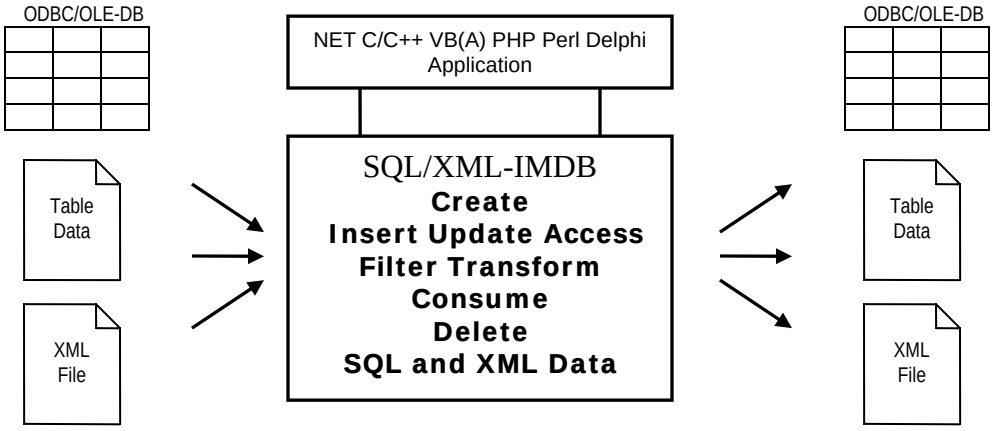
SQL/XML-IMDB

In Memory SQL / XML Database Component for Universal Data Management

User's Guide
V 4.1

www.quilogic.cc



Part No DOC-2007-19
Version 4.1

Contact

You can contact us via any of the following paths

Web	www.quilogic.cc
Support	support@quilogic.cc
Sales Inquiries	sales@quilogic.cc
Executive Office	office@quilogic.cc
FAX	+43 (533) 93544
Telephone	+43 (533) 93544

Before requesting support, it would save both your time and ours if you could do the following:

- Make sure you have read any relevant portions of the manual and the file readme.txt.
- Isolate the problem to a small test case
- Have the version number ready (see readme.txt)
- Have the type of environment, compiler, version number and operating system ready.
- Give us an example of the faulting SQL or XQuery statement including all necessary DDL statements.

License Agreement

This software package and its documentation are subject to the following license agreement. By installing and using the package, you are implicitly accepting these terms and conditions:

END-USER LICENSE AGREEMENT FOR SQL/XML-IMDB SOFTWARE

IMPORTANT-READ CAREFULLY.

This QuiLogic, Inc**. SQL/XML-IMDB** End-User License Agreement ("EULA") is a legal AGREEMENT between you (either as a registered individual user or as the registered user/representative and on behalf of a single entity, “Licensee”) and QuiLogic Software Corporation for the SQL/XML-IMDB software product identified above, which product includes computer software and may include associated media, printed materials, and "online" or electronic documentation ("SOFTWARE PRODUCT"). By installing, copying, or otherwise using the SOFTWARE PRODUCT, you agree to be bound by the terms of this EULA. If you do not agree to the terms of this EULA, then DO NOT install or use the SOFTWARE PRODUCT.

SOFTWARE PRODUCT LICENSE

1) GRANT OF LICENSE. The SOFTWARE PRODUCT is licensed, not sold. This EULA grants you, the registered computer software developer, the following rights:

Permission is given to the buyer of this software package for one software developer (Developer Seat) to use this software on one CPU (one workstation) and to make unlimited backup copies. You may utilize and or modify this software package for use in your projects. You may distribute and sell any executable which results from using this software package in your applications, except a utility program, library or application of similar nature to this product. You my redistribute this product in the form of a dynamic linked library, but solely to be used with your compiled executable product for the purpose of dynamic loading and /or linked to your application through the static library. You may NOT redistribute the header files, object modules, or static libraries of object modules in any form, put them on a bulletin board or sell them.

You may not use, copy, modify, merge, or transfer the software package, except as expressly provided above in this agreement.

This material is sold “as is”. QuiLogic, Inc. makes no warranties, either expressed or implied, regarding the enclosed software package, its merchantability, or its fitness for any particular purpose. Information in this document is subject to change without notice and does not represent a commitment on the part of QuiLogic, Inc. While every effort is made to insure that the above mentioned product and its documentation is free of defects, QuiLogic, Inc. shall NOT be held responsible for any loss of profit or any other commercial damage, including but not limited to special, incidental, consequential or other damages occasioned by the use of this product.

It is assumed that purchasers of this product are familiar with basic programming skills. This is a highly technical product, offered in a rapidly evolving programming environment. QuiLogic, Inc. will provide support to purchasers of this product for 365 days after its purchase and receipt (bug reports and comments are always welcome). Support questions may be submitted either by e-mail or fax. QuiLogic, Inc. reserves the right to respond to questions in responding by e-mail or fax.

Volume License Use: If the Software is licensed with volume license terms specified in the applicable product invoicing or packaging for the Software, you may make, use and install as many additional copies of the Software on the number of Developer Seats as the volume license terms specify. You must have a reasonable mechanism in place to ensure that the number of Developer Seats on which the Software has been installed does not exceed the number of licenses you have obtained. This license authorizes you to make or download one copy of the Documentation for each additional copy authorized by the volume license, provided that each such copy contains all of the Documentation's proprietary notices.

Enterprise License Use: If the Software is licensed with enterprise license terms specified in the applicable product invoicing or packaging for the Software, you may make, use and install as many additional copies of the Software on the unlimited number of Developer Seats within Licensee's organization. You must have a reasonable mechanism in place to ensure that the number of Developer Seats on which the Software has been installed is controlled for reference and audit purposes. This license authorizes you to make or download one copy of the Documentation for each additional copy authorized by the enterprise license, provided that each such copy contains all of the Documentation's proprietary notices.

2) DESCRIPTION OF OTHER RIGHTS AND LIMITATIONS.

Limitations on Reverse Engineering, Decompilation, and Disassembly. You may not modify, reverse engineer, decompile, or disassemble the SOFTWARE PRODUCT, except and only to the extent that such activity is expressly permitted by applicable law notwithstanding this limitation. The SOFTWARE PRODUCT is licensed as a single product. Except with respect to the Redistributables, its component parts may not be separated for use on more than one computer.

Not for Resale Software. If the SOFTWARE PRODUCT is labeled "Not for Resale" or "NFR" or "Evaluation Copy", then, notwithstanding other sections of this EULA, you may not use the SOFTWARE PRODUCT for commercial purposes nor sell, or otherwise transfer it for value. Commercial purposes include the use of the SOFTWARE PRODUCT to create publicly distributed computer software.

Rental. You may not rent, lease, or lend the SOFTWARE PRODUCT to any party.

Software Transfer. You may permanently and wholly transfer all of your rights under this EULA, provided you (a) retain no copies (whole or partial), (b) permanently and wholly transfer any and all of the SOFTWARE PRODUCT (including all component parts, the media and printed materials, any upgrades, this EULA, and, if applicable, the Certificate of Authenticity) to the recipient, and (c) the recipient first agrees to abide by all of the terms of this EULA. If the SOFTWARE PRODUCT is an upgrade, any transfer must include any and all prior versions of the SOFTWARE PRODUCT and any and all of your rights therein, if any.

Support Services. QuiLogic, Inc. may provide you with support services related to the SOFTWARE PRODUCT ("Support Services"). The provision and use of Support Services is governed by the QuiLogic, Inc. policies and programs described in the SOFTWARE PRODUCT user manual and/or in "online" documentation. Any supplemental software code provided to you as part of the Support Services shall be considered part of the SOFTWARE PRODUCT and subject to the terms and conditions of this EULA. With respect to technical information you provide to QuiLogic, Inc. as part of the Support Services, QuiLogic, Inc. may use such information for its business purposes, including for product updates and development.

Termination. Without prejudice to any of QuiLogic's other rights, QuiLogic, Inc. may terminate this EULA if you fail to comply with the terms and conditions of this EULA. In such event, you must destroy any and all copies of the SOFTWARE PRODUCT and all of its component parts.

3) UPGRADES. If the SOFTWARE PRODUCT is labeled or otherwise identified by QuiLogic, Inc. as an "upgrade", you must be properly licensed to use a product identified by QuiLogic, Inc. as being eligible for the upgrade in order to use the SOFTWARE PRODUCT. A SOFTWARE PRODUCT, labeled or otherwise

identified by QuiLogic, Inc. as an upgrade, replaces and/or supplements the product that formed the basis for your eligibility for such upgrade. You may use the resulting upgraded product only in accordance with the terms of this EULA. If the SOFTWARE PRODUCT is an upgrade of a component of a package of software programs that you licensed as a single product, the SOFTWARE PRODUCT may be used and transferred only as part of that single product package and may not be separated for use on more than one computer.

4) COPYRIGHT AND TRADEMARKS.

All title, trademarks and copyrights in and pertaining to the SOFTWARE PRODUCT, the accompanying printed materials, and any copies of the SOFTWARE PRODUCT, are owned or licensed by QuiLogic, Inc. or its affiliated companies. The SOFTWARE PRODUCT is protected by copyright and trademark laws and international treaty provisions. You may make one copy of the SOFTWARE PRODUCT for back-up and archival purposes. You may not copy the printed materials accompanying the SOFTWARE PRODUCT.

You may not remove, modify or alter any QuiLogic, Inc. copyright or trademark notice from any part of the SOFTWARE PRODUCT, including but not limited to any such notices contained in the physical and/or electronic media or documentation, in the QuiLogic, Inc. Setup Wizard dialogue or ‘about’ boxes, in any of the runtime resources and/or in any web-presence or web-enabled notices, code or other embodiments originally contained in or otherwise created by the SOFTWARE PRODUCT.

5) DUAL-MEDIA SOFTWARE. You may receive the SOFTWARE PRODUCT in more than one medium. Regardless of the type or size of the medium you receive, you may use only that one medium that is appropriate for your single computer. You may not use or install the other medium on another computer, including but not limited to portable computers under the exclusive control of the registered developer. You may not loan, rent, lease, or otherwise transfer the other medium to another user, except as part of the permanent transfer (as provided above) of the SOFTWARE PRODUCT.

6) AUSTRIAN GOVERNMENT RESTRICTED RIGHTS. The SOFTWARE PRODUCT and documentation are provided with RESTRICTED RIGHTS. This EULA shall be construed, interpreted and governed by the laws of the Austrian country.

7) HIGH RISK ACTIVITIES. The Software is not fault-tolerant and is not designed, manufactured or intended for use or resale as on-line control equipment in hazardous environments requiring fail-safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, direct life support machines, or weapons systems, in which the failure of the Software could lead directly to death, personal injury, or severe hysical or environmental damage ("High Risk Activities"). QuiLogic and its suppliers specifically disclaim any express or implied warranty of fitness for High Risk Activities.

8) LIMITED WARRANTY. QuiLogic, Inc. warrants that (a) the SOFTWARE PRODUCT will, for a period of ninety (90) days from the date of delivery, perform substantially in accordance with QuiLogic's written materials accompanying it, and (b) any Support Services provided by QuiLogic, Inc. shall be substantially as described in applicable written materials provided to you by QuiLogic, Inc.

CUSTOMER REMEDIES. In the event of any breach of warranty or other duty owed by QuiLogic, Inc., QuiLogic’s and its suppliers’ entire liability and your exclusive remedy shall be, at QuiLogic’s option, either (a) return of the price paid by you for the SOFTWARE PRODUCT (not to exceed the suggested U.S. retail price) if any, (b) repair or replacement of the defective SOFTWARE PRODUCT or (c) re-performance of the Support Services. This Limited Warranty is void if failure of the SOFTWARE PRODUCT has resulted from accident, abuse, or misapplication. Any replacement SOFTWARE PRODUCT will be warranted for the remainder of the original warranty period or thirty (30) days, whichever is longer.

NO OTHER WARRANTIES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, QUILOGIC, INC. AND ITS SUPPLIERS DISCLAIM ALL OTHER WARRANTIES AND

CONDITIONS, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, WITH REGARD TO THE SOFTWARE PRODUCT AND THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES. THE LIMITED WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY HAVE OTHERS, WHICH VARY FROM STATE/JURISDICTION TO STATE/JURISDICTION. Some states and jurisdictions do not allow disclaimers of or limitations on the duration of an implied warranty, so the above limitation may not apply to you. To the extent implied warranties may not be entirely disclaimed but implied warranty limitations are allowed by applicable law, implied warranties on the SOFTWARE PRODUCT, if any, are limited to ninety (90) days.

9) LIMITATION OF LIABILITY. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL QUILOGIC, INC. OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE PRODUCT OR THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES, EVEN IF QUILOGIC, INC. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN ANY CASE, QUILOGIC's ENTIRE LIABILITY UNDER ANY PROVISION OF THIS EULA SHALL BE LIMITED TO THE AMOUNT YOU ACTUALLY PAID TO QUILOGIC, INC. FOR THE SOFTWARE PRODUCT OR SERVICE THAT DIRECTLY CAUSED THE DAMAGE. BECAUSE SOME STATES AND JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY, THE ABOVE LIMITATION MAY NOT APPLY TO YOU

Table of Contents

Introduction and Overview

QuiLogic has developed SQL / XML-IMDB, a high performance in-memory native xml database engine with SQL and XQuery interface, transaction and multi-threading support. SQL/XML IMDB simplifies application development and integration through declarative data management and by providing a seamless integration between SQL and XML data.

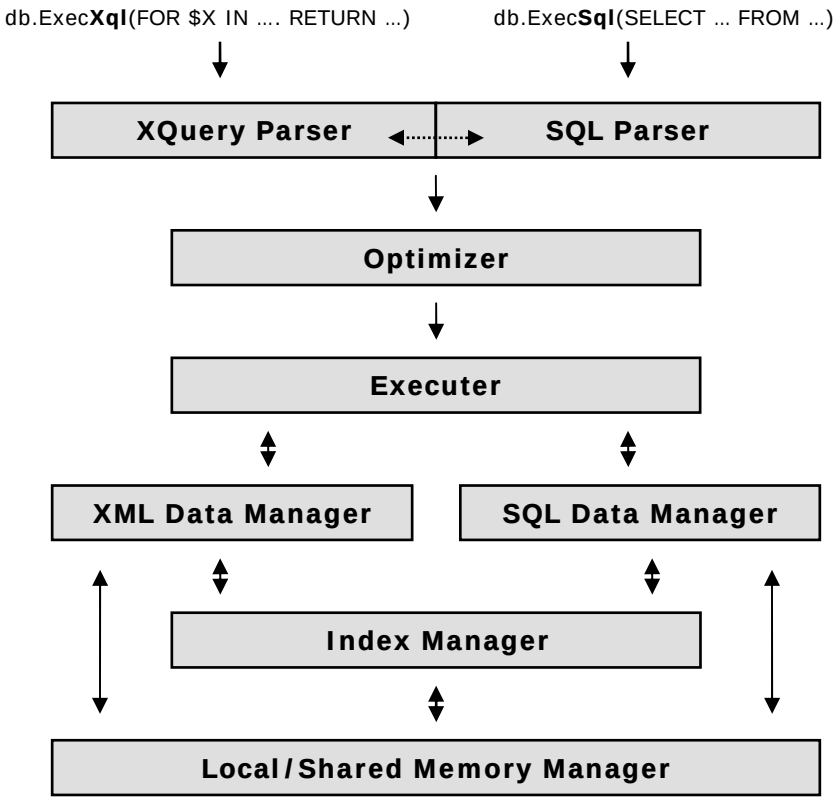
SQL/XML-IMDB is part of QuiLogic's information integration strategy to unify structured and unstructured data from sources such as relational databases, XML documents, flat files and Web services (SOAP). With SQL/XML-IMDB, users can access relational data as if it were XML data; exchange complex data structures between different applications; access real-time info and search across XML text documents and/or relational tables.

Features

- Combined relational and XML database engine with SQL and XQuery interface. Supports **.**NET, Visual Basic, VBA, C, C++, Delphi, Perl, and PHP application development environments. Available as NET Assembly, ActiveX, DLL and LIB component including ANSI, as well as UNICODE libraries.
- Supports process local-memory tables for high speed application-internal data management and **shared-memory** tables for high performance data sharing and exchange between different applications and development environments.
- Ability to create XML views over relational data and to create, manipulate and transfer data between XML and SQL tables.
- Build in XML Data-Binding facility for connecting XML files to in-memory objects and for easy processing of SOAP messages.
- Supports UPDATE, DELETE and INSERT operations on SQL and XML data.
- Capability to store and manage up to 2 billion XML nodes and SQL rows.
- Export /Import content and query results to/from XML string variables and disk files.
- Filter, manipulate and update SQL/XML data with an extreme easy to use API.
- Supports a significant subset of the SQL92 language and the current XQuery draft.
- Supports Prepared SQL statements and Transactions (Begin, Commit, Rollback).
- Includes a .NET Managed Provider for standard based data access.

Database Architecture

SQL/XML-IMDB stores XML documents and relational data in their native structure, meaning that they remain in their natural form, either as complex, hierarchical XML objects (not broken down into tables and columns), or as collection of rows for relational data. When they are stored or accessed there is **no** conversion of the structures, resulting in unsurpassed store and retrieval speeds.

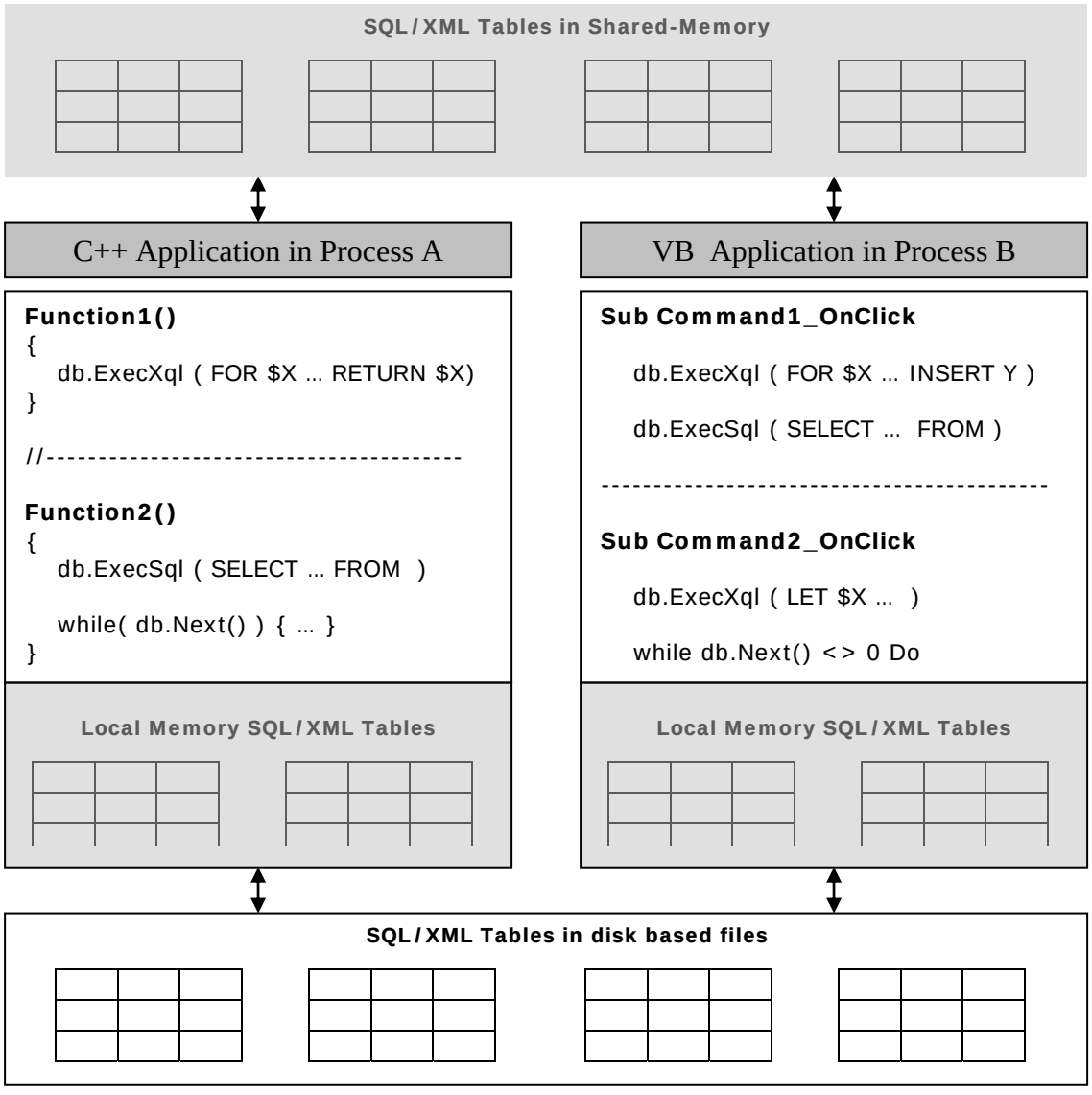


SQL/XML-IMDB's data storage includes compression technology to keep the memory requirement as low as possible. String data is always stored in variable length byte arrays and a bool data type requires only one bit. The optimizer dynamically re-optimizes plans during execution based on the actual size information of intermediate results. Queries can even be further accelerated by the use of prepared statements. Once the result set is ready, data values will be retrieved with speed comparable to linked list operations.

SQL/XML-IMDB uses a TST-tree as the main indexing algorithm. This algorithm combines the speed advantage of a hash table with the ordered access of a binary tree. Additional for XML data, the engine uses Reverse-Lookup indexing and a special Token-Segment-Build-Up indexing mechanism invented by QuiLogic for ultra fast loading and processing of XML files.

Principal Application Architecture

SQL/XML-IMDB provides the flexibility to work in different application environments and to share data in an easy and seamless way between different processes. It is a **data management framework** with an extreme easy to use API for sharing and management of complex data structures, internal and across applications. The database is multi-user and multi-threading ready and reduces the implementation effort for custom applications to an absolute minimum and helps to produce scalable and well designed software.



An XML Data-Binding Facility

It would be much easier to write XML enabled programs if we could simply map the components of an XML document to simple in memory objects that represent, in an obvious and useful way, the XML document's intended meaning according to its schema.

SQL/XML-IMDB supports this, through its cursor based data binding facility, which allows a developer to bind the returned XML query result to specific columns of a relational view. The application developer can then iterate over the set of bound objects, visiting the XML objects one by one. The cursor columns contain either simple typed values (int, float, string…) or xml trees of any complexity. Access the column values with Get/SetXXXValue() functions, walking the cursor set with one of the navigating functions Next(), Previous(), Last(), First().

SQL/XML-IMDB let you **create, insert, update, access, filter, transform** and **consume** sql and xml data either trough declarative statements or by using read only and read/write cursor based access from within your application code with simple to use API calls.

It is possible to create XML views over relational data and to manipulate and transfer the data between SQL and XML tables.

Titel	Author	Year
TCP/IP …	Walter …	1994
Programming…	…	…
…		

```
<book year="1994">
   <title>TCP/IP Illustrated</title>
   <author>
     <last>Stevens</last>
     <first>Walter</first></author>
   <publisher>Addison-Wesley</publisher>
   <price> 65.95</price>
</book>
  . . . .
```

SELECT … FROM … WHERE
INSERT/ UPDATE/ DELETE

FOR/LET .. WHERE .. RETURN
INSERT/ UPDATE/ DELETE

Cursor

→	TCP/IP Illustrated	Walter Stevens	1994	65.95
→	Programming for UNIX	Frank Novak	1992	57.30
	SQL Unleashed	Gordon Brovis	1990	32.00
	…			

Extreme Ease of Use

SQL and XQuery commands will be executed by calling two simple functions (`ExecSql`, `ExecXql`), providing a command string as argument. If the executed command returned result rows, a cursor is automatically opened and the data is accessed through cursor based API calls. By using either SQL or XQuery statements you declarative specify what to filter out from any XML structure or relational data table and access it by simple traversing the returned cursor row by row or node by node with almost no or little coding.

SQL	XQuery
db.**ExecSql**("SELECT ... FROM ... ")	db.**ExecXql**("FOR $X IN ... RETURN ...")

SQL and XQuery query results are retrieved through cursor based access:

```
while( db.Next() )
{
   ...
   ...
}
```

SQL/XML-IMDB provides a rich set of API functions:

- **Command execution** (ExecSql, ExecXql, Prepare…)
- **Data manipulation** (Update, Delete, Insert)
- **Cursor movement** (Next, Previous, Last, First, RowCount…)
- **Accessing data** (Get/SetIntVal, Get/SetCharVal, Get/SetBoolVal…)
- **Import/Export SQL+XML** (Export, ExportX, Import, ImportX, ExportToMemory, ImportFromMemory…)
- **Format handling** (SetOutputDateFormat, SetInputDateFormat, SetXmlFormat...)
- **Transaction support** (Commit, Rollback, Begin)

and more …

XML documents contain text and values. Sure, even the value content is in text form, but it can be interpreted as having a value of a certain type. For example, the string "07/12/1999" can be interpreted as being a sequence of characters (text) or being of type date having the value of a given date. To specify the data type of the return value use one of the data type modifiers added to the end of the bound variable separated by a "/".

- **…/text()** Text
- **…/number()** Integer
- **…/real()** Double
- **…/datetime()** Date/Time
- **…/date ()** Date
- **…/time()** Time
- **…/bool()** Bool
- **…/name()** Name of Element as text
- **…/node()** Unique id of xml element as integer
- **…/position()** Position of xml element as integer

If you omit the data type modifier, type text is assumed and the query returns both the element name **and** content of the parent and all contained child elements:

```
<author><first>Stanislav</first> ... <author/>
```

For example, to return the text content of an element enclosed between the tag-name, use the following example expression:

```
... RETURN { $X/author/name() + ' ' + $X/author/text() + ' ' + $X/author/name() }
```

which returns a result string like: "author Stanislav Lem author"

Example1 (XQuery)

List book titles published by Addison-Wesley after 1993.

XQuery:

```
for $b in document('bib.xml')/bib/book where $b/@year > 1993 return $b/title/text()
```

Code example in C++

```
-----------------------

CIMDb db;
CString bookTitle;

db.Open();
db.ExecXql(xquery);
while( db.Next() ) { bookTitle = db.GetCharVal(1) }
```

Returns the list of book titles in variable bookTitle:

TCP/IP Illustrated
Data on the Web
….
….

Example2 (XQuery)

Up to 64 cursor columns can be bound to different XML result nodes/tuples by simple using a " , " in the RETURN statement. An ORDER BY can be used to order the result set on different columns in the result cursor.

List book titles published by Addison-Wesley after 1993. Order the result by year descending.

XQuery:

```
for $b in document('bib.xml')/bib/book where $b/@year > 1993
RETURN  $b/title, $b/@year/number() ORDER BY 2 DESC

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

CIMDb db;
CString bookTitle;
int nYear;

db.Open();
```

```
db.ExecXql(xquery);
while( db.Next() ) { bookTitle = db.GetCharVal(1); nYear = db.GetIntVal(2); }
```

Returns the list of book titles in variable bookTitle:

<title>Data on the Web</title>
<title>TCP/IP Illustrated</title>
….

and the publishing year in variable nYear as of type number:

2000
1994
…
…

Example3 (SQL)

Accessing a relational table:

Code example in C++

```
CIMDb db;
CString resultA;
int x;

db.Open();
db.ExecSql("SELECT * FROM A, B WHERE a.A = b.B");

while( db.Next() )
{
    result = db.GetCharVal(1);
    x = db.GetIntVal(2);
}

db.Close();
```

Working with SQL and XQuery together

SQL/XML-IMDB let you seamless work with both, relational and XML data at the same time. The engine uses separated tables for SQL and XML data but you can freely mix the calls to the SQL as well as the XQuery part of the engine at any time and any location in your application. To do so, use one of the two functions below:

- `ExecSql( sql statement )`
- `ExecXql( xquery statement )`

The result of the query is accessed in both cases by iterating over the automatically opened cursor. The values can be retrieved by using one of the `GetXXXVal` functions.

For XQuery, depending on the formulation of your `RETURN` clause, the returned data is either the element content value of the specified type (string, integer, double…) or an entire sub-tree of the XML tree as of type string containing element names and value.

…RETURN $X/author	…RETURN $X/author/last/text()
<author><first>Walter ….</last></author>	Stevens
<author><first>Frank ….</last></author>	Novak
<author><first>Gordon ….</last></author>	Provis
…	…
…	…

Each bound variable accessed in the `RETURN` part of the XQuery statement creates a new column in the resulting cursor when the variable (or expression) is separated by a "," . This is very similar to the SQL `SELECT` statement with one or more projection columns.

```
SELECT last ,          First ,          Birthday FROM …

FOR/LET … RETURN
$X/last/text() ,    $X/first/text() ,   $X/birthday/date()
```

Stevens	Walter	1960-12-03
Novak	Frank	1965-02-14
…	…	…

To sort the result on a specific column, simply append an ORDER BY statement:

```
1. SELECT … FROM TR ORDER BY a, b
2. FOR/LET … RETURN $X/..., $Y/...  ORDER BY n1, n2
```

For XQuery, the column to sort is specified with position numbers starting from 1…

Mixing XQuery with SQL statements

In an attempt to ease the management and access of relational data in xml based environments, QuilLogic has developed an extension to the XQuery draft specification which allows the use of SQL statements within any part of the XQuery statements where an expression is allowed too.

- `FOR/LET … IN ( SQL SELECT query ) …`
- `… WHERE $X/… = ( SQL SELECT query )`
- `… WHERE $X/… IN ( SQL SELECT query )`
- `… WHERE $X/… = All/Any/Some ( SQL SELECT query )`
- `… RETURN { (SQL INSERT/UPDATE/DELETE statement ) }, …`

Columns in the `WHERE` clause section of the SQL query can be compared against any bound variable from the earlier part of the XQuery query, making it effectively possible to use correlated values between XQuery and the SQL sub-queries.

Example:

```
… FROM TR WHERE TR.price <> $Y/[path]/number() AND …
```

Please note that the other way around, using XQuery expressions within SQL statements (`ExecSql`) is not supported!

When using a SQL statement within the `RETURN` part of XQuery it is possible to manipulate the content of relational tables by using SQL Insert/Update/Delete statements. Applying a SQL `SELECT` statement has no effect (although it is possible).

Bridging Relational Technology and XML

One of the main features provided by SQL/XML-IMDB is the ability to create XML views of existing relational data. SQL/XML-IMDB does this by automatically mapping the data of the underlying relational database system to a low-level default XML view. Users can then create **application-specific XML views** on top of the default XML view. These application-specific views are created using XQuery. Moreover, users often need to synthesize and extract data from multiple relational and XML sources. SQL/XML-IMDB allows arbitrarily complex views and queries to be expressed, combining any number of sources, xml documents or relational data in one query.

Default XML view of relational data when using a SQL statement within XQuery:

```
<row>
   <col1>xxx</col1>
   <col2>xxx</col2>
   <col3>xxx</col3>
</row>
...
```

In summary, you can:

- **Compose XML Views over Relational Data**

```
FOR $X IN (SELECT a,b,c FROM A,B WHERE …) RETURN <> $X/… </>
```

- **Mix XML Data Sources with Relational Data Sources**

```
FOR $X IN (SELECT a,b,c FROM A,B WHERE …)
   FOR $Y IN DOCUMENT('abc.xml')
      WHERE …
      RETURN $X, $Y
```

- **Transfer Relational Data to XML Documents**

```
INSERT (SELECT * FROM … WHERE …) INTO TX   (TX as XML table)
```

- **Transfer XML Data to Relational Tables**

```
FOR $X IN DOCUMENT('abc.xml')
   WHERE …
      RETURN {(INSERT INTO TR VALUES($X/title/text(),
      $X/year/number()) }
```

- **Use Correlate Variables between XML and SQL Queries**

```
… WHERE $Y/[path]/number() = $X/colX/number() …
```

Working with SQL and XML Tables

SQL/XML-IMDB promotes a tight coupling of SQL and XML tables. Not only is the creation and loading of data into the memory tables extremely simple and similar for both data domains, as well as easy will be the exchange of information between both domains.

Creating tables

SQL	XML
db.ExecSql(“CREATE TABLE TR(...)”)	db.ExecXql(“CREATE TABLE TX”)

Use the keyword **SHARED** (*CREATE TABLE SHARED T...)* to create the tables in the shared memory space for sharing and exchanging simple and complex data ultra fast between different processes and application environments like C++, NET, Perl, VBA…

Loading tables from file data:

LOAD ‘abc.txt’ INTO TR	LOAD ‘abc.xml’ INTO TX

Loading tables from other tables:

INSERT INTO TR SELECT ... FROM ..	INSERT (SQL/XML SUBSELECT) INTO TX

Saving table data into files:

SAVE TR INTO ‘abc.txt’	SAVE TX INTO ‘abc.xml’

Relational tables store there data content in row/column format, XML tables as tree.

Deleting table data:

DELETE FROM TR WHERE ...	DELETE FROM TX

Destroying tables:

DROP TABLE TR	DROP TABLE TX

Based on our special indexing technology, the loading and processing of XML as well as relational data is ultra fast and the response times for queries in general are far below compared to traditional file based database systems.

SQL Data Update

To insert/edit relational data you may either use declarative DML statements like `INSERT… / UPDATE… / DELETE…` executed through calling the function `ExecSql(),` or use one of the update-cursor based API functions.

Example of DML:

```
db.ExecSql("UPDATE TR SET ab = 'Walter' WHERE isbn = '134-3447-838'")
```

For `SELECT` queries, the cursor is bi-directional but read only by default. If you need an update cursor, append the SQL command with a „**`FOR UPDATE`**“ clause. This opens the cursor in bi-directional read/write mode.

The data edit functions `Insert(), Update(), Delete()` affect the database row(s) at the current cursor position. The Insert function appends a new row on the table. Before inserting new data, provide the values for this new row with one of the `SetXXXVal` functions. The Delete function deletes the row at the current cursor position.

Example of API usage:

```
Db.ExecSql("SELECT first FROM TR FOR UPDATE")

While( db.Next() )
{
    db.SetCharVal("first", "xxxx")
    db.Update();
}
```

Both methods above for updating data may be enclosed by a transaction for safe rollback in case of update failure.

```
db.BeginTransaction()
```

Update data…
…
…

```
db.Commit()  or db.Rollback()
```

XML Data Update

W3C is considering letting XQuery go to recommendation status without UPDATE or DELETE semantics being a part of the recommendation. QuiLogic therefore has designed and implemented an XQuery extension based on the simple `INSERT-UPDATE-DELETE` semantics that is found by users of relational databases (SQL users). This extension allows the manipulation of XML data in a declarative and very easy to use style, comparable to that found in the SQL manipulation language. Use the DML expressions below in conjunction with the `ExecXql("DML expression")` function.

Deleting Data:

For $X in TX where …. DELETE $X/[path] or …. DELETE $X/@attribut

Rename Nodes:

For $X in TX where …. RENAME $X/[path] TO 'NewName'

Update Node Values:

For $X in TX where …. REPLACE $X/[path]/text() WITH value

Update entire Nodes:

For $X in TX where …. REPLACE $X/[path] WITH <> … <>
For $X in TX where …. REPLACE $X/[path] WITH $Y
For $X in TX where …. REPLACE $X/[path] WITH (SQL/XML Query)
. . . .

Update Attribute:

For $X/@attr in TX where …. REPLACE $X WITH ATTRIBUT('abc',value)

Insert new Data:

For $X in TX where …. INSERT <> … <> INTO [BEFORE] [AFTER] $X/[path]
For $X in TX where …. INSERT (SQL/XML Query) INTO [BEFORE] [AFTER] $X/[path]
. . . .

…where TX is an in-memory XML table.

Note: Transaction support is available only for SQL based data manipulation, for queries like `INSERT UPDATE DELETE` on relational tables. An (Update-) Cursor based xml data update is currently not supported but will be available in a future version.

Memory Management

Due to performance reasons, the database engine internally uses there own dynamic memory manager, bypassing the operating system memory manager. At database start up, the engine reserves a large block of memory with the operating system and sub-allocates required memory out of this reserved space. Both, shared-memory and process local-memory are reserved in this manner. The default size is 64 MB for local-memory and 16MB for shared-memory. These values can be changed to a custom value before opening the database, by using the functions **SetMaxLMemory** or **SetMaxSHMemory**.

Note: *At start-up, the memory is only reserved! No memory is consumed until the engine starts to allocate the memory for creating tables, cursors and managing internal structures.*

For local-memory, the engine extends (and shrinks) the range of reserved memory as needed. Therefore, the amount of reserved memory at start-up is not critical, because this value will change during operation time, as needed by the engine. But choose a value large enough to fulfill your predictable memory needs, because extending the reserved space consumes processor cycles, hence costs performance. A large initial value reduces the number of extents. Remember that the memory is only consumed until the engine actually starts to allocate it.

For shared memory things are different. The initial specified value reflects the **upper limit** to which the shared-memory can be sub-allocated. Due to technical reasons no extending (or shrinking) is possible. But again, the initial shared memory is only reserved. No shared-memory is consumed until the engine starts to actually allocate it.

Note: *The combined values (shared and local) of initial reserved memory space can not be greater than 2 GB (operating system limit). The actual size depends on your available virtual memory size. For best performance results, choose a value that reflects your installed physical memory size. The minimal value for the shared memory pool is 1 MB and 4 MB for the local pool.*

Shared memory data exchange

Although, several methods exist for inter-process communication (Pipes, files, tcp…) the data exchange through shared memory is ultra-fast and in theory the easiest one. But in practice this method is tricky and cumbersome to implement. One has to consider meaningless pointer values in different process spaces, synchronizing shared access to the common memory spaces, and so on. But these complexities are hidden from you. With the use of our database engine, data exchange between applications is as simple as executing INSERT and SELECT statements in the different application processes. Shared indexing is fully supported.

But you have to consider one important rule!

Do not exchange pointers between different application processes. Pointer values are meaningless in different processes, even when exchanged through the shared database tables.

If you exchange entire objects (in BLOB type columns) follow the same rule again. Any object member-attribute that represents a pointer reference will contain a meaningless value in the other process space.

Data values can be exchanged without problems. Even strings will be exchanged without problems, because they are copied internally and owned by the database.

Ownership of data

Any data, which you insert into the database tables, is owned by the database. For strings and binary data, a copy of the content is made and stored internally. When querying the database, the returned values for strings and binary data are pointers to the internal stored copies and therefore, must not be changed in any way. If you plan to change the content of the returned string or binary data, make a copy first. The content, referenced by the returned pointer, is read only.

In no case, make any changes to the data referenced by the returned pointer. This can corrupt the entire database. Make a copy first.

The above rule is true for C++ environments. For the ActiveX and NET component version of the database, the rule is different.

In **VB(A)** environments (using the ActiveX version of the database), for any returned value, the client is the owner. Even for strings and binary data the client holds the ownership. (Internally the database makes a copy of the string value before it hands out the data). For the ActiveX component, any returned data can immediately changed without making a copy first.

In **NET** returned strings and BLOB data is copied first (internal) and the copy is allocated from the managed heap. Therefore any returned data can immediately be edited by the caller.

Using shared and local tables together

Shared-memory and local-memory based tables can be freely intermixed in the `FROM` clause or `FOR/LET/WHERE` part of any SQL/XQuery statement. The engine manages the complexities behind the scene.

ANSI and UNICODE

SQL/XML-IMDB includes ANSI, as well as UNICODE libraries, to assist you in creating your international applications. Both versions have the same functional content, and differ only in the way character strings are stored.

The ANSI standard uses a single byte to represent each character. UNICODE is a character set where 2 bytes are used to represent each character.

Depending on your application type (ANSI or UNICODE) use the appropriated library. All IMDB UNICODE library names have a lowercase ('u') appended.

Although Visual Basic and NET internally is based on UNICODE, you can always use **either** the ANSI **or** the UNICODE version of the IMDB ActiveX library and NET Assembly in these development environments.

1. The ANSI **ActiveX and NET** version of SQL/XML-IMDB automatically converts any UNICODE string to the corresponding ANSI string and stores the result as ANSI characters. When queried, the strings are converted back to UNICODE on output. Therefore use the ANSI version to save memory space for stored string values in these environments.

2. The UNICODE ActiveX and NET version stores the string in UNICODE. No conversion is done on input and output.

3. The C/C++ DLL and LIB versions of the library store and output string values in the corresponding library format. Use the ANSI-DLL/LIB version for ANSI development and the UNICODE-DLL(u)/LIB(u) version for UNICODE development. **No conversion** is done on both, input and output.

4. Do not mix the ANSI and UNICODE versions of the library in data exchange scenarios between applications (shared memory tables), because the strings contain garbage when exchanged between applications with **different** character encoding schemas.

SQL/XML-IMDB contains a built in xml parser for loading xml files. This parser is flexible enough to work with xml files in various encoding schemas regardless of the used library. Both, the ANSI as well as the UNICODE libraries can read xml files in a number of character formats, 8 bit and 16 bit (UTF8, UTF16…) at the same time. After parsing the xml file the character data is stored in the corresponding library type format.

Catalog Functionality

Information about the created SQL and XML tables and SQL column layout of each table in the database can be found in two special **catalog tables**. These tables can be queried (SQL!) in the same manner as it will be done for user defined tables. The tables are created and managed by the system and therefore are read-only. Information about shared tables is accessible from all processes, information about local-memory tables is available only for the creating process.

QSYS_TABLE_INFO holds the information about user defined tables and column information is stored in QSYS_COLUMN_INFO.

Layout of QSYS_TABLE_INFO:

TID	TNAME	CCOUNT	XML	TSHARED
int	varchar	int	bool	bool

TID	Unique id of table
TNAME	Name of table
CCOUNT	Count of columns in table
XML	Is the table a XML or SQL table ? (XML = TRUE, SQL = FALSE)
TSHARED	Is the table in shared or local memory space ? (shared = TRUE, local = FALSE)

Layout of QSYS_COLUMN_INFO:

REFTID	CID	CNAME	CTYPE	CSIZE	CINDEX
int	int	varchar	int	int	int

REFTID	Reference to the table where the columns belong to
CID	Unique column id
CNAME	Name of column
CTYPE	Data type of column
CSIZE	Size of column if of character type, else 0
CINDEX	Type of index

The possible values for column CTYPE are:

SQL type	Value	Defined constant
bool, boolean, yesno	1	COLUMN_TYPE_BOOL
date	6	COLUMN_TYPE_DATE
time	7	COLUMN_TYPE_TIME
datetime	8	COLUMN_TYPE_DATETIME
int, integer, number, long, short, smallint	3	COLUMN_TYPE_INT
counter	2	COLUMN_TYPE_COUNTER
double, realno, currency, real, single, float	4	COLUMN_TYPE_DOUBLE
char, varchar, text, string, character	5	COLUMN_TYPE_CHAR
BLOB	9	COLUMN_TYPE_BLOB

The possible values for column CINDEX are:

COLUMN_NO_INDEX	0
COLUMN_UNIQUE_INDEX	1
COLUMN_MULTI_INDEX	2

Querying the catalog for table/column information is simple.

- List all tables in shared-memory which are XML tables:

```
Select * from QSYS_TABLE_INFO Where XML = TRUE and TSHARED = TRUE
```

Call ExecSql() with above command string and iterate through the result with db.Next()

- List all tables with a table name starting with T:

```
Select * FROM QSYS_TABLE_INFO Where TNAME like ‘T%’
```

- List all columns for table T1:

```
Select CNAME from QSYS_TABLE_INFO, QSYS_COLUMN_INFO
         Where TID = REFTID and TNAME = ‘T1’
```

Working with Date and Time

SQL/XML-IMDB uses a high resolution date/time format spanning a range from year 0 to 20.000 with a resolution down to 100 ns. There are three date/time column types available for handling date/time values.

- **Date** The date portion is significant, time is set to 00:00:00 by the engine.
- **Time** The time portion is significant, date is set to 1. Jan. 1601 by the engine.
- **DateTime** Both, the date and time portion is significant.

There are four main input formats for parsing strings containing date/time values:

Type	Format	Value
Date	**USA**	**12/01/2002**
Date	**Europe**	**01/12/2002**
Date	**ISO**	**2002/12/01**
DateTime	USA	12/01/2002 12:58:30.199.299.300)
DateTime	Europe	01/12/2002 12:58:30.199.299.300
DateTime	ISO	2002/12/01 12:58:30.199.299.300
Time	**USA**	**12:58:30.199.299.300 (hh:mm:ss.ms.us.ns**
Time	**Europe**	**12:58:30.199.299.300**
Time	**ISO**	**12:58:30.199.299.300**
DateTime	(ISO)Special	2002-12-01T12:58:30(Z) (Used in VISIO or Excel)

For all formats the second, millisecond, microsecond and nanosecond part is optional.

IMDB is quite flexible in parsing date/time values when reading the values as strings. To set the recognized input format for parsing date/time values use the API function:

```
SetInputDateFormat(format).
```

The function affects both, arguments for SQL queries as well as XML queries.

`SetInputDateFormat` can be used at any time in your application and sets the recognized format for the given IMDB object instance on which the function is called. The default format is USA. For importing (or querying) SQL/XML file-data, the recognized date format is set as well by that function.

Date/time strings are parsed in a very flexible way. For example, having different date/time strings in the following form (USA):

- Jan. 1. 2002 08:56:30.43.199
- 01/01/2002 8:56:30.043.199
- 1-1-2002 8:56

are recognized as the same date/time value.

When exporting date/time values from SQL/XML tables to ASCII files or XML files the output format can be set by one of the following API functions.

```
SetOutputDateFormat( format )
SetOutputDatetimeFormat( format )
SetOutputTimeFormat( format )
```

For detailed information on the available *output* format strings, please see the API-Reference.

Please Note!

When using XQuery or mixed queries (XQuery and SQL together) the above functions must be used to set the recognized date format in queries which use explicit type conversions or comparing:

```
FOR $X IN …..
     WHERE $X/birthday/date() = ‘1999-01-01’
RETURN
     { INSERT INTO T VALUES($X/birthday/date(), …) }
```

To recognize the above birthday correctly as ISO date, use `SetInputDateFormat` just before issuing the Query.

The same mechanism is required on output of (SQL/XML) date/time values into files or application string variables:

```
FOR $X IN ( SELECT birthday, … FROM TR … )
     FOR $Y IN Document() …
RETURN
     <A> $Y </A>
     <B> $X/birthday/text() </B>
```

The string format for the outputted date value ($X) is set by `SetOutputDateFormat`.

Summary of Date/Time Formats

DATE format is:

- MM/DD/YYYY (USA),
- DD/MM/YYYY (Europe),
- YYYY/MM/DD (ISO)

DATETIME format is:

- MM/DD/YYYY HH:MM.SS.MS.US.NS (USA)
- DD/MM/YYYY HH:MM.SS.MS.US.NS (Europe)
- YYYY/MM/DD HH:MM.SS.MS.US.NS (ISO)

TIME format is:

- HH:MM:SS.MS.US.NS

The second (SS), millisecond (MS), microsecond (US) and nanosecond (NS) part for time or date/time values is optional. The year part can have 2 or 4 digits, month and day 1 or 2 digits, with or without leading zeros.

A date separator can be:

. - /

Month names can be:

"January", February, "March", "April", "May", "June", "July", "August", "September", "October", "November", "December".

or

"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec".

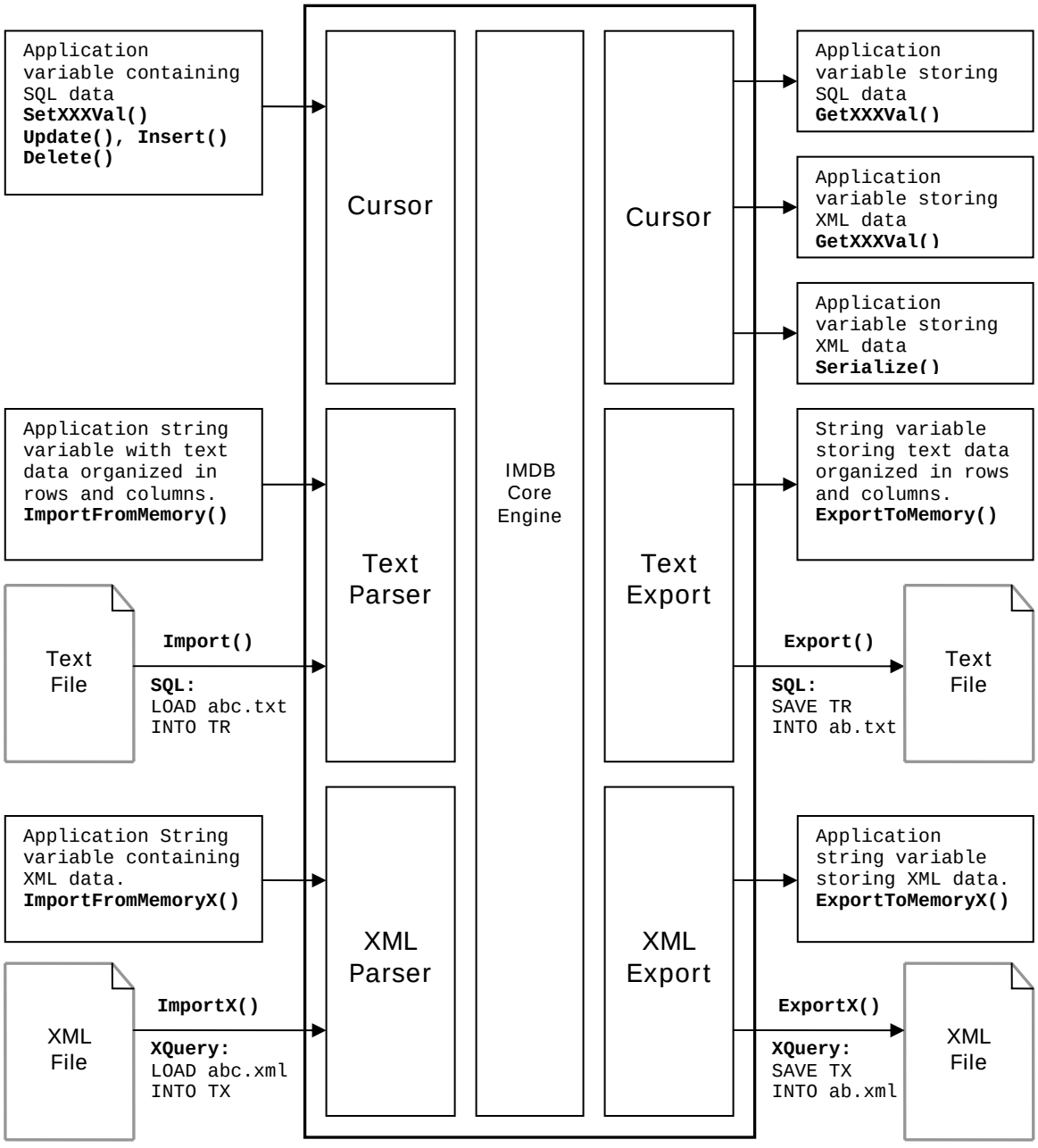
See example on next page.

Example in VB:

```
Private Sub Form_Load()

    Dim xDate As Date
    Dim db As Object
    Set db = New IMDBu

    db.Open
    db.ExecSql "CREATE TABLE A (a DATE, b TIME, c DATETIME )"

    db.ExecSql "INSERT INTO A VALUES('7-11-2000','09:00','7-11-2000 09:00')"
    db.ExecSql "INSERT INTO A VALUES('7.12.2000','10:00','7.12.2000 10:00')"
    db.ExecSql "INSERT INTO A VALUES('7/13/2000','11:00','7/13/2000 12:00')"
    db.ExecSql "INSERT INTO A VALUES('7/13/2000','11:00:59','7/13/2000 12:00:59')"
    db.ExecSql "INSERT INTO A VALUES('July.30.2000','11:00:59','July.30.2000 12:00:59')"
    db.ExecSql "INSERT INTO A VALUES('Aug.30.2000','11:00:59','Aug.30.2000 12:00:59')"

    db.ExecSql "SELECT * FROM A "

    While (db.Next)
        a = db.GetDateTimeVal(1)
        b = db.GetDateTimeVal(2)
        c = db.GetDateTimeVal(3)
    Wend

    'update the column values of the first row
    db.ExecSql "SELECT * FROM A FOR UPDATE"
    db.First

    xDate = DateValue("09/25/1973")

        db.SetDateVal 1, xDate
        db.SetTimeVal 2, xDate
        db.SetDateTimeVal 3, xDate

    db.Update

    'check values after update

    db.ExecSql "SELECT * FROM A "
    xy = db.RowCount
    db.First

    a = db.GetDateTimeVal(1)
    b = db.GetDateTimeVal(2)
    c = db.GetDateTimeVal(3)

End Sub
```

Data Import/Export and Persistence

Given the rich set of API functions as well as SQL/XQuery declarative statements, SQL/XML-IMDB provides an enormous flexibility in making the in-memory data content persistent. The graphic below shows the various paths to import/ export data. Import/Export formats are customizable by setting the corresponding parameters.



.NET Managed Provider

Overview

The SQL/XML-IMDB package includes a custom Data Provider for use with the .NET Data Access Framework (ADO.NET). That provider can be used together with the IMDb native API access functions. For highest performance demands we recommend to use the native API because it gives you the greatest flexibility and is really very simple to use. For cases where a more standard conforming access is needed (such as for modifying existing applications where a managed provider is already in use) the IMDb Managed Provider is a good alternative.

In your application you are always free to mix native IMDb API calls with calls to the Managed Provider Classes.

SQL/XML-IMDB provides a full implementation of the standard ADO.NET classes Connection, Transaction, Command, DataReader, and DataAdapter.

Interface	Description
IMDbConnection	Represents a unique session with the IMDB data source. Establishing a connection is lightweight because the DB is already in memory and initialized, and no other time consuming actions are performed when opening a new connection.
IMDbTransaction	Represents a transaction on the connection object.
IMDbCommand	Represents a query or command that is used when connected to a data source. You may use both SQL and XQuery commands through the command object.
IMDbDataParameter	Allows a user to provide a parameter to a command (SQL commands only) and its mapping to DataSet columns.
IMDbDataParameterCollection	Allows a user to provide a parameter to a command and its mapping to DataSet columns (SQL only).
IMDbDataReader	Provides a method of reading a forward-only read-only stream of data from your data source. Can be used for accessing SQL and XML datasets returned from the respective queries.
IMDbXmlDataReader	Provides a forward-only, read-only access to a stream of XML data returned from a XQuery command.
IMDbDataAdapter	Allows a user to populate a DataSet and resolving changes in the DataSet back to the data source.

Usage

The implementation of the .NET Managed Provider is full standard compliant with some enhancements made due to the possibility to execute SQL and XQuery commands.

You may issue SQL commands as well as XQuery commands through the same IMDbCommand interface. You need to set a special Command-Attribute immediate before firing the command, to tell the engine what command type to expect next.

```
Connection.SqlXmlCommandType = { QL_TYPE_SQL | QL_TYPE_XQuery }
```

`QL_TYPE_SQL` is the default value.

The returned result set is either a standard SQL like set of values (organized as rows and columns) or a stream of XML data accessible through the XmlDataReader object.

Note! You may even return SQL like result-sets by XQuery commands through our XML data binding facility. (For more information see chapter: *XML Data-Binding Facility*). Streams of XML data can be retrieved from SQL data sources too (see chapter: *Mixing XQuery with SQL statements*).

You may use at any time and any location in your program either the API functions or the managed provider classes. There is only one prerequisite where you need always at least one API call. **Before** using the managed provider classes

an initialization step is required !

To initialize and setup the database in your application there is at least one IMDb instance object required on which you must call the API function Open(). When you are done with your application you need to call Close(), which essential releases back all allocated resources (memory) back to the OS.
You typically place an initializing IMDb object in your applications main-object (the main-form for example) which must remain valid during the whole application runtime. This will ensure that your database is held open and valid until the application stops.

The namespace for the provider is: `QuiLogic.IMDB(u) (u is for UNICODE)`

The managed provider supports full **shared-memory table** access, which will make data exchange between different applications (processes) really a snap compared to the resource hungry marshalling mechanism used traditionally. The shared-memory data exchange through the managed provider classes will instantly resolve a lot of common performance problems in a simple and standard conforming way.

Example (C#)

```
// Note! QuiLogic.IMDBu stores all char data as UNICODE in the database
// QuiLogic.IMDB stores in ANSI format. Do not! use both together

using QuiLogic.IMDBu;

public class MyForm : System.Windows.Forms.Form
{
      private IMDb imDb;
      . . .

      public MyForm()
      {
            InitializeComponent();
            // Initializes the internal memory manager
            imDb.Open();
      }

      protected override void Dispose( bool disposing )
      {
            // Releases all memory resources back to OS
            imDb.Close();
            base.Dispose( disposing );
      }

      private void MyForm_Load(object sender, System.EventArgs e)
      {
            IMDb dbLocal = new IMDb; // a local defined IMDb object
            // create a SQL table (native API call)
            dbLocal.ExecSql("CREATE TABLE T(a TEXT, b DATE)");

            // Other native API calls, insert/update data ...
      }

      private void button_Click(object sender, System.EventArgs e)
      {
            string sql = "SELECT OrderID, customerID FROM Orders";
            IMDbConnection conn = new IMDbConnection();
            IMDbCommand command = new IMDbCommand(sql,conn);
            conn.Open();
            IDataReader myReader;
            myReader = command.ExecuteReader();

            while (myReader.Read()) {
                  Console.WriteLine(myReader.GetInt32(0) + ", " +
                  myReader.GetString(1));
            }

            // always call Close when done reading.
            myReader.Close();
      }
```

```
        private void button2_Click(object sender, System.EventArgs e)
        {
                // prepared statement
                string prepQuery = "INSERT INTO TR VALUES(?,?,?)";

                IMDbConnection myConnection = new IMDbConnection();

                IMDbCommand myCommand = new
                IMDbCommand(prepQuery,myConnection);

                myConnection.Open();

                myCommand.Prepare();

                // prepare the 3 parameters
                myCommand.Parameters.Add("1234");
                myCommand.Parameters.Add(DateTime.Now.ToUniversalTime());
                myCommand.Parameters.Add(999);

                // execute the prepared statement with above parameters
                myCommand.ExecuteNonQuery();

                // clear parameter list
                myCommand.Parameters.Clear();

                . . .
                . . .


                // always call Close when done with the connection.
                myConnection.Close();
        }
}
```

API Overview

To use the API functions, simple declare an Object of type **IMDB** local in your function. Declare as many objects, as you need in your local functions. These objects will have instant access to the common table area in the local-memory and shared-memory database file. Therefore, this common table area acts like a form of „global variables“. No special „reference forwarding" of IMDB objects between functions is necessary.

Example:

C++

```
Void MyFunction()
{
   CIMDb db1, db2;

   Use the db Objects
   ……
   ……
}
```

VBA

```
Sub MyFuntion
Dim db As Object

Set db = New IMDB(u) // u for the UNICODE version of the library !!!
or
Set db = CreateObject(QuiLogic.IMDB(u).1)

Use the db Objects
……
……
```

Initialization and Shutdown

Before using any of the API functions you must call `Open`. The `Open` function initializes internal data structures and the dynamic memory manager. This function is called once and **only once** in your application. You typically call the Open function in your applications initialization part. When you are done with your application, call `Close`. All allocated storage is released back to the operating system. For shared- memory operations you can provide a file name to the `Open` function. If you provide a name, the tables in the shared memory space will be made persistent to that file, after closing the

database. The shared-table content is immediately available the next time, when you open the database from the same file.

`Open( TCHAR *dbFileName = NULL )`
Opens the database and initializes the shared-memory and local-memory database files. Call this function before using any other API functions.

`Close()`
Closes the database and releases any allocated storage back to the operating system. Do not use any other API function after Close.

Executing SQL Commands

SQL commands are executed by calling `ExecSql` or `ExecPrepared` providing a valid SQL command string as argument. If the SQL command returns a result set, a cursor is automatically opened and the data can be accessed through cursor based API calls.

`ExecSql( TCHAR *sql)`
Executes the SQL command. If the command completed successfully a cursor is automatically opened for walking/retrieving the returned result set.

`Exist(const TCHAR *sql)`
Executes the given SQL command and returns TRUE if the found result set is not empty. An empty result returns FALSE. No cursor is opened. Use this function for simple „Exists“ queries.

`Prepare( TCHAR *sql)`
The give SQL command is parsed and optimized but not executed. The prepared SQL command can than be executed multiple times in subsequent calls with different parameters provide.

`ExecPrepared( TCHAR *values)`
Executes an already prepared SQL command.

`SetQueryTimeOut (int timeOutValue)`
Sets the time-out value for the above query executing functions. If a query accesses a table which is currently locked by another thread/process, the engine waits until the lock has gone, or the time out value has been reached, to complete the query.
On VB and .NET platforms the value is set through a property (Prop: QueryTimeOut).
The default value is 10.000 ms.

`IgnoreCaseOnLIKESearch (BOOL ignoreCase)`
Controls the behavior for a “LIKE” search .

Executing XQuery Commands

XQuery commands are executed by calling `ExecXql` providing a valid XQuery command string as argument. If the XQuery command returns a result set, a cursor is automatically opened and the data can be accessed through cursor based API calls.

`ExecXql( TCHAR *xql)`
Executes the given XQuery command.

`IgnoreCaseOnLIKESearch (BOOL ignoreCase)`
Controls the behavior for a “LIKE” search .

Cursor

When executing `SELECT` commands, a cursor is automatically opened. This cursor is closed when executing the next SQL command on the same object-instance or when the object goes out of scope and the destructor is called.

To explicit close the cursor use `CloseCursor`.

The cursor is bi-directional and read only by default. If you need an update cursor, append the SQL command with a „**`FOR UPDATE`**“ clause. This opens the cursor in bi-directional read/write mode.

A range of functions exists to retrieve the data from a specific column. Always use the correct typed functions because if the expected data type of the function and the content of the column do not match the function returns meaningless values.

`Value = GetXXXVal ( unsigned colNr )`
Retrieves the Value from the specified column number at the current cursor position. Column numbering goes from left to right and start from 1.

`Value = GetXXXVal ( TCHAR *columnName )`
Retrieves the Value from the specified column name at the current cursor position.

`Xml = Serialize( unsigned size,unsigned column,TCHAR *root)`
Returns the entire cursor result of an XQuery query as an XML tree.

A range of functions exists to insert or update the data at a specific column. These functions are used in combination with the `Insert` or `Update` cursor functions (see below). The `SetXXXVal` functions take the provided data value and store them in an internal structure. This storage structure is later read out by the `Insert`/`Update` functions and the contained values moved to the corresponding table columns.

`SetXXXVal ( unsigned colNr, Value )`
Set the new value to the specified column number.

`SetXXXVal ( TCHAR *columnName, Value )`
Set the new value to the column with the specified name.

… where XXX is replaced by Char, Int, Dbl … for the actual type.

Insert, Delete, Update Column Data

The data edit functions affect the database row(s) at the current cursor position. The Insert function appends a new row on the table. Before inserting new data, provide the values for this new row with one of the **`SetXXXVal`** functions. The Delete function deletes the row at the current cursor position.

`Update()`
Updates the row at the current cursor position

`Delete()`
Deletes the row at the current cursor position.

`rowNumber = Insert( TCHAR *tableName)`
Appends a new row on the specified table.
If the table contains a **`COUNTER`** **column** than the new column value is returned.

Cursor Navigation

When a new cursor is created (after executing the SQL command) the current cursor position is undefined. Before using any of the `SetXXXVal` or `GetXXXVal` functions position the cursor with one of the following functions.

First ()
Position's the cursor to the first row

Last ()
Position's the cursor to the last row

Next ()
Advances the cursor to the next row.

Previous ()
Advances the cursor to the previous row.

RowCount()
Returns the number of rows in the cursor row set.

CloseCursor()
To explicit close the cursor.

Miscellaneous Cursor based Functions

IsNull (unsigned colNr)
Returns TRUE if the value of the specified column is **NULL**.

IsNull (TCHAR* columnName)
Returns TRUE if the value of the specified column is **NULL**.

SetNull (unsigned colNr)
Sets the value of the specified column to **NULL**.

SetNull (TCHAR* columnName)
Sets the value of the specified column to **NULL**.

GetSqlColumnType (unsigned col)
Returns the data type of the specified column.

GetSqlColumnType(const TCHAR* columnName)
Returns the data type of the specified column.

GetColumnName(unsigned col)
Returns the name of the column with the given index.

GetColumnCount()
Returns the number of columns in the returned result set .

`GetColumnIndex(const TCHAR* columnName)`
Returns column index number of the named column.

Date/Time Format Control

Several functions are available to control the parsing and output formatting of date/time values.

`SetOutputDateFormat (TCHAR *format)`
`SetOutputDateTimeFormat (TCHAR *format)`
`SetOutputTimeFormat (TCHAR *format)`
Controls the string formatting for outputted date/time values

`SetInputDateFormat (TCHAR *format)`
Sets the input format for parsing date/time values.(US, ISO, …)

Import/Export from/to Text Files (SQL)

Relational table data can be imported/exported from/to ASCII/UNICODE files

Import (TCHAR *tableName, TCHAR *importFile, TCHAR delimiter = 0, TCHAR endLine = 0)
Import the data from the „importFile“ into the (relational) table with the specified name.

Export (TCHAR *tableName, TCHAR *exportFile, BOOL appendMode, BOOL writeColumnNames TCHAR delimiter = 0, TCHAR endLine = 0)
Export the data from the (relational) table with the specified name into the file „exportFile“.

Import/Export from/to Text Files (XML)

XML data can be imported/exported from/to ASCII/UNICODE files

ImportX (TCHAR *tableName, TCHAR *importFile)
Import the XML data from the „importFile“ into the table with the specified name.

ExportX (TCHAR *tableName, TCHAR *exportFile, BOOL appendMode)
Export the data from the XML table with the specified name into the file „exportFile“.

Import/Export from/to memory streams (SQL)

Relational table data can be imported/exported from/to memory streams (Character - arrays)

ImportFromMemory (TCHAR *tableName, TCHAR *pData, TCHAR delimiter = 0, TCHAR endLine = 0)
Import the data from a memory area into the (relational) table with the specified name.

TCHAR * ExportToMemory (TCHAR *tableName, TCHAR delimiter = 0, TCHAR endLine = 0)
Export the data from the (relational) table with the specified name into a memory area.

Import/Export from/to memory streams (XML)

XML data can be imported/exported from/to memory streams

`ImportFromMemoryX (TCHAR *tableName, TCHAR *pData)`
Import the XML data from a memory location into the table with the specified name.

`ExportToMemoryX (TCHAR *tableName, TCHAR *exportFile)`
Export the data from the XML table with the specified name into memory area.

Controlling XML Output Format

`SetXmlFormat (TCHAR *format)`
Sets the string formatting for the XML output. (see Ref. Manual for details)

Memory Control and Reporting Functions

Because the entire data set is in memory, it is important to have some control and knowledge about the internal memory management engine. (For a more detailed discussion of memory issues, see the chapter: **Memory Management**).

`GetLMemoryUsed ()`
Returns the size (in bytes) of currently allocated memory in the local-memory space.

`GetSHMemoryUsed ()`
Returns the size (in bytes) of currently allocated memory in the shared-memory space.

`SetMaxLMemory (unsigned long maxMemory)`
Sets the initial size of process local-memory space to be reserved by the dynamic memory manager (see: Memory Management). Must be called before `Open().`

`SetMaxSHMemory ( unsigned long maxMemory)`
Sets the maximal size of shared-memory space to be reserved by the dynamic memory manager (see: Memory Management). Must be called before `Open().`

Error Management

Most of the API functions return an error code reporting about the success or failure of the function. If the function succeeds the returned value is zero, non zero-values signal an error.

Most of the returned error codes refer the parsing and execution of SQL statements. Therefore, **always** check the returned error status of the SQL executing API functions.

Every instance of an IMDb object contains an internal variable, holding the latest error code. The next occurring error overwrites this with a new value. Be sure to query the error code always immediately after the error causing function.

The **VBA** (ActiveX) version of the database throws IMDB exceptions.

The **NET** version throws IMDB exceptions (IMDbException).

Functions with possible table lock conflicts throwing a “table read/write lock” exception:

ExecSql
ExecPrepared
Update
Delete
Insert*

*Note: The Insert function returns 1 (true) or the value of a COUNTER type column (if defined) after the call was successful. Either 0 (c++) is returned or an exception is thrown (VBA, NET) when the function failed.

A detailed description of the error can be retrieved, by calling one of the following functions below immediately after the error causing function.

`GetLastErrorNumber ()`
Returns the error number of the latest occurring error.

`TCHAR *GetErrorDescription (int errCode)`
Returns the pointer to a string, describing the error. (Not available for VBA and NET)

In shared memory scenarios it might be possible that another process my put a lock on a table which is never released due to a sudden process dead. To avoid a dead-lock on that (shared) table you can use these function to resolve the conflict.

`TCHAR * ClearAllLocksOnTable (TCHAR *tableName)`
The function clears any read and write lock on local or shared memory tables.

Transactions

The database supports one active transaction per IMDB instance object. But you can have multiple IMDB instances active at the same time.

`BeginTransaction ()`
Starts a transaction.

`InTransaction ()`
Returns the current transaction status.

`Commit ()`
Makes the change to the database permanent

`Rollback ()`
Discharges any changes made to the database

Remark

The database supports ACID level 3 only. Degree 3 isolation is true isolation, no lost updates, no dirty reads no unrepeatable read. The lock protocol is two-phase. All transaction operations (commit, rollback) are optimized for maximal execution speed. Simple flags where mostly used for implementing the transaction engine. We have done a lot of practical experiments and found that the implemented protocol with level 3 isolation outperformed in a lot of real world cases the lower level isolations, (which in theory should be faster). So we settled on level 3 only.

Remember: Only one transaction can be active at one time for a given object instance. But of course, you can have multiple instances active at a time.

SQL Language

SQL/XML-IMDB supports a significant subset of the SQL92 definition of the SQL database language, including support of the following:

SELECT, UPDATE, INSERT, DELETE
CREATE TABLE...[PRIMARY KEY...]
DROP [TABLE]
SELECT qualifiers: DISTINCT , TOP n
SELECT clauses FROM, WHERE, GROUP BY, HAVING, ORDER BY
WHERE expressions: AND, OR, NOT, LIKE, BETWEEN, + - * /, IS [NOT] NULL, <, >, =, < >, <=, >=, Constants, Parameters, ColumnNames
SELECT list expressions: MAX, MIN, AVG, SUM, COUNT, +, -, *, /, Constants, Parameters, ColumnNames
Value list qualifiers: ANY, ALL, SOME, IN
UPDATE expressions: +, -, *, /, Constants, Parameters, ColumnNames
INSERT values expressions: Constants, Parameters
INSERT ... SELECT
Subqueries within SELECT statements

Remarks

- Column and table names can not contain spaces or punctuation characters and must begin with an alpha character.
- Views are not supported.
- Select list field aliases names are not supported.
- Security statements, such as COMMIT, GRANT, and LOCK are not supported.
- No build in user identification
- The LIMIT TO *nn* ROWS clause used to limit the number of rows returned by a query is not supported. Use TOP instead to return the top nn rows of a query.
- Outer Joins are not supported.
- No indexes create and drop statements. Creation of indexes must be done at table create time.

XQuery Support

The language is currently being developed by the W3C XML Query Working Group and has working draft status (as of Dec. 2002, see References for details). Even though the current language definition is quite huge based on functional principles and contains at least 7 types of expressions, there is a simple to understand core principle behind all the complexities. It is possible to write really simple constructs which, as you will see, satisfies all your needs for querying and manipulating xml data.

The core of the language is based on the FLWR (pronounced "flower") expression, and is very similar to the **SELECT-FROM-WHERE** construction in SQL.

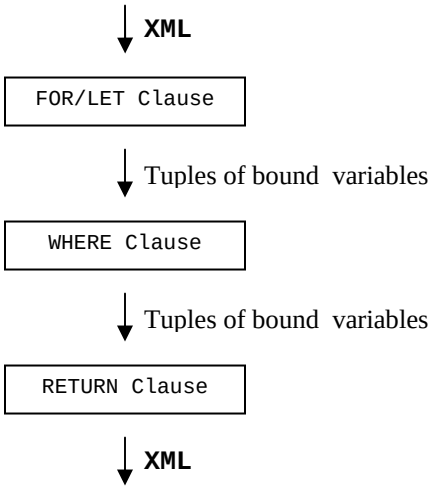
1.) A FLWR expression consists of:

- **FOR**-clause: binds one or more variables ($X…) to a sequence of nodes returned by another expression (usually a path expression, see below) and iterates over the nodes. The variable therefore represents an array of bound nodes.

- **LET**-clause: also binds one or more nodes but without iterating. A single sequence of nodes is therefore bound to the variable.

- **WHERE**-clause: contains one or more predicates that filters or limits the set of nodes as generated by the FOR/LET-clauses.

- **RETURN**-clause: generates the output of the FLWR expression. The RETURN-clause usually contains the references to variables and is executed once for each bound node-reference that was returned by the FOR/LET/WHERE-clauses.



The input to the expression consists of one or more XML **documents**, XML memory **tables** or SQL / XQuery **sub-queries**. The result of the FOR and LET clauses is an **ordered** list of tuples, each containing a value for each of the bound variables. The value of a variable bound by a FOR clause is an array of nodes and its descendants. The value of a variable bound by a LET clause is a (possibly empty) single set of nodes.

The `RETURN` clause is executed for each surviving tuple, generating output nodes from the values of the bound variables. The nodes generated by the `RETURN` clause can be accessed by walking the **cursor** or linearized into an output XML string with **Serialize**().

The `FOR` and `LET` clauses work together to generate tuples of variable bindings. Unlike a `FOR` clause, however, a `LET` clause does not affect the number of tuples that are generated. Each `LET` clause binds its variable to exactly one.

- If a query contains a LET clause but no FOR clause, exactly one tuple of variable bindings is generated.
- If there are more than on FOR clauses a Cartesian product of all tuples is formed.
- The `WHERE` clause serves as a filter that discards some of the tuples and retains others.

The result of the FOR/LET clause can be thought of as being equivalent to the rows and columns of a relational table where each column represents a bound variable.

The data model that XQuery uses is based on that of XPath (see References) and defines each XML document as a tree of nodes. Therefore XPath is heavily used in XQuery to select sub trees out of a larger xml tree just as it is used as the path selection language for XSLT. XQuery uses abbreviated XPath expressions.

2.) Path expressions

The second important construct are path expressions. The syntax is similar to the abbreviated syntax of XPath, the XML standard for specifying "paths" in an XML document. For example:

Find all titles of chapters in document books.xml:

```
document("books.xml")//chapter/title
```

Find all books in document bib.xml published by Addison-Wesley after 1991:

```
document(bib.xml")//book[publisher = "WROX" AND @year > "1991"]
```

In general, an XPath expression evaluates to a set of nodes. The FOR clause generates an ordered list of tuples, each containing of a value for each of the bound variables. A tuple is generated for each possible way of binding the list of variables to nodes that **satisfy their respective XPath expressions**. When a node is bound to a variable, its descendant nodes are carried along with it.

XPath path expressions may contain wildcards:

```
document("books.xml")/books/*/title
document("books.xml")/books/*@isbn
```

3.) Element Constructor

This type of expression is used when a query needs to create new elements, which is typically found in the return part of queries.

```
… RETURN
 <book year={$X/date()}
      <title> { $Y/title/text() }</title>
      <author> {$Z/last/text() + ‘ ’ +
      $Z/first/text() } </author>
 </book>
```

During query runtime the return clause iterates over the list of bound variables from the earlier part of the query, creating as many new <book> elements as referenced nodes found in the select/restrict part of the query .

The following example returns the title of all books published by Addison-Wesley:

```
FOR $X IN DISTINCT(document("bib.xml")/book/title)
FOR $Y IN document("bib.xml")/book[title = $X]
    WHERE $Y/publisher = ‘Addison-Wesley’
RETURN $X/text()
```

Although the XQuery draft specifies more constructs (conditional expressions, function expressions…) SQL/XML-IMDB queries are currently restricted to FLWR, Path and Element Constructor expressions, with some SQL stylish **enhancements** (IN clause, LIKE, SQL sub-selects…) to aid in query formulation as described in more detail below.

XML Input

There are 5 different methods to formulate the input expressions for the FOR/LET clause.

1. For/Let $X in [Distinct] **DOCUMENT**(‘abc.xml’)/[xpath] …
2. For/Let $X in [Distinct] **TX**/[xpath] …
3. For/Let $Y in [Distinct] **$X**/[xpath] …
4. For/Let $X in (**XQuery Query**) …
5. For/Let $X in (**SQL Query**) …

The contents of the input xml elements for the examples 1, 2 and 3 may be further restricted by the optional XPath expression. The DISTINCT keyword can be applied independently to each expression in a FOR/LET clause, serving to eliminate duplicate values from the node-sets returned by the input expression (see DISTINCT below).

1. DOCUMENT

The Document function returns the root node of a file based xml document. Any XPath expression following DOCUMENT selects one ore more child elements below the root. The file name argument string can contain path specifications and wildcard characters and must be enclosed in single quotation marks. Example: ‘C:\\docs\\xml*.xml’

Note!
The db engine creates a temporary in memory structure for keeping the loaded xml tree (with space optimizations applied) as long as the result cursor is active. After closing the cursor the in memory structure is automatically deleted by the engine.

2. TX

The xml data is taken from an already created and filled in-memory xml table TX (see Loading XML Tables below). Any XPath expression following TX selects one ore more child elements below the root. Note that an xml table may contain multiple root notes in contrast to file based xml data which usually can (and should) contain only a single root node. XML tables can be named arbitrary.

3. $Y

This example illustrates how contend of a variable can be taken from other variable defined in the earlier part of the query. The variable $Y, defined in the example query, ranges over the set of nodes bound by the variable $X earlier. XPath is used to select specific child nodes out of the bound nodes from $X.

4. XQuery Query

Example 4 illustrates how contend of a variable can be taken from the result of an other XQuery query. The variable $X, defined in the example query, ranges over the set of nodes returned by the expression (**XQuery Query**). Enclose the expression in "( )".

5. SQL Query

The xml data is taken from an SQL query by automatically mapping the returned data of the underlying SELECT projection to a low-level default XML view. The Default XML view of relational data when using a SQL statement within XQuery has the following structure:

```
SELECT COUNT(*), colB, colC FROM T …

<row>
     <col1>xxx</col1>
     <colB>xxx</colB>
     <colC>xxx</colC>
</row>
…
```

The SQL query can be any full blown query, including Aggregate functions like Count, Avg, Sum…

Columns in the WHERE clause section of the SQL query can be compared against any bound variable from the earlier part of the XQuery query, making it effectively possible to use correlated values between XQuery and the SQL sub-queries.

Example:

```
... (SELECT … FROM TR WHERE TR.price = $Y/[path]/number() AND …) ...
```

Variables bound to the returned content of SQL queries can be used and accessed like any other XQuery bound variable.

Example:

```
For $X in (SELECT colA FROM T) WHERE $X/colA = 'abc' RETURN …
```

The name for a column from the SQL query is simple taken from the underlying table. For columns created by aggregate functions like "COUNT" a default name is created consisting of the word fragment "COL" followed by an incrementing number: COL1, COL2…

WHERE clause

In the WHERE clause, predicates may be combined using parentheses, AND, OR, and NOT. Predicates are based on **XPath expressions** that contain the variables bound in the FOR and LET clauses. Comparing against values returned by sub-queries is possible too.

Examples:

```
WHERE $X/last/text() = ‘abc’ AND $X/price/number() = 99

WHERE $X/last[3]/text() = ‘abc’ AND $X/price/number() = 99

WHERE $X/book[@isbn = ’12-333-456’]/price/number() = 99

WHERE $X/pubdate/date() = ‘1994-12-03’

WHERE $X/price/number() = ( For … Return $X/price/number() )
```

Joins are possible too:

```
WHERE $X/last/text() = $Y/last/text()
```

The semantics of comparisons is the same as in XPath. For example, consider the comparison $X/last = "abc". In general, an XPath expression such as $X/last evaluates to a **set** of nodes. The comparison therefore is considered to be True if at least **one** of the nodes returned by $X/last has a string-value equal to "abc".

To specify the type of the bound variable in the comparison, use one of the data type modifiers added to the end of the bound variable separated by a “/”.

- **text()** String
- **number()** Integer
- **real()** Double
- **datetime()** Date/Time
- **date()** Date
- **time()** Time
- **bool()** Bool (TRUE/FALSE)
- **name()** Name of Element as string
- **node()** Unique id of xml node as integer
- **position()** Position of xml element as integer

The type modifiers shown above may be used for XPath expressions; /abc[expression/xxx()]/xxx() and in the RETURN clause too.

Modifier `text()` can be omitted in the WHERE clause as shown below.

```
WHERE $X/last/text() = 'abc'
WHERE $X/last = 'abc'
```

Remember that variables bound in a `FOR` clause are bound to individual nodes (with their descendants), but variables bound in a `LET` clause are bound to ordered sets of nodes (with their descendants). In the `WHERE` clause, appropriate predicates must be used with each type of variable. For example, in the following query, `$book` is bound to a **set** of books (by using `LET`), and the `WHERE` clause appropriately applies a `count()` function to count the number of books in the **set**. The query returns publishers who have published more than 100 books.

```
FOR $pub IN DISTINCT TX//publisher
LET $book := TX//book[pubinfo/publisher = $pub]
WHERE count($book) > 100
RETURN $pub/text()
```

If we require to add an additional condition on books, such as "find publishers who published more than 100 books in 2002", this condition could **not** be added to the `WHERE` clause, since the `WHERE` clause has access only to **sets** of books, not to individual books. The proper place to add such a condition would be in the XPath expression that defines $book, as follows:

```
FOR $pub IN DISTINCT TX//publisher
LET $book := TX//book[pubinfo/publisher=$pub AND
pubinfo/year="2002"]
WHERE count($book) > 100
RETURN $pub/text()
```

! Note that this will change in a future release of SQL/XML-IMDB because the newer XQuery drafts made this restriction obsolete now. Access to nodes bound by LET will be allowed then.

The `WHERE` clause may also use several operators taken from SQL. These operators will be illustrated below:

! Note that this is an extension implemented by QuiLogic and not part of the XQuery draft.

- `[NOT] LIKE`
- `[NOT] BETWEEN`
- `[NOT] IN`
- `SQL Sub-Query`
- `ALL, ANY, SOME, EXISTS`

Above operators (except for Subquery, All … Exists) may also be used in XPath expressions like: `$X/book[@isbn IN ('554-0772-03','776-1299-01')]/title`

Examples of SQL operators:

```
WHERE $X/last/text() IN (‘abc’, ‘def’, ‘xyz’)

WHERE $X/book[@isbn = ’12-333-456’]/price/number() IN (SQL query)

WHERE $X/book[@isbn = ’12-333-456’]/price/number() IN (XQuery)

WHERE $X/last[3]/text() LIKE ‘abc%’

WHERE $X/pubdate/date() BETWEEN ‘1994-12-03’ AND ‘2002-01-01’

WHERE $X/last/number() = [ANY,ALL,SOME] (SQL or XQuery)

WHERE EXISTS (SQL or XQuery)
```

DISTINCT

Distinct serves the same purpose as found in SQL:

```
FOR $X IN DISTINCT(document(‘bib.xml’)/book/title)…
```

The `DISTINCT` keyword can be applied independently to each expression in a `FOR/LET, WHERE and RETURN` clause, serving to eliminate duplicate values from the node-sets returned by the expression. Equality is defined by equality of value rather than by identity.

When `DISTINCT` is specified and several candidate nodes of equal value are available for binding, SQL/XML-IMDB does not specify which of the candidate nodes is bound to the variable.

Xml Elements having a content value of `NULL` are ignored by `DISTINCT` with the exception when adding the data type specify `/name()` to the variable.

Counts only distinct title elements having a title

```
Count(Distinct(document(‘bib.xml’)/book/title))
```

Counts ALL distinct child elements below book regardless of having a value or not (null)

```
Count(Distinct(document(‘bib.xml’)/book/*/name()))
```

Counts distinct numeric values of child elements (excluding null values)

```
Count(Distinct(document(‘bib.xml’)/book/*/number()))
```

Aggregate Functions

A LET clause is often used to bind a variable to a set of values that is used as the argument of some aggregate function such as avg(). For example, the following query returns the average price of all the books in the table TX:

```
LET $b :=  TX//book/price
RETURN
   <avgprice> {avg($b)} </avgprice>
```

Aggregate functions can be applied in `LET, WHERE and RETURN` clauses. For example the above query could be rewritten as:

```
LET $b :=  avg(TX//book/price/real())
RETURN
   <avgprice> $b </avgprice>
```

Use of an aggregate function in the WHERE clause:

```
FOR $pub IN DISTINCT TX//publisher
LET $b :=  $pub//book/price
WHERE avg($b) < 100
RETURN
   <publisher> $pup/text() </publisher>
   <avgprice> {avg($b)} </avgprice>
```

Available aggregate functions are:

- `COUNT`
- `SUM`
- `AVG`
- `MAX`
- `MIN`

Aggregate functions may be combined with DISTINCT.

RETURN Clause

The RETURN clause generates the output of the FLWR-expression, which may be a node, an "ordered forest" of nodes, or a primitive value. The RETURN clause is invoked once for each tuple of variable bindings that was generated by the FOR and LET clauses.

The RETURN clause may contain arithmetic expressions, structured XML text, bound variables with XPath expressions associated, and SQL and XQuery sub-queries.

The result of the RETURN clause is accessed either by iterating over the automatically opened cursor (using one of the GetXXXVal functions) or by calling Serialize() which outputs the result as an linearized XML document string.

Depending on the formulation of your RETURN clause, the returned data is either the element content value of the specified type (string, integer, double…) or an entire sub-tree of the XML tree as of type string containing element names and value.

…RETURN $X/author	…RETURN $X/author/last/text()
<author><first>Walter ….</last></author>	Stevens
<author><first>Frank ….</last></author>	Novak
<author><first>Gordon ….</last></author>	Provis
…	…
…	…

Each bound variable accessed in the RETURN part of the XQuery statement creates a new column in the resulting cursor when the variable (or expression) is separated by a "," . This is very similar to the SQL SELECT statement with one or more projection columns.

```
SELECT last ,          First ,          Birthday FROM …

FOR/LET … RETURN
$X/last/text() ,   $X/first/text() ,   $X/birthday/date()

     Stevens             Walter             1960-12-03
      Novak              Frank              1965-02-14
        …                  …                    …
```

To sort the result on a specific cursor column, simply append an ORDER BY statement:

```
3. FOR/LET … RETURN $X/..., $Y/...  ORDER BY n1, n2 [ASC, DESC]
```

For XQuery, the column to sort is specified with position numbers starting from 1…

Examples

Returning element content:

```
RETURN $X/last/text(), $X/price/number()

RETURN $X/book[@isbn = '12-333-456']/price/number()

RETURN $X/last/text(), $X/price/number() ORDER BY 1,2 ASC

RETURN { $X/last/text() + ' ' + $X/first/text() }

RETURN { $X/price/real() + 12.99 }, { SUM($X/price/real() }

RETURN { COUNT($X/book) }

RETURN $X, $Y, $Z, $X/price/number() ORDER BY 4 DESC
```

Remember that expressions must be enclosed with "{ }" whereas for simple variable names (even if XPath expressions attached) this is not required.

Construction of new elements:

```
RETURN <book> $X/title/text() </book>

RETURN <book isbn={$Y/isbn/text()}> $X/title/text() </book>

RETURN <book>
            <name> { $X/first/text() + $X/last/text() } </name>
            <price> { $X/price/number() + 99 } </price>
            <sold> { SUM($X/sold/number() } </sold>
       <book>
       <publisher> $X </publisher>
       ...
```

Subselects:

```
FOR $a IN DISTINCT TX1//author
RETURN
  <BooksByAuthor>
     <Author>
         $a/lastname/text()
     </Author>
     {
        (
           FOR $b IN documentTX2//book[author=$a]
           RETURN $b/title ORDER BY 1
        )
     }
  </BooksByAuthor> ,
  $a/lastname/text() ORDER BY 2
```

Subselects require to be enclosed in "{ (subselect) }"

SQL INSERT / UPDATE / DELETE statements within RETURN:

When using a SQL statement within the `RETURN` part of XQuery it is possible to manipulate and update the content of relational tables by using SQL Insert/Update/Delete statements. Applying a SQL `SELECT` statement has no effect (although it is possible).

! Note that this is an extension implemented by QuiLogic and not part of the XQuery draft.

```
RETURN { (INSERT INTO TR VALUES($X/last/text(),
$X/price/number()) ) }

RETURN { (UPDATE TR SET last = $X/last/text() WHERE price =
$X/price/number() ) }
```

Columns in the `WHERE` clause section of the SQL query can be compared against any bound variable from the earlier part of the XQuery query, making it effectively possible to use correlated values between XQuery and the SQL DML sub-query.

You may execute more than one DML statements in the RETURN clause.

```
RETURN { (SQL1) }, { (SQL2) }, …
```

Managing XML Tables

You may create any number of XML tables and populate it with the required data. XML table creation is as simple as creating an SQL table.

```
1. ExecXql( "CREATE TABLE [SHARED] MyXmlTable")
```

The difference compared to creating SQL tables is:

- Using `ExecXql()` instead of `ExecSql()`
- You do not require specifying any columns.

To populate the table with data you can either use one of the API based import functions or use a declarative statement executed by ExecXql().

```
2. ExecXql( "LOAD 'abc.xml' INTO MyXmlTable")
```

The file name argument string can contain path specifications and wildcard characters, and must be enclosed in single quotation marks. Example: `'C:\\docs\\xml\\*.xml'`

Note that loading several xml files into the same xml table effectively results in stored data having multiple root elements.

To copy data from table to table (SQL **and** XML) use:

```
3. ExecXql( "INSERT (SQL or XQuery query) INTO MyXmlTable")
```

When copying data from SQL tables a default view of the underlying relational data is created as described above (chapter: XML Input). Remember, to transfer xml data into relational tables use XQuery with SQL Insert/Update statements in the `RETURN` clause.

To save the content of an xml table to file you can either use one of the API based export functions or use a declarative statement executed by ExecXql().

```
4. ExecXql( "SAVE MyXmlTable INTO 'abc.xml'")
```

To delete the xml content use:

```
5. ExecXql( "DELETE FROM MyXmlTable")
```

Contrary to the corresponding SQL `DELETE` statement you can **not** apply a `WHERE` clause. (To delete specific nodes use the XQuery Delete expression as described below)

To remove the entire table use:

```
6. ExecXql( "DROP TABLE MyXmlTable")
```

XML Data Update

W3C is considering letting XQuery go to recommendation status without UPDATE or DELETE semantics being a part of the recommendation. QuiLogic therefore has designed and implemented an XQuery extension based on a simple `INSERT-UPDATE-DELETE` semantic. This extension allows the manipulation of XML data in a declarative and very easy to use style, comparable to that found in the SQL manipulation language.

Note that xml data manipulation is available for in-memory tables only. To manipulate file based data you have to use a simple three step process as described below.

- (Create a XML table).

    1. Load the file into memory.
    2. Update the table data.
    3. Save the table back into file.

- (Destroy the table).

Four different data manipulation categories are available:

- Renaming Elements and Attributes.
- Deleting Elements and Attributes.
- Updating Elements, Element Content and Attribute content.
- Positional Inserting of new Elements and Attributes.

The layout of a data manipulation statement always follows the same simple schema:

```
FOR/LET … WHERE … [RENAME, DELETE, REPLACE, INSERT] …
```

The exact “target” for the Update operation may be selected either in the `FOR/LET` clause or later in the update expression part:

```
FOR $X in TX/abc/target … [R,D,R,I] $X

FOR $X in TX/abc … [R,D,R,I] $X/target
```

First you select the xml nodes to manipulate with a typical XQuery expression and then instead of the `RETURN` clause you apply one of the data update keywords. Bound variables used for update **must always** be selected with a **`FOR`** clause, but you may also use LET clauses in the Select part for additional conditions in selecting the node set.

Please note that in the current version of SQL/XML IMDB it is only possible to apply **one** update keyword at a time. Therefore multiple updates on the same node set are not allowed currently, but this will change in a future version of SQL/XML-IMDB.

Rename Element or Attribute

```
FOR $X in TX[/xpath] ... RENAME $X[/xpath] TO 'NewName'
```

To select the exact element for renaming specify it either in the FOR or the RENAME part of the query.

```
FOR $X in TX/book/price ... RENAME $X TO 'soldPrice'

FOR $X in TX/book ... RENAME $X/price TO 'soldPrice'

FOR $X in TX/book ... RENAME $X/@isbn TO 'bookId'
```

Delete Element or Attribute

```
FOR $X in TX[/xpath] ... DELETE $X[/xpath]
```

You may either delete an Element, or an Attribute. To delete the element content only use REPLACE as described below. Deleting an Element also deletes the descendants.

```
FOR $X in TX/book/price ... DELETE $X

FOR $X in TX/book ... DELETE $X/@isbn

FOR $X in TX/book/@isbn ... DELETE $X
```

Insert a new Element or Attribute

```
FOR $X in TX... INSERT '<>...</>' [INTO, BEFORE, AFTER] $X

FOR $X in TX..., FOR/LET $Y in TX... INSERT  $Y  [INTO,BEFORE,AFTER] $X

FOR $X in TX... INSERT ATTRIBUTE('name', value) [INTO,BEFORE,AFTER] $X

FOR $X in TX... INSERT (SQL/XQuery) [INTO,BEFORE,AFTER] $X
```

- **INTO** inserts the new element always at the end (= append modus) of $X .
- **BEFORE,AFTER** inserts the new element immediately before or after the element whose identity and position is determined by the element $X. If $X is bound to a **set** of elements, the first element is chosen as the reference element.
- **<>...</>** must be a well formed xml string enclosed in single quotation marks.
- **$Y** let you insert elements from the same query on other positions.
- **$Y** might be bound by FOR and LET expressions.
- **value** might be a string or a number. Enclose strings in single quotation marks.

Update Element content

```
FOR $X in TX[/xpath]... REPLACE $X[/xpath]/xxx() WITH value

FOR $X in TX[/xpath]... REPLACE $X[/xpath]/xxx() WITH expression

FOR $X in TX[/xpath]... REPLACE $X[/xpath]/xxx() WITH NULL
```

To replace the value content of an element you must have the suitable type modifier appended to the bound variable (`/xxx()`). Use the corresponding type for "value". Remember to enclose string-values in single quotation marks. To delete the value of an element simply replace it with `NULL`.

```
FOR $X in TX/book/price ... REPLACE $X/number() WITH 99

FOR $X in TX/book... REPLACE $X/title/text() WITH 'XML Unleashed'

FOR $X in TX/book/sold ... REPLACE $X/bool() WITH true

FOR $X in TX/book/@published  REPLACE $X/date() WITH '2002-12-22'

FOR $X in TX/book  REPLACE $X/@published/date() WITH '2002-12-22'
```

Expression examples:

```
FOR $X in TX/book/price ... REPLACE $X/number() WITH $X/number() + 99

FOR $X in TX/... REPLACE $X/title/text() WITH $Y/title/text() + 'abc'
```

Update entire Attribute

```
FOR $X in TX[/xpath]/@attrib... REPLACE $X WITH ATTRIBUTE('name',value)

FOR $X in TX... REPLACE $X[/xpath]/@attrib WITH ATTRIBUTE('name',value)
```

`Value` might be a string or a number.

```
FOR $X in TX/book/@price ... REPLACE $X WITH ATTRIBUTE('sold',99)

FOR $X in TX/book... REPLACE $X/@isbn WITH ATTRIBUTE('id','1234')

FOR $X in TX/bib/@ab REPLACE $X WITH ATTRIBUTE('xy','2002-12-22')

FOR $X in TX/bib REPLACE $X/@ab WITH ATTRIBUTE('xy','2002-12-22')
```

Update entire Element

```
FOR $X in TX... REPLACE $X[/xpath] WITH '<>...</>'

FOR $X in TX..., FOR/LET $Y in TX... REPLACE $X[/xpath] WITH $Y

FOR $X in TX... REPLACE $X[/xpath] WITH (SQL/XQuery)

FOR $X in TX... REPLACE $X[/xpath]/children() WITH ...
```

- The argument following **WITH** completely replaces the element $X. If $X is bound to a **set** of elements, **all** elements will be replaced one by one.
- **<>...</>** must be a well formed xml string enclosed in single quotation marks.
- **$Y** let you replace the $X element with elements from the same query but from other part of the xml tree.
- **$Y** might be bound by FOR and LET expressions.
- Use the keyword **/children()** to replace all child elements with a new content leaving the parent ($X) intact.

API Usage for C++

The database is accessed through API function calls. No ODBC or ADO connection is required.

Prerequisites

Add the include file to every c++ source file which uses the database object CIMDb.

```
#include <imdb.h>
```

Add the library file to your project (Visual C++).

To do these, click on Project/Settings… The Project Settings Dialog opens. In the configurations Combobox select „All configurations“. Click on the „Link“ Tab Sheet. In the field „Object/library modules:“ enter the name of the IMDB library IMDBvc(u).lib. For UNICODE applications use IMDBvcu. For BORLAND use IMDBbo(u).

Initialize and close the database

First, a variable of type IMDb must be declared which **must** remain valid during the whole application runtime. You can create an application global variable by defining the variable outside any function, or by making the variable a member of any application object, which remains valid during the whole application runtime. This variable keeps the database open, until application shutdown. If this variable goes out of scope and no other IMDb object is active at that time , **the entire database is closed automatically**!

For windows applications make the variable a member of the Application object. To do this, declare the CIMDb variable in the Application header file. For non-windows applications select any other appropriated object or create the variable at global scope.

```
class CMyApp : public CWinApp
{
    public:
        CMyApp();
          ...

          CIMDb  m_db ;
          ...
          ...
}
```

In your applications initialize function open the database.

```
BOOL CMyApp::InitInstance()
{
      ...

      m_db.Open() ;
      ...
      ...
}
```

Note: *If you want to overwrite the default reserved memory space, call* ***SetMaxLMemory*** *and* ***SetMaxSHMemory*** *just before the call to the* ***Open*** *function.*

In your applications last function call, close the database.

```
int CMyApp::ExitInstance()
{
      ……
      ……

      m_db.Close() ;
}
```

Creating SQL Tables

Tables can be created at any time and any location in your application. But for best practice try to put together all table create statements in one “table instancing” function.

```
bool CMyApp::CreateTables()
{
  CIMDb db
  CString sql

  sql =  "CREATE TABLE A
         (
             a COUNTER PRIMARY KEY,
             b BOOL NULL ,
             c INT NULL KEY,
             d REAL NULL ,
             e CHAR(21)
         )"

  if( !db.ExecSql(sql) )
  {
     return db.GetLastErrorNumber()
  }
```

```
    sql =  "CREATE TABLE SHARED B
           (
               a COUNTER PRIMARY KEY,
               b BOOL NULL ,
               c INT NULL KEY,
               d REAL NULL ,
               e CHAR(21) UNIQUE KEY
           )"

    if( !db.ExecSql(sql) )
    {
       return db.GetLastErrorNumber()
    }

    return true
  }
```

Note: *To create a table in **shared memory** simple use the keyword „SHARED“ in the CREATE TABLE statement.*

In the CREATE TABLE statement use KEY to create a multi-value index, use UNIQUE KEY or PRIMARY KEY to create a single-value index.

Inserting Data

Use the standard SQL insert statement to insert data into the tables or insert the data through the API **Insert** function.

SQL insert:

```
CIMDb db

if( !db.ExecSql( "INSERT INTO A VALUES(1, 2.2, 'Hello worl')" ) )
{
   return db.GetLastErrorNumber();
}
```

API Insert function:

```
db.SetIntVal("a",1)
db.SetDblVal("b",1.1)
db.SetCharVal("c","Hello world")

// If the table contains a COUNTER column
// than the next incremented value is returned by Insert
count = db.Insert("A")
```

Updating Data

Use the standard SQL update statement to update data in the tables or update the data through the cursor based API **Update** function.

SQL update:

```
if(!db.ExecSql( "UPDATE A SET a=99, b=10.3" ))
{
      return db.GetLastErrorNumber();
}
```

API Update function:

```
if(!db.ExecSql("SELECT a,b,c FROM A WHERE c IS NULL FOR UPDATE"))
{
      return db.GetLastErrorNumber();
}

while( db.Next() )
{
      db.SetCharVal(3,"Hello world")
      db.Update()
}
```

Note: Use „**FOR UPDATE**“ in the select statement to open an updateable cursor. Columns which contain BLOB data can only updated (or inserted) through the API Update (or Insert) function.

Retrieving data

Every SELECT statement automatically opens a cursor through which you can retrieve the column data. Retrieving data is really very simple as shown by the following examples:

```
if( !db.ExecSql( "SELECT a,b,c FROM A WHERE c = 5 ORDER BY a" ) )
{
      return db.GetLastErrorNumber();
}

while( db.Next() )
{
      a = db.GetIntVal(1)
      b = db.GetDblVal(2)
      c = db.GetCharVal(3)
}
```

Note: *If you use **Next** on a cursor with undefined position (after it was created), the cursor position is automatically set to the first row. The function returns FALSE when the function tries to move to a position after the last row. Therefore the above example is equivalent to the following.*

```
if( !db.ExecSql( "SELECT a,b,c FROM A WHERE c = 5 ORDER BY a" ) )
{
      return db.GetLastErrorNumber();
}

db.First()

do
{
      a = db.GetIntVal(1)
      b = db.GetDblVal(2)
      c = db.GetCharVal(3)

} while( db.Next() )

db.First()

for( int i=0;  i < db.RowCount(); i++ )
{
      db.Next()
      a = db.GetIntVal(1)
      b = db.GetDblVal(2)
      c = db.GetCharVal(3)
}
```

Transactions

Supporting transactions is easy.

```
CIMDb db

db.BeginTransaction()

if( !db.ExecSql( "UPDATE A SET a=99, b=10.3" ) )
{
      db.Rollback()
      return db.GetLastErrorNumber();
}

if( !db.ExecSql( "INSERT INTO A VALUES(1,2,3,...)" ) )
{
      db.Rollback()
      return db.GetLastErrorNumber();
}

db.Commit()


CIMDb db

db.BeginTransaction()

if(!db.ExecSql("SELECT a,b,c FROM A WHERE c IS NULL FOR UPDATE"))
{
      db.Rollback()
      return db.GetLastErrorNumber();
}

while( db.Next() )
{
      db.SetCharVal(3,"Hello world")
      db.Update()
}

db.Commit()
```

Note: *Tables can only created outside a transaction. If an object goes out of scope without finishing the transaction, the databases will automatic rollback the entire transaction.*

API Usage for C Style and Generic Environments

SQL/XML-IMDB was developed with portability in mind. The DLL version of the library can be used in **all** environments (PERL, PHP, DELPHI, …) where generic access to the database library is based on simple "C" style DLL function calls. Either use the provided import library (IMDBg.lib) with the static function declarations or load the library through the dynamic load facilities of your development environment. Consult your development environment documentation for the specific details of loading and using generic DLL libraries.

The database is accessed through API function calls. No ODBC or ADO connection is required.

Prerequisites

Add the include file to every source file which uses the database (for C based development environments). An import library is available (IMDBg(u).LIB).

```
#include <imdbdll.h>
```

For dynamic DLL loading, please consult the documentation of your development environment.

Initialize and close the database

All API functions for the generic version of the library are preceded by "IMDB_".
The API functions require as the first parameter a handle to an internal database object. Before using any API functions, create the handle for the subsequent API calls. Creating a handle can be done in whatever order you want, at any time and any location in your application. To create the handle, use the function IMDB_Create:

```
long myDbHandle = IMDB_Create()
```

You can create as many handles as you require in your application functions. When done with a handle, destroy the handle. The library frees any allocated resources associated with this handle. Destroying a handle can be done in whatever order at any time and any location in your application.

```
IMDB_Destroy( myDbHandle )
```

Note!
At least, one special handle should be created which must **remain valid** during the whole application runtime. Use this handle to open the database (calling IMDB_Open()) and to keep the data base open during the application runtime. You create an application global handle by declaring the variable outside any function.

Destroy this special handle in your last called function.

Before using any other API function you must call **IMDB_Open**. The IMDB_Open function initializes internal data structures and the dynamic memory manager. This function is called once and **only once** in your application. You typically call the Open function with the global created handle as the parameter. When you are done with your application, call **IMDB_Close**. All allocated storage is released back to the operating system. Destroy the global handle after calling Close (IMDB_Destroy).

For shared-memory operations you can use the **IMDB_OpenWithName** function. If you use this function, the tables in the shared memory space will be made persistent to that file, after closing the database. The shared-table content is immediately available the next time, when you open the database from the same file.

Note: For UNICODE applications use the IMDBgu.dll DLL.

See the example on the next page:

```
long hDbg;  // global handle
-----------------------------------------

YourFirstFunction()
{
      hDbg = IMDB_Create();
      IMDB_Open(hDbg);
}

AnyFunction()
{
      char *pString;
      long hDb1, hDb2;
      int count;

      hDb1 = IMDB_Create();

      if( !IMDB_ExecSql(hDb1,"SELECT * FROM A") )
      {
            return IMDB_GetLastErrorNumber(hDb1);
      }

      IMDB_First(hDb1);
      char *pString = IMDB_GetCharVal(hDb1,3,&size);

      // any other todo's
      ...

      // create a 2nd handle
      hDb2 = IMDB_Create();

      if( !IMDB_ExecSql(hDb2,"SELECT * FROM A WHERE a=3") )
      {
            return IMDB_GetLastErrorNumber(hDb2);
      }

      count = IMDB_RowCount(hDb2);

      // when done, destroy
      // the 2nd handle
      IMDB_Destroy(hDb2);

      ...

      // when don with the 1st
      // handle, destroy it
      IMDB_Destroy(hDb1);
}

YourLastFunction()
{
      // close database
      IMDB_Close(hDbg);
      // destroy global handle
      IMDB_Destroy(hDbg);
}
```

API Usage for VB

Access the database through API calls. No ODBC or ADO connection is required.

Prerequisites

Register the IMDBvb ActiveX DLL with Visual Basic. (for text data to be stored in UNICODE format, use the IMDBvbu DLL)
To do this click on Project/References… The References Dialog opens. Click on the Browse… Button. The Add Reference Dialog opens. Navigate to the directory where the IMDBvb DLL is located. Click to select the DLL and click Open. The DLL is now registered with visual basic.

Initialize and close the database

First, a variable of type IMDb must be declared which must remain valid during the whole application runtime. You create an application global variable by declaring the variable outside any function. This variable keeps the database open. If this variable goes out of scope and **no** other IMDb object is active, the entire database is closed.

In your applications initialize function open the database.

```
Dim dbg As Object   // global variable keeps database open

Private Sub Form_Load()

    Set dbg = CreateObject("QuiLogic.IMDB.1")
  or
    Set dbg = CreateObject("QuiLogic.IMDBu.1")  // UNICODE

    dbg.Open

End Sub
```

Note: *If you want to overwrite the default reserved memory space, call* ***SetMaxLMemory*** *and* ***SetMaxSHMemory*** *just before the call to the* ***Open*** *function.*

```
dbg.maxLMemory = 67108864
dbg.maxSHMemory = 67108864
dbg.Open
```

In your applications last function, close the database.

```
Private Sub Form_Unload(Cancel As Integer)

    dbg.Close

End Sub
```

Creating SQL Tables

Tables can be created at any time and any location in your application. For best practice, try to put together all table create statements in one table instancing function.

```
Private Sub Form_Initialize()

    Set db = New IMDB(u)
    db.ExecSql "CREATE TABLE A (a int, b real, c varchar(10) )"
    db.ExecSql "CREATE TABLE SHARED B (a int, b real,c varchar(10))"

End Sub
```

Note: *To create a table in shared memory simple use the keyword SHARED in the CREATE TABLE statement*.

Inserting Data

Use the standard SQL insert statement to insert data into the tables or insert the data through the API **Insert** function.

SQL insert:

```
Private Sub Command1_Click()

    Set db = New IMDB(u)
    db.ExecSql "INSERT INTO A VALUES(1,1.1,'hello world')"
    db.ExecSql "INSERT INTO A VALUES(2,2.2,'hello world2')"

End Sub
```

API Insert function:

```
Private Sub Command1_Click()

    Set db = New IMDb(u)

    db.SetIntValByName "a", 1
    db.SetDblValByName "b", 1.1
    db.SetCharValByName "c", "Hello world"
    db.Insert "A"

    db.SetIntVal 1, 2
    db.SetDblVal 2, 2.2
    db.SetCharVal 3, "Hello world2"

    // If the table contains a COUNTER column
    // than the new value is returned
    counter = db.Insert "A"

End Sub
```

Updating Data

Use the standard SQL update statement to update data in the tables or update the data through the cursor based API **Update** function.

SQL update:

```
Private Sub Command1_Click()

    db.ExecSql "UPDATE A SET a = 10, b = 99 WHERE c IS NULL"
End Sub
```

API Update function:

```
Private Sub Command1_Click()

    Set db = New IMDb(u)

    db.ExecSql "SELECT * FROM A WHERE c IS NOT NULL FOR UPDATE"

    While db.Next

        db.SetCharVal 3, "Hello earth"
        db.Update
    Wend

End Sub
```

Note: *Use „FOR UPDATE“ in the select statement to open an updateable cursor. Columns which contains binary data can only updated (or inserted) through the **Update** (Insert) function.*

Retrieving data

Every SELECT statement automatically opens a cursor to retrieve the column data. Retrieving data is really very simple, consider the following example:

```
db.ExecSql "SELECT * FROM A WHERE a <> 0"

    While db.Next

        A = db.GetIntVal(3)
        B = db.GetDblVal(3)
        C = db.GetCharVal(3)

    Wend
```

Note: *If you use **Next** on a cursor with undefined position (after it was created), the cursor position is automatically set to the first row. The function returns FALSE when the function attempts to move to a position after the last row.*

The above code example is equivalent to examples below.

```
db.ExecSql "SELECT * FROM A WHERE a <> 0"

db.First

Do
        A = db.GetIntVal(1)
        B = db.GetDblVal(2)
        C = db.GetCharVal(3)

Loop While db.Next
```

OR

```
db.ExecSql "SELECT * FROM A WHERE a <> 0"

For i = 1  To db.RowCount

     db.Next
     C = db.GetCharVal(3)

Next i
```

OR

```
db.ExecSql "SELECT * FROM A WHERE a <> 0"

db.First

For i = 1  To db.RowCount

     C = db.GetCharVal(3)
     db.Next

Next i
```

Transactions

Supporting transactions is easy

```
db.BeginTransaction

db.ExecSql "INSERT INTO A VALUES(1,1.1,'hello world')"
db.ExecSql "INSERT INTO A VALUES(2,2.2,'hello world2')"

db.Commit()


db.BeginTransaction()

db.ExecSql "SELECT * FROM A WHERE c IS NOT NULL FOR UPDATE"

    While db.Next

        db.SetCharVal 3, "Hello earth"
        db.Update
    Wend

db.Commit()


db.BeginTransaction()

If db.ExecSql ("UPDATE A SET a = 99 WHERE x = 1") = 0 Then
         db.Rollback
         Return
Else
         db.Commit

End If
```

Note: *Tables can only created outside a transaction. If an object goes out of scope without finishing the transaction, the databases will automatic rollback the entire transaction.*

API usage for Active Server Pages (ASP)

The database is accessed through API calls. No ODBC or ADO connection is required.

Prerequisites

Register the IMDBvb DLL. (For text data to be stored in UNICODE format use the IMDBvbu DLL)
To do this, at the command line prompt enter the following line:

```
Regsvr32 Drive:\Path\imdbvb(u) [Enter]
```

The DLL is now registered with the system

Note: *The installation program has done this for you. Manual registering is only necessary when moving the library to other computers, by manual copying the library.*

Initialize and close the database

First, an object of type IMDb must be declared in your global.asa file that remains valid during the whole application runtime.

To do this, copy the following example lines to your own global.asa file

```
<OBJECT RUNAT=Server SCOPE=APPLICATION ID=IMDb PROGID="QuiLogic.IMDB(u).1"></OBJECT>
    <SCRIPT language="VBScript" runat="server">
    Sub Application_OnStart
      IMDb.Open
    End Sub

    Sub Application_OnEnd
            IMDb.Close
    End Sub
    </SCRIPT>
```

For UNICODE applications use **QuiLogic.IMDBu.1**.

The IMDb object instance created with the above lines has application scope. This is required by the database engine to keep there internal management structures valid and initialized during the whole application runtime.
Don’t use or reference in any way this object instance in your application ASP pages. It exists just for the engines internal book keeping mechanism and to hold the in-memory structures.

Use page scope declared IMDB objects to access the database.

Using SQL/XML-IMDB in your ASP applications

An object created by using **Server.CreateObject** on an ASP page exists for the duration of that page. The object is accessible to any script commands on that page, and it is released when ASP has finished processing the page. Thus, the page-scope created IMDb database objects represent the *workhorses* for your ASP application.

Alternative (or in combination) you my use objects, created in session-scope, but remember:
A session-scope object is created for each new session in an application and released when the session ends; thus, there is one IMDb object per active session. Session scope is used for objects that are called from multiple scripts but affect one user session. A cursor opened in one page for that session object remains open until a new SELECT statement is executed in other pages. Unpredictable side effects can occur. Use session objects with caution. You should give an object session-scope only when really needed, but this should rarely the case because you can always access the table content from every page-scope object. Instancing the page-scope objects with **Server.CreateObject** is fast, because most of the database engine internal initializing work was done when instancing the application-scope object, and the DLL already sits in the memory, waiting for the function calls.

An example for using page-scope objects is given:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final //EN">
<HTML>
<HEAD> <TITLE>IMDB Test</TITLE>
</HEAD>
<BODY>

    <%
        Set db = Server.CreateObject("QuiLogic.IMDB(u).1")

        db.ExecSql "SELECT * FROM A WHERE a <> 0"

        For i = 1 To db.RowCount

        db.Next

    %>
       'output the column values
       <%=db.GetCharVal(3) %>
    <%
        Next
    %>

</BODY>

</HTML>
```

For a detailed discussion of database usage, see: API usage for VB.
For UNICODE applications use **QuiLogic.IMDBu.1**.

API Usage for VBA (including MS-Office)

Many of today commercial available applications have build-in support for VBA scripting. By using our in-memory database engine, developing custom applications in these environments is easy and fast. The engine presents the developer a simple, standards-based SQL/XQuery interface for all their data management requirements.

The database is accessed through API calls. No ODBC or ADO connection is required.

Prerequisites

Register the IMDBvb(u) DLL.
For UNICODE applications use IMDBvbu.
To do this, at the command line prompt enter the following line:

```
Regsvr32 Drive:\Path\imdbvb(u) [Enter]
```

The DLL is now registered with the system

Note: *The installation program has done this for you. Registration is required only, when copying the library to other computers without using the installation program.*

Use the database engine as described in: **API Usage for VB**.

Note: *The object creation mechanism in VBA environments can be different from the Visual Basic case (see below).*

Late Binding and early Binding

Late binding declares a variable as an object. Calling **`GetObject`** or **`CreateObject`** and naming the OLE Automation programmatic identifier (ProgID) initializes the variable. For example, if the ProgID were "QuiLogic.IMDB.1" the code would appear like this:

```
Dim db As Object
Set db = CreateObject("QuiLogic.IMDb.1")
or
Set db = CreateObject("QuiLogic.IMDbu.1")  // UNICODE
```

Late binding was the first binding method implemented in VBA controller products. It consumes more overhead than early binding. It is available in all products capable of being VBA controllers. All OLE-Automation objects support late binding.

Early binding declares a variable as an application-defined object type. A type library, object library, or dynamic-link library is required to declare a variable as an application-defined object type. This library must be checked in the controller application's References dialog box!

The code for early binding would appear like:

```
Dim db As Object

Set db = New IMDB
or
Set db = New IMDBu  // UNICODE
```

Early binding is faster than late binding, but not all VBA environments support it. Check the manual of your VBA Environment for early binding support.

For a detailed discussion of database scripting, see the chapter: API usage for VB.

Note: *VBA environments differ in the extent the VBA language was implemented. Always consult the language reference manual that comes with your VBA environment for the exact syntax details*.

Exchange Data between Applications

To see how easy it is to exchange data by the shared-memory capabilities of the database, follow the steps below. The example steps below demonstrate direct data exchange between an IIS-ASP based application and a Visual Basic application. Other examples (e.g. data exchange between Excel, Word, C++…) can be outlined similar.

1. Step
Create a new physical directory that will receive the files we will create below. It is the home directory of our ASP based application

2. Step
Create a "global.asa" file and add the lines shown below

```
 <OBJECT RUNAT=Server SCOPE=APPLICATION ID=IMDb
PROGID="QuiLogic.IMDBu.1"></OBJECT>

    <SCRIPT language="VBScript" runat="server">

    Sub Application_OnStart

         IMDb.Open

    End Sub

    Sub Application_OnEnd

         IMDb.Close

    End Sub

    </SCRIPT>
```

3. Step
Create a "default.asp" file containing your applications code as below

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final //EN">
<HTML>
<HEAD>
<TITLE>IMDB Test</TITLE>
</HEAD>

<BODY>

    <%
         Set db = Server.CreateObject("QuiLogic.IMDBu.1")

         db.ExecSql "SELECT * FROM A WHERE a <> 0 ORDER BY c"
```

```
            For i = 1 To db.RowCount

            db.Next

        %>
             'output the column values
             <P>
            <%=db.GetCharVal(3) %>
            </P>
        <%
            Next
        %>

    </BODY>
    </HTML>
```

4. Step
Add the files "global.asa" and "default.asp" to your applications newly created physical directory. Create a new Web-site with the Internet Information Manager. Use the default values as guided by the Web-site creation wizard.

5. Step
Access the Web-site with your favorite Browser and the output is:

```
QuiLogic.IMDB.1- Error '80004005'
Syntax Error: Table not found in catalog
/Default.asp, line 18
```

OOP's, you got an error because no table named "A" exist in the database.

Let us start creating the table now. We do that in the other application process. Let the IIS running, let the Browser open, we will create the table and its data content on the fly, demonstrating the dynamic behavior of IMDB based data exchange.

6. Step
Create a new Visual Basic application. Add the following code to the application:

```
Private Sub Form_Load()

    Dim db As Object
    Set db = CreateObject("QuiLogic.IMDBu.1")
    db.Open
    db.ExecSql "CREATE TABLE SHARED A (a int,b real,c varchar(10))"
    db.ExecSql "INSERT INTO A VALUES(1,1.1,'Hello world1')"
    db.ExecSql "INSERT INTO A VALUES(2,1.1,'Hello world2')"
    db.ExecSql "INSERT INTO A VALUES(3,1.1,'Hello world3')"
    db.ExecSql "INSERT INTO A VALUES(4,1.1,'Hello world4')"
    db.Close

End Sub
```

7. Step

Run the application.

Let the Visual Basic application stay active, switch to the Browser and reload the page. Now you get the output:

Output:

Hello world1
Hello world2
Hello world3
Hello world4

Close the Visual Basic application, switch to the Browser and reload the page.

Output:
Hello world1
Hello world2
Hello world3
Hello world4

The output remains the same, although the VB application (where the table and data content was defined) was shutdown.

The table and data content was created in the shared-memory space of the database by the Visual Basic application and outputted by the ASP application. As long as the table is shared by **at least** one running application, the data in the shared tables remain in memory. Applications can connect and disconnect, edit the data, and output the data with a few simple statements of scripting code. All the complexities are managed by the IMDb database engine behind the scene.

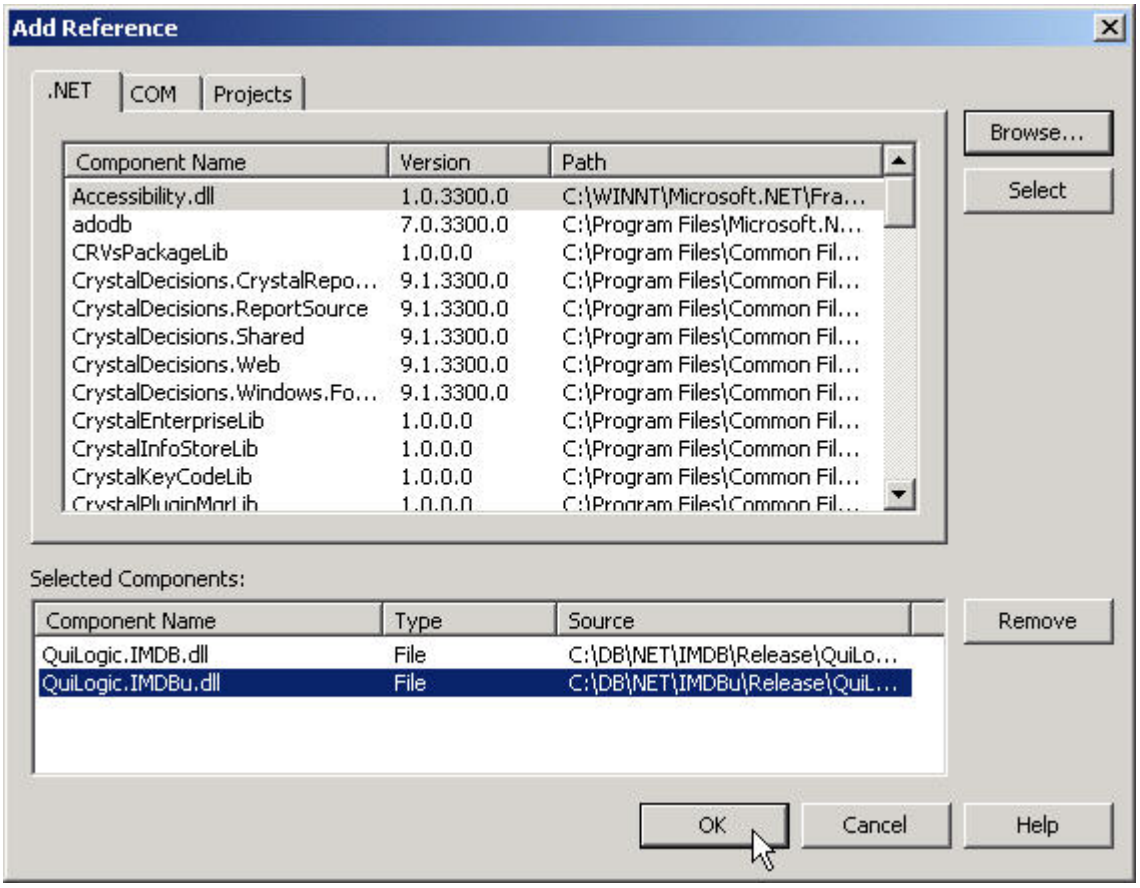
Remember: Do not mix the ANSI and UNICODE versions of the library in data exchange scenarios, because the strings will contain garbage when exchanging between environments with different character encoding schema.

API Usage for NET (C#, VB.NET, ASP.NET)

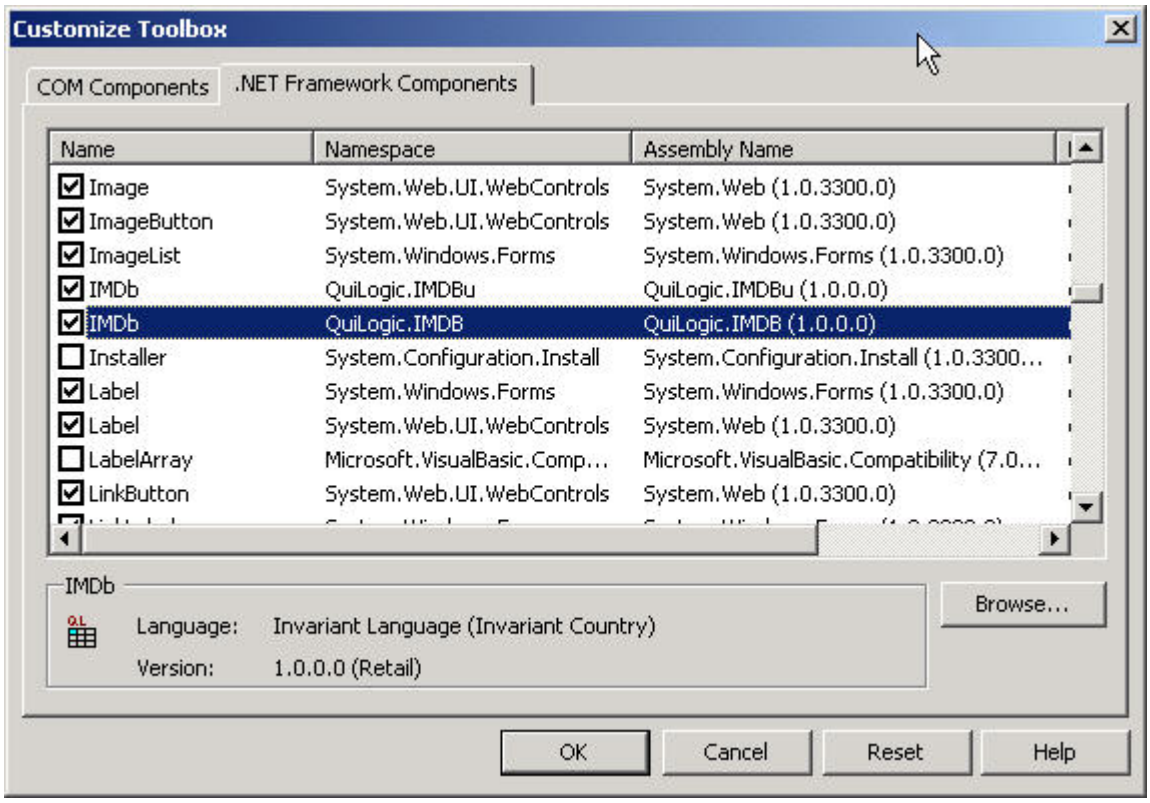
The database is accessed through API calls. No ODBC or ADO connection is required.

To install SQL/XML-IMDB component for .NET:

Select the Project->Add->Reference menu. In the displayed dialog select the .NET tab, browse to .../NET/QuiLogic.IMDB(u).dll, and hit OK. (u is for the UNICODE version)



To add the IMDB(u) Assembly to the Visual Studio .NET toolbox, select the "Components" tab of the toolbox and select "Customize Toolbox...", Select ".NET Framework Components" and then browse for the QuiLOgic.IMDB(u).dll. Hit the OK button and go ...



Additional Prerequisites

Add the `using` directive:

```
using QuiLogic.IMDB;
or
using QuiLogic.IMDBu; // UNICODE
```

For more details please refer to the C#, VB.NET and ASPX examples located in the ../Example/NET directory of the SQL/XML-IMDB installation.

Supported Data Types

SQL/XML-IMDB supports the following data types for the table columns:

Numeric	4/8 bytes (32/64 bit value) INT, INTEGER, SHORT, LONG, SMALLINT
Decimal	8 bytes (DOUBLE) REAL, FLOAT, DOUBLE, SINGLE, CURRENCY
Bool	1 bit BOOL, BOOLEAN, YESNO
Counter	8 bytes Auto-Increment value COUNTER
Character	1 byte per character (2 bytes for UNICODE) Zero to a maximum of 256 MB/row CHAR(n), VARCHAR(n), CHARACTER(n), TEXT
Date/Time	8 bytes 0 to year 20.000, Time 00:00:00.000.000.0 to 24:59:59.999.999.999 100 nano-seconds resolution DATE, TIME, DATETIME
Binary	Zero to a maximum of 256 MB/row BLOB, LONGBINARY
GUID	For storing Guid values.

Remarks

- The engine stores columns of type **Bool** very space efficient. Bool columns consume only one Bit/Column. Therefore use **Bool** type columns instead of integer columns in all two-valued data storage scenarios.
- No index can be created for **Bool** and **Binary** typed columns.
- Binary columns can only be edited/updated by API based functions like `Insert`, `Update`, `SetBlobVal` and `GetBlobVal`.
- Counter type columns are read only. The increment step is 1 by default for every inserted row and can not be changed. Counter values for deleted rows are never reused.

Defaults and Limits

Maximum data store size	~ 2GB
Maximum number of tables	65,535
Maximum number of columns on table	256
Maximum number of indexes on table	64
Maximum length of table names	64
Maximum length of column names	64
Maximum number of rows in a table	2 Billion
Maximum length for fixed-length column	4096
Maximum length for variable-length column	256 MB
Maximum size for binary columns	256 MB
Maximum number of simultaneously open cursors	Unlimited (memory depending)
Maximum number of columns in an index	1
Maximum number of XML nodes in a table	2 Billion
Maximum of simultaneously active IMDB objects	Unlimited (memory depending)

Remarks

The maximum data store size is limited by the operating system to the value shown above. The actual, **usable** size depends on the available virtual memory size. Please note that you have to consider the combined size of both, the process-local memory and the shared-memory space. For best performance always try to have the size of the installed physical memory (RAM) comparable to the expected memory requirements of your application.

As a rule of thumb the required size for the data in memory is approximately 2-3 times the size of the data on disk file depending on the number and type of indexes.

Character data is always stored in variable length arrays regardless of defining it as either CHAR(n) or VARCHAR. This safes a lot of memory and ensures maximum performance.

Note: *Use **GetLMemoryUsed** and **GetSHMemoryUsed** to query the actual allocated memory when running your application, and use the reported values for adjusting the installed physical (RAM) memory.*

Supported Development Environments

SQL/XML-IMDB ships in a wide variety of library forms.

IMDBvc.lib	Static link library for MS-Visual C++
IMDBbo.lib	Static link library for Borland C++
IMDBvcu.lib	Static link library for MS-Visual C++ (UNICODE)
IMDBbou.lib	Static link library for Borland C++ (UNICODE)
IMDBg	Import library for the generic C style DLL
IMDBgu	Import library for the generic C style DLL (UNICODE)
IMDBvb	ActiveX library for all VB(A) environments
IMDBvbu	ActiveX library for all VB(A) environments (UNICODE)
QuiLogic.IMDB.dll	NET Assembly
QuiLogic.IMDBu.dll	NET Assembly (UNICODE)
IMDB.PAS	Static Import Unit for Delphi
IMDBu.PAS	Static Import Unit for Delphi (UNICODE)

SQL/XML-IMDB was developed with portability in mind. The dependence to external libraries where keep to a minimum. Only the standard C/C++ runtime library was linked in the build process for SQL/XML-IMDB.

SQL/XML-IMDB was tested on all standard 32 Bit Windows platforms, respective Win95/98/NT/2000/XP

At time of this writing, the following application development environments have been tested:

- Microsoft .NET
- Microsoft Visual C++ 6.0
- Microsoft Visual Basic 6.0
- Microsoft Office 97/2000/XP
- Microsoft IIS/ASP
- Borland C++ 5.0/6.0
- Borland Delphi 5.0/6.0/7.0
- Automation Controller Environments (VBA)
- PHP
- Perl

Contact QuiLogic for additional Compiler supports. We do our best do support you, If possible we can send you a special version. Depending on our effort, there might be an additional handling fee for the special versions.

The ActiveX version of SQL/XML-IMDB should work in all automation controller environments (VBA). We have tested more than 25 controller environments and found no problems in development and running of SQL/XML-IMDB in these environments.

QuiLogic is currently developing special versions of SQL/XML-IMDB for serving the embedded controller market. Please contact QuiLogic for further informations or visit our Web-site.

QuiLogic is currently developing a **UNIX** version of SQL/XML-IMDB. Please contact QuiLogic for further informations or visit our Web-site.