Database Application Development Using Borland® Kylix™ and IBM® DB2® Universal Database v7.2

Whitepaper

Version 1, January 2002

Paul Yip
DB2 Business Partner Enablement Services
IBM Toronto Labs

Ramesh Theivendran
Technical Lead, Connectivity
Borland RAD Products

Acknowledgements

Technical Editing:

Bill Wilkins - DB2 UDB Vendor Enablement, IBM Toronto Labs

Introduction

Borland Kylix is a Rapid Application Development (RAD) tool that provides a thin, cross-platform, database independent, disconnected data-access model (dbExpress) that can be used to quickly develop user-interactive, event-driven database applications for Linux and Windows platforms (Borland Delphi).

IBM DB2 Universal Database (DB2) is an industry-leading database management system used by the world's largest and most advanced application developers. DB2 is supported on a wide range of distributed platforms including Linux, AIX, HP-UX, Sun Solaris and Windows operating systems (same code base).

By combining Kylix and DB2, you can write applications once and deploy them to run natively on both Linux and Windows without any code changes for optimal performance and flexibility.

This document is intended to highlight the ways to best integrate Kylix and DB2. It is recommended that this document be reviewed before any application development actually takes place, so that optimal practices documented in this paper are under consideration at application design time.

This is a working document: new topics, tips and ideas will be added as we learn from experience and hear from contributors. If you have any comments or suggestions, we encourage you to provide feedback to **Paul Yip (ypaul@ca.ibm.com) or Ramesh Theivendran (rtheivendran@borland.com)**

We present this paper in two parts: Part I is a description of the underpinnings of the Kylix dbExpress database access layer and Part II highlights the best practices to integrate Kylix and DB2, using a scenario/solutions approach so that it is easier to identify problems and find possible solution(s).

Who Should Read This Document?

This document is intended for application developers who wish to write Kylix applications for DB2 and those who have an existing application written for another database and wish to understand how to optimize it for DB2. We assume that you are working with DB2 UDB v7.2 Personal Edition (PE) or higher. Note that DB2 UDB v7.1 with FixPak 3 is equivalent to DB2 UDB v7.2, although we recommend that you use the latest available patches for both products. Details of DB2 UDB Enterprise-Extended Edition (EEE) are not covered, but applications are generally portable without modification from EE to EEE.

We assume that readers are familiar with Kylix, SQL, and have basic DB2 skills.

Part 1: n-tier data-access with dbExpress

Data access, data remotting and data manipulation are the three main components of a n-tier data access model. Kylix and Delphi both come with a rich set of components for database connectivity, making it a snap for developers to write n-tier database applications.

Data access:
Kylix and Delphi 6 introduces dbExpress, a new cross-platform, database-independent and extensible interface for dynamic SQL processing. Some of the benefits dbExpress provides when compared to other data-access technologies such as BDE (Borland Database Engine), ODBC, JDBC and ADO are as follows:

dbExpress returns only unidirectional cursors and therefore does no caching on the data retrieved. The DataSnap component, with the ClientDataset technology, can be used for caching, scrolling, indexing, and filtering for the result set on each client.

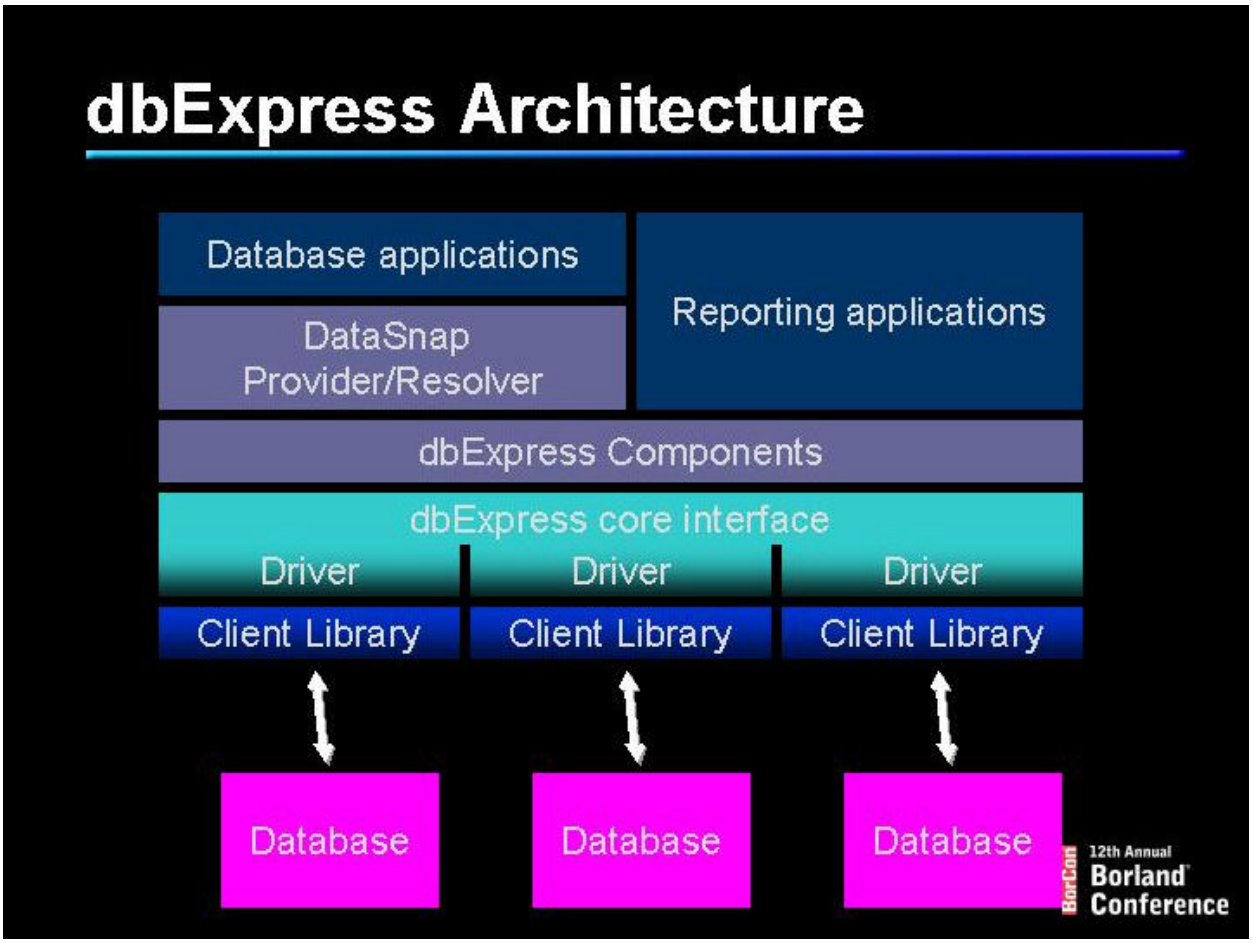
dbExpress does no metadata caching, and the design time metadata access interface is implemented using the core data-access interface. DataSnap components retrieve minimum runtime metadata so that datasets can be resolved back to the databases efficiently.

Runtime performance with SQL databases is affected by the internal query generation for navigation, Blob access, and metadata retrieval. dbExpress executes only queries requested by the user, thereby optimizing database access by not introducing any extra queries.

dbExpress manages a record buffer or a block of record buffers internally and provides clients with individual field values. This is less error-prone than managing the client-allocated record buffer.

In summary, dbExpress uses:

- Improved runtime performance
- Less configuration
- Easy deployment
- Enhance access to database specific features
- Easy adaptation to new data sources



Data remotting and manipulation:

The DataSnap architecture that ships with Kylix and Delphi provides the functionality needed for data remotting and manipulation in an n-tier model. Dataset provider, resolver and ClientDataset are the three main components of DataSnap. Data from dbExpress is streamed into a data-packet in XML or binary format and transferred to the client application. ClientDataset presents the data to the client application and tracks changes made to the data. When the client decides to persist the updates, the changes are streamed into a delta packet (XML or binary format) and transferred to the resolver. Finally, the resolver takes these changes and persists the data on to the database server.

Part 2: Integration tips for Kylix DB2

This section discusses the lesser known gems that are available to take DB2 and Kylix integration further, which can lead to enhanced performance of your applications. We use a scenario/solutions approach to present the topics so that it is easier to identify problems find possible solution(s).

Multi-user application experiencing locking/hanging problems.

Description:

Lock-waits are a common but not unavoidable problem with multi-user applications that must share the same data. If one user is updating data without frequently committing, other users who also need that data can be locked out until the first user commits the changes. There are many potential solutions to this problem and they revolve around the following rules:

- Do not show/fetch the user any more data than what is needed for the user to get the work done
- Make the application issue commits as often as possible
- Do not allow users the ability to control when a transactions begins and ends
- Update using a unique key of a table whenever possible
- Use the minimum isolation level that is needed for your queries/application

Lock-waits would occur, for example, in the following sequence of events:

User 1: Updates a row in table T1 without immediately committing, decides to go to lunch, and then does some shopping

User 2: Issues a SELECT statement that qualifies the row just updated by User 1, but experiences an application lock-wait because DB2 cannot respond with data until User 1 commits his or her work. The lock-wait will continue (indefinitely, by default) until User 1's application commits (perhaps a few hours later).

User 1: Transaction commits (or issues rollback).

User 2: Application resumes.

If you experience an application hang and want to determine if it is due to a database lock-wait, type:

```
db2 list applications show detail
```

A list of application connections and their status will be displayed. Any connections with a state of ***lock-wait*** are those that are hanging due to lock contention.

Note: You will need SYSADM, SYSCTRL, or SYSMAINT privilege, and you will need to first

ATTACH to the instance if working with a remote database.

Solution 1: Commit frequently and use only short transactions

The best solution to this problem is to ensure that all transactions are controlled by the application, are short (execute sub-second), and commit frequently, especially when that data must be updated/inserted frequently and shared by many users.

Solution 2: Use Uncommitted Read (UR) statement level isolation

If your business logic allows, or if the risk is tolerable, make use of statement level isolation. The default isolation used by DB2 is Cursor Stability, which does not allow applications to see dirty data (uncommitted data that has been updated or inserted by another application).

For example:

Suppose you wish to display the results of the following query in a DBGrid Control :

```
SELECT * FROM T1
```

And the expected result contains many rows that may be in the process of being updated by others. The way the query is currently written, the application will lock-wait until all applications have committed whenever a DBGrid refresh is issued. A possible solution then, is to modify the query as follows:

```
SELECT * FROM T1 WITH UR
```

SELECT statements with UR statement level isolation will never lock-wait. But, you may see uncommitted, changed, or new rows of data that have not been committed by other users. This method can be messy, however, if you have to change the syntax of potentially large numbers of statements.

Solution 3: Do not use more connections than you need

The default isolation level in DB2 is Cursor Stability, which does not allow one to see dirty data. For a given user, do not use one connection to view data and another to do updates. Keep in mind that if your application maintains ***n*** connections for a given user, it will appear to DB2 as though ***n*** independent users are connected and therefore must be protected from each other's activities.

Solution 4: Use DB2_RR_TO_RS registry variable

Next-key locking (used by DB2 to support Repeatable Read isolation level) can sometimes be a source of lock-waits. If your applications do not use Repeatable Read isolation level, it is safe to set the DB2_RR_TO_RS registry variable to reduce next-key locking. Note that this change is instance-wide

and therefore would apply to all databases in that instance. If applications do try to use Repeatable Read isolation level, they are automatically downgraded to Read Stability isolation level.

To reduce next-key locking, from the command line, type:

db2set DB2_RR_TO_RS=yes

Then restart the instance for changes to take effect.

db2stop
db2start

Solution 5: Bypass locked rows with an index

Write your SQL and create indexes such that minimal rows are read to build the result set. This can increase the chances that exclusive(X) row locks can be by-passed using the available indexes. Creating indexes for columns that represent identity values is obviously beneficial. The benefits of creating indexes on a non-unique column for small tables, however, are not as obvious but will reduce the opportunities of lock-waits.

To illustrate, consider the following small table, with an index on column id, but no index on column val:

T1

id	val
1	a
2	b
3	c
4	d
5	e

Suppose there are two users, and the following sequence of events occur:

User 1: UPDATE T1 SET val = 'X' where id=3 (not yet committed)

User 2: SELECT id from T1 where val='e' (will lock-wait)

User 2 will lock-wait until User 1 commits because the only way DB2 can build the result set to satisfy the query is by using a tablescan (which means to read every row from top to bottom). During this operation, User 2 will wait on the row exclusively locked by user1. The table is small, and so under normal circumstances, an index would not make sense on column ***val***. In this situation, however, creating an index on ***val*** would allow User 2 to continue without lock-wait because an index could be used to directly retrieve the row required.

After creating an index, be sure to do 'runstats' on the table to make DB2 aware that the index is available for use. If you make use of stored procedures (or any static SQL application), rebuild or rebind the appropriate packages.

Solution 6: Adjust lock-wait timeout

DB2, by default, will allow applications to lock-wait forever. In other words, applications can be allowed to hang indefinitely until the locks the application needs have been freed. You can change this behavior by updating the database configuration file (db cfg)

From the command line, using a database administrative id, issue:

db2 update db cfg for <dbname> using locktimeout 10

With locktimeout changed to 10, applications with transactions that wait on locks longer than 10 seconds will be rolled back. The application can handle the database exception and have the user try the operation again later. You can, of course, use any value of locktimeout that you wish (using -1 will cause DB2 to return to its default behavior of waiting indefinitely).

Setting this value will improve the end-user experience because a well-handled rollback is better than a lengthy database lock-wait (which will appear to the user as though the system has hanged and the user may be tempted to kill the application or reboot the machine).

Note: For this database configuration change to take effect, all applications must disconnect from the database.

Application has slow response times due to large result sets being returned

Description

Your application may need to allow the user to see the contents of a table using a query that can potentially contain hundreds (or more) rows. The user may be able to manage only tens of rows at a time. But no matter how you restrict the query, the resultset can still potentially return many more rows than are practical. This creates a poor user-experience as DB2 will take longer to return data for large resultsets, and Kylix takes longer to cache the large amount of rows.

Your query may have the form of:

SELECT * FROM T1 WHERE ***condition***

...where ***condition*** makes no guarantees about the size of the resultset.

Solution 1:

An easy solution would be to modify the query as follows:

SELECT * FROM T1 WHERE ***condition*** OPTIMIZE FOR ***n*** ROWS

This tells DB2 to execute the query using an access plan that fetches the first n qualifying rows as quickly as possible. While the user works with the first n rows, DB2 will continue to process the remaining rows in the background. Keep in mind that when this query modifier is used, the cost of retrieving the remaining rows could be higher than if the modifier was not used. (This might not matter since what does matter is the user *experience*. In the time it took the user to process the first ***n rows***, DB2 has likely successfully retrieved all remaining rows in the background.)

Solution 2:

Kylix uses the DB2 CLI native interface to interact with DB2 and the cursors opened are never used to do positional updates. Therefore, we can make our queries unambiguous and allow DB2 to use block fetching (fetch rows in groups rather than row by row) to significantly improve SELECT performance.

To do this, use the modifier FOR READ ONLY in your select statements. For example:

SELECT * FROM T1 WHERE ***condition*** FOR READ ONLY

To combine solutions 1 and 2 for this scenario, you can use:

SELECT * FROM T1 WHERE ***condition*** OPTIMIZE FOR ***n*** ROWS FOR READ ONLY

The schema used to create tables/views does not match user IDs

Description:

This is a common problem for two-tier applications (or n-tier applications that have complex deployment requirements) where many users connect to the database with their own user IDs, but the tables are all created with a single schema. The problem here is that when a user connects to a DB2 database, the user's schema defaults to the user's ID. For example, if all the tables for a given application are created under schema X, and user User 1 does the following query:

SELECT * FROM T1;

DB2 will interpret the command as SELECT * FROM USER1.T1 because the table name was not qualified with X and will therefore fail.

Note: To solve this problem, you could be tempted to fully qualify all your table names in the application itself (hardcode it). For flexibility and maintainability, this is inadvisable unless the schema name will never change (this is unlikely if you are creating a re-sellable application). In each of the potential solutions that we present below, we assume that no queries use fully qualified names.

Solution 1:

For every new connection, issue:

SET CURRENT SCHEMA <target schema>

All subsequent queries will assume the new schema for unqualified database objects.

Solution 2:

You can create aliases for tables that the user should be allowed to access. For example, if the tables that support your application are called:

ACCT.table1
ACCT.table2
ACCT.view1

You would alias each table, for each user, by doing:

CREATE ALIAS <userid>.table1 FOR ACCT.table1
CREATE ALIAS <userid>.table1 FOR ACCT.table1
CREATE ALIAS <userid>.table1 FOR ACCT.view1

Solution 3:

Modify the DB2 CLI configuration file, db2cli.ini, located in the ***sqllib*** directory of each client machine and add the following lines.

[myAppDB]
CURRENTSCHEMA=X

Note: ensure that there is a blank line after he last line of the file, or else the db2cli.ini file might not get parsed properly by DB2.

In the above example, myAppDB is the database name enclosed in square brackets. CURRENTSCEMA=X tells DB2 to automatically change the default schema name to X, regardless of what user ID is used to connect. This schema change applies to the client, and for all new connections

after the setting has been applied.

Want to alias columns, but aliased columns are not updateable

Description:

At the database level, column names of tables are sometimes shortened and/or abbreviated**.** For users, the columns identified by a SELECT statement can be aliased as part of the query to make column names more user-friendly. For example, a table for maintaining student data may be created as follows:

```
Create table Student (sid int, fname varchar(20), lname varchar(20))
```

A query that uses column aliases to retrieve the data may look like this:

```
SELECT sid as StudentId, fname as FirstName, lname as LastName from STUDENT
    WHERE .....
```

You can quickly discover that a problem arises when you want to do UPDATEs, which do not work when columns are aliased, since Kylix does not use a cursor to perform positional updates. Direct UPDATE statements based on the original SELECT query are generated at runtime. For example, upon ApplyUpdates(), the following query is generated if you were to use the above SELECT statement and modified the first and last name of the row with studentid=1000.

```
UPDATE student SET FirstName='foo', LastName='bar' WHERE StudentId=1000
```

Of course, the above query will fail since the aliased names are used instead of the real column names.

Solution 1: An easy solution is to avoid using shortened and/or abbreviated column names. Then aliasing is never required.

Solution 2:

In DB2 (and most other databases), views are deleteable, updateable and insertable as long as certain conditions are met. For a description of these conditions, refer to the DB2 SQL reference for CREATE VIEW.

In the scenario above, a view could be created as follows:

```
Create view student_v as
    SELECT sid as StudentID, fname as FirstName, lname as LastName
    FROM STUDENT WHERE ....
```

And now, the new query will look like this:

SELECT * FROM STUDENT_V .....

Common errors and solutions

This section discusses common problems encountered by Kylix developers while running DB2.

Table/View does not exist (SQL0204N <tabname> is an undefined name. SQLSTATE=42704**)**

Some database management systems, including DB2, differentiate between upper case and lower case table names. DB2 object names are case sensitive only if they are created with quotations around the name. For example, if you did:

CREATE TABLE “t1” (c1 int)
CREATE TABLE “T1” (c1 int)

two tables would be successfully created. However, if quotes are not used, DB2 will fail on the second command.

The best practice for dealing with DB2 objects is to use upper-case names in your queries and not rely on case sensitivity at all when creating objects (i.e., do not use quotes). If you never use quotations to create your objects, DB2 behaves as if it is a case-insensitive database. Kylix requires that you specify object names in upper case.

Another common mistake that Kylix and Delphi developers have reported is the improper use of syntax to reference schema and table name when using quotation marks. For example, you may write your SQL to reference table abc.tabname using the following query:

SELECT * from “abc.tabname”

The above query will fail because the proper way to use quotation marks for schemas and table names is:

SELECT * from “abc”.”tabname”

Summary

In summary, the combination of Kylix and DB2 is a powerful solution for developing cross-platform applications. In this document, we have discussed how to best integrate Kylix and DB2 to maximize concurrency, flexibility and overall performance.

Notices, Trademarks, Service Marks and Disclaimers

This document contains proprietary information of IBM. The information contained in this publication does not include any product warranties, and any statements provided in this document should not be interpreted as such.

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States and/or other countries: IBM, DB2, DB2 Universal Database,

The following terms are trademarks or registered trademarks of Borland Software Corporation in the United States and/or other countries: Borland, Kylix, Delphi, dbExpress, DataSnap.

Windows and Windows-based trademarks and logos are trademarks or registered trademarks of Microsoft Corporation.

Other company, product or service names may be the trademarks or service marks of others.

The furnishing of this document does not imply giving licence to any IBM patents.
References in this document to IBM products, Programs, or Services do not imply that IBM intends to make these available in all countries in which IBM operates.