to identify the record that was inserted.

Note that the SQL engine imposes no requirement on a storage manager relative to the order of records within a table.

After calling *TableHandle.insert,* the SQL engine will call *StorageManagerHandle.getStorageManagerInfo* with the StorageCodes.IX_UPD_REQUIRED flag:

•If TRUE is returned, then the SQL engine will update any corresponding indexes appropriately by calling *IndexHandle.insert.*

•If FALSE is returned, then the SQL engine assumes that the storage system will update the corresponding indexes during the execution of *TableHandle.insert.* If inserting the record into the associated indexes would result in a duplicate index key value for a unique index, then the record should not be stored in the table or the index, and an error returned.

6.2.1.2 getRecord

Fetches a specified record from a table.

```
public DharmaRecord
getRecord (RecordID recordID, int fetchHint,
           FieldValues refFields
          ) throws DharmaStorageException
```

Returns

A *DharmaRecord* Object containing the retrieved record. Throws *DharmaStorageException* if there is an error.

Arguments

IN RecordID recordID

ID of the record to retrieve.

IN int fetchHint

Indicates if the record is being fetched in the context of a SQL statement which only performs read operations or if it is being executed in the context of a SQL statement that could perform writes. *fetchHint* will be one of the following values:

- StorageCodes.TPL_FH_READ: The record being fetched is not a candidate for being updated in the context of the current SQL statement.
- StorageCodes.TPL_FH_WRITE: The record being fetched is a candidate for being updated in the context of the current SQL statement.

Note that *fetchHint* is in fact just a hint. It is strictly relative to the current SQL statement. Even if *fetchHint* is set to StorageCodes.TPL_FH_READ, it does not imply that the record being fetched was not already updated earlier in the transaction, or that it will not be updated at some future point during the execution of the transaction.

IN FieldValues refFields

List of fields to be returned (only fields required by the query are fetched).

Description

TableHandle.getRecord fetches a record from a table. *recordID* identifies the record within the table that is to be fetched. *refFields* is a pointer to a list of fields. *refFields* may contain entries for all of the fields within the record, or it may contain entries for only a subset of fields.

Using *FieldValue.fieldID*, the storage system should extract the appropriate field value from the retrieved record and store it in the *DharmaRecord*.

6.2.1.3 update

Updates a record in a table.

```
public void
update (RecordID recordID, FieldValues fieldValues)
        throws DharmaStorageException
```

Returns

None

Arguments

IN RecordID recordID

The ID of the record to update.

IN FieldValues fieldValues

The values for the fields in the record to be updated

Description

TableHandle.update updates a record in a table. *recordID* identifies the record within the table that is to be updated. *fieldValues* contains a list of the fields to be updated and the new data for each field. Note that only fields to be updated are contained in the list.

For each field that is to be updated the value of that field is replaced by the value that was extracted from *fieldValue*s.

After calling *TableHandle.update*, the SQL engine will call *StorageManagerHandle.getStorageManagerInfo* with the StorageCodes.IX_UPD_REQUIRED flag:

•If TRUE is returned, then the SQL engine will update any corresponding indexes appropriately.

•If FALSE is returned, then the SQL engine assumes that the storage system will update the corresponding indexes during the execution of *TableHandle.update*.

6.2.1.4 delete

Deletes a record from a table.

public void

delete(RecordID recordID) throws DharmaStorageException

Returns

None

Arguments

IN RecordID recordID

The ID of the record to be deleted.

Description

TableHandle.delete deletes a record in a table. recordID identifies the record within the table that is to be deleted.

After calling TableHandle.delete, the SQL engine will call StorageManagerHandle.

getStorageManagerInfo with the StorageCodes.IX_UPD_REQUIRED flag:

•If TRUE is returned, then the SQL engine will update any corresponding indexes appropriately.

•If FALSE is returned, then the SQL engine assumes that the storage system will

update the corresponding indexes during the execution of TableHandle.delete.

6.2.1.5 close

Close the table handle.

```
public void
close ()  throws DharmaStorageException
```

Returns

None

Arguments

None

Description

Closes an index that was opened for scanning within a storage manager.

6.2.2 TableScanHandle

The *TableScanHandle* class is the equivalent of the *tpl_scan_t* class in C++. It is created using the *getTableScanHandle* method in the *StorageManagerHandle* class. The *TableScanHandle* class is used to scan the records that are contained within a specified table. It provides the basic operation of fetching all of the records from the table one record at a time and the basic functionality of closing a scan on a table.

Definition

```
public class TableScanHandle
```

Members

None

Methods

6.2.2.1 getNextRecord

Fetches the next record in a table scan.

```
public DharmaRecord
getNextRecord (FieldValues refFields)
               throws DharmaStorageException
```

Returns

DharmaRecord object containing the field values fetched from the table record that meets the criteria specified in *operator* and *searchValues*, and the *RecordID* for the corresponding table record.

Arguments

IN FieldValues refFields

A *FieldValues* object in which the SQL engine lists the fields for which values are to be fetched from the next table record.

Description

TableScanHandle.getNextRecord fetches the next record from a table scan. When a table scan is opened, the scan is positioned before the first record of the table. Each call to *TableScanHandle.getNextRecord*, results in the scan being moved to the next record of the table, and the field values from the record being returned. With each call to *TableScanHandle.getNextRecord*, the storage manager:

- Returns values to non-null members of the *refFields* array
- Returns the *RecordID* for the record
- Moves the scan to the next record of the table

refFields identifies fields within the retrieved table record whose values are to be returned. If *refFields* has a size of 0, it indicates that no field values are to be

returned for the table record (in which case it is only the *RecordID* that is required by the SQL engine. If *refFields* has a non-zero size, then a value must be returned for each field specified in *refFields*.

6.2.2.2 close

Closes TableScanHandle.

public void

```
close ()  throws DharmaStorageException
```

Returns

None

Arguments

None

Description

Closes an index that was opened for scanning within a storage manager.

6.3 INDEX INTERFACES

6.3.1 IndexHandle

The *IndexHandle* class is the equivalent of the *ix_hdl_t* class in the C++ stubs. It is created using the *getIndexHandle* method of the *StorageManagerHandle* class. This class is used to manipulate an index. It provides the functionality of opening and closing an index, as well as inserting, appending and deleting records in an index.

Definition

```
public class IndexHandle
```

Members

None

Methods

6.3.1.1 insert

Inserts a record into an index.

```
public void
insert (FieldValues indexValues, RecordID recordID)
        throws DharmaStorageException
```

Returns

None

Arguments

IN FieldValues indexValues

The list of index key component values for the record that is to be inserted into the index. A value exists for each component in the index.

IN RecordID recordID

The *RecordID* of the table record for which this index entry is being inserted.

Description

IndexHandle.insert is used by the SQL engine to insert an index record into an index. *indexValues* contains the list of values, one for each component of the index. *recordID* identifies the record within the table associated with this index that the index key component values correspond to. The index key component values and the *recordID* values taken together form an index record.

When inserting the index record into the index, the record must logically be stored according to the criteria that was established when the index was created. If duplicate records are not allowed, the storage system must compare the key component values of the index record to the key component values of records already contained within the index. If a record exists with the same values, then the storage system should return an error.

> **Note** In the case where the storage system determines a duplicate record exists, the storage system is also responsible for removing the table record already inserted during execution of the *TableHandle.insert* method. The SQL engine does not call *TableHandle.delete* to enforce the constraint against duplicate records. The storage system should remove the table record during its processing of the *StorageEnvironment.rollbackTransaction* method.

Within *indexValues* there is one component value for each index key component that makes up the index. The details of how an index record is stored within an index is storage manager specific, but it must be stored in such a way that the index component key values, along with the associated *RecordID*, can be retrieved as a unit via the *IndexScanHandle.getNextRecord* method.

Before calling *IndexHandle.insert*, the SQL engine will call *StorageManagerHandle.getStorageManagerInfo* with the StorageCodes.IX_UPD_REQUIRED flag. If TRUE is returned, then the SQL engine will execute *IndexHandle.insert*. If FALSE is returned, then the SQL engine will not call *IndexHandle.insert*. Instead it will assume that the storage system will update the corresponding indexes during the execution of the *TableHandle.insert* and *TableHandle.update* methods.

6.3.1.2 delete

Deletes a record from an index.

```
public void
delete (FieldValues  indexValues, RecordID recordID)
        throws DharmaStorageException
```

Returns

None

Arguments

IN FieldValues indexValues

The list of index key component values for the record that is to be deleted from the index. A value exists for each component in the index.

IN RecordID recordID

The *RecordID* of the table record for which this index entry is being deleted.

Description

IndexHandle.delete is used by the SQL engine to delete an index record from an index. *indexValues* contains the list of index key component values for the record to be deleted. *recordID* identifies the record within the table associated with this index that the index key component values correspond to. The index key component values and the tid values taken together form an index record. Within *indexValues* there is one component value for each index key component that makes up the index. The record to be deleted from the index is the one whose component key values match the ones provided in *indexValues*, and whose *recordID* value matches the value provided by *recordID*.

Before calling *IndexHandle.delete*, the SQL engine will call *StorageManagerHandle.getStorageManagerInfo* with the StorageCodes.IX_UPD_REQUIRED flag. If TRUE is returned, then the SQL engine will execute *IndexHandle.delete*. If FALSE is returned, then the SQL engine will not call *IndexHandle.delete*. Instead it will assume that the storage system will update the corresponding indexes during the execution of the *TableHandle.delete* function.

6.3.1.3 close

Closes an IndexHandle.

public void

```
close ()  throws DharmaStorageException
```

Returns

None

Arguments

None

Description

Closes an index within a storage manager.

6.3.2 IndexScanHandle

The *IndexScanHandle* class is the equivalent of *ix_scan_t* in the C++ stubs. It is created using the *getIndexScanHandle* method in the *StorageManagerHandle* class. The *IndexScanHandle* class is used to scan the records that are contained within a specified index. It provides the basic operation of fetching selected records from the index one record at a time and the basic functionality of closing a scan on an index.

Definition

```
public class IndexScanHandle
```

Members

None

Methods

6.3.2.1 getNextRecord

Fetches the next record in an index scan.

```
public DharmaRecord
getNextRecord (int  operator, FieldValues searchValues,
       FieldValues refFields) throws DharmaStorageException
```

Returns

DharmaRecord object containing the field values fetched from the index record that meets the criteria specified in *operator* and *searchValues*, and the *RecordID* for the corresponding table record.

Arguments

IN int operator

Indicates the type of scan to perform. The SQL engine supplies the same value here as on the corresponding call to *StorageManagerHandle.getIndexScanHandle.* It is up to the storage manager to decide whether to process the operator value during execution of *StorageManagerHandle.getIndexScanHandle* or *IndexScanHandle.getNextRecord*. See Table 5-2: on page 5-37 for a list of the valid operators and their meanings.

IN FieldValues searchValues

The list of values to use for comparison when searching for an index record. The SQL engine supplies the same values here as on the corresponding call to *StorageManagerHandle.getIndexScanHandle*.

IN FieldValues refFields

A *FieldValues* object in which the SQL engine lists the fields for which values are to be fetched from the index record that meets the criteria specified by *operator* and *SearchValues*.

Description

IndexScanHandle.getNextRecord fetches the next record from an index based on the operator and comparison values stored in *searchValues*. When an index scan is opened, the scan is positioned before the first record of the index that matches the comparison values based on the operator. With each call to *IndexScanHandle.getNextRecord*, the storage manager:

- Returns values to non-null members of the *refFields* array
- Returns the *RecordID* for the record

• Moves the scan to the next record of the index that matches the comparison criteria

refFields identifies fields within the retrieved index record whose values are to be returned. If *refFields* has a size of 0, it indicates that no field values are to be returned for the index record (in which case it is only the *RecordID* that is required by the SQL engine. If *refFields* has a non-zero size, then a value must be returned for each field specified in *refFields*.

The SQL engine may set *refFields.fieldID* to StorageCodes.INVAL_FLDID rather than to a valid index key id. This means the storage system indicated it supports the fetch all fields feature by returning TRUE when the SQL engine called *StorageManagerHandle.getStorageManagerInfo* with an info_type of StorageCodes.IX_FETCH_ALL_FIELDS. In that case, the *FieldValue* represents a field which is not part of the index, but is a field within the table that the index being scanned is associated with. (See the following discussion.)

Fetching All Fields Through Index Scans: StorageCodes.IX_FETCH_ALL_FIELDS

A storage system typically returns a subset of the index key component fields and a record identifier (*RecordID*) in response to a *IndexScanHandle.getNextRecord* call. If the SQL engine needs field values beyond those that make up the index key, then it specifies the appropriate *RecordID* when calling *TableHandle.getRecord* to get the remaining field values for the row. However, many storage systems, hierarchical systems in particular, have direct access to all the field values of a row when performing an index fetch. For cases where the SQL engine needs field values beyond the fields that make up the index key, a significant performance advantage is possible if all the field values that are needed are returned in response to *IndexScanHandle.getNextRecord* rather than just the index key fields. The performance gain occurs because the *TableHandle.getRecord* call is eliminated.

The SQL engine determines support for obtaining field values in this manner through the StorageCodes.IX_FETCH_ALL_FIELDS property. A storage system indicates support for this method by returning TRUE for the StorageCodes.IX_FETCH_ALL_FIELDS info type from *StorageManagerHandle.getStorageManagerInfo*. The SQL engine identifies all the fields that it needs, whether they are index keys or not, in the refFields argument of the *IndexScanHandle.getNextRecord* call. The *refFields* array has two parts with the index fields listed first. For index fields, the *fieldID* member contains the index key identifier. The *tableFieldID* contains the field identifier for the field in the corresponding table. The second part of the array contains table fields that are not included in the index. For such fields the *fieldID* member is set to INVAL_FLDID, and the *tableFieldID* contains the field identifier for the field in the corresponding table.

The index and table fields to be retrieved can thus be identified by comparing the *tableFieldID* member to StorageCodes.INVAL_FLDID. Note that the second part of the array could be empty if the query refers only to the index key fields.

To process the *refFields* list, the stub implementation must loop through each ele-

ment of the array:

•Use the *fieldID* member to identify desired index key fields and store their values in the *DharmaRecord* or retrieve their values using the *tableFieldID*, whichever method is more efficient.
•Use the *tableFieldID* member to identify desired table fields and store their values in the *DharmaRecord*.

The following examples show how values in the data structures used by *IndexScanHandle.getNextRecord* would appear after some specific SQL statements:

Example 6-1: Eliminating Tuple Scans Using StorageCodes.IX_FETCH_ALL_FIELDS

```
create table t1(c1 int, c2 int, c3 int, c4 int, c5 int, c6 int)
create index t1_idx on t1(c1, c2 , c3)
insert into t1 values(10, 20, 30, 40, 50, 60)
commit work
select * from t1 where c1 = 10
IndexScanHandle.getNextRecord(
searchValues :
fieldID = 0, tableFieldID = 0, data = 10
refFields :
fieldID = 0, tableFieldID = 0, data = 10
fieldID = 1, tableFieldID = 1, data = 20
fieldID = 2, tableFieldID = 2, data = 30
fieldID = INVAL_FLDID, tableFieldID = 3, data = 40
fieldID = INVAL_FLDID, tableFieldID = 4, data = 50
fieldID = INVAL_FLDID, tableFieldID = 5, data = 60
operator = IXOP_EQ
...

C1 C2 C3 C4 C5 C6
-- -- -- -- -- --
10 20 30 40 50 60
1 record selected

select c1, c2, c5, c6 from t1 where c1 = 10
IndexScanHandle.getNextRecord(
searchValues :
fieldID = 0, tableFieldID = 0, data = 10
refFields :
fieldID = 0, tableFieldID = 0, data = 10
fieldID = 1, tableFieldID = 1, data = 20
```

```
fieldID = INVAL_FLDID, tableFieldID = 4, data = 50
fieldID = INVAL_FLDID, tableFieldID = 5, data = 60
operator = IXOP_EQ
...

C1 C2 C5 C6
-- -- -- --
10 20 50 60
1 record selected

select c2, c3 from t1
IndexScanHandle.getNextRecord(
searchValues :
fieldID = 0, tableFieldID = 0, data = NULL
fieldID = 1, tableFieldID = 1, data = NULL
fieldID = 2, tableFieldID = 2, data = NULL
refFields :
fieldID = 1, tableFieldID = 1, data = 20
fieldID = 2, tableFieldID = 2, data = 30
operator = IXOP_FIRST
...

C2 C3
-- --
20 30
1 record selected
```

6.3.2.2 close

Closes an IndexScanHandle.

public void

```
close ()  throws DharmaStorageException
```

Returns

None

Arguments

None

Description

Closes an index that was opened for scanning within a storage manager.

6.4 STORAGE SYSTEM INTERFACES

6.4.1 StorageEnvironment

Used to establish a connection with the storage environment.

Definition

```
public class StorageEnvironment
```

Members

None

Methods

6.4.1.1 createStorageEnvironment

This method is used by the SQL engine to get a handle to the storage environment.

```
public static StorageEnvironment
createStorageEnvironment(String databaseName,String userName,
            String password ) throws DharmaStorageException
```

Returns

StorageEnvironment object.

Arguments

IN String databaseName

The name of the database being opened.

IN String userName

The name of the user as provided on the connect call..

IN String password

The password as provided on the connect call..

Description

The *StorageEnvironment.createStorageEnvironment* method is used to initialize a connection by opening a database and initializing the storage manager environment. *StorageEnvironment.createStorageEnvironment* is only called when the SQL engine starts and it is the only function called at startup. Implementations must perform whatever specific functions are required to initialize a connection to the proprietary storage system. Note that the database name, user name, and password arguments are opaque strings to the SQL engine. No attempt is made by the SQL engine to verify the format or validity of any of these strings. The storage environment should authenticate the database name, user name, and password according to the specific requirements of the storage environment.

6.4.1.2 createStorageManagerHandle

This method is used by the SQL engine to get a handle to the storage environment.

```
public static StorageManagerHandle
createStorageEnvironment() throws DharmaStorageException
```

Returns

StorageManagerHandle object.

Arguments

None

Description

The *StorageEnvironment.createStorageManagerHandle* method is used to return a *StorageManager* object for use by the SQL engine.

6.4.1.3 beginTransaction

This method is used by the SQL engine to start a transaction.

```
public void
beginTransaction () throws DharmaStorageException
```

Returns

None.

Arguments

None

Description

The SQL engine calls StorageEnvironment.beginTransaction to begin a transaction. The transaction that is started is the current transaction within the storage environment. All operations that are executed once the transaction is begun, until either StorageEnvironment.commitTransaction or StorageEnvironment.rollbackTransaction is executed, are considered to be part of this current transaction. A storage system must take whatever actions are appropriate to ensure the transaction properties of atomicity, isolation, consistency, and durability. The SQL engine does not enforce these properties.

6.4.1.4 rollbackTransaction

This method is used by the SQL engine to rollback a transaction.

```
public void
rollbackTransaction () throws DharmaStorageException
```

Returns

None.

Arguments

None

Description

Terminates the current transaction begun by the last call to StorageEnvironment.beginTransaction. The storage system must undo all changes made to tables and indexes during the transaction.

6.4.1.5 commitTransaction

This method is used by the SQL engine to commit a transaction.

```
public void
rollbackTransaction () throws DharmaStorageException
```

Returns

None.

Arguments

None

Description

Terminates the current transaction begun by the last call to StorageEnvironment.beginTransaction. The storage system must make permanent any changes to tables and indexes made during the transaction, and make the changes visible so that they may be accessed by current and subsequent transactions according to the concurrency control policies implemented by the storage manager

6.4.1.6 close

Close the Storage environment handle.

```
public void
close ()  throws DharmaStorageException
```

Returns

None

Arguments

None

Description

Closes storage environment handle.

6.4.2 StorageManagerHandle

Represents a storage manager in a storage environment. Storage managers provide the base functionality for manipulating tables and indexes in storage systems. The StorageManagerHandle object is created using the createStorageManagerHandle() method of the StorageEnvironment class.

Definition

```
public class StorageManagerHandle
```

Members

None

Methods

6.4.2.1 createTable

This method is used by the SQL engine to create a table in the storage system.

```
public int
createTable (String tableName, boolean metadataOnly,
             TableFields tableFields,  TableFields primaryKeyL-
ist,
             String tableOwner) throws DharmaStorageException
```

Returns

An int value representing the table id for the table that was created or generated for an existing table.

The table id is a unique identifier that will be used on subsequent calls to identify the table. The SQL engine stores this id in the SYSTABLES catalog table along with the table name. The SQL engine reserves table identifiers below 1000 and above 32767. Implementations must generate table identifiers within those values. Implementations must keep track of table identifiers and their corresponding table names. The SQL engine passes only the identifier, not the name, in subsequent calls. It is the implementation's responsibility to associate the identifier with the correct table.

Arguments

IN String tableName

The name of the table that is being created. *tableName* will contain the name as specified in the CREATE TABLE statement. If the CREATE TABLE statement also specified 'METADATA_ONLY' in the STORAGE_ATTRIBUTES clause, *tableName* will contain the name of an existing table in the proprietary storage system.

IN boolean metadataOnly

A flag that indicates the SQL engine is inserting metadata into the system catalog tables for a table that already exists in the proprietary storage system. The storage manager should not create a new table, but instead return a table id for the SQL engine to associate with the existing table name.

The SQL engine sets this flag to TRUE when the CREATE TABLE statement specifies 'METADATA_ONLY' in the STORAGE_ATTRIBUTES clause. Implementations use this mechanism to load metadata for existing tables.

IN TableFields tableFields

A list of field descriptions for the columns in the table. Field definition information includes the field name, the field identifier, the field type, and a flag that indicates whether null values are allowed. Additionally, depending on the data type of the field, the length or the maximum length of the data type may be provided.

IN TableFields primaryKeyList

A list of primary key fields. *primaryKeyList* will be a subset of the fields specified in the *fldlList* argument. This list will be empty unless primary key fields were

specified with the CREATE TABLE statement. A primary key is characterized by the constraint that no two records in a table may have the same primary key value, and that no fields of the primary key may have a null value.

Storage systems that support primary keys can use this information to create the primary key for the table. Storage systems that do not support primary keys can ignore this information. In addition to passing down the primary key list, the SQL engine will automatically create a unique index on the primary key fields. Creating this index allows storage systems that do not directly support primary keys to support them indirectly via the index.

To create the primary-key index, the SQL engine calls *StorageManagerHandle.createIndex* with the isUnique argument set to TRUE, and the indexType argument set to B. The SQL engine generates a unique name for the index, prefixed with SYS_, and passes it as the indexName argument. The components of the index will be the fields that make up the primary key in the order that they appear in the table, and the sort order for each index component is ascending.

IN String tableOwner

A character string that specifies the user issuing the CREATE TABLE statement. Many implementations will ignore this argument.

Description

Adds a table to a storage manager, or generates an identifier for an existing table.

6.4.2.2 dropTable

This method is used by the SQL engine to drop a table in the storage system.

```
public void
dropTable (int tableID, boolean metadataOnly)
           throws DharmaStorageException
```

Returns

None

Arguments

IN int tableID

The id for the table that is being dropped.

IN boolean metadataOnly

A flag that indicates the SQL engine is only deleting metadata from the system catalog tables for the specified table. The SQL engine sets this flag to TRUE when the DROP TABLE statement specified 'METADATA_ONLY' in the STORAGE_ATTRIBUTES clause. Implementations use this mechanism to unload metadata for tables that have been deleted in the underlying storage system through means other than the Dharma SDK or to disallow access through SQL to a table that will remain in the storage system.

Description

StorageManagerHandle.dropTable is called as a direct result of the DROP TABLE statement. *tableID* serves to identify the table to be dropped. By calling *StorageManagerHandle.dropTable*, the SQL engine is informing the storage system that the

table is no longer needed, and that it effectively may be destroyed (or only the metaddata associated with it may be destroyed if the *metadataOnly* flag is TRUE).

6.4.2.3 createIndex

This method is used by the SQL engine to create an index in the storage system.

```
public int
createIndex (int tabIeID, boolean isUnique,
             boolean metadataOnly, char indexType,
             String indexName, IndexFields indexFields)
             throws DharmaStorageException
```

Returns

An int value representing the index id for the index that was created or generated for an existing index.

The index id is a unique identifier that will be used on subsequent calls to identify the index. The SQL engine stores this id in the SYSINDEXES catalog table along with the table name. The SQL engine reserves index identifiers below 1000 and above 32767. Implementations must generate index identifiers within those values. Implementations must keep track of index identifiers and their corresponding index names. The SQL engine passes only the identifier, not the name, in subsequent calls. It is the implementation's responsibility to associate the identifier with the correct index.

Arguments

IN int tableID

The table for which the index is being created.

IN boolean isUnique

A flag that indicates whether records in the index must be unique. If TRUE, the index is unique. If FALSE, then the index allows duplicate records.

IN boolean metadataOnly

A flag that indicates the SQL engine is inserting metadata into the system catalog tables for an index that already exists in the proprietary storage system. The storage manager should not create a new index, but instead return an index id for the SQL engine to associate with the existing index name.

The SQL engine sets this flag to TRUE when the CREATE INDEX statement specifies 'METADATA_ONLY' in the STORAGE_ATTRIBUTES clause. Implementations use this mechanism to load metadata for existing indexes.

IN char indexType

A flag that indicates the type of index. The SQL engine passes the TYPE argument specified in an application's SQL CREATE INDEX statement. If the CREATE INDEX statement did not include the TYPE argument, indexType is set to B. The *indexType* argument does not imply any particular indexing technique. It is an arbitrary flag that allows the storage manager to indicate differing support for multiple types of indexes. The SQL engine calls *StorageManagerHandle.getStorageManagerInfo* for each index type, and the storage manager can respond with different index properties for each type. (For instance, that different index types support dif-

ferent comparison operators.) The SQL engine also passes the *indexType* value when it opens the index through the *StorageManagerHandle.getIndexHandle* or *StorageManagerHandle.getIndexScanHandle* methods.

IN String indexName

The name of the index that is being created. *indexName* will contain the name as specified in the CREATE INDEX statement. If the CREATE INDEX statement also specified 'METADATA_ONLY' in the STORAGE_ATTRIBUTES clause, *indexName* will contain the name of an existing index in the proprietary storage system.

IN IndexFields indexFields

A description of the index key fields. Field information includes a key-field identifier, the corresponding field identifier in the table, data type, maximum length, and sort order.

Description

Creates an index for a table. tableID identifies the table for which the index is being created. indexFieldList provides the descriptive information that is needed to create the index, including the number of components in the index and the sort order for records in the index. This routine returns an indexid. The indexid is a number generated by the storage system that will be used on subsequent calls to identify the index. The SQL engine will store this id in the sysindexes catalog table along with index name.

6.4.2.4 dropIndex

This method is used by the SQL engine to drop an index in the storage system.

```
public void
dropIndex(int tableID, int indexID, boolean metadataOnly)
          throws DharmaStorageException
```

Returns

None

Arguments

IN int tableID

The id of the table for the index that is being dropped.

IN int indexID

The id of the index that is being dropped.

IN boolean metadataOnly

A flag that indicates the SQL engine is only deleting metadata from the system catalog tables for the specified index. The SQL engine sets this flag to TRUE when the DROP TABLE statement specified 'METADATA_ONLY' in the STORAGE_ATTRIBUTES clause. Implementations use this mechanism to unload metadata for indexes that have been deleted in the underlying storage system through means other than the Dharma SDK or to disallow access through SQL to a index that will remain in the storage system.

Description

StorageManagerHandle.dropIndex is called as a direct result of the DROP INDEX statement. *indexID* and *tableID* serve to identify the index to be dropped. By calling *StorageManagerHandle.dropIndex*, the SQL engine is informing the storage system that the index is no longer needed, and that it effectively may be destroyed (or only the metaddata associated with it may be destroyed if the *metadataOnly* flag is TRUE)

6.4.2.5 getTableHandle

Opens a table for non-scan operations.

```
public TableHandle
getTableHandle (int tableID) throws DharmaStorageException
```

Returns

TableHandle object for the specified table.

Arguments

IN int tableID

The id of the table that is being opened.

Description

The SQL engine calls *StorageManagerHandle.getTableHandle* to open a table for non-scan operations (See “TableHandle” on page 31.). In response, the storage manager makes sure the table is open and supplies a handle that the SQL engine uses to call subsequent methods. Although the SQL engine presumes that the table specified by *tableID* is open after calling *StorageManagerHandle.getTableHandle*, the storage manager should not automatically open files or load data structures each time the SQL engine calls this function. This is because previous SQL statements may have resulted in calls to functions that already opened the table. Instead, the storage manager should use whatever file-caching mechanism exists in the underlying storage system to check if the table is already open, and open it only if necessary.

6.4.2.6 getIndexHandle

Opens an index for non-scan operations.

```
public IndexHandle
getIndexHandle(int tableID, int indexID, char indexType)
                throws DharmaStorageException
```

Returns

TableHandle object for the specified table.

Arguments

IN int tableID

The id of the table for the index that is being opened.

IN int indexID

The id of the index that is being opened.

IN char indexType

A flag that indicates the type of index. The SQL engine passes the TYPE argument that was specified in an application's SQL CREATE INDEX statement. If the CREATE INDEX statement did not include the TYPE argument, indexType is set to B.The *indexType* argument does not imply any particular indexing technique. It is an arbitrary flag that allows the storage manager to indicate differing support for

multiple types of indexes. The SQL engine calls *StorageManagerHandle.getStorageManagerInfo* for each index type, and the storage manager can respond with different index properties for each type. (For instance, that different index types support different comparison operators.)

Description

The SQL engine calls *StorageManagerHandle.getIndexHandle* to open an index for non-scan operations (See "IndexHandle" on page 36.). In response, the storage manager makes sure the index is open and supplies a handle that the SQL engine uses to call subsequent methods. Although the SQL engine presumes that the index specified by *indexID* and *tableID* is open after calling *StorageManagerHandle.getIndexHandle*, the storage manager should not automatically open files or load data structures each time the SQL engine calls this function. This is because previous SQL statements may have resulted in calls to functions that already opened the index. Instead, the storage manager should use whatever file-caching mechanism exists in the underlying storage system to check if the index is already open, and open it only if necessary.

6.4.2.7 getTableScanHandle

Opens a table for scan operations.

```
public TableScanHandle
getTableScanHandle (int tableID, int fetchHint)
                    throws DharmaStorageException
```

Returns

TableScanHandle object for the specified table.

Arguments

IN int tableID

The id of the table that is being opened for scanning.

IN int fetchHint

Indicates if the record is being fetched in the context of a SQL statement which only performs read operations or if it is being executed in the context of a SQL statement that could perform writes. *fetchHint* will be one of the following values:

- StorageCodes.TPL_FH_READ: The record being fetched is not a candidate for being updated in the context of the current SQL statement.
- StorageCodes.TPL_FH_WRITE: The record being fetched is a candidate for being updated in the context of the current SQL statement.

Note that *fetchHint* is in fact just a hint. It is strictly relative to the current SQL statement. Even if *fetchHint* is set to StorageCodes.TPL_FH_READ, it does not imply that the record being fetched was not already updated earlier in the transaction, or that it will not be updated at some future point during the execution of the transaction.

Description

The SQL engine calls *StorageManagerHandle.getTableScanHandle* to open a table for scanning. In response, the storage manager makes sure the table is open and supplies a scan handle that the SQL engine uses to call subsequent methods. Although the SQL engine presumes that the table specified by *tableID* is open after calling *Stor-*

ageManagerHandle.getTableScanHandle, the storage manager should not automatically open files or load data structures each time the SQL engine calls this function. This is because previous SQL statements may have resulted in calls to functions that already opened the table. Instead, the storage manager should use whatever file-caching mechanism exists in the underlying storage system to check if the table is already open, and open it only if necessary.

6.4.2.8 getIndexScanHandle

Opens an index for scan operations.

```
public IndexScanHandle
getIndexScanHandle (int tableID,int indexID,char indexType,
            int oper, int numberOfFieldValues,
            FieldValues searchValue, int scanHint,
            int fetchHint) throws DharmaStorageException
```

Returns

IndexScanHandle object for the specified index.

Arguments

IN int tableID

The id of the table for the index that is being opened for scanning.

IN int indexID

The id of the index that is being opened.

IN char indexType

A flag that indicates the type of index. The SQL engine passes the TYPE argument that was specified in an application's SQL CREATE INDEX statement. If the CREATE INDEX statement did not include the TYPE argument, indexType is set to B. The *indexType* argument does not imply any particular indexing technique. It is an arbitrary flag that allows the storage manager to indicate differing support for multiple types of indexes. The SQL engine calls *StorageManagerHandle.getStorageManagerInfo* for each index type, and the storage manager can respond with different index properties for each type. (For instance, that different index types support different comparison operators.)

IN int oper

A comparison operator that indicates the type of scan to perform. The operators specify a condition that is true or false about a given row or group of rows. They correspond to SQL predicates. The operator is one of the list returned by the storage manager in response to the StorageCodes.IX_PUSH_DOWN_RESTRICTS info type argument of StorageManagerHandle.getStorageManagerInfo. Table 6-2: on page 6-52 lists the possible values for the index operators. Refer to “ Index Operator Notes” on page 6-55 for more detail.

The SQL engine also passes the operator when it calls IndexScanHandle.getNextRecord. The storage manager can process it during execution of either routine.

Table 6-2: Index Scan Comparison Operators

Operator	Type of Scan	Number of Comparison Values
StorageCodes.IXOP_EQ	Equal	One for each index component used
StorageCodes.IXOP_GT	Greater than	One for each index component used
StorageCodes.IXOP_GE	Greater than or equal	One for each index component used
StorageCodes.IXOP_LE	Less than or equal	One for each index component used
StorageCodes.IXOP_LT	Less than	One for each index component used
StorageCodes.IXOP_NE	Not equal	One for each index component used
StorageCodes.IXOP_BET	Inclusive between	Two for each index component used
StorageCodes.IXOP_BET_IE	Low-end inclusive between	Two for each index component used
StorageCodes.IXOP_BET_EI	High-end inclusive between	Two for each index component used
StorageCodes.IXOP_BET_EE	Exclusive between	Two for each index component used
StorageCodes.IXOP_NOTBET	Not between (inclusive)	Two for each index component used
StorageCodes.IXOP_FIRST	Start at first record	None
StorageCodes.IXOP_LAST	Return last record	None
StorageCodes.IXOP_IN	Equal to any of a list of one or more values	One for each index component used and each value in the list
StorageCodes.IXOP_NOTIN	Not equal to any of a list of one or more values	One for each index component used and each value in the list

IN int numberOfFieldValues

The number of index components used in the predicate for which the scan is to return records. This varies from zero (for StorageCodes.IXOP_FIRST or StorageCodes.IXOP_LAST) up to the number of components in the index.

The implication of this number depends on the operator. For instance, a numberOfFieldValues of 3 means:

- For basic predicates (StorageCodes.IXOP_EQ, StorageCodes.IXOP_GT, IXOP_GE, StorageCodes.IXOP_LE, IXOP_LT, and StorageCodes.IXOP_NE),

there are 3 values in *searchValue*. A predicate for a StorageCodes.IXOP_EQ operator would be of the form:

A = index_search_val1 AND B = index_search_val2 AND C = index_search_val3

- For between operators (StorageCodes.IXOP_BET, StorageCodes.IXOP_BET_IE, IXOP_BET_EI, StorageCodes.IXOP_BET_EE, and StorageCodes.IXOP_NOTBET), there are 6 values in *searchValue*, and the predicate is of the form:

A BETWEEN index_search_val1 AND index_search_val2 AND

B BETWEEN index_search_val3 AND index_search_val4 AND

C BETWEEN index_search_val5 AND index_search_val6

- For StorageCodes.IXOP_IN and StorageCodes.IXOP_NOTIN, there are 3 sets of values (for these operators, *numberOfFieldValues* does not imply the number of values in the sets) and the predicate is of the form:

A IN (index_search_val1 , index_search_val2 , ...) AND

B IN (index_search_valx, ...) AND

C IN (index_search_valy, ...)

in FieldValues searchValue

The list of values to use for comparison when searching for an index record. The SQL engine passes the same list when it calls *IndexScanHandle.getNextRecord*. The storage manager can process the values during execution of either routine.

int scanHint

Indicates if fixed length keys are used.

IN int fetchHint

Indicates if the record is being fetched in the context of a SQL statement which only performs read operations or if it is being executed in the context of a SQL statement that could perform writes. *fetchHint* will be one of the following values:

- StorageCodes.TPL_FH_READ: The record being fetched is not a candidate for being updated in the context of the current SQL statement.

- StorageCodes.TPL_FH_WRITE: The record being fetched is a candidate for being updated in the context of the current SQL statement.

fetchHint indicates that a selected index record may be updated via the *IndexHandle.insert, IndexHandle.delete, TableHandle.update* or *TableHandle.delete* functions. This flag may be used by certain storage managers whose concurrency control policy (locking policy) needs to differentiate or wishes to differentiate between reading a record and reading a record for update.

Description

The SQL engine calls *StorageManagerHandle.getIndexHandle* to open an index for update operations. In response, the storage manager makes sure the index is open and supplies a handle that the SQL engine uses for subsequent index update methods. The *tableID* and *indexID* arguments taken in combination identify the particular index to

be opened. The SQL engine obtains *indexID* and *tableID* from the SYSINDEXES catalog table.

Although the SQL engine presumes that the index specified by *indexID* is open after calling *StorageManagerHandle.getIndexHandle*, the storage manager should not automatically open files or load data structures each time the SQL engine calls this function. This is because previous SQL statements may have resulted in calls to functions that already opened the index. Instead, the storage manager should use whatever file-caching mechanism exists in the underlying storage system to check if the index is already open, and open it only if necessary.

Index Operator Notes

The operator argument describes the type of index scan to perform by indicating the comparison criteria for selecting records from the index.

Implementations indicate support for various comparison operators by including them in the array of values they return in response to the StorageCodes.IX_PUSH_DOWN_RESTRICTS info type argument of *StorageManagerHandle.getStorageManagerInfo*.

Implementations must at least support the StorageCodes.IXOP_FIRST operator. If the storage manager does not support a particular operator, the SQL engine processes such predicates internally. If the storage manager indicates it does not support processing of any but the StorageCodes.IXOP_FIRST index operator, the SQL engine requests that the storage manager return all records by passing the StorageCodes.IXOP_FIRST operator.

The SQL engine "pushes-down" processing of supported predicates to the storage manager. The objective of pushing down such index predicates is to reduce the overall cost of executing an SQL statement by allowing the SQL engine optimizer to consider options not otherwise available.

For operator values that supply comparison values, *searchValue* contains the values to be compared as well as their field ids and data types. Note that the values of operator and *searchValue* the SQL engine provides in *StorageManagerHandle.getIndexScanHandle* are also provided on each call to *IndexScanHandle.getNextRecord*. It is up to the storage manager to decide whether to process the operator value during execution of *StorageManagerHandle.getIndexScanHandle* or *IndexScanHandle.getNextRecord*.

The following discussion gives some more detail on the individual operators.

StorageCodes.IXOP_EQ, StorageCodes.IXOP_GT, StorageCodes.IXOP_GE, StorageCodes.IXOP_LE, StorageCodes.IXOP_LT, and StorageCodes.IXOP_NE

For these operators, the number of comparison values provided will be from one (1) up to the number of components in the index. All index records whose components values match the comparison values according to the operator that is provided should be returned via *IndexScanHandle.getNextRecord*.

StorageCodes.IXOP_BET, StorageCodes.IXOP_BET_IE, StorageCodes.IXOP_BET_EI, StorageCodes.IXOP_BET_EE, and StorageCodes.IXOP_NOTBET

For these operators, there are two comparison values for each index component. Each pair of comparison values indicates the upper and lower bounds of a range. So, the number of comparison values the SQL engine supplies in *SearchValue* is twice the value passed in the *numberOfFieldValues* input argument.

When the SQL engine calls *IndexScanHandle.getNextRecord* with one of the range operators, the storage manager should return all index records whose components meet the criteria detailed in the following table:

Table 6-3: BETWEEN Range Operators

Operator	Returns
StorageCodes.IXOP_BET	Records whose components are greater than or equal to the lower bound of the range, and less than or equal to the upper bound of the range.
StorageCodes.IXOP_BET_IE	Records whose components are greater than or equal to the lower bound of the range, and less than the upper bound of the range.
StorageCodes.IXOP_BET_EI	Records whose components are greater than the lower bound of the range, and less than or equal to the upper bound of the range.
StorageCodes.IXOP_BET_EE	Records whose components are greater than the lower bound of the range, and less than the upper bound of the range.
StorageCodes.IXOP_NOTBET	Records whose components are less than the lower bound of the range, and greater than the upper bound of the range.

StorageCodes.IXOP_FIRST and StorageCodes.IXOP_LAST

For operators StorageCodes.IXOP_FIRST and StorageCodes.IXOP_LAST, there are no comparison values:

- StorageCodes.IXOP_FIRST indicates that the index scan will start with the first record of the index. The storage system should iterate through all other records on successive calls to *IndexScanHandle.getNextRecord*.
- StorageCodes.IXOP_LAST indicates that the index scan need only return the last record in the index. The storage system will not need to iterate through the index backwards.

Note that which record is first or last is dependent on the sort order of the fields within the index. See *IndexHandle.insert* page 6-36 for more detailed description of how records are ordered within an index.

StorageCodes.IXOP_IN

For the StorageCodes.IXOP_IN operator, there is a set of comparison values for each index component. The storage manager must determine how many comparison values there are for each index component by examining the *searchValue* argument.

With StorageCodes.IXOP_IN, the storage manager should return all index records whose components have values in the cross-product of the sets of comparison values, as shown in the following table:

Table 6-4: Rows Returned for IXOP_IN

Component 1 comparison values	Component 2 comparison values	Component 3 comparison values	Rows Returned
a, b	1, 2	x, y	a 1 x a 1 y a 2 x a 2 y b 1 x b 1 y b 2 x b 2 y

If a storage manager does not support StorageCodes.IXOP_IN (as indicated in the storage manager response to *StorageManagerHandle.getStorageManagerInfo*), the SQL engine checks whether the storage manager supports StorageCodes.IXOP_EQ. If it does, the SQL engine translates an IN predicate to a series of calls to *StorageManagerHandle.getIndexScanHandle* using StorageCodes.IXOP_EQ. If it does not, the SQL engine processes the predicate internally.

StorageCodes.IXOP_NOTIN

The StorageCodes.IXOP_NOTIN operator is similar to StorageCodes.IXOP_IN, with the following differences:

- With StorageCodes.IXOP_NOTIN, the storage manager should return all index records whose components do not have values in the cross product of the sets of comparison values.
- If a storage manager does not support StorageCodes.IXOP_NOTIN (as indicated in the storage manager response to *StorageManagerHandle.getStorageManagerInfo*), the SQL engine processes the predicate internally, without first checking for support of StorageCodes.IXOP_NE.

6.4.2.9 getStorageManagerInfo

Returns details on storage systems support for indexes and other properties.

```
public Object
getStorageManagerInfo (int infoType, char indexType)
                              throws DharmaStorageException
```

Returns

An object containing the request information. The type of the object varies depending on the information requesed.

Arguments

IN int infoType

The type of information which is being requested. See “infoType values” on page 58. for details on valid *infoType* values.

IN char indexType

A one-character flag that specifies the type of index the SQL engine is requesting information about. CREATE INDEX statements specify the index type in the optional TYPE argument, and the SQL engine calls *StorageManagerHandle.getStorageManagerInfo* for details on the properties of each index type. (If the CREATE INDEX statement omits the TYPE argument, the SQL engine sets the index type to B.)

The index type does not imply any particular indexing technique. It is an arbitrary flag that allows the storage manager to indicate different properties for multiple types of indexes. The SQL engine calls *StorageManagerHandle.getStorageManagerInfo* for each index type, and the storage manager can respond with different index properties for each type. (For instance, that different index types support different comparison operators.)

Note The SQL engine does not supply an index type when it calls *StorageManagerHandle.getStorageManagerInfo* with the IX_UPD_REQUIRED *infoType* value. In that case, *indexType* is ignored, and the SQL engine assumes that the response is true for all index types.

Description

The SQL engine calls *StorageManagerHandle.getStorageManagerInfo* to get details on the properties of different types of indexes supported by a storage manager. The *infoType* argument specifies the property of interest, and the *indexType* argument specifies the index type.

infoType values

StorageCodes.IX_ALL_COMPONENTS

Description:	Specific to indexes that include multiple table columns (multiple-component indexes). When performing an index scan, whether search values must be provided for all components. If TRUE is returned in response to StorageCodes.IX_ALL_COMPONENTS, then the storage manager is indicating that when the SQL engine is performing an index scan, within the *searchValue* list, a comparison value must be provided for all components of the index.
Input Parameter:	*indexType*
Output Type:	Boolean

StorageCodes.IX_COMPUTE_AGGR

Description: Whether the storage manager supports the SQL MIN and MAX aggregate functions. (In other words, if the storage manager returns TRUE to StorageCodes.IX_SORT_ORDER, and returns StorageCodes.IXOP_FIRST, and StorageCodes.IXOP_LAST in response to StorageCodes.IX_PUSH_DOWN_RESTRICTS, it should also return TRUE to StorageCodes.IX_COMPUTE_AGGR.)

Input Parameter: *indexType*

Output Type: Boolean

StorageCodes.IX_FETCH_ALL_FIELDS

Description: If TRUE is returned in response to StorageCodes.IX_FETCH_ALL_FIELDS, then the storage manager is indicating that in response to a call to *IndexScanHandle.getNextRecord*, the storage system is able to return all of the fields of the record, and not just the index component fields. The SQL engine takes advantage of this property to avoid *TableHandle.getRecord* calls.

Input Parameter: *indexType*

Output Type: Boolean

StorageCodes.IX_PUSH_DOWN_RESTRICTS

Description: Comparison operators which the storage manager can process during index scans for the specified type of index. The return value is an array of Integers indicating which operators are supported by the storage manager. The SQL engine will only push down processing of operators that are contained within the list that the storage manager returns. The SQL engine uses this list as the basis for the operator input argument to *IndexScanHandle.getNextRecord* and *StorageManagerHandle.getIndexScanHandle*. See Table 6-2: on page 6-52 for a list of the valid values. If the storage manager indicates it does not support processing of an index comparison operator, the SQL engine processes the operator internally.

In the following example, the storage manager supports 6 different operators for the index type 'B':

```
public Object
getStorageManagerInfo (int infoType,
                       char indexType) throws DharmaStorageException
{
  . . .
 switch (infoType)
 {
   . . .
   case StorageCodes.IX_PUSH_DOWN_RESTRICTS:
   {
      if (indexType== 'B' || indexType =='b' )
      {
         Integer tmp[] = new Integer[6];
         tmp[0] = new Integer(StorageCodes.IXOP_EQ);
         tmp[1] = new Integer(StorageCodes.IXOP_GT);
         tmp[2] = new Integer(StorageCodes.IXOP_GE);
         tmp[3] = new Integer(StorageCodes.IXOP_LE);
         tmp[4] = new Integer(StorageCodes.IXOP_LT);
         tmp[5] = new Integer(StorageCodes.IXOP_FIRST);
        return tmp;
      }
      break;
   }
  . . .
```

Input Parameter: *indexType*

Output Type: Array of Integer.

StorageCodes.IX_SCAN_ALLOWED

Description: Whether indexes of the specified type support index scans. Storage managers return FALSE for indexes that are inherently non-scan-oriented, such as hash indexes.

Input Parameter: *indexType*

Output Type: Boolean

StorageCodes.IX_SORT_ORDER

Description: Whether indexes of the specified type are sorted. In other words, whether a scan on the index returns records in the order of the index key.

Input Parameter: *indexType*

Output Type: Boolean

StorageCodes.IX_TID_SORTED

Description: Whether indexes of the specified type return records sorted by *RecordID*. Ordinarily, indexes only guarantee to return records that meet the provided comparison criteria and the records are not sorted by *RecordID*. However, if the storage manager sets IX_TID_SORTED to TRUE, the SQL engine can significantly optimize processing of compound predicates that specify multiple indexes on the same table (including specifying the same index multiple times).

For instance, the following search condition benefits from returning records that are sorted by tuple identifier:

```
WHERE C1 > 12 AND C2 > 24
```

In this case, the SQL engine first retrieves the *RecordID*s returned by the first predicate, then retrieves the *RecordID*s returned by the second predicate, and performs an intersect operation on the two sets. This intersect operation is much more efficient if the SQL engine can assume the sets are returned in tuple identifier order.

Typically, to support StorageCodes.IX_TID_SORTED, storage managers need to perform special processing at run time. Or, if a table is loaded with rows in index key order (resulting in the index key and tuple identifier sort order being the same), storage managers can support StorageCodes.IX_TID_SORTED for that index key.

Input Parameter: *indexType*

Output Type: Boolean

StorageCodes.IX_UPD_REQUIRED

Description:	Whether the SQL engine must update indexes after an insert, update, or delete operation on a table. If the storage manager sets StorageCodes.IX_UPD_REQUIRED to TRUE, it indicates that the SQL engine must directly manage the updating of indexes in addition to tables. When an SQL INSERT, UPDATE, or DELETE statement is executed on some table, in addition to calling *TableHandle.insert, TableHandle.update*, or *TableHandle.delete* on the table, the SQL engine will execute *IndexHandle.insert*, or *IndexHandle.delete* on the corresponding indexes. If FALSE is returned, then the SQL engine will assume that the index will be updated indirectly by the storage manager as a side effect of the execution of *TableHandle.insert*, *TableHandle.update*, or *TableHandle.delete*.
Input Parameter:	None
Output Type:	Boolean

6.4.2.10 close

Close the Storage manager handle.

```
public void
close ()  throws DharmaStorageException
```

Returns

None

Arguments

None

Description

Closes storage manager handle.

6.5 MISCELLANEOUS CLASSES

6.5.1 StorageCodes

This class is used to hold all the constant values used by the storage implementation. The error codes are listed in this class. Stub developers add required constants including error codes in this class.

Definition

```
public class StorageCodes
```

Members

All the error codes and commonly used literals are listed here.

Methods

None

6.5.2 DharmaStorageException

This is the Exception class thrown when there is an error condition. All the error messages are stored in a static table of error messages. Stub developers can add new error messages and error codes into this table. Error codes must be defined in the StorageCodes class.

Definition

```
public class DharmaStorageException extends Exception
```

Members

int errorcode;

Error code for exception being thrown.

private static Hashtable ErrorTable;

Table containing all error codes and their associated text strings.

Methods

6.5.2.1 DharmaStorageException

This constructor should be used to thow exceptions.

```
public
DharmaStorageException (int errcode)
```

Returns

DharmaStorageException object.

Arguments

IN int errorcode

The id of the exception being raised. The error code should be in the *ErrorTable.*

Description

This constructor is used to throw an exception.

Example:

```
// Property not supported.
throw new DharmaStorageException
        (StorageCodes.ERR_UNSUPPORTED_RSSINFO);
```

6.5.2.2 getErrorMessage

Return the error message for a given error code.

```
public static String
getErrorMessage (int errorcode)
```

Returns

A String object containing the error message.

Arguments

IN int errorcode

The id of the exception being raised. The error code should be in the *ErrorTable.*

Description

This method returns a string associated with the argument *errorcode*.

6.5.2.3 getErrorMessage

Return the error code associated with the *DharmaStorageException* object.

```
public int
getErrorCode()
```

Returns

An int containing the error code.

Arguments

None

Description

This method returns the value of the *errorcode* member.

6.6 MAPPING BETWEEN SQL AND JAVA DATA TYPES

SQL Engine follows a strict mapping between SQL types and Java data types. The Mapping is given in the following table.

Table 6-5: Mapping Between SQL and Java Data Types

SQL Data Type	Java Data Type
StorageCodes.CHAR	java.lang.String
StorageCodes.NVARCHAR	java.lang.String
StorageCodes.INTEGER	java.lang.Integer
StorageCodes.SMALLINT	java.lang.Short
StorageCodes.BIT	java.lang.Boolean
StorageCodes.REAL	java.lang.Float
StorageCodes.FLOAT	java.lang.Double
StorageCodes.DATE	java.sql.Date
StorageCodes.TIME	java.sql.Time
StorageCodes.TIMESTAMP	java.sql.Timestamp
StorageCodes.TINYINT	java.lang.Byte
StorageCodes.BIGINT	java.lang.Long
StorageCodes.NUMERIC	java.math.BigDecimal

Table 6-5: Mapping Between SQL and Java Data Types

StorageCodes.MONEY	java.math.BigDecimal
StorageCodes.BINARY	java.io.BufferedInputStream

SQL Engine constructs Java objects as per the mapping given in the above table for each column and sets them in FieldValue object for TableHandle.insert, TableHandle.update, IndexHandle.insert, IndexHandle.delete, IndexScanHandle.getNextRecord and StorageManagerHandle.getIndexScanHandle calls. SQL Engine expects objects of the specfic Java types to be present in DharmaRecord returned by TableHandle.getRecord, TableScanHandle.getNextRecord and IndexScanHandle.getNextRecord methods.

Appendix A

Server Utility Reference

A.1 OVERVIEW

This sections contains reference information on utilities used to configure the Dharma SDK Server.

- The *dhdaemon* executable image starts the Dharma SDK Server and enables network access from clients.
- On Windows, the *pcntreg* utility registers the *dhdaemon* executable image as a service in the system registry.
- *isql* loads metadata into the data dictionary and provides a simple, general-purpose SQL interface on the server.
- *mdcreate* creates a data dictionary and provides a name for access to the proprietary storage system

A.2 DHDAEMON

The *dhdaemon* executable image starts the Dharma SDK Server and enables network access from clients:

- On UNIX, *dhdaemon* is the only way to start the server process.
- On Windows, *dhdaemon* is an alternative to starting the server process as a service.

Syntax

```
dhdaemon  [ option [ option ... ] ] { start | stop | status }

option ::

          -c

|         -e server_name

|         -s service_name

|         -q
```

Arguments

-c

On Windows, starts the server as a console application. This approach allows you to use debugging tools and allows user-level environment variables (such as TPESQLDBG) to affect the *dhdaemon* process. (When started as a service, the

dhdaemon process only sees system environment variables.) The *-c* option is applicable only to Windows, and required there to start the server from the command line.

-e server_name

The name of the executable to use for the Dharma SDK Server process. For example, use the *-e* option to specify the sample implementation executable demo as the Dharma SDK Server process:

```
$ dhdaemon -e $TPEROOT/bin/dhdemo start
```

-s service_name

The name of a network service in the services file. If the *dhdaemon* command does not include the *-s* option, the default is sqlnw.

-q

Starts the *dhdaemon* process in "quiet mode", which displays fewer messages.

start

Starts the *dhdaemon* process.

stop

Stops the *dhdaemon* process.

status

Displays the status of the process and any child processes it has spawned. For example:

```
$ dhdaemon status


             Dharma/dhdaemon Version 09.00.0000

        Dharma Systems Inc              (C) 1988-2005.

        Dharma Systems Pvt Ltd          (C) 1988-2005.


Daemon version:           Feb 10 2005 17:02:43

       running since:     02/19/2005 17:46:22   on bhima

Working directory:        /vol6/sdkdir

SQL-Server version:       /vol6/sdkdir/bin/dhdaemon

Nr of servers started:    101

              running:    0

              crashed:    0
```

A.3 PCNTREG

Adds and deletes entries for the Dharma SDK in the Windows registry.

Note The *pcntreg* utility is only applicable to Windows.

Syntax

```
pcntreg { p path | d }
```

Arguments

p path

Register *dhdaemon*. The path argument specifies the disk and directory name for the top-level *dharma* directory (for example, *c:\dharma*). If the path argument contains spaces, it needs to be delimited by double quotes, as shown below:

```
C:\>pcntreg p "C:\Program Files\Dharma Systems
Inc\dhsdk_product"
```

d

Delete the registry entry for *dhdaemon*.

A.4 MDCREATE

Creates a data dictionary that stores metadata (details on the structure of SQL tables and indexes).

Syntax

```
mdcreate [ -v ] [ -d directory_spec ] dbname
```

Arguments

-v

Specifies verbose mode, so *mdcreate* generates detailed status messages.

-d directory_spec

Specifies an alternative directory specification in which to create the data dictionary. This argument is valid only for the Desktop configuration.

The *mdcreate* utility creates a subdirectory to contain the data dictionary files. It uses the name specified in the dbname argument for the subdirectory. There are three levels of defaults that determine where *mdcreate* creates this subdirectory:

- The directory specified by the *-d* argument
- If the *mdcreate* does not specify *-d*, the directory specified by the TPE_DATADIR environment variable
- If TPE_DATADIR is not set, *mdcreate* creates the dbname subdirectory under the directory specified by the TPEROOT directory.

For example:

```
%TPEROOT%\bin\mdcreate -d “E:\Data Files\Dharma Databases”
demo_db
```

This command creates a subdirectory called *demo_db.dbs* under the *e:\Data Files\Dhamra Databases* directory and populates the directory with the necessary files.

Once you create the database subdirectory in this manner, you must explicitly specify its location in *isql* command lines and when you add ODBC data source names:

- In *isql*, use the *-d* option to specify the same directory path as you used for *mdcreate*. For example:

```
isql -s %TPEROOT%\odbcsdk\sample\md_template -d "E:\Data
Files\Dharma Databases" demo_db
```

- In the Microsoft ODBC Administrator utility, the ODBC Setup dialog box contains a Data Dir text-box field. Use it to specify the same directory path as you used for *mdcreate*.

dbname

The name of the database. ODBC applications and the *isql* utility specify *dbname* to access the database. The name of the database should not exceed 32 characters, excluding the *.dbs* extension. Also while specifying the database name, it should not include the *.dbs* extension.

A.5 ISQL

The primary use for *isql* is to load metadata into data dictionaries via a SQL script, which contains CREATE TABLE and INDEX statements with the STORAGE_ATTRIBUTES 'METADATA_ONLY' clause. This clause directs the SQL engine to insert metadata into the data dictionary without requiring the proprietary storage system to create an empty table or index. The table or index name used in the CREATE statement must be the same as an existing table or index in the proprietary storage system.

You can also use *isql* to create new tables or issue SQL queries interactively. Invoke it without the *-s* option and specify the database you want to access. Terminate statements with a semicolon. To exit from interactive *isql*, type CTRL/D.

Syntax

```
isql [-s script_file] [-u user_name] [-a password] [ -d
directory_spec ] dbname
```

Arguments

-s script_file

The name of a SQL script file *isql* executes.

Note: If the file name has a space, such as:

```
test script.sql
```

The file name must be enclosed in doubles quotes, such as:

```
isql -s "test script.sql" testdb
```

-u user_name

The user name to connect to the database specified. The default is the current user of the operating system. Unless you log in as *dharma*, you should specify *-u dharma* on the *isql* command line.

-a password

The password to connect to the database specified. The default is null.

-d directory_spec

An alternative location for the data dictionary directory. This argument is valid only for the Desktop configuration. If the *mdcreate* utility specified the *-d* argument, *isql* must specify the same argument (or the TPE_DATADIR environment variable should specify *directory_spec*).

dbname

The name of the database, as specified to the *mdcreate* utility. The name of the database should not exceed 32 characters, excluding the *.dbs* extension. Also while specifying the database name, it should not include the *.dbs* extension.

Appendix B

System Catalog Tables

B.1 OVERVIEW

The Dharma SDK maintains a set of system tables for storing information about tables, columns, indexes, constraints, and privileges. These tables are called system catalog or dictionary tables.

SQL data definition statements and GRANT and REVOKE statements update system catalog tables. Users have read access to the system catalog tables. The database administrator has update access to the tables, but should avoid modifying them directly.

There are two types of tables in the system catalog: base tables and extended tables. Base tables store the information on the table spaces, tables, columns, and indexes that make up the database. The extended tables contain information on constraints, privileges, and statistical information.

The owner of the system tables is *dharma*. If you connect to a Dharma environment with a User ID other than *dharma*, you need to qualify references to the tables in SQL queries. For example:

```
SELECT * FROM DHARMA.SYSTABLES
```

The following table shows details of the columns in each system table. Here is the SQL query that generated the data for the table. You can modify it to generate a similar list that includes user-created tables by omitting the line and *st.tbltype = 'S'*.

```
select sc.tbl 'Table', sc.col 'Column',

  sc.coltype 'Data Type', sc.width 'Size'

from dharma.syscolumns sc, dharma.systables st

where sc.tbl = st.tbl

  and st.tbltype = 'S'

order by sc.tbl, sc.id
```

B.2 SYSTEM CATALOG TABLES DEFINITIONS

The following table lists all the tables in the system catalog. It gives a brief description of their purpose and lists the column definitions for every table.

Table B-1: System Catalog Table Definitions

Table	Purpose	Column	Data Type	Size
sys_chk_constrs	Contains the CHECK clause for each check constraint specified on a user table.	chkclause	varchar	2000
		chkseq	integer	4
		cnstrname	varchar	32
		owner	varchar	32
		tblname	varchar	32
sys_chkcol_usage	Contains one entry for each column on which the check constraint is specified	cnstrname	varchar	32
		colname	varchar	32
		owner	varchar	32
		tblname	varchar	32
sys_keycol_usage	Contains one entry for each column on which primary or foreign key is specified	cnstrname	varchar	32
		colname	varchar	32
		colposition	integer	4
		owner	varchar	32
		tblname	varchar	32
sys_ref_constrs	Contains one entry for each referential constraint specified on a user table	cnstrname	varchar	32
		deleterule	varchar	1
		owner	varchar	32
		refcnstrname	varchar	32
		refowner	varchar	32
		reftblname	varchar	32
		tblname	varchar	32

Table B-1: System Catalog Table Definitions

Table	Purpose	Column	Data Type	Size
sys_tbl_constrs	Contains one entry for each table constraint.	cnstrname	varchar	32
		cnstrtype	varchar	1
		idxname	varchar	32
		owner	varchar	32
		tblname	varchar	32
syscalctable	Contains exactly one row with a single column with a value of 100.	fld	integer	4
syscolauth	Contains the update privileges held by users on individual columns of tables in the database.	col	varchar	32
		grantee	varchar	32
		grantor	varchar	32
		ref	varchar	1
		tbl	varchar	32
		tblowner	varchar	32
		upd	varchar	1
syscolumns	Contains one row for each column of every table in the database.	charset	varchar	32
		col	varchar	32
		collation	varchar	32
		coltype	varchar	10
		dflt_value	varchar	250
		id	integer	4
		nullflag	varchar	1
		owner	varchar	32
		scale	integer	4
		tbl	varchar	32
		width	integer	4
sysdatatypes	Contains information on each data type supported by the database.	autoincr	smallint	2
		casesensitive	smallint	2

Table B-1: System Catalog Table Definitions

Table	Purpose	Column	Data Type	Size
		createparams	varchar	32
		datatype	smallint	2
		dhtypename	varchar	32
		literalprefix	varchar	1
		literalsuffix	varchar	1
		localtypename	varchar	1
		nullable	smallint	2
		odbcmoney	smallint	2
		searchable	smallint	2
		typeprecision	integer	4
		unsignedattr	smallint	2
sysdbauth	Contains the database-wide privileges held by users.	dba_acc	varchar	1
		grantee	varchar	32
		res_acc	varchar	1
sysidxstat	Contains statistics for each index in the database.	idxid	integer	4
		nleaf	integer	4
		nlevels	smallint	2
		recsz	integer	4
		rssid	integer	4
		tblid	integer	4
sysindexes	Contains one row for each component of an index in the database. For an index with n components, there will be n rows in this table.	colname	varchar	32
		id	integer	4
		idxcompress	varchar	1
		idxmethod	varchar	1
		idxname	varchar	32
		idxorder	varchar	1

Table B-1: System Catalog Table Definitions

Table	Purpose	Column	Data Type	Size
		idxowner	varchar	32
		idxsegid	integer	4
		idxseq	integer	4
		idxtype	varchar	1
		rssid	integer	4
		tbl	varchar	32
		tblowner	varchar	32
syssynonyms	Contains one entry for each synonym in the database.	ispublic	smallint	2
		screator	varchar	32
		sname	varchar	32
		sowner	varchar	32
		sremdb	varchar	32
		stbl	varchar	32
		stblowner	varchar	32
systabauth	Contains privileges held by users for tables, views, and procedures.	alt	varchar	1
		del	varchar	1
		exe	character	1
		grantee	varchar	32
		grantor	varchar	32
		ins	varchar	1
		ndx	varchar	1
		ref	varchar	1
		sel	varchar	1
		tbl	varchar	32
		tblowner	varchar	32
		upd	varchar	1
systables	Contains one row for each table in the database.	creator	varchar	32
		has_ccnstrs	varchar	1
		has_fcnstrs	varchar	1

Table B-1: System Catalog Table Definitions

Table	Purpose	Column	Data Type	Size
		has_pcnstrs	varchar	1
		has_ucnstrs	varchar	1
		id	integer	4
		owner	varchar	32
		rssid	integer	4
		segid	integer	4
		tbl	varchar	32
		tbl_status	varchar	1
		tblpctfree	integer	4
		tbltype	varchar	1
systblspaces	No longer used.	id	integer	4
		tsname	varchar	32
systblstat	Contains table statistics for each user table.	card	integer	4
		npages	integer	4
		pagesz	integer	4
		recsz	integer	4
		rssid	integer	4
		tblid	integer	4
sysviews	Contains information on each view in the data-base.	creator	varchar	32
		owner	varchar	32
		seq	integer	4
		viewname	varchar	32
		viewtext	varchar	2000

Appendix C

Storing NUMERIC Data Directly

C.1 OVERVIEW

This section describes how storage managers can store and return values defined as the SQL NUMERIC data type using the internal Dharma SDK storage format.

This is typically not necessary, since storage managers can use the function *dhcs_conv_data* to convert from the internal NUMERIC storage format to a form that is easy to manipulate. (For details on *dhcs_conv_data*, see page 5-73.) Using *dhcs_conv_data* allows storage managers to avoid the complexities of dealing with the NUMERIC storage format directly.

However, some implementations choose to directly manipulate NUMERIC values for better performance, or because their own internal storage format is similar.

In SQL, type NUMERIC corresponds to a number with the given precision (maximum number of digits) and scale (the number of digits to the right of the decimal point). By default, NUMERIC columns have a precision of 32 and scale of 0.

Internally, Dharma SDK uses the *dhcs_num_t* structure to store and return values of NUMERIC type. Storage managers can access *dhcs_num_t* to directly store and retrieve NUMERIC values, as discussed in the rest of this section.

C.2 INTERNAL STORAGE FORMAT FOR NUMERIC DATA

The *dhcs_num_t* structure is as follows:

```
typedef struct {
   short   dec_num ;
   char    dec_digits [17] ;
} dhcs_num_t  ;
```

dec_num

The *dec_num* field of the structure contains the number of valid bytes in the *dec_digits* array.

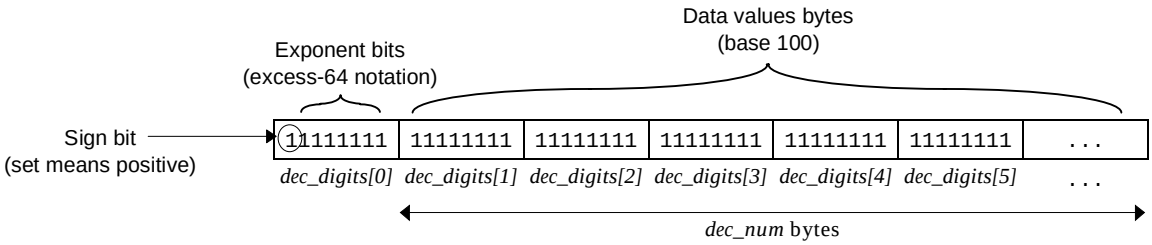
dec_digits [17]

The *dec_digits* array contains the actual numeric data, stored in two parts:

- The first byte of *dec_digits* (*dec_digits[0]*) contains a sign bit and the exponent for the value, stored in "excess-64" notation.
- The second and subsequent valid bytes of *dec_digits (dec_digits[1]* through *dec_digits[dec_num])* contain base-100 values each representing two digits.

The following figure shows the format for the *dec_digits* array.

Figure C-1: Format for NUMERIC Data Stored in the *dhcs_num_t* Structure



The following section describes in detail how to interpret data in the *dec_digits* array.

C.3 INTERPRETING NUMERIC DATA STORED IN INTERNAL FORMAT

C.3.1 Interpreting the Sign/Exponent Byte of *dec_digits*

The high-order bit of *dec_digits[0]* specifies the sign of the NUMERIC data: 1 means positive and 0 means negative.

The 7 lower-order bits of *dec_digits[0]* contain the exponent, stored in excess 64 notation. In excess-64 notation, you subtract 64 from the stored value to determine the actual value.

For *dec_digits[0]*, this means you subtract 64 from the value stored in the 7 lower-order bits to determine the value of the exponent. However, if the sign bit of *dec_digits[0]* is 0 (indicating a negative value), you must first perform a one's complement of the 7 lower-order bits before subtracting 64. (To perform a one's complement, swap zeroes with ones and ones with zeroes.)

The following example shows how to determine the sign of the NUMERIC data and the value of its exponent when *dec_digits[0]* contains a base-10 value of 223.

Example 3-1: Determining Sign and Exponent of NUMERIC Values

Decimal value in dec_digits[0]	223
Binary equivalent	11011111
Sign bit	1 (Positive)
One's complement of exponent bits	Not necessary, since sign bit is positive
Binary value of exponent bits (excess-64 notation)	1011111
Decimal value of exponent bits (excess-64 notation)	95
Actual value of exponent	95 - 64 = 31

The following example shows another example of *dec_digits[0]*, containing the base-10 value of 100, which represents a negative data value and a negative exponent value.

Example 3-2: Determining Sign and Exponent of NUMERIC Values

Decimal value in dec_digits[0]	100
Binary equivalent	01100100
Sign bit	0 (Negative)
One's complement of exponent bits	0011011
Binary value of exponent bits (excess-64 notation)	0011011
Decimal value of exponent bits (excess-64 notation)	27
Actual value of exponent	27 - 64 = -37

C.3.2 Interpreting the Data Values Bytes of *dec_digits*

The rest of the bytes in the *dec_digits* array contain the base-100 digits of the NUMERIC data, two digits in each byte. To extract the values from each data byte:

1. Convert the binary value to decimal.
2. If the sign bit indicated a negative number, perform a 100's complement on the value (subtract the value from 100).

The resulting base-100 digits represent a number between 0 and 1. Multiply that result by 100 raised to the value of the exponent to get the final NUMERIC value.

C.3.3 Complete Examples: Interpreting Sign/Exponent and Data Bytes of *dec_digits*

The following examples detail how to extract base-10 values from *dec_digits*. Use them as a guide for interpreting values returned by Dharma SDK in *dhcs_num_t* or to store values in the database using *dhcs_num_t*.

Example 3-3: Interpreting NUMERIC Storage: Positive Exponent and Data

This example shows how the (base-10) value 123456 is stored in *dec_digits*.

11000011	**1101**	**100011**	**111001**
dec_digits[0]	*dec_digits[1]*	*dec_digits[2]*	*dec_digits[3]*

Sign/Exponent Byte			
Binary value in dec_digits[0]	11000011		
Sign bit	1 (Positive)		
One's complement of exponent bits	Not necessary, since sign bit is positive		

Sign/Exponent Byte			
Binary value of exponent bits (excess-64 notation)	1000011		
Decimal value of exponent bits (excess-64 notation)	67		
Actual value of exponent	67 - 64 = 3		
Data Value Bytes	**1**	**2**	**3**
Binary value	1100	100010	111000
Decimal equivalent	12	34	56
100's complement of resulting value	N/A	N/A	N/A
Base-100 digits	12	34	56

So, the resulting numeric value is 0.123456 x 1003, or 123456.

Example 3-4: Interpreting NUMERIC Storage: Negative Exponent and Data

This example shows how the (base-10) value -123456.789 is stored in *dec_digits*.

00111100	01011001	01000011	00101101	00010111	00001011
dec_digits[0]	*dec_digits[1]*	*dec_digits[2]*	*dec_digits[3]*	*dec_digits[4]*	*dec_digits[5]*

Sign/Exponent Byte					
Binary value in *dec_digits[0]*	00111100				
Sign bit	0 (Negative)				
One's complement of exponent bits	11000011				
Binary value of exponent bits (excess					
64 notation)	1000011				
Decimal value of exponent bits (excess-64 notation)	67				
Actual value of exponent	67 - 64 = 3				
Data Value Bytes	1	2	3	4	5
Binary value	1011000	1011000	101100	10110	1010
Decimal equivalent	88	66	44	22	10

100's complement of resulting value	12	34	56	78	90
Base-100 digits	12	34	56	78	90

So, the resulting numeric value is - 0.1234567890 x 1003, or - 123456.789

Appendix D

Glossary

D.1 TERMS

add [an ODBC data source]

Make a data source available to ODBC through the Add operation of the ODBC Administrator utility. Adding a data source tells ODBC where a specific database resides and which ODBC driver to use to access it. Adding a data source also invokes a setup dialog box for the particular driver so you can provide other details the driver needs to connect to the database.

cardinality

Number of rows in a result table.

client

Generally, in client/server systems, the part of the system that sends requests to servers and processes the results of those requests.

client/server configuration

The version of the Dharma SDK Desktop that implements a network ODBC architecture. In client/server configuration, the ODBC tool and the Dharma SDK ODBC Driver run on Windows or UNIX clients, while the Dharma SDK Server library runs on the UNIX or Windows server hosting the proprietary storage system.

data dictionary

Another term for system catalog.

Dharma SDK Server

The executable that results from building an implementation of the storage interfaces with the SQL engine library. To get started with the Dharma SDK, you can build a Dharma SDK Server from the supplied sample implementation of the storage interfaces. Eventually, you will build a Dharma SDK Server from your own implementation of the storage system to provide access to a proprietary storage system.

data source

See ODBC data source

desktop configuration

The version of the Dharma SDK that implements a "single-tier" ODBC architecture. In the desktop configuration, the ODBC tool, the Dharma SDK software, and the proprietary data all reside on the same Windows XP or 2000 computer.

dharma

The default owner name for all system tables in a Dharma database. Users must qualify references to system tables as *dharma.table-name*.

field handle

A handle that identifies storage for data stored in columns defined with the SQL LONG VARCHAR or LONG VARBINARY data type. Implementations create field handles when the SQL engine calls the *dhcs_tpl_insert* routine. (This is in contrast to conventional data-type columns, for which the SQL engine passes actual values to the insert routine.) Similarly, for fetch routines, implementations return field handles instead of the actual long data values.

handle

A temporary identifier for database objects. Implementations generate handles when the SQL engine calls routines to open tables, indexes, table scans, and index scans, or to access long data-type columns. The SQL engine uses the handle on subsequent calls to scan, fetch, insert, and update operations.

index handle

A handle that identifies an index open for updating. Implementations generate index handles when the SQL engine calls *dhcs_ix_open*. The SQL engine passes index handles to *dhcs_ix_insert*, *dhcs_ix_delete*, and *dhcs_ix_close*.

info type

An argument the SQL engine supplies when it calls the *dhcs_rss_get_info* storage interface. The various info types describe a storage manager's support for indexes.

metadata

Data that details the structure of tables and indexes in the proprietary storage system. The SQL engine stores metadata in the system catalog.

ODBC application

Any program that calls ODBC functions and uses them to issue SQL statements. Many vendors have added ODBC capabilities to their existing Windows-based tools.

ODBC data source

In ODBC terminology, a specific combination of a database system, the operating system it uses, and any network software required to access it. Before applications can access a database through ODBC, you use the ODBC Administrator to add a data source -- register information about the database and an ODBC driver that can connect to it -- for that database. More than one data source name can refer to the same database, and deleting a data source does not delete the associated database.

ODBC driver

Vendor-supplied software that processes ODBC function calls for a specific data source. The driver connects to the data source, translates the standard SQL statements into syntax the data source can process, and returns data to the application. The Dharma SDK ODBC Driver provides access to proprietary storage systems underlying the ODBC server.

ODBC driver manager

A program that routes calls from an application to the appropriate ODBC driver for a data source.

primary key

A subset of the fields in a table, characterized by the constraint that no two records in a table may have the same primary key value, and that no fields of the primary key may have a null value. Primary keys are specified in a CREATE TABLE statement.

query expression

The fundamental element in SQL syntax. Query expressions specify a result table derived from some combination of rows from the tables or views identified in the FROM clause of the expression. Query expressions are the basis of SELECT, CREATE VIEW, and INSERT statements

result set

Another term for result table.

result table

A temporary table of values derived from columns and rows of one or more tables that meet conditions specified by a query expression.

row identifier

Another term for tuple identifier.

scan handle

A handle that identifies an index or table open for scan operations. Implementations generate scan handles when the SQL engine calls *dhcs_tpl_scan_open* or *dhcs_ix_scan_open*. The SQL engine passes scan handles to *dhcs_tpl_scan_fetch*, *dhcs_tpl_scan_close*, *dhcs_ix_scan_fetch*, and *dhcs_ix_scan_close*.

search condition

The SQL syntax element that specifies a condition that is true or false about a given row or group of rows. Query expressions and UPDATE statements can specify a search condition. The search condition restricts the number of rows in the result table for the query expression or UPDATE statement. Search conditions contain one or more predicates. Search conditions follow the WHERE or HAVING keywords in SQL statements.

selectivity

The fraction of a table's rows returned by a query.

server

Generally, in client/server systems, the part of the system that receives requests from clients and responds with results to those requests.

SQL engine

The core component of the Dharma SDK Server. The SQL engine receives requests from the Dharma SDK ODBC Driver, processes them, and returns results. To build a Dharma SDK Server, you link implemented storage interfaces with the SQL engine library file *$TPEROOT/lib/libserver.a*.

storage interfaces

Template C routines provided with the Dharma SDK for implementing access to proprietary storage systems. The SQL engine calls the routines to access and manipulate data in a proprietary storage system. A proprietary storage system must implement supplied templates to map the storage interfaces to the underlying storage system.

Once filled in for a particular storage system, the completed stubs are called storage managers.

storage manager

A completed implementation of the Dharma SDK storage interfaces. A storage manager receives calls from the SQL engine and accesses the underlying proprietary storage system to retrieve and store data.

storage system

The proprietary database system that underlies a storage manger. The Dharma SDK provides a SQL interface to a storage system through the SQL engine and its stub interfaces.

stub interfaces

Another term for storage interfaces.

stubs

Another term for storage interfaces.

system catalog

Tables created by the SQL engine that store information about tables, columns, and indexes that make up the database. By default, the SQL engine creates and manages the system catalog independently of the proprietary storage system. The storage manager can choose to manage the system catalog by setting the DH_DYNAMIC_METADATA environment variable.

system tables

Another term for system catalog.

table handle

A handle that identifies a table open for non-scan operations. Implementations generate table handles when the SQL engine calls *dhcs_tpl_open*. The SQL engine passes table handles to *dhcs_tpl_insert*, *dhcs_tpl_delete*, *dhcs_tpl_update*, *dhcs_tpl_fetch* and *dhcs_tpl_close*.

tid

Another term for tuple identifier.

transaction

A group of operations whose changes can be made permanent or undone only as a unit. Once implementations add the ability to change data in the proprietary storage system, they must also implement transaction management to protect against data corruption.

tuple identifier

A unique identifier for a tuple (row) in a table. Storage managers return a tuple identifier for the tuple that was inserted after an insert operation. The SQL engine passes a tuple identifier to the delete, update, and fetch stubs to indicate which tuple is affected. The SQL scalar function ROWID and related functions return tuple identifiers to applications.

view

A virtual table that recreates the result table specified by a SELECT statement. No data is stored in a view, but other queries can refer to it as if it were a table containing data corresponding to the result table it specifies.

Index

M

N

O

P

R

S

T

U

V

W