NITROSBASE SQL V2.6

Описание языка SQL
ID:NBSQL26-SQL-001

ООО «НитросДэйта Рус»
2025

Оглавление

Введение 4
Синтаксические элементы языка SQL 4
Создание таблиц....5
Оператор CREATE TABLE....5
Создание временных таблиц....7
Создание Edge-таблиц....7
Изменение структуры таблицы 8
Оператор ALTER TABLE 8
Удаление таблицы....9
Оператор DROP TABLE....9
Добавление данных в таблицу....9
Оператор INSERT 9
Оператор INSERT INTO … SELECT 11
Оператор BULK INSERT 11
Модификация данных в таблице 12
Оператор UPDATE....12
Удаление данных из таблицы 12
Оператор DELETE 12
Оператор TRUNCATE TABLE 13
Оператор SELECT 13
Оператор UNION 22
Резервное копирование / восстановление 22
Резервное копирование 22
Восстановление базы из backup файла....23
Функции....23
Функция FORMAT 26
Типы данных 28
Работа с датой и временем 30
Тип DATE 30
Тип DATETIME 31

Тип DATETIME2 31
Оператор SET DATEFORMAT 32
Работа с индексами 33
Оператор CREATE INDEX 33
Оператор DROP INDEX 34
Использование индексов 34
Транзакции 35
Уровни изоляции транзакций 35
ПСЕВДОТАБЛИЦЫ 36
Работа со схемой данных 36
Работа с Edge-таблицами 37

ВВЕДЕНИЕ

В документе описан диалект SQL, поддерживаемый NitrosBase. Подавляющее большинство приведенных здесь примеров можно воспроизвести, используя приложение NBAdmin (см. документ «Руководство администратора»).

В примерах используется демонстрационная база данных, описанная в разделе *«Демонстрационная база данных»* документа *«Руководство администратора»*.

СИНТАКСИЧЕСКИЕ ЭЛЕМЕНТЫ ЯЗЫКА SQL

В следующей таблице перечислены обозначения, которые используются в синтаксических диаграммах данного документа.

Обозначение	Комментарий
ПРОПИСНЫЕ БУКВЫ	Ключевые слова SQL.
курсив	Пользовательские параметры синтаксиса SQL.
полужирный	Имена баз данных типов, таблиц, столбцов, индексов, текст и т.д. должны вводиться в точном соответствии с примером.
| (вертикальная черта)	Разделяет элементы синтаксиса внутри квадратных или фигурных скобок. Может быть использован только один из этих элементов.
[ ] (квадратные скобки)	Необязательный элемент синтаксиса.
{ } (фигурные скобки)	Обязательные элементы синтаксиса. Фигурные скобки не вводятся.
[, ...n]	Означает, что предшествующий элемент можно повторить n раз. Отдельные вхождения элемента разделяются запятыми.
[...n]	Означает, что предшествующий элемент можно повторить n раз. Отдельные вхождения элемента разделяются пробелами.
;	Признак конца SQL запроса.
<label> ::=	Имя синтаксического блока. Синтаксические блоки используются для • группирования сегментов с длинным синтаксисом или • обозначения синтаксического элемента, используемого в нескольких расположениях данного документа. Каждое расположение, в котором может быть использован данный блок, обозначается меткой, заключенной в шевроны: <label>.

Ключевые слова языка SQL, например, SELECT, UPDATE и VALUES не чувствительны к регистру, например, следующие запросы эквивалентны:

```
SELECT * FROM car WHERE Year > 2011 AND Year < 2017;
select * from car where Year > 2011 and Year < 2017;
Select * From car Where Year > 2011 And Year < 2017;
```

Идентификаторы, например названия таблиц или полей также не чувствительны к регистру, например, следующие запросы эквивалентны:

```
SELECT * FROM car WHERE Year > 2011 AND Year < 2017;
SELECT * FROM CAR WHERE year > 2011 AND year < 2017;
SELECT * FROM Car WHERE YEAR > 2011 AND YEAR < 2017;
```

Поддерживаются однострочные (--) и многострочные (/* … */) комментарии, например:

```
/*
 * Два одинаковых запроса на разных языках
 * Демонстрируют преимущества графового подхода
 */
-- Запрос на SQL
SELECT car.number FROM car
  JOIN owner ON owner.fromid = car.id
  JOIN person ON owner.toid = person.id
  WHERE person.id = 'person679'
-- Запрос на Graph-SQL
SELECT c.number MATCH! (person p, p.id='person679')<-[owner]-(car c)
```

СОЗДАНИЕ ТАБЛИЦ

Оператор CREATE TABLE

NitrosBase поддерживает все классические опции и параметры оператора *CREATE TABLE.*

Синтаксис

```
CREATE TABLE tbl_name
  ( create_definition [, create_definition] ... )

create_definition:
  col_name col_data_type [NOT NULL | NULL] [DEFAULT literal]
    [IDENTITY [(seed, increment)]
    [PRIMARY KEY] | [CONSTRAINT [symbol]] FOREIGN KEY
    [index_name] (col_name) REFERENCES tbl_name
```

Пример

```
CREATE TABLE vehicle (
  id nvarchar PRIMARY KEY,
  model nvarchar,
  year int,
);
```

Пример

```
CREATE TABLE IF NOT EXISTS car (
  id int NOT NULL,
  model nvarchar,
  year int,
  color nvarchar,
  number nvarchar,
  owner nvarchar FOREIGN KEY(owner) REFERENCES person
);
```

Поле IDENTITY

поле identity — это числовое поле, которое автоматически получает целое значение, когда вставляется запись. Поля identity могут быть объявлены как int или bigint. В таблице может быть только одно поле identity.

Значения поля identity генерируются автоматически для каждой вставляемой строки на основе параметров seed (начальное значение) и increment (приращение) столбца identity. По умолчанию seed = 1, increment = 1. Если вы хотите изменить значения параметров, необходимо указать оба значения.

Пример.

```
CREATE TABLE Engine
(
    EngineID int identity(10,5) not null,
    EngineName varchar(50) not null,
    EngineDesc varchar(100) not null
);
INSERT INTO Engine (EngineName, EngineDesc) VALUES
    ("Bentley", "The perfrct car in the world");
INSERT INTO Engine (EngineName, EngineDesc) VALUES
    ("Renault", "The OK car");

SELECT * from Engine;
```

Результатом оператора SELECT выше будет

EngineID	EngineName	EngineDesc
10	Bentley	The perfect car
15	Renault	The OK car

Создание временных таблиц

Если имя таблицы начинается со знака '#', то создается временная таблица. Временная таблица существует только в пределах создавшей ее сессии и не видна извне. Когда сессия заканчивается, временная таблица автоматически удаляется.

Пример

```
CREATE TABLE #temp1 (
  p_id varchar PRIMARY KEY,
  name varchar,
  lastname varchar,
);
```

В остальном временные таблицы ничем не отличаются от обычных таблиц. В частности, если необходимо удалить временную таблицу до завершения сессии, следует выполнить запрос DROP TABLE для этой таблицы.

Создание Edge-таблиц

Edge-таблица - специальная (псевдо) таблица для представления графовой связи (связь типа «многие-ко-многим»). Edge-таблица может интерпретироваться как промежуточная таблица в обычной реляционной базе данных, имеющая два фиксированных поля: ***fromid*** и ***toid***.

Синтаксис

```
CREATE TABLE link_name AS EDGE from_tablename to_tablename;
```

Пример

Edge-таблица *owner* связывает (обычные) таблицы *car* и *person* (см. раздел «Демонстрационная база данных» документа «Руководство администратора»).

```
CREATE TABLE owner AS EDGE car person;
```

После создания, такая таблица может использоваться, например, в SQL запросах:

```
SELECT p.id, name, lastname, age, c.id, c.model
FROM person p
  JOIN owner o ON p.id = o.toid
  JOIN car   c ON c.id = o.fromid
WHERE age > 20 AND model = 'Toyota';
```

или Graph-SQL запросах:

```
SELECT p.id, name, lastname, age, c.id, c.model
    MATCH (car c) - [owner] -> (person p)
WHERE age > 20 AND model = 'Toyota';
```

Подробнее об Edge-таблицах и работе с ними см. документ «Работа с графами».

ИЗМЕНЕНИЕ СТРУКТУРЫ ТАБЛИЦЫ

Оператор ALTER TABLE

Синтаксис

```
ALTER TABLE tbl_name
  <alter_specification> [,...]

<alter_specification> ::=
  | ADD [COLUMN] <column_definition>,...
  | {ALTER|MODIFY} [COLUMN] column_definition,...
  | DROP [COLUMN [if exists]] col_name

<column_definition> ::=
  col_name data_type
    [NOT NULL | NULL] [DEFAULT literal]
  | [CONSTRAINT [symbol]] FOREIGN KEY
    [index_name] (col_name) REFERENCES tbl_name
```

Пример.

Добавление столбца

```
ALTER TABLE person ADD car_id int, nameday DATE
```

Переименование столбца

```
ALTER TABLE car RENAME id car_id, Model car_model;
```

Удаление столбца

```
ALTER TABLE car DROP Color;
```

УДАЛЕНИЕ ТАБЛИЦЫ

Оператор DROP TABLE

Синтаксис

```
DROP TABLE [IF EXISTS] tbl_name
```

tbl_name - имя таблицы или Edge-таблицы

Примеры

```
DROP TABLE person;
CREATE TABLE person(id int PRIMARY KEY, value varchar);
```

Если таблица *person* существует, она будет удалена, затем будет создана новая таблица person. Если таблица *person* не существует, то оператор *DROP TABLE вернет ошибку* и оператор *CREATE TABLE* выполнен не будет. Для более гладкого прохождения процесса используйте выражение *IF EXISTS*:

```
DROP TABLE IF EXISTS person;
CREATE TABLE person(id int PRIMARY KEY, value varchar);
```

Если таблица *person* существует, она удаляется, затем в любом случае создается новая таблица person.

ДОБАВЛЕНИЕ ДАННЫХ В ТАБЛИЦУ

Оператор INSERT

Синтаксис

```
INSERT INTO tbl_name
  [(col_name,...)]
  VALUES (value,...)
```

Примеры

Если в таблице *person* заполняются все поля в порядке следования, то можно не указывать их наименования:

```
INSERT INTO person
VALUES
  (1, 'John', 'Goodman', 30, 'Boston', 65000, 3.62, '2022/07/28 00:00:00:00'
```

Если заполняются только первые 5 полей, то также можно не указывать их наименования:

```
INSERT INTO person
VALUES
  ('person5000', 'John', 'Tester', 30, 'Lisbon');
```

Однако, если среди полей есть пропуски и/или изменен порядок полей в операторе, то необходимо указывать наименования полей:

```
INSERT INTO person
  (id, name, lastname, city)
VALUES
  ('person5000', 'John', 'Tester', 'Lisbon');

INSERT INTO person
  (name, id, lastname, age, city)
VALUES
  ('Lamar', 'person2', 'Tanon', 44, 'Lisbon');
```

Замечание. Вставка нескольких строк данных в одном операторе INSERT не поддерживается. Например, вместо

```
INSERT INTO Person
  (id, name, lastname, age, city)
VALUES
('person2', 'Lamar', 'Tanon', 44, 'Lisbon'),
('person3', 'Vanda', 'Tanon', 22, 'Istanbul'),
('person4', 'Libby', 'Courtenay', 39, 'Jakarta'),
('person5', 'Libby', 'Courtenay', 67, 'Paris');
```

Следует писать

```
INSERT INTO Person VALUES
('person2', 'Lamar', 'Tanon', 44, 'Lisbon');
INSERT INTO Person VALUES
('person3', 'Vanda', 'Tanon', 22, 'Istanbul');
INSERT INTO Person VALUES
('person4', 'Libby', 'Courtenay', 39, 'Jakarta');
INSERT INTO Person VALUES
('person5', 'Libby', 'Courtenay', 67, 'Paris');
```

Оператор INSERT INTO … SELECT …

Синтаксис

```
INSERT INTO tbl_name
  [(col_name,...)]
  SELECT ...
```

Пример

```
INSERT INTO young_person
  (id, name, lastname, age, city)
SELECT
  id, name, lastname, age, city
FROM person
WHERE age < 30
```

Оператор BULK INSERT

Синтаксис

```
BULK INSERT table_name (col_name,...)
  FROM 'file_path'
  [WITH ([FIRSTROW = number,] [FIELDTERMINATOR = 'character'])]
```

Параметр *FIRSTROW* указывает, с какой строки файла начинать импорт (нумерация строк начинаются с 1). Например, для того чтобы пропустить заголовок (пропустить одну первую строку), нужно указать *FIRSTROW = 2*.

Путь к файлу как правило указывается абсолютный. Возможно также, хотя и не удобно в большинстве случаев, указывать путь относительно файла сервера NitrosBase. Например, если исполняемый файл сервера *nbserver.exe* находится в каталоге *c:/nitrosbase/bin*, то относительный путь *../data/csv/person.csv* будет раскрыт как абсолютный путь *c:/nitrosbase/data/csv/person.csv*.

Пример

```
BULK INSERT car (id, model, year, color, number)
FROM 'C:/MyApp/data/csv/car.csv'
WITH (FIRSTROW = 2, FIELDTERMINATOR = ',');
```

МОДИФИКАЦИЯ ДАННЫХ В ТАБЛИЦЕ

Оператор UPDATE

Синтаксис

```
UPDATE table_reference
  SET col_name = value,...
  [WHERE where_condition]

UPDATE table_name_or_alias
  SET col_name = value,...
  FROM tablename [alias] | join_expression
  [WHERE where_condition]
```

Синтаксис выражений FROM и WHERE тот же, что и в запросе SELECT.

Пример

```
UPDATE person SET name = 'LAMAR' WHERE name = 'Lamar';
```

УДАЛЕНИЕ ДАННЫХ ИЗ ТАБЛИЦЫ

Оператор DELETE

Синтаксис

```
DELETE FROM tbl_name
  [WHERE where_condition]

DELETE FROM table_name_or_alias FROM join_expression
  [WHERE where_condition]
```

Синтаксис выражений *FROM* и *WHERE* тот же, что и в запросе *SELECT*.

При удалении записи будут автоматически удалены все ссылки на нее, а также будут удалены записи из Edge-таблиц, в которых удаляемая запись была в поле *fromid* или *toid*

Примеры

```
DELETE FROM person WHERE name = 'Livia';
```

удаление с условием на связанные таблицы

```
DELETE FROM p
FROM person p join car c ON p.id = c.owner
WHERE c.model = 'Toyota';
```

Оператор TRUNCATE TABLE

Удаляет все строки из таблицы. Оператор TRUNCATE TABLE аналогичен оператору DELETE без предложения WHERE. TRUNCATE TABLE работает быстрее и использует меньше системных ресурсов.

Синтаксис

```
TRUNCATE TABLE tbl_name
```

Пример

```
TRUNCATE TABLE #temptable1;
```

ОПЕРАТОР SELECT

Синтаксис

```
SELECT [ ALL | DISTINCT ]
  [TOP ( expression )]
  <select_list>
  [ FROM { <table_source> } [ ,...n ] ]
  [ WHERE <search_condition> ]
  [ GROUP BY <group_by_list> ]
  [ HAVING <search_condition> ]
  [ ORDER BY <order_by_list> ]
  [ OFFSET <offset_expression> ]

<select_list> ::=
{
  *
  | { table_name | table_alias }.*
  | {
      [ { table_name | table_alias }. ] { column_name }
      | expression
      [ [ AS ] column_alias ]
    }
  | column_alias = expression
} [ ,...n ]

/*
[ FROM { <table_source> } [ ,...n ] ]
*/

<table_source> ::=
{
  table_name [ AS ] table_alias ]
  | rowset_function [ [ AS ] table_alias ]
  | derived_table [ [ AS ] table_alias ]
  | <joined_table>
}
```

```
<joined_table> ::=
{
  <table_source> <join_type> <table_source> ON <search_condition>
  | <table_source> CROSS JOIN <table_source>
  | [ ( ] <joined_table> [ ) ]
}

<join_type> ::=
  [ { INNER | { { LEFT | RIGHT | FULL } [ OUTER ] } } [ <join_hint> ] ]
  JOIN

<group_by_list> ::=
{
  [ { table_name | table_alias }. ] { column_name }
  | expression
  | column_number
} [ ,...n ]

<order_by_list> ::=
{
  [ { table_name | table_alias }. ] { column_name }
  | expression
  | column_number
  [ ASC | DESC ]
} [ ,...n ]

<offset_expression> ::=
{
  OFFSET { integer_constant | offset_row_count_expression } { ROW | ROWS }
  [
    FETCH { FIRST | NEXT } {integer_constant | fetch_row_count_expression } { ROW |
ROWS } ONLY
  ]
}
```

Select выражение

Select выражение – это часть оператора, расположенная между ключевыми словами *SELECT* и *FROM.* Она может содержать:

Простой список полей:

```
SELECT name, lastname FROM person;
```

Все поля:

```
SELECT * FROM person
SELECT p.*, c.model FROM person p JOIN car c ON p.id = c.owner;
```

AS:

```
SELECT COUNT(*) AS c FROM person;
```

Простые выражения и функции:

```
SELECT upper(name) AS uname, age*2 AS dbl FROM person;
```

Агрегатные функции:

```
SELECT COUNT(age), COUNT_BIG(age), MIN(age), MAX(age), AVG(age) FROM person;
```

ALL | DISTINCT:

```
SELECT ALL name, lastname FROM person WHERE name LIKE 'Vanda';
SELECT DISTINCT name, lastname FROM person WHERE name LIKE 'Vanda';;
```

Замечание. Ключевое слово ALL как правило является избыточным, поскольку оно является случаем по умолчанию. То есть, вместо *SELECT ALL name …* можно писать *SELECT name …*

TOP

```
SELECT TOP 1000 * FROM car;
```

Вложенный *SELECT*:

```
SELECT
  name,
  (SELECT model FROM car WHERE p.id = owner)
    AS modelname
FROM person p;
```

Конструкцию CASE…WHEN:

```
SELECT name,
  CASE name
    WHEN 'Lamar' THEN 'L'
    WHEN 'Mercedez' THEN 'M'
    ELSE 'U' END AS n
FROM person;

SELECT id,
  CASE WHEN age < 30 THEN 1
  ELSE 2 END as age_category,
  name, lastname, income
from person
```

Условие поиска

Эта часть оператора *SELECT* может содержать:

Простые операторы сравнения и логические операторы (см. раздел "Операторы"):

```
SELECT * FROM person WHERE age > 20 AND name = 'Lamar';
```

Оператор LIKE, выполняющий сопоставление с образцом для строковых типов (см. раздел "Операторы"):

```
SELECT * FROM person WHERE name LIKE '%amar%';
SELECT * FROM person WHERE name LIKE 'L[ai]%';
```

Оператор IN и NOT IN:

```
SELECT name, age FROM person
WHERE age IN (20, 22, 24) AND name IN ('Lamar', 'Susan');

SELECT * FROM customers
WHERE ctype IN (
  SELECT distinct otype
  FROM orders
  WHERE odate >= '2020-03-03'
);
```

Оператор BETWEEN и NOT BETWEEN

```
select * from Person
where
  age between 23 and 44

select * from Person
where
  age NOT between 23 and 44
order by age
```

Проверка на NULL:

```
SELECT * FROM person WHERE age IS NULL;
SELECT * FROM person WHERE age IS NOT NULL;
```

Вложенный запрос с условиями EXISTS | NOT EXISTS

```
SELECT * FROM person WHERE age > 40
  AND EXISTS( SELECT * FROM car WHERE car.owner = person.id );
```

Другие операторы и функции (список операторов и функций см. далее):

Выражения FROM и JOIN

Запрос к одной таблице:

```
SELECT * FROM person;
```

JOIN операции через поле FOREIGN-KEY или через промежуточную Edge-таблицу являются быстрыми операциями. Мы рекомендуем при разработке стараться максимально использовать такие простые JOIN операции.

NitrosBase существенно ускоряет обработку таких *JOIN* операций, поскольку для хранения связей использует возможности графовой подсистемы. В реляционном представлении связи между объектами представляются как ссылки через поля с *FOREIGN KEY* на *id* записей в этой же или другой таблице.

JOIN между двумя таблицами через поле с FOREIGN KEY:

```
SELECT name, model FROM person p JOIN car c ON p.id = c.owner;
```

JOIN между двумя таблицами через промежуточную таблицу (в данном примере через Edge-таблицу *owner*, предназначенную для описания ребер графа):

```
SELECT p.id, name, lastname, age, c.id, c.model
FROM person p
  JOIN owner o ON p.id = o.toid
  JOIN car c ON c.id = o.fromid
WHERE age > 20 AND model = 'Toyota';
```

Это же самое можно выразить на языке Graph-SQL:

```
SELECT p.id, name, lastname, age, c.id, c.model
    MATCH (car c) - [owner] -> (person p)
WHERE age > 20 AND model = 'Toyota';
```

Подробнее об Edge-таблицах и работе с ними см. документ «Работа с графами».

JOIN операции через простые (не FOREIGN-KEY) поля являются медленными. Мы рекомендуем по возможности перестроить структуру базы данных чтобы в запросах использовать FOREIGN-KEY или хотя бы создать индексы по полям, используемым в JOIN выражении.

JOIN через обычное поле:

```
SELECT p1.id, p2.id
FROM person p1
  JOIN person2 p2
  ON p1.name = p2.name
```

Для выполнения такого запроса требуется индекс:

```
CREATE INDEX p_name ON person (name)
```

и/или

```
CREATE INDEX p2_name ON person2 (name)
```

JOIN через несколько полей с условием *AND*

```
SELECT p1.id, p2.id
FROM person p1
  JOIN person2 p2
  ON p1.name = p2.name AND p1.lastname = p2.lastname
```

Для выполнения такого запроса желательно создать индекс по двум полям

```
CREATE INDEX p_name_lastname ON person (name, lastname)
```

или (для таблицы person2)

```
CREATE INDEX p2_name_lastname ON person2 (name, lastname)
```

Но достаточно иметь один из простых индексов по одному полю name или lastname для таблицы person или person2

```
CREATE INDEX p_name ON person (name)      -- or
CREATE INDEX p_name ON person (lastname)  -- or
CREATE INDEX p_name ON person2 (name)     -- or
CREATE INDEX p_name ON person2 (lastname)
```

JOIN с результатом вложенного запроса SELECT

```
SELECT p.age, p2.age
FROM person p
  JOIN ( SELECT name, lastname, age FROM person ) p2
  ON p.name = p2.name and p.lastname = p2.lastname
```

NitrosBase поддерживает все виды JOIN, а именно, INNER JOIN, LEFT [OUTER] JOIN, RIGHT [OUTER] JOIN, FULL [OUTER] JOIN, CROSS JOIN.

Примеры

```
SELECT *
FROM tab1 t1
  JOIN tab2 t2 ON t1.c1 = t2.c1;

SELECT *
FROM tab1 t1
  LEFT JOIN tab2 t2 ON t1.c1 = t2.c1;

SELECT *
FROM tab1 t1
  RIGHT JOIN tab2 t2 ON t1.c1 = t2.c1;

SELECT *
FROM tab1 t1
  FULL JOIN tab2 t2 ON t1.c1 = t2.c1;

SELECT *
FROM tab1 t1
  CROSS JOIN tab2 t2;
```

Выражение GROUP BY

Простая группировка по одному полю:

```
SELECT COUNT(dbl), AVG(dbl), MIN(dbl), MAX(dbl)
FROM person
GROUP BY city;
```

Группировка по нескольким полям:

```
SELECT COUNT(dbl), AVG(dbl), MIN(dbl), MAX(dbl)
FROM person
GROUP BY city, name;
```

Выражение HAVING

Фильтрация результатов агрегации с использованием HAVING:

```
SELECT city, AVG(dbl) AS avg_dbl
FROM person
GROUP BY city
HAVING (avg_dbl <= 0.5);
```

В агрегирующих функциях возможно использовать различные выражения и *DISTINCT*:

```
SELECT AVG(income*dbl) AS avg_expr FROM person

SELECT AVG(
  CASE WHEN name = 'Lamar' THEN 1
       WHEN name = 'Mercedez' THEN 2
       ELSE 0 end
  ) AS avg_expr
FROM person'

SELECT city, COUNT(distinct id) FROM person GROUP BY city;
```

Если выражение *GROUP BY* опущено, но присутствуют агрегирующие функции, весь результат трактуется как одна группа.

Выражение ORDER BY

Простая сортировка по одному полю:

```
SELECT name, lastname, age
FROM person
ORDER BY name;
```

Сортировка по нескольким полям и с указанием порядка сортировки:

```
SELECT name, lastname, age
FROM person
ORDER BY name, lastname ASC, age DESC;
```

Сортировка по номеру колонки (номер колонки в выражении SELECT начинается с 1):

```
SELECT name, lastname, age
FROM person
ORDER BY 1, lastname ASC, 3 DESC;
```

Сортировка по выражению:

```
SELECT name, lastname, age
FROM person
ORDER income/age, name, lastname;
```

Сортировка с использованием функций:

```
SELECT COUNT(name), age
FROM person
GROUP BY age
ORDER BY COUNT(name);
```

Выражения LIMIT и OFFSET

LIMIT и *OFFSET* служат для ограничения количества записей в результате:

Вывести только 100 записей:

```
SELECT * FROM person LIMIT 100;
```

Вывести 10 записей, начиная с 100-й:

```
SELECT * FROM person ORDER BY id LIMIT 10 OFFSET 10;
```

Замечание. Поскольку нумерация записей начинается с нуля, это значит "пропустить первые 100 записей":

Замечание. Поскольку в SQL в общем случае порядок получения данных не определен, то в примере выше без конструкции *ORDER BY* мы получили бы неопределенный результат.

Операторы

операторы сравнения: *>*, *>=*, *<*, *<=*, *=*, *<>*, *!=*, *IS NULL*, *IS NOT NULL*, *IN*

```
SELECT * FROM person WHERE age IS NULL;
```

Оператор *LIKE*, выполняющий сопоставление с образцом для строковых типов.

Наряду с *%* (любая последовательность символов) и *_* (любой один символ) в шаблонах разрешается использовать заключенные между *"[" и "]"* перечисления символов. При задании перечислений разрешается использовать *^* (отрицание) и *-* (диапазон). Для экранирования спецсимволов следует заключать их в квадратные скобки.

```
SELECT * FROM person WHERE name LIKE '%amar%';
SELECT * FROM person WHERE name LIKE 'L[ai]%';
```

Логические операторы: *AND* (*&&*), *OR* (*||*), *NOT*

```
SELECT * FROM person
WHERE age > 50 AND (name = 'Lamar' OR lastname = 'Wurdeman');
```

Арифметические операторы: *+*, *-*, ***, */*

```
SELECT age*2 FROM person WHERE age*100/income > 3;
```

ОПЕРАТОР UNION

Синтаксис

```
SELECT ...
UNION [ALL]
SELECT ...
[UNION [ALL] SELECT ...]
```

Пример

```
SELECT id, name, lastname, city FROM person WHERE id = 'person22'
UNION
SELECT id, name, lastname, city FROM person WHERE id = 'person33'
UNION
SELECT id, name, lastname, city FROM person WHERE id = 'person55
```

РЕЗЕРВНОЕ КОПИРОВАНИЕ / ВОССТАНОВЛЕНИЕ

Резервное копирование

Для резервного копирования можно использовать CLI утилиту nbase.exe. Для этого необходимо вызвать команду, например,

```
$ nbase –c backup –p 3023 –b "C:\snt0"
```

В этом случае необходимо указать порт и абсолютный путь к backup каталогу.

Либо можно использовать утилиту администрирования NBAdmin (см. документ " Руководство администратора"). В утилите администрирования можно произвести резервное копирование с помощью пользовательского интерфейса, либо же, выбрав базу данных, в окне запросов набрать SQL запрос.

```
BACKUP DATABASE TO Path
```

Где Path – абсолютный путь.

Пример

```
BACKUP DATABASE TO 'c:/data/sntest1/backup'
```

В результате выполнения этой команды backup копия текущей базы данных сохраняется в каталог *'c:/data/sntest1/backup'.*

Восстановление базы из backup файла

Для восстановления базы данных можно использовать как CLI утилиту nbase.exe, так и NBAdmin (см. документ "NitrosBase Admin Tool").

В случае использования nbase.exe, следует выполнить, например, следующую команду:

```
$ nbase sntest2 -c restore -p 3023 -b "C:\sntest2.2022-07-26.12-57-17"
```

При использовании NBAdmin следует пользоваться средствами пользовательского интерфейса. Подробнее см. документ " Руководство администратора"

ФУНКЦИИ

Функция	Описание
ABS	Возвращает абсолютное значение (модуль) числа
ACOS	Возвращает арккосинус действительного числа
ASCII	Возвращает ASCII-код символа
ASIN	Возвращает арксинус действительного числа
ATAN	Возвращает арктангенс действительного числа
CAST	Синтаксис: CAST(expression AS data_type[(length)]) Выполняет преобразование типов. Возвращает значение аргумента *expression*, преобразованное в тип *data_type*.
CEIL	Возвращает наименьшее целое, большее переданного аргумента или равное ему
CEILING	Возвращает наименьшее целое, большее или равное указанному числовому выражению
CHR	Возвращает символ с соответствующим ASCII-кодом
CONCAT	Соединение (конкатенация) двух или более строк

CONVERT	Синтаксис: CONVERT(data_type[(length)], expression) Выполняет преобразование типов. Возвращает значение аргумента *expression*, преобразованное в тип *data_type*.
COS	Возвращает косинус числа
COT	Возвращает котангенс числа
DATEADD	Синтаксис: DATEADD (datepart , number , date) Возвращает измененное значение даты или времени, специфицированного аргументом *date*, в результате добавления целого числа, специфицированного аргументом *number*.
DATEDIFF	Синтаксис: DATEDIFF (datepart , startdate , enddate) Возвращает количество полных единиц, указанных в аргументе *datepart*, прошедших за период времени, указанный в аргументах *startdate* и *enddate.* Аргумент *datepart* может принимать следующие значения: YEAR, QUARTER, MONTH, DAYOFYEAR, DAY, WEEK, WEEKDAY, HOUR, MINUTE, SECOND
DATEPART	Возвращает целочисленное значение указанной части даты
DATETIME2FROMPARTS	Возвращает datetime2, построенное из указанных параметров: *year, month, day, hour, minute, seconds, fractions, precision*
DIV	Целочисленное деление, возвращается целое — результат деления с остатком первого аргумента на второй
EXP	Возвращает основание натуральных логарифмов *e*, возведенное в степень, переданную как аргумент
FORMAT	См раздел «Функция FORMAT» ниже.
GETDATE	Возвращает текущую дату и время с точностью до миллисекунд
ISNULL	Проверяет выражение на NULL и возвращает указанное значение если NULL иначе возвращает значение выражения
LEFT, *STRLEFT*	Возвращает начальную подстроку исходной строки (первый аргумент) с определенной длиной (второй аргумент)
LEN, *LENGTH*	Возвращает длину (количество байтов) строки

LN, LOG	Возвращает натуральный логарифм числа
LOG10	Возвращает логарифм по основанию 10
LOG2	Возвращает логарифм по основанию 2
LOWER	Преобразует символы строки в нижний регистр
LPAD	Дополняет слева первую строку второй до достижения определенной длины
MOD	Остаток от целочисленного деления первого аргумента на второй
MONTH	Возвращает месяц даты
PI	Возвращает значение числа пи
POWER	Возводит число (первый аргумент) в степень (второй аргумент)
POS, POSITION	Возвращает позицию первого вхождения второй строки в первую
RAND	Возвращает случайное число большее 0 и меньшее 1
REPEAT	Сцепляет строку саму с собой определенное число раз; строка — первый аргумент, количество повторений — второй
REPLACE	Заменяет в исходной строке (первый аргумент) все вхождения одной строки (второй аргумент) другой (третий аргумент)
REVERSE	Обращает строку
ROUND	Округляет число до указанного количества десятичных разрядов
STRLEFT	Начальная подстрока исходной строки (первый аргумент) определенной длины (второй аргумент)
STRRIGHT	Конечная подстрока исходной строки (первый аргумент) определенной длины (второй аргумент)
RPAD	Дополняет справа первую строку второй до достижения определенной длины
REGEX	Проверка совпадения строки с образцом, задаваемым регулярным выражением

RIGHT, STRRIGHT	Возвращает начальную подстроку исходной строки (первый аргумент) с определенной длиной (второй аргумент)
SIGN	Возвращает знак числа: -1 для отрицательных, 0 для 0, 1 для положительных
SIN	Возвращает синус числа
SQRT	Возвращает квадратный корень числа
SUBSTR, SUBSTRING	Выделяет подстроку из строки
TAN	Возвращает тангенс числа
TRIM	Удаляет начальные и конечные пробелы из строки
TRUNCATE	Усекает число до указанного числа знаков после запятой
UPPER	Преобразует переданную строку в верхний регистр
YEAR	Возвращает год переданной даты

Функция FORMAT

Функция FORMAT возвращает строковое значение в указанном формате. Выполняет форматирование даты, времени и чисел.

Синтаксис

```
FORMAT (value, format [, culture])
```

Параметры

Value Выражение поддерживаемого типа данных для форматирования

Format Шаблон формата

Culture Языковые и региональные параметры. Необязательный строковый аргумент. Игнорируется в текущей версии NitrosBase.

Для значений даты и времени используются следующие шаблоны в форматной строке:

Шаблон	Комментарий
d	День месяца (1..31) одной или двумя цифрами.
dd	День месяца (1..31) двумя цифрами. Для значений меньших 10 добавляется лидирующий 0.
M	Номер месяца (1..12) одной или двумя цифрами.
MM	Номер месяца (1..12) двумя цифрами. Для значений меньших 10 добавляется лидирующий 0.
yy	Год двумя цифрами. При необходимости добавляется лидирующий 0.
yyyy	Год четырьмя цифрами. При необходимости дополняется лидирующими нулями до требуемого числа разрядов.
h	Час суток в 12-часовом представлении (1..12) одной или двумя цифрами.
hh	Час суток в 12-часовом представлении (1..12) двумя цифрами. При необходимости добавляется лидирующий ноль.
H	Час суток в 24-часовом представлении (0..23) одной или двумя цифрами.
HH	Час суток в 24-часовом представлении (0..23) двумя цифрами. При необходимости добавляется лидирующий ноль.
m	Минута часа (0..59) одной или двумя цифрами.
mm	Минута часа (0..59) двумя цифрами. При необходимости добавляется лидирующий ноль.
s	Секунда (0..59) одной или двумя цифрами.
ss	Секунда (0..59) двумя цифрами. При необходимости добавляется лидирующий ноль.
t	Маркер AM/PM одним символом. То есть, AM представляется как A, а PM как P.
tt	Маркер AM/PM двумя символами.

Примеры

```
SELECT FORMAT(CAST('0021-01-02 13:03:02' AS DATETIME), 'd.MM.yyyy hh:mm:ss tt')
SELECT FORMAT(CAST('0000.00.00 07:35:00' AS DATETIME), N'hh.mm');
SELECT FORMAT(CAST('2018-01-01 01:00:00' AS datetime2), N'hh:mm tt');
SELECT FORMAT(CAST('2018-01-01 01:00:00' AS datetime2), N'hh:mm t');
SELECT FORMAT(123456789,'###-##-####') AS [Custom Number];
```

ТИПЫ ДАННЫХ

NitrosBase поддерживает следующие типы данных в SQL запросах:

Тип	Псевдонимы	Описание
BIT	*BOOL*	0 или 1, аналог булевого типа
INT	*INTEGER*, *TINYINT*	4-байтовое целочисленное значение от *-2 147 483 648* до *2 147 483 647*
BIGINT	—	8-байтовое целочисленное значение от *-9 223 372 036 854 775 808* до *9 223 372 036 854 775 807*
CHAR(size)		Строки фиксированной длины. Если строка короче указанной длины, она дополняется пробелами до нужной длины. CHAR без параметров означает CHAR(1) а в функциях CAST и CONVERT CHAR(30)
NCHAR(size)		Способ хранения аналогичен CHAR, но поддерживает хранение данных в формате Юникод (UTF-16). NCHAR без параметров означает NCHAR(1) а в функциях CAST и CONVERT NCHAR(30)
VARCHAR(size)		Строки переменной длины. Переменная этого типа занимает столько места, сколько необходимо для хранения фактической строки. VARCHAR без параметров означает VARCHAR(1) а в функциях CAST и CONVERT VARCHAR(30)
NVARCHAR(size)		Способ хранения аналогичен VARCHAR, но поддерживает хранение данных в формате Юникод (UTF-16). NVARCHAR без параметров означает NVARCHAR(1) а в функциях CAST и CONVERT NVARCHAR(30)
TEXT		Используется для хранения больших текстовых данных
FLOAT(s,d)	*REAL(s,d)*, *DOUBLE(s,d)*	8-байтовое с плавающей точкой. Диапазон точности в отрицательной области: от *-1.79E+308* до *-2.23E-308, в положительной области* от *2.23E-308* до *1.79E+308* Значения аргументов в круглых скобках игнорируются

Тип	Псевдонимы	Описание
DECIMAL(p,s)	*NUMERIC(p,s)*	Число с фиксированной точностью. Занимает от 5 до 17 байт в зависимости от количества цифр после запятой. Принимает параметры p (precision) и s (scale). Параметр p - максимальное количество цифр, которые может хранить число - должен находиться в диапазоне от 1 до 38. По умолчанию 18. Параметр s - максимальное количество цифр после запятой - должен находиться в диапазоне от 0 до значения параметра p. По умолчанию оно 0. См. пример ниже.
DATE	—	См. раздел «Работа с датой и временем»
DATETIME	—	См. раздел «Работа с датой и временем»
DATETIME2(prec)	—	См. раздел «Работа с датой и временем»
VARBINARY(size)	—	Бинарные данные. Пример строкового представления: *0x000000A4*
ROWVERSION	*TIMESTAMP*	Тип данных, который представляет собой генерируемые автоматически уникальные двоичные числа. Этот тип данных используется как правило для фиксации версии записи. Представляет собой увеличивающееся при каждом изменении записи двоичное число размером 8 байт.

Пример 1

Тип *DECIMAL* предназначен для хранения чисел с фиксированной точкой. Он гарантирует заданную точность, в отличие от типов с плавающей точкой (REAL, DOUBLE, FLOAT), в которых накапливается ошибка.

```
CREATE TABLE bill(id_ INT, float_ FLOAT, dec_ DECIMAL(10,2));
INSERT INTO bill(id_, float_, dec_) VALUES(1, 0, 0);
UPDATE bill SET float_=float_+0.01, dec_=dec_+0.01 WHERE id_=1;
UPDATE bill SET float_=float_+0.01, dec_=dec_+0.01 WHERE id_=1;
UPDATE bill SET float_=float_+0.01, dec_=dec_+0.01 WHERE id_=1;
UPDATE bill SET float_=float_+0.01, dec_=dec_+0.01 WHERE id_=1;
UPDATE bill SET float_=float_+0.01, dec_=dec_+0.01 WHERE id_=1;
UPDATE bill SET float_=float_+0.01, dec_=dec_+0.01 WHERE id_=1;
UPDATE bill SET float_=float_+0.01, dec_=dec_+0.01 WHERE id_=1;
UPDATE bill SET float_=float_+0.01, dec_=dec_+0.01 WHERE id_=1;
UPDATE bill SET float_=float_+0.01, dec_=dec_+0.01 WHERE id_=1;
UPDATE bill SET float_=float_+0.01, dec_=dec_+0.01 WHERE id_=1;
SELECT * FROM bill;
```

Данная последовательность запросов прибавляет на каждом шаге по 0.01 к переменным типа: FLOAT и DECIMAL. Результирующий SELECT запрос вернет такие данные:

Id_	Float_	Dec_
1	0.10000000000000007	0.1

Пример 2

Демонстрация работы с ROWVERSION.

```
-- Создать пустую таблицу MyTest

CREATE TABLE MyTest (myKey int PRIMARY KEY, myValue int, RV ROWVERSION);

-- Вставить 2 записи в таблицу.

INSERT INTO MyTest (myKey, myValue) VALUES (1, 0);
INSERT INTO MyTest (myKey, myValue) VALUES (2, 0);

-- После вставки поле RV стало равно:
--   для первой записи RV = 0x0000000000000001
--   для второй записи RV = 0x0000000000000002

-- Модифицировать таблицу

update MyTest set myValue = myValue + 1

-- После модификации поле RV стало равно
--   для первой записи RV = 0x0000000000000003
--   для второй записи RV = 0x0000000000000004
```

РАБОТА С ДАТОЙ И ВРЕМЕНЕМ

Тип DATE

Даты от 0001-01-01 (1 января 0001 года) до 9999-12-31 (31 декабря 9999 года). Поддерживаемые строковые форматы значений:

- YYYY-MM-DD
- YYYY/MM/DD
- YYYY.MM.DD

Пример

```
CREATE TABLE tb (
    fld_date DATE,
    comment VARCHAR (50)
);/**/
INSERT INTO tb VALUES('2022-01-20', 'Date with -')
INSERT INTO tb VALUES('2022/01/20', 'Date with /')
INSERT INTO tb VALUES('2022.01.20', 'Date with .')
```

Тип DATETIME

Дата и время суток, где дата — в том же диапазоне, что и в *DATE*.

Поддерживаемые строковые форматы значений:

```
YYYY-MM-DD hh:mm:ss.nnnnnnn
YYYY/MM/DD hh:mm:ss.nnnnnnn
YYYY.MM.DD hh:mm:ss.nnnnnnn
```

Пример

```
CREATE TABLE tb (
    fld_datetime DATETIME,
    comment VARCHAR (50)
);
INSERT INTO tb VALUES ('2022-01-20 12:37:23.1234567', 'Datetime with -')
INSERT INTO tb VALUES ('2022/01/20 12:37:23.1234567', 'Datetime with /')
INSERT INTO tb VALUES ('2022.01.20 12:37:23.1234567', 'Datetime with .')
```

DATETIME может принимать параметр *Precision*, аналогично типу DATETIME2 (см. описание типа DATETIME2 ниже). При объявлении поля типа DATETIME без параметра, количество знаков в дробной части секунд будет равно 3.

Пример

```
CREATE TABLE [tab1] (
[dt] datetime,
[dt0] datetime(0),
[dt1] datetime(1),
[dt2] datetime(2),
[dt7] datetime(7)
)

INSERT into tab1([dt] ) values ('2005-05-10 23:30:12.1234567');
INSERT into tab1([dt0]) values ('2005-05-10 23:30:12.1234567');
INSERT into tab1([dt1]) values ('2005-05-10 23:30:12.1234567');
INSERT into tab1([dt2]) values ('2005-05-10 23:30:12.1234567');
INSERT into tab1([dt7]) values ('2005-05-10 23:30:12.1234567');
```

Результирующие значения полей:

```
dt  = 2005-05-10 23:30:12.123
dt0 = 2005-05-10 23:30:12
dt1 = 2005-05-10 23:30:12.1
dt2 = 2005-05-10 23:30:12.12
dt7 = 2005-05-10 23:30:12.1234567
```

Тип DATETIME2

Тип DATETIME2 практически идентичен DATETIME за исключением указания в качестве параметра количества знаков в дробной части секунд.

Синтаксис

```
DATETIME2(Precision)
```

Где *Precision* – целое - максимальное количество знаков в дробной части секунд. Значение параметра *Precision* можно задавать в диапазоне от 1 до 7 Если количество цифр в дробной части секунд выше заданной точности, происходит округление до заданного в параметре *Precision*.

Пример

```
CREATE TABLE tb (
    fld_datetime2 DATETIME2(4),
    comment VARCHAR (50)
);

INSERT INTO tb VALUES ('2022-01-20 12:37:23.12', 'Datetime2 with -')
INSERT INTO tb VALUES ('2022/01/20 12:37:23.1234', 'Datetime2 with /')
INSERT INTO tb VALUES ('2022.01.20 12:37:23.123456', 'Datetime2 with .')

select * from tb
```

Результат

```
2022-01-20 12:37:23.1200    Datetime2 with -
2022-01-20 12:37:23.1234    Datetime2 with /
2022-01-20 12:37:23.1235    Datetime2 with .
```

Оператор SET DATEFORMAT

Определяет порядок следования чисел в дате.

Синтаксис

```
SET DATEFORMAT mdy | dmy | ymd | ydm | myd | dym
```

По умолчанию DATEFORMAT = ymd.

Пример

```
CREATE TABLE tb (
    fld_datetime DATETIME,
    comment VARCHAR(50)
);

INSERT INTO tb VALUES('2022-01-20 12:37:23', 'Date format is ymd')
SET DATEFORMAT dmy
INSERT INTO tb VALUES('20-01-2022 12:37:23', 'Date format is dmy')
SET DATEFORMAT mdy
INSERT INTO tb VALUES('01-20-2022 12:37:23', 'Date format is mdy)
```

Результат

```
2022-01-20 12:37:23.1200    Date format is ymd
2022-01-20 12:37:23.1234    Date format is dmy
2022-01-20 12:37:23.1235    Date format is mdy
```

ПОДДЕРЖКА UNICODE

NitrosBase поддерживает символы unicode.

Примеры

```
SELECT UNICODE('мама');
SELECT UNICODE('MAMA');
SELECT UNICODE(0.5);
SELECT UNICODE(55);
```

РАБОТА С ИНДЕКСАМИ

Оператор CREATE INDEX

Синтаксис

```
CREATE [UNIQUE] INDEX index_name ON tbl_name (col_name,...)
```

Пример

```
CREATE INDEX p_ndx_age ON person (age);
```

Оператор DROP INDEX

Синтаксис

```
DROP INDEX index_name [ON tbl_name]
```

Пример

```
DROP INDEX p_ndx_age;
```

Использование индексов

Предположим, для таблицы person (см. раздел «Демонстрационная база данных» документа «Руководство администратора») мы создаем следующие индексы:

```
CREATE INDEX p_age ON person (age)
CREATE INDEX p_name ON person (name)
```

Тогда при вызове следующих операторов SELECT будет использоваться индекс *p_age*.

```
SELECT * FROM person WHERE age = 22
SELECT * FROM person WHERE age >= 22 and age < 24
```

А в следующем запросе используются два индекса: p_*age* и p_*name*

```
SELECT * FROM person WHERE age = 22 and name = 'Lamar'
```

Возможно задавать индексы по нескольким полям, например,

```
CREATE INDEX p_name_lastname ON person (name, lastname)
SELECT * FROM person WHERE name = 'Lamar' and lastname = 'Tanon'
```

При обработке простых запросов эффективное использование индексов рекомендуется только в тех случаях, когда условие, для обработки которого может использоваться индекс, выделяет менее 6% записей.

Для простых сравнений можно явно потребовать использования предварительно созданного индекса. Указание использовать индекс — ключевое слово INDEX после WHERE:

```
SELECT name, lastname, city, model
FROM person p JOIN car c ON c.owner = p.id
WHERE INDEX name = 'Lamar';
```

Индексы также используются при обработке запросов, содержащих выражение JOIN (см. раздел “Выражения FROM и JOIN”).

ТРАНЗАКЦИИ

Транзакция является важнейшим понятием в теории и практике баз данных. Транзакция объединяет некоторую совокупность SQL запросов в единую атомарную сущность. Транзакция начинается с директивы BEGIN TRANSACTION и заканчивается либо директивой COMMIT TRANSACTION, если все запросы на модификацию данных, находящиеся внутри транзакции выполнены успешно, либо директивой ROLLBACK TRANSACTION. Если все изменения внутри транзакции принимаются (COMMIT TRANSACTION), то они фиксируются в базе данных как окончательные. Если в результате выполнения транзакции происходит хотя бы одна ошибка, то срабатывает директива ROLLBACK TRANSACION и должна быть произведена отмена (выполнен откат), все изменения данных будут отменены.

Синтаксис:

```
SET TRANSACTION ISOLATION LEVEL
    { READ UNCOMMITTED
    | READ COMMITTED
    | REPEATABLE READ
    | SNAPSHOT
    | SERIALIZABLE
    }
```

Пример

```
BEGIN TRANSACTION;
DELETE FROM Candidates WHERE CandidateID = 13;
UPDATE JobPositions SET…
UPDATE Managers SET…
COMMIT TRANSACTION;
```

Уровни изоляции транзакций

NitrosBase поддерживает все базовые уровни изоляции транзакций:

Уровень изоляции	Комментарий
READ UNCOMMITTED	Низший уровень изоляции. Если несколько параллельных транзакций пытаются изменять одну и ту же строку таблицы, то в окончательном варианте строка будет иметь значение, определенное всем набором успешно выполненных транзакций. При этом возможно считывание не только логически несогласованных данных, но и данных, изменения которых ещё не зафиксированы.

READ COMMITTED	На этом уровне обеспечивается защита от чернового, «грязного» чтения, тем не менее, в процессе работы одной транзакции другая может быть успешно завершена и сделанные ею изменения зафиксированы. В итоге первая транзакция будет работать с другим набором данных.
REPEATABLE READ	Уровень, при котором читающая транзакция «не видит» изменения данных, которые были ею ранее прочитаны. При этом никакая другая транзакция не может изменять данные, читаемые текущей транзакцией, пока та не окончена.
SNAPSHOT	На этом уровне транзакция видит то состояние данных, которое было зафиксировано до её запуска, а также изменения, внесённые ею самой, то есть ведёт себя так, как будто получила при запуске моментальный снимок данных БД и работает с ним.
SERIALIZABLE	Самый высокий уровень изолированности; транзакции полностью изолируются друг от друга, каждая выполняется так, как будто параллельных транзакций не существует.

Уровень изоляции транзакций устанавливается директивой SET TRANSACTION ISOLATION LEVEL.

Пример:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

ПСЕВДОТАБЛИЦЫ

Работа со схемой данных

Доступ к информации о схеме данных нужен как правило приложениям. При работе с данными из NBAdmin объекты схемы данных доступны непосредственно из пользовательского интерфейса. Подробную информацию о функциях доступа к схеме данных, а также примеры работы с объектами схемы данных можно получить, в разделе «Получение информации о схеме базы данных» документа "Программный интерфейс C/C++".

Работа с Edge-таблицами

На примере операций с данными из демонстрационной базы данных (см. раздел «Демонстрационная база данных» документа «Руководство администратора»).

Создание таблицы:

```
CREATE TABLE friends AS EDGE person person;
```

Заполнение таблицы:

```
INSERT INTO friends VALUES ( 'person22', 'person1022' );
INSERT INTO friends VALUES ( 'person22', 'person1023' );
```

Получение данных:

```
SELECT p1.id, p2.id
  FROM person p1
  JOIN friends f ON p1.id = f.fromid
  JOIN person p2 ON f.toid = p2.id
WHERE p1.id = 'person22';
```

Обновление данных:

```
UPDATE friends SET fromid = 'person7', toid = 'person67'
WHERE fromid = 'person7' AND toid = 'person1007'

UPDATE car SET owner = 'person23' WHERE id = 'car4022',
```

Удаление данных:

```
DELETE FROM friends WHERE fromid = 'person22' AND toid = 'person1022';
DELETE FROM owner   WHERE toid   = 'person22',
DELETE FROM owner   WHERE fromid = 'car4021',
```

Более подробно об Edge-таблицах и работе с ними см. документ " Работа с графами".

ОТЛИЧИЯ ДИАЛЕКТА SQL NITROSBASE ОТ MS SQL SERVER

Различия преобразования типов и округления в операторе UNION

Следующий SQL оператор даст различные результаты на разных СУБД

```
SELECT CAST (1 AS DECIMAL(38,3))
UNION ALL
(
SELECT CAST (1.444456 AS DECIMAL(38,6))
UNION
SELECT CAST (1 AS DECIMAL(38,4))
)
```

Результат

MS SQL	NitrosBase
1.000 1.000 1.445	1.444 1.000 1.000

Отличия проявляются из-за различия в преобразовании типов и, соответственно, округления.
MS SQL и NitrosBase преобразуют тип DECIMAL к общему при выполнении операций, но делают это по-разному. MS SQL делает это дважды:

1. 1.444456 приводится к типу DECIMAL(38,4) и получает 1.4445
2. 1.4445 приводится к типу DECIMAL(38,3) и получает 1.445

NitrosBase сразу вычисляет конечный тип и 1.444456 приводится к типу DECIMAL(38,3) и получает 1.444

Различие типа возвращаемого значения функцией POWER

У NitrosBase в функции POWER первый аргумент имеет тип double, и возвращаемое значение также имеет тип double. Если же первым аргументом стоит число отличного типа, то оно преобразуется в тип double, и возвращается число типа double.

У MS SQL тип возвращаемого значения соответствует типу аргумента. В случае, если первый аргумент целого типа, то возвращаемое значение тоже целого типа.

Например, следующий оператор вернет существенно различные значения под NitrosBase и MS SQL:

```
SELECT POWER(123, 0.2) ORDER BY 1;
```

Под NitrosBase данный оператор вернет значение **2,618068601888982**, под MS SQL вернется **2**.

Отличия в обработке чисел с плавающей точкой

Числа с плавающей точкой в NitrosBase всегда занимают 8 бит. Диапазон значений чисел с плавающей точкой совпадает с оным у типа FLOAT MS SQL.

Типы FLOAT(size,d), REAL(size,d) и DOUBLE(size,d) в NitrosBase являются синонимами. Значения атрибутов в круглых скобках игнорируются, число загружается в поле или переменную целиком, если умещается в максимальный диапазон FLOAT (как MS SQL, так и NitrosBase), в противном случае выдается ошибка

Типы полей DATETIME и DATETIME2

В NitrosBase DATETIME и DATETIME2 почти идентичны по поведению. Так же как DATETIME2, в отличие от MS SQL можно задавать DATETIME с параметром DATETIME(scale).

Значение scale по умолчанию для DATETIME = 3.

Значение scale по умолчанию для DATETIME2 = 7.

Различия в реакции на избыточные знаки в дробной части типа DATETIME

Следующий пример вызовет различную реакцию на MS SQL и NitrosBase:

```
SELECT CAST('2012-12-07 14:59:59.55555' AS DATETIME);
```

MS SQL вернет ошибку, поскольку в строке первого аргумента указано большее количество знаков после точки, чем это поддерживается типом DATETIME. NitrosBase присвоит результат округления до миллисекунд: **2012-12-07 14:59:59.556**

Особенности поведения функции DATETIME2FROMPARTS

На примере оператора

```
SELECT DATETIME2FROMPARTS(2000,1,1,0,0,0,1,2);
```

Последний параметр precision, (в данном случае 2).

Если precision константа - то результат функции имеет тип datetime2(precision), и результатом будет **2000-01-01 0:0:0.01**

Если precision не константа - то результат функции имеет тип datetime2(7), а precision используется для правильного вычисления fraction.

Например

```
DATETIME2FROMPARTS(2000,1,1,0,0,0, 1, func())
```

Если func() возвращает 1, то результат будет **2000-01-01 0:0:0.1000000**.

Если func() возвращает 5 то результат будет **2000-01-01 0:0:0.0000100**.

Различия в поведении автоинкрементных полей

При наличии автоинкрементного поля NitrosBase и MS SQL ведут себя по-разному. В MS SQL если операция вставки записи (INSERT) вернула ошибку, и запись не вставилась, внутренний счетчик все равно меняет значение на следующее, и при следующей успешной вставке значение будет увеличено еще раз. Это приводит к фрагментированию последовательности значений автоинкрементного поля. У NitrosBase в этом случае значение автоинкрементного поля не меняется до реальной вставки записи. Например, в результате следующей последовательности SQL операторов значение поля c12 будет различным для MS SQL и NitrosBase.

```
Drop TABLE IF EXISTS tab1;

CREATE TABLE tab1 (
c11 VARCHAR(712) NOT NULL,
c12 INT IDENTITY(5,10) NOT NULL,
c13 FLOAT NOT NULL
);

INSERT INTO tab1 (c11, c13) VALUES (N'Один', NULL);

/* WARNING!!! */
/* MS SQL: Не удалось вставить значение NULL в столбец "c13", таблицы "tmp.dbo.tab1;
в столбце запрещены значения NULL. Ошибка в INSERT. Выполнение данной инструкции был
о прервано. */
/* NitrosBase: Field c13 cannot be NULL*/

INSERT INTO tab1 (c11, c13) VALUES (N'Два', 0.411715);

SELECT c12 FROM tab1;
```

Результат:

```
MS SQL: c12 = 15
NitrosBase: c12 = 5
```

Различия в поведении ORDER BY при отборе одной записи

Когда запрос возвращает 1 запись, NitrosBase не отрабатывает ORDER BY. Например, поведение баз различно:

```
SELECT 1, 2, 3 ORDER BY 4;
```

Результат:

```
MS SQL: “ERROR: The ORDER BY position number 4 is out of range of the number of
         items in the select list.”
NB: 1, 2, 3
```