NitrosBase InMemory DB
Программный интерфейс C++
Версия 2.2
Руководство разработчика

ID: NBIM.02.02.00.01

Оглавление

Программный интерфейс NitrosBase InMemory DB 3
Необходимые условия 3
Первый пример 4
Подключение к базе данных 5
Закрытие соединения 6
Выполнение запросов 7
Непосредственный вызов 7
Подготовленный вызов 8
Параметризованные запросы 9
Позиционная и ключевая передача параметров 9
Если Вам нужно больше параметров 10
Навигация по результату запроса 10
Чтение полей записи 10
Метод 1 – наиболее быстрый 10
Метод 2 11
Метод 3 – самый медленный, но самый удобный 11
Модификация записей (Insert, Update, Delete) 11
Выполнение SQL запроса 11
Вызов методов CQuery 12
Типы данных NitrosBase InMemory DB 13
Сопоставление типов данных C++ и NitrosBase InMemory DB 14
Datetime 14
Индексы NitrosBase InMemory DB 15
Tree Index 15
Индексы для уникальных полей 16
Индексы для неуникальных полей 16
Multidimensional Hash Index 17
Переменнные и методы класса CQuery 18
Обработка ошибок 19
Текущее состояние nitrosbase 20
Ограничения текущей версии 20
Ограничения NitrosBase SQL 20
Другие ограничения 20

Программный интерфейс NitrosBase InMemory DB

Данный интерфейс представляет собой инструментарий языка C++ для приложений, использующих возможности базы данных NitrosBase In-Memory DB. В рамках данного документа для краткости будем использовать термины «Интерфейс NitrosBase» или “API”.

В этом документе описываются 3 разновидности продукта NitrosBase InMemory DB:

- NitrosBase InMemory DB для Windows 32 bit,
- NitrosBase InMemory DB для Windows 64 bit,
- NitrosBase InMemory DB для Linux.

Необходимые условия

При создании приложения на C ++, работающего с NitrosBase InMemory DB, в первую очередь необходимо связать NitrosBase.dll с приложением. Для этого необходимо правильно оформить файлы nitrosbase.h, NitrosBase.lib и NitrosBase.dll.

Windows (MS Visual Studio)

1. Разместите файлы nitrosbase.h и NitrosBase.lib
    a. В папке Вашего решения или
    b. В любой папке, где расположен код Вашей программы
2. Разместите файл NitrosBase.dll
    a. *В папках Release* и *Debug* Вашего решения или
    b. Любой другой папке, где расположен Ваш исполняемый файл.
    c. Другой путь – расположите файл NitrosBase.dll в папке *windows/system32* или любой папке, доступной через системную переменную PATH.

Linux

1. Разместите файл libnitrosbase_v2.so в директории
    a. /opt/lib directory или
    b. Любой другой директории доступной через переменную LD_LIBRARY_PATH
2. Разместите файл nitrosbase.h в директории
    a. /opt/include или
    b. В той же директории, где расположено Ваше приложение или
    c. В любой директории, которая указана в командной строке компьютера.
3. Командная строка может выглядеть так:
    a. gcc -Wall -L/opt/lib prog.c -lnitrosbase_v2 -o prog или
    b. gcc -Wall –I*<path to include-files>* -L*<path to libraries>* prog.c -lctest -o prog

После чего надо включить заголовочный файл *nitrosbase.h* в код приложения:

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

Теперь Ваша среда настроена для работы с NitrosBase API.

Первый пример

Вот простая программа, которая использует основные функции NitrosBase. Пример показывает, как

- Подключиться к базе данных
- Создать таблицу
- Добавить записи в таблицу
- Выбрать записи из таблицы
- Распечатать выбранные записи

```
#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);
```

```
    }
}
```

В следующих разделах подробно рассказывается о приведенном выше коде; кроме того, в них также объясняются и более сложные примеры.

Подключение к базе данных

Чтобы открыть существующую базу данных или создать новую, используйте функцию dbconnect. В случае успеха функция возвращает дескриптор базы данных. При ошибке генерируется исключение.

Если вы работаете только в режиме In-Memory (без сохранения данных на диск), вызовите функцию без параметров:

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

Но если вам нужно прочитать и сохранить данные на жестком диске, воспользуйтесь функцией с параметрами:

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

Функция dbconnect работает со следующими параметрами:

Path путь к папке с файлами базы данных. Его можно указать либо как абсолютный путь, либо как путь относительно каталога приложения. Если path = NULL или path = “”, то база данных создается в каталоге приложения. Если каталог, указанный в пути, не существует, генерируется ошибка. Примеры:

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

dbname имя файла базы данных.

mode флаг режима базы данных. Его можно установить в следующие значения:

- CREATE__NEW - Создать новую базу данных. Если база данных с таким именем уже существует, генерируется ошибка.
- CREATE__ALWAYS - Создать новую базу данных. Если база данных с таким именем уже существует, она заменяется новой базой данных.
- OPEN__EXISTING - Открыть существующую базу данных. Если в параметрах path и dbname не указан файл, то генерируется ошибка.
- OPEN__ALWAYS - Откройте существующую базу данных, если она существует, или создайте новую базу данных в противном случае.

save_flag флаг сохранения базы данных. Его можно установить в следующие значения:

- SAVE__DB - Сохранить результаты на диск.

DONT__SAVE - Не сохранять результаты на диск.

Примеры вызовов функций dbconnect:

```
// 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);
```

Все взаимодействие с базой данных осуществляется через класс CQuery. Типичная начальная часть приложения NitrosBase будет выглядеть следующим образом:

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

Теперь ваше приложение готово к работе с Nitro Base InMemory DB.

Закрытие соединения

После завершения работы необходимо закрыть соединение с базой данных. Это можно сделать, вызвав функцию закрытия. Эта функция освобождает все буферы, освобождает память и закрывает все файлы NitrosBase InMemory DB. Типичная финальная часть приложения NitrosBase InMemory DB будет выглядеть следующим образом:

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

Теперь ваше приложение отключено от NitrosBase.

Выполнение запросов

После установления соединения обычно создаются таблицы и индексы, записываются или читаются данные. Все эти операции можно легко выполнить с помощью SQL-запросов. В NitrosBase InMemory DB API есть две формы выполнения SQL-запросов:

- Непосредственный вызов с использованием ExecuteSQL;
- Подготовленный вызов с использованием Prepare и Execute.

Непосредственный вызов

Метод ExecuteSQL, указанный в классе CQuery, выполняет немедленное выполнение запроса. Вызвать метод можно следующими способами:

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

Для первого способа вызова функции ExecuteSQL, имеющего 1-й параметр типа char *, NitrosBase анализирует SQL-запрос и выполняет его при каждом вызове. Например,

```
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
```

При работе со вторым способом вызова функции ExecuteSQL, имеющим 1-й параметр типа const char *, NitrosBase действует следующим образом:

- Для первого вызова запрос анализируется и выполняется.
- Для каждого последующего вызова:
  - NitrosBase Сравнивает указатель на запрос с указателем на CQuery.Query.
  - Если указатели равны, NitrosBase предполагает, что запрос не изменился. Запрос не анализируется, выполняется немедленно
  - Если указатели не равны, запрос анализируется и выполняется.

Этот механизм имеет большое преимущество в производительности при работе с параметризованными запросами (см. Раздел «Параметризованные запросы»). Например:

```
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
```

В то же время это не доставляет неудобств, когда пользователь имеет дело с изменением запроса, скажем, в приведенном ниже примере вызовы ExecuteSQL выполняются при изменяющемся указателе, поэтому пример будет работать правильно.

```
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);
```

Примечание. Если для хранения запросов вы используете строковый класс, будьте осторожны при использовании метода *string::c_str()*. вы должны помнить, что c_str () возвращает const char *. Это означает, что объект, объявленный как строка, может быть изменен, но указатель, возвращаемый методом c_str (), не изменится. В этом случае следует выполнить cast вручную. См. Следующий пример:

```
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 */
```

Подготовленный вызов

Для подробного описания параметризованных запросов см. Раздел «Параметризованные запросы». Подготовленный вызов подразумевает разбиение процесса выполнения SQL-запроса на 2 этапа.

1. 1. Подготовка запроса с помощью метода Prepare, а затем
2. Выполнение запроса методом *Execute*.

Простейший пример:

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

Подготовленный вызов также позволяет передавать параметры, например:

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

Подготовленный вызов — это гораздо более гибкий способ выполнения SQL-запросов. Но из приведенного выше примера этого не видно. Раздел «Параметризованные запросы» позволяет увидеть мощь подготовленного звонка.

Параметризованные запросы

Хотя немедленный вызов также позволяет использовать параметризованные запросы, в дальнейшем мы будем использовать в основном подготовленный вызов, как более гибкий и выразительный. Вот простейший пример параметризованного запроса с использованием подготовленного вызова:

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

В приведенном выше примере функция Execute выполняет запрос, подготовленный функцией Prepare. Параметры указываются в запросе с помощью стандартного синтаксиса SQL, который подразумевает «@» в качестве префикса имени параметра. В NitrosBase используется альтернативное обозначение «:» в качестве префикса имени параметра. В программном коде можно одинаково использовать как «@», так и «:». NitrosBase InMemory DB допускает любое количество параметров любого разрешенного типа (см. Раздел «Типы полей NitrosBase InMemory DB»).

Позиционная и ключевая передача параметров

Есть 2 способа передачи параметров: позиционный и ключевой. Позиционная передача параметров — это передача параметров в том же порядке, в котором они перечислены в операторе SQL. Например,

```
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);
}
```

В приведенном выше примере p1 получает значение i, p2 получает значение i * 10 и так далее. Функция Execute может обрабатывать до 8 параметров.

Позиционная передача параметров - самый быстрый способ выполнения параметризованных запросов.

Часто более удобной является ключевая передача параметров. Например:

```
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();
}
```

В приведенном выше примере метод par явно присваивает значения параметрам.

Ключевая передача параметров происходит значительно медленнее, чем позиционная.

Если Вам нужно больше параметров

Простые методы выполнения параметризованных запросов, такие как ExecuteSQL (p1, p2, ...) или Execute (p1, p2, ...), поддерживают до 8 параметров. Если вам нужно больше параметров, используйте ключевую передачу параметров, описанную выше. Этот метод не ограничен по количеству параметров.

Навигация по результату запроса

Навигация по результату запроса с использованием класса CQuery напоминает навигацию в .NET с использованием класса SQLReader. Можно перейти только к следующей записи. Перемещение осуществляется методом Next. Метод читает следующую запись и возвращает TRUE при успешном чтении. Метод Next возвращает FALSE, если в наборе больше не осталось записей.

Типичный сценарий:

```
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
}
```

Чтение полей записи

Следующая функция перемещает курсор к следующей записи в наборе и копирует запись в буфер. Когда функция Next вызывается в первый раз, она помещает курсор на первую запись. вы можете прочитать значения поля записи из буфера. Сделать это можно следующими способами.

Метод 1 – наиболее быстрый

Получите указатель на буфер. Объявите структуру C ++, отображающую структуру записи запроса, и объявите указатель на буфер как указатель на структуру. Теперь вы можете сразу получить доступ к полям записи как к переменным структуры.

Пример:

```
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";
}
```

Или то же самое, если вы предпочитаете ссылки:

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

Этот способ самый быстрый, но требует тщательного обращения. Изменяя запрос, вы должны аккуратно изменить объявление структуры.

Метод 2

Вы можете получить доступ к полю по его номеру и избежать объявления структуры. Например,

```
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]);
}
```

Доступ к полю по его номеру позволяет избежать объявления структуры, но, поскольку Метод 2 требует точного упорядочивания полей. Это может показаться утомительным, особенно для длинных записей.

Метод 3 – самый медленный, но самый удобный

Вы можете использовать имя поля для доступа к его содержимому. Это самый удобный способ. Например:

```
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"]);
}
```

Модификация записей (Insert, Update, Delete)

Есть следующие способы модификации записей.

Выполнение SQL запроса

Это наиболее очевидный и широко используемый способ, например:

```
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");
```

Вызов методов CQuery

Это проприетарный подход NitrosBase InMemory DB, который намного быстрее, чем использование SQL. Вы можете вызывать методы запроса: Insert, Update, Delete.

Вставка с помощью CQuery

Есть 2 способа вставки записей через CQuery. Первый самый простой: он вставляет инициализированную вручную запись. Например,

```
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);
}
```

В приведенном выше примере q.Prepare ("Select * from PersonTable") получает структуру записи. Остальные строки инициализируют ВСЕ ПОЛЯ записи. Для этого метода важно, чтобы все поля были инициализированы перед вызовом q.Insert.

Второй способ используется, когда вы рассматриваете возможность создания новой записи как копии другой записи с некоторыми изменениями. Например:

```
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);
}
```

Как видно из этого примера, способы немного отличаются:

- Во втором подходе вам нужно получить копию записи в буфер, вызвав все функции: Prepare, Execute, Next; в то время как в первом вам нужно только Prepare.

- В первом случае вам нужно инициализировать все поля, а во втором вам нужно инициализировать только некоторые поля (Id в этом примере). Остальные значения остаются такими же, как в записи, из которой они скопированы.

Изменение записи с помощью CQuery

Обновление записей с помощью Cquery аналогично 2-му способу вставки записи. Например,

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

Некоторые комментарии к приведенному выше примеру:

- В запросе Select SQL указываются записи для обновления. В этом конкретном случае необходимо обновить все записи с Id менее 4.
- Next() выставляет курсор на следующую обновляемую запись
- Поле Id записи увеличено на 20 в буфере
- Update () передает изменения в запись базы данных.

Удаление записей с помощью CQuery

Удаление записей с помощью Cquery легко понять из примера:

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

Некоторые комментарии к приведенному выше примеру:

- В запросе Select SQL указываются записи для удаления, в данном конкретном случае Id меньше 14.
- Next() помещает курсор на следующую запись, которую нужно удалить.
- Delete() удаляет запись.

Типы данных NitrosBase InMemory DB

В настоящее время NitrosBase поддерживает следующие типы полей:

NitrosBase type	Comments
INT	Целочисленное 32-битное значение в диапазоне [-2 147 483 648..2 147 483 647]
BIGINT	Целочисленное 64-битное значение в диапазоне [-9 223 372 036 854 775 808.. 9 223 372 036 854 775 807]
FLOAT	Значения с плавающей запятой в диапазоне [$-1.79769*10^{308}$..$+1.79769*10^{308}$]
DATETIME	Дата и время (См. раздел «Тип даты и времени»)

VARCHAR()	Строка. Длина строки должна быть в диапазоне: [1..8000]

Сопоставление типов данных C++ и NitrosBase InMemory DB

При разработке приложений NitrosBase на C ++ необходимо учитывать следующее сопоставление типов данных:

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

DATETIME

NitrosBase InMemory DB поддерживает следующий формат даты и времени: 'mm.dd.yyyy hh: mm: ss'.

Например,

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

Внутренний формат данных datetime описывается следующей структурой:

```
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
    ...
};
```

Пример добавления данных datetime в таблицу NitrosBase с использованием структуры CDateTime:

```
    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";
        }
    }
```

Или то же самое можно сделать без объявления структуры Birth:

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

Индексы NitrosBase InMemory DB

NitrosBase поддерживает следующие типы индексов:

Notation	Description
TREE	**Tree Index.** Используется для любого вида фильтрации данных: «<», «>», «=», «= <», «> =». Индекс доступен для уникальных и неуникальных полей.
HASH	**Hash Index.** Используется для высокопроизводительного поиска по «=». Эффективно работает для уникальных полей.
MD_HASH	**Multidimensional Hash Index.** 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.

Примечание. Hash index создается автоматически для уникального поля. Индекс дерева для уникального поля не создается автоматически и должен быть создан с помощью запроса «CREATE INDEX».

Большинство индексов, поддерживаемых этой версией Nitro Base In Memory DB, представляют собой индексы по одному полю. Исключение составляет многомерный хеш-индекс.

Tree Index

Это наиболее универсальный тип индекса. ОН может использоваться в любых запросах. Он полезен для запросов использующих следующие операции сравнения: “<”, “>”, “=”, “=<”, “>=”. Tree index может использоваться ка для уникальных, так и для неуникальных полей.

Синтаксис запроса на создание Tree Index:

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

Tree Index является индексом по умолчанию, то есть вы можете написать:

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

Замечание: до версии 2.0 синтаксис Tree Index был следующим:
CREATE INDEX <index_name> ON <table_name> (<field_name>) WITH 1. Этот устаревший синтаксис, он не рекомендуется к использованию, но поддерживается для совместимости.

Tree index может неявно использоваться для оптимизации выражения ORDER BY. А именно, если вы знаете, что какое-то поле будет использоваться в предложении ORDER BY в запросе, который вы готовите, было бы очень полезно заранее создать Tree index для этого поля.

```
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.

В приведенном выше примере вы собираетесь упорядочить результат запроса по полю веса. Создание Tree index в этом случае очень полезно с точки зрения производительности.

Индексы для уникальных полей

Hash index используется для высокопроизводительного поиска по “=”. Хотя вы можете использовать либо Tree, либо Hash index при выборе записей с помощью “=”, Hash index работает значительно лучше. Единственное требование - поле, используемое в Hash Index, должно быть уникальным.

Если поле объявлено как уникальное (в предложении CREATE TABLE), то Hash index для этого поля создается автоматически. Кроме того, вы можете вручную создать Tree index, если собираетесь производить интервальный поиск по уникальным полям. Например:

```
// 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'");
```

Кроме того, вы можете вручную создать древовидный индекс для этого поля.

Замечание: Если вы знаете, что какое-то поле таблицы заполнено уникальными значениями (даже если не объявлено как UNIQUE в запросе CREATE TABLE), вы можете создать хеш-индекс для этого поля вручную, чтобы использовать его уникальность. Однако мы не рекомендуем этот метод. Лучший способ обрабатывать уникальные поля - объявить их как УНИКАЛЬНЫЕ при создании таблицы.

Индексы для неуникальных полей

Hash Index не работает для неуникального поля. И tree index, и hash index должны быть объединены, если вы хотите ускорить поиск с помощью знака “=” для неуникального поля. NitrosBase реализует такую комбинацию следующим образом:

```
// 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 ");
```

т. е. всякий раз, когда вы создаете хэш-индекс для неуникального поля, автоматически создается tree index для этого поля.

Note: До NitrosBase 2.0 использовался HASH_AND_TREE index. В настоящее время он устарел, но поддерживается для совместимости. Этот индекс может быть создан с помощью следующих команд:
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 по нескольким полям) позволяет очень быстро искать по “=”, когда условия связаны логической операцией AND. Он поддерживает различные типы полей. И типы не обязаны быть уникальными.

Синтаксис создания Multidimensional Hash Index:

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

Есть несколько правил использования Multidimensional Hash Index:

- Список полей должен заканчиваться специальным полем типа int. Это поле должно быть создано вместе с таблицей и не должно использоваться для чего-либо еще, кроме обслуживания индекса.
- Порядок полей в предложении CREATE INDEX должен быть сохранен в следующем предложении SELECT.

Следующий пример иллюстрирует детали. Давайте создадим таблицу

```
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)");
```

Предположим, мы хотим выбрать записи, касающиеся 3-дверной Toyota Corolla за все годы. Мы проектируем следующие вызовы API:

```
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");
```

Заметим, что

- порядок имен полей в запросе CREATE INDEX и запросе SELECT такой же, и он отличается от порядка в CREATE TABLE.
- Поле FieldMF создается запросом CREATE TABLE, но не используется где-либо еще, кроме списка полей запроса CREATE INDEX.

Рассмотрим еще один пример. Допустим, мы выбрали все модели 4-дверной Toyota, у которых больше 4-х цилиндров. Тогда запросы CREATE INDEX и SELECT будут выглядеть следующим образом:

```
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 ");
```

Здесь можно заметить, что список полей в запросе SELECT шире, чем в CREATE INDEX. Он включает марку, двери и цилиндры, тогда как индексированные поля включают марку и двери. Более того, Cylinders > 4, по-видимому, нарушает правила. Ответ таков: как только будут представлены Make =… AND Doors =…, они будут найдены с использованием индекса MDIndex02, а ограничение Cylinders >… будет участвовать в поиске отдельно. Но все запросы верные и будут выполняться правильно.

Переменнные и методы класса CQuery

	Description
Db	Указатель на класс базы данных, возвращаемый dbconnect. Присоединяет CQuery к базе данных.
Delete	Удаляет текущую запись
Execute	Выполняет запрос, подготовленный методом Prepare. Может обрабатывать до 8 параметров.
ExecuteSQL	Немедленный запуск запроса. Запрос может обрабатывать до 8 параметров.
GetId	Returns identifier for current record. Usually is called after Next method
GetNewRecId	Возвращает идентификатор новой записи. Обычно вызывается после метода Insert
GetNitrosBaseVersion	Возвращает версию Nitro Base In Memory DB.
GetRec	Возвращает указатель на копию записи, полученную запросом. Обычно вызывается после метода Next
Insert	Вставляет текущую запись
Next	Устанавливает курсор на 1-ю запись при первом вызове или на следующую запись при повторном вызове. Используется после запроса Select для навигации по результату запроса.
Par	Разрешает доступ к параметрам запроса по их номерам или именам с помощью оператора []
Prepare	Подготавливает запрос к повторным запускам. Обычно используется для параметризованных запросов
Update	Обновляет текущую запись

Обработка ошибок

Обработку ошибок NitrosBase отлично поддерживает стандартный механизм обработки исключений C ++. Обычно все, что вам нужно сделать, это заключить код в блоки TRY / CATCH.

```
...
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);
}
...
```

Если в коде блока TRY нет ошибок, управление передается оператору сразу после блока CATCH. В противном случае управление передается первому оператору в связанном блоке CATCH; сообщение об ошибке создается и сохраняется в переменной ошибки.

Текущее состояние NitrosBase

Ограничения текущей версии

- Обычно поставляется 32-битная версия NitrosBase для семейства ОС Microsoft Windows..
  Позвоните в NitrosBase для 64-битных версий и для версий Linux..
- Обычно поставляется встроенная версия NitrosBase. Реализована в виде динамически загружаемой библиотеки..
  Позвоните в NitrosBase для получения серверной версии NitrosBase.
- Обычно поставляется без встроенных функций резервного копирования / восстановления.
  Позвоните в NitrosBase, если вам нужна эта функция.
- Обычно поставляется без административной оболочки.
  Call NitrosBase if you need one.
- Нет поддержки транзакций (функция запланирована в будущих версиях).

Ограничения NitrosBase SQL

Эта версия NitrosBase SQL поддерживает только следующие операторы SQL:

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

Эта версия NitrosBase SQL не поддерживает:

- ALTER statement (запланирована реализация в версии 3).
- Агрегатные функции (запланирована реализация в версии 3)
- Функции (частичная реализация запланирована в версии 3)
- IN operator (частичная реализация запланирована в версии 3 или 4)
- Вложенные запросы (частичная реализация запланирована в версии 4)
- DISTINCT (запланирована реализация в версии 4)
- Хранимые процедуры и функции
- CONSTRAIN
- OUTER JOIN
- SELECT TO …
- Старый синтаксис JOIN запросов

Другие ограничения

Имена таблиц и имена полей чувствительны к регистру, в то время как ключевые слова SQL и типы данных NitrosBase нечувствительны к регистру. Это сделано из соображений производительности. Такие объекты, как таблицы, поля, индексы и т. д. обнаруживаются в хэш-таблицах во время синтаксического анализа. Это может быть намного быстрее, если их имена помещены в хеш-таблицу как есть.

Размер записи ограничен 4 МБ, включая все строки, даты, числа и т. д.