NitrosBase InMemory DB
C++ Application Program Interface
Version 2.2
Programmer's Guide

ID: NBIM.02.02.00.01

CONTENTS

NITROSBASE INMEMORY DB C++ CLIENT API

NitrosBase InMemory DB Client Application Program Interface (API) presents Program Interface for a database application which uses power of NitrosBase In-Memory database. This document describes C++ API. For short we'll call it "NitrosBase API" or "API" further.

This document covers 3 flavors of NitrosBase InMemory DB:

- Windows 32 bit,
- Windows 64 bit,
- Linux.

PREREQUISITES

When you are creating C++ application working with NitrosBase InMemory DB the first you should link NitrosBase.dll to your application. To do that, first you should arrange properly *nitrosbase.h*, *NitrosBase.lib* and *NitrosBase.dll* files.

Windows

1. Place nitrosbase.h and NitrosBase.lib files into
   a. Your solution folder or
   b. Any other folder where your program source code is located.
2. Place NitrosBase.dll into
   a. *Release* and *Debug* folders of your solution or
   b. Any other folder where your executable file is created when building the application.
   c. Another way is to put NitrosBase.dll file into *windows, windows/system32* or any other folder accessible via PATH system variable.

Linux

1. Place libnitrosbase_v2.so into
   a. /opt/lib directory or
   b. into any other directory accessible via LD_LIBRARY_PATH variable or
2. Place nitrosbase.h into
   a. /opt/include directory or
   b. into the same directory where the application is located
   c. into any directory which is specified in the compiler command line
3. The command line may look as follows:
   a. gcc -Wall -L/opt/lib prog.c -lnitrosbase_v2 -o prog or
   b. gcc -Wall –I<*path to include-files*> -L<*path to libraries*> prog.c -lctest -o prog

Then you must include *nitrosbase.h* header file into your application code.

```
#include "nitrosbase.h";
```

Now your environment is tuned up for working with NitrosBase API.

THE FIRST EXAMPLE

Here’s a simple program that uses the basic NitrosBase features. The example shows how to

- Connect to database
- Create a table
- Add records to a table
- Select records form the table
- Print the records selected

```
#include "stdafx.h"
#include "nitrosbase.h"

#define APPNO 001

struct PersonStruct {
    int id;
    char name[51];
    double weight;
};

int main(int argc, char* argv[])
{
    CQuery q;
    CNitrosBaseV2 * db;

    try{
        // Connect to database
        db = dbconnect();
        q.db = db;

        printf("App # %d NitrosBase version %s\n\n",
               APPNO , q.GetNitrosBaseVersion());

        // Create a table
        q.ExecuteSQL("CREATE TABLE PersonTable(Id int, Name varchar(50),
                     Weight float)");

        // Insert 3 records into the table
        q.ExecuteSQL("INSERT INTO PersonTable(Id, Name, Weight)
                     VALUES(1,'Gary',    61.7) ");
        q.ExecuteSQL("INSERT INTO PersonTable(Id, Name, Weight)
                     VALUES(2,'Scott',  122.3) ");
        q.ExecuteSQL("INSERT INTO PersonTable(Id, Name, Weight)
                     VALUES(3,'Brian',  165.3) ");

        // Find a record where id = 3
        q.ExecuteSQL("SELECT * FROM PersonTable WHERE Id = 3");

        // Print the found records
        while(q.Next()) {
            PersonStruct * Person = (PersonStruct *)q.GetRec();
            cout << Person->id << "   " << Person->name << "   "
                  << Person->weight << "\n";
        }
        db->close();

        printf("Example Application %d successfully completed\n\n", APPNO);

    }catch(const char * error){
        printf("EROOR: %s", error);
    }
}
```

The following sections reveal the details of the code above; besides, they also introduce and explain more complicated examples.

Connecting to database

To open existing database or create a new one use *dbconnect* function. On success the function returns database handle. On error Exception is generated.

If you work with In-Memory only (without saving data to disk) data call the function without parameters:

```
CNitrosBaseV2 * db = dbconnect();
```

But if you need read and save data on the hard drive use function with parameters:

```
CNitrosBaseV2 * db = dbconnect(path, dbname, mode, save_flag);
```

The function *dbconnect* works with the following parameters:

Path path to database files folder. It can be specified as either **absolute** path or the path **relative** to the application directory. If path = NULL or path = “” then database is created in application directory. If the directory specified by path does not exist, then error is generated. Examples:

```
path = "c:\\NitrosBase\\dbfiles"; // absolute path
path = "dbfiles";                 // relative path
path = "..\\..\\dbfiles";         // relative path
path = "";                        // relative path
```

dbname database file name.

mode database mode flag. It can be set to the following constants:

CREATE__NEW	-	Create new database. If the database with the same name already exists, then error is generated.
CREATE__ALWAYS	-	Create new database. If the database with the same name already exists, then it is overridden by new database.
OPEN__EXISTING	-	Open existing database. If there is no file specified by *path* and *dbname* parameters, then error is generated.
OPEN__ALWAYS	-	Open existing database if exists or create new database otherwise.

save_flag database save flag. It can be set to the following constants:

SAVE__DB	-	Save the results on disk.
DONT__SAVE	-	Don’t save the results.

The examples of dbconnect function calls are here:

```
// Create temporary database don’t save the results
db = dbconnect();

// Create new database in application directory. If the database
// already exists - generate an error. Don't save the result.
db = dbconnect(NULL, "SampleDatabase", CREATE__NEW, DONT__SAVE);

// Create new database in application directory. If the database
// already exists overwrite it. Don't save the result.
db = dbconnect("", "SampleDatabase", CREATE__ALWAYS, DONT__SAVE);

// Open database in application's dbfiles directory.
// If the database doesn't exists create a new one. Save the result.
db = dbconnect("dbfiles", "SampleDatabase", OPEN__ALWAYS, SAVE__DB);

// Open database in D:\Data directory.
// If the database doesn't exists generate an error. Save the result.
db = dbconnect("D:\\Data", "SampleDatabase", OPEN__EXISTING, SAVE__DB);
```

All the interaction to database is performed through *CQuery* class. The typical NitrosBase application beginning part would look as follows:

```
#include "nitrosbase.h";
...
int main(int argc, char* argv[])
{
    CQuery q;
    CNitrosBaseV2 * db;
    db = dbconnect(...);
    q.db = db;
    ...
```

Now your application is ready for work with NitrosBase InMemory DB.

CLOSING THE CONNECTION

After all the work with the database is finished you need to close database connection. It could be done calling *close* function. This function frees all the buffers, releases memory and closes all NitrosBase InMemory DB files. The typical NitrosBase InMemory DB application final part would look as follows:

```
    ...
    Db->close();
}
```

Now your application is disconnected from NitrosBase.

Running the Queries

After the connection is established, you usually create tables and indexes, write and read data. All those operations can be easily done via SQL queries. There are two forms of SQL query execution in NitrosBase InMemory DB API:

- Immediate call, using *ExecuteSQL* method;
- Prepared call, using *Prepare* and *Execute* methods

Immediate Call

ExecuteSQL method specified in CQuery class performs immediate query execution. There are the following ways of calling the method:

1. ExecuteSQL(char *query [, *parameters*])
2. ExecuteSQL(const char *query [, *parameters*])

For the first way of ExecuteSQL function call, having 1st parameter type **char ***, NitrosBase parses SQL query and executes it every call. For example,

```
CNitrosBaseV2* db_handle = dbconnect();
CQuery q;
q.db = db_handle;
char* str = "CREATE TABLE Example1(id int, value float)";
q.ExecuteSQL(str); // the query is parsed and executed
str = "INSERT INTO Example1 (id, value) VALUES (2, 3.1415)"
q.ExecuteSQL(str); // the query is parsed and executed
str = "SELECT * FROM Example1 WHERE id=2"
q.ExecuteSQL(str); // the query is parsed and executed
```

When dealing with the second way of ExecuteSQL function call, having 1st parameter type **const char ***, NitrosBase acts as follows:

- For the first call the query is parsed and executed
- For each consequent query
  - NitrosBase Compares pointer to query with pointer to CQuery.Query.
  - If pointers are equal, then NitrosBase assumes that the query is not changed. The query is not parsed, but executed immediately
  - If pointers are not equal, then the query is parsed and executed.

This mechanism has big performance advantage when dealing with parameterized queries (see “Parameterized queries” section). For example:

```
CNitrosBaseV2* db_handle = dbconnect();
CQuery q;
q.db = db_handle;
q.ExecuteSQL("CREATE TABLE Example2(id int, value float)");
const char* SQLstr = "INSERT INTO Example2(id, value) VALUES(:p1,:p2)";
q.ExecuteSQL(SQLstr,1,100);  // SQLstr is parsed and executed
q.ExecuteSQL(SQLstr,2,3223); // SQLstr is executed quickly
q.ExecuteSQL(SQLstr,3,21);   // SQLstr is executed quickly
```

The same time, it doesn’t bring inconvenience when user deals with changing query, say, in the example below ExecuteSQL calls are performed against changing pointer, so the example is correct and would run perfectly.

```
CNitrosBaseV2* db_handle = dbconnect();
CQuery q;
q.db = db_handle;
q.ExecuteSQL("CREATE TABLE Example5(id int, value float)");
q.ExecuteSQL("INSERT INTO Example5(id, value) VALUES(:p1,:p2)", 10, 20);
```

```
q.ExecuteSQL("SELECT * FROM Example5 WHERE id=:p1 AND value=:p2", 10, 20);
```

Note: If you use *string* class to store your queries then be careful using *string::c_str()* method. You should remember that *c_str()* returns const char *. It means that the object declared as *string* can be changed, but the pointer returned by *c_str()* method wouldn't change. You should perform manual cast in this case. See the following example:

```
CNitrosBaseV2* db_handle = dbconnect();
CQuery q;
q.db = db_handle;
string query = "CREATE TABLE Example2(id int, value float)";
q.ExecuteSQL(query.c_str()); // The query is parsed and executed
query = "INSERT INTO Example2(id,value)VALUES(1,2)";
q.ExecuteSQL((char*)query.c_str());  /* we need (char *) here, otherwise
                                        "CREATE TABLE..." query is called */
```

Prepared call

For the detailed description of parameterized queries see ("Parameterized queries" section). Prepared call implies splitting SQL query execution process into 2 steps.

1. Preparing the query via *Prepare* method and then
2. Executing the query via *Execute* method.

The simplest example is as follows:

```
q.Prepare("SELECT * FROM TblNames");
q.Execute();
```

Prepared call also allows parameters passing, for example:

```
q.Prepare("SELECT * FROM TblNames WHERE Id = @par");
q.Execute(i);
```

Prepared call is much more flexible way of running SQL queries. But it can't be seen from the example above. The "Parameterized queries" section allows seeing the power of prepared call.

Parameterized queries

Though immediate call enables parameterized queries, but further we will use mainly prepared call as more flexible and expressive. The simplest example of Parameterized query using prepared call is here:

```
...
q.Prepare("SELECT * FROM PersonTable WHERE Id = @param");
for(int i=1; i<100; i++) {
     q.Execute(i);
     while(q.Next()) {
          ... // do something
}
```

In the example above the Execute function runs the query arranged by the Prepare function. The parameters are specified in the query using standard SQL syntax, which implies "@" as a prefix for parameter name. The alternative notation is used in NitrosBase is ":"as a prefix for parameter name. You can use both "@" and ":" equally in program code. NitrosBase InMemory DB allows any number of parameters of any allowed type (see "NitrosBase InMemory DB Field types" section for the allowed types).

Positional and keyword parameter transmission

There are 2 ways of parameter transmission: positional and keyword. Positional parameter transmission is the transmission of the parameters in the same order as they are listed in SQL statement. For example,

```
q.Prepare("INSERT INTO PersonTable(Id, Name, Weight) \
                VALUES( @p1, @p2, @p3) ");
for (i = 1; i < 5; i++)
{
    sprintf(name,"%s%02d", "NamePos", i);
    q.Execute(i, name, i*i);
}
```

In the example above p1 gets value i, p2 gets value i*10 and so on. Execute function can handle up to 8 parameters.

Positional parameter transmission is the fastest way of performing parameterized queries.

More suitable is keyword parameter transmission. An example is shown here:

```
q.Prepare("INSERT INTO PersonTable(Id, Name, Weight) \
                VALUES( @p1, @p2, @p3) ");
for (int i = 1; i < 4; i++)
{
    q.par["p1"] = i;
    q.par["p3"] = i*i;
    sprintf(name,"%s%d", "Name", i);
    q.par["p2"] = name;
    q.Execute();
}
```

In the example above *par* method explicitly assigns values to parameters.

Keyword parameter transmission is considerably slower than positional parameter transmission.

If you need more parameters

The easy methods of running parameterized queries, like ExecuteSQL(p1, p2, ...) or Execute(p1, p2, ...) support up to 8 parameters. If you need more parameters use keyword parameter transmission described above. That method is not limited in number of parameters.

Navigating through the query result

Navigating through the query result using CQuery class reminds the one in .NET using SQLReader class. You can only move forward to the next record. Moving is performed by *Next* method. The method reads the next record and returns TRUE on successful read. *Next* method returns FALSE when no more records left in the record set.

Typical scenario looks as follows:

```
q.Prepare("SELECT * FROM Table1 WHERE Field0 = :namedparam");
for(int i=0; i<1000; i++)
{
q.Execute(i);
while(q.Next())
    // operations with the record
}
```

Reading record fields

Next function moves cursor to next record in the selection set and copies the record into the buffer. When *Next* function is called first time it positions cursor to the 1st record. You can read the record field values from the buffer. It can be done the following ways.

Method 1 – the fastest way

Get the pointer to the buffer. Declare a C++ structure mapping the query record structure and declare the pointer to the buffer as pointer to the structure. Then you can access immediately to record fields as the structure variables.

Example:

```
struct PersonStruct {
    int id;
    char name[51];
    double weight;
};
...
q.Prepare("SELECT * FROM PersonTable WHERE Id > 1");
q.Execute();
while(q.Next()) {
    PersonStruct * Person = (PersonStruct *)q.GetRec();
    cout << "Pointer  : " << Person->id << " " << Person->name << " " <<
              Person->weight << "\n";
}
```

Or the same if you prefer references:

```
    ...
    PersonStruct & Person = *(PersonStruct *)q.GetRec();
    cout << "Reference: " << Person.id << " " << Person.name << " " <<
            Person.weight << "\n";
    ...
```

This way is the fastest but requires thorough handling. Modifying the query you have to modify the structure declaration neatly.

Method 2

You can access a field by its number and avoid structure declaration. The example below illustrates the feature.

```
q.Prepare("SELECT * FROM PersonTable WHERE Id > 1");
q.Execute();
while(q.Next()) {
    printf("Numbers   : %d %s %3.1f\n",(int)q[0],(char*)q[1].p,(double)q[2]);
}
```

Access to the field using its number allows you avoiding structure declaration, but as the Method 1 requires accurate ordering the fields. It may occur boring especially for long records.

Method 3 – the slowest but most handy way

You can use the field name for accessing its content. This is the most comfortable way. The example below illustrates the feature.

```
q.Prepare("SELECT * FROM PersonTable WHERE Id > 1");
q.Execute();
while(q.Next()) {
    printf("Names    : %d %s %3.1f\n", (int)q["Id"], (char*)q["Name"].p,
            (double)q["Weight"]);
}
```

Modifying the records (Insert, Update, Delete)

There are the following ways to modify the records.

Performing SQL statement

This is most evident and widely used way, for example:

```
q.Prepare("INSERT INTO PersonTable(Id, Name, BirthDate, Shares, Weight) \
          VALUES(:p1,'Dominic', '09.03.2010 00:00:00', :p2, :p3) ");
for(int i = 10; i < 20; i++)
    q.Execute(i, (int64_t)i*10, (double)i*100);

q.ExecuteSQL("UPDATE PersonTable SET Shares = 0 WHERE Id = 10");
q.ExecuteSQL("DELETE FROM PersonTable WHERE Id > 9");
```

Calling CQuery methods

This is NitrosBase InMemory DB proprietary approach, which is much faster than using SQL. You can call CQuery methods: Insert, Update, Delete.

Inserting via CQuery

There are 2 ways of inserting records via CQuery. The first is simplest and just inserts a manually initialized record. For Example,

```
struct PersonStruct {
    int id;
    char name[51];
    double weight;
};
PersonStruct *rec;

q.Prepare("Select * from PersonTable");
for(i=1; i<5; i++)
{
    rec = (PersonStruct *)q.GetRec();
    rec->id = i;
    sprintf(name,"%s%02d", "Fresh", i);
    strcpy(rec->name, name);
    rec->weight = i*100;
    q.Insert(rec);
}
```

In the example above q.Prepare("Select * from PersonTable") gets the record structure. Other lines initialize ALL THE FIELDS of the record. It is important for this method to have all the fields initialized before you call q.Insert.

The 2nd way is used when you consider creating a new record as a copy of another record with some modifications. For example:

```
struct PersonStruct {
    int id;
    char name[51];
    double weight;
};
PersonStruct *rec;

q.Prepare("Select * from PersonTable");
q.Execute();
while(q.Next())
{
    rec = (PersonStruct *)q.GetRec();
    rec->id += 10;
    q.Insert(rec);
}
```

As you can see from this example, the ways slightly differ:

- In the second approach you have to get the copy of record into the buffer, calling all the functions: Prepare, Execute, Next; while in the first you need only Prepare.
- In the first way you have to initialize all the fields, while in the second you only need some fields (Id in this example) to have initialized. Other values remain the same as in the record they are copied from.

Updating via CQuery

Updating the records using Cquery is similar to the 2nd way of inserting a record. For example:

```
q.Prepare("Select * from PersonTable where Id < 4");
q.Execute();
while(q.Next())
{
    rec = (PersonStruct *)q.GetRec();
    rec->id += 20;
    q.Update();
}
```

Some comments to the example above:

- The Select SQL query specifies the records for updating. All the records having Id less than 4 are to be updated in this case.
- Next() positions cursor on the next record to be Updated
- Id field of the record is increased by 20 in the buffer
- Update() transfers modifications to database record.

Deleting via CQuery

Deleting the records using Cquery can be easily understood from the example:

```
q.Prepare("Select * from PersonTable where Id > 14");
q.Execute();
while(q.Next())
{
    q.Delete();
}
```

Some comments to the example above:

- The Select SQL query specifies the records for deletion, Id less than 14 in this particular case
- Next() positions cursor on the next record to be deleted
- Delete() deletes the record.

NitrosBase InMemory DB Field types

This moment NitrosBase supports the following field types:

NitrosBase type	Comments
INT	32 bit Integer value in range [-2 147 483 648..2 147 483 647]
BIGINT	64 bit Integer value in range [-9 223 372 036 854 775 808.. 9 223 372 036 854 775 807]
FLOAT	Floating point values in range [-1.79769*10^{308}..+1.79769*10^{308}]
DATETIME	Date and time (see “ Datetime type” chapter for details)
VARCHAR()	String. The length of the string should be in range: [1..8000]

C++ and NitrosBase InMemory DB type mapping

When developing C++ NitrosBase applications the following data type mapping should be taken into account:

NitrosBase type	C++ type
INT	int
BIGINT	int64_t
FLOAT	double
DATETIME	CDateTime
VARCHAR(LEN)	Char[LEN+1]

Datetime type

NitrosBase InMemory DB supports the following *datetime* format in SQL expressions: 'mm.dd.yyyy hh:mm:ss'.

For example,

```
q.ExecuteSQL("UPDATE Table1 SET EventTime = '02.03.2010 11:22:00'
             WHERE EventID = 893");
```

Internal format of *datetime* data is described by the following structure:

```
struct CDateTime
{
   unsigned short tm_msec;     // millisec after the minute - [0,59999]
   unsigned char  tm_min;      // minutes after the hour - [0,59]
   unsigned char  tm_hour;     //hours since midnight - [0,23]
   unsigned char  tm_mday;     // day of the month - [1,31]
   unsigned char  tm_mon;      // months since January - [1,12]
   unsigned short tm_year;     // year
   ...
};
```

Example of adding *datetime* data into *NitrosBase* table using *CDateTime* structure:

```
    q.ExecuteSQL(
        "CREATE TABLE PersonTable(Id        INT, \
                                  Name      varchar(50), \
                                  BirthDate datetime, \
                                  Shares    bigint, \
                                  Weight    float)");
    q.ExecuteSQL("INSERT INTO PersonTable(Id, Name, BirthDate, \
                 Shares, Weight) VALUES(1, 'Patrick', \
                 '01.03.2010 00:00:00', 333222111000, 28.0) ");
    q.ExecuteSQL("INSERT INTO PersonTable(Id, Name, BirthDate, \
                 Shares, Weight) VALUES(2, 'Scott', \
                 '02.03.2010 00:00:00', 333222111000, 55.5) ");
    q.Prepare("SELECT * FROM PersonTable \
               WHERE Id = @namedparam and Weight > @namedparam1");
    for(int i=1; i<100; i++) {
        if (i%2 == 0) continue;
        q.Execute(i, 70);
        while(q.Next()) {
            PersonStruct * Person = (PersonStruct *)q.GetRec();
            CDateTime Birth = Person->birthdate;
            cout << Birth.tm_year << "." << (int)Birth.tm_mday << "." <<
                  (int)Birth.tm_mon << "\n";
            cout << Person->id << "    " << Person->name << "    " <<
                  Person->shares << "    " << Person->weight << "\n";
        }
    }
```

Or the same could be done without declaring Birth structure:

```
cout << (int)Person->birthdate.tm_mday << "." <<
        (int)Person->birthdate.tm_mon << "." <<
        Person->birthdate.tm_year << "\n";
```

NitrosBase InMemory DB Indexes

NitrosBase supports the following index types:

Notation	Description
TREE	**Tree Index.** It is used for any kind of data filtering: “<”, “>”, “=”, “=<”, “>=”. Tree index is available for unique or non-unique fields.
HASH	**Hash Index.** Hash index is used for high performance search by “=”. It works effectively for unique fields.
MD_HASH	**Multidimensional Hash Index.** Такой индекс позволяет очень быстро выполнять запросы на поиск по равенству, в случае, если несколько полей в запросе связаны логическим условием AND. Поля в таком индексе могут быть разного типа.

Note: Hash index is created automatically for a unique field. Tree index for a unique field is not created automatically and should be created using “CREATE INDEX” query.

Most of the indexes supported by this version of NitrosBase InMemory DB are one field indexes. The exclusion presents Multidimensional Hash Index.

Tree Index

Tree Index is the most universal kind of index. It can be used for any kind of selection. It is useful for queries using "<", ">", "=", "=<", ">=" comparison operations. Tree index is available for unique or non-unique fields.

The syntax of the query creating Tree Index is as follows:

```
CREATE INDEX <index_name> ON <table_name> (<field_name>) WITH TREE
```

Tree Index is "by default" index, so you can simply type

```
CREATE INDEX <index_name> ON <table_name> (<field_name>)
```

Note: before version 2.0 the Tree Index syntax was as follows: CREATE INDEX <index_name> ON <table_name> (<field_name>) WITH 1. This obsolete syntax is not encouraged but still supported for compatibility.

Tree index can be implicitly used for ORDER BY clause optimization. Namely, if you know that some field will be used in ORDER BY clause in the query you are preparing, it would be very useful creating tree index on that field in advance.

```
q.ExecuteSQL("CREATE INDEX Idx_Weight ON PersonTable (Weight) WITH TREE ");
...
q.ExecuteSQL("SELECT Id, Name, Weight FROM PersonTable WHERE Weight > 70 "
             "ORDER BY Weight");
```

In the example above you are going to order the query result by weight field. Creating Tree Index in this case is very helpful in terms on performance.

Indexes for unique fields

Hash index is used for high performance search by "=". Though you can use either Tree or Hash index when selecting records by "=", Hash index performs considerably better. The only requirement – the field used in Hash Index should be unique.

If a field is declared as unique (in CREATE TABLE clause) then hash index for that field is created automatically. Besides, you can manually create a tree index if you are going to make interval search in unique fields. The following example shows the usage:

```
// Owner is unique field in CarTable

q.ExecuteSQL("CREATE INDEX Idx_Owner on CarTable(Owner) WITH TREE");
...
q.ExecuteSQL("SELECT * FROM CarTable WHERE Owner >= 'John Smith'");
```

Besides, you can manually create a tree index for this field.

Note: If you know that some field of the table is filled in by unique values (even not being declared as UNIQUE in CREATE TABLE query) you can create hash index for that field manually to benefit from its uniqueness. We don't recommend that method though. Always the better way to handle unique fields is to declare them as UNIQUE when creating the table.

Indexes for non-unique fields

Hash Index doesn't work for non-unique field. Both tree index and hash index should be combined if you want to speed up the search by "=" for a non-unique field. NitrosBase implements that kind of combination the following way:

```
// The following code creates only tree index for Idx_Weight field
q.ExecuteSQL("CREATE INDEX Idx_Weight ON PersonTable (Weight) WITH TREE ");
// The following code creates both tree and hash indexes for Idx_Weight field
q.ExecuteSQL("CREATE INDEX Idx_Weight ON PersonTable (Weight) WITH HASH ");
```

i.e. whenever you create a hash index for a non-unique field tree index is created for that field automatically.

Note: HASH_AND_TREE index used before version 2.0 is obsolete but is still supported for compatibility. That index can be specified using one of the following forms:
q.ExecuteSQL("CREATE INDEX I_Doors on CarTable(Doors) WITH HASH_AND_TREE ");
q.ExecuteSQL("CREATE INDEX I_Doors on CarTable(Doors) WITH 3");

Multidimensional Hash Index

Multidimensional Hash Index (hash index by several fields) allows you very quick search by "=" when the conditions are tied by logic operation AND. It supports various field types. And the types are not obliged to be unique.

The syntax of the query creating Multidimensional Hash Index is as follows:

```
CREATE INDEX <index_name> ON <table_name> (<field_name_1>, ... , <field_name_N>) WITH MD_HASH
                [SORT <sort_field_name> ASC | DESC]
```

There are several rules for using Multidimensional Hash Index:

- The field list should end by special field having type int. That field should be created along the table and it shouldn't be used for anything else that serving the index.
- The order of the fields in CREATE INDEX clause should be kept in the following it SELECT clause.

The following example illustrates the details. Let's create a table

```
q.ExecuteSQL("CREATE TABLE CarTable (Id int \
                                    , Model VARCHAR(50) \
                                    , Make VARCHAR(50) \
                                    , Year int \
                                    , Doors int \
                                    , Cylinders int \
                                    , Weight float \"
                                    , FieldMF int \
                                    , FieldMF2 int)");
```

Suppose we want to select all the Years of 3-door Toyota Corolla. Then we design the following API calls:

```
q.ExecuteSQL("CREATE INDEX MDIndex01 ON CarTable \
                  (Make, Model, Doors, FieldMF) WITH MD_HASH \
                  SORT Model ASC ");

q.ExecuteSQL("SELECT Year FROM CarTable \
                  WHERE Make = 'TOYOTA' \
                  AND Model = 'COROLLA' \
                  AND Doors = 3");
```

Note, that

- the order of field names in CREATE INDEX query and SELECT query is the same and it differs from the order in CREATE TABLE.
- The FieldMF field is created by CREATE TABLE query, but it is not used anywhere else than the CREATE INDEX query field list.

Let's see another example. Suppose we select all the models of 4-door Toyota, having more than 4 cylinders. Then the CREATE INDEX and SELECT queries would look as follows:

```
q.ExecuteSQL("CREATE INDEX MDIndex02 ON CarTable (Make, Doors, FieldMF2) WITH
MD_HASH SORT Model ASC ");

q.ExecuteSQL("SELECT Model FROM CarTable WHERE Make = 'TOYOTA' AND Doors = 4
AND Cylinders > 4 ");
```

Here you can notice that the field list in SELECT query is wider than the one in CREATE INDEX one. It includes Make, Doors and Cylinders, whereas indexed fields include Make and Doors. And more, Cylinders > 4 constrain seemingly breaks the rules. The answer is: as soon as Make = … AND Doors = … are presented they will be searched for using MDIndex02 index and Cylinders > … constrain will be searched separately. But all the query correct and will be performed correctly.

CQuery class methods and variables

	Description
Db	Pointer to database class returned by dbconnect. Attaches CQuery to database.
Delete	Deletes current record
Execute	Runs query prepared by Prepare method. Can handle up to 8 parameters.
ExecuteSQL	Performs immediate query run. Besides a query can handle up to 8 parameters.
GetId	Returns identifier for current record. Usually is called after Next method
GetNewRecId	Returns identifier for new record. Usually is called after Insert method
GetNitrosBaseVersion	Returns NitrosBase InMemory DB version.
GetRec	Returns pointer to record copy received by query. Usually called after Next method
Insert	Inserts current record
Next	Sets cursor to 1st record when called 1st time or to the next record when called repeatedly. Used after Select query to navigate through the query result.
Par	Allows access to query parameters by their numbers or names, using operator []
Prepare	Prepares query for repeated query runs. Usually is used for parameterized queries
Update	Updates current record

ERROR HANDLING

NitrosBase error handling perfectly supports powerful standard C++ exception handling mechanism. Normally all you have to do is enclosing your code into TRY/CATCH blocks.

```
...
try{
    ...
    q.ExecuteSQL("CREATE TABLE Tbl(Id int, Name varchar(50))");
    q.ExecuteSQL("INSERT INTO Tbl(Id, Name) VALUES(1, 'Patrick') ");
    q.ExecuteSQL("SELECT * FROM Tbl");
    ...

}catch(const char * error){
    printf("EROOR: %s", error);
}
...
```

If there are no errors in the code within a TRY block, control passes to the statement immediately after the CATCH block. Otherwise, the control is passes to the first statement in the associated CATCH block; error message is generated and saved in *error* variable.

CURRENT STATUS OF NITROSBASE

LIMITATIONS OF CURRENT VERSION

- This is the 32-bit version of NitrosBase for Microsoft Windows OS family.
  Call NitrosBase for 64 bit and for Linux versions.
- This is embedded version of NitrosBase. It is implemented as dynamically loaded library.
  Call NitrosBase for NitrosBase Server.
- There is no built-in backup/restore features.
  Call NitrosBase if you need that feature.
- No administrative shell.
  Call NitrosBase if you need one.
- There is no transaction support (planned feature for future versions).

LIMITATIONS OF NITROSBASE SQL

This version of NitrosBase SQL supports only the following SQL statements:

- SELECT
- INSERT, UPDATE, DELETE
- CREATE TABLE
- CREATE INDEX
- DROP TABLE
- DROP INDEX

This version of NitrosBase SQL does not support:

- ALTER statement (is planned for version 3).
- Aggregation (is planned for version 3)
- Functions (partial implementation is planned for version 3)
- IN operator (partial implementation is planned for version 3 or 4)
- Nested queries (partial implementation is planned for version 4)
- DISTINCT (is planned for version 4)
- Stored procedures
- CONSTRAIN
- OUTER JOIN
- SELECT TO …
- Old syntax of JOIN queries

OTHER LIMITATIONS

Table names and field names are case sensitive, while SQL keywords and NitrosBase data types are case insensitive. This is done for performance reason. The objects like tables, fields, indexes, etc. are found in hash tables while parsing. It can be much faster when their names are put into hash table as is.

The record size is limited by 4 MB, including all strings, dates, numbers, etc.