

# Лабораторная работа №1

## Проектирование базы данных

### Задание

Ниже приведен список предметных областей.

- Вариант 1. Учет регистрации автомобилей в ГИБДД
- Вариант 2. Кинотеатр
- Вариант 3. Продажа авиабилетов
- Вариант 4. Продажа автомобилей
- Вариант 5. Аптека
- Вариант 6. Компания по продаже бензина
- Вариант 7. Туристическая фирма
- Вариант 8. Библиотека
- Вариант 9. Театр
- Вариант 10. Банк
- Вариант 11. Гостиница
- Вариант 12. Видеопрокат
- Вариант 13. Обслуживание банковских карт
- Вариант 14. Запись на прием к врачу
- Вариант 15. Ипподром
- Вариант 16. Баскетбольный клуб
- Вариант 17. Продажа железнодорожных билетов

1. Для выбранной предметной области построить диаграмму базы данных, с указанием первичных ключей, связей между таблицами. Пример приведен на рисунке 1.

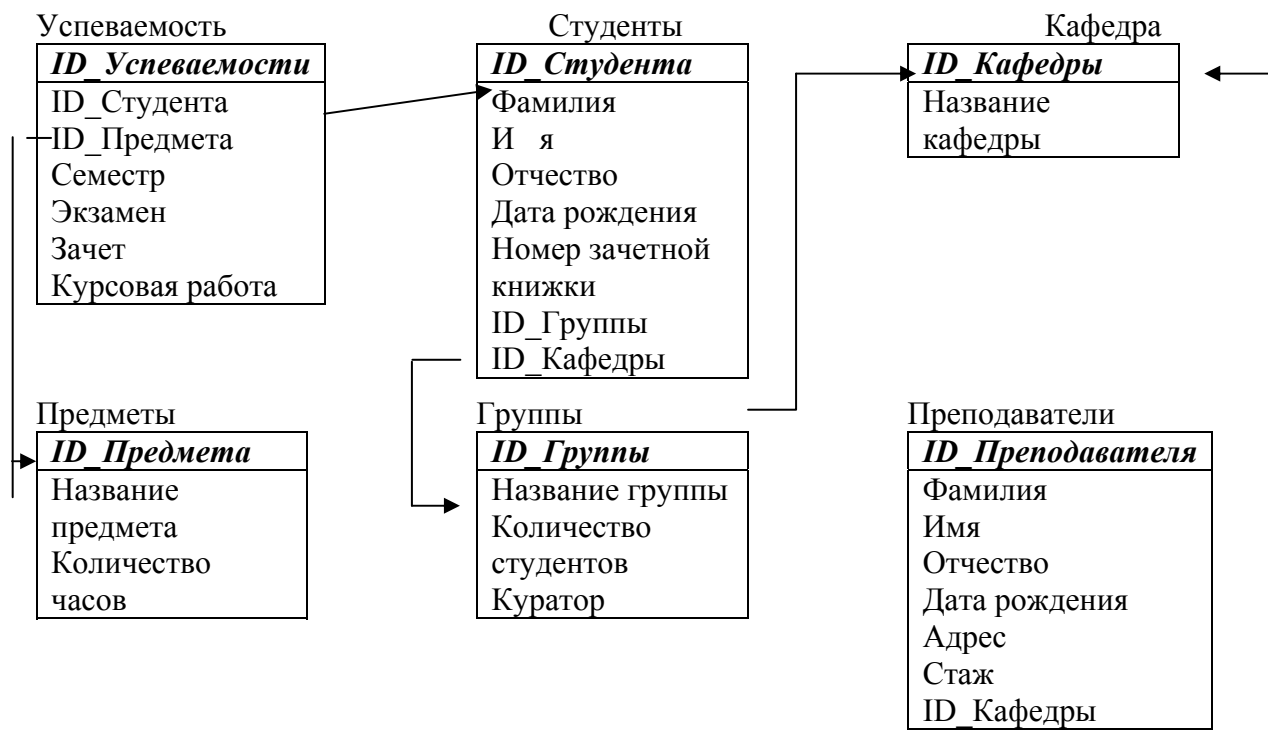


Рисунок 1 – Диаграмма базы данных *Institute*

2. Составить таблицу сущностей по образцу, приведенному в таблице 1.

Таблица 1 – Таблица сущностей

Название сущности	Идентификатор сущности	Назначение сущности
Кафедра	<b><i>Kafedra</i></b>	содержит сведения о кафедрах
Предметы	<b><i>Subject</i></b>	содержит сведения о предметах
Преподаватели	<b><i>Teachers</i></b>	содержит сведения о преподавателях
Группы	<b><i>Groups</i></b>	содержит сведения об учебных группах
Студенты	<b><i>Student</i></b>	содержит сведения о студентах
Успеваемость	<b><i>Progress</i></b>	содержит сведения об успеваемости студентов

3. Для каждой сущности базы данных привести описание атрибутов. Заполнить таблицу по образцу, приведенному в таблице 2.

Таблица 2 – Сущность ***Студент***

Имя столбца	Содержательное описание	Тип данных	Размерность	Область допустимых значений	Возможность значения Null	Роль	Пример	Примечание
ID_Stud	Номер студента	целый	4	0001-9999	нет	РК	1736	
Surname	Фамилия	символьный	30	‘А-я’, ‘-’	нет		Иванов	
First_Name	Имя	символьный	15	‘А-я’	нет		Сергей	
Last_Name	Отчество	символьный	20	‘А-я’	да		Петрович	
Data	Дата рождения	Дата/время	8	01.01.1980 – 31.12.1996	нет		12.10.1995	
Nomer	Номер зачетной книжки	символьный	7	‘0-9’, ‘А-Я’	нет	АК	11ВП112	
ID_Group	ID_Группы	целый	4	0001-9999	нет		113	
ID_Kaf	ID_Кафедры	символьный	3	001-999	нет		3	

В столбце **Область допустимых значений** указывается:

- для символьных полей – набор допустимых символов. Например, если атрибут может принимать значение, состоящее из любых букв русского алфавита, то область допустимых значений для этого атрибута будет задана в виде ‘А-я’.
- для числовых полей – диапазон возможных значений. Например, 001–999.
- Для полей типа дата/время – диапазон возможных значений. Например, 01.01.1980 – 31.12.1996

В столбце **Примечание** следует указать значение по умолчанию или другие особые условия.

В столбце **Роль** указываются значения:

- РК – для первичного ключа,
- АК – для альтернативного (потенциального) ключа,
- FK – для внешнего ключа.

Отчет по лабораторной работе должен содержать схему базы данных и указанный выше набор таблиц.

## Лабораторная работа №2

### Создание базы данных

#### Задание

1. С помощью команды **CREATE DATABASE** создать базу данных для заданной предметной области. Подробное описание всех команд приведено в [1].
2. С помощью команды **CREATE TABLE** создать все таблицы и ввести *все* ограничения, разработанные на этапе проектирования.
3. Добавить в БД таблицу, содержащую несколько столбцов. Выполнить модификацию характеристик столбцов, добавить столбец в таблицу, удалить столбец из таблицы (с помощью команды **ALTER TABLE**)
4. Отчет по лабораторной работе должен содержать команды SQL для создания БД и всех таблиц БД, команды модификации, а также диаграмму БД, построенную в SQL Server 2008. При создании таблиц следует задать ограничения целостности, разработанные при выполнении лабораторной работы №1.

### Работа в среде Microsoft SQL SERVER 2008

1. Для формирования запросов в Microsoft SQL Server 2008 нужно запустить SQL Server Management Studio либо с помощью ярлыка на рабочем столе либо из главного меню (**Пуск / Все программы / Microsoft SQL Server 2008 / SQL Server Management Studio**). Microsoft SQL Server Management Studio 2008 – это интегрированная среда для доступа, настройки, управления, администрирования и разработки всех компонентов SQL Server.
2. На рис. 2 представлено главное окно (Соединение с сервером) программы SQL Server Management Studio при первом запуске. Здесь следует выбрать компонент базы данных, с которым требуется установить соединение. Обычно требуется подключиться к Database Engine (Ядро базы данных). В поле *Server name* выбираем сервер, заданный по умолчанию, – *local*. Далее выбираем тип аутентификации – проверку подлинности (в нашем случае это проверка подлинности *Windows*). После задания всех параметров выполняем соединение с сервером.

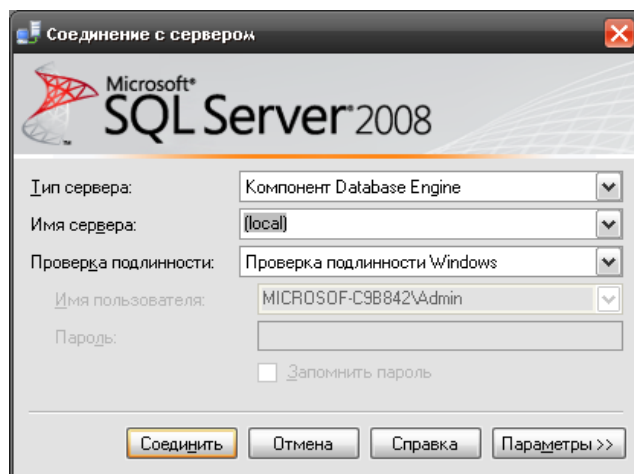


Рисунок 2 – главное окно программы SQL Server Management Studio

3. После соединения с сервером появляется основное рабочее окно среда (Рис. 3):

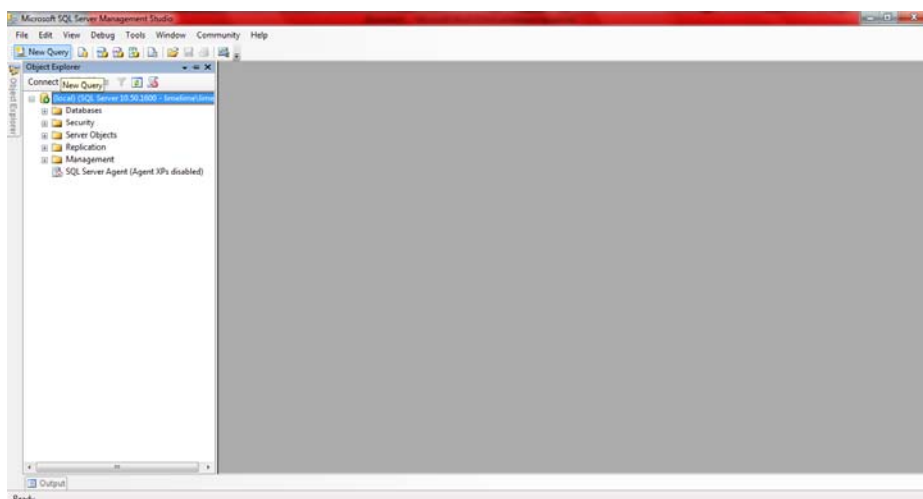


Рисунок 3 – главное окно программы SQL Server Management Studio

К экземпляру сервера можно подключиться, нажав на кнопку Connect в верхней части панели Object Explorer (Обозреватель объектов). После этого становятся доступны следующие ресурсы, организованные в виде узлов иерархического дерева объектов:

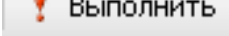
- Databases (Базы данных). Управляет системными базами данных, включая master и model (узел System Databases), а также пользовательскими базами данных и моментальными снимками БД (узел Database Snapshots)
- Security (Безопасность). Управляет учетными записями (узел Logins), ролями сервера (узел Server Roles), связанными серверами и хранимыми учетными данными (узел Credentials)

- Server Objects (Объекты сервера). Настраивает устройства резервного копирования, конечные точки HTTP (узел Endpoints), связанные серверы (узел Linked Servers) и триггеры (узел Triggers).
- Replication (Репликация). Настраивает распространение данных, обновляет пароли репликации и запускает утилиту Replication Monitor (Монитор репликации)
- Management (Управление). Настраивает планы обслуживания (узел Maintenance Plans), журналы SQL Server (узел SQL Server Logs), полнотекстовый поиск (узел Full-Text Search), координатор распределенных транзакций (узел Distributed Transaction Coordinator), монитор деятельности (Activity Monitor) и почту БД (узел Database Mail).
- Notification Services (Службы уведомлений). Регистрирует экземпляры службы уведомлений, а также управляет их списками и отменяет регистрацию
- SQL Server Agent (Агент SQL Server). Настраивает задания (узел Jobs), оповещения (узел Alerts), операторов (узел Operations), представителей (узел Proxies) и журналы ошибок (узел Error Logs).

Для создания запроса к базе данных следует нажать на кнопку *New Query*, создавая тем самым новый запрос. Для **создания базы данных** нужно ввести следующую команду:

***CREATE DATABASE <имя\_базы\_данных>***

Имя новой базы данных не должно совпадать с названием уже существующих баз.

После этого запрос следует выполнить (кнопка  в левом верхнем углу окна либо F5 на клавиатуре).

В нижней части окна на вкладке «Сообщения» появится подтверждение успешного создания базы данных (Рисунок 4):

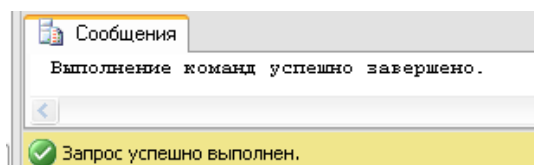



Рисунок 4 - подтверждение успешного создания базы данных

Для того, чтобы убедиться, что база данных создана, в обозревателе объектов откроем контекстное меню для пункта «Базы данных» и выберем «Обновить». Затем откроем пункт «Базы данных» и найдем название созданной базы данных.

Следующий этап работы – **создание таблиц базы данных**. Для продолжения работы следует указать базу данных, в которой будут создаваться требуемые таблицы. Для этого

следует выбрать нашу БД в выпадающем списке (левее кнопки ) либо, прежде чем писать команды создания таблиц, ввести команду *USE имя\_базы\_данных*. После этого все действия будут выполняться над этой выбранной БД.

Следует помнить, что в первую очередь создаются таблицы, в которых отсутствуют внешние ключи.

### 1. Создание таблиц, входящих в БД. Задание для их атрибутов типов данных и ограничений целостности, соответствующих смыслу таблиц.

Таблицы БД создаются с помощью команды `CREATE TABLE`. Эта команда создает пустую таблицу, то есть таблицу, не имеющую строк. Значения в эту таблицу вводятся с помощью команды `INSERT`. Команда `CREATE TABLE` определяет имя таблицы и множество поименованных столбцов в указанном порядке. Для каждого столбца должен быть определен тип и размер. Тип данных, для которого обязательно должен быть указан размер, — это `CHAR`. Реальное количество символов, которое может находиться в поле, изменяется от нуля (если в поле содержится `NULL`-значение) до заданного в `CREATE TABLE` максимального значения.

#### Ограничения на множество допустимых значений данных

Ограничения (***CONSTRAINTS***) являются частью определения таблицы.

При создании (изменении) таблицы могут быть определены ограничения на вводимые значения. В этом случае `SQL` будет отвергать любое из них при несоответствии заданным критериям. Ограничения могут быть статическими, ограничивающими значения или диапазон значений, вставляемых в столбец (`CHECK`, `NOT NULL`). Они могут иметь связь со всеми значениями столбца, ограничивая новые строки значениями, которые не содержатся в столбцах или их наборах (уникальные значения, первичные ключи). Ограничения могут также определяться связью со значениями, находящимися в другой таблице, допуская, например, вставку в столбец только тех значений, которые в данный момент содержатся также в другом столбце другой или этой же таблицы (внешний ключ). Эти ограничения носят динамический характер.

#### Ограничение `NOT NULL`

Чтобы запретить возможность использования в поле `NULL`-значений, можно при создании таблицы командой `CREATE TABLE` указать для соответствующего столбца ключевое слово `NOT NULL`. `NULL` — это специальный маркер, обозначающий тот факт, что поле пусто. Но он полезен не всегда. Первичные ключи, например, в принципе не должны содержать `NULL`-значений (быть пустыми), поскольку это нарушило бы требование уникальности первичного ключа (более строго — функциональную зависимость атрибутов таблицы от первичного ключа). Во многих других случаях также необходимо, чтобы поля обязательно содержали определенные значения. Если ключевое слово `NOT NULL` размещается непосредственно после типа данных (включая размер) столбца, то любые попытки оставить значение поля пустым (ввести в поле `NULL`-значение) будут отвергнуты системой.

Например, для того, чтобы в определении таблицы `STUDENT` запретить использование `NULL`-значений для столбцов `STUDENT_ID`, `SURNAME` и `NAME`, можно записать следующее:

```
CREATE TABLE STUDENT  
(STUDENT_ID INTEGER NOT NULL,  
SURNAME CHAR (25) NOT NULL,
```

```

NAME CHAR (10) NOT NULL,
STIPEND INTEGER,
KURS INTEGER,
CITY CHAR (15),
BIRTHDAY DATE,
UNIV_ID INTEGER);

```

Важно помнить: если для столбца указано NOT NULL, то при использовании команды INSERT обязательно должно быть указано конкретное значение, вводимое в это поле. При отсутствии ограничения NOT NULL в столбце значение может отсутствовать, если только не указано значение столбца по умолчанию (DEFAULT). Если при создании таблицы ограничение NOT NULL не было указано, то его можно указать позже, используя команду ALTER TABLE. Однако для того, чтобы для вновь вводимого с помощью команды ALTER TABLE столбца можно было задать ограничение NOT NULL, таблица, в которую добавляется столбец, должна быть пустой.

### **Уникальность как ограничение на столбец**

Иногда требуется, чтобы все значения, введенные в столбец, отличались друг от друга. Например, этого требуют первичные ключи. Если при создании таблицы для столбца указывается ограничение UNIQUE, то база данных отвергает любую попытку ввести в это поле какой-либо строки значение, уже содержащееся в том же поле другой строки. Это ограничение применимо только к тем полям, которые были объявлены NOT NULL.

```

CREATE TABLE STUDENT
(STUDENT_ID INTEGER NOT NULL UNIQUE,
  SURNAME CHAR (25) NOT NULL,
  NAME CHAR (10) NOT NULL,
  STIPEND INTEGER,
  KURS INTEGER,
  CITY CHAR (15),
  BIRTHDAY DATE,
  UNIV_ID INTEGER);

```

Объявляя поле STUDENT\_ID уникальным, можно быть уверенным, что в таблице не появится записей для двух студентов с одинаковыми идентификаторами. Столбцы, отличные от первичного ключа, для которых требуется поддерживать уникальность значений, называются возможными ключами или уникальными ключами (CANDIDATE KEYS или UNIQUE KEYS).

### **Присвоение имен ограничениям**

Ограничениям таблиц можно присваивать уникальные имена. Преимущество явного задания имени ограничения состоит в том, что в этом случае при выдаче системой сообщения о нарушении установленного ограничения будет указано его имя, что упрощает обнаружение ошибок.

```

CREATE TABLE EXAM_MARKS
(EXAM_ID INTEGER NOT NULL,
  STUDENT_ID INTEGER NOT NULL,
  SUBJ_ID INTEGER NOT NULL,
  MARK CHAR (1),
  EXAM_DATE DATE NOT NULL,
  CONSTRAINT STUD_SUBJ_CONSTR
  UNIQUE (STUDENT_ID, EXAM_DATE);

```

В этом запросе STUD\_SUBJ\_CONSTR — это имя, присвоенное указанному ограничению таблицы.



### Ограничение первичных ключей

Первичные ключи таблицы — это специальные случаи комбинирования ограничений UNIQUE и NOT NULL. Первичные ключи имеют следующие особенности:

- таблица может содержать только один первичный ключ;
- внешние ключи по умолчанию ссылаются на первичный ключ таблицы;
- первичный ключ является идентификатором строк таблицы (строки, однако, могут идентифицироваться и другими способами).

Улучшенный вариант создания таблицы STUDENT1 с объявленным первичным ключом имеет теперь следующий вид:

```
CREATE TABLE STUDENT  
(STUDENT_ID INTEGER PRIMARY KEY,  
  SURNAME CHAR (25) NOT NULL,  
  NAME CHAR (10) NOT NULL,  
  STIPEND INTEGER,  
  KURS INTEGER,  
  CITY CHAR (15),  
  BIRTHDAY DATE,  
  UNIV_ID INTEGER);
```

### Составные первичные ключи

Ограничение PRIMARY KEY может также быть применено для нескольких полей, составляющих уникальную комбинацию значений — составной первичный ключ. Рассмотрим таблицу EXAM\_MARKS. Очевидно, что ни к полю идентификатора студента (STUDENT\_ID), ни к полю идентификатора предмета обучения (EXAM\_ID) по отдельности нельзя предъявить требование уникальности. Однако для того, чтобы в таблице не могли появиться разные записи для одинаковых комбинаций значений полей STUDENT\_ID И EXAM\_ID (конкретный студент на конкретном экзамене не может получить более одной оценки), имеет смысл объявить уникальной комбинацию этих полей. Для этого мы можем применить ограничение таблицы PRIMARY KEY, объявив пару EXAM\_ID И STUDENT\_ID первичным ключом таблицы.

```
CREATE TABLE NEW_EXAM_MARKS  
(STUDENT_ID INTEGER NOT NULL,  
  SUBJ_ID INTEGER NOT NULL,  
  MARK INTEGER,  
  DATA DATE,  
  CONSTRAINT EX_PR_KEY PRIMARY KEY (EXAM_ID, STUDENT_ID));
```

### Проверка значений полей

Ограничение CHECK позволяет определять условие, которому должно удовлетворять вводимое в поле таблицы значение, прежде чем оно будет принято. Любая попытка обновить или заменить значение поля такими, для которых предикат, задаваемый ограничением CHECK, имеет значение ложь, будет отвергаться.

Рассмотрим таблицу STUDENT. Значение столбца STIPEND в этой таблице выражается десятичным числом. Наложим на значения этого столбца ограничение — величина размера стипендии должна быть меньше 200.

Соответствующий запрос имеет следующий вид:

```
CREATE TABLE STUDENT  
(STUDENT_ID INTEGER PRIMARY KEY,  
  SURNAME CHAR (25) NOT NULL,  
  NAME CHAR (10) NOT NULL,
```

```
STIPEND  INTEGER CHECK (STIPEND < 200),  
KURS    INTEGER,  
CITY    CHAR (15),  
BIRTHDAY DATE,  
UNIV_ID INTEGER);
```

### **Проверка ограничивающих условий с использованием составных полей**

При необходимости можно включить более одного поля в ограничивающее условие.

Предположим, что ограничение на размер стипендии (меньше 200) должно распространяться только на студентов, живущих в Воронеже. Это можно указать в запросе со следующим табличным ограничением CHECK:

```
CREATE TABLE STUDENT  
(STUDENT_ID INTEGER PRIMARY KEY,  
SURNAME   CHAR(25) NOT NULL,  
NAME     CHAR (10) NOT NULL,  
STIPEND  INTEGER,  
KURS    INTEGER,  
CITY    CHAR(15),  
BIRTHDAY DATE,  
UNIV_ID  INTEGER UNIQUE,  
CHECK (STIPEND < 200 AND CITY = 'Воронеж'));
```

или в несколько другой записи:

```
CREATE TABLE STUDENT  
(STUDENT_ID INTEGER PRIMARY KEY,  
SURNAME   CHAR(25) NOT NULL,  
NAME     CHAR (10) NOT NULL,  
STIPEND  INTEGER,  
KURS    INTEGER,  
CITY    CHAR (15),  
BIRTHDAY DATE,  
UNIV_ID  INTEGER UNIQUE,  
CONSTRAINT STUD_CHECK CHECK (STIPEND < 200 AND CITY = 'Воронеж'));
```

### **Установка значений по умолчанию**

В SQL имеется возможность при вставке в таблицу строки, не указывая значений некоторого поля, определять значение этого поля по умолчанию. Наиболее часто используемым значением по умолчанию является NULL. Это значение принимается по умолчанию для любого столбца, для которого не было установлено ограничение NOT NULL.

Значение поля по умолчанию указывается в команде CREATE TABLE с помощью ключевого слова

```
DEFAULT <значение по умолчанию>.
```

Строго говоря, опция DEFAULT не имеет ограничительного свойства, так как она не ограничивает значения, вводимые в поле, а просто конкретизирует значение поля в случае, если оно не было задано.

Предположим, что основная масса студентов, информация о которых находится в таблице STUDENT, проживает в Воронеже. Чтобы при задании атрибутов не вводить для большинства студентов название города 'Воронеж', можно установить его как значение поля CITY по умолчанию, определив таблицу STUDENT следующим образом:

```
CREATE TABLE STUDENT
```

```

(STUDENT_ID INTEGER PRIMARY KEY,
SURNAME     CHAR (25) NOT NULL,
NAME        CHAR (10) NOT NULL,
STIPEND     INTEGER CHECK (STIPEND < 200),
KURS        INTEGER,
CITY        CHAR (15) DEFAULT 'Воронеж',
BIRTHDAY    DATE,
UNIV_ID     INTEGER);

```

Другая цель практического применения задания значения по умолчанию — это использование его как альтернативы для NULL. Присутствие NULL в качестве возможных значений поля существенно усложняет интерпретацию операций сравнения, в которых участвуют значения таких полей, поскольку NULL представляет собой признак того, что фактическое значение поля неизвестно или неопределенно. Следовательно, строго говоря, сравнение с ним любого конкретного значения в рамках двузначной булевой логики является некорректным, за исключением специальной операции сравнения `is NULL`, которая определяет, является ли содержимое поля каким-либо значением или оно отсутствует. Действительно, каким образом в рамках двузначной логики ответить на вопрос, истинно или ложно условие `CITY = 'Воронеж'`, если текущее значение поля `CITY` неизвестно (содержит NULL)?

Во многих случаях использование вместо NULL значения, подставляемого в поле по умолчанию, может существенно упростить использование значений поля в предикатах.

Например, можно установить для столбца опцию `NOT NULL`, а для неопределенных значений числового типа установить значение по умолчанию «равно нулю», или для полей типа `CHAR` — пробел, использование которых в операциях сравнения не вызывает никаких проблем.

## 2. Модификация структуры таблицы

(добавление и удаление нового столбца, добавление ограничений, изменение типа данных, ограничений целостности)

Для модификации структуры и параметров существующей таблицы используется команда `ALTER TABLE`. Синтаксис команды `ALTER TABLE` для добавления столбцов в таблицу имеет вид

```

ALTER TABLE <ИМЯ ТАБЛИЦЫ> ADD (<ИМЯ СТОЛБЦА> <ТИП ДАННЫХ> <раз-
мер>);

```

По этой команде для существующих в таблице строк добавляется новый столбец, в который заносится NULL-значение. Этот столбец становится последним в таблице. Можно добавлять несколько столбцов, в этом случае их определения в команде `ALTER TABLE` разделяются запятой.

Возможно изменение описания столбцов. Часто это связано с изменением размеров столбцов, добавлением или удалением ограничений, накладываемых на их значения. Синтаксис команды в этом случае имеет вид

```

ALTER TABLE <ИМЯ ТАБЛИЦЫ> ALTER COLUMN <ИМЯ СТОЛБЦА> <ТИП
ДАННЫХ> <размер/точность>;

```

Следует иметь в виду, что модификация характеристик столбца может осуществляться не в любом случае, а с учетом следующих ограничений:

- изменение типа данных возможно только в том случае, если столбец пуст;
- для незаполненного столбца можно изменять размер/точность. Для заполненного столбца размер/точность можно увеличить, но нельзя понизить;
- ограничение `NOT NULL` может быть установлено, если ни одно значение в столбце не содержит NULL. Опцию `NOT NULL` всегда можно отменить;

- разрешается изменять значения, установленные по умолчанию.

Синтаксис команды ALTER TABLE для удаления столбцов в таблице имеет вид  
***ALTER TABLE <ИМЯ ТАБЛИЦЫ> DROP COLUMN (<ИМЯ СТОЛБЦА>);***

**3. Построение диаграммы базы данных.** Для этого в Обозревателе объектов следует найти созданную базу, в ней открыть контекстное меню для папки *Диаграммы баз данных* и выбрать *Создать диаграмму базы данных*. Появится окошко *Добавить таблицы*, в котором следует выбрать все таблицы базы данных. Последовательно нажимая на кнопку *Добавить*, получим диаграмму созданной базы (рис. 5).

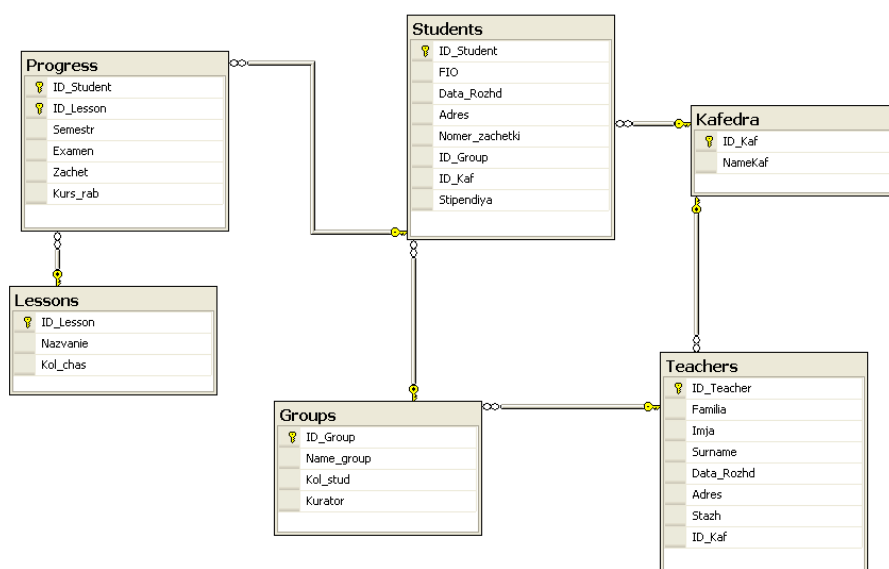


Рисунок 5 – диаграмма базы данных

### Лабораторная работа №3

#### Команды манипулирования данными

В SQL для выполнения операций ввода данных в таблицу, их изменения и удаления предназначены три команды языка манипулирования данными (DML). Это команды ***INSERT*** (вставить), ***UPDATE*** (обновить), ***DELETE*** (удалить).

1. Команда ***INSERT*** осуществляет вставку в таблицу новой строки. В простейшем случае она имеет вид:

***INSERT INTO <имя таблицы> VALUES (<значение>, <значение>);***

При такой записи указанные в скобках после ключевого слова ***VALUES*** значения вводятся в поля добавленной в таблицу новой строки в том порядке, в котором соответствующие столбцы указаны при создании таблицы, то есть в операторе ***CREATE TABLE***.

Например, ввод новой строки в таблицу ***STUDENT*** может быть осуществлен следующим образом:

***INSERT INTO STUDENT***

***VALUES (111, 'Петров', 'Александр', 'Иванович', '1994-05-05', 23, 3, 5);***

Если в какое-либо поле необходимо вставить NULL-значение, то оно вводится как обычное значение:

***INSERT INTO STUDENT***

***VALUES (101, 'Иванов', NULL, 200, 3, 'Москва', '6/10/1979', 15);***

В случаях, когда необходимо ввести значения полей в порядке, отличном от порядка столбцов, заданного командой ***CREATE TABLE***, или требуется ввести значения не во все столбцы, следует использовать следующую форму команды ***INSERT***:

***INSERT INTO STUDENT (ID\_STUDENT, SURNAME, FIRST\_NAME,)***

***VALUES (111, 'Иванов', 'Александр');***

Столбцам, наименования которых не указаны в приведенном в скобках списке, автоматически присваивается значение по умолчанию, если оно назначено при описании таблицы (команда ***CREATE TABLE***), либо значение ***NULL***.

С помощью команды ***INSERT*** можно извлечь значение из одной таблицы и разместить его в другой, например, запросом следующего вида:

***INSERT INTO STUDENT1***

***SELECT \****

***FROM STUDENT***

***WHERE ID\_Group = 23;***

При этом таблица **STUDENT1** должна быть предварительно создана командой **CREATE TABLE** и иметь структуру, идентичную таблице **STUDENT**.

2. Удаление строк из таблицы осуществляется с помощью команды **DELETE**.

Следующее выражение удаляет все строки таблицы **STUDENT**.

**DELETE FROM STUDENT;**

В результате таблица становится пустой (после этого она может быть удалена командой **DROP TABLE**).

Для удаления из таблицы сразу нескольких строк, удовлетворяющих некоторому условию, можно воспользоваться предложением **WHERE**, например:

**DELETE FROM STUDENT**

**WHERE STUDENT\_ID = 111;**

Можно удалить группу строк:

**DELETE FROM STUDENT1**

**WHERE CITY = 'Москва';**

3. Команда **UPDATE** позволяет изменять значения некоторых или всех полей в существующей строке или строках таблицы. Например, чтобы для всех предметов, сведения о которых находятся в таблице **SUBJECT**, изменить количество часов на значение 64, можно использовать конструкцию:

**UPDATE SUBJECT**

**SET HOURS = 64;**

Для указания конкретных строк таблицы, значения полей которых должны быть изменены, в команде **UPDATE** можно использовать предикат, указываемый в предложении **WHERE**.

**UPDATE SUBJECT**

**SET HOURS = 64**

**WHERE ID\_ SUBJECT=12;**

В результате выполнения этого запроса будет изменено количество часов только у предмета с **ID\_ SUBJECT**, равным 12.

Команда **UPDATE** позволяет изменять не только один, но и множество столбцов. Для указания конкретных столбцов, значения которых должны быть модифицированы, используется предложение **SET**.

Например, наименование предмета обучения 'Математика' (для него **ID\_ SUBJECT=12**) должно быть заменено на название 'Дискретная математика', при этом идентификационный

номер необходимо сохранить, но в соответствующие поля строки таблицы ввести новые данные об этом предмете обучения. Запрос будет выглядеть следующим образом:

**UPDATE SUBJECT**

**SET SUBJ\_NAME = 'Дискретная математика', HOURS = 36**

**WHERE SUBJ\_ID = 12;**

### *Задание*

1. Ввести данные в таблицы с помощью команды **Insert**
2. Изменить некоторые данные с помощью команды **Update**
3. Удалить несколько строк с помощью команды **Delete**

## **Лабораторная работа №4**

### **Проектирование запросов**

1. Спроектировать запросы и представить формулировку запросов в виде предложений русского языка. Количество запросов – 10. Из общего количества запросов простыми могут быть не более 3, остальные запросы должны быть сложными. (Простые запросы – это запросы, с помощью которых информация выбирается из одной сущности. В сложных запросах информация должна выбираться из двух или более сущностей одновременно.)

Запросы должны быть спроектированы так, чтобы одновременно обладать физическим смыслом с точки зрения предметной области и продемонстрировать следующие возможности языка SQL:

- а) работа с агрегатными функциями (подсчет количества, расчет средних значений, минимальных, максимальных значений, сумм и т.д.),
- б) применение вложенных запросов (вложенное обращение одного оператора **SELECT** к результатам другого оператора **SELECT**),
- в) применение кванторов (запросы на всеобщность, существование – ключевые слова **EXISTS** и **ALL**),
- г) запросы с объединением результатов двух и более запросов (ключевое слово **UNION**),
- д) запросы с применением группировки,
- е) запросы с применением маски, запросы с предикатами,
- ж) запросы для работы с датами.

В запросах должны быть заданы условия поиска (Сравнение, Диапазон, Принадлежность множеству, Соответствие шаблону, Значение *NULL*)

Отчет по лабораторной работе должен содержать список запросов в виде:

***Простые запросы:***

1. Найти группу с минимальным/максимальным количеством студентов.
2. Вывести фамилии студентов, дата рождения которых попадает в диапазон от 12.05.1994г до 28.12.1996

***Сложные запросы:***

1. По фамилии студента определить фамилию его куратора
2. По фамилии преподавателя определить название кафедры, на которой он работает.
3. Найти преподавателей, у которых стаж работы больше, чем средний стаж работы всех преподавателей

## **Лабораторная работа №5**

### **Создание запросов**

Все запросы на получение любого количества данных из одной или нескольких таблиц выполняются с помощью предложения ***SELECT***. В общем случае результатом реализации предложения ***SELECT*** является другая таблица. К этой новой (рабочей) таблице может быть снова применена операция ***SELECT*** и т.д., т.е. такие операции могут быть вложены друг в друга. Именно возможность включения одного предложения ***SELECT*** внутри другого послужила мотивировкой использования прилагательного «структурированный» в названии языка SQL.

### ***Задание***

Реализовать запросы, спроектированные в лабораторной работе №1, на языке Transact-SQL.



## Лабораторная работа №6

### Проектирование представлений, хранимых процедур и триггеров

#### 1. Проектирование представлений

##### Задание

1. Создать необновляемое представление и обновляемое представление для определенных категорий пользователей.
2. Привести примеры команд, подтверждающих, что данное представление является необновляемым / обновляемым.
3. Создать агрегирующее представление.
4. Создать представление, основанное на нескольких таблицах.

#### 2. Проектирование хранимых процедур

##### Задание

Разработать процедуры, в которых предусмотрены операции добавления, удаления, изменения данных, а также какие-либо вычисления.

#### 3. Проектирование триггеров

##### Задание.

Создать триггеры, срабатывающие при добавлении, удалении, обновлении данных. Ниже приведены примеры триггеров для различных предметных областей.

##### Примеры триггеров

Предметная область: **Библиотека**

**Триггер:** при удалении читателя, на руках у которого имеются книги, уменьшить количество экземпляров этих книг.

Предметная область: **Оптовая база**

**Триггер:** при удалении поставщика удаляются товары, поставлявшиеся только этим поставщиком.

Предметная область: **Производство**

**Триггер:** при удалении изделия материал, используемый только в этом изделии, помечается как неиспользуемый.

Предметная область: **Автомастерская**

**Триггер:** при досрочном завершении ремонта премия автомеханика увеличивается на 1 % стоимости ремонта за каждый выигранный день.

Предметная область: **Сессия**

**Триггер:** общий объем часов для группы вычисляется вновь при изменении объема часов дисциплины или при переназначении дисциплины на сессию.

Предметная область: **Поликлиника**

**Триггер:** больной-пенсионер направляется к врачу не ниже 2-й категории.

Предметная область: **Телефонизация**

**Триггер:** контроль – не может быть более 2 заблокированных телефонов.

Предметная область: **Спорт**

**Триггер:** исправление мирового рекорда при появлении соответствующего результата.

Предметная область: **Сельскохозяйственные поставки**

**Триггер:** при удалении предприятия, являющегося единственным поставщиком какой-то продукции, удаляется и эта продукция.

Предметная область: **Городской транспорт**

**Триггер:** количество машин в парке не может превышать сумму количеств машин на маршрутах.

Предметная область: **Аэропорт**

**Триггер:** количество проданных билетов не должно превышать числа мест на рейсе.

Предметная область: **Деканат**

**Триггер:** количество студентов, зачисленных на I курс, не может превышать число мест на факультете.

Предметная область: **Автотранспортное предприятие**

**Триггер:** при зачислении контролируется превышение числа штатных единиц и фонда зарплаты.

Предметная область: **Театр**

**Триггер:** если при назначении актера на роль оказывается, что роль уже «занята», в старом назначении делается отметка о снятии.

Предметная область: **Справочная аптек**

**Триггер:** при изменении количества медикаментов запись удаляется, если количество становится 0.

Предметная область: **Отдел кадров**

**Триггер:** при зачислении контролируется превышение числа штатных единиц и фонда зарплаты.

## Лабораторная работа №7

### Представления

*Представления*, или *просмотры* (VIEW), – это временные, производные (иначе - виртуальные) таблицы. Они являются объектами БД, информация в которых не хранится постоянно, как в базовых таблицах, а формируется динамически при обращении к ним. Обычные таблицы относятся к базовым, т.е. содержащим данные и постоянно находящимся на устройстве хранения информации. *Представление* не может существовать само по себе, а определяется только в терминах одной или нескольких таблиц. Применение *представлений* позволяет разработчику БД обеспечить каждому пользователю или группе пользователей наиболее подходящие способы работы с данными, что решает проблему простоты их использования и безопасности. Содержимое *представлений* выбирается из других таблиц с помощью выполнения запроса, причем при изменении значений в таблицах данные в *представлении*

автоматически меняются. *Представление* - это фактически тот же запрос, который выполняется всякий раз при участии в какой-либо команде. Результат выполнения этого запроса в каждый момент времени становится содержанием *представления*. У пользователя создается впечатление, что он работает с настоящей, реально существующей таблицей.

Создания и изменения *представлений* представлены следующей командой:

```
<определение_просмотра> ::=
{ CREATE| ALTER} VIEW имя_просмотра
[(имя_столбца [...n])]
[WITH ENCRYPTION]
AS SELECT_оператор
[WITH CHECK OPTION]
```

По умолчанию имена столбцов в *представлении* соответствуют именам столбцов в исходных таблицах. Явное указание имени столбца требуется для вычисляемых столбцов или при объединении нескольких таблиц, имеющих столбцы с одинаковыми именами. Имена столбцов перечисляются через запятую, в соответствии с порядком их следования в *представлении*.

Параметр WITH ENCRYPTION предписывает серверу шифровать SQL-код запроса, что гарантирует невозможность его несанкционированного просмотра и использования. Если при определении *представления* необходимо скрыть имена исходных таблиц и столбцов, а также алгоритм объединения данных, необходимо применить этот аргумент.

Параметр WITH CHECK OPTION предписывает серверу исполнять проверку изменений, производимых через *представление*, на соответствие критериям, определенным в операторе SELECT. Это означает, что не допускается выполнение изменений, которые приведут к исчезновению строки из *представления*. Такое случается, если для *представления* установлен горизонтальный фильтр и изменение данных приводит к несоответствию строки установленным фильтрам. Использование аргумента WITH CHECK OPTION гарантирует, что сделанные изменения будут отображены в *представлении*. Если пользователь пытается выполнить изменения, приводящие к исключению строки из *представления*, при заданном аргументе WITH CHECK OPTION сервер выдаст сообщение об ошибке и все изменения будут отклонены.

*Представление* можно использовать в команде так же, как и любую другую таблицу. К *представлению* можно строить запрос, модифицировать его (если оно отвечает определенным требованиям), соединять с другими таблицами. Содержание *представления* не фиксировано и обновляется каждый раз, когда на него ссылаются в команде. *Представления* значительно расширяют возможности управления данными. В частности, это прекрасный способ разрешить доступ к информации в таблице, скрыв часть данных.

### Обновление данных в представлениях

Не все *представления* в SQL могут быть модифицированы. *Модифицируемое представление* определяется следующими критериями:

- основывается только на одной базовой таблице;
- содержит первичный ключ этой таблицы;
- не содержит DISTINCT в своем определении;
- не использует GROUP BY или HAVING в своем определении;
- по возможности не применяет в своем определении подзапросы;
- не использует константы или выражения значений среди выбранных полей вывода;
- должен быть включен каждый столбец таблицы, имеющий атрибут NOT NULL;
- оператор SELECT представления не использует агрегирующие (итоговые) функции, соединения таблиц, хранимые процедуры и функции, определенные пользователем;
- основывается на одиночном запросе, поэтому объединение UNION не разрешено.

Пример 1

```
CREATE VIEW DATEEXAM (EXAM_DATE, QUANTITY)
AS SELECT EXAM_DATE, COUNT (*)
FROM EXAM_MARKS
GROUP BY EXAM_DATE;
```

Данное представление является необновляемым из-за присутствия в нем агрегирующей функции и GROUP BY.

Пример 2

```
CREATE VIEW LCUSTT
AS SELECT *
FROM UNIVERSITY
WHERE CITY = 'Москва';
```

Это обновляемое представление.

Если *просмотр* удовлетворяет этим условиям, к нему могут применяться операторы INSERT, UPDATE, DELETE. Различия между *модифицируемыми представлениями* и *представлениями*, предназначенными только для чтения, не случайны. Цели, для которых их используют, различны. С *модифицируемыми представлениями* в основном обходятся точно так же, как и с базовыми таблицами. Фактически, пользователи не могут даже осознать, является ли объект, который они запрашивают, базовой таблицей или *представлением*, т.е. прежде всего это средство защиты для сокрытия конфиденциальных или не относящихся к потребностям данного пользователя частей таблицы. *Представления* в режиме <только для чтения> позволяют получать и форматировать данные более рационально. Они создают целый арсенал сложных запросов, которые можно выполнить и повторить снова, сохраняя полученную информацию. Результаты этих запросов могут затем использоваться в других запросах, что позволит избежать сложных предикатов и снизить вероятность ошибочных действий.

Обычно в представлениях используются имена, полученные непосредственно из имен полей основной таблицы. Однако иногда необходимо дать столбцам новые имена, например, в случае итоговых функций или вычисляемых столбцов.

```
CREATE VIEW view4(Код, Название,
Тип, Цена, Налог) AS
SELECT КодТовара, Название,
```

```
Тип, Цена, Цена*0.05 FROM Товар
```

### **Модифицирование представлений**

Данные, предъявляемые пользователю через представление, могут изменяться с помощью команд модификации DML, но при этом фактическая модификация данных будет осуществляться не в самой виртуальной таблице-представлении, а будет перенаправлена к соответствующей базовой таблице. Например, запрос на обновление представления NEW\_STUDENT

```
UPDATE NEW_STUDENT
SET CITY = 'Москва'
WHERE STUDENT_ID = 1004;
```

эквивалентен выполнению команды UPDATE над базовой таблицей STUDENT. Следует, однако, в общем случае учитывать, что обычно в представлении отображаются данные из базовой таблицы в преобразованном или усеченном виде, в результате чего применение команд модификации к таблицам-представлениям имеет некоторые особенности, рассматриваемые ниже.

### **Операции модификации в представлениях, маскирующих строки и столбцы**

Рассмотренная выше проблема возникает и при вставке строк в представление с предикатом, использующим поля базовой таблицы, не присутствующие в самом представлении. Например, рассмотрим представление

```
CREATE VIEW MOSC_STUD AS  
SELECT STUDENT_ID, SURNAME, STIPEND  
FROM STUDENT  
WHERE CITY = 'Москва';
```

Видно, что в данное представление не включено поле CITY таблицы STUDENT.

Что будет происходить при попытках вставки строки в это представление? Так как мы не можем указать значение CITY в представлении как значение по умолчанию (ввиду отсутствия, в нем этого поля), то этим значением будет NULL, и оно будет введено в поле CITY базовой таблицы STUDENT (считаем, что для этого поля опция NOT NULL не используется). Так как в этом случае значение поля CITY базовой таблицы STUDENT не будет равняться значению 'Москва', вставляемая строка будет исключена из самого представления и поэтому не будет видна пользователю. Причем так будет происходить для любой вставляемой в представление MOSC\_STUD строки. Другими словами, пользователь вообще не сможет видеть строки, вводимые им в это представление. Данная проблема не решается и в случае, если в определение представления будет добавлена опция WITH CHECK OPTION:

```
CREATE VIEW MOSC_STUD AS  
SELECT STUDENT_ID, SURNAME, STIPEND  
FROM STUDENT  
WHERE CITY = 'Москва'  
WITH CHECK OPTION;
```

Таким образом, в определенном указанными способами представлении можно модифицировать значения полей или удалять строки, но нельзя вставлять строки. Исходя из этого, рекомендуется даже в тех случаях, когда этого не требуется по соображениям полезности (и даже безопасности) информации, при определении представления включать в него все поля, на которые имеется ссылка в предикате. Если эти поля не должны отображаться в выводе таблицы, всегда можно исключить их уже в запросе к представлению. Другими словами, можно было бы определить представление MOSC\_STUD подобно следующему:

```
CREATE VIEW MOSC_STUD AS  
SELECT *  
FROM STUDENT  
WHERE CITY = 'Москва'  
WITH CHECK OPTION;
```

Эта команда заполнит в представлении поле CITY одинаковыми значениями, которые можно просто исключить из вывода с помощью другого запроса уже к этому сформированному представлению, указав в запросе только поля, необходимые для вывода.

```
SELECT STUDENT_ID, SURNAME, STIPEND  
FROM MOSC_STUD;
```

### **Агрегированные представления**

Создание представлений с использованием агрегирующих функций и предложения GROUP BY является удобным инструментом для непрерывной обработки и интерпретации извлекаемой информации. Предположим, необходимо следить за количеством студентов, сдающих экзамены, количеством сданных экзаменов, количеством сданных предметов, средним баллом по каждому предмету. Для этого можно сформировать следующее представление:

```
CREATE VIEW TOTALDAY AS  
SELECT EXAM_DATE, COUNT(DISTINCT SUBJ_ID) AS SUBJ_CNT,  
COUNT(STUDENT_ID) AS STUD_CNT,  
COUNT(MARK) AS MARK_CNT,  
AVG(MARK) AS MARK_AVG, SUM(MARK) AS MARK_SUM  
FROM EXAM_MARKS
```

**GROUP BY EXAM\_DATE;**

Теперь требуемую информацию можно увидеть с помощью простого запроса к представлению:

**SELECT \* FROM TOTALDAY;**

### ***Представления, основанные на нескольких таблицах***

Представления часто используются для объединения нескольких таблиц (базовых и/или других представлений) в одну большую виртуальную таблицу. Такое решение имеет ряд преимуществ:

- представление, объединяющее несколько таблиц, может использоваться при формировании сложных отчетов как промежуточный макет, скрывающий детали объединения большого количества исходных таблиц;
- предварительно объединенные поисковые и базовые таблицы обеспечивают наилучшие условия для транзакций, позволяют использовать компактные схемы кодов, устраняя необходимость написания для каждого отчета длинных объединяющих процедур;
- позволяет использовать при формировании отчетов более надежный модульный подход;
- предварительно объединенные и проверенные представления уменьшают вероятность ошибок, связанных с неполным выполнением условий объединения.

Можно, например, создать представление, которое показывает имена и названия сданных предметов для каждого студента:

**CREATE VIEW STUD\_SUBJ AS**

**SELECT A.STUDENT\_ID, C.SUBJ\_ID, A.SURNAME, C.SUBJ\_NAME  
FROM STUDENT A, EXAM\_MARKS B, SUBJECT C  
WHERE A.STUDENT\_ID = B.STUDENT\_ID  
AND B.SUBJ\_ID = C.SUBJ\_ID;**

Теперь все названия предметов, сданных студентом, или фамилии студентов, сдававших какой-либо предмет, можно выбрать с помощью простого запроса. Например, чтобы увидеть все предметы, сданные студентом Ивановым, подается запрос:

**SELECT SUBJ\_NAME  
FROM STUD\_SUBJ  
WHERE SURNAME = 'Иванов';**

## **Лабораторная работа №8**

### **Разработка хранимых процедур**

#### **Хранимые процедуры**

**Хранимая процедура (Stored procedure)** – программа, которая выполняется внутри базы данных и может предпринимать сложные действия на основе информации, задаваемой пользователем. Поскольку хранимые процедуры выполняются непосредственно на сервере базы данных, обеспечивается более высокое быстродействие, чем при выполнении тех же операций средствами клиента базы данных.

Хранимая процедура объединяет запросы и процедурную логику (операторы присваивания, логического ветвления и т.п.) и хранится в базе данных.

Одна процедура может быть использована в любом количестве клиентских приложений, что позволяет существенно сэкономить трудозатраты на создание прикладного программного обеспечения и эффективно применять стратегию повторного использования ко-

да. Так же, как и любые процедуры в стандартных языках программирования, хранимые процедуры могут иметь входные и выходные параметры или не иметь их.

Преимущества выполнения в базе данных хранимых процедур вместо отдельных команд *Transact SQL*:

- необходимые команды уже содержатся в базе данных;
- все они прошли этап синтаксического анализа и находятся в исполняемом формате;
- хранимые процедуры поддерживают модульное программирование, так как позволяют разбивать большие задачи на самостоятельные, более мелкие и удобные в управлении части;
- хранимые процедуры могут вызывать другие хранимые процедуры и функции;
- хранимые процедуры могут быть вызваны из прикладных программ других типов;
- как правило, хранимые процедуры выполняются быстрее, чем последовательность отдельных команд;
- хранимые процедуры проще использовать: они могут состоять из десятков и сотен команд, но для их запуска достаточно указать всего лишь имя нужной хранимой процедуры. Это позволяет уменьшить размер запроса, посылаемого от клиента на сервер, а значит, и нагрузку на сеть.

Хранимые процедуры вызываются клиентской программой, другой хранимой процедурой или триггером. Разработчик может управлять правами доступа к хранимой процедуре, разрешая или запрещая ее выполнение. Изменять код хранимой процедуры разрешается только ее владельцу или члену фиксированной роли базы данных. При необходимости можно передать права владения ею от одного пользователя к другому.

### Создание, изменение и удаление хранимых процедур

Создание новой и изменение имеющейся хранимой процедуры осуществляется с помощью следующей команды:

```
<определение_процедуры>::=
{CREATE | ALTER } PROC[EDURE] имя_процедуры
[/номер]
[/{@имя_параметра тип_данных } [VARYING ]
[/значение_по_умолчанию][OUTPUT] ][,...n]
[WITH { RECOMPILE | ENCRYPTION | RECOMPILE,
ENCRYPTION }]
[FOR REPLICATION]
AS
Тело процедуры;
```

**Номер в имени** – это идентификационный номер хранимой процедуры, однозначно определяющий ее в группе процедур. Для удобства управления процедурами логически однотипные хранимые процедуры можно группировать, присваивая им одинаковые имена, но разные идентификационные номера.

Для передачи входных и выходных данных в создаваемой хранимой процедуре могут использоваться параметры, имена которых, как и имена локальных переменных, должны

начинаться с символа **@**. В одной хранимой процедуре можно задать множество параметров, разделенных запятыми. В теле процедуры не должны применяться локальные переменные, чьи имена совпадают с именами параметров этой процедуры.

Для определения типа данных, который будет иметь соответствующий параметр хранимой процедуры, годятся любые типы данных SQL, включая определенные пользователем.

**OUTPUT** означает, что соответствующий параметр предназначен для возвращения данных из хранимой процедуры. Указание ключевого слова **OUTPUT** предписывает серверу при выходе из хранимой процедуры присвоить текущее значение параметра локальной переменной, которая была указана при вызове процедуры в качестве значения параметра. При указании ключевого слова **OUTPUT** значение соответствующего параметра при вызове процедуры может быть задано только с помощью локальной переменной. Не разрешается использование любых выражений или констант, допустимое для обычных параметров.

Необязательное ключевое слово **VARYING** определяет заданное значение по умолчанию для определенного ранее параметра.

Ключевое слово **DEFAULT** представляет собой значение, которое будет принимать соответствующий параметр по умолчанию. Таким образом, при вызове процедуры можно не указывать явно значение соответствующего параметра.

Так как сервер кэширует план исполнения запроса и компилированный код, при последующем вызове процедуры будут использоваться уже готовые значения. Однако в некоторых случаях все же требуется выполнять перекомпиляцию кода процедуры. Указание ключевого слова **RECOMPILE** предписывает системе создавать план выполнения хранимой процедуры при каждом ее вызове.

Параметр **FOR REPLICATION** востребован при репликации данных и включении создаваемой хранимой процедуры в качестве статьи в публикацию.

Ключевое слово **ENCRYPTION** предписывает серверу выполнить шифрование кода хранимой процедуры, что может обеспечить защиту от использования авторских алгоритмов, реализующих работу хранимой процедуры.

Ключевое слово **AS** размещается в начале собственно тела хранимой процедуры. В теле процедуры могут применяться практически все команды SQL, объявляться транзакции, устанавливаться блокировки и вызываться другие хранимые процедуры. Выход из хранимой процедуры можно осуществить посредством команды **RETURN**.

Удаление хранимой процедуры осуществляется командой:

**DROP PROCEDURE {имя\_процедуры} [...n];**

Выполнение хранимой процедуры

Для выполнения хранимой процедуры используется команда:

**EXEC [ UTE] имя\_процедуры [;номер]  
 [/@имя\_параметра={значение} | @имя\_переменной}  
 [/OUTPUT ]/[DEFAULT ]/[...n]**

Если вызов хранимой процедуры не является единственной командой в пакете, то присутствие команды **EXECUTE** обязательно. Более того, эта команда требуется для вызова процедуры из тела другой процедуры или триггера.

Использование ключевого слова **OUTPUT** при вызове процедуры разрешается только для параметров, которые были объявлены при создании процедуры с ключевым словом **OUTPUT**.



Если при вызове процедуры для параметра указывается ключевое слово **DEFAULT**, то будет использовано значение по умолчанию. Естественно, указанное слово **DEFAULT** разрешается только для тех параметров, для которых определено значение по умолчанию.

Из синтаксиса команды **EXECUTE** видно, что имена параметров могут быть опущены при вызове процедуры. Однако в этом случае пользователь должен указывать значения для параметров в том же порядке, в каком они перечислялись при создании процедуры. Присвоить параметру значение по умолчанию, просто пропустив его при перечислении, нельзя. Если же требуется опустить параметры, для которых определено значение по умолчанию, достаточно явного указания имен параметров при вызове хранимой процедуры. Более того, таким способом можно перечислять параметры и их значения в произвольном порядке.

При вызове процедуры указываются либо имена параметров со значениями, либо только значения без имени параметра. Их комбинирование не допускается.

**Пример 1. Процедура без параметров.** Создать процедуру для уменьшения размера стипендии на 10 %:

```
CREATE Procedure Reduce  
AS  
UPDATE Students SET Stipendiya = Stipendiya * 0.9  
WHERE Stipendiya IS NOT NULL;
```

Для обращения к процедуре можно использовать команду:

```
EXECUTE Reduce;
```

**Пример 2. Процедура с входными параметрами.** Создать процедуру для выдачи списка студентов, получивших определенную оценку по определенному экзамену:

```
CREATE Procedure Subject  
@Subject varchar(50), @Mark tinyint  
AS  
SELECT s.FIO AS 'ФИО', l.Nazvanie AS 'Дисциплина', p.Examen AS 'Оценка за  
экзамен'
```

```
FROM Students AS s  
INNER JOIN  
Progress AS p ON p.ID_Student = s.ID_Student  
INNER JOIN  
Lessons AS l ON l.ID_Lesson = p.ID_Lesson  
WHERE l.Nazvanie = @Subject AND p.Examen = @Mark;
```

Для обращения к процедуре можно использовать команду:

```
EXEC Subject 'Объектно-ориентированное программирование', 5;
```

**Пример 3. Процедура с входными параметрами и значениями по умолчанию.** Создать процедуру для выдачи списка студентов, получивших определенную оценку по определенному экзамену. По умолчанию вывести фамилии студентов, получивших оценку «3» по дисциплине «Алгебра и геометрия»:

```
CREATE Procedure ExamResultsDef
```

```

@Subject varchar(50)= VARYING 'Алгебра и геометрия',
@Mark tinyint = 3
AS
SELECT s.FIO AS 'ФИО', l.Nazvanie AS 'Дисциплина', p.Examen AS 'Оценка за
экзамен'
FROM Students AS s INNER JOIN
Progress AS p ON p.ID_Student = s.ID_Student INNER JOIN
Lessons AS l ON l.ID_Lesson = p.ID_Lesson
WHERE (@Subject IS NOT NULL AND l.Nazvanie = @Subject AND p.Examen =
@Mark) OR
(@Subject IS NULL AND p.Examen = @Mark);

```

Для обращения к процедуре можно использовать команды:

1. **EXEC ExamResultsDef** – в этом случае выводятся значения по умолчанию, т.е. заданные в процедуре, – оценка «3» и дисциплина «Алгебра и геометрия».
2. **EXEC ExamResultsDef @Subject = 'Объектно-ориентированное программирование', @Mark = 5** – в этом случае выводится список студентов, получивших оценку «5» по дисциплине «Объектно-ориентированное программирование».
3. **EXEC ExamResultsDef @Subject = 'Объектно-ориентированное программирование'** – выводится список студентов, получивших оценку «3» по дисциплине «Объектно-ориентированное программирование».
4. **EXEC ExamResultsDef @Mark = 5** выводится список студентов, получивших оценку «5» по дисциплине «Алгебра и геометрия».

**Пример 4. Использование вложенных процедур.** Создать процедуру для определения куратора группы, в которой учится определенный студент.

Сначала разработаем процедуру для определения групп и их кураторов:

```

CREATE PROCEDURE Curator
@Grp VARCHAR(10),
@Srn VARCHAR(20) OUTPUT
AS
SELECT @Srn = Familia
FROM Teachers
INNER JOIN Groups ON Groups.Kurator = Teachers.ID_Teacher
WHERE Groups.Name_group = @Grp;

```

Затем создадим процедуру, определяющую студентов и их кураторов:

```

CREATE PROCEDURE StudentsCurator
@FIO VARCHAR(70),
@Crtr VARCHAR(20) OUTPUT
AS
DECLARE @GNm VARCHAR(10)
SELECT @GNm = Name_group
FROM Students s
INNER JOIN Groups g ON g.ID_Group = s.ID_Group
WHERE s.FIO = @FIO

```

**EXEC Curator @GNm,@Crtr OUTPUT**

Вызов процедуры осуществляется следующим образом:

**DECLARE @Crtr VARCHAR(20)**

**EXECUTE StudentsCurator 'Иванов А.С.', @Crtr OUTPUT**

**PRINT @Crtr;**

## Лабораторная работа №9

### Создание триггеров

**Триггеры** – это предварительно определенное действие или последовательность действий, **автоматически** осуществляемых при выполнении операций обновления, добавления или удаления данных.

Исключительно важно в этом определении слово **«автоматически»**. Ни пользователь, ни приложение не могут активизировать триггер, он выполняется автоматически, когда пользователь или приложение выполняют с базой данных определенные действия.

Триггер – это специальный вид хранимой процедуры. Триггеры обеспечивают проверку любых изменений на корректность, прежде чем эти изменения будут приняты.

Каждый триггер привязывается к конкретной таблице. Все производимые им модификации данных рассматриваются как одна транзакция. В случае обнаружения ошибки или нарушения целостности данных происходит откат этой транзакции. Тем самым внесение изменений запрещается. Отменяются также все изменения, уже сделанные триггером.

Создать триггер может только владелец базы данных. Это ограничение позволяет избежать случайного изменения структуры таблиц, способов связи с ними других объектов и т.п.

### Компоненты триггера

1. **Ограничения**, для реализации которых создается триггер.
2. **Событие**, которое будет характеризовать возникновение ситуации, требующей проверки ограничений. Триггерные события чаще всего связаны с изменением состояния базы данных и состоят из вставки, удаления и обновления строк в таблице. События могут учитываться и дополнительные условия (например, добавление записи только с отрицательным значением).
3. **Предусмотренное действие** осуществляется за счет выполнения процедуры или последовательности процедур, с помощью которых реализуется логика, требуемая для реализации ограничений.

Триггер выполняется неявно в каждом случае возникновения триггерного события. Приведение его в действие называют **запуском триггера**. С помощью триггеров достигаются следующие цели:

- проверка корректности введенных данных и выполнение сложных ограничений целостности данных, которые трудно, если вообще возможно, поддерживать с помощью ограничений целостности, установленных для таблицы;
- выдача предупреждений, напоминающих о необходимости выполнения некоторых действий при обновлении таблицы, реализованном определенным образом.

## Типы триггеров

Существует три типа триггеров:

1. **Insert** – определяет действия, которые будут выполняться после добавления новой записи в таблицу.
2. **Update** – определяет действия, которые будут выполняться после изменения записи таблицы.
3. **Delete** – определяет действия, которые будут выполняться после удаления записи из таблиц.

Часто в СУБД определяется большее число событий, с которыми можно связать триггеры. Например, до вставки, после вставки, до изменения, после изменения и т.д.

SQL Server поддерживает два вида триггеров: замещающие (**INSTEAD OF**) и завершающие (**AFTER**). Замещающие триггеры вызываются вместо обработки запроса на вставку, обновление или удаление, а завершающие – после обработки запроса. Всего имеется шесть возможных типов триггеров: замещающий триггер вставки, обновления и удаления и завершающий триггер вставки, обновления и удаления.

Для триггеров вставки и обновления в SQL Server новые значения столбца обрабатываемой таблицы хранятся в псевдотаблице под названием *inserted*. В случае обновления и удаления старые значения каждого столбца будут храниться в псевдотаблице с именем *deleted*.

## Создание триггеров

Основной формат команды **CREATE TRIGGER**:

```
<Определение_триггера>::=
CREATE TRIGGER [имя_триггера]
ON имя_таблицы
{ FOR | AFTER | INSTEAD OF } {[INSERT] [,] [UPDATE] [,]
[ DELETE]}
[WITH ENCRYPTION]
AS SQL_операторы
Или используя предложение IF UPDATE
<Определение_триггера>::=
CREATE TRIGGER [имя_триггера]
ON имя_таблицы
{ FOR | AFTER | INSTEAD OF } {[INSERT] [,] [UPDATE] [,] [ DELETE]}
[WITH ENCRYPTION]
AS
IF UPDATE (имя_столбца)
[ {AND | OR} UPDATE (имя_столбца)...]
SQL_операторы;
```

CREATE TRIGGER [имя\_триггера] – создается новый триггер с именем имя\_триггера

ON имя\_таблицы – объявляется таблица или представление, от которых зависит триггер.

**WITH ENCRYPTION** имеет тот же смысл, что и для хранимых процедур, он скрывает исходный текст тела триггера.

{ FOR | AFTER | INSTEAD OF } – указывает, когда должен запускаться триггер. Ключевые слова FOR и AFTER являются синонимами. Предложение AFTER показывает, что триггер запускается только после успешного выполнения операции по модификации данных (и других каскадно запускаемых действий и проверок ограничений). Триггер INSTEAD OF может полностью заменить операцию по модификации данных. При этом триггер запускается вместо операции по модификации, которая запустила триггер. Триггер INSTEAD OF DELETE нельзя использовать, если удаление вызывает каскадные действия. Доступ к столбцам TEXT или IMAGE имеют только триггеры INSTEAD OF.

IF UPDATE (имя\_столбца) [{AND | OR} UPDATE (имя\_столбца)...] – позволяет выбрать конкретный столбец, запускающий триггер. Триггеры, специфичные для столбца, запускаются только при операциях INSERT или UPDATE, но не DELETE.

Конструкции **FOR {INSERT, UPDATE, DELETE}** определяют, на какую команду будет реагировать триггер. При его создании должна быть указана хотя бы одна команда. Допускается создание триггера, реагирующего на две или на все три команды.

Неправильно написанные триггеры могут привести к серьезным проблемам, таким, например, как появление «мертвых» блокировок. Триггеры способны длительное время блокировать множество ресурсов, поэтому следует обратить особое внимание на сведение к минимуму конфликтов доступа.

В большинстве СУБД действуют следующие ограничения:

- Нельзя использовать в теле триггера операции создания объектов базы данных (новой базы данных, новой таблицы, новой хранимой процедуры, нового триггера, новых представлений).
- Нельзя использовать в триггере команду удаления объектов **DROP** для всех типов базовых объектов базы данных.
- Нельзя использовать в теле триггера команды изменения базовых объектов **ALTER TABLE, ALTER DATABASE**.
- Нельзя изменять права доступа к объектам базы данных, т.е. выполнять команду **GRANT** или **REVOKE**.
- Нельзя создать триггер для представления (**VIEW**).
- В отличие от хранимых процедур триггер не может возвращать никаких значений, он запускается автоматически сервером и не может связаться самостоятельно ни с одним клиентом.
- Внутри триггера не допускается выполнение восстановления резервной копии БД или журнала транзакций.

Выполнение этих команд не разрешено, так как они не могут быть отменены в случае отката транзакции, в которой выполняется триггер.

### *Преимущества использования триггеров:*

1. Триггеры всегда выполняются при совершении соответствующих действий. Разработчик продумывает использование триггеров при проектировании базы данных и может больше не вспоминать о них при разработке приложения для доступа к данным. Если для работы с этой же базой данных нужно создать новое приложение, триггеры и там будут обрабатывать заданные ограничения.
2. При необходимости триггеры можно изменять централизованно непосредственно в базе данных. Пользовательские программы, использующие данные из этой базы данных, не потребуют модернизации.
3. Система обработки данных, использующая триггеры, обладает лучшей переносимостью в архитектуру клиент-сервер за счет меньшего объема требуемых модификаций.

### **Программирование триггеров**

При выполнении команд добавления, изменения и удаления записей сервер создает две специальные таблицы: *inserted* и *deleted*. В них содержатся списки строк, которые будут вставлены или удалены по завершении транзакции. Структура таблиц *inserted* и *deleted* идентична структуре таблиц, для которой определяется триггер. Для каждого триггера создается свой комплект таблиц *inserted* и *deleted*, поэтому никакой другой триггер не сможет получить к ним доступ.

В зависимости от типа операции, вызвавшей выполнение триггера, содержимое таблиц *inserted* и *deleted* может быть разным:

- команда **INSERT** – в таблице *inserted* содержатся все строки, которые пользователь пытается вставить в таблицу. В таблице *deleted* не будет ни одной строки. После завершения триггера все строки из таблицы *inserted* переместятся в исходную таблицу;
- команда **DELETE** – в таблице *deleted* будут содержаться все строки, которые пользователь попытается удалить. Триггер может проверить каждую строку и определить, разрешено ли ее удаление. В таблице *inserted* не окажется ни одной строки;
- команда **UPDATE** – при ее выполнении в таблице *deleted* находятся старые значения строк, которые будут удалены при успешном завершении триггера. Новые значения строк содержатся в таблице *inserted*. Эти строки добавятся в исходную таблицу после успешного выполнения триггера.

Для получения информации о количестве строк, которое будет изменено при успешном завершении триггера, можно использовать функцию **@@ROWCOUNT**; она возвращает количество строк, обработанных последней командой. Следует помнить, что триггер запускается не при попытке изменить конкретную строку, а в момент выполнения команды изменения. Одна такая команда воздействует на множество строк, поэтому триггер должен обрабатывать все эти строки.

Если триггер обнаружил, что из 100 вставляемых, изменяемых или удаляемых строк только одна не удовлетворяет тем или иным условиям, то никакая строка не будет вставлена, изменена или удалена. Такое поведение обусловлено требованиями транзакции – должны быть выполнены либо все модификации, либо ни одной.

Триггер выполняется как неявно определенная транзакция, поэтому внутри триггера допускается применение команд управления транзакциями. В частности, при обнаружении нарушения ограничений целостности для прерывания выполнения триггера и отмены всех изменений, которые пытался выполнить пользователь, необходимо использовать команду **ROLLBACK TRANSACTION**. Для фиксации изменений, внесенных при выполнении транзакции, следует использовать команду **COMMIT TRANSACTION**.

Для удаления триггера используется команда  
**DROP TRIGGER {имя\_триггера} [...n].**

### Примеры использования *триггеров*

**Пример 1.** Реализовать ограничение на значение.

При добавлении записи в таблицу *Students* автоматически проверяется количество студентов в заданной группе и, если их количество больше 20, то происходит откат транзакции. Если же студентов в данной группе меньше 20, то происходит увеличение количества студентов на 1 и происходит добавление записи в таблицу *Students*.

Команда вставки записи в таблицу *Students* может быть такой:

**INSERT INTO Students (FIO, Nomer\_zachetki, ID\_Group, Stipendiya) VALUES ('Ильин С.В.', '08ВП131', 2, 1250);**

```
CREATE TRIGGER InsertStudent
ON Students FOR Insert
AS
DECLARE @ID INT
IF @@ROWCOUNT=1
BEGIN
SELECT @ID=ID_Group
FROM INSERTED
BEGIN
IF 20>(SELECT Kol_stud
FROM Groups
WHERE ID_Group=@ID
)
BEGIN
UPDATE Groups
SET Kol_stud=Kol_stud+1
WHERE ID_Group=@ID
PRINT 'студент успешно добавлен в данную группу'
END
ELSE
BEGIN
ROLLBACK TRANSACTION
PRINT 'Группа переполнена!Выберите другую группу!'
END
END
END;
```

**Пример 2.** Создать триггер для обработки операции удаления записи из таблицы *Students*, например, такой команды:

**DELETE FROM Students WHERE ID\_Student=82;**

При удалении студента из группы количество студентов в группе уменьшается на единицу:

```
CREATE TRIGGER TriggerDelete
ON Students FOR Delete
AS
DECLARE @ID INT, @ID_Group INT
IF @@ROWCOUNT=1
BEGIN
    SELECT @ID=ID_Group
    FROM DELETED
UPDATE Groups
SET Kol_stud=Kol_stud-1
WHERE ID_Group=@ID
PRINT 'студент успешно удален из группы'
END;
```

#### 17.6. Использование хранимых процедур в триггерах

Хранимые процедуры могут быть активизированы не только пользовательскими приложениями, но и триггерами.

**Пример 3.** Создадим процедуру, обновляющую количество студентов в группе:

```
CREATE PROCEDURE UpdateKolStud
@group INT
AS
DECLARE @newKolStud SMALLINT
BEGIN
SELECT @newKolStud = COUNT(*) FROM Students WHERE ID_Group = @group
UPDATE Groups SET Kol_Stud = @newKolStud WHERE ID_Group = @group
END;
```

Теперь создадим триггер, который будет срабатывать при удалении студента из базы данных или добавлении студента в базу данных:

```
CREATE TRIGGER KolStudTrigger
ON Students
AFTER INSERT, DELETE
AS
DECLARE @gr1 INT, @gr2 INT
if @@rowcount = 1
BEGIN
    SELECT @Gr1 = ID_Group FROM deleted
    SELECT @Gr2 = ID_Group FROM inserted
```



```

IF (SELECT DISTINCT ID_Group FROM deleted) IS NOT NULL
    EXEC UpdateKolStud @group = @gr1;
IF (SELECT DISTINCT ID_Group FROM inserted) IS NOT NULL
    EXEC UpdateKolStud @group = @gr2;
END;

```

Вид таблицы до транзакции приведен на рис. 1.

ID_Student	FIO	Data_Rozhd	Adres	Nomer_zachetki	ID_Group	ID_Kaf	Stipendiya
7	Макарь В.А.	NULL	NULL	06ВП118	4	1	583
10	Ивкин И.Ю.	NULL	NULL	06ВП110	4	1	1239
11	Заваровский К....	NULL	NULL	06ВП109	4	1	583
12	Заваровский К.В.	NULL	NULL	06ВП108	4	1	874

Рис. 1 – Таблица *Students*

В группе № 4 обучается 24 человека.

Результат выполнения запроса приведен на рис. 2.

ID_Group	Name_group	Kol_stud	Kurator
1	06ВП2	25	1
2	06ВП1	23	2
3	06ВВ1	22	5
4	07ВП1	24	6
5	07ВП2	24	3

Рис. 2 – Таблица *Groups*

Удалим студента № 7, который учится в группе № 4:

```
DELETE FROM Students WHERE ID_Student = 7;
```

После удаления записи о студенте количество студентов в группе уменьшилось на единицу.

Вид таблицы после выполнения транзакции приведен на рис. 3.

ID_Group	Name_group	Kol_stud	Kurator
1	06ВП2	25	1
2	06ВП1	23	2
3	06ВВ1	22	5
4	07ВП1	23	6
5	07ВП2	24	3

Рис. 3 – Таблица *Groups*

Теперь вновь добавим удаленного нами студента:

***INSERT INTO Students (FIO, Nomer\_zachetki, ID\_Group, Stipendiya) VALUES ('Макарь В.А.', '06ВП118', 4, 1200);***

Результат выполнения запроса приведен на рис. 4.

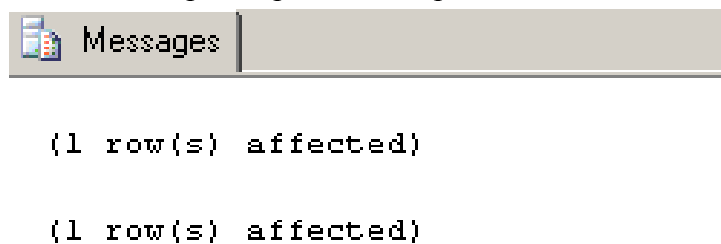


Рис. 4. Добавление строки в таблицу

Вид таблицы после транзакции приведен на рис. 5.

ID_Group	Name_group	Kol_stud	Kurator
1	06ВП2	25	1
2	06ВП1	23	2
3	06ВВ1	22	5
4	07ВП1	24	6
5	07ВП2	24	3

Рис. 5 – Таблица *Groups*