OpenBase®
Users Guide

Version 6.0
March 15, 1999

Because of last-minute changes to OpenBase, some of the information in this manual may be inaccurate. Please read the Release Notes on the OpenBase CD for the latest up-to-date information.

How to Contact OpenBase:

U.S.A. and international	OpenBase International, Ltd. 58 Greenfield Road Francestown, NH 03043 U.S.A.
Ordering	Voice: (603) 547-8404 Fax: (603) 547-2423
World Wide Web	`http://www.openbase.com`
Information	`info@openbase.com`

OpenBase License Agreement

LICENSE

You may only use this software on a single network and in compliance with the specified connection limitation. You may not use OpenBase with an internet or web browser related application unless you have an OpenBase WebServer license. Additional locations or networks must obtain another license to use this software. You may transfer the program and license to another party to use, if the other party agrees to accept the terms and conditions of this licensing agreement. If you transfer the programs, you must destroy any remaining copies of the software.

TERM

The license is effective until terminated. You may terminate it at any time by destroying all the copies of the software and the manual. OpenBase International, Ltd. maintains the right to terminate your license immediately if you fail to comply with any term or condition of this agreement. You agree upon any such termination that you will destroy all materials contained herein.

PROPIETARY RIGHTS

OpenBase is copyrighted by OpenBase International, Ltd. and is proprietary. OpenBase International, Ltd. retains title and ownership of OpenBase and all copies of OpenBase. You agree to hold OpenBase authorization codes in confidence and to take all reasonable steps to prevent disclosure. You also agree to pay any illegal use of your authorization code.

LIMITED WARRANTY

OpenBase International, Ltd. warrants to you that the disk, CD or internet download package from which you obtained OpenBase, is free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of purchase.

LIMITATION OF LIABILITY

Neither OpenBase International, Ltd. nor any one else involved in the creation, production, delivery, or licensing of OpenBase makes any warranty or representation of any kind with respect to OpenBase, its quality, reliability, performance or fitness for any purpose. OpenBase is licensed in its present form, and you assume the risk as to the quality, reliability, and performance of the software and documentation.

It is your responsibility to maintain frequent backups of database and other files. In no event shall OpenBase International, Ltd. be liable to you for any consequential, special, incidental, direct, or indirect damages of any kind arising out of the use of OpenBase.

INTEGRATION

You acknowledge that you have read and understood this agreement, and by opening the seal on this package or checking the button “To continue click on the check button indicating you agree with the license” from the OpenBase Setup.app program, you agree to be bound by its terms and conditions. In addition, you agree that this agreement is the complete and exclusive statement of the agreement between you and OpenBase International, Ltd.

Table of Contents

1

OpenBase Overview

OpenBase is a client-server database providing fast data storage for multi-user applications using SQL (Standard Query Language).

This manual is meant for software developers who have a working knowledge of Objective-C or C programming. This manual is organized as follows:

In this introductory section, we will introduce you to client-server databases, and conclude with a description of our OpenBase licensing scheme.

This chapter contains the following sections:

Introduction

For nearly a decade, the OpenBase family of products have been enabling some of the most innovative business applications at work today.

Used by thousands of companies spanning 37 countries, OpenBase is proven in demanding corporate production environments. At the same time, it is an ideal SQL-standard database choice for organizations who also want a future-safe environment that lets them take advantage of the latest object technologies and web tools.

OpenBase exceeds the industry standards by providing powerful features not found in other database systems. OpenBase pioneered the concept of true data file portability and change notification across operating environments and the internet.

With its industrial-strength, fault-tolerant design, OpenBase continues to be the choice of organizations that want both the flexibility of innovative design and the proven reliability of a traditional production environment.

OpenBase comes with a host of development tools to make building applications straightforward. EOF adaptors for Apple's Enterprise Objects Framework is included so you can build applications that conform to the Apple standard for database retrieval. OpenBase's Objective-C SQL API complements this with an interface for executing dynamic SQL and retrieving results directly into your program variables. A native Java Level 4 JDBC driver and C API's for MacOS and other platforms are also provided.

Client-Server Architecture

OpenBase can be described as a server program because it runs in the background and serves the requests of client applications or applications that interface with users. While client programs run on the user's local computer, OpenBase may be located on more powerful computers that act as high-speed, central-control centers for your applications. The advantages of using this architecture can be seen as we review a traditional approach to PC databases.

In more traditional PC database systems, each user's program searches for information by itself. This means that when the database files are located at remote locations, all of the information must travel across the network each time a search is executed. With large databases and just a few users, this architecture can bring even small networks to a halt.

In contrast to traditional PC systems, OpenBase sends only the information required by client applications. All searches are performed remotely and only requested information travels across the network to client programs. This means that if a user looks for companies in Connecticut, only Connecticut-based companies will be sent across the network. This greatly improves perceived network performance and makes modem database access practical.

License Scheme

OpenBase can be licensed on a per-application, per-connection or per-seat basis. With application based licenses each application is counted as a user of the database. So a three application license allows three applications to simultaneously use the database server. Connection based licensing counts database connections regardless of origin. So connections that originate from the same application are counted the same as connections from different applications. Seat based licenses count simultaneous users, regardless of the number of database applications they may be running. A seat is defined as one user logged on to a particular CPU.

OpenBase International also offers a WebServer license for internet applications. The OpenBase Webserver license offers a single machine license for unlimited database access from the Internet or a web browser based application. OpenBase Webserver is designed to work with WebObjects.

2

Getting Started

This section is meant to assist you in setting up OpenBase to work on your network.

This chapter contains the following sections:

Installing OpenBase

Choose the following platforms to correctly install the OpenBase server. Make sure to read the release notes before installing.

MacOS X Server

1. Log in as the Administrator.
2. Copy the OpenBase setup.app to your main UFS hard drive. You can not run it from an HFS partition.
3. Run the OpenBase Setup.app program to install the OpenBase server.
4. Reboot

Windows NT

1. Log in as the Administrator.
2. Run the Setup.exe program to install the OpenBase server.
3. Launch the OpenBase manager and answer the questions.
4. Reboot

OpenStep 4.2

1. Log in as the root user.
2. Install all packages on computers where the database server process will run. Please read the README file on the Open-Base distribution for any special instructions.

3. Install the OpenBase Framework package on all client computers.
4. Copy /usr/database/bin/openisql and /usr/database/bin/launchisql on the server computer to the same location on all client computers.
5. Run the OpenBase Manager to complete the installation.
6. Reboot your computer.

After you have installed all the OpenBase packages on your server computer you need to complete the installation by running the OpenBase Manager application. The OpenBase Manager is located in your /Local/Applications, /Network/Applications or /LocalApps directory by default.

If you are not installing on MacOS X Server, the OpenBase Manager will ask you a series of questions so that OpenBase can be configured for your computing environment. On MacOS X Server the installation program asks these questions instead. They are described as follows:

1. Would you like a 30 day demo of OpenBase? If you do not have a license (or do not have your license handy) press the Demo button. Later you can enter a license key but the demo will get you going. Alternatively, if you have your license key ready, press the License button.
2. Please select sort preferences for your language. If your language is English we strongly recommend that you press OK for the default settings. Other languages can be selected using the Sort Language popup menu.

Finally, you will be presented with a form to enter your name and address. If you chose to license your software in step 1 above, fields for the serial number, authorization code, users and version will also be displayed. This is described further in the next section.

Nameserver and Localhost Setup

Localhost Setup

OpenBase has been extended to include an ip address over-ride file which helps the databases know which ip address they should use to identify your local computer. If your computer has multiple ip addresses or if you are having problems with the nameserver, you can install a localhost file in /usr/openbase which identifies the address and host you wish to use.

Some of the installation procedures automatically install this file. If you have problems you may want to edit the localhost file and make sure the ip addresses are correct. Also, if you change the ip address of your local computer you will need to edit this file to reflect the change.

NameserverHosts Setup

The NameserverHosts file in /Network/Library/OpenBase or /usr/openbase is mandatory. Almost all of the time you do not need to do anything to it because the installation process sets it up for you. However, if you need to change it, you should choose one or two computers that are normally not turned off and list their ip addresses in the NameserverHosts file. The computers must have OpenBase installed on them. Here is an example:

```
107.23.45.34
107.23.45.35
```

The nameserver acts as a directory of databases on your network. Without this file installed the servers will not operate correctly. Database clients rely on the nameserver when they connect to database servers.

OpenBase Manager

Once you have installed OpenBase on your local computer, use the workspace manager to launch the OpenBase Manager application located in the /Network/Applications directory by default (OpenBase/Apps on Windows NT). A database window will appear listing a series of demo databases you can use to start exploring OpenBase.

Sample Databases

OpenBase comes with a variety of sample databases. We provide these databases so that you can begin using OpenBase right away with a variety of examples.

The Movies and Rentals databases are used for the EOF and WebObjects demos. Before using the EOF demos you may need to configure them using a special script. Please see the OpenBase Release Notes for details.

The Company database sample is also provided for use with OpenContacts, OpenOrder, and OpenBooks. Two databases, PEOPLE and pubs, are also provided since some standard examples use these databases to illustrate the use of SQL.

Starting Databases

To start a database:

1. Select the name of the database you want to start in the database manager window. Since this manual uses the Movie database for most of its examples, select the Movie database.

2. Press the Start button.

A yellow triangle icon should appear beside the database you started, indicating that the database is in the process of starting. When the database becomes available, a green dot will appear beside the database.

In this chapter we covered the installation process and provided a brief introduction to the OpenBase Manager. In the next chapter we will discuss the OpenBase Manager application in more detail.

3

OpenBase Manager

The OpenBase Manager application provides graphical tools for managing database servers across your local area network. It includes tools for viewing database tables, editing the database schema, managing database security and starting database servers.

This chapter contains the following sections:

Managing Database Servers

The OpenBase Manager application provides graphical tools for managing database servers across your local area network. It includes tools for viewing database tables, editing the database schema, managing database security and starting database servers.

• Figure 1: OpenBase Manager Database Window

Database Window

The OpenBase Manager Database Window provides a list of databases and server hosts available. The window acts as a database control

panel allowing users to perform a variety of functions including starting and stopping databases remotely from across the network.

The icons in the first column show whether the databases are starting, running, stopping or stopped. The different icon meanings are explained as follows:

	(Yellow)	The database is in the process of starting. During this time database files are checked and rebuilt if necessary.
	(Green)	The database is running on the specified host and is ready to be accessed by client programs.
	(Yellow)	The database is in the process of cleaning up before stopping.
	(Red)	The database has been stopped.
	(Blue)	The host is unreachable.

The buttons on the right of the database list provide easy access to a number of common functions.

Starting Databases

To start a database:

1. Select the name of the database you want to start in the database window.

2. Press the Start button. This may bring up a password window asking for the password of the host where you are starting the database. If so, type in the password and press the return key. The password can be set in the OpenBase Manager Preferences Panel.

A yellow triangle icon should appear beside the database you started, indicating that the database is in the process of starting. The start-up procedure may take several seconds, during which time the server checks the data and loads some of the indexes into memory. When the database becomes available for user programs, a green dot will appear beside the database.

Stopping Databases

To stop databases select each one in the OpenBase Manager and press the Stop button.

The icon beside the database will turn to a yellow dot, indicating that it is shutting down. When the database actually stops, a red dot will be displayed. While it is not necessary, it is a good idea to stop your databases before shutting down your computer.

Creating New Databases

To create a new database press the New Database button. A panel will appear allowing you to specify a new database name and the host computer you would like your database to run on. The database name must not contain any spaces. Select the host from the popup menu. Press the Set button when you are finished. This will add the database to your database list.

Before you can use your new database you need to start it. To start the database, select the database and press the start button. When you do this a new server process will be started on the destination computer.

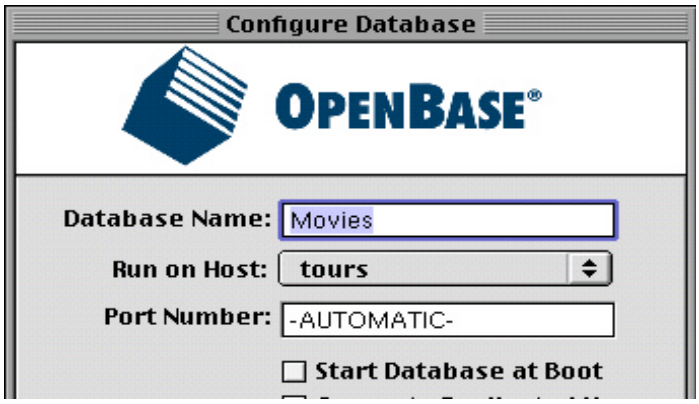
Duplicating Databases

To duplicate a database, select the database in the OpenBase Manager and press the Duplicate Database button. Only stopped databases can be duplicated. Use the Configure button to edit the name of the new database created.

Changing Database Name and Host

The OpenBase Manager provides controls for renaming the databases and / or moving them between host computers.

Selecting a database and pressing the configure button will bring up the configure database panel. From this panel you can rename your database by editing the name field. Changing the Run-on-Host popup will physically move the database from one host to another.



• Figure 2: Edit Window

To make your database automatically start when the specified computer boots up, check the Start Database at Boot check-box.

Checking the Generate Replicated Keys check-box tells the server that it needs to generate globally unique keys. This is necessary when you have several databases that need to be synchronized periodically. Each database will generate its own set of keys. Checking this box will bring up a panel asking you to specify a unique number for this database. You will need to do this for all databases in your replication group.

You can also set the encoding that the database will use internally for representing data. ASCII encoding (based on NSNextstepStringEncoding) is recommended and will work with most single-character languages. For two byte character sets we recommend that you try one of the other character encodings.

Press the Set button to save any changes.

Administration and Schema Design

When you press the Administration button on a running database, the

Administration window will appear. This window allows you to edit User access to the database, change the database schema, manage online backups, and configure database replication. By selecting the tabs on the top of the window you can go to the section you wish to work with. The five different sections are: Adding and Editing Database Users, Setting and Changing Table Permissions, Editing the Database Schema, Database Backup Manager and Replication Manager.

Adding and Editing Database Users

Adding users is not always necessary since the login always defaults to the admin user. However, if you are running in an environment where security is an issue, you may want to define users and set user access permissions. The OpenBase Manager allows you to do this through it's user management panel shown below.

• Figure 3: Administration Window User Manager

To edit the user information, select the user in the browser located on

the left side of the window. The user information will display in the editable fields on the right. You may edit this information and press the Save button to save it to the database.

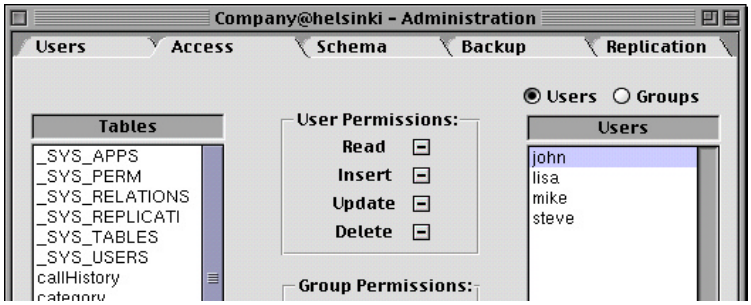
To add a new user, press the New button. A new user will be added to the list. Select the new user, edit the user information and press the Save button to save the user to the database.

To set a user's password, you need to press the Set Password button. You will be asked for the old password (if one exists) and then asked to type in the new password twice correctly. You will need to press the Save button to save the changed password to the database.

If the information on the user window appears read-only, this may mean that your current database login does not have permission to update the users table. You may need to log in as the admin user to edit this table.

Setting & Changing Table Permissions

The permissions panel of the Administration window allows you to assign access to groups and users. By default all users have access to all tables. However, as soon as you assign access to a particular table, that table becomes inaccessible to all groups or users not assigned to it. It is recommended that you don't touch the permissions if you aren't concerned about security.



• Figure 4: Administration - Permission Manager

The middle of the permissions panel contains controls to grant table access to selected users or groups. The radio buttons on the upper right toggle between assigning user access and group access.

You have the ability to give the user and the user's group, permission to perform the four standard types of database operations on each database table. These operations are select, insert, update, and delete.

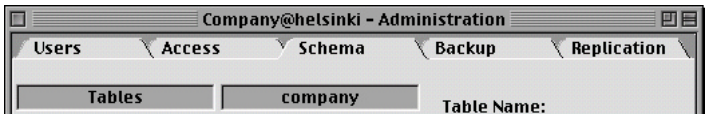
To give users permission to access a set of tables, select the tables in the viewer on the left, and select the users or groups on the right. Toggle the check-boxes in the center of the view to grant and revoke access. A `-' revokes access and a `+' grants access. When neither a plus of minus is shown on a check-box it means that it will have no affect.

To keep unauthorized users from editing user information or granting table permissions, you may want to revoke access to the tables _SYS_USERS and _SYS_PERM for each user. Revoking permissions for _SYS_TABLES will block users from creating, dropping, or renaming tables.

SQL statements or transactions are allowed to be executed on tables when the user or the user's group has the correct permission. Otherwise an error is returned.

Editing the Database Schema

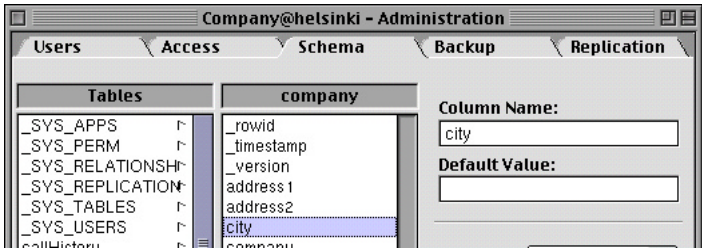
The Database Schema panel on the Administration window allows users to change the database schema and generate modify scripts for later use. This schema editor has been written to provide a robust tool for changing tables and column attributes on the fly.



• Figure 5: Administration - Edit Table panel

The Database Schema panel has a two column browser on the left for selecting database tables and columns, and a swap panel on the right for displaying database attribute controls. As you select tables and database columns the swap panel on the right will change to display the attributes of the selected item.

The Natural Order of Data popup provides a way to arrange the physical data in each table by an indexed column. This provides significant performance benefits in cases where data is accessed through a foreign key or is sorted by the specific column. By clustering the column the database is able to find records likely to be accessed together on consecutive pages.

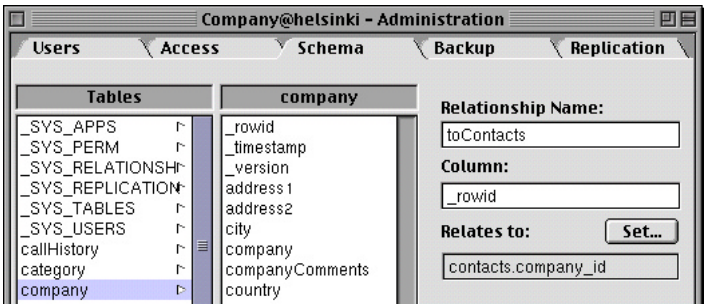


• Figure 6: Administration - Edit Column panel

To add a table, select an existing table in the first column of the browser and press the New Table button. This will create a new table entry with the default _rowid column. Change the default table name to a name of your choosing and press enter so that the new name appears in the browser on the left.

To add columns to a table, select an existing column in the table and press the New Column button. A new column with a default name will be added to the list. Edit the column name and attributes to your choosing.

The schema editing tool which comes with the OpenBase Manager also allows you to specify simple relationships. While you are free to use compound relationships in your programs, we would like to encourage you to use single clause relationships for best performance.



• Figure 7: Administration - Edit Relationships

To add a relationship, select the source column or a column that makes up the relationship and press the Relationship To button. You will be presented with a browser. Choose the matching column in the destination table and press the Ok button. You may then edit the characteristics of the relationship in the panel provided.

When you are finished changing the database schema, press the Save button to save the changes to the database or generate a script.

Database Backup Manager

OpenBase comes with an online backup mechanism which allows you to schedule complete data dumps of the database to timestamped ASCII files. The Database Backup Manager allows you to schedule when you would like the backup to take place and where the backup files should be placed.

Before you can get the backup system to work, however, you need to setup a backup server (called databackup) to point to the database you want to backup. To set this up you need to choose Setup Backup Server

from the edit menu. Add your database to the list and the backup server will read the schedule that you setup on the Backup Manager.

Once the databackup program knows which databases to monitor, it will read the backup schedule each day around midnight and schedule the backups for the following day.

When setting the backup schedule for your databases it is very important that you specify a valid pathname (with no spaces). Without a valid path the backup will not work.

Replication Manager

The Replication Manager allows you to specify which tables you would like to synchronize with other databases. This is useful when you have two databases which share the same data (or some of the same data). By using replication you can keep the data in one database in sync with another.

Pressing the new button on the Replication Manager interface brings up a login panel which allows you to login to the remote database you want to synchronize with. Then all you need to do is specify a table to replicate and tell the replication mechanism what you want to do about conflicts.

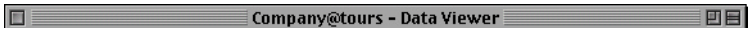
Sometimes conflicts arise when two people from different locations edit the same record. The rules regarding how to deal with these conflicts are simple. Either the local database has priority and overwrites changes made by the remote master database or the remote database has priority.

When there is a conflict and changes will be overwritten you can elect to create a backup of the overwritten information into a separate table. Backup tables are created on an as needed basis and include all information overwritten by the replication process.

The Reset Date button is for resetting the last backup date. It forces the replication to compare all records and merge the data in both tables. This is important to do if you drop a table because you want to regenerate it. In this case if you do not reset the date it may also delete all the records in other replicated databases. Resetting the date will cause the dropped table to be regenerated.

Viewing Database Information

The OpenBase Manager allows you to view and edit database table information using the Database Viewer. To do this select a started database (one with a green dot) and press the View Data button. This will bring up the Data Viewer window:



• Figure 8: Data Viewer Window

Data Viewer Window

The table names listed on the left of the Data Viewer window offers an easy mechanism for viewing choosing what information you want to view. Selecting a table will show the data for that table on the right.

The Data Viewer allows you to edit as well as view table content. As long as the record you select hasn't been locked by another user, you can double click to select and change information. The data viewer

keeps two people from editing the same row of information so that your changes will not be overwritten by other users.

If you are interested in seeing OpenBase system tables in addition to your own tables, you can check the show all tables checkbox. When you do this the table list is refreshed to include all of the system tables. **We strongly recommend that you do not try to edit or change any of the information in the system tables directly.**

The columns on the data viewer may be reorganized allowing you to view information in a different order. You can change the position of a column by pointing to the column's title, pressing and holding down the mouse button, and dragging the title to its new location. When you let go of the mouse, the columns will reorganize and the SQL will change in the field at the bottom of the window.

The field entitled Maximum Fetch at the top right enables you to specify the maximum number of rows you wish to be returned. Pressing the Execute-SQL button will refresh the display using the new maximum.

Selecting on the column titles at the top of the Data Viewer will bring up a search panel giving you the ability to narrow the search results or specify sort ordering.

• Figure 9: Data Viewer Search Window

The Hide button on the search panel allows you to hide the selected column from the result. Pressing Cancel will return you to the Data Viewer and will leave the SQL unchanged.

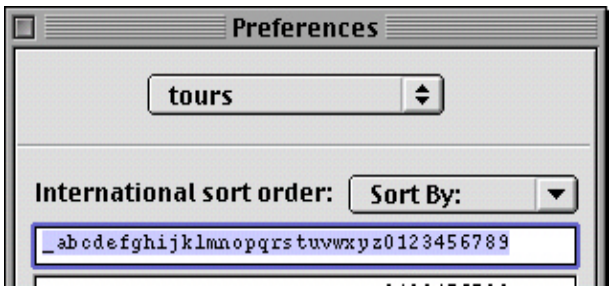
The second tab on the Data Viewer window allows you to create custom queries. This is useful if you have a select that you need to perform frequently. All you need to do is enter your SQL in the space provided (or choose a template to edit) and name it something you will remember. Only select statements are allowed. When you go back to

the Data Viewer, your query will show up in the box just below the table list. Selecting the title will run the query.

Finally, the last tab on the Data Viewer window gives you the ability to send any SQL statement to the database. If you are using MacOS X Server or OpenStep you should use openisql for this function. However, this panel is useful for Windows NT users who do not have a command line tool.

Preference Panel

The OpenBase Manager has a preferences panel to personalize your view of date, time, and money data types throughout all of your applications, as well as set various parameters for your OpenBase databases. For many of these parameters to take affect, databases must be restarted. To display the preferences panel, select the Preferences menu item located under Info in your main menu.



• **Figure 10: Preferences Panel**

Setting Preferences

Preferences need to be set for each of the database server hosts on your network. By selecting the desired host name on the Server Host popup you can view and set the preferences for each host. The preference settings are described as follows:

Cleanup before exit

Configures the server to go through a database cleanup process before stopping the database under normal conditions. Checking this option reduces your start-up time if the database needs to do recovery.

Log SQL to file

Tells locally running servers to log SQL commands to a log file located in the directory specified in the Log File field. The generated log file, combined with a backup of the database, can be used to roll forward backup databases in case of hard disk failures.

Date, Time, and Money

Specifying preferences for date, time and money will change the default format for databases on the specified host. If you would like to display several different formats at the same time in your program, you will need to change these formats programmatically. The datetime type only uses the default NeXT calendar date format, so it is not listed on this panel.

Localized Sorting

Since different languages have different characters and sorting rules, the international sort order section provides a way to specify upper and lower case letters in the natural sort order. The top text field should contain the set of lower case letters in order of their importance and the bottom text field should contain capital letters in the same order. Letters that are not specified in this list will assume their ASCII value and in a sorted list will appear after the letters in the text fields.

Change Password

For added security you can specify a password for each database server host. In order to start, stop, duplicate, configure or add databases you must have the host password (if one exists). To set the password for the host selected in the popup list press the Change Password button.

Interactive SQL

OpenISQL allows you to execute SQL from a terminal window. Pressing the Interactive SQL Terminal menu item on the Tools menu will launch the OpenISQL program and automatically connect it to the selected database. Please refer to the chapter on OpenISQL for more information on how to use this program.

• Figure 11: ISQL Window

Backup, Restore and Script Functions

The best way to backup your databases is to make copies of them. We recommend that you make regular backup copies of your databases located in /usr/openbase/Databases (OpenBase/Databases on WindowsNT computers). If you need to restore a database, stop the database, remove the database's work directory in /usr/openbase/work and replace the database files with the backup.

The OpenBase manager also provides some tools for transferring database information to and from a bulk saved ASCII format. The Backup to ASCII and Restore from ASCII functions are located on the Tools menu. These functions transfer all information in the database including archived Objects and BLOBs. It does not save user accounts or permissions.

Also included on the Tools menu is a function for executing OpenISQL scripts. These can either be schema upgrade scripts generated by the Edit Database Schema panel or it can be an ASCII file with a list of OpenISQL commands.

4

OpenBase SQL Language

In this chapter we will discuss the Standard Query Language (SQL), the means by which client applications communicate with database server programs and to lay the groundwork for understanding the Objective-C API. This chapter is only meant to give you a basic understanding of how SQL can be used to create, read and manage database information. For more in depth information we recommend that you refer to a book on ANSI SQL.

SQL can be entered and executed using the OpenISQL program described later in this manual. Please use OpenISQL for experimenting with the SQL described in this chapter.

This chapter contains the following sections:

SQL Standards

First we would like to dispel a myth about SQL standards. These days a lot of emphasis is placed on SQL syntax standards. The problem is that the standardization stops with the easiest part -- the SQL syntax.

Every database server uses different communication mechanisms, embedded SQL, return codes, library calls, and data retrieval techniques. Some provide dynamic SQL mechanisms, allowing your application to build the SQL dynamically (i.e. OpenBase and Sybase), while others require a precompiler that converts the static SQL in your applications to some other form. Every server seems to be completely different, so the fact that they support the ANSI standard syntax, for instance, does not automatically make your application portable to other systems.

Enterprise Objects Framework has standardized the interface to make applications portable, yet at the same time isolating them from the SQL syntax. This is a more important standard because it solves the problem of portability between back-end servers.

Please note that while OpenBase supports standard SQL, we have taken the liberty to deviate from the standards in order to more closely match the needs of customers.

SQL Statements

SELECT...FROM

Select statements can be the most complicated of all SQL statements. It is important to note that select statements are the only statements that will operate on multiple tables at the same time. For this reason, select statements support a slightly different, although standard, SQL syntax. This will be explained further when we talk about joins between tables, but let's first examine a simple SELECT statement.

```
SELECT CATEGORY, TITLE, REVENUE FROM MOVIE WHERE REVENUE
> 10000000
```

SELECT is a keyword that specifies that the following will be a list of information to be returned by the search.

CATEGORY specifies that the MOVIE CATEGORY is the first column of the result.

TITLE specifies that the MOVIE TITLE is the second column of the result.

REVENUE specifies that the MOVIE REVENUE is the third column of the result.

FROM is a keyword that tells the database that the following will be a list of tables.

MOVIE specifies that these columns come from the MOVIE table.

WHERE is a keyword that specifies that the following will be a list of search conditions.

REVENUE > 10000000 specifies that we want all of the movies in the database that have a REVENUE of more than $10,000,000.

Select statements like the one shown above provide an easy way to retrieve specific sets of information from the database. This first example only retrieves information from a single table. However, it is often necessary to retrieve data from several related tables at the same time. To do this we use joins.

Joins

A join is necessary when you want to view two tables with related information at the same time.

Using the Movies database you can display each movie title along with the studio name from the STUDIO table. The relationship can be made using the STUDIO_ID columns in each of the tables because each MOVIE's STUDIO_ID value points to a corresponding STUDIO_ID value in the STUDIO table. This enables the server to match movies with their studios.

```
SELECT mov.TITLE, mov.CATEGORY, stu.NAME FROM MOVIE mov,
STUDIO stu WHERE mov.STUDIO_ID = stu.STUDIO_ID
```

This example uses the table aliases mov and stu to tell the server which table the TITLE, CATEGORY, and STUDIO columns come from. Table aliases or table names should be used whenever performing joins.

Inner & Outer Joins

The example above uses an inner join to match records in the MOVIE table with records in the STUDIO table. Inner joins only return complete matches, so movies with no matching studio will be removed from the result. However, outer joins include records even when there is no match.

OpenBase supports left outer joins, meaning that records from tables specified first are included even when they cannot be matched with records from tables on the right. An outer join is specified by using the * operator in place of the = operator in the WHERE statement. Here is an example:

```
SELECT mov.TITLE, mov.CATEGORY, stu.NAME FROM MOVIE mov,
STUDIO stu WHERE mov.STUDIO_ID * stu.STUDIO_ID
```

The order of the tables in the FROM clause determine the direction of the outer join. Following is a more advanced example of a join.

```
SELECT mov.TITLE, mov.CATEGORY, tal.FIRST_NAME,
tal.LAST_NAME, rol.ROLE_NAME FROM MOVIE mov, TALENT tal,
MOVIE_ROLE rol WHERE mov.MOVIE_ID = rol.MOVIE_ID AND
rol.TALENT_ID = tal.TALENT_ID AND tal.FIRST_NAME ? 'Jon'
ORDER BY mov.CATEGORY, mov.TITLE
```

This complex query joins three tables together and returns each movie role, who played the role and the movie the role was played in. The last constraint narrows the query even more by only returning rows where the actor's firstname sounds like Jon. A more detailed explanation follows.

```
SELECT mov.TITLE, mov.CATEGORY, tal.FIRST_NAME,
tal.LAST_NAME, rol.ROLE_NAME
```

This specifies the columns of the result. Each column name is preceded by a table alias so the database will know which table each value comes from.

```
FROM MOVIE mov, TALENT tal, MOVIE_ROLE rol
```

This specifies the tables and table aliases used in the search. The aliases tal and rol are optional as long as you specify the full table name in front of each column in the query.

```
WHERE mov.MOVIE_ID = rol.MOVIE_ID
```

This constraint joins the movies in the MOVIE table to roles located in the MOVIE_ROLE table.

```
AND rol.TALENT_ID = tal.TALENT_ID
```

This joins each movie role with a talent. Failing to include these constraints will have a multiplication effect on the results, so it is very important to define how the tables relate to one another.

```
AND tal.FIRST_NAME ? 'Jon'
```

This constraint narrows the search by only returning result sets where the actor's FIRST_NAME sounds like Jon. OpenBase supports a modified soundex function designated by the operator '?'. This is one of those OpenBase specific extensions to SQL that we mentioned earlier in this chapter.

```
ORDER BY mov.CATEGORY, mov.TITLE
```

This simply orders the result rows by CATEGORY and TITLE.

Derived columns

Derived columns return calculated values or concatenated strings. Normally derived columns do not map directly to columns in the database, but they often use column values. As with regular result columns, derived columns are expressed after the SELECT keyword.

For instance, the following SELECT statement concatenates the FIRST_NAME, a space and the LAST_NAME of each record in the TALENT table.

```
SELECT FIRST_NAME+" "+LAST_NAME FROM TALENT
```

Your output might look like this:

Harrison Ford
George Lucas
Mark Hammil
Marlon Brando
Dianne Keaton
Francis Ford Coppola
Al Pacino
Humphrey Bogart

Another example uses a function to return only the first 3 characters of each concatenated name. A variety of functions are discussed later in this chapter.

```
SELECT LEFT(FIRST_NAME+" "+LAST_NAME, 3) FROM TALENT
```

Using the LEFT() function your output might look like this:

Har
Geo
Mar
Mar
Dia
Fra
Al
Hum

Another example calculates 10% tax on the revenue of each MOVIE in the Movie database:

```
SELECT (REVENUE * 0.10) FROM MOVIE
```

In this case your output might look like this:

$1,440,000.00
$20,000.00
$30,000.00
$20,000.00

You may also mix string concatenations with value calculations as follows:

```
SELECT “tax = “]; (REVENUE * 0.10) FROM MOVIE
```

In this case the calculation is done before the string concatenation. Here is some example output:

tax = $1,440,000.00
tax = $20,000.00
tax = $30,000.00
tax = $20,000.00

Derived columns can be extremely useful for changing the form of data returned from the database and providing summary information for database records.

The FROM clause explained

The FROM clause in reference to the SELECT statement specifies the tables that will be used in the search. Using the FROM clause, you can also specify table abbreviations to be used when joining tables. Following are some examples of FROM clauses:

FROM MOVIE

In this case, all columns in the query are from the purchase table and no abbreviations are needed. However, the following example shows the definition of table abbreviations.

FROM MOVIE mov, STUDIO stu

In this example, two tables are specified and abbreviations are specified. These abbreviations should be used throughout the query to

tell OpenBase which table each column belongs to. When joining tables you should specify and use table aliases.

The WHERE clause

The where clause specifies the scope of the action to be performed by the SQL statement. The following simple expressions use the letters A and B to denote column names, values or other expressions.

Expression	Description
A = B	include record if A and B have the same value.
A != B	include record if A is not equal to B.
A < B	include record if A is less than B
A <= B	include record if A is less than or equal to B
A > B	include record if A is greater than B
A >= B	include record if A is greater than or equal to B
A? B	include record if A sounds like B
A IN (B)	include record if A is in the list of values denoted by B. In this case B may be a comma separated list of values or a subselect.
A NOT IN (B)	include record if A is NOT in the list of values denoted by B. In this case B may be a comma separated list of values or a subselect.
EXISTS (B)	include record if B exists, where B is a correlated select statement which returns one or more records.
A LIKE B	include record if A matches the wild card pattern B

You may include the keyword NOT before each of the previous expressions to negate it. For instance, if you don't want A to equal B, you could write the expression:

```
NOT A = B
```

Operators can be strung together to produce multifaceted sets of criteria by using the AND and OR keywords. Here is an example of two criteria, both of which must be true for each record in the result.

```
FIRST_NAME ='Peter' AND LAST_NAME > 'P'
```

The EXISTS and NOT EXISTS keywords allow you to easily perform queries which list rows that do not match with records in another table. For instance, perhaps you want to select records in the MOVIE table where correlated records do not exist in the STUDIO table. The following example will list all movies which do not have studios on file.

```
SELECT * FROM MOVIE mov WHERE NOT EXISTS (SELECT
STUDIO_ID FROM STUDIO stu WHERE mov.STUDIO_ID =
                stu.STUDIO_ID)
```

In this example, rows are only returned when matching records are not found in the subselect.

While the WHERE clause can be used with SELECT, UPDATE, INSERT, and DELETE queries, the SELECT query is the only one that operates on multiple tables. However, all SQL operations support subqueries which can be used to narrow a search using multiple tables.

The ORDER BY clause

The ORDER BY clause will order records in ascending order by default. The following ORDER BY statement incorporates the ASC (for ascending order) and DESC (for descending order) keywords to tell the server what order you would like to sort the results in.

```
ORDER BY LAST_NAME DESC, FIRST_NAME ASC
```

In this case your output might look like this:

Zanin	Bruno
Young	Sean
Yoba	Malik
Wright	Robin
Wood Jr.	Edward D.
Winters	Shelley

Remember that if you are joining tables you will need to specify a table name or abbreviation for each column listed. Here is an example:

ORDER BY tal.LAST_NAME, tal.FIRST_NAME

Value Conversion Functions

Value conversion functions provide a method to convert values between SQL types or control the return type of calculations.

TOCHAR(value)

Converts value into a character string. The value parameter can be of any type.

TODOUBLE(value)

Converts value to a double value. The value parameter can be of any type.

TOINT(value)

Converts value to an integer value. The value parameter can be of any type.

TOLONG(value)

Converts value to a long value. The value parameter can be of any type.

TOLONGLONG(value)

Converts value to a long long value. The value parameter can be of any type.

TOMONEY(value)

Converts value to a money value. The value parameter can be of any type.

String Manipulation Functions

This section describes the SQL functions that can be performed on strings. These include:

LENGTH(string)

Returns the length of *string*.

Example: Bring back all of the movies which have a title longer than 10 characters.

```
SELECT LENGTH(TITLE), TITLE FROM MOVIE WHERE
LENGTH(TITLE) > 10
```

INDEXOF(string, substring)

Returns the numerical index of *substring* in *string* or -1 if the *substring* cannot be found.

Example: Select all movie titles which have the word 'the' in the title.

```
SELECT INDEXOF(TITLE,'the'), TITLE FROM MOVIE WHERE
INDEXOF(TITLE,'the') >= 0
```

REPLACE(string, startpos, length, replacestring)

Inserts *replacestring* into *string* at *startpos* and replaces *length* characters. The replacestring is not truncated by *length*. The entire replacement string is inserted at the specified point.

Example: Select all movie titles and replace the first word with 'BLANK'

```
SELECT REPLACE(TITLE, 0, INDEXOF(TITLE,' '), 'BLANK '),
TITLE from MOVIE where INDEXOF(TITLE,' ') >= 0
```

SUBSTRING(string, startpos, length)

Returns the substring of *string* starting at *startpos* and including *length* characters.

Example: Select the three first characters of all movie titles.

```
SELECT SUBSTRING(TITLE,0,3) from MOVIE
```

UPPER(string), LOWER(string)

Converts *string* to upper or lower case.

Example: Select movie titles in upper and lower case.

```
      SELECT UPPER(TITLE) from MOVIE
```

```
      SELECT LOWER(TITLE) from MOVIE
```

RIGHT(string, length)

Returns the right portion of *string* with *length* characters.

Example: Select the right 5 characters of all movie titles.

```
SELECT RIGHT(TITLE,5) from MOVIE
```

LEFT(string, length)

Returns the left portion of *string* with *length* characters.

Example: Select the first 5 characters of all movie titles.

```
SELECT LEFT(TITLE,5) from MOVIE
```

TRIM(string)

Removes extra blank spaces from *string*.

```
SELECT TRIM(TITLE) from MOVIE
```

PROPER(string)

Capitalizes the first letter of each word in *string*.

Example: Show all movie titles with proper capitalization.

```
SELECT PROPER(TITLE) from MOVIE
```

IF(condition, returnValueIfTrue, retValueIfFalse)

Returns conditional values depending on the expression *condition*. All values and expressions can be made up of multiple elements.

Example: Return the movie title or the string 'Smaller Than 10' for movie titles that are shorter than 10 characters.

```
SELECT IF( (LENGTH(TITLE) < 10), 'Smaller Than 10', TITLE
MOVIE) FROM MOVIE
```

CHOOSE(number, value1, value2,...).

Returns the value in the location indicated by number. Values may be constants, column names, expressions or any combination of these.

Example: Return 'value1' for each of the movie titles. 1 could be replaced by a calculation or database column.

```
SELECT CHOOSE(1, “value0”, “value1”, “value2”), TITLE
FROM MOVIE
```

Aggregate Functions and GROUP BY

Aggregate functions and the optional GROUP BY clause provide ways to retrieve summary information about table content. If no GROUP BY clause is specified in the SQL statement, aggregate functions will return a single summary row for all information satisfying the WHERE constraint. Otherwise, the information will be placed in groups based on a common set of values, and a summary line will be generated for each.

The aggregate functions provided in this release are as follows:

Aggregate Function	Description
count(*)	Returns the number of rows returned for each group.
sum(columnName)	Returns the sum of the column columnName for each group.
avg(columnName)	Returns the average of the column columnName for each group.

Aggregate Function	Description
min(columnName)	Returns the minimum value of column columnName for each group.
max(columnName)	Returns the maximum value of column columnName for each group.

The GROUP BY clause allows you to group results with common values. It works similarly to the ORDER BY clause, in that columns are specified and separated by commas. Here is an example:

```
SELECT count(*) MOVIES, sum(mov.REVENUE), stu.NAME
STUDIO_NAME, mov.CATEGORY FROM MOVIE mov, STUDIO stu
WHERE mov.STUDIO_ID = stu.STUDIO_ID AND mov.RATING = 'R'
GROUP BY stu.NAME, mov.CATEGORY
```

In this example, only studios with a movie rating of ‘R’ are returned. Results are then grouped by the studio name and movie category. Finally, each group is converted into a single row of summary information. This example also demonstrates how you can specify columns from the GROUP BY section in the select statement to give the result more meaning.

The GROUP BY clause should be specified after the WHERE clause and before the ORDER BY clause (if you use these optional clauses).

Subqueries

Subqueries are useful when you want to include values from another table in a search without creating a direct join to the table. This might be the case if you are spanning a relationship to a second table only to qualify records in the first table.

Subqueries are entered between parentheses and normally appear in the where clause. However, they can also be used as a data source for any function parameter.

Example: In what movie did they have a character referred to as 'The Godfather'?

Here is a subquery that uses the IN operator to answer this question:

```
select TITLE from MOVIE where MOVIE_ID in (select
t2.MOVIE_ID from MOVIE_ROLE t2 WHERE t2.ROLE_NAME = 'The
Godfather')
```

The above example is referred to as a non-correlated subquery because the select in the subquery is only done once for all records searched in the movie table. In other words, the subquery does not have a qualifier that refers back to the parent select statement. The subquery can stand on its own. However, this is not true for the next example.

Here is a statement that uses the EXISTS operator and a correlated subquery to answer this question:

```
select TITLE t1 from MOVIE t1 where EXISTS (select
t2.MOVIE_ID from MOVIE_ROLE t2 where t2.ROLE_NAME = 'The
Godfather' and t1.MOVIE_ID = t2.MOVIE_ID)
```

This is a correlated subquery because of the t1.MOVIE_ID = t2.MOVIE_ID in the subquery. t1 is an alias for the MOVIE table in the parent select. For each record searched in the MOVIE table the subselect needs to be re-evaluated to see if records exist. For this reason correlated subqueries are generally less efficient than non-correlated subqueries. However, they can be very useful in some circumstances.

Inserting Database Information

INSERT... INTO

The INSERT statement is used to create new records and set initial field values. In terms of an application, the insert statement might be used when the user presses a New button. Here is the basic format of the INSERT statement:

INSERT INTO <table> (<field1>, <field2>...) VALUES (<value1>, <value2>...)

Here is an example of an INSERT statement with a detailed description of each part:

```
INSERT INTO TALENT (FIRST_NAME, LAST_NAME, TALENT_ID)
VALUES  ('John','Smith', 700)
```

INSERT INTO TALENT - tells the database that you are inserting a group of values into the TALENT table.

(FIRST_NAME, LAST_NAME, TALENT_ID) specifies the fields or table columns that you are inserting.

VALUES tells the database that the following values correspond to the fields just specified.

('John', 'Smith', 700) are the values that correspond to and will be inserted into the FIRST_NAME, LAST_NAME and TALENT_ID columns.

Updating Database Records

UPDATE...SET

The UPDATE statement is used to update values in the database. Update statements have the ability to update multiple records at a time depending on the search constraints given.

Here is an example of an UPDATE statement:

UPDATE <Table Name> SET <field1> = <value1> [,<field2>=<value2>...] where <search conditions>

The following example identifies a single record by its _rowid column and changes the FIRST_NAME value to Bill.

```
UPDATE TALENT SET FIRST_NAME = 'Bill' WHERE _rowid = 100
```

UPDATE is a keyword that tells the database that you want to update the values in existing records.

TALENT is the name of the table whose records will be updated.

SET is a keyword that tells the database that the following list will be a list of columns and corresponding values.

FIRST_NAME = 'Bill' means that every record in the result will have its first name set to Bill.

WHERE is a keyword that specifies that there will be a list of search conditions.

_rowid = 100 is the search condition that identifies a single record by its _rowid column.

The following example updates multiple records with one SQL statement. Let's say that the rating has been changed from G to PG for all movies with a drama category. The LANGUAGE field in this example does not have to be updated (let's say that 100 is the code for English Language), but we update it anyway to demonstrate that multiple changes can be specified by separating them with commas.

```
UPDATE MOVIE SET RATING='PG', LANGUAGE = 100 WHERE
CATEGORY = 'Drama' AND RATING = 'G'
```

UPDATE is a keyword that tells the database that you want to update the values in existing records.

MOVIE is the name of the table whose records will be updated.

SET is a keyword that tells the database that the following list will be a list of columns and corresponding values.

RATING='PG', means that every record in the result will have its RATING column changed to PG.

LANGUAGE= 100 means that every record in the result will have its LANGUAGE column value changed to 100.

WHERE is a keyword that specifies that there will be a list of search conditions.

CATEGORY = 'Drama' AND RATING = 'G' tells the database that you are looking for records whose CATEGORY is 'Drama' and RATING is 'G'.

Deleting Database Records

DELETE FROM

DELETE FROM provides a way to remove records from a table. Deleting records from a table works similarly to updating in that the WHERE clause specifies which records will be affected.

Following is the general format for the DELETE statement:

DELETE FROM <Table Name> WHERE <conditions>

Here are a few examples of how the DELETE statement can be used:

```
DELETE FROM TALENT WHERE _rowid = 100
```

```
DELETE FROM TALENT WHERE FIRST_NAME = 'Bill'
```

You may also want to delete a set of records that have common characteristics. For instance, the following SQL statement will delete all records with a CATEGORY of Drama and RATING of PG.

```
DELETE FROM MOVIE WHERE CATEGORY = 'Drama' and RATING =
'PG'
```

This concludes our overview of the basic set of SQL statements. Next we will discuss some more general SQL topics.

Expressing String Values

There are a few different ways to express string values in your SQL statements. Traditionally, an SQL string must be enclosed in single or

double quotes. Here is an example using a table that keeps track of messages:

```
insert into messages (note) values ('Joe Called')
```

The string 'Joe Called' is enclosed in quotes to tell OpenBase that it is a string value. However, sometimes you will need to store a string that has quotes in it. You have the choice of handling this in one of a few ways. Here is a description of the standard way:

```
insert into messages (note) values ('Joe said he can\'t call back')
```

or

```
insert into messages (note) values ('Joe said he can''t call back')
```

This method requires you to programmatically insert a backslash '\' or a second quote for each instance of a quote in your string that matches the boundary quotes.

Creating Tables

CREATE TABLE

The CREATE TABLE keyword at the beginning of an SQL statement tells OpenBase that you want to create a new table. Creating new tables can also be accomplished by using the graphical tools provided with the OpenBase Manager. However, this section will give you an idea of what happens on the SQL level.

The general format of the create statement is as follows:

CREATE TABLE <Table Name> (<field> <type> [NOT NULL]
[[CLUSTERED] [UNIQUE] INDEX] [default '<value>']
[REFERENCES <table>.<column>], ...)

NOT NULL

The NOT NULL clause makes sure that the column always has a value when inserting records. Any insertion that doesn't include a value for the column will fail.

NULLs should not be confused with empty strings. An empty string ('') is a value where as a NULL is the absence of a value.

INDEX and UNIQUE INDEX

Indexes are necessary to provide better performance when joining tables or searching information. The UNIQUE keyword indicates that the column value must be unique in order to be inserted into the table.

CLUSTERED INDEX and CLUSTERED UNIQUE INDEX

Adding the CLUSTERED keyword to your index definition arranges the physical data in the table by the indexed column. This provides significant performance benefits in cases where data is accessed through a foreign key or is sorted by the specific column. By clustering the column the database is able to find records likely to be accessed together on consecutive pages. Only one column per table can be used with clustering.

REFERENCES table.column

The REFERENCES keyword provide referential integrity checking between tables. Records in the target table will not be deleted if they are referenced by records in the table with the references clause. Deleting records that are referenced by another table will return an error.

Here is an example of creating a table called customer with fields number, name, and balance:

```
CREATE TABLE customer (number longlong NOT NULL UNIQUE
INDEX, name char(30) NOT NULLDEFAULT 'yourName',
balance money)
```

The numbers that are specified for the character lengths in the above example (name char(30) means the string length is 30) will be enforced by the database server. All character lengths are assumed to be between 0 and 1024 characters. If you want to store text information longer than 1024 you will need to use an Object column.

Following is a list of data types and their Objective-C equivalents, supported by OpenBase:

SQL type	C type	Description
char(n)	(char *)	String of n characters
varchar	(char *)	Variable length string
float	(double)	Double value
int	(int)	Integer
long	(long)	Long integer
longlong	(long long)	Long Long
money	(long long)	Money
date	(char *)	Date
time	(char *)	Time
datetime	(double)	EOF datetime
object	(void *)	Object

All values may be expressed and retrieved as strings. OpenBase will automatically convert the data to the appropriate representations.

The object data-type is used to store BLOB (Binary Large OBjects) ids for image or other binary data. Object fields have a special mechanism that will automatically erase old BLOBs when updated with a new BLOB id. When records are deleted, the database will also automatically remove associated BLOBs.

In addition to the column's data-type, column specifications within create statements can include NOT NULL, INDEX, UNIQUE INDEX, and DEFAULT <value>. Each of these are described as follows:

NOT NULL tells the server that the column must always have a value. Trying to insert a row without specifying a value for this column will result in an error. Likewise, updating that column and setting it to NULL will also generate an error. The NOT NULL option provides a safeguard so that critical data is not absent.

INDEX and UNIQUE INDEX indicate whether the column should be indexed. If UNIQUE INDEX is used, the server will ensure that the values of each row is unique. In this case, inserting an existing value will cause an error.

DEFAULT <value> is used when you want the column to have a default value when it is not specified in an insert statement. The default value is used instead of NULL for unspecified columns.

CREATE VIEW

CREATE VIEW statements provide a way to specify database views. A database view is a phantom table which includes data from one or more tables. Views are normally used to provide a way of flattening out complex relationships and queries so that the data appears like it is coming from a single table.

In the SELECT... FROM section of this chapter we discussed table aliasing. An alias is simply an abbreviation of the table that is defined in the FROM portion of a select statement and used before each column belonging to the aliased table. Aliasing tables is sometimes optional. However, when defining views it is important to always use aliases.

CREATE VIEW is defined as follows:

CREATE VIEW <view name> (view-colum1 [, view-column2, ...]) AS SELECT <column specification> FROM <table specification> [WHERE <joins and constraints>] [ORDER BY <column>]

CREATE VIEW defines a lightweight table which is defined by the select statement after the keyword AS.

The following CREATE VIEW statement, which uses the Movies database, shows how to create a view that combines the data in two tables.

```
CREATE VIEW TalentMovie (fullname, first, last, role) AS
SELECT t.FIRST_NAME +' '+t.LAST_NAME, t.FIRST_NAME,
t.LAST_NAME, r.ROLE_NAME FROM MOVIE_ROLE r,
TALENT t WHERE t.TALENT_ID = r.TALENT_ID
```

TalentMovie is the name of the new view. The view name must be different than those of previously defined tables and views.

(fullname, first, last, role) specifies the view columns. Each of these maps to the columns specified in the select statement. The fullname column is derived by concatenating the FIRST_NAME and LAST_NAME columns in the TALENT table.

Views make it easy to flatten complex sets of information into a table-like form. While you are not restricted as to how the tables underneath the views may be updated and changed, with views you can only update existing records. Update statements referring to a view must only update columns that map to columns from a single table. In the above example, for instance, you can not update FIRST_NAME and ROLE_NAME in the same UPDATE statement.

Selecting information in a view is similar to selecting information in any other table. Here is an example of how to select the information from the view created above:

```
SELECT fullname, first, last, role FROM TalentMovie ORDER
BY last
```

This select statement will return all the talent names and their roles. It is important to understand that the TalentMovie defines a window into the TALENT and MOVIE_ROLE tables of the MOVIE database. All modifications to data in these tables will be reflected when selecting data from the view.

Creating Indexes

CREATE [UNIQUE] INDEX

The CREATE INDEX and CREATE UNIQUE INDEX commands are used to create indexes on columns. Indexes are necessary to provide better performance when joining tables or searching information. The syntax is defined as follows:

CREATE [UNIQUE] INDEX <table> <column>

Here are two examples using the MOVIE database:

```
CREATE INDEX TALENT LAST_NAME
```

```
CREATE UNIQUE INDEX MOVIE_ROLE ROLE_NAME
```

CREATE CLUSTERED [UNIQUE] INDEX

Adding the CLUSTERED keyword to your index definition arranges the physical data in the table by the indexed column. This provides significant performance benefits in cases where data is accessed through a foreign key or is sorted by the specific column. By clustering the column the database is able to find records likely to be accessed together on consecutive pages.

Only one column per table can be used with clustering. Setting the clustered column replaces any previous setting.

You can also set the clustered column using the OpenBase Manager Schema Window. The popup entitled Natural Order allows you to set the clustered index.

Dropping and Renaming Tables

DROP TABLE

OpenBase supports dropping and renaming tables. The syntax for dropping a table is as follows:

```
DROP TABLE tableName
```

This will remove the table tableName from the database. In other cases you may want to rename a table. The following command will rename tableName to newName:

```
RENAME tableName newName
```

We recommend that you use the OpenBase Manager schema editing tools for dropping and renaming tables.

DROP VIEW

The syntax for dropping a view is as follows:

```
DROP VIEW TalentMovie
```

This will remove the view TalentMovie from the database.

Changing Database Schemas

ALTER TABLE

There are two types of alter table commands. Some are fairly light weight and do not require the database to rebuild it's pages on the disk. Changing a column name, or column parameters are good examples and are fairly easy for the database to do. Another type, however, requires the database to do more work. Examples include, adding columns, removing columns or changing the type of columns. The good news is that you can include all of your changes in a single

command statement to avoid making the database perform multiple passes.

To change the name of a column you can perform ALTER TABLE using the rename feature. The following command will rename the column FIRST_NAME to FIRSTNAME.

```
ALTER TABLE TALENT RENAME FIRST_NAME TO FIRSTNAME
```

You can also change the parameters on a column, the following examples show how you can assign default values for columns and change some of the other column parameters.

Sometimes you may want to assign a default value for a previously defined column. The following example demonstrates how this is done. When the column value is not specified by subsequent inserts, the column value will be set to the default value.

```
ALTER TABLE MOVIE COLUMN RATING SET DEFAULT 'PG'
```

The following example shows how you can set the length of a character string column. The length is enforced for future inserts and updates, but altering the length will not truncate existing data.

```
ALTER TABLE MOVIE COLUMN TITLE SET LENGTH 50
```

The following example shows how to make sure required column values are always specified.

```
ALTER TABLE MOVIE COLUMN RATING SET NOT NULL
```

The following command reverses the SET NOT NULL shown above, allowing the column to contain NULL values again.

```
ALTER TABLE MOVIE COLUMN RATING SET NULL
```

The following two commands show how to create and remove indexes on a column. You can also use create index to do the same thing.

```
ALTER TABLE MOVIE COLUMN CATEGORY SET INDEX
```

```
ALTER TABLE MOVIE COLUMN RATING SET NO INDEX
```

You can also create a unique index on columns.

```
ALTER TABLE MOVIE COLUMN MOVIE_ID SET UNIQUE INDEX
```

Adding, removing and changing the type of columns are more time consuming than the ALTER TABLE commands discussed above.
The reason is that they require the database to translate data between the old and new data structure.

The following example adds two columns, removes two columns, and changes the parameters for the name column. The ADD COLUMN section should be a list of fields as they would appear in a create statement. If columns listed in this section are already in the table, the characteristics of the columns are replaced and the data is converted.

```
ALTER TABLE MOVIE ADD COLUMN COUNTRY CHAR(20) DEFAULT
'USA'.COUNTRY_ID INT NOT NULL, DIRECTOR VARCHAR NOT NULL
INDEX DEFAULT 'Unknown' REMOVE COULMN RATING
```

When ADD COLUMN and / or REMOVE COLUMN are used, the table must be rebuilt by the server. This may take awhile if you are modifying a large table.

Finally, ALTER TABLE can be used to set buffer length defaults for each table. This can be used to fine tune performance by decreasing buffer sizes for tables that are used less frequently and increasing buffer sizes for tables that are used more frequently. If you don't have a need to change these numbers, we recommend that you leave them alone.

Data buffers are counted in terms pages which may contain a maximum of 128 records per 20K page (depending on the size of the records). The number of memory based data pages are 100 by default. By increasing this value you force the server to buffer more of the data in memory and improve performance. The data page value can be set as follows:

```
ALTER TABLE MOVIE SET DATA_PAGES 200
```

Changing User Access

GRANT... ACCESS... TO

GRANT... ACCESS... TO provides a way to grant and revoke permission for users to access tables. For your convenience, all users have access to all tables before any permissions have been assigned. When a specific user is assigned access to a table, all other users which had access previously by default are assumed not to have access. **If you are not concerned about security, we recommend that you leave the permissions alone.**

The OpenBase Manager includes graphical tools that make it easy to edit and maintain user permissions. However, for those applications that require a knowledge of how permissions are granted and revoked, we have included this section. Please note that GRANT... ACCESS... TO is not part of the ANSI SQL standard. Here is an example:

```
GRANT joe, fred ACCESS +su-di TO MOVIES, TALENT
```

GRANT... ACCESS... TO will assign the specified access to the users and tables listed in the statement. The access portion is specified as a string of characters as shown above. They are defined as follows:

s SELECT access gives the ability to select data from the specified tables.

u UPDATE access gives the ability to update data in the specified tables.

i INSERT access gives the ability to insert data into the

specified tables.

d DELETE access gives the ability to delete data from the specified tables.

The '+' and '-' which precede the letters tell whether to revoke access (-) or grant access (+). If you wish to assign one type of access without altering setting for other types of access, only include the access letters which you wish to change.

```
GRANT staff, marketing ACCESS +sui-d TO MOVIES
```

This example shows how you can grant and revoke access to groups of people. In this case, the staff and marketing groups have been given access to read and change data but not to delete.

The group permissions do not overwrite the permissions for individuals. Users are allowed to perform inserts, updates, selects and deletes on tables where either the user or the user's group has been granted access.

This concludes our discussion on basic SQL and OpenBase specific commands. In the next chapter we will talk about transaction management.

5

Transaction Management

In this chapter we will discuss the OpenBase SQL syntax for and concepts behind, transactions.

This chapter contains the following sections:

Transaction Overview

Transactions provide a mechanism for managing updates in multi-user environments. This is especially important when client programs must complete a series of update, inserts and deletes independently from other users or risk leaving the database in an inconsistent state.

The START TRANSACTION command marks the beginning of a transaction, and a COMMIT finalizes changes. Inserts, updates and deletes performed between these commands may be undone if a problem is detected by the application. This undo feature is accomplished by sending the database a ROLLBACK command in the place of a COMMIT.

The server will automatically rollback a transaction if the server loses communication with the client during a transaction (for example, if the client crashes). This serves as a safeguard to ensure that the database is never left in an inconsistent state.

OpenBase transactions are interactive with applications, allowing applications to control the direction of the transaction based on return codes. Changes to the database become visible to the client immedi-

ately. Transactions can not include any schema modifying SQL, such as ALTER TABLE or CREATE TABLE statements. The reason is that these operations can not be rolled back in OpenBase.

A couple of other OpenBase specific commands are discussed in this chapter. They are, LOCK RECORD, UNLOCK RECORD, FOR UPDATE and WRITE TABLE. These commands can be used for a variety of purposes within transactions to ensure data integrity in a multi-user environment.

The examples used in this chapter show transaction commands being sent to the server using OpenISQL. However, the same SQL will work from your applications.

The OpenBase API includes functions for starting, rolling back and committing transactions for your convenience. However, the same thing can be accomplished by issuing the SQL described below.

Starting a Transaction

START TRANSACTION

Starting a new transaction is accomplished by sending the start transaction SQL to the database. You can do this from OpenISQL by typing the following:

```
openbase 1> START TRANSACTION

openbase 2> go
```

Now you can perform a series of inserts, updates and deletes. Each one should be executed individually.

Committing Changes to the Database

COMMIT

When you are done making changes to the database, you need to commit the transaction to make it final. The commit command can be issued using OpenISQL as follows:

```
openbase 1> COMMIT
```

```
openbase 2> go
```

COMMIT makes all changes final.

Aborting a Transaction

ROLLBACK

ROLLBACK provides a mechanism for aborting database changes since the last BEGIN TRANSACTION. The ROLLBACK command must be issued before committing the transaction. Here is an example using OpenISQL.

```
openbase 1> ROLLBACK

openbase 2> go
```

While this sums up the basic transaction mechanism, there are some other features that you may want to take advantage of. They are described in the following sections.

Locking Options

WRITE TABLE

The WRITE TABLE command obtains wants-to-write locks on tables you wish to change. Other users are blocked from updating locked tables until the locks are released by a COMMIT or ROLLBACK from the lock owner. However, table locks do not block other users from reading table information (unless they first try to obtain a lock).

WRITE TABLE waits for other users to release locks before obtaining them and returning control to the client program.

WRITE TABLE solves a particular problem. You may want to select a value from a row and then update the value within the same transaction. Specifying which table you plan to write to in the beginning can also help prevent deadlocks. It will also ensure that the value you select will not change before you have a chance to update the table.

Here is an example of how it is used.

```
openbase 1> WRITE TABLE EMPLOYEE, DEPARTMENT

openbase 2> go
```

FOR UPDATE

FOR UPDATE can be used instead of WRITE TABLE if you want to obtain locks on specific records instead of locking the whole table.

FOR UPDATE is used in conjunction with a select statement to mark all rows in the result. The select statement returns no result if any of the result rows have been marked by another user. Whether a record is locked or not will have no effect on the user's ability to read the record, unless of course, they try to obtain their own FOR UPDATE.

FOR UPDATE may appear after each table specification in the FROM portion of select statements. Following is an example.

```
openbase 1> SELECT * FROM EMPLOYEE FOR UPDATE

openbase 2> go
```

This example will mark all the records in the EMPLOYEE table. By constraining the search you can mark just the records you are interested in.

```
openbase 1> select * from EMPLOYEE t0 FOR UPDATE

openbase 2> where t0.FIRST_NAME ='Joe'

openbase 3> go
```

Locks are automatically released when COMMIT, ROLLBACK or ROLLBACK LOCKS (which only releases the locks, not the transaction) are sent to the server.

LOCK RECORD / UNLOCK RECORD

LOCK RECORD is a simple way of marking a single row using its _rowid value. The same thing can be accomplished using FOR UPDATE(described above), but LOCK RECORD is faster and more efficient. LOCK RECORD returns an error if the record is already locked.

The OpenBase API includes the markRow:ofTable: and markRow:ofTable:alreadyMarkedByUser: for performing this task from Objective-C. However, you can issue the SQL commands directly and obtain the same effect. Here is an example.

The following example marks record 50500 in the EMPLOYEE table.

```
openbase 1> LOCK RECORD 50500 EMPLOYEE

openbase 2> go
```

The next example releases the mark on record 50500.

```
openbase 1> UNLOCK RECORD 50500 EMPLOYEE

openbase 2> go
```

Any marks you obtain during a transaction, whether by LOCK RECORD or FOR UPDATE, will be removed when you execute the COMMIT, ROLLBACK or ROLLBACK LOCKS commands.

6

Programming Interface

This chapter will present instructions on how to communicate with OpenBase from your programs.

To use the Objective-C API to OpenBase, you need to include the OpenBaseAPI.framework in the frameworks section of the project builder. This framework contains all the supporting methods discussed in this chapter.

Connecting to a Database Server

The first thing that your application should do is initialize its connection with the database server. We recommend that you do this after your application has initialized. You may create as many OpenBase objects as you want and connect to several different databases at the same time.

The following code shows one way to connect to the OpenBase server. For this example, we have stripped out some of the error handling to make the necessities clear.

```
 #import <OpenBaseAPI/OpenBase.h>
 ...
- (OpenBase *)connectDatabase
    {
          int returnCode;
          OpenBase *database;
          database = [[OpenBase alloc] init];
          if (![database connectToDatabase:"Movies"
              onHost:"*"
              login:"admin"
              password:"" return:&returnCode) {
               printf("failed with error code %d\n",
              returnCode);
              exit(0);
          }
          return database;
    }
```

In the above example, the call to: connectToDatabase: onHost: login: password: return: shows how to connect to the OpenBase server. Since the above code sample specifies a "*" instead of a target host, the interface will search all hosts on your network for the Movies database. Specifying a host will minimize the time it takes to connect with the database, but will not affect overall performance.

Constructing SQL

Once you have initialized a connection with OpenBase, you are ready to interact with it using SQL. Constructing an SQL statement can be done all at once or in little pieces using the makeCommand: method. Here is an example where the entire SQL statement is defined with a single makeCommand: call.

```
[database makeCommand: "select TITLE from MOVIE where
REVENUE > 300000"];
```

While this is correct, you may find it more convenient to specify the statement in smaller pieces.

```
[database makeCommand: “select TITLE “];
[database makeCommand: “from MOVIE “];
[database makeCommand: “where REVENUE > 300000"];
```

When constructing queries in pieces, one common mistake is forgetting the spaces at the beginning or end of each segment. For instance, if you send the strings “select TITLE” and “from MOVIE”, the result is “select TITLEfrom MOVIE”. As you can see, there is a space missing that will cause an error. Therefore, be very careful to include the appropriate spaces when constructing SQL statements.

SQL Statement Execution

Once you are done constructing an SQL statement, you can send it to the database by calling the executeCommand method. Here is an example of how the executeCommand method is invoked:

```
[database executeCommand];
```

When you execute a statement, the content of the command buffer is sent to the OpenBase server. The executeCommand method returns TRUE if the query is executed, otherwise FALSE is returned.

Using Row IDs

While it is not necessary, we recommend that application designers use the _rowid column to identify records. _rowid holds automatically generated unique values for identifying records in each table.

If you don't specify _rowid’s in your insert statements, they will be automatically generated for you by the server. However, if you want to insert a record that you want an immediate handle on, you need to ask the database for the next available _rowid for use with your insert statement. In this section, we show what must be done to generate and use unique _rowid’s.

Before you insert a record into the database, you should get a new unique identification number from the OpenBase object. This new unique number (returned as a string) should be used when inserting

the record into the database. The following code segment gets a new record identifier for the contact table.

```
char uniqueKey[20];
strcpy(uniqueKey, [database uniqueRowIdForTable: “MOVIE”]);
```

Once you have this unique identification value, you can use it to construct an SQL statement for inserting a record into the database. Here is an example of how this is done:

```
[database makeCommand: “insert into MOVIE (_rowid, TITLE,
REVENUE) values “];
sprintf(buff, “('%s','%s',%f’) “, uniquekey, movieTitle,
revenue);
[database makeCommand:buff];
[database executeCommand];
```

The above code segment demonstrates how to insert a set of values into a database table.

Linking BLOBs To Your Records

Linking BLOBs (Binary Large OBjects) to database records is easily accomplished through the OpenBase API and SQL language.

All you need to do is specify a column of type object in your database schema to hold a BLOB key. A BLOB key is simply an 8-character string that uniquely identifies the BLOB in the database. By saving the key along with other information, you are able to associate the information with BLOBs of text or image data. All you need to retrieve a BLOB is the key that is stored in the associated record.

Retrieving Records

You can use a select statement to retrieve previously saved information. Here is a sample select statement that retrieves all PG rated Movies and sorts them by title:

```
SELECT _rowid, TITLE FROM MOVIE WHERE RATING ='PG'
ORDER BY TITLE
```

This query can be sent to the database by the following Objective-C commands:

```
[database makeCommand:"SELECT _rowid, TITLE FROM MOVIE
WHERE RATING ='PG' ORDER BY TITLE"];
[database executeCommand];
```

When we get each row of the result, the column values are dumped into variables specified by the programmer. The following code segments demonstrate how to retrieve the results from the SQL request made above. After executing a query, you should call rowsAffected to see if any results have been returned.

```
if ([database rowsAffected]== 0) return NO_RESULT;
```

If there is a result, you should bind your program variables to the columns of the result. Since there are two columns, we will call the bindString: method twice. This tells the OpenBase interface where to put the information as it retrieves each row of the result.

```
[database bindString: uniqueKey];
[database bindString: movieTitle];
```

In this case, the target variables are both character strings. If one of the columns in your result is of another type, bindString: will make sure that the values are converted to a string. You may also want to use the bindInt:, bindDouble:, bindLong: and bindLongLong: methods for other target values. The interface will always try to convert column data to the target variable's type. Variables must be bound in order, starting with the first and ending with the last column.

Now that the columns have been bound to the program variables, you will need to retrieve the rows. This is done using the nextRow method, which should be called once for each row in the result. Each time it is called, the

program variables bound to the columns will be filled with the column data for the next available row. When nextRow returns FALSE, it means therearenomorerowstoberetrieved.Hereisanexample:

```
while ([database nextRow]) {
        / / do something with uniqueKey and movieTitle
        printf("%s,%s", uniqueKey, movieTitle);
}
```

SimpleTool Example

Previously in this chapter we showed some examples using Objective-C. In this section we will discuss a program written using the C API. All Objective-C calls have equivalent C calls which perform the same functions. The Objective-C version of this program is available in the examples directory.

SimpleTool demonstrates interaction with a relational database, without using the tedious programming overhead, common with databases.

Using C or Objective-C is the simplest way to access OpenBase. SimpleTool will retrieve from the database the movies and the revenue from the producing studios. The listing below illustrates the OpenBase API framework. A discussion follows.

```
#import <Foundation/Foundation.h>
#import <OpenBaseAPI/OpenBase.h>


int main (int argc, const char *argv");
{
        NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
        int returnCode;
        OpenBase *connection = ob_newConnection"];

        / /variables to hold values
        char movieTitle[256];
        char studioName[256];
        long revenue;
```

```
        if (!ob_connectToDatabase(connection,"Movie",
                "*","","",&returnCode)) {
            printf("%s\n", ob_connectErrorMessage(connection));
            return -1;
        }

        ob_makeCommand(connection,
"select t0.TITLE, t0.REVENUE, t1.NAME from MOVIE t0, STUDIO t1
where t0.STUDIO_ID = t1.STUDIO_ID order by t0.REVENUE DESC");

        if (!ob_executeCommand(connection)) {
            printf("ERROR%s\n",ob_serverMessage(connection));
            ob_invalidate(connection);
            return -1;
        }

        ob_bindString(connection, movieTitle);
        ob_bindLong(connection, &revenue);
        ob_bindString(connection, studioName);

        while (ob_nextRow(connection)) {
            printf("%s made $%ld for %s.\n",movieTitle,
                revenue, studioName);
        }

        ob_invalidate(connection);
        [pool release];
        return 0;
}
```

Discussion: SimpleTool_main.m

Main begins by establishing a connection to the database, if a connection was not made, print the offending message returned from the connection object and exit. Using the ob_connectToDatabase() function, establish a connection to the database with the database name, hostname, logon id, and password.

```
int returnCode;
OpenBase *connection = ob_newConnection();
if (!ob_connectToDatabase(connection, “Movie”, “*”, ““,””,
&returnCode)) {
        printf(“%s\n”, ob_connectErrorMessage(connection));
        return -1;
}
```

After a successful connection has been established, the ob_makeCommand() function is used to send SQL statements. The TITLE and REVENUE data columns from the MOVIE table as well as the associated studio NAME from the STUDIO table are retrieved. The SQL statements are now buffered for later execution by the database server.

```
ob_makeCommand(connection,”select t0.TITLE, t0.REVENUE,
t1.NAME from MOVIE t0, STUDIO t1 where t0.STUDIO_ID =
t1.STUDIO_ID order by t0.REVENUE DESC”);
```

The ob_executeCommand() passes the buffered SQL statements to the database server and returns TRUE for successful and FALSE for failed execution.

```
if (!ob_executeCommand(connection)) {
        printf(“ERROR - %s\n”,ob_serverMessage(connection));
        ob_invalidate(connection);
        return -1;
}
```

The ob_bindString() and ob_bindLong() functions, bind the resulting data columns from the database, to the receiving program variables. SimpleTool binds the variables movieTitle, revenue and studioName respective to the order of the initial SELECT statement.

```
ob_bindString(connection, movieTitle);
ob_bindLong(connection, &revenue);
ob_bindString(connection, studioName);
```

ob_nextRow() increments through the result rows and retrieves the data. FALSE is returned when all data is processed.

```
while (ob_nextRow(connection)) {
   printf("%s made $%ld for %s.\n",movieTitle, revenue,studioName);
}
```

Main ends with a call to terminate the connection to the database server.

```
ob_invalidate(connection);
```

7

Application Notification

This chapter will help you understand how to take advantage of notification in your applications, including the following:

Overview

Application Notification means that your applications can be notified when the database changes. This allows you to build data views or other information displays that will always stay up-to-date with the information in the database.

Registering for Notification

Before you can receive notifications, you will need to prepare an object to respond to the appropriate delegate method.

To start notification for object self, do the following:

```
[OpenBaseObj startNotificationFor:self];
```

To remove self from the notification list, do the following:

```
[OpenBaseObj removeNotificationFor:self];
```

Executing these commands will register your object to receive notification of database changes. Finally, you will need to implement the notifyChange:database:intable:vid:field:value: delegate method. This method is defined as follows:

notifyChange:database:intable:vid:field:value:

```
- notifyChange:(const char *)action
      database:(const char *)databaseName
      intable:(const char *)tableName
      vid:(const char *)rowid
      field:(const char *)fieldName
      value:(const char *)aValue
```

This delegate method is called to notify your application when a change is made to the database. The field and value parameters will always be an empty string, but they will be used in a future version of OpenBase. All objects that set themselves up as the delegate of the notifier object will be notified of database changes.

The action parameter is one of the following values: (lock), (unlock), (update), (insert) or (delete). (lock) and (unlock) indicate when a record has been marked or unmarked by another user.

When your object is notified of a change, you can select the new values from the database. Since others will be asking for the same piece of information, OpenBase buffers the information for faster access.

8

OpenBase API

This chapter will define the methods in the OpenBase-SQL Objective-C library.

Overview

To use the OpenBase-SQL Objective-C library, you will need to include the OpenBaseAPI.framework in the frameworks section of your project. The OpenBase header file may be included in your program source by importing <OpenBaseAPI/OpenBase.h>.

OpenBase-SQL Objective-C methods

The following methods will be illustrated:

beginTransaction:

Defined in <OpenBaseAPI/OpenBase.h>

- (BOOL)beginTransaction

Returns TRUE if the transaction started correctly. If an error is detected, FALSE is returned.

Objective-C Example:

```
OpenBase *connection;
...
if ([connection beginTransaction])
   return SUCCEED;
```

ANSI C Example:

```
OpenBase *connection;
...
if (ob_beginTransaction(connection))
   return SUCCEED;
```

See Also: "commitTransaction" on page 103., "rollbackTransaction" on page 124.

bindDouble:

Defined in <OpenBaseAPI/OpenBase.h>

- bindDouble:(double *)var

- bindDouble:(double *)var column:(int)col;

bindDouble: binds the double variable var to the next result column. Each time a call to bindString:, bindInt:, bindDouble:, bindLong:, or bindLongLong: is made, the column pointer is incremented to the next column. You should make as many calls as needed to bind your program variables to all the columns in your query result.

bindDouble:column: binds the double variable var to the specific column col. col is an integer which counts from column 0.

These methods should be used after a query has been executed and results have been returned. Column values will automatically be converted to the target variable's type.

Objective-C Example:

```
OpenBase *connection;
double doubleValue;
...
[connection bindDouble:&doubleValue];
```

ANSI C Example:

```
OpenBase *connection;
double doubleValue;
...
ob_bindDouble(connection, &doubleValue);
```

bindInt:

Defined in <OpenBaseAPI/OpenBase.h>

- bindInt:(int *)var

- bindInt:(int *)var column:(int)col;

bindInt: binds the integer variable var to the next result column. Each time a call to bindString:, bindInt:, bindDouble:, bindLong:, or bindLongLong: is made, the column pointer is incremented to the next column. You should make as many calls as needed to bind your program variables to all the columns in your query result.

bindInt:column: binds the character string variable var to the specific column col. col is an integer which counts from column 0.

These methods should be used after a query has been executed and results have been returned. Column values will automatically be converted to the target variable's type.

Objective-C Example:

```
OpenBase *connection;
int intValue;
...
[openbase bindInt:&intValue];
```

ANSI C Example:

```
OpenBase *connection;
int intValue;
...
ob_bindInt(connection, &intValue);
```

bindLong:

Defined in <OpenBaseAPI/OpenBase.h>

- bindLong:(long *)var;

- bindLong:(long *)var column:(int)col;

bindLong: binds the long integer variable var to the next result column. Each time a call to bindString:, bindInt:, bindDouble:, bindLong:, or bindLongLong: is made, the column pointer is incremented to the next column. You should make as many calls as needed to bind your program variables to all the columns in your query result.

bindLong:column: binds the long integer variable var to the specific column col. col is an integer which counts from column 0.

These methods should be used after a query has been executed and results have been returned. Column values will automatically be converted to the target variable's type.

Objective-C Example:

```
OpenBase *connection;
long longValue;
...
[connection bindLong:&longValue];
```

ANSI C Example:

```
OpenBase *connection;
long longValue;
...
ob_bindLong(connection, &longValue);
```

bindLongLong:

Defined in <OpenBaseAPI/OpenBase.h>

- bindLongLong:(long long *)var;

- bindLongLong:(long long *)var column:(int)col;

bindLongLong: binds the long long integer (64 bit) variable var to the next result column. Each time a call to bindString:, bindInt:, bindDouble:, bindLong:, or bindLongLong: is made, the column pointer is incremented to the next column. You should make as many calls as needed to bind your program variables to all the columns in your query result.

bindLongLong:column: binds the long long integer variable var to the specific column col. col is an integer which counts from column 0.

These methods should be used after a query has been executed and results have been returned. Column values will automatically be converted to the target variable's type.

Objective-C Example:

```
OpenBase *connection;
long long longlongValue;
...
[connection bindLongLong:&longlongValue];
```

ANSI C Example:

```
OpenBase *connection;
long long longlongValue;
...
ob_bindLongLong(connection, &longlongValue);
```

bindString:

Defined in <OpenBaseAPI/OpenBase.h>

- bindString:(const char *)var

- bindString:(const char *)var column:(int)col;

bindString: binds the character string variable var to the next result column. Each time a call to bindString:, bindInt:, bindDouble:, bindLong:, or bindLongLong: is made, the column pointer is incremented to the next column. You should make as many calls as needed to bind your program variables to all the columns in your query result.

bindString:column: binds the character string variable var to the specific column col. col is an integer which counts from column 0.

These methods should be used after a query has been executed and results have been returned. Column values will automatically be converted to the target variable's type.

Objective-C Example:

```
OpenBase *connection;
char firstname[256], lastname[256];

[connection makeCommand: "select FIRST_NAME,
LAST_NAME
from EMPLOYEE order by LAST_NAME"];
if (![connection executeCommand]) return;
if (![connection resultReturned]) return;

[connection bindString:firstname];
[connection bindString:lastname];

while ([connection nextRow]) {
   printf("%s, %s\n",lastname,firstname);
}
```

ANSI C Example:

```
OpenBase *connection;
char firstname[256], lastname[256];
```

```
ob_makeCommand(connection, "select FIRST_NAME,
LAST_NAME
from EMPLOYEE order by LAST_NAME");
if (!ob_executeCommand(connection)) return;
if (!ob_resultReturned(connection)) return;

ob_bindString(connection, firstname);
ob_bindString(connection, lastname);

while (ob_nextRow(connection)) {
   printf("%s, %s\n",lastname,firstname);
}
```

bufferHasCommands

Defined in <OpenBaseAPI/OpenBase.h>

- (BOOL)bufferHasCommands

Returns TRUE if unexecuted commands are in the command buffer, FALSE otherwise.

Objective-C Example:

```
OpenBase *connection;
...
if ([connection bufferHasCommands])
   return SUCCEED;
```

ANSI C Example:

```
OpenBase *connection;
...
if (ob_bufferHasCommands(connection))
   return SUCCEED;
```

clearCommands

Defined in <OpenBaseAPI/OpenBase.h>

- clearCommands;

Clears the command buffer. This method removes any unexecuted commands from the command buffer.

Objective-C Example:

```
OpenBase *connection;
...
[connection clearCommands];
```

ANSI C Example:

```
OpenBase *connection;
...
ob_clearCommands(connection);
```

commandBuffer

Defined in <OpenBaseAPI/OpenBase.h>

- (const char *)commandBuffer;

This method is used to look at SQL commands in the command buffer. The buffer will return the most current SQL command.

Returns the contents of the command buffer.

Objective-C Example:

```
OpenBase *connection;
...
prinf("%s\n", [connection commandBuffer]);
```

ANSI C Example:

```
OpenBase *connection;
...
printf("%s\n",ob_commandBuffer(connection));
```

commitTransaction

Defined in <OpenBaseAPI/OpenBase.h>

- (BOOL)commitTransaction;

Returns TRUE if the transaction committed to the database. If an error is detected, FALSE is returned.

Objective-C Example:

```
      OpenBase *connection;
      ...
      if ([connection commitTransaction]) return
SUCCEED;
```

ANSI C Example:

```
      OpenBase *connection;
      ...
      if (ob_commitTransaction(connection))
         return SUCCEED;
```

connectErrorMessage:

Defined in <OpenBaseAPI/OpenBase.h>

- (const char *)connectErrorMessage:(int)errorCode

Returns an error message corresponding to the error code errorCode set by connectToDatabase:onHost:login:password:return:.

Objective-C Example:

```
OpenBase *connection;
int errorNum;
...
prinf("%s\n", [connection
connectErrorMessage:errorNum]);
```

ANSI C Example:

```
OpenBase *connection;
...
printf("%s\n",ob_connectErrorMessage(connection,
errorNum));
```

connectToDatabase:onHost:login:password:return:

Defined in <OpenBaseAPI/OpenBase.h>

- (BOOL)connectToDatabase:(const char *)dbName
onHost:(const char *)hostName
login:(const char *)loginName
password:(const char*)password return:(int*)returnCode;

This method initializes a program's connection to a OpenBase database. Upon success, this method returns TRUE. Otherwise it returns FALSE.

The parameters are described as follows:

- dbName is the name of the database that you would like to connect to.
- hostName is the host name of the computer where the database resides. A "]; may be substituted to search the network for the designed database. An empty string specifies that the database is on the local computer.
- loginName is the name that the user uses to log into the database. In order for the initialization to be successful, this login name must correspond to an entry in the user table.
- password is the password corresponding to the login name.
- returnCode is one of the following values:

ERR_SFTUSRLIM means that the software license limit has been reached.

ERR_DBSUSRLIM means that the OpenBase licensed user limit has been reached.

ERR_NOSERVER means that the server is not running.

ERR_INCORRECT_LOGIN means that the user login and password were not correct.

This method performs the initialization necessary to work with the database. You can not use the OpenBase object before it has been initialized using this method.

```
- (BOOL)connectToDatabase:(const char *)dbName
      onHost:(const char *)hostName
      login:(const char *)loginName
```

```
password:(const char *)password
return:(int *)returnCode;
```

Objective-C Example:

```
OpenBase *connection;
char dbase[50];
char host[50];
char login[50];
char passwd[50];
int errorNum;
...
if ([connection connectToDatabase:dbase
     onHost:host
     login:login
     password:passwd
     return:&errorNum]) {
   return SUCCEED;
} else {
   printf("%s\n", [connection
   connectErrorMessage:errorNum]);
}
```

ANSI C Example:

```
OpenBase *connection;
char dbase[50];
char host[50];
char login[50];
char passwd[50];
int errorNum;
...
if (ob_connectToDatabase(connection, dbase,
host,login,passwd, &errorNum)) {
   return SUCCEED;
} else {
   printf("%s\n",
   ob_connectErrorMessage(connection,errorNum));
}
```

databaseName

Defined in <OpenBaseAPI/OpenBase.h>

- (const char *)databaseName;

Returns the name of the database.

Objective-C Example:

```
OpenBase *connection;
...
prinf("%s\n", [connection databaseName]);
```

ANSI C Example:

```
OpenBase *connection;
...
printf("%s\n",ob_databaseName(connection));
```

executeCommand

Defined in <OpenBaseAPI/OpenBase.h>

- (BOOL)executeCommand;

Executes the SQL commands in the command buffer.

Returns TRUE if the SQL was executed. If an error is detected FALSE is returned and the server message is set with the error message (see serverMessage).

Objective-C Example:

```
OpenBase *connection;
...
if ([connection executeCommand]) return SUCCEED;
```

ANSI C Example:

```
OpenBase *connection;
...
if (ob_executeCommand(connection) return SUCCEED;
```

hostName

Defined in <OpenBaseAPI/OpenBase.h>

- (const char *)hostName

Returns the name of the host where the OpenBase server is running.

Objective-C Example:

```
OpenBase *connection;
...
prinf("%s\n", [connection hostName]);
```

ANSI C Example:

```
OpenBase *connection;
...
printf("%s\n", ob_hostName(connection));
```

isColumnNULL:

Defined in <OpenBaseAPI/OpenBase.h>

- (BOOL)isColumnNULL:(int)col;

Checks to see if a returned column has a NULL value. col specifies the column position where position 0 is the first column. This method must be used in conjunction with nextRow.

Returns TRUE if the column specified by col is NULL. Otherwise FALSE is returned.

Objective-C Example:

```
OpenBase *connection;
int colNum;
...
if ([connection isColumnNULL:colNum])
   return SUCCEED;
```

ANSI C Example:

```
OpenBase *connection;
int colNum;
...
if (ob_isColumnNULL(connection, colNum))
   return SUCCEED;
```

loginName

Defined in <OpenBaseAPI/OpenBase.h>

- (const char *)loginName

Returns the current database login.

Objective-C Example:

```
OpenBase *connection;
...
prinf("%s\n", [connection loginName]);
```

ANSI C Example:

```
OpenBase *connection;
...
printf("%s\n",ob_loginName(connection));
```

makeCommand:

Defined in <OpenBaseAPI/OpenBase.h>

- makeCommand:(char *)cmd;

This method allows you to build an SQL statement. Each time you call this method, the content of cmd is appended to the end of an SQL buffer. When the SQL query is completely constructed, the execute-Command method will send the query to the database.

Objective-C Example:

```
OpenBase *connection;
...
[connection makeCommand:"insert into EMPLOYEE "];
[connection makeCommand:"(FIRST_NAME) values "];
[connection makeCommand:"(`John Smith')"];

if ([connection executeCommand]) return SUCCEED;
```

ANSI C Example:

```
OpenBase *connection;
...
ob_makeCommand(connection,"insert into EMPLOYEE ");
ob_makeCommand(connection, "(FIRST_NAME) values ");
ob_makeCommand(connection, "(`John Smith')");

if (ob_executeCommand(connection))
   return SUCCEED;
```

markRow:ofTable:alreadyMarkedByUser:

Defined in <OpenBaseAPI/OpenBase.h>

- (int)markRow:(const char *)anId ofTable:(const char *)tableName
  alreadyMarkedByUser:(char *)userName;

These methods attempt to mark a row in table tableName where _rowid equals anId. These methods are useful for managing a pessimistic locking strategy.

Marking records before being edited will enable your programs to coordinate database changes between multiple users. If a record has been marked by another user, we recommend that the record be displayed in read-only mode. Marks are only flags to other programs. They do not block SQL from updating the records in question.

Marks are automatically released if the database is restarted or if the communication link goes down between the server and client.

Returns TRUE if the database row is marked successfully. These methods return FALSE if the record was already marked by another user. userName is set to the user who owns the mark if the attempt fails.

Objective-C Example:

```
OpenBase *connection;
char rowId[30];
char table[50];
...
if ([connection markRow:rowId ofTable:table])
   return SUCCEED;
```

ANSI C Example:

```
OpenBase *connection;
...
if (ob_markRow(connection, rowId, table))
   return SUCCEED;
```

nextRow

Defined in <OpenBaseAPI/OpenBase.h>

- (BOOL)nextRow;

This method retrieves the next row in a search result. Each time a database row (or record) is processed using nextRow, column values are placed in Objective-C variables previously bound to each column.

If nextRow successfully processes a result row it returns TRUE. If there are no more result rows to process it returns FALSE.

Objective-C Example:

```
OpenBase *connection;
...
if ([connection nextRow]) return SUCCEED;
```

ANSI C Example:

```
OpenBase *connection;
...
if (ob_nextRow(connection)) return SUCCEED;
```

password

Defined in <OpenBaseAPI/OpenBase.h>

- (const char *)password

Returns the password used to login to the database.

Objective-C Example:

```
OpenBase *connection;
...
prinf("%s\n", [connection password]);
```

ANSI C Example:

```
OpenBase *connection;
...
printf("%s\n", ob_password(connection));
```

removeMarkOnRow:ofTable:

Defined in <OpenBaseAPI/OpenBase.h>

- (int)removeMarkOnRow:(const char *)anId ofTable:(const char *)tableName;

removeMarkOnRow:ofTable: releases a previously marked record, where anId maps to the record's _rowid column in the table tableName.

Returns TRUE if it succeeds. Otherwise FALSE is returned.

Objective-C Example:

```
OpenBase *connection;
char rowId[30];
char table[50];
...
if ([connection removeMarkOnRow:rowId
ofTable:table]) return SUCCEED;
```

ANSI C Example:

```
OpenBase *connection;
...
if (ob_removeMarkOnRow(connection, rowId, table))
   return SUCCEED;
```

removeNotificationFor:

Defined in <OpenBaseAPI/OpenBase.h>

- removeNotificationFor:notificationDelegate;

Removes the notificationDelegate from the notification list.

Objective-C Example:

```
OpenBase *connection;
id notify;
...
[connection removeNotificationFor:notify];
```

resultColumnCount

Defined in <OpenBaseAPI/OpenBase.h>

- (int)resultColumnCount;

This method may be used to get the number of columns in a result.

Returns the number of columns returned by a query result.

Objective-C Example:

```
OpenBase *connection;
...
prinf("%d\n", [connection resultColumnCount]);
```

ANSI C Example:

```
OpenBase *connection;
...
printf("%d\n",ob_resultColumnCount(connection));
```

resultColumnName:

Defined in <OpenBaseAPI/OpenBase.h>

-(const char *)resultColumnName:(int)col;

This method is used to get the column name of the result column specified by col. col specifies the column position where position 0 is the first column. This method should only be called after executing a query.

Returns the column name of the column specified by col.

Objective-C Example:

```
OpenBase *connection;
int colNum;
...
prinf("%s\n", [connection
resultColumnName;colNum]);
```

ANSI C Example:

```
OpenBase *connection;
int colNum;
...
printf("%s\n", ob_resultColumnName(connection,
colNum));
```

resultColumnType

Defined in <OpenBaseAPI/OpenBase.h>

- (int)resultColumnType:(int)col;

This method may be used after results have been detected to get the natural type of the result columns. col specifies the column position where position 0 is the first column.

The types are defined as follows:

OBTYPE_CHAR	Character String
OBTYPE_INT	Integer
OBTYPE_DOUBLE	Double
OBTYPE_LONG	Long
OBTYPE_LONGLONG	Long Long
OBTYPE_MONEY	Money
OBTYPE_DATE	Date
OBTYPE_TIME	Time
OBTYPE_DATETIME	Date-time
OBTYPE_OBJECT	Object / BLOB

Returns the natural type of the column col.

Objective-C Example:

```
OpenBase *connection;
int colNum;
...
if([connection resultColumnType:colNum] ==
        OBTYPE_MONEY)
printf("MONEY!!!!!!!");
```

ANSI C Example:

```
OpenBase *connection;
int colNum;
...
if(ob_resultColumnType(connection, colNum) ==
     OBTYPE_MONEY) printf("MONEY!!!!!!!");
```

resultReturned

Defined in <OpenBaseAPI/OpenBase.h>

- (BOOL)resultReturned;

Checks to see if results need to be processed by nextRow.

Returns TRUE if there are results that need processed. Otherwise FALSE is returned.

Objective-C Example:

```
OpenBase *connection;
...
if ([connection resultReturned]) return SUCCEED;
```

ANSI C Example:

```
OpenBase *connection;
...
if (ob_resultReturned(connection))
   return SUCCEED;
```

resultTableName:

Defined in <OpenBaseAPI/OpenBase.h>

- (const char *)resultTableName:(int)col;

This method is used to get the table name of the result column specified by col. col specifies the column position where position 0 is the first column. This method should only be called after executing a query.

Returns the table name of the column specified by col.

Objective-C Example:

```
OpenBase *connection;
int colNum;
...
printf("%s\n",[connection
resultTableName:colNum]);
```

ANSI C Example:

```
OpenBase *connection;
int colNum;
...
printf("%s\n",ob_resultTableName(connection,
   colNum));
```

rollbackTransaction

Defined in <OpenBaseAPI/OpenBase.h>

- (BOOL)rollbackTransaction;

Returns TRUE if the transaction is rolled back. If an error is detected FALSE is returned.

Objective-C Example:

```
OpenBase *connection;
...
if ([connection rollbackTransaction])
   return SUCCEED;
```

ANSI C Example:

```
OpenBase *connection;
...
if (ob_rollbackTransaction(connection))
   return SUCCEED;
```

rowsAffected

Defined in <OpenBaseAPI/OpenBase.h>

- (int)rowsAffected;

This method is called after executing an SQL command using execute-Command.

Returns the number of database rows affected by the most recent SQL command.

Objective-C Example:

```
OpenBase *connection;
...
prinf("Found %d items\n", [connection
rowsAffected]);
```

ANSI C Example:

```
OpenBase *connection;
...
printf("Found %d items\n",
ob_rowsAffected(connection));
```

serverMessage

Defined in <OpenBaseAPI/OpenBase.h>

- (const char *)serverMessage;

Returns a message from the server pertaining to the last query executed.

Objective-C Example:

```
OpenBase *connection;
...
prinf("%s\n", [connection serverMessage]);
```

ANSI C Example:

```
OpenBase *connection;
...
printf("%s\n",ob_serverMessage(connection));
```

startNotificationFor:

Defined in <OpenBaseAPI/OpenBase.h>

- startNotificationFor:notificationDelegate;

startNotificationFor: starts a notification session for notificationDelegate. notificationDelegate must implement notifyChange:database:intable:vid:field:value:.

```
- notifyChange:(const char *)action
  database:(const char *)databaseName
  intable:(const char *)tableName
  vid:(const char *)rowid
 field:(const char *)fieldName
  value:(const char *)aValue
```

This delegate method is called to notify your application when a change is made to the database. The field and value parameters will always be an empty string, but they will be used in a future version of OpenBase. All objects that set themselves up as the delegate of the notifier object will be notified of database changes.

Please see the chapter on notification for more information about this method.

Objective-C Example:

```
OpenBase *connection;
id notificationDelegate;
...
[connection
startNotificationFor:notificationDelegate];
```

uniqueRowIdForTable:

Defined in <OpenBaseAPI/OpenBase.h>

- (const char *)uniqueRowIdForTable:(const char *)tablename

Returns a unique _rowid value for the specified table tablename. This value must be inserted with a new record.

Objective-C Example:

```
OpenBase *connection;
char table[50];
char newRowId[30];
...
strcpy(newRowId,[connection
uniqueRowIdForTable:table]);
```

ANSI C Example:

```
OpenBase *connection;
char table[50];
char newRowId[30];
...
strcpy(newRowId,
ob_uniqueRowIdForTable(connection,
table));
```

BLOB/Object Handling Methods:

This section will discuss the following methods:

"retrieveBinary:" on page 130.

"insertBinary:size:" on page 131.

retrieveBinary:

Defined in <OpenBaseAPI/OpenBase.h>

- (NSData *)retrieveBinary:(const char *)anId;

This method retrieves a BLOB by its identification key and returns an NSData object. The NSData object will release itself automatically.

The ANSI-C API returns newly allocated memory containing the BLOB data. The BLOB size is returned through the size parameter.

Objective-C Example:

```
OpenBase *connection;
char blobID[10];
NSData *blobData;
...
blobData = [connection retrieveBinary:blobID];
```

ANSI C Example:

```
OpenBase *connection;
char blobID[10];
int returnSize;
char *bytes;
...
bytes = ob_retrieveBinary(connection, blobID,
&returnSize);
```

insertBinary:size:

- (const char *)insertBinary:(char *)data size:(int)size;

This method loads the BLOB of length size and location data into the database. This method returns a key string (10 characters maximum) that can be used to retrieve the data later. Information that is saved using this method may be retrieved as a file as long as the filename is specified.

BLOBs that are saved into the database are automatically removed when the record referencing the BLOBs are deleted. You should never reference the same BLOB key string in multiple records. Doing so could cause BLOBs to be removed prematurely.

Objective-C Example:

```
OpenBase *connection;
char data[BIGVALUE];
int size = BIGVALUE;
char key[10];
...
strcpy(key, [connection insertBinary:data
size:size]);
```

ANSI C Example:

```
OpenBase *connection;
char data[BIGVALUE];
int size = BIGVALUE;
char key[10];
...
strcpy(key,ob_insertBinary(connection, data,
size));
```

9

Interactive SQL

This chapter introduces OpenISQL, the interactive command line interface to OpenBase. The OpenBase version of ISQL allows users to execute SQL from the command line and import data files. Due to a lack of shell support on Windows NT, OpenISQL is not yet available on that platform. However, there is now an OpenBase Manager tool that will allow you to do similar things through a GUI interface for Windows users. In this chapter, we present the following topics on how to use OpenISQL:

Getting Started

You can start the OpenISQL program by launching it from the OpenBase Manager (See Interactive SQL Terminal in the Tools menu) or by entering /usr/openbase/bin/openisql in a terminal window. This will bring up the openbase prompt. The number following the word openbase in the prompt indicates the current SQL line.

```
openbase 1>
```

The first thing you want to do is connect to a database. For the sake of discussion, let's use the Movies database. You should make sure the Movies database has been started using the OpenBase manager. If you have not started the database, you may want to review the Getting Started section of this manual for information on how to do this. To connect to the customers database, type the following:

```
openbase 1> use Movies
```

The program will prompt you for a login and password. If you have not altered the user table, type admin when it asks for the login. For the password simply press the return key. Here is an example of what you should see on your screen:

```
openbase 1> use Movies
login: admin
password:
using database `Movies' on host 'localhost'
openbase 1>
```

Specifying just the database name will tell ISQL to perform a network wide search for the database named Movies. If you want to specify the hostname you can follow your database name with an @ and the host name. Here is an example:

```
openbase 1> use Movie@helsinki
login:root
password:
using database `Movie' on host 'helsinki'
openbase 1>
```

Executing SQL Queries

The first step is to type in the SQL query. You may type the entire query on one line, or, if you find it more convenient, you may type it on multiple lines. After entering your query, type go on a new line and press return to tell the server that you would like to execute the query. Here is an example:

```
openbase 1> select TITLE from MOVIE
openbase 2> order by TITLE
openbase 3> go
```

This will produce a list of movie titles in alphabetical order.

Clearing a Mistake

If you make a mistake, use the clear command to reset the command buffer. Here is an example of what you will see on your display.

```
openbase 3> clear
Command cleared.
openbase 1>
```

Comments

OpenISQL filters out comments when executing SQL allowing you to place comments in your script files. Comments start with /* and end with */. Everything between these will be ignored by the OpenISQL.

Importing SQL Data

Sometimes you may want to execute a list of SQL statements to populate your database. The load command will run an OpenISQL script containing SQL commands, each followed by `go'. The following example sends the contents of the file /tmp/mySQL to the database.

```
openbase 1> load /tmp/mySQL.file
```

This method of loading the database can be especially useful when executing upgrade scripts from the command line.

Bulk Loading Data

Some databases provide a bulk copy program to import and export data in a comma delimited format. For this reason we provide a bulk load function for importing comma delimited data into OpenBase.

Before loading a comma delimited file you need to specify the table and fields you wish to load the information into. This is done by adding a header to the file. Here is an example of what the top of your data file should look like.

```
TABLE customers
firstname, lastname, company
```

"Joe","Smith","Smith Inc."

"Fred","Jones","Jones Inc"

The first line should specify the table name using the TABLE key word. The second line specifies the columns that the data is to be loaded into separated by commas and the following lines contain data. Each line of data should represent the data to be inserted into a single record and string values must be enclosed in quotes or the ? symbol.

Once you have added the appropriate header to your data file you can load it by using the bulk load command. Here is an example:

openbase1> bulk load /tmp/mydatafile.dta

ISQL will print any errors that it encounters to the ISQL window.

Bulk Saving Data

OpenISQL's bulk save function exports database information to an ASCII file. Tables that have been bulk saved can be bulk loaded into other databases.

To bulk save data to a file type the following command into the OpenISQL window and press the return key.

openbase 1> bulk save /tmp/mydatafile.dta

OpenISQL will open the file and return the following message and prompt:

Enter select query and type go to bulk save data.

bulk save 1>

At the prompt type in a select statement that returns the data you want to bulk save. Here is an example using the Company database:

Enter select query and type go to bulk save data.

bulk save 1> select * from contacts

bulk save 2> order by firstname

bulk save 3> go

If no errors are printed, the data is successfully saved to an ASCII file.

Backup & Restore

The best way to backup your databases is to make copies of them. We recommend that you make regular backup copies of your databases located in /usr/openbase/Databases (OpenBase/Databases on WindowsNT computers). If you need to restore a database, stop the database, remove the database's work directory in /usr/openbase/work and replace the database files with the backup.

Backup and restore functions are also provided for bulk loading and saving ASCII snapshots of the database. The backup function includes schema information as well as bulk saved data for all tables. Files generated by the backup function and read by the restore function are compatible with the backup and restore functions found in the OpenBase Manager.

To backup the database to an ASCII file, type the following:

```
openbase 1> backup /tmp/myfile
```

To restore the database from an ASCII file, type the following:

```
openbase 1> restore /tmp/myfile
```

The backup and restore functions do not include user or permission information. For a more complete backup we recommend copying the database.db bundle located in the /usr/openbase/Databases directory on your server computer.

Import

The OpenBase Manager Data Viewer has an export function which allows you to export data in a character delimited format. The import command in openisql allows you to read character delimited files. Here is the command syntax:

```
import /tmp/myfile
```

The format of the file looks like this when ~ is the separator character:

```
[table]
column1
column2
column3
.
data~data~data
data~data~data
```

History

The history function lists previously executed commands. Here is an example:

```
openbase 1> history

1: backup /tmp/myfile

2: restore /tmp/myfile
```

You can execute one of these commands by either pressing the up arrow key or by typing ! followed by the number or the first letter of the command. Here is an example of executing command #1

```
openbase 1> !1
```

or

```
openbase 1> !b
```

Exiting OpenISQL

To stop the OpenISQL program, type quit at the prompt and press the return key. This will quit the program.

10

Advanced Administration

This chapter has been included to help system administrators manage OpenBase in a networked environment. Topics including:

Exporting Databases

Exporting the /usr/openbase directory **-- DON'T DO IT!**

For many of our customers it has been a temptation to NFS mount the same /usr/openbase directory on all computers across your network. Besides the performance problems with doing this, it is dangerous. Keeping the database directories local and private to each computer ensures that two databases running on different hosts will not try to access the same data files. **We strongly recommend that you do not export this directory.**

Improving Connect Time Using Ports

When OpenBase clients connect to databases they first connect to a nameserver process to look up the database's TCP/IP ports. If connect time is particularly important, you can skip this process by telling the client exactly where to find the server.

The first step is to set the port number of your database. You can do this from the OpenBase Manager Configuration Window or by creating a port file in your database's .db bundle.

Once you have restarted your database it will use the port that you set. To connect to it you need to use # followed by the port number in place

of the database name. For instance, if your database is using port 35000 you should specify #35000 as the database name.

Setting and using port numbers does not affect the ability for clients to connect by database name.

Fine Tuning Database Memory Usage

The ALTER TABLE command can be used to set buffer length defaults for each table. This can be used to fine tune performance by decreasing buffer sizes for tables that are used less frequently and increasing buffer sizes for tables that are used more frequently. If you don't have a need to change these numbers, we recommend that you leave them alone.

Data buffers are counted on pages which may contain a maximum of 128 records on a single 20K page. But it depends on the size of the records. The number of memory based data pages are 100 by default. By increasing this value you will force the server to buffer more of the data in memory and improve performance. The data page value can be set as follows:

```
ALTER TABLE Company SET data_pages 200
```

Improving Select Performance

Adding the CLUSTERED keyword to your index definition arranges the physical data in the table by the indexed column. This provides significant performance benefits in cases where data is accessed through a foreign key or is sorted by the specific column. By clustering the column the database is able to find records likely to be accessed together on consecutive pages.

Only one column per table can be used with clustering. Setting the clustered column replaces any previous setting.

You can also set the clustered column using the OpenBase Manager Schema Window. The popup entitled Natural Order allows you to set the clustered index.

Configuring the Network

There are two processes that provide network functions: openexec and nameserver. Each computer which has the OpenBase server installed will have an openexec process and usually only one computer on your network will have a nameserver process.

The openexec process is responsible for performing a variety of tasks on the local computer. These tasks include: setting the host password, providing a list of all local databases to OpenBase Managers across the network, copying databases, deleting databases, moving databases, setting preferences, and starting up the nameserver process on the correct host.

The nameserver process keeps track of databases and their ports so clients programs can find their databases on the network. Each database that starts up also connects to the nameserver so knowing where the nameserver is running is important.

OpenBase uses configuration files to figuring out where the database nameserver is running on your network. The NameserverHosts file provides this information. Here is where the NameserverHosts file is located on the different operating systems.

OpenStep 4.2: /LocalLibrary/OpenBase/NameserverHosts and /usr/openbase/NameserverHosts

MacOS X Server: /Network/OpenBase/NameserverHosts and /usr/openbaseNameserverHosts

Windows NT: /Apple/OpenBase/NameserverHosts

Solaris: /usr/openbase/NameserverHosts

Here is an example of what a NameserverHosts file looks like:

208.28.195.5

Each client computer should have access to a NameserverHosts file in order to use the OpenBase Managers. Clients can connect to databases without a NameserverHosts file when the database host is specified in your connection dictionary. However, it is better to have the NameserverHosts file be accessible by clients.

Another file which should be setup is the localhost file. It tells the OpenBase databases what the local hostname and ipaddress are. If this file isn't present it will get this information from the operating system. However, this is a good way to ensure that OpenBase is using the ip address you want it to use.

Here is an example of what the localhost file looks like:

myhostname

208.28.195.5

The localhost file should be placed in your "/usr/openbase" directory.

Loading data into OpenBase

There are many ways to import legacy data from another database into OpenBase. One way that we recommend is to use EOUtil and EOModeler to perform the conversion. Using this method will allow you to import data from any ODBC database or any database with an EOF adaptor.

To demonstrate the process we will export contacts data from the Company demo database into another database. Here are the steps to move your database data between two databases.

1. Create a model of your source database using EOModeler. If you are on Windows you can use the ODBC adaptor to connect to your existing database. After the database is created, save your model. For this example we will save our model in /tmp (on MacOS X Server).

2. The next step is to export the data to a plist file. A plist file is essentially a common representation for your database data which we will later import into the target database. The following example moves the data from the Contacts EOF entity to a plist file. To create the plist file type the following command:

```
#/System/Developer/Examples/EnterpriseObjects/Setup-
Wizard.app/Resources/eoutil dump /tmp/contacts.eomodeld -task -
entities Contacts -source database -dest plist /tmp/contacts.plist -
modelGroup /tmp/contacts.eomodeld
```

Keep in mind that if you are on Windows NT you need to change the path names to something like c:\Temp\contacts.eomodel, etc.

4. The next step is to make your adapter point to a newly created OpenBase database. If you do not know how to create and start a new database, please review the OpenBase Manager chapter for details.

Once a new database is created, go to the Model menu in EOModeler and select Switch Adapter. Choose the OpenBase adapter and login to the new database.

5. Now you need to check all your attributes to make sure they are setup correctly. Since some adapters don't put all the information necessary in the attribute definition you may need to make some changes. Here is a list of things to check:

All columns that have a date or time type should have a length of 30.

All string columns should have an appropriate length less than 1024.

Any column that requires more than a 1024 characters should be changed to an object column. If you do this remove the length completely. There should be no length for object columns.

Make sure to save the EOModeler file when you are done.

6. The next step is to create your database tables in the target database. To do this, select the Generate SQL menu item on the Property menu.

A panel will pop up with SQL statements. Since this is a blank database and you don't need to drop the tables first, uncheck the Drop Tables check box.

To create the target tables press the Execute SQL button. If it gives you an error you may need to go back and check your model again. If you do it a second time you may need to drop any database tables first before performing this task again.

7. Now you are ready to import your database data into the OpenBase database. Here is what we did to import the plist back into our database.

```
#/System/Developer/Examples/EnterpriseObjects/Setup-
Wizard.app/Resources/eoutil dump /tmp/contacts.eomodeld -task -
```

entities Contacts -source plist /tmp/contacts.plist -dest database -modelGroup /tmp/contacts.eomodeld

If you get insert errors, you may need to go back and check to make sure your model attributes are correct.

Index

S

T

U

V

W