

A Java Database Framework for SQL Anywhere

Maintaining Control of the Development Environment

Written by: Todd A. Loomis
CodeWorks Software Systems, LLC
June 13, 2006

iANYWHERE

TABLE OF CONTENTS

1	Introduction
1	Development Frameworks
2	Object Oriented Design and the Database
2	JDBC Problems and Pitfalls
3	An Example Framework
3	The Database Interface Class - CWDatabase and CWParamList
4	The Connection Pool - CWConnectionPool
5	The SQL Repository - CWSQLRepository
5	Wrapping JDBC Classes
6	CWConnection
6	CWStatement, CWPreparedStatement, CWCallableStatement
7	Advanced Framework Ideas
7	CWResultSetSerialized
8	Performance Levels
9	SQL Performance Tracking
9	ConnectionPool Command Console
10	Conclusion
11	About the Author

INTRODUCTION

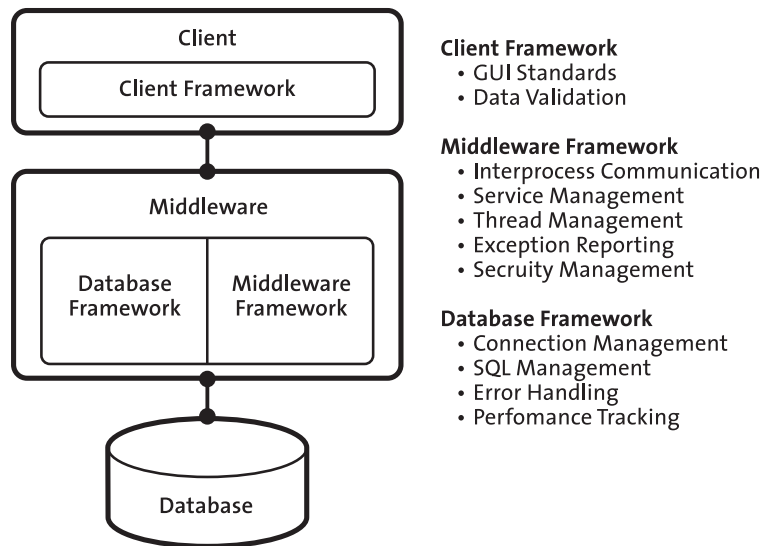
There is nothing more detrimental to database performance than a team of Java programmers armed with a stack of requirements, a database, and a rudimentary knowledge of JDBC. Connections will be kept open and idle for hours, and when they timeout the problem will be assigned to the DBA. Unclosed statements and result sets tying up system resources will be the problem of the DBA. Slow performance due to programmers using obscure JDBC methods and dynamic SQL, is whose fault? You guessed it, the DBA. Add to this the fact that the Development DBA is completely outnumbered by Java programmers, and we quickly end up with a slow performing mess of a database and every problem being assigned to the DBA. As a development DBA myself, if you allow people to do whatever they want to your database, with no control, I believe all these problems are your fault.

This paper discusses the use of a Java Database Framework to help protect the database from marauding Java developers. It provides connection pooling and management; tracking and reporting of JDBC objects; and the ability to easily and selectively remove slow performing structures and methods. The goal is to keep developers from hurting performance and falling into common bad practices and to provide tracking where it can't be prevented. This way you can find the problems and correct them prior to the system going into production. A second goal is to keep things simple so developers can quickly come up to speed. If it is truly simple they will consistently use the framework rather than bypass it.

DEVELOPMENT FRAMEWORKS

With any framework, the goal is to hide complexity and provide a standard operating procedure for complex tasks. An equally important aspect is to provide consistency in how tasks are performed. This leads to better encapsulation and far more maintainable code. Just think if everyone constructed their own class and method for creating a database connection. Chaos and anarchy would ensue. Some common framework areas beyond the database would include inter-process communication, multi thread management, and GUI standards.

Frameworks Provide Standards for Complex Tasks



The framework described herein is meant to be used by all middleware developers, and it provides a layer over the actual JDBC implementation provided by the database vendor. For this paper and the free software I make available, I utilize the JDBC implementation from Sybase iAnywhere's SQL Anywhere, as well as JConnect, also implemented by Sybase. Before we dive into the framework details I think it is important to understand some of the underlying inspiration that led me to concentrate on frameworks.

OBJECT ORIENTED DESIGN AND THE DATABASE

In the early 90's I was contracted to write an object oriented methodology for NASA. When it comes to Software Engineering, I still believe to this day there is no place on Earth that puts as much emphasis on process. In the first draft of the methodology I took the best that I could glean from Coad, Yourdan, Booch, NASA developers, and everything I'd learned from working at NASA for 4 years. I reworked processes into what I felt was a workable flow, and added a streamlined set of documentation and reviews. When I submitted it, I had no idea that the two reviewers were well published in OOD, and were also reviewers of Grady Booch's foremost book on the subject. When I received the two copies back, they were what we called 'bleeding'. A term used for reviewers remarks, which were written in red ink. There were a lot of positive comments, but the key area where I got slaughtered, was in the transition from design to implementation. Their point that there was not enough consideration for architecture, and I believe the word 'protection' of critical components was even used, was driven home. In the second draft, I reworked the Logical OOD Model into a clearer Architectural Model that accounted for the database and all physical components.

Through that experience I now see two basic approaches to designing a database framework for OO projects. One where each class provides meta data definitions and inherits the knowledge of how to save, read, and update itself in the database, making the database framework an integral part of every class that requires persistence. The second is to consider the database as a separate entity with a tight interface for submitting queries. If you want data, you create a Database instance, issue a query, process the result set and close out the transaction.

I lean towards the latter because the DBA is responsible for managing the database only, and doesn't need to be intricately involved in the middleware framework. My decision to lean this way was confirmed during a project with 15 Java middleware programmers all with differing and very strong opinions about the middleware architecture. The debate ended with the senior middleware architect telling me to give them a simple class for accessing data which I owned 100%. If there was a problem getting data, it was my fault. He then volunteered to drive for consensus on a middleware framework, and I gladly accepted. I later learned that multi vendor support is also easier to manage in the second case as well; but that's a separate topic.

JDBC PROBLEMS AND PITFALLS

It is important to remember that JDBC is an implementation provided by a database vendor, and not all implementations are equal. A database framework allows you to equalize them to some extent. In rare instances there will be differences between versions of the same vendor's JDBC drivers. Invariably there are always differences between different vendor's JDBC drivers. Areas where common discrepancies occur are:

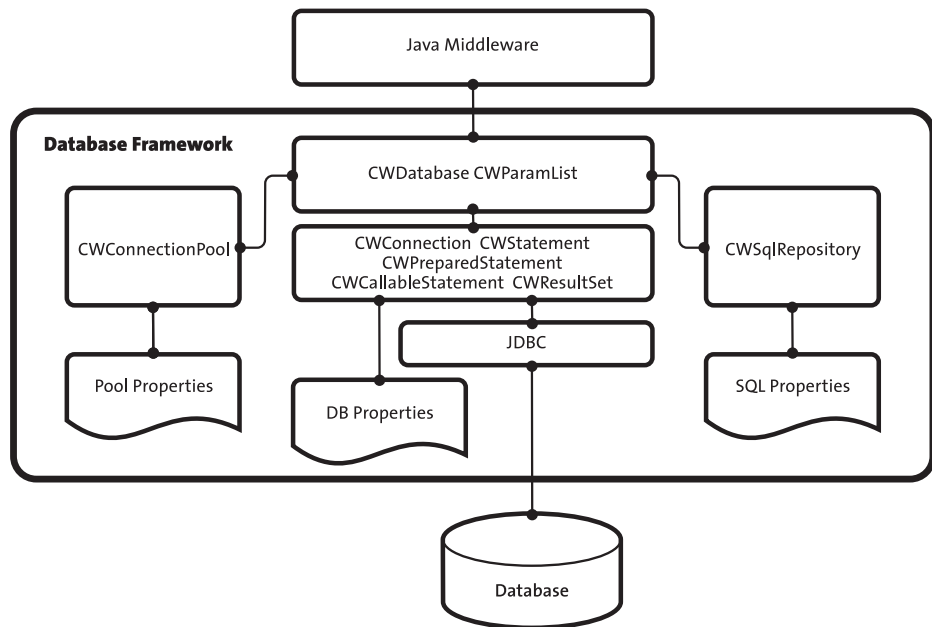
- Connection management
- Stored procedures and returning ResultSets
- Processing ResultSets
- MetaData support
- Resources being consumed due to Statements and ResultSet's not being closed
- Problems with Connections not being closed
- Performance anomalies and slow implementations
- Changes in the database optimizer from one version to the next
- New database features added from one version to the next

I've had the 'pleasure' of supporting JDBC implementations from both Sybase and Oracle, and they are an excellent contrast in approaches. I often joke that Oracle is Dad, and Sybase is Mom in their approaches. If you are heading outside without your coat on in winter, Mom will stop you and stuff you in it so you don't get cold. Dad will watch, say nothing and think that you will figure it out once you freeze your butt off. The Sybase drivers do quite a bit for you in the areas of connection, statement and result set management. Oracle will do exactly as you tell it to do. If you don't close the statement, it is probably not going to be closed automatically, and it's certainly not going to clean up the sessions, processes, and open cursors. I will not venture as to which direction is correct, I just add code to my framework to make them look similar.

AN EXAMPLE FRAMEWORK

The diagram below presents an example framework, meant to handle the JDBC issues discussed above. It also provides a layer of abstraction over the database, so developers can more quickly and safely access the database. The CWDatabase class manages all access for the developers. It utilizes the CWConnectionPool for connection management and the CWSqlRepository to manage SQL strings. The lower level Connection and Statement classes are all wrapped to provide tracking, and to guarantee that all statements and result sets are closed when a connection is checked back into the connection pool. These classes are discussed below, followed by a discussion of more advanced framework features for tracking execution performance, restricting JDBC features, and connection pool reporting. All the framework classes below are prefaced with 'CW' for my company CodeWorks Software, which allows them to be readily identified.

The Classes and Property Files that make up the Example Framework



THE DATABASE INTERFACE CLASS - CWDATABASE AND CWPARAMLIST

Arm developers with one simple class to use for database access. By creating an instance of the framework class they get a connection and simple methods for running SQL commands and stored procedures. Exception handling is properly taken care of and reported, and by using CWPParamList to allow users to create a parameter list, the DBA has tight control on Java data types and how they bind to the underlying datatypes in the database. The goal is to never allow the user to directly get a hold of a jdbc.Connection instance. An additional benefit of having such tight control is that high data rates can typically be maintained since you are controlling database access.

```
public class CWDatabase
{
    Connection m_conn = null;

    public CWDatabase()
    {
        m_conn = CWConnectionPool.checkOut();
    }

    public int executeUpdate(String queryName, CWPParamList plist)
    public int executeUpdate(String queryName)
    public ResultSet executeQuery(String queryName, CWPParamList plist)
    public ResultSet executeQuery(String queryName)

    private void processException(String msg, Throwable ex)
}
}
```

With the above class the user would run a simple query as follows:

```
public void updateSensorType()
{
    CWDatabase theDB = null;
    try
    {
        // create database and parameter list instances
        theDB = new CWDatabase();
        CWParamList plist = new CWParamList();

        // add parameters
        plist.addParameter(1,"TYPE1");
        plist.addParameter(2,1);
        plist.addParameter(3,"ACT");

        // execute the update
        int numupdate =
            theDB.executeUpdate("sensor.updateSensorType",plist);
    }
    catch( Exception exception )
    {
        CWExceptionReporter.write(this,CWExceptionReporter.FATAL,
            "Error updating sensor type.");
    }
    finally
    {
        theDB.close();
    }
    return;
} // end method
```

THE CONNECTION POOL - CWCONNECTIONPOOL

Creating connections is expensive, so by reusing connections you avoid the cost of creating a fresh one each time. At system startup you can prime the pool with several connections that are ready and available. When a user creates an instance of the database interface class, a connection is checked out. When they call close, the connection is checked back in and becomes available for others to use.

Though reuse of connections is the primary purpose of the pool, there are many background tasks and benefits. Since all connections are managed and created in one place, the DBA has the ability to strictly manage isolation levels and database options. Examples of these for SQL Anywhere include DELAYED_COMMITS, ISOLATION_LEVEL, COOPERATIVE_COMMITS, and the setting of authorization codes for OEM versions of the software.

One problem I've seen when pooling connections is that they retain some 'baggage', such that repeated reuse makes them slow down over time. I have never figured out the cause of this, but suspect it has to do with PreparedStatements, ResultSets, or some other internal tracking that is happening. Given all connections are managed by the pool it is possible to keep track of how long they have been in existence, and 'refresh' them when checked back in. By discarding connections after a set time limit you can better maintain high performance rates.

Another benefit to this tight tracking is that you can also monitor who has a connection checked out and for how long. Connections that are held for a long time, especially as member variables, can cause problems. If they are held dormant for hours they will time out causing further problems. With this additional tracking you will be able to report on who has what connection and how long they've been held open. If you know the line of code that checked out a problem connection, you will be able to identify the developer and reeducate them on how to use the framework. Since there is some expense to tracking a connection I've written my framework to have this feature off by default, but can be turned on during testing.

```

public class CWConnectionPool
{
    // maintain connections on free and out lists.
    private static ArrayList m_freePool = null;
    private static HashMap m_outPool = null;

    public CWConnectionPool()
    {
        m_freePool = new ArrayList();
        m_outPool = new HashMap();
    }

    public static void initialize()
    public static synchronized Connection checkOut()
    public static synchronized void checkIn(Connection conn)
    public static void discardConnection(Connection conn)
    private static CWConnection createConnection()
    public static reportConnectionPool()
}

```

THE SQL REPOSITORY - CWSQLREPOSITORY

SQL statements are best kept in a separate file so they can be modified without having to recompile code. For example, if a performance problem is identified, and when researched you discover that the database optimizer is picking the wrong index, a SQL hint can easily be added. To manage SQL statements I use the CWSQLRepository class. The repository also allows code reuse and since all SQL statements are in the same place the DBA can more easily find all statements that may be affected by a schema change.

```

public class CWSqlRepository
{
    private static CWSqlRepository m_SQLRepository;
    private static Properties m_SQLRepositoryTable;

    public static void initialize()
    {
        if (m_SQLRepository == null)
            m_SQLRepository = new CWSqlRepository();
        return;
    }

    private static void loadSQLRepository(String sqlfile)
    public static String getSQLString(String tag)
}

```

WRAPPING JDBC CLASSES

When it comes to simplifying database access, much can be gained from providing those few simple classes I discussed above. However, when it comes to correcting lower level discrepancies with a JDBC implementation there is no sense reinventing the wheel. Problems can be handled and functionality can be added by wrapping JDBC classes. JDBC is well documented, developers are familiar with it, you are familiar with it, and it is easy to teach people and to provide examples of proper use. By creating wrapper classes you can add whatever tracking, timing, and reporting you want, and you can correct discrepancies. Only in the rarest of cases will a developer ever know they are using your framework implementation of Connection.prepareStatement() and not the real thing.

CWCONNECTION

The most important class to wrap is `java.sql.Connection`. Here you can keep track of how long a connection has been checked out and keep a list of all statements that were created. For debug purposes, you can maintain the stack trace from when the connection was created, so that you can better pinpoint the code that created the connection when 'connection abuse' is discovered. For example, when a connection is checked out for extended periods of time.

```
public class CWConnection implements Connection
{
    // Basic info for the connection
    private Connection _conn = null;    //the wrapped java.sql.Connection
    private int _connNum ;              // a tracking number
    private long _createTime;           // time created
    private ArrayList _stmtTracker;     // for tracking statements

    public int getConnectionNum()
    public int getElapseTime()
    public void closeStatements()

    private ArrayList getStatementTracker()
    private void clearStatementTracker()
    String reportConnection()

    // wrapped jdbc methods
    public Statement createStatement() throws SQLException
    {
        CWStatement stmt = new CWStatement(_conn.createStatement());
        _stmtTracker.add(stmt);
        return stmt;
    }
}
```

CWSTATEMENT, CWPREPAREDSTATEMENT, CWCALLABLESTATEMENT

Your framework can be written such that you rarely allow developers to get a hold of `Statements`, `CallableStatements`, and `PreparedStatements`. The primary reason to wrap these classes is to track timing, and to track the `ResultSet` instance if one is returned. Though I discuss a `Serialized ResultSet` below, I don't recommend hiding `ResultSets` from developers, since their interface is well documented. The key abuse is to leave a result set open, but tracking them is rather simple. When a `Connection` is checked in, all statements are closed and each statement will guarantee that their result set is closed. I still wrap `ResultSet`, but the primary purpose is to remove slow performing methods so developers can not use them. This is covered later in this paper.

```
public class CWStatement implements Statement
{
    // Basic info for the statement
    private Statement m_stmt = null;    //the wrapped java.sql.Statement
    private int m_stmtNum;              // a tracking number
    private long m_createTime;          // time created
    protected ResultSet m_rsTracker;   // track the result set
    private String m_sqlTracker;        // Sql issued with this statement
    public String reportStatement()

    public CWStatement( Statement stmt, int stmtNum)
    {
        m_stmt = stmt;
        m_stmtNum = stmtNum;
        m_createTime = System.currentTimeMillis();
        m_sqlTracker = null;
        m_rsTracker = null;
    }
}
```

```

public ResultSet executeQuery(String sql) throws SQLException
{
    CWResultSet rs = new CWResultSet(m_stmt.executeQuery(sql));
    m_rsTracker = rs;
    m_sqlTracker = sql;

    return rs;
}
}

```

ADVANCED FRAMEWORK IDEAS

The topics discussed above have concentrated on simplifying the interface to the database, connection management and providing safe guards to common JDBC problems. The next few sections address additions to the framework to provide greater monitoring and control over developers using the database, including:

- Fixing problems with result sets
- Restricting slow performing operations
- Monitoring SQL performance
- A connection pool console

CWRESULTSETSERIALIZED

Many Java programmers don't realize (or forget) that ResultSets are actually cursors to the database. They pass around references, store them as member variables, and often never close them, thus tying up database resources. To remove all risk, a Serialized ResultSet is provided that can be created from a JDBC ResultSet. This also allows a result set to be sent via RMI between processes.

In an extreme case, a result set causing blocking problems was passed so often that I could not find where to close it. Once replaced with the serialized version I was able to close the real ResultSet and all problems disappeared. Any implementation of this needs to set a limit on the number of rows that can be created, since a million row serialized ResultSet will cause performance problems. I start with a limit of 5000 rows.

The code below shows poor management of the result set, since it is passed out of the method. It is now difficult to know if it is closed or not, since the method only closes the database instance upon an error. The code will work, however the connection will be left checked out and the cursor left open if the result set is not closed in the calling method.

```

public ResultSet getAllSensors()
{
    CWDatabase theDB = null;
    ResultSet rs = null;
    try
    {
        // create database and parameter list instances
        theDB = new CWDatabase();

        // execute the query
        rs = theDB.executeQuery("sensor.getAllSensors");
    }
    catch( Exception exception )
    {
        CWExceptionReporter.write(this, CWExceptionReporter.FATAL,
            "Error during query: " + " sensor.getAllSensors ");
        theDB.close();
    }
    return rs;
} // end method

```

Since problems like this are sometimes found too late in the development cycle to correct by rewriting the method, you can correct the problem by passing back a serialized result set and closing the database instance in a 'finally' block, thus guaranteeing that everything is gracefully closed. I still believe the DBA should make life painful for the engineer who caused the problem, but admit that one day before the system's code freeze date is not a time to change a lot of code. The changes made are shown in bold below.

```
public ResultSet getAllSensors()
{
    CWDatabase theDB = null;
    CWResultSetSerialized rss = null;
    try
    {
        // create database and parameter list instances
        theDB = new CWDatabase();

        // execute the update
        ResultSet rs = theDB.executeQuery("sensor.getAllSensors");
        rss = new ResultSetSerialized(rs);
    }
    catch( Exception exception )
    {
        CWExceptionReporter.write(this,CWExceptionReporter.FATAL,
            "Error during query: " + " sensor.getAllSensors ");
    }
    finally
    {
        theDB.close();
    }
    // return the serialized version
    return rss;
} // end method
```

PERFORMANCE LEVELS

Depending on your performance requirements, you could range from, 'Only the fastest database access is allowed' to 'I don't care, use any and all features'. Most will fall somewhere in between. Having the ability to 'shut off' known slow-performing JDBC methods is very helpful. For example using `ResultSet.update()` is much slower than running a separate `Statement.execute()` to do the same update. Allowing users to use non 'forward only' result sets using methods like `rs.last()` can also be slow. I use three basic performance levels:

- Level 1 - Developers are not allowed access to Connections or the ability to issue dynamic SQL. All slow performers are turned off, including result set updates and access to metadata.
- Level 2 - Dynamic Queries are allowed, but all other slow performers are still restricted. This is the default in my framework.
- Level 3 - Full JDBC access; nothing is prohibited.

When turning off JDBC features, I recommend issuing an Exception, which explains that a feature is turned off and for them to contact the DBA. During development the DBA can determine if the feature is truly needed and decide whether to add it back to the framework, or to change the performance level on the feature.

```
public class CWResultSet implements ResultSet
{
    private ResultSet m_rs;
    private long m_createTime;
    private int m_perf_level;
```

```

public CWResultSet( ResultSet rs, perf_level)
{
    m_rs = rs;
    m_createTime = System.currentTimeMillis();
    m_perf_level = perf_level;
}

public void updateRow() throws SQLException
{
    if(perf_level < CWDatabase.HIGH_PERF_ONLY)
    {
        CWExceptionReporter.write(this,CWExceptionReporter.ERROR,
            "Use of method prohibited due to slow performance, "
            + " as per the DBA.");
    }
    m_rs.updateRow();
}
}

```

It would be pretty simple to give a name to each part of JDBC functionality you wish to turn on or off and utilize a property file to control these values. However, I find I don't need this level of flexibility and keep with basic levels.

SQL PERFORMANCE TRACKING

The old saying 'you can't manage it, if you can't measure it' rings true for query tuning. Though SQL Anywhere provides the ability to monitor queries, I still provide statement timing and reporting. This allows the DBA to turn on tracking for specific SQL statement, all statements, or to set a timing threshold to report on queries that exceed the given threshold. I'm also looking into timing at the result set level, such that ResultSets that exceed a threshold can be identified. This will be helpful, since timing the execution of a Statement only gives you the elapse time of the first row being returned, unless there is a 'order by' or similar clause in the statement. Much like Connection tracking, I default to having this turned off, and utilize it during the testing phase.

```

public int executeUpdate(String queryName)
{
    private long startTime;

    <snip>
    // execute the update
    startQueryTimer();
    int numupdate =
        theDB.executeUpdate queryName,plist);
    stopQueryTimer(queryName);
}

```

If query tracking is turned on, query times will be logged to the database.

CONNECTIONPOOL COMMAND CONSOLE

The capturing of stack traces as discussed earlier can help in debugging problem connections by printing out the trace in a debug statement. A far more advanced capability is to telnet into the connection pool and generate reports that show connections that are checked out and those on the free pool. You can also add reporting on Statements and the last run SQL statement. This feature is typically used in a support mode, where the DBA can turn tracking on either in their lab or at a customer site in order to recreate the problem. During the recreation, they can login to the ConnectionPool and monitor what is actually happening, and determine if connection management is the cause. The implementation of the console capability is well beyond the scope of this paper, but worth thinking about.

CONCLUSION

As a Development DBA it is your responsibility to protect the database. By adding a solid, well tested layer between the “evil” Java developers and your beloved database, you can greatly reduce risk and provide additional tools for tracking and reporting problems.

The ideas and sample framework provided are a starting point. Every time an engineer does something new and different that causes a problem for the database, you should immediately evaluate whether you can prohibit or report on the issue should it happen again. If you expect the need to move between drivers, this layer of abstraction can make it far easier, and in the extreme case it can allow a move between database platforms.

ABOUT THE AUTHOR

Todd Loomis is an independent consultant, who specializes in database design and development frameworks. He has designed systems in many areas to include the aerospace, legal, financial, energy, computer networking and pharmaceutical industries. He has been developing and supporting products using SQL Anywhere for over 6 years. An open source Java database framework is available at his website: www.DatabaseGunForHire.com, or by contacting him at t.loomis@comcast.net.

iANYWHERE SOLUTIONS, INC.
WORLDWIDE HEADQUARTERS
ONE SYBASE DRIVE
DUBLIN, CA 94568-7902
U.S.A.

CONTACT_US@iANYWHERE.COM
NORTH AMERICA
T 1-800-801-2069
1-519-883-6898
EUROPE, MIDDLE EAST, AFRICA
+44 1628 597 100
ASIA PACIFIC
+852 2506 8700
JAPAN
+81 3 5210 6380

www.iAnywhere.com

iANYWHERE SOLUTIONS IS A SUBSIDIARY OF SYBASE, INC. COPYRIGHT © 2006 iANYWHERE SOLUTIONS, INC. ALL RIGHTS RESERVED. iANYWHERE, SYBASE, AND THE SYBASE LOGO ARE TRADEMARKS OF SYBASE, INC. OR ITS SUBSIDIARIES. ALL OTHER TRADEMARKS ARE PROPERTIES OF THEIR RESPECTIVE OWNERS. ® INDICATES REGISTRATION IN THE UNITED STATES OF AMERICA.

SYBASE®
iAnywhere