CompleteDB

CompleteDB Embedded

User Guide Version 3.3

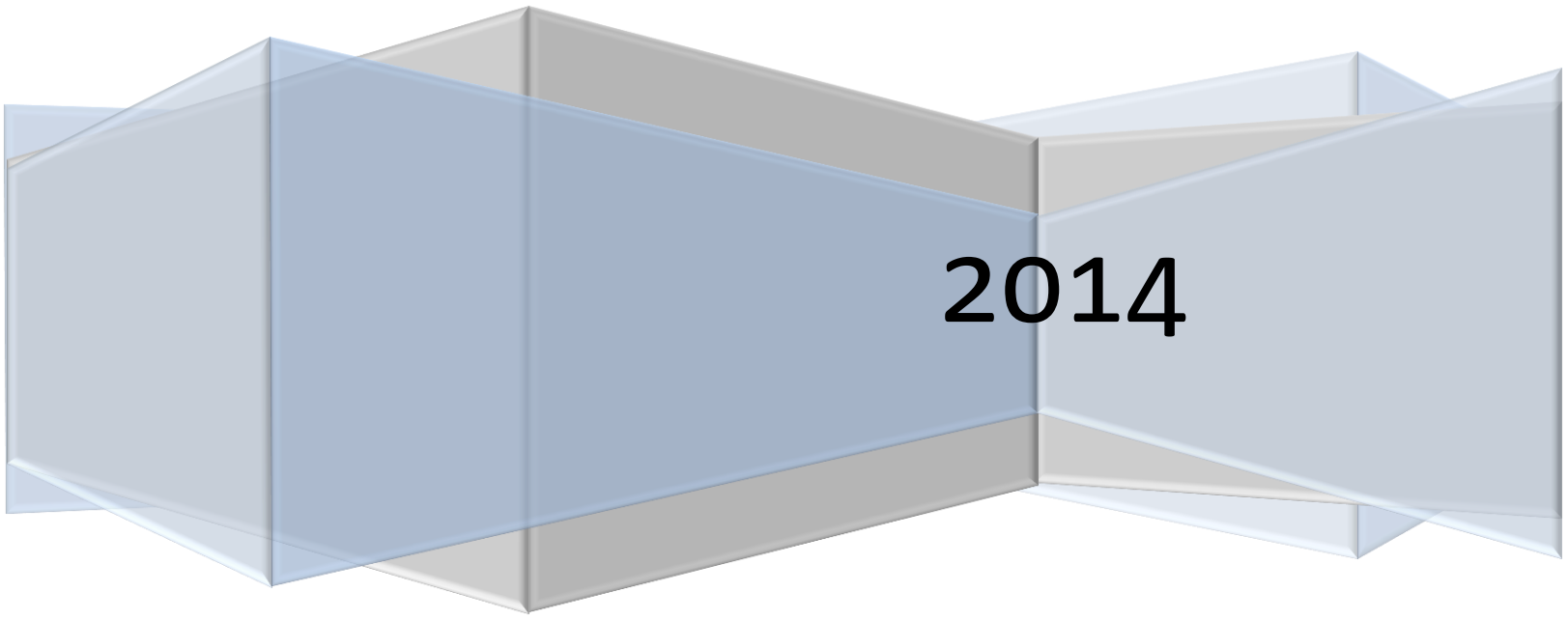

Table of Contents

CompleteDB Embedded

CompleteDB Embedded is an all-in-memory, high-performance, embedded database. The first version of our product is an innovative combination of uniquely engineered Transaction and Index Engines that

- Exploit the latest trends and innovations in multi-core parallel algorithms,
- Execute at the speed of tens of millions of transactions per second, and
- Achieve vertical scalability--more cores, more RAM.

Our product is optimized for applications that require fast, low-latency transactional processing. With newly added persistence feature, we now fully support all **ACID** properties: Atomicity, Consistency, Isolation and Durability.

Product advantages

Fast

CompleteDB Embedded is the fastest product on the market. It exploits hardware to its fullest extent through utilization of the most advanced multi-core data processing algorithms and optimal memory allocations.

Easy to use

CompleteDB Embedded enables immediate programmer productivity. It provides an intuitive and easy to use API.

Roadmap

Features in Release 3.3

Tables, columns, metadata, primitive data-types, string data-type, C++ Direct API (DDL and DML), concurrent ACID transactions, MVCC, durability, DB Service Instance, cross platform (Windows and Linux), surrogate key, column hash indexes (unique and non-unique), composite hash indexes (unique and non-unique).

Planned features in future releases

Expressions, joins, views, events, SQL, composite indexes, tree indexes, additional non-primitive data-types, full DDL, .Net API, Java API, Python API, ODBC and JDBC.

OS / Platform and Compiler support

Release 2.0 supports all 64-bit versions of Microsoft Windows and Linux RedHat running on an x64 platform.

The Direct C++ API compiles in Microsoft Visual Studio 2012, 2010 and 2008.

Configuration

To configure Microsoft Visual Studio C++ Project to use C++ Direct API.

1. Create **x64 configuration** for your VC++ project.
2. Add **include** directory to C++-->General-->Additional Include Directories
3. Add **completedb.lib** to Linker-->Input-->Additional Dependencies
4. Add **#include "CompleteDb.hpp"** directive to your source file
5. Make sure that **bin** directory is part of the **PATH** environmental variable or copy **completedb.dll** from bin directory to one of the existing PATH directories.
6. Optionally add `using namespace completedb;` directive to your source files.

Sample Projects

CompleteDB Embedded Direct C++ API project samples provided in sample directory of the installed package.

QuickStart

QuickStart project demonstrates how to use the main features of Direct C++ API.

UserGuide

UserGuide project contains all code examples used in this document.

Benchmarks Projects (Visual Studio 2012 only)

Delete
In this benchmark project we test DELETE only performance on a single thread.

Insert
In this benchmark project we test INSERT only performance on multiple threads.

InsertDelete
In this benchmark project we test INSERT/DELETE only performance on multiple threads.

Scan
In this benchmark project we test SCAN only performance on multiple threads.

Select
In this benchmark project we test SELECT only performance on multiple threads.

Update
In this benchmark project we test UPDATE only performance on multiple threads.

Direct C++ API

completedb namespace

All Direct C++ API classes are contained in completedb namespace.

```
using namespace completedb;
```

Quick Start

```
// connect to new or existing database
Database db;
bool done = db.connect("TickDB");

//-----------------------------------------------------------------------
// 1) DDL
//-----------------------------------------------------------------------

// CREATE TABLE and columns
CreateTable createTable = db.createTable("Stocks");

createTable.addColumn("Bid", ColumnType::Double);
createTable.addColumn("Ask", ColumnType::Double);

ColumnMetadata colSymbol = createTable.addColumn("Symbol", ColumnType::String);
colSymbol.setLength(8);

done = createTable.execute();

//-----------------------------------------------------------------------
// 2) DML
//-----------------------------------------------------------------------

// bind columns to the Record statement
Table table = db.getTable("Stocks");
done = table.exists();
Record record = table.getRecord();

ColumnDouble bid = record.getColumnDouble("Bid");
ColumnDouble ask = record.getColumnDouble("Ask");
ColumnString symbol = record.getColumnString("Symbol");

// optional step: prepare bound columns and performs validation
done = record.prepare();
```

```
//------------------------------------------------------------------------
// 3) INSERT data using user defined key value: SurrogateId
//------------------------------------------------------------------------

// INSERT (long transaction)
SurrogateId idMSFT = 1;
done = record.insertSurrogate(idMSFT);
symbol.setValue("MSFT");
bid.setValue(31.08);
ask.setValue(31.34);

// INSERT (short transaction)
SurrogateId idGOOG = 2;
done = record.insertSurrogate(idGOOG);
symbol.setValue("GOOG");
bid.setValue(614.29);
ask.setValue(615);
// COMMIT individual row (short transaction)
done = record.commitRow();

// COMMIT long transaction
done = db.commit();

//------------------------------------------------------------------------
// 4) SELECT
//------------------------------------------------------------------------

done = record.loadSurrogate(idMSFT);
std::cout
       << "Symbol: " << symbol.getValue()
       << "   Bid: " << bid.getValue()
       << "   Ask: " << ask.getValue()
       << std::endl;
```

```
//---------------------------------------------------------------------------
// 5) DELETE and UPDATE as part of the same transaction
//---------------------------------------------------------------------------

// DELETE
done = record.deleteSurrogate(idMSFT);

// UPDATE
done = record.holdSurrogate(idGOOG);
bid.setValue(615);
ask.setValue(615.30);

// COMMIT
done = db.commit();

// error handling example
if (!done) {
	std::cout
		<< "  error: " << db.getLastErrorId()
		<< "message: " << db.getLastError()
		<< std::endl;
}

//---------------------------------------------------------------------------
// 6) TABLE SCAN
//---------------------------------------------------------------------------
// print all symbols in the table
record.resetRowIterator();

while (record.nextRow()) {
	std::cout
		<< symbol.getValue()
		<< std::endl;
}
```

```
//-----------------------------------------------------------------------
// 7) METADATA USAGE
//-----------------------------------------------------------------------

// print all tables names
TableIterator itTable = db.getTables();
while (itTable.next()) {
	std::cout
		<< "Table: " << itTable.getName()
		<< std::endl;

	// print all columns names
	ColumnIterator itColumn = itTable.getColumns();
	while (itColumn.next()) {
		std::cout
			<< "\t"
			<< "Column: " << itColumn.getName()
			<< std::endl;
	}
	std::cout << std::endl;
}
```

Entities and Statements

Standard database entities are represented by Entity Classes. SQL operations are represented by Statement Classes.

Entity Classes	**Statement Classes**
Database Table Column TableIterator ColumnIterator	CreateTable Record

Entities have particular properties and accessors that are logically related to other Entities and Statements. Statements are retrieved from related Entities on which operations are performed.

Record statement

The **Record** statement is used to insert, select, update or delete a single row in a database table.
It was inspired by "Active Record" architectural pattern.

Columns

The **Column** entity is used to insert, select or update row data in a database table. **Columns** have to be bound to a **Record** statement.

Column_Type boundColumn = Record.getColumn_Type();

Supported column types

Column	C/C++ equivalent
`ColumnBool`	`bool`
`ColumnDouble`	`double`
`ColumnFloat`	`float`
`ColumnInt8`	`int8_t`
`ColumnInt16`	`int16_t`
`ColumnInt32`	`int32_t`
`ColumnInt64`	`int64_t`
`ColumnString`	`const char*`

Objects not pointers

All functions that return non-built-in data-types return C++ objects rather than pointers. This feature makes for easy syntax without compromising performance. When one object is assigned to another, they both refer to the same instance. Once all objects of the same instance go out of scope, the instance is automatically destroyed. This approach is similar to *shared pointer semantics*.

```
//Both d1 and db2 refer to the same instance.
Database db1;
db1.connect("TickDB");
Database db2 = db1;
```

Storing objects in STL collections

The above feature enables objects to be stored in STL collections without additional performance loss and memory overhead.

```
Database db;
db.connect("TickDB");
createTable();
Record record = db.getTable("Stocks").getRecord();

vector<ColumnInfo> columns;
columns.push_back(record.getColumnString("Symbol"));
columns.push_back(record.getColumnDouble("Bid"));
columns.push_back(record.getColumnDouble("Ask"));

//must be called to bind retrieved columns
record.prepare();

for (vector<ColumnInfo>::iterator it = columns.begin(); it != columns.end(); it++) {
	cout << (*it).getName() << endl;
}
```

RTTI

Direct C++ API permits performing safe typecasts and querying class information at run time.
All API classes support RTTI by inheriting from Object base class.

Object base class RTTI functions:
Type::Enum getClassType()
Type::Enum getObjectType()
bool cast(const Object& from)

```
Database db;
assert(db.getClassType() == Type::Database);
assert(db.getObjectType() == Type::Database);

Object from = db;
assert(from.getClassType() == Type::Object);
assert(from.getObjectType() == Type::Database);

//downcast
Database db2;
assert(db2.cast(from));
```

Error Handling

The Direct C++ API does not use C++ exceptions and does not have the global error handling function. Objects containing functions that return a boolean value indicating success or failure have the following two error accessors;

getLastError() and ***getLastErrorId().***

```
Database db;
db.connect("TickDB");
CreateTable createTable = db.createTable("Stocks");
CreateTable duplicateCreateTable = db.createTable("Stocks");

createTable.execute();
if (!duplicateCreateTable.execute()) {
	cout << duplicateCreateTable.getLastError() << endl;
}
// or
duplicateCreateTable.execute();
if (duplicateCreateTable.getLastErrorId() != Error::None) {
	cout << duplicateCreateTable.getLastError() << endl;
}
```

Errors can be handled on the last operation in the sequence of logically connected operations to avoid bloated error handling code.

Transactions

CompleteDB Embedded supports database transactions for all DML (INSERT, UPDATE, DELETE) operations with **READ_COMMITED** isolation level. Transaction is always implied and therefore does not need to be and cannot be explicitly started. After completing DML operations that are part of the same transaction, it must be committed or roll-back. **Database** entity is used to commit or roll-back current transaction.

CreateTable statement operations are not transaction bound and cannot be roll-back.

Completing transaction on individual rows.

Transactions can be completed immediately on individual rows within the context of a larger transaction, which itself is not completed until the final commit or roll-back. This is done by using the Record ***commitRow****() or* ***rollbackRow****()* function.

Selecting outside current Transaction

loadRowOutside(), loadDirectOutside() and ***loadSurrogateOutside()*** functions enable the user to select the original row from outside of the current transaction context. For example the user deletes a row but does not commit it yet. If the user will try to select the deleted row from the same current pending transaction he will not be able to do it since the row is marked deleted from the view of the current transaction context. These functions enable the user to see the data as it is currently seen by another user.

```
Database db;
db.connect("TickDB");
createTable();
Record record = db.getTable("Stocks").getRecord();
ColumnDouble bid = record.getColumnDouble("Bid");
//insert row
SurrogateId sid = 1;
record.insertSurrogate(sid);
bid.setValue(100);
DirectId id;
record.getDirectId(id);
record.commitRow();
//delete the row
record.deleteDirect(id);
//as far as current transaction concerned the row is gone
bool notFound = !record.loadDirect(id);
assert(notFound);
//but it is still in the table for other transactions
//until current transaction is committed
bool found = record.loadDirectOutside(id);
assert(found);
//ok let's look at the current bid
//last time the value was 100
assert(bid.getValue() == 100);
//commit the transaction
db.commit();
//now the row is gone for everybody
notFound = !(record.loadDirectOutside(id));
assert(notFound);
```

Multithreading

CompleteDB Embedded is optimized to be used in a heavily parallel multithreaded environment. There is no limit on how many database connections from multiple threads or processes can be opened against a single database instance.

All API objects created in a single thread must be used in the context of that thread only and cannot be passed to a different thread.

Access methods

Records in a database table can be accessed using ***DirectId*** or/and ***SurrogateId***.

DirectId

DirectId is an internal database record identifier representing the physical in-memory location where the record resides. It is the fastest method to access a record. ***DirectId*** may be used to select, update or delete records in a table. ***DirectId*** can always be retrieved from the current row.

Usage of uninitialized DirectId has undefined behavior.

SurrogateId

SurrogateId is a system defined key index used to uniquely identify a row in a database table. The use of ***SurrogateId*** is optional at the row level, but if used it must be assigned by the user before inserting the row. ***SurrogateId*** may be used to select, update or delete records in a table.

Database operations

Database Creating /Connecting

To create a Database or connect to a Database call ***connect()*** function on the Database object.
The Database will be auto-created when ***connect()*** function is called for the first time.

```
Database db;
db.connect("TickDB");
assert(db.isConnected());
```

Disconnecting from a Database

When the **Database** object goes out of scope all resources are released and child objects are invalidated. Any operation on a disconnected object will have no effect.

To reclaim resources explicitly call ***disconnect()*** on the Database object.

```
db.disconnect();
assert(db.isDisconnected());
```

When the last Database object is disconnected the Database itself will automatically be deleted.

Creating Table and adding columns

The **CreateTable** statement is used to create a new table. The ***addColumn()*** function on the **CreateTable** statement is used to add a column to a new table. ***addColumn()*** returns **ColumnMetadata** object that has functions to set various properties of a new column.

```
Database db;
db.connect("TickDB");
CreateTable createTable = db.createTable("Stocks");
//add new column; set length and description
ColumnMetadata symbol = createTable.addColumn("Symbol", ColumnType::String);
symbol.setLength(8);
symbol.setDescription("Short abbreviation for traded stock");
//add new column and set column type
ColumnMetadata bid = createTable.addColumn("Bid", ColumnType::Double);
bid.setType(ColumnType::Double);
//add new column and set column type as one operation
createTable.addColumn("Ask", ColumnType::Double);
bool success = createTable.execute();
assert(success);
```

Inserting Rows

Record ***insertSurrogate()*** should be used when a user wants to insert a row with an index key defined by the user, SurrogateId, which may later be used to select, update or delete the row.

DirectId will be automatically generated and can be retrieved from the current record by calling *Record* ***getDirectId()*** function.

```
Database db;
db.connect("TickDB");
createTable();
Record record = db.getTable("Stocks").getRecord();
//bind columns
ColumnString symbol = record.getColumnString("Symbol");
ColumnDouble bid = record.getColumnDouble("Bid");
ColumnDouble ask = record.getColumnDouble("Ask");
//user defined unique value
SurrogateId sid = 1;
//insert using insertSurrogate
bool success = record.insertSurrogate(sid);
assert(success);
symbol.setValue("IBM");
bid.setValue(186);
ask.setValue(187.5);
//commit transaction
success = db.commit();
assert(success);
```

Selecting single Row using DirectId

Record ***loadDirect()*** function is used to a select single record from a database table by **DirectId**.

```
Database db;
DirectId id;
db.connect("TickDB");
createTable();
Record record = db.getTable("Stocks").getRecord();
ColumnString symbol = record.getColumnString("Symbol");
ColumnDouble bid = record.getColumnDouble("Bid");
//insert
SurrogateId sid = 1;
record.insertSurrogate(sid);
record.getDirectId(id);
symbol.setValue("MSFT");
bid.setValue(33);
record.commitRow();
//SELECT symbol, bid FROM Ticks WHERE DirectId = ?
if (record.loadDirect(id)) {
	cout << symbol.getValue() << endl;
	cout << bid.getValue() << endl;
}
```

Selecting single Row using SurrogateId

Record ***loadSurrogate()*** function is used to select a single record from a database table by **SurrogateId**.

```
Database db;
SurrogateId id = 1;
db.connect("TickDB");
createTable();
Record record = db.getTable("Stocks").getRecord();
ColumnString symbol = record.getColumnString("Symbol");
ColumnDouble bid = record.getColumnDouble("Bid");
//insert
record.insertSurrogate(id);
symbol.setValue("MSFT");
bid.setValue(33);
record.commitRow();
//SELECT symbol, bid FROM Ticks WHERE SurrogateId = ?
if (record.loadSurrogate(id)) {
	cout << symbol.getValue() << endl;
	cout << bid.getValue() << endl;
}
```

Table scan (Selecting all Rows)

Record ***resetRowIterator()*** and ***nextRow()*** functions are used to traverse all rows in a database table.

```
Database db;
db.connect("TickDB");
createTable();
Record record = db.getTable("Stocks").getRecord();
ColumnString symbol = record.getColumnString("Symbol");
//generate data
char str[2] = {0};
SurrogateId id = 1;
for (char ch = 'a'; ch < 'z'; ch++) {
	str[0] = ch;

	record.insertSurrogate(id);
	symbol.setValue(str);
    id++;
}
db.commit();
//SELECT symbol FROM Stocks
record.resetRowIterator();
while (record.nextRow()) {
	cout << symbol.getValue() << endl;
}
```

Updating Rows

Rows must be locked for writing before updating to maintain concurrent consistency of the data. Record ***holdRow(), holdDirect()*** and ***holdSurrogate()*** functions are used to lock a row in a database table before setting new values. Updated rows will be unlocked when the transaction is completed.

Updating current Row

holdRow()

```
Database db;
db.connect("TickDB");
createTable();
Record record = db.getTable("Stocks").getRecord();
ColumnString symbol = record.getColumnString("Symbol");
ColumnDouble bid = record.getColumnDouble("Bid");
ColumnDouble ask = record.getColumnDouble("Ask");
//insert
SurrogateId sid = 1;
record.insertSurrogate(sid);
symbol.setValue("MSFT");
bid.setValue(33);
ask.setValue(34);
record.commitRow();
//UPDATE current row
if (record.holdRow()) {
	bid.setValue(35);
	ask.setValue(36);
	record.commitRow();
}
```

Updating Row using DirectId

holdDirect()

```
Database db;
DirectId id;
db.connect("TickDB");
createTable();
Record record = db.getTable("Stocks").getRecord();
ColumnString symbol = record.getColumnString("Symbol");
ColumnDouble bid = record.getColumnDouble("Bid");
ColumnDouble ask = record.getColumnDouble("Ask");
//insert
SurrogateId sid = 1;
record.insertSurrogate(sid);
record.getDirectId(id);
symbol.setValue("MSFT");
bid.setValue(33);
ask.setValue(34);
record.commitRow();
//UPDATE Ticks SET Bid = 35, Ask = 36 WHERE DirectId = ?
if (record.holdDirect(id)) {
	bid.setValue(35);
	ask.setValue(36);
	record.commitRow();
}
```

Updating Row using SurrogateId

holdSurrogate()

```
Database db;
db.connect("TickDB");
createTable();
Record record = db.getTable("Stocks").getRecord();
ColumnString symbol = record.getColumnString("Symbol");
ColumnDouble bid = record.getColumnDouble("Bid");
ColumnDouble ask = record.getColumnDouble("Ask");
//insert
SurrogateId id = 1;
record.insertSurrogate(id);
symbol.setValue("MSFT");
bid.setValue(33);
ask.setValue(34);
record.commitRow();
//UPDATE Ticks SET bid = 35, ask = 36 WHERE SurrogateId = ?
if (record.holdSurrogate(id)) {
	bid.setValue(35);
	ask.setValue(36);
	record.commitRow();
}
```

Deleting Rows

Record ***deleteRow(), deleteDirect() and deleteSurrogate()*** functions are used to delete a row in a database table. Deleted rows will be deleted when the transaction is completed.

Deleting current Row

deleteRow()

```
Database db;
db.connect("TickDB");
createTable();
Record record = db.getTable("Stocks").getRecord();
ColumnString symbol = record.getColumnString("Symbol");
ColumnDouble bid = record.getColumnDouble("Bid");
//insert
SurrogateId id = 1;
record.insertSurrogate(id);
symbol.setValue("MSFT");
bid.setValue(33);
record.commitRow();
//delete current row
record.deleteRow();
record.commitRow();
```

Deleting Row using DirectId

deleteDirect()

```
Database db;
db.connect("TickDB");
createTable();
Record record = db.getTable("Stocks").getRecord();
ColumnString symbol = record.getColumnString("Symbol");
ColumnDouble bid = record.getColumnDouble("Bid");
//insert
SurrogateId sid = 1;
record.insertSurrogate(sid);
DirectId id;
record.getDirectId(id);
symbol.setValue("MSFT");
bid.setValue(33);
record.commitRow();
//DELETE FROM Stocks WHERE DirectId = ?
record.deleteDirect(id);
record.commitRow();
```

Deleting Row using SurrogateId

deleteSurrogate()

```
Database db;
db.connect("TickDB");
createTable();
Record record = db.getTable("Stocks").getRecord();
ColumnString symbol = record.getColumnString("Symbol");
ColumnDouble bid = record.getColumnDouble("Bid");
//insert
SurrogateId id = 1;
record.insertSurrogate(id);
symbol.setValue("MSFT");
bid.setValue(33);
record.commitRow();
//DELETE FROM Stocks WHERE DirectId = ?
record.deleteSurrogate(id);
record.commitRow();
```

Metadata

The Direct C++ API provides ability to check for existence of a particular database Table or Column, or iterate through database metadata.

Checking for existence of a Table

```
Database db;
db.connect("TickDB");
bool exists = db.containsTable("Stocks");
```

Checking for existence of a Column

```
Database db;
db.connect("TickDB");
bool exists = db.containsColumn("Stocks", "Symbol");
//or
Table table = db.getTable("Stocks");
exists = table.containsColumn("Symbol");
```

Iterating through Tables and Columns

The **TableIterator** entity is used to iterate through all tables in a database.

The **ColumnIterator** entity is used to iterate through all columns in a table.

```
Database db;
db.connect("TickDB");
createTable();
TableIterator tables = db.getTables();
while (tables.next()) {
	cout << tables.getName() << endl;
	ColumnIterator columns = tables.getColumns();
	while (columns.next()) {
		cout << '\t' << columns.getName() << endl;
	}
}
```

Indices

Using column unique indices

```
Database database;
database.connect("IndexedDatabase");
//----------------------------------------
// 1) CREATE
//----------------------------------------
CreateTable createTable = database.createTable("IndexedTable");
createTable.addColumn("Column", ColumnType::Int16);
createTable.addColumn("IndexedColumn", ColumnType::Int16, IndexType::Unique);
createTable.execute();
//----------------------------------------
// 2) BIND
//----------------------------------------
Record tableRecord = database.getTableRecord("IndexedTable");
ColumnInt16 indexedColumn = tableRecord.getColumnInt16("IndexedColumn");
//----------------------------------------
// 3) INSERT
//----------------------------------------
indexedColumn.setValue(123);
tableRecord.insertSurrogate(1);
//----------------------------------------
// 4) SELECT
//----------------------------------------
indexedColumn.setValue(123);
tableRecord.selectByColumn(indexedColumn);
```

Using column non-unique indices

```
Database database;
database.connect("IndexedDatabase");
//----------------------------------------
// 1) CREATE
//----------------------------------------
CreateTable createTable = database.createTable("IndexedTable");
createTable.addColumn("Column", ColumnType::Int16);
createTable.addColumn("IndexedColumn", ColumnType::Int16, IndexType::NonUnique);
createTable.execute();
//----------------------------------------
// 2) BIND
//----------------------------------------
Record tableRecord = database.getTableRecord("IndexedTable");
ColumnInt16 indexedColumn = tableRecord.getColumnInt16("IndexedColumn");
//----------------------------------------
// 3) INSERT
//----------------------------------------
indexedColumn.setValue(123);
tableRecord.insertSurrogate(1);
tableRecord.insertSurrogate(2);
//----------------------------------------
// 4) SELECT
//----------------------------------------
indexedColumn.setValue(123);
tableRecord.resetByColumnIndex(indexedColumn);
tableRecord.nextRow();
tableRecord.nextRow();
```

Using composite unique indices

```
Database database;
database.connect("IndexedDatabase");
//----------------------------------------
// 1) CREATE
//----------------------------------------
CreateTable createTable = database.createTable("IndexedTable");
createTable.addColumn("IndexedColumnA", ColumnType::Int16);
createTable.addColumn("IndexedColumnB", ColumnType::Int16);
IndexMetadata createIndex = createTable.addIndex("Index", IndexType::Unique);
createIndex.addColumn("IndexedColumnA");
createIndex.addColumn("IndexedColumnB");
createTable.execute();
//----------------------------------------
// 2) BIND
//----------------------------------------
Record tableRecord = database.getTableRecord("IndexedTable");
ColumnInt16 indexedColumnA = tableRecord.getColumnInt16("IndexedColumnA");
ColumnInt16 indexedColumnB = tableRecord.getColumnInt16("IndexedColumnB");
//----------------------------------------
// 3) INSERT
//----------------------------------------
indexedColumnA.setValue(123);
indexedColumnB.setValue(321);
tableRecord.insertSurrogate(1);
//----------------------------------------
// 4) SELECT
//----------------------------------------
indexedColumnA.setValue(123);
indexedColumnB.setValue(321);
tableRecord.selectByIndex("Index");
```

Using composite non-unique indices

```
Database database;
database.connect("IndexedDatabase");
//----------------------------------------
// 1) CREATE
//----------------------------------------
CreateTable createTable = database.createTable("IndexedTable");
createTable.addColumn("IndexedColumnA", ColumnType::Int16);
createTable.addColumn("IndexedColumnB", ColumnType::Int16);
IndexMetadata createIndex = createTable.addIndex("Index", IndexType::NonUnique);
createIndex.addColumn("IndexedColumnA");
createIndex.addColumn("IndexedColumnB");
createTable.execute();
//----------------------------------------
// 2) BIND
//----------------------------------------
Record tableRecord = database.getTableRecord("IndexedTable");
ColumnInt16 indexedColumnA = tableRecord.getColumnInt16("IndexedColumnA");
ColumnInt16 indexedColumnB = tableRecord.getColumnInt16("IndexedColumnB");
//----------------------------------------
// 3) INSERT
//----------------------------------------
indexedColumnA.setValue(123);
indexedColumnB.setValue(321);
tableRecord.insertSurrogate(1);
tableRecord.insertSurrogate(2);
//----------------------------------------
// 4) SELECT
//----------------------------------------
indexedColumnA.setValue(123);
indexedColumnB.setValue(321);
tableRecord.resetByCompositeIndex("Index");
tableRecord.nextRow();
tableRecord.nextRow();
```

Persistent vs. non-persistent databases

Persistence is managed by the CDB server demon (`cdbServerDaemon.exe`).

A persistent database consists of

- An instance of the CDB server demon running as a Windows service.
- A collection of files storing the database data.

A non-persistent database only exists in memory while any CompleteDB application maintains an active connection to it.

The cdbConsole command line utility

Overview

The cdbConsole.exe command line utility allows creating, modifying and deleting CompleteDB databases in both persistent and non-persistent mode.

Installation

Extract content of bin to a directory of your choice. cdbConsole`.exe` and `cdbServerDaemon.exe` executables can run standalone and do not require Windows-style installation.

Starting cdbConsole

Open a command prompt window and change to the directory where cdbConsole is located.

```
C:\CompleteDB\bin\cdbConsole

Copyright (c) CompleteDB. All rights reserved.

Type "help" for help, type "exit" to quit cdbConsole.

> help

help | exit | desc [TableName] | disconnect | connect [DatabaseName]

{start|stop|install|uninstall} [DaemonName]

create table TableName(ColumnName [ColumnType], ...)

   [ColumnType] == {bool|int8|int16|int32|int64|float|double}

   [ColumnType] == {boolean|byte|short|int|long}
```

Creating a non-persistent database

```
> connect demo1

Done.

demo1>
```

Note:

- The prompt now shows the name of the active database.
- The database is ready to accept commands. The database only lives in memory. Once the last CompleteDB application has disconnected the database is destroyed.
- 'connect' without argument creates a database named 'default'.

Creating a persistent database

```
> install demo1


Install daemon: "completedb_demo1" ...

service path: C:\CompleteDB\bin\cdbServerDaemon.exe

Done.
```

Note: This command creates the associated Windows service.

Starting a persistent database

```
> start demo1

Start daemon: "completedb_demo1" ...

Daemon start pending...


Done.
```

Note: This command starts the associated Windows service and creates a subdirectory with the same name as the database, containing the associated database files.

Connecting to a persistent database

```
> connect demo1

Done.

demo1>
```

Note: The prompt now shows the name of the active database. The database is ready to accept commands.

Creating database tables

```
demo1> create table table1(col1 byte, col2 float)

Done
```

Note: This command creates a table called table1 with two columns (col1 with datatype byte, col2 with datatype float).

Listing database tables

```
demo1> desc


table1(col1,col2)


1 tables found.
```

Stopping a persistent database

```
demo1> stop demo1

Stop daemon: "completedb_demo1" ...

Done.
```

Note: This command stops the associated Windows service.

Dropping a persistent database

```
> uninstall demo1


Done.
```

Note:

- This command uninstalls the associated Windows service.
- To avoid accidental data loss, the files associated with the database remain. Recreating a database with the same name in the same directory will reuse the files.

cdbConsole command summary

Command	Non-persistent database	Persistent database
`install`	N/A	Creates Windows service.
`start`	N/A	Starts Windows service. Creates database files if files not present and starts persistent database.
`stop`	N/A	Stops Windows service. Stops persistence. A subsequent `start` restarts the database from the disk files.
`uninstall`	N/A	Stops Windows service. Keeps database files to prevent accidental data loss.
`connect`	Creates non-persistent database if it doesn't exist and connects to it.	Connects to persistent database.
`disconnect`	Disconnects from non-persistent database. Non-persistent database is destroyed when the last client application disconnects.	Disconnects from persistent database.

Note:

Trying to install/start a persistent database when a non-persistent database with the same name exists returns an error.