

Теория Баз Данных



ЧАСТЬ 2

Введение в теорию баз данных

Содержание

Unit 4. Многотабличные базы данных.....	5
1. Аномалии взаимодействия	
с однотабличной базой данных.....	6
Аномалии обновления	7
Аномалии вставки.....	8
Аномалии удаления	9
2. Принципы создания	
многотабличной базы данных	10
Причины создания многотабличной базы данных..	10
Внешний ключ.....	10
Связи. Типы связей	12
Целостность данных	19
Нормализация.....	20
3. Многотабличные запросы	31
Принципы создания многотабличного запроса.....	31

Декартовое произведение.....	36
4. Домашнее задание	38
Unit 5. Функции агрегирования.....	39
1. Функции агрегирования	40
Функция COUNT.....	41
Функция AVG	43
Функция SUM.....	44
Функция MIN	44
Функция MAX	44
2. Понятие группировки.	
Ключевое слово GROUP BY.....	48
3. Ключевое слово HAVING	52
Принципы использования HAVING	52
Сравнительный анализ HAVING и WHERE.....	55
4. Подзапросы	57
Необходимость создания использования подзапросов.....	57
Сравнение подзапросов и многотабличных запросов.....	61
Принцип работы подзапросов	64
5. Домашнее задание	67
Unit 6. Объединения.....	68
1. Операторы для использования в подзапросах.....	69
Оператор EXISTS	70
Оператор ANY/SOME.....	72

Оператор ALL	74
2. Объединение результатов запроса	77
Принципы объединения.....	77
Ключевое слово UNION.....	78
Ключевое слово UNION ALL.....	79
3. Объединения JOIN	83
Понятие INNER JOIN	84
Необходимость использования внешнего объединения	88
Понятие LEFT JOIN.....	89
Понятие RIGHT JOIN	92
Понятие FULL JOIN	94
4. Домашнее задание	96

Unit 4.
Многотабличные
базы данных

1. Аномалии взаимодействия с однотабличной базой данных

При использовании однотабличных баз данных имеют место три специфические проблемы, связанные с обновлением, удалением и вставкой данных. Осуществление указанных операций с таблицами базы данных может привести к противоречивости хранящихся в них данных, что в целом отрицательно скажется на работе с этой базой данных. Такого рода операции называются аномалиями, то есть тем, что является отклонением от нормы.

Для того чтобы продемонстрировать вам проблемы, связанные с однотабличной базой данных, мы воспользуемся таблицей **Teachers**, представленной на рисунке 1.1.

Name	BirthDate	Department	Phone	Group	Subject
Sophia Nelson	1984-12-08	Software development	32-12	31PPS11	C#
Emma Kirk	1973-05-12	Mathematics	55-34	32PR31	Discrete Math
Henry MacAlister	1975-02-17	Software development	32-12	30PR11	SQL Server
Michael Cooper	1978-11-23	Software development	32-12	29PR21	ADO.NET
Daniel Williams	1979-07-30	Cybersecurity	37-65	32PPS11	ITE1
Sophia Nelson	1984-12-08	Software development	32-12	30PR11	JavaScript
Daniel Williams	1979-07-30	Cybersecurity	37-65	32PPS11	WIN10

Рисунок 1.1. Таблица Teachers

Как вы заметили, в данной таблице хранятся сведения о кафедрах, преподавателях, предметах и группах, в которых преподаются указанные предметы.

Аномалии обновления

Аномалия обновления связана с избыточность данных, хранимых в соответствующей таблице. Избыточность данных является причиной того, что в процессе обновления информации в таблице часть дублируемых данных не будут изменены, что приведет к противоречивости хранимой информации.

Существует два вида избыточности: явная и неявная. Явная избыточность выражается в дублировании одинаковой информации, например, в таблице *Teachers* некоторые данные о преподавателе *Sophia Nelson* повторяются несколько раз. В том случае если у нее изменится фамилия на *Davies* (она выйдет замуж), то для того чтобы избежать противоречивости данных в таблице, необходимо будет вносить изменения в каждую запись об этом преподавателе (рис. 1.2).

Name	BirthDate	Department	Phone	Group	Subject
<u>Sophia Davies</u>	1984-12-08	Software development	32-12	31PPS11	C#
Emma Kirk	1973-05-12	Mathematics	55-34	32PR31	Discrete Math
Henry MacAlister	1975-02-17	Software development	32-12	30PR11	SQL Server
Michael Cooper	1978-11-23	Software development	32-12	29PR21	ADO.NET
Daniel Williams	1979-07-30	Cybersecurity	37-65	32PPS11	ITE1
<u>Sophia Nelson</u>	1984-12-08	Software development	32-12	30PR11	JavaScript
Daniel Williams	1979-07-30	Cybersecurity	37-65	32PPS11	WIN10

Рисунок 1.2. Явная избыточность

Неявная избыточность проявляется в виде неявной зависимости между различными записями одной таблицы, например, номер телефона кафедры *Software development* повторяется в сведениях о нескольких пре-

подавателях. Допустим, если в случае изменения на кафедре номера телефона (на 48-22) мы внесем изменения только в запись о преподавателе **Henry MacAlister**, то при последующей работе с таблицей **Teachers** мы, фактически, не сможем определить правильный номер телефона кафедры, так как в записях о преподавателях **Sophia Nelson** и **Michael Cooper** он останется прежним (рис. 1.3).

Name	BirthDate	Department	Phone	Group	Subject
Sophia Nelson	1984-12-08	Software development	32-12	31PPS11	C#
Emma Kirk	1973-05-12	Mathematics	55-34	32PR31	Discrete Math
Henry MacAlister	1975-02-17	Software development	48-22	30PR11	SQL Server
Michael Cooper	1978-11-23	Software development	32-12	29PR21	ADO.NET
Daniel Williams	1979-07-30	Cybersecurity	37-65	32PPS11	ITE1
Sophia Nelson	1984-12-08	Software development	32-12	30PR11	JavaScript
Daniel Williams	1979-07-30	Cybersecurity	37-65	32PPS11	WIN10

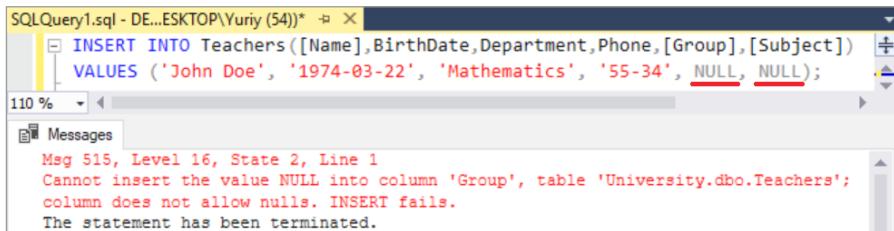
Рисунок 1.3. Неявная избыточность

Аномалии вставки

Аномалия вставки проявляется в тех случаях, когда существует необходимость поместить в таблицу запись, у которой отсутствует часть информации. Отсутствие некоторых данных в таблице, в дальнейшем, может послужить причиной получения неверных результатов при выполнении запросов к базе данных. Это не является критичным если поле с отсутствующими данными может иметь неопределенное значение (**NULL**-значения), однако для полей, обязательных к заполнению, такая ситуация приведет к возникновению ошибки. В нашем примере добавление в таблицу **Teachers** преподавателя, о котором неизвестны группа и читаемый им предмет невозможно,

1. Аномалии взаимодействия с однотабличной базой данных

поэтому мы не сможем «принять на работу» человека пока не выясним эту информацию (рис. 1.4).



The screenshot shows a SQL query window titled "SQLQuery1.sql - DE...ESKTOP\Yuriy (54)*". The query is:

```
INSERT INTO Teachers ([Name], BirthDate, Department, Phone, [Group], [Subject])
VALUES ('John Doe', '1974-03-22', 'Mathematics', '55-34', NULL, NULL);
```

In the "Messages" pane, there is an error message:

```
Msg 515, Level 16, State 2, Line 1
Cannot insert the value NULL into column 'Group', table 'University.dbo.Teachers';
column does not allow nulls. INSERT fails.
The statement has been terminated.
```

Рисунок 1.4. Ошибка: невозможно записать NULL-значение

Аномалии удаления

Аномалия удаления связана с уникальностью информации в определенных записях, при удалении которых эти данные будут потеряны. Например, в нашем случае если преподаватель [Emma Kirk](#) уволиться с работы, и мы удалим из таблицы [Teachers](#) запись содержащую информацию о ней, то это приведет к потере данных о предмете [Discrete Math](#), кафедре [Mathematics](#) и группе [32PR31](#), потому что информация о них содержится только в этой записи (рис. 1.5).

Name	BirthDate	Department	Phone	Group	Subject
Sophia Nelson	1984-12-08	Software development	32-12	31PPS11	C#
Emma Kirk	1973-05-12	Mathematics	55-34	32PR31	Discrete Math
Henry MacAlister	1975-02-17	Software development	32-12	30PR11	SQL Server
Michael Cooper	1978-11-23	Software development	32-12	29PR21	ADO.NET
Daniel Williams	1979-07-30	Cybersecurity	37-65	32PPS11	ITE1
Sophia Nelson	1984-12-08	Software development	32-12	30PR11	JavaScript
Daniel Williams	1979-07-30	Cybersecurity	37-65	32PPS11	WIN10

Рисунок 1.5. Аномалия удаления

2. Принципы создания многотабличной базы данных

Многотабличная база данных — это определенная структура для хранения различной информации, которая состоит из связанных между собой таблиц. Создание многотабличной базы данных является трудоемким процессом, но для хранения данных необходимо использовать именно такой подход, на что существует ряд причин.

Причины создания многотабличной базы данных

При хранении всей информации в одной таблице необходимо будет создавать большое количество столбцов, при этом объем хранимой информации будет увеличиваться пропорциональным образом. Записи в этой таблице будут во многом дублировать данные, что также приведет к увеличению объема информации, которая храниться в базе. Все это сказывается на процессе обработки данных, увеличивая время выполнения SQL-запросов. Нельзя также забывать и об аномалиях, описанных в предыдущем разделе, которые присущи однотабличным базам данных. Все вышеперечисленное является вескими причинами для использования именно многотабличных баз данных.

Внешний ключ

Прежде чем мы начнем обсуждать типы связей между таблицами в многотабличной базе данных, вам необходимо понять сам механизм этих связей.

Из урока № 2 вы узнали, что в любой таблице должен быть столбец, который обеспечивает уникальность записей в таблице — первичный ключ. Также в таблицах может быть еще одно специализированное поле, которое называется внешним ключом.

Внешний ключ — это поле таблицы, в котором содержится значение первичного ключа другой таблицы. Именно благодаря соотношению первичного ключа одной таблицы и внешнего ключа другой и формируются связи между таблицами в многотабличной базе данных.

В качестве примера приведем часть диаграммы базы данных, на которой в таблицах **Students** и **Groups** первичными ключами являются столбцы **Id**, что визуально отмечено изображениями ключа (рис. 2.1).

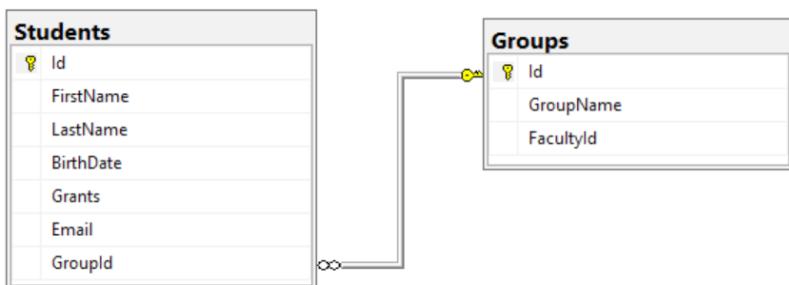


Рисунок 2.1. Пример формирования связи между таблицами

В таблице **Students** внешним ключом является столбец **GroupId**, в котором будут храниться данные из столбца **Id** — первичного ключа таблицы **Groups**. В свою очередь столбец **FacultyId** в таблице **Groups** является внешним ключом таблицы **Faculties** (на диаграмме не изображена).

Важно отметить, что в одной таблице допускается наличие нескольких столбцов, являющихся внешними ключами, каждый из которых будет хранить значения первичных ключей различных таблиц текущей базы данных. В качестве примера приведем таблицу **Achievements**, которая содержит информацию об успеваемости студентов по различным предметам (рис. 2.2).

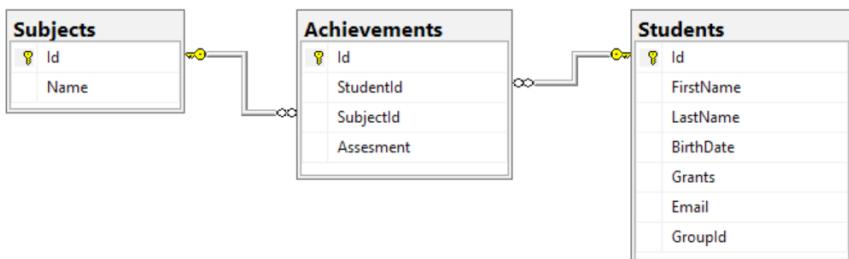


Рисунок 2.2. Пример таблицы с несколькими внешними ключами

В таблице **Achievements** внешними ключами являются столбцы **StudentId** и **SubjectId**, в которых будут храниться значения первичных ключей (столбцы **Id**) таблиц **Students** и **Subjects** соответственно.

Связи. Типы связей

На рисунке 2.1 показаны таблицы и схематичное изображение связи между таблицами **Students** и **Groups**, которая создается благодаря наличию первичного ключа в таблице **Groups** (столбец **Id**) и внешнего ключа в таблице **Students** (столбец **GroupId**). Как вы, наверное, догадались, одна сторона связи с изображением ключа соединяется с таблицей, в которой находится первичный ключ, противоположная сторона связи в виде знач-

ка бесконечности соединяется с таблицей, содержащей внешний ключ.

При проектировании любой реляционной базы данных используются три типа связей: «один к одному», «один ко многим» и «многие ко многим».

Связь «один к одному» существует между двумя таблицами в том случае, если строка данных в первой таблице соответствует только одной строке второй таблицы, а строка во второй таблице связана только с одной строкой данных первой таблицы. Этот тип связи графически представляется в виде линии, соединяющей две таблицы. На рисунке 2.3 приведен пример связи такого типа, между таблицами **Teachers** и **Authentications**.

Teachers				Authentications		
Id	LastName	FirstName	BirthDate	Id	Login	Password
1	Nelson	Sophia	1984-12-08			
2	Kirk	Emma	1973-05-12			
3	MacAlister	Henry	1975-02-17			
4	Cooper	Michael	1978-11-23			
5	Williams	Daniel	1979-07-30			

Рисунок 2.3. Пример связи «один к одному»

В обеих таблицах существует первичный ключ (столбец **Id**), значения в этих столбцах должны быть одинаковыми для соответствующих записей в таблицах — у любого преподавателя может быть только одно сочетание логина и пароля для входа в систему выставления оценок — тем самым формируя связь «один к одному».

Связь такого типа используется в реляционных базах данных очень редко. Данный тип связи может быть реализован путем разделения одной таблицы на две, в том

случае если в таблице содержится большое количество столбцов, и информация по некоторым из них требуется крайне редко. На рисунке 2.3 продемонстрирована еще одна ситуация, которая может возникнуть, когда часть данных в таблице носит секретный характер, в этом случае секретные данные выносятся в отдельную таблицу, которой устанавливается более высокий уровень безопасности (будет рассмотрено в курсе MS SQL Server), тем самым ограничивая доступ к этой информации.

Связь «один ко многим» является наиболее часто используемым типом отношений и реализуется между двумя таблицами, когда строке в первой таблице соответствует множество строк во второй таблице, но строка второй таблицы должна быть связана только с одной строкой данных первой таблицы. Этот тип связи графически представляется в виде линии, один конец которой оканчивается трезубцем, называемым также «вороньей лапкой». На рисунке 2.4 приведен пример связи такого типа, между таблицами **Students** и **Groups**.

Students							Groups		
Id	FirstName	LastName	BirthDate	Grants	Email	GroupId	Id	GroupName	FacultyId
1	Jack	Jones	1997-11-05	1256.00	jj@net.eu	1	1	29PR21	1
2	Harry	Miller	1998-02-11	1100.00	hm@net.eu	1	2	30PR11	2
3	Grace	Evans	1997-06-24	NULL	eg@net.eu	2	3	31PPS11	1
4	Lily	Wilson	1998-09-12	NULL	lw@net.eu	2	4	32PR31	2
5	Joshua	Johnson	1997-05-23	1100.00	jo@net.eu	3	5	32PPS11	3
6	Emily	Taylor	1997-12-27	1100.00	et@net.eu	4			
7	Charlie	Thomas	1998-01-31	1256.00	ct@net.eu	4			
8	Oliver	Moore	1997-07-05	NULL	om@net.eu	4			
9	Jessica	Brown	1997-07-17	1100.00	jb@net.eu	5			

Рисунок 2.4. Пример связи «один ко многим»

Действительно в любой группе может учиться определенное количество студентов (множество), но каждый

2. Принципы создания многотабличной базы данных

конкретный студент находится только в одной группе. Например, в нашем случае в группе 32PR31 учатся несколько студентов: *Emily Taylor, Charlie Thomas и Oliver Moore*.

Связь «многие ко многим» может существовать между двумя таблицами реляционной базы данных в том случае, если строке первой таблицы соответствует множество строк со второй таблицы и строке во второй таблицы соответствует множество строк из первой таблицы. Этот тип связи графически представляется в виде пунктирной линии, оба конца которой оканчиваются трезубцем. Например, преподаватель может проводить занятия в нескольких группах и в одной группе могут преподавать определенное количество преподавателей. На рисунке 2.5 приведен пример связи такого типа, между таблицами *Teachers* и *Groups*.

The diagram shows two tables, 'Teachers' and 'Groups', connected by a dashed arrow pointing from the 'Groups' table to the 'Teachers' table. The 'Teachers' table has columns: ID, Name, Date of Birth, and Age. The 'Groups' table has columns: ID, Name, and Description.

ID	Name	Date of Birth	Age
4	Scooper	1998-11-23	53
3	Marcia Miller	1992-05-17	27
5	Kirk	1993-02-15	26
1	Natalie	1991-11-08	29
6	Willyiams	1995-07-20	24

ID	Name	Description
2	3566211	3
4	3566212	5
3	3121111	1
5	3008111	5
1	5694151	1
6	3100111	2

Рисунок 2.5. Пример связи «многие ко многим»

Данный тип связи является логическим и не может быть физически реализован в таком виде в базе данных. Поэтому для организации такого типа связи в базе данных используется следующий подход: между двумя таблицами, соединенными связью «многие ко многим», добавляется третья таблица, в которой находятся внешние ключи для первой и второй таблицы, тем самым

устанавливается связь «один ко многим» с каждой из этих таблиц. На основании вышеизложенного пример связи на рисунке 2.5 может быть реализован в следующем виде (рис. 2.6).

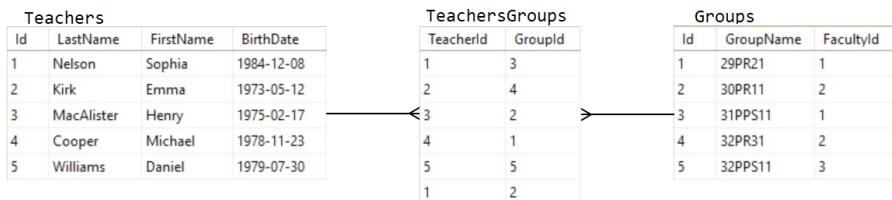


Рисунок 2.6. Реализация связи «многие ко многим»

В таблице **TeachersGroups** столбцы **TeacherId** и **GroupId** являются внешними ключами для первичных ключей (столбцы **Id**) таблиц **Teachers** и **Groups**, соответственно, благодаря чему формируются связи «один ко многим» между таблицами **TeachersGroups** и **Teachers**, и **TeachersGroups** и **Groups**.

После того как вы получили информацию о связях между таблицами было бы неплохо продемонстрировать вам как они формируются при помощи MS SQL Server Management Studio 17. Именно это мы сейчас и рассмотрим на примере связи «один ко многим» (как наиболее часто применяемой), которая показана на рисунке 2.4.

Для того чтобы сформировать связи между таблицами необходимо создать диаграмму базы данных, вызвав контекстное меню на пункте **Database Diagrams** в разделе **Databases** окна **Object Explorer** у требуемой базы данных (**University**), и выбрать в нем пункт **New Database Diagram** (рис. 2.7).

2. Принципы создания многотабличной базы данных

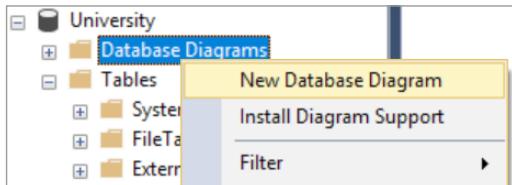


Рисунок 2.7. Создание диаграммы базы данных (начало)

После этого появится форма **Add Table** со списком всех таблиц, которые существуют в текущей базе данных. Для того чтобы добавить их на диаграмму необходимо выбрать в списке нужные таблицы и нажать кнопку **Add** (рис. 2.8).

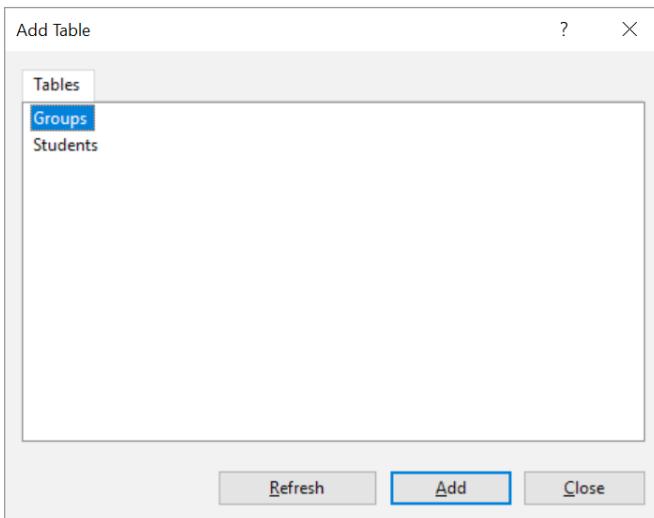


Рисунок 2.8. Добавление необходимых таблиц

В результате выполнения этих действий вы получите диаграмму требуемой базы данных. Нам осталось установить связи между таблицами, для того чтобы это сделать необходимо кликнуть левой клавишей мыши по

изображению ключа в поле первичного ключа **Id** таблицы **Groups** и не отпуская клавиши протянуть мышь на поле внешнего ключа **GroupId** таблицы **Students** и отпустить клавишу (рис. 2.9).

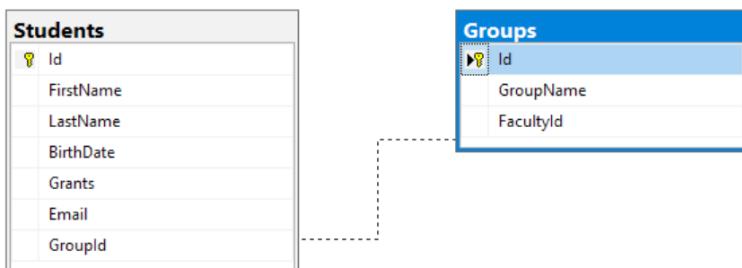


Рисунок 2.9. Создание связи между таблицами

Сразу же после этого вы увидите две формы **Tables and Columns** и **Foreign Key Relationship** (рис. 2.10).

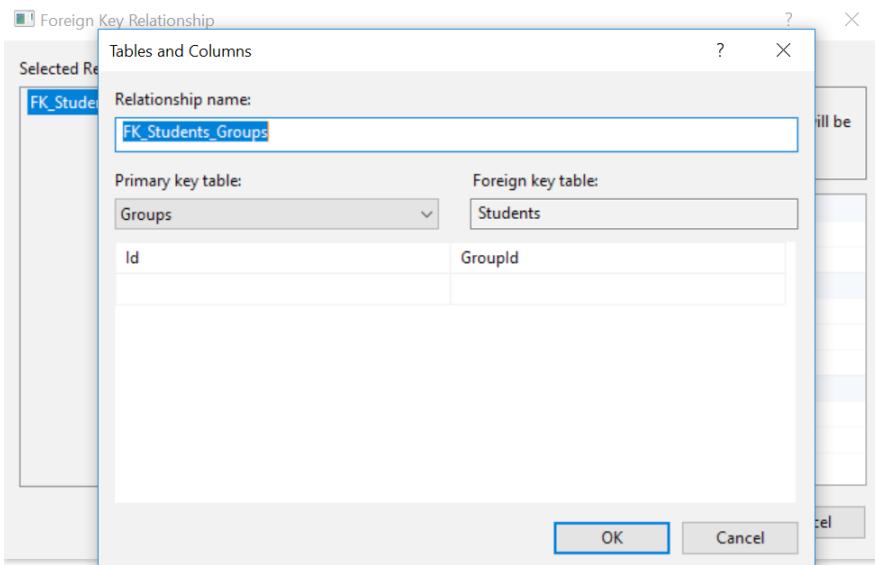


Рисунок 2.10. Формирование связи между таблицами

На форме **Tables and Columns** выводится информация о первичном и внешнем ключе и название отношения между ними, в случае необходимости вы можете подкорректировать любые данные, если вас все устраивает, тогда вы нажимаете кнопку ОК. Следующая форма **Foreign Key Relationship** позволяет более точно настроить само отношение, для того чтобы закончить формирования связи между таблицами необходимо нажать кнопку ОК. После этого отношение между таблицами будет сохранено в текущей базе данных, и вы увидите его на диаграмме базы данных, которая представлена на рисунке 2.1.

Целостность данных

При работе с различными базами данных существует вероятность записи в таблицы ошибочной информации, которая приведет к нарушению целостности данных и как результат к некорректной работе всей базы данных.

Язык T-SQL, как язык реляционной СУБД, поддерживает проверку входных данных и обеспечивает их целостность благодаря наличию ряда ограничений:

- **NOT NULL** — ограничение, которое гарантирует, что в текущий столбец каждой записи обязательно будут записаны данные, иначе попытка добавления записи в таблицу без этого значения приведет к ошибке;
- **DEFAULT** — ограничение, которое обеспечивает запись в столбец значения заданного по умолчанию в случае, если не указана другая информация;
- **CHECK** — ограничение, которое задает условие проверки вводимых данных, если условие не выполняется, то запись в таблицу не добавляется;

- **UNIQUE** — ограничение, которое обеспечивает уникальность записанной информации в соответствующем столбце;
- **PRIMARY KEY** — ограничение, которое указывает что текущий столбец (столбцы) является первичным ключом данной таблицы, данное ограничение может быть только одно в таблице и является комбинацией ограничений **UNIQUE** и **NOT NULL**;
- **FOREIGN KEY** — ограничение, которое указывает что текущий столбец (столбцы) является внешним ключом, данное ограничение обеспечивает ссылочную целостность данных.

На самом деле вам не обязательно указывать ограничения для столбцов таблицы, потому что целостность данных можно обеспечить и другими способами, например, при помощи триггеров (будут рассмотрены в курсе MS SQL Server) или на уровне приложения. Однако этот способ защиты данных является самым лучшим, потому что проверка ограничений выполняется сервером в первую очередь и естественно происходит быстрее всех других проверок.

Нормализация

Одной из самых важных концепций в реляционной базе данных является концепция нормализованных данных.

Необходимость нормализации

Нормализованные данные — это данные, организованные в такую структуру, которая обеспечивает целостность информации, хранящейся в базе данных, и сводит к минимуму количество избыточных данных. Структу-

рирование информации в базе данных делается не только с целью экономии памяти, но и для предотвращения появления несогласованных данных, связанных с различными аномалиями, а также для исключения потерь информации в процессе эксплуатации базы данных.

Нормализация данных осуществляется путем разбиения (декомпозиции) одной таблицы на две или более. При этом формируются необходимые связи между полученными таблицами с целью сохранения согласованности между данными.

Понятие нормальной формы

Процесс нормализации данных выполняется в соответствии с общепринятыми правилами нормализации, которые регламентируют порядок получения нормализованной базы данных и называются **нормальными формами**.

Впервые понятие нормальные формы было введено Эдгаром Коддом при описании им реляционной модели базы данных. Изначально было предложено три нормальные формы, но прогресс не стоит на месте и на текущий момент времени количество нормальных форм достигло уже восьми:

- первая нормальная форма (1NF);
- вторая нормальная форма (2NF);
- третья нормальная форма (3NF);
- нормальная форма Бойса-Кодда(BCNF);
- четвертая нормальная форма (4NF);
- пятая нормальная форма (5NF);
- доменно-ключевая нормальная форма (DKNF);
- шестая нормальная форма (6NF).

Перечисленные выше нормальные формы не являются обязательными для выполнения при проектировании базы данных, они скорее являются рекомендациями, которые вы вправе реализовать частично. При этом вам следует знать, что нормальные формы связаны между собой неразрывной последовательностью, в которой каждая следующая нормальная форма основана на предыдущей.

Существенным показателем качества спроектированной базы данных является ее производительность, которая напрямую связана со степенью нормализации данных — чем более высокой нормальной форме соответствует база данных, тем больше ресурсов системы необходимо для ее сопровождения. Поэтому спроектированная вами база данных должна соблюдать баланс между степенью нормализации данных и производительностью системы. Обычно чтобы достигнуть этой цели достаточно реализовать в базе данных первые четыре нормальные формы из приведенного выше списка.

Первая нормальная форма

Первая нормальная форма является наиболее важной и служит основой для всех остальных нормальных форм. Для того чтобы таблица соответствовала первой нормальной форме она должна отвечать следующим требованиям:

- каждое значение в записи должно быть атомарным (неделимым), то есть любой столбец в таблице должен содержать только одно значение для каждой строки;
- все записи в таблице должны быть разными, даже если в нескольких записях содержится одинаковая информа-

2. Принципы создания многотабличной базы данных

мация, то вся запись в целом должна быть уникальной для таблицы.

Последнее требование соответствует наличию в каждой таблице уникального поля — первичный ключ.

Таблица **Teachers** на рисунке 1.1 не соответствует требованиям первой нормальной формы, так как записи в столбце **Name** не являются атомарными, потому что содержат как имя, так и фамилию преподавателя, также в этой таблице отсутствует первичный ключ, что не гарантирует уникальности каждой записи (рис. 2.11).

Name	BirthDate	Department	Phone	Group	Subject
Sophia Nelson	1984-12-08	Software development	32-12	31PPS11	C#
Emma Kirk	1973-05-12	Mathematics	55-34	32PR31	Discrete Math
Henry MacAlister	1975-02-17	Software development	32-12	30PR11	SQL Server
Michael Cooper	1978-11-23	Software development	32-12	29PR21	ADO.NET
Daniel Williams	1979-07-30	Cybersecurity	37-65	32PPS11	ITE1
Sophia Nelson	1984-12-08	Software development	32-12	30PR11	JavaScript
Daniel Williams	1979-07-30	Cybersecurity	37-65	32PPS11	WIN10

Рисунок 2.11. Несоответствие таблицы
первой нормальной форме

Id	LastName	FirstName	BirthDate	Department	Phone	Group	Subject
1	Nelson	Sophia	1984-12-08	Software development	32-12	31PPS11	C#
2	Kirk	Emma	1973-05-12	Mathematics	55-34	32PR31	Discrete Math
3	MacAlister	Henry	1975-02-17	Software development	32-12	30PR11	SQL Server
4	Cooper	Michael	1978-11-23	Software development	32-12	29PR21	ADO.NET
5	Williams	Daniel	1979-07-30	Cybersecurity	37-65	32PPS11	ITE1
6	Nelson	Sophia	1984-12-08	Software development	32-12	30PR11	JavaScript
7	Williams	Daniel	1979-07-30	Cybersecurity	37-65	32PPS11	WIN10

Рисунок 2.12. Приведение таблицы
к первой нормальной форме

Для того чтобы привести таблицу `Teachers` в соответствие с первой нормальной формой необходимо внести ряд изменений: добавить первичный ключ (столбец `Id`) и создать отдельные столбцы для имени и фамилии (рис. 2.12).

Вторая нормальная форма

Прежде чем дать определение второй нормальной формы необходимо ввести ряд понятий, ключевым из которых является понятие функциональной зависимости.

Функциональная зависимость определяется следующим образом: при существовании двух полей **X** и **Y**, поле **X** функционально зависит от **Y**, если с любым значением поля **X** всегда связано ровно одно значение поля **Y**. Значения полей **X** и **Y** могут изменяться в течение времени, но только таким образом, чтобы любое поле **X** имело, связанное с ним уникальное значение **Y**. Обозначается функциональная зависимость следующим образом: **X** \rightarrow **Y**. В нашем случае можно привести следующие примеры функциональной зависимости: `LastName` \rightarrow `FirstName`, `LastName` \rightarrow `Department` и т.д.

Следующее понятие **полной функциональной зависимости** можно сформулировать следующим образом: если ключевое поле **X** состоит из нескольких полей, то неключевое поле **Y** должно функционально полно зависеть от **X** в целом и не зависеть функционально от какого-либо поля, входящего в него. Например, если в таблице `Teachers` первичный ключ состоит из столбцов `LastName` и `FirstName`, то столбец `BirthDate` будет находиться в полной функциональной зависимости от первичного ключа.

Частичной функциональной зависимостью, соответственно, называется зависимость неключевого поля от части составного ключевого поля.

Неключевое поле — это столбец, который не входит в состав ни одного из возможных потенциальных ключей.

Требования второй нормальной формы расширяют первую нормальную форму, беря ее за основу:

- таблица должна выполнять условия первой нормальной формы;
- все столбцы, которые не входят в первичный ключ, должны зависеть от первичного ключа в целом, то есть должны быть связаны полной функциональной зависимостью с первичным ключом.

Последнее требование соответствует отсутствию в таблице частичных функциональных зависимостей.

Если предположить, что в таблице **Teachers** первичный ключ состоит из столбцов **LastName**, **Group** и **Subject**, тогда столбцы **FirstName**, **BirthDate**, **Department** и **Phone** образуют частичную функциональную зависимость, так как они зависят от столбца **LastName** и не зависят от столбцов **Group** и **Subject** (рис. 2.13).

Id	LastName	FirstName	BirthDate	Department	Phone	Group	Subject
1	Nelson	Sophia	1984-12-08	Software development	32-12	31PPS11	C#
2	Kirk	Emma	1973-05-12	Mathematics	55-34	32PR31	Discrete Math
3	MacAlister	Henry	1975-02-17	Software development	32-12	30PR11	SQL Server
4	Cooper	Michael	1978-11-23	Software development	32-12	29PR21	ADO.NET
5	Williams	Daniel	1979-07-30	Cybersecurity	37-65	32PPS11	ITE1
6	Nelson	Sophia	1984-12-08	Software development	32-12	30PR11	JavaScript
7	Williams	Daniel	1979-07-30	Cybersecurity	37-65	32PPS11	WIN10

Рисунок 2.13. Несоответствие таблицы второй нормальной форме

Данная проблема решается путем разбиения существующей таблицы **Teachers** на две таблицы следующим образом:

- в первой таблице (**Teachers**) будут содержаться все столбцы, которые находятся в функциональной зависимости от части первичного ключа вместе с этой частью;
- во второй таблице (**GroupsSubjects**) будут находиться остальные части первичного ключа и зависящие от них столбцы (рис. 2.14).

Teachers						GroupsSubjects		
Id	LastName	FirstName	BirthDate	Department	Phone	Id	Group	Subject
1	Nelson	Sophia	1984-12-08	Software development	32-12	1	31PPS11	C#
2	Kirk	Emma	1973-05-12	Mathematics	55-34	2	32PR31	Discrete Math
3	MacAlister	Henry	1975-02-17	Software development	32-12	3	30PR11	SQL Server
4	Cooper	Michael	1978-11-23	Software development	32-12	4	29PR21	ADO.NET
5	Williams	Daniel	1979-07-30	Cybersecurity	37-65	5	32PPS11	ITE1
6	Nelson	Sophia	1984-12-08	Software development	32-12	6	30PR11	JavaScript
7	Williams	Daniel	1979-07-30	Cybersecurity	37-65	7	32PPS11	WIN10

Рисунок 2.14. Приведение таблиц ко второй нормальной форме

Третья нормальная форма

С понятием третьей нормальной формы напрямую связано понятие транзитивной функциональной зависимости.

Допустим, в таблице существуют три поля **X**, **Y** и **Z**, тогда функциональная зависимость между полями **X** и **Z** ($X \rightarrow Z$) называется **транзитивной**, если существуют зависимости между полями **X** и **Y** ($X \rightarrow Y$) и полями **Y** и **Z** ($Y \rightarrow Z$).

Требования третьей нормальной формы основаны на второй нормальной форме и звучат следующим образом:

2. Принципы создания многотабличной базы данных

- таблица должна соответствовать требованиям второй нормальной формы;
- все столбцы, не входящие в первичный ключ, должны функционально зависеть только от первичного ключа и не зависеть от любого другого неключевого столбца.

Последнее требование соответствует отсутствию транзитивных зависимостей в таблице. Например, в таблице **Teachers** присутствует транзитивная зависимость между столбцами **Id** и **Phone**, так как существуют функциональные зависимости **Id -> Department** и **Department -> Phone**, потому что номер телефона зависит от кафедры (рис. 2.15).

Id	LastName	FirstName	BirthDate	Department	Phone
1	Nelson	Sophia	1984-12-08	Software development	32-12
2	Kirk	Emma	1973-05-12	Mathematics	55-34
3	MacAlister	Henry	1975-02-17	Software development	32-12
4	Cooper	Michael	1978-11-23	Software development	32-12
5	Williams	Daniel	1979-07-30	Cybersecurity	37-65
6	Nelson	Sophia	1984-12-08	Software development	32-12
7	Williams	Daniel	1979-07-30	Cybersecurity	37-65

Рисунок 2.15 Несоответствие таблицы третьей нормальной форме

Для того чтобы таблица **Teachers** соответствовала третьей нормальной форме необходимо осуществить ее разбиение на две таблицы, вынеся столбцы, которые являются причиной транзитивной зависимости, в отдельную таблицу (**Departments**) (рис. 2.16).

Teachers				Departments		
Id	LastName	FirstName	BirthDate	Id	Department	Phone
1	Nelson	Sophia	1984-12-08	1	Software development	32-12
2	Kirk	Emma	1973-05-12	2	Mathematics	55-34
3	MacAlister	Henry	1975-02-17	3	Cybersecurity	37-65
4	Cooper	Michael	1978-11-23			
5	Williams	Daniel	1979-07-30			

Рисунок 2.16. Приведение таблиц к третьей нормальной форме

Нормальная форма Бойса-Кодда

Нормальная форма Бойса-Кодда иногда называют усиленной третьей нормальной формой, потому что она накладывает большие ограничения на функциональные зависимости между столбцами и применяется при сочетании нескольких условий: в таблице должно быть два или более составных потенциальных ключа при этом они должны перекрываться, то есть иметь хотя бы один общий столбец. При невыполнении указанных выше условий нормальная форма Бойса-Кодда эквивалентна третьей нормальной форме.

Требования нормальной формы Бойса-Кодда берут за основу требования третьей нормальной формы, расширяя их:

- таблица должна соответствовать требованиям третьей нормальной формы;
- столбцы, входящие в первичный ключ, не должны иметь функциональных зависимостей с любым из неключевых столбцов.

Из последнего требования следует, что в таблице должны отсутствовать перекрывающиеся потенциаль-

2. Принципы создания многотабличной базы данных

ные первичные ключи. В качестве примера рассмотрим таблицу **Achievements** (рис. 2.17).

StudentId	RecordBookNumber	Subject	Assesment
1	07-15-18	ADO.NET	11
2	12-23-56	C#	10
1	07-15-18	SQL Server	10
3	67-34-61	C#	8
2	12-23-56	ADO.NET	9
3	67-34-61	SQL Server	7

Рисунок 2.17. Таблица Achievements

Данная таблица соответствует второй и третьей нормальным формам, так как в ней отсутствуют частичные и транзитивные зависимости. Однако таблица **Achievements** не соответствует нормальной форме Бойса-Кодда потому что в ней существуют два потенциальных составных ключа. Первый состоит из столбцов **StudentId** (*идентификатор студента*) и **RecordBookNumber** (*номер зачетки*), а второй из столбцов **StudentId** и **Subject**, при этом эти потенциальные ключи перекрываются, потому что столбец **StudentId** является частью обоих ключей (рис. 2.18).

StudentId	RecordBookNumber	Subject	Assesment
1	07-15-18	ADO.NET	11
2	12-23-56	C#	10
1	07-15-18	SQL Server	10
3	67-34-61	C#	8
2	12-23-56	ADO.NET	9
3	67-34-61	SQL Server	7

Рисунок 2.18. Несоответствие таблицы нормальной форме Бойса-Кодда

Для того чтобы привести таблицу *Achievements* в соответствие с нормальной формой Бойса-Кодда необходимо на ее основе создать две таблицы *RecordBooks* и *Assesments*, в каждой из которых столбец *StudentId* будет первичным ключом (рис. 2.19).

RecordBooks		Assesments		
StudentId	RecordBookNumber	StudentId	Subject	Assesment
1	07-15-18	1	ADO.NET	11
2	12-23-56	2	C#	10
1	07-15-18	1	SQL Server	10
3	67-34-61	3	C#	8
2	12-23-56	2	ADO.NET	9
3	67-34-61	3	SQL Server	7

Рисунок 2.19. Реализация нормальной формы Бойса-Кодда

После того как вы привели таблицы вашей базы данных к требуемой нормальной форме, необходимо установить связи между таблицами, которые уже были описаны в текущем разделе ранее.

3. Многотабличные запросы

Мы надеемся, что к текущему времени вы осознали всю важность проектирования именно многотабличных баз данных. Однако одновременно с этим приходит необходимость получения данных из различных таблиц, то есть написания многотабличных запросов, которые в простонародье называют «сложными».

Принципы создания многотабличного запроса

Многотабличные запросы позволяют получить различную информацию из взаимосвязанных таблиц, результаты при этом, как обычно, будут отображаться в виртуальной таблице.

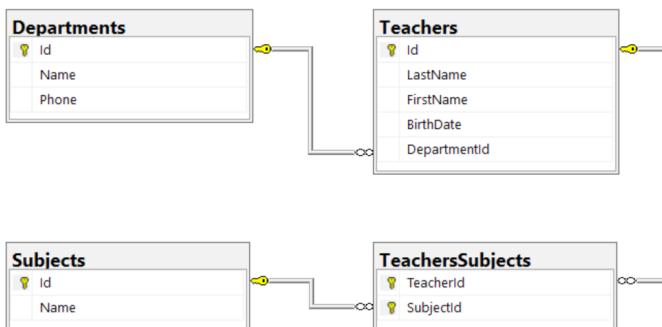


Рисунок 3.1 Диаграмма базы данных

При написании многотабличных запросов вы можете использовать все операторы языка SQL, рассмотренные нами в уроке № 3, однако существует несколько особенностей, которые необходимо всегда учитывать.

Для демонстрации написания многотабличных запросов мы воспользуемся некоторыми таблицами из тех, которые мы рассматривали в предыдущем разделе, установив между ними необходимые связи. Для того чтобы вы лучше ориентировались в этих связях, на рисунке 3.1 приведена диаграмма полученной базы данных.

На приведенной диаграмме между таблицами `Departments` и `Teachers` существует связь «один ко многим» (на кафедре может быть множество преподавателей, но отдельно взятый преподаватель может работать только на одной кафедре), которая формируется благодаря наличию внешнего ключа `DepartmentId` в таблице `Teachers`. Между таблицами `Teachers` и `Subjects` существует связь «многие ко многим» (один преподаватель может читать множество предметов и один предмет могут вести множество преподавателей), поэтому была создана третья таблица `TeachersSubjects`, в которой находятся внешние ключи `TeacherId` и `SubjectId`, которые устанавливают связь «один ко многим» с таблицами `Teachers` и `Subjects` соответственно. Как вы, наверное, заметили, в таблице `TeachersSubjects` оба столбца формируют составной первичный ключ, тем самым обеспечивается уникальность каждой записи в этой таблице.

Для того чтобы получить информацию о принадлежности преподавателей к кафедрам можно было написать SQL-запрос к одной таблице `Teachers`, но в этом случае мы получим только идентификаторы кафедр, что является не особо информативным. Поэтому необходимо написать SQL-запрос, при помощи которого мы будем получать информацию из двух таблиц `Departments` и `Teachers`:

3. Многотабличные запросы

```
SELECT FirstName + ' ' + LastName AS FullName, Name  
FROM Departments, Teachers  
WHERE Teachers.DepartmentId = Departments.Id;
```

Результат, полученный после выполнения этого SQL-запроса, отображен на рисунке 3.2.

FullName	Name
Sophia Nelson	Software development
Emma Kirk	Mathematics
Henry MacAlister	Software development
Michael Cooper	Software development
Daniel Williams	Cybersecurity

Рисунок 3.2. Многотабличный запрос

Как вы заметили, для того чтобы написать многотабличный запрос необходимо соединить требуемые таблицы, указав столбцы, которые формируют соответствующую связь, в операторе **WHERE**.

Допустим, в следующем запросе мы помимо имеющейся информации захотим получить еще и идентификатор кафедры, но, если мы попытаемся в SQL-запросе указать столбец, название которого присутствует в обеих таблицах, то выполнение этого запроса приведет к ошибке (рис. 3.3).

The screenshot shows a SQL query window titled "SQLQuery1.sql - DE...ESKTOP\Yuriy (54)*". The query is:

```
SELECT Id, FirstName + ' ' + LastName AS FullName, Name  
FROM Departments, Teachers  
WHERE Teachers.DepartmentId = Departments.Id;
```

In the "Messages" pane at the bottom, there is an error message:

```
Msg 209, Level 16, State 1, Line 1  
Ambiguous column name 'Id'.
```

Рисунок 3.3. Ошибка: двусмысленность при использовании столбца Id

Данная ошибка произошла из-за наличия столбца с одинаковым названием (`Id`) и в таблице `Departments`, и в таблице `Teachers`. Для того чтобы устраниТЬ ошибки, связанные с одинаковым названием столбцов в различных таблицах, необходимо указывать полное имя столбца, которое состоит из названий таблицы и столбца, разделенных точкой, например, в нашем случае: `Departments.Id`.

Также вместо полного имени таблицы можно назначить ей псевдоним при помощи оператора `AS`, который рассматривался нами в уроке №3. Применим полученные знания на практике и перепишем ошибочный SQL-запрос:

```
SELECT FirstName + ' ' + LastName AS FullName,
       Name, D.Id AS DeptId
  FROM Departments AS D, Teachers AS T
 WHERE T.DepartmentId = D.Id;
```

Результат этого SQL-запроса представлен на рисунке 3.4.

FullName	Name	DeptId
Sophia Nelson	Software development	1
Emma Kirk	Mathematics	2
Henry MacAlister	Software development	1
Michael Cooper	Software development	1
Daniel Williams	Cybersecurity	3

Рисунок 3.4 Использование псевдонимов

При помощи следующего SQL-запроса мы хотим определить какие предметы ведет каждый из преподавателей, в этом случае нам необходимо связать между

себой таблицы `Teachers` и `Subjects`, используя таблицу `TeachersSubjects`:

```
SELECT FirstName + ' ' + LastName AS FullName,
       Name AS SubjectName
  FROM Teachers AS T, Subjects AS S,
       TeachersSubjects AS TS
 WHERE T.Id=TS.TeacherId AND S.Id=TS.SubjectId;
```

Результат выполнения данного SQL-запроса представлен на рисунке 3.5.

FullName	SubjectName
Sophia Nelson	C#
Sophia Nelson	JavaScript
Emma Kirk	Discrete Math
Henry MacAlister	SQL Server
Michael Cooper	ADO.NET
Daniel Williams	ITE1
Daniel Williams	WIN10

Рисунок 3.5. Получение информации «Преподаватели-Предметы»

При помощи заключительного SQL-запроса мы хотим получить информацию о том какие предметы читаются на той либо иной кафедре, для чего нам придется установить связь между всеми имеющимися таблицами:

```
SELECT D.Name AS DeptName, S.Name
       AS SubjectName
  FROM Departments AS D, Teachers AS T, Subjects
       AS S, TeachersSubjects AS TS
 WHERE D.Id = T.DepartmentId AND T.Id=TS.
       TeacherId AND S.Id=TS.SubjectId;
```

Результат этого SQL-запроса представлен на рисунке 3.6.

DeptName	SubjectName
Software development	C#
Software development	JavaScript
Mathematics	Discrete Math
Software development	SQL Server
Software development	ADO.NET
Cybersecurity	ITE1
Cybersecurity	WIN10

Рисунок 3.6. Получение информации «Кафедры-Предметы»

Декартово произведение

При написании многотабличных запросов особое внимание следует уделять установлению взаимосвязи между таблицами, вы уже видели это в предыдущих примерах, например: `Teachers.DepartmentId = Departments.Id` — равенство соответствующего первичного и внешнего ключей. Такое сравнение является критически важным, если его не указывать, то в результате SQL-запроса вы получите всевозможные сочетания записей одной таблицы со всевозможными записями из другой таблицы, полученное множество называется **декартовым произведением** этих таблиц. Продемонстрируем это на примере:

```
SELECT FirstName + ' ' + LastName
  AS FullName, Name
FROM Departments, Teachers;
```

В результате выполнения этого SQL-запроса у нас получится, что все преподаватели одновременно работают

3. Многотабличные запросы

на всех кафедрах, однако это не соответствует действительности. Полученный результат частично отображен на рисунке 3.7.

FullName	Name
Sophia Nelson	Software development
Emma Kirk	Software development
Henry MacAlister	Software development
Michael Cooper	Software development
Daniel Williams	Software development
Sophia Nelson	Mathematics
Emma Kirk	Mathematics
Henry MacAlister	Mathematics

Рисунок 3.7. Декартово произведение таблиц

4. Домашнее задание

1. В домашнем задании к уроку №3 вам необходимо было создать однотабличную базу данных, в которой должна содержаться произвольная информация о некой виртуальной больнице. Мы предлагаем вам:

- a. Создать на ее основе многотабличную базу данных, которая должна соответствовать нормальной форме Бойса-Кодда;
- b. Создать диаграмму этой базы данных;
- c. Написать многотабличные SQL-запросы к этой базе данных:
 - вывести информацию обо всех пациентах, находящихся в больнице;
 - показать данные о пациентах, которые лежат в определенном отделении;
 - получить данные о пациентах, которые лежат в больнице больше месяца, отсортировав их по возрастанию даты поступления;
 - вывести информацию о пациентах, которые были выписаны в прошлом месяце;
 - показать информацию о пациентах, которые лежали в больнице с октября по декабрь прошлого года в определенном отделении;
 - вывести данные о пациентах, которых лечит определенный врач с одинаковыми заболеваниями.

Unit 5.

Функции агрегирования

1. Функции агрегирования

Как вы знаете, основным предназначением баз данных является упорядоченное хранение различной информации, которая может быть извлечена удобным способом в случае необходимости. Однако существует еще одно немаловажное применение баз данных — статистический анализ хранимой информации. Для того, чтобы получить статистические данные в цифровом виде используются функции агрегирования, которые предназначены для получения обобщающего значения из определенного количества столбцов, то есть результатом выполнения любой функции будет единственное значение.

Id	FirstName	LastName	BirthDate	Grants	Email	GroupId
1	Jack	Jones	1997-11-05	1256.00	jj@net.eu	1
2	Harry	Miller	1998-02-11	1100.00	hm@net.eu	1
3	Grace	Evans	1997-06-24	NULL	eg@net.eu	2
4	Lily	Wilson	1998-09-12	NULL	lw@net.eu	2
5	Joshua	Johnson	1997-05-23	1100.00	jo@net.eu	3
6	Emily	Taylor	1997-12-27	1100.00	et@net.eu	4
7	Charlie	Thomas	1998-01-31	1256.00	ct@net.eu	4
8	Oliver	Moore	1997-07-05	NULL	om@net.eu	4
9	Jessica	Brown	1997-07-17	1100.00	jb@net.eu	5

Рисунок 1.1. Таблица Students

В языке T-SQL существует пять функций агрегирования: **COUNT()**, **AVG()**, **SUM()**, **MIN()** и **MAX()**. Прежде чем приступить к их подробному рассмотрению, следует отметить немаловажную особенность — функции

AVG() и **SUM()** могут применяться только к столбцам цифровых типов данных, а функции **COUNT()**, **MIN()** и **MAX()** как к полям цифровых, так и символьного типа данных.

Для того, чтобы продемонстрировать практическое применение функций агрегирования, мы воспользуемся таблицей **Students**, известной вам из предыдущего урока (рис. 1.1).

Функция COUNT

Функция **COUNT()** позволяет определить количество записей в определенном столбце либо во всей таблице.

Если вызвать функцию **COUNT()** и передать в качестве параметра символ ***** (звездочка):

```
SELECT COUNT(*) AS [Number of records]  
FROM Students;
```

То в результате вы получите общее количество записей в таблице, включая как повторяющиеся, так и неопределенные значения (NULL-значения) (рис. 1.2).

Number of records
9

Рисунок 1.2. Определение общего количества записей в таблице

Для того чтобы получить количество записей в конкретном столбце таблицы, при вызове функции **COUNT()** необходимо передать в качестве параметра название требуемого столбца. При использовании функции в таком виде не учитываются записи, имеющие неопределенное

значение в данном столбце. Например, SQL-запрос возвращающий количество записей в столбце **Grants** будет выглядеть следующим образом:

```
SELECT COUNT(Grants) AS [Number of grants]
FROM Students;
```

Результат, полученный при выполнении этого SQL-запроса, отличается от результата, представленного на рисунке 1.2, так как в столбце **Grants** имеются NULL-значения (рис. 1.3).

Number of grants
6

Рисунок 1.3. Количество записей в столбце Grants

В том случае если вам требуется определить количество уникальных записей в определенном столбце, тогда при вызове функции **COUNT()** необходимо указать ключевое слово **DISTINCT** перед именем столбца. Внесем изменения в предыдущий пример:

```
SELECT COUNT(DISTINCT Grants) AS [Unique of grants]
FROM Students;
```

В результате выполнения данного SQL-запроса мы получим количество записей в столбце **Grants** без повторений (рис. 1.4).

Unique of grants
2

Рисунок 1.4. Количество записей в столбце Grants без повторений

Функция AVG

Функция **AVG()** позволяет получить среднее арифметическое значений определенного столбца.

Использование функции **AVG()** рассмотрим на простом примере нахождения размера средней стипендии студентов, соответствующий SQL-запрос будет выглядеть следующим образом:

```
SELECT AVG(Grants) AS [Average grant]
FROM Students;
```

Результат выполнения данного запроса представлен на рисунке 1.5.

Average grant
1152.000000

Рисунок 1.5. Средняя стипендия студентов

Следующий пример будет немного сложнее, допустим нам необходимо определить средний возраст студентов, тогда мы можем написать следующий SQL-запрос:

```
SELECT AVG(DATEDIFF(dd,BirthDate,
GETDATE())/365.25) AS [Average age]
FROM Students;
```

Для того чтобы определить возраст студентов мы используем функцию **DATEDIFF()**, получая разницу в днях между текущей датой и датой рождения студента, производя деление полученного результата на 365.25 мы получаем более точные значения (рис. 1.6).

Average age
20.789717

Рисунок 1.6. Средний возраст студентов

Функция SUM

Функция **SUM()** позволяет вычислить сумму значений заданного столбца.

Например, выполнив следующий SQL-запрос, мы сможем узнать, сколько денег выплачивается студентам ежемесячно в качестве стипендии, результат представлен на рисунке 1.7.

```
SELECT SUM(Grants) AS [Sum grants]
FROM Students;
```

Sum grants
6912.00

Рисунок 1.7. Сумма стипендий студентов

Функция MIN

Использование функции **MIN()** обеспечивает получение самого малого значения в определенном столбце таблицы.

Для того чтобы определить наименьшую дату рождения студентов необходимо выполнить следующий запрос, результат которого представлен на рисунке 1.8:

```
SELECT MIN(BirthDate) AS [Min date of birth]
FROM Students;
```

Min date of birth
1997-05-23

Рисунок 1.8. Минимальная дата рождения студентов

Функция MAX

Функция **MAX()** позволяет определить наибольшее значение в указанном столбце.

Как говорилось ранее, функции **MIN()** и **MAX()** могут использоваться при работе со столбцами символьного типа, в этом случае будут сравниваться цифровые коды соответствующих символов. Например, нам необходимо определить максимальное значение из фамилий студентов, то есть найти фамилию, у которой значение символов наибольшее. Ведь на самом деле будут последовательно сравниваться не сами символы, а их числовые значения в кодировке Unicode, так как в нашем случае столбец **Last-Name** имеет тип данных **nvarchar**. Для того чтобы решить эту поставленную задачу можно написать следующий SQL-запрос:

```
SELECT MAX(LastName) AS [Maximum last name]
FROM Students;
```

Результат запроса представлен ниже (рис. 1.9).

Maximum last name
Wilson

Рисунок 1.9. Максимальная фамилия из списка студентов

При написании SQL-запросов с использованием функций агрегирования, при необходимости, можно осуществлять ограничение результата по условию, например, нам нужно получить количество студентов, имя которых начинается на букву **J**. Выполнив следующий запрос, вы получите требуемый результат (рис. 1.10):

```
SELECT COUNT(*) AS [Number of students]
FROM Students
WHERE FirstName LIKE 'J%';
```

Number of students
3

Рисунок 1.10. Количество студентов, имя которых начинается на букву J

Для того чтобы у вас не сложилось впечатление, что функции агрегирования применяются только в простых запросах мы, в заключение текущего раздела, продемонстрируем пример многотабличного SQL-запроса. Для того чтобы освежить вашу память, мы приведем часть диаграммы базы данных из прошлого урока (рис. 1.11).

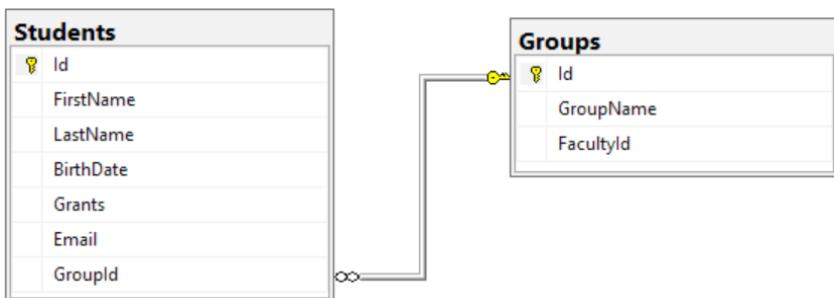


Рисунок 1.11. Связь между таблицами Students и Groups

Допустим нам необходимо узнать количество студентов, которые учатся на 32 потоке:

```

SELECT COUNT(*) AS [Count of students]
FROM Students AS S, Groups AS G
WHERE G.Id = S.GroupId
AND G.GroupName LIKE '32%';

```

В текущем запросе для того чтобы получить из базы данных нужную информацию необходимо установить

1. Функции агрегирования

связь между таблицами Students и Groups по первично-му/внешнему ключу (`G.Id = S.GroupId`) и оставить только те записи, которые начинаются на 32 (`G.GroupName LIKE '32%`), после чего получить их количество при помощи функции `COUNT()`. Результат данного запроса представлен на рисунке 1.12.

Count of students
4

Рисунок 1.12. Количество студентов 32-го потока

2. Понятие группировки. Ключевое слово GROUP BY

Все примеры предыдущего раздела возвращали в качестве результатов SQL-запросов только значения функций агрегирования. А что произойдет, если нам понадобится дополнительная информация? Например, необходимо узнать какое количество студентов учится в каждой группе.

«Что тут сложного? — скажете вы. — Нужно добавить в оператор **SELECT** название группы». Давайте попробуем написать соответствующий запрос, выполнение которого завершится ошибкой (рис. 2.1).

The screenshot shows a SQL query window titled "SQLQuery1.sql - DE...ESKTOP\Yuriy (56)*". The query is:

```
SELECT GroupName, COUNT(S.GroupId) AS [Number of students]
FROM Groups AS G, Students AS S
WHERE G.Id = S.GroupId;
```

In the "Messages" pane below, there is an error message:

```
Msg 8120, Level 16, State 1, Line 1
Column 'Groups.GroupName' is invalid in the select list because it is
not contained in either an aggregate function or the GROUP BY clause.
```

Рисунок 2.1. Ошибка: недопустимо использовать столбец **GroupName** в списке выбора

Ошибка произошла потому что любая функция агрегирования (в данном случае **COUNT()**) всегда возвращает одно единственное значение, а обращение к столбцу **GroupName** в операторе **SELECT** возвращает множество значений строк и такая ситуация не может быть корректно обработана оператором **SELECT**. Если представить

2. Понятие группировки. Ключевое слово GROUP BY

невозможный результат выполнения этого SQL-запроса, то выглядел бы он следующим образом (рис. 2.2).

GroupName	Number of students
29PR21	2
29PR21	2
30PR11	2
30PR11	2
31PPS11	1
32PR31	
32PR31	3
32PR31	
32PPS11	1

Рисунок 2.2. Невозможный результат использования функций агрегирования

Как вы можете заметить, некоторым названиям группы соответствует одно значение функции агрегирования, а это в принципе невозможно.

Для того чтобы данный SQL-запрос выполнился правильно, необходимо сделать так чтобы название группы тоже было в единственном числе, то есть каким-то образом сгруппировать значения в столбце **GroupName** (рис. 2.3).

GroupName	Number of students
29PR21	2
30PR11	2
31PPS11	1
32PPS11	1
32PR31	3

Рисунок 2.3. Требуемый результат выполнения запроса

Намек на решение данной проблемы сделан в тексте, который описывает ошибку — необходимо использовать

оператор **GROUP BY**, который позволяет группировать результирующие строки по указанному столбцу:

```
SELECT GroupName, COUNT(S.GroupId)
    AS [Number of students]
  FROM Groups AS G, Students AS S
 WHERE G.Id = S.GroupId
 GROUP BY GroupName;
```

Использование оператора **GROUP BY** в предыдущем запросе позволило создать группы строк, по одной для каждого уникального значения в столбце **GroupName**. После чего при помощи функции **COUNT()** было получено количество записей в каждой группе строк и уже итоговые строки были включены в результирующую таблицу. Результат выполнения этого SQL-запроса совпадает с ожидаемым (рис. 2.3).

При необходимости вы можете группировать строки на основе нескольких столбцов, однако в этом случае группирование будет осуществляться, основываясь на уникальном сочетании данных во всех столбцах, которые участвуют в группировке.

Например, если нам понадобиться получить количество студентов с одинаковой стипендией по группам с отображением этих стипендий, то в группировке данных помимо названий групп будут участвовать еще и значения стипендий:

```
SELECT GroupName, Grants, COUNT(S.GroupId)
    AS [Number of students]
  FROM Groups AS G, Students AS S
 WHERE G.Id = S.GroupId
 GROUP BY GroupName, Grants;
```

В результате выполнения данного SQL-запроса мы получим количество студентов в группе больше одного только тогда, когда размер стипендии совпадет у нескольких студентов в этой группе. В нашем случае в группе 30PR11 два студента не получают стипендию, тем самым формируя для запроса одинаковое значение — **NULL**, это происходит потому что функция **GROUP BY** интерпретирует все NULL-значения как равные. Во всех остальных группах значения стипендий у студентов не совпадает (рис. 2.4).

GroupName	Grants	Number of students
30PR11	NULL	2
32PR31	NULL	1
29PR21	1100.00	1
31PPS11	1100.00	1
32PPS11	1100.00	1
32PR31	1100.00	1
29PR21	1256.00	1
32PR31	1256.00	1

Рисунок 2.4. Количество студентов в каждой из групп с одинаковой стипендией

3. Ключевое слово HAVING

При использовании функций агрегирования существует необходимость накладывать ограничения на возвращаемые ими результаты, именно это является одним из предназначений оператора **HAVING**. Еще одна возможность применения этого оператора — фильтрация результатов группировки строк на основании значений столбцов, указанных после оператора **GROUP BY**.

Принципы использования HAVING

Оператор **HAVING** синтаксически прописывается после оператора **GROUP BY**, но до оператора **ORDER BY**:

```
SELECT columnName1, columnName2, ...
FROM tableName
[WHERE condition]
[GROUP BY columnName1, columnName2, ...]
HAVING condition
[ORDER BY columnName1 ASC | DESC, ...];
```

В действительности наличие операторов **WHERE**, **GROUP BY** и **ORDER BY** является необязательным.

Приведем несколько примеров использования оператора **HAVING**. В первом из них запрос вернет только тех студентов, средняя стипендия которых не превышает 1200, при этом результат будет отсортирован по их фамилиям:

```
SELECT LastName, Grants
FROM Students
```

3. Ключевое слово HAVING

```
GROUP BY LastName, Grants  
HAVING AVG(Grants) <= 1200  
ORDER BY LastName;
```

Данный SQL-запрос будет выполняться следующим образом: все записи из таблицы `Students` будут сгруппированы по фамилиям и стипендиям студентов (оператор `GROUP BY`), потом полученные группы фильтруются по условию в операторе `HAVING` и после этого полученные результаты сортируются по фамилиям (рис. 3.1).

LastName	Grants
Brown	1100.00
Johnson	1100.00
Miller	1100.00
Taylor	1100.00

Рисунок 3.1. Студенты со средней стипендией меньше 1200

При помощи следующего SQL-запроса мы сможем определить название групп, количество студентов в которых превышает 2 человека:

```
SELECT GroupName, COUNT(*)  
AS [Number of students]  
FROM Groups AS G, Students AS S  
WHERE G.Id = S.GroupId  
GROUP BY GroupName  
HAVING COUNT(S.GroupId)>2;
```

В этом запросе соединяем таблицы `Students` и `Groups` по первичному/внешнему ключу (`G.Id = S.GroupId`), группируем полученные строки по названию группы, после чего ограничиваем количество записей по условию в опе-

раторе **HAVING**, сравнивая количество идентификаторов групп у студентов (**COUNT(S.GroupId)>2**). Результат данного запроса представлен на рисунке 3.2.

GroupName	Number of students
32PR31	3

Рисунок 3.2. Группы, в которых учатся больше 2 человек

Еще один SQL-запрос демонстрирует возможность проверки в операторе **HAVING** значений одного из столбцов, по которым группируются данные. Допустим необходимо вывести имена и фамилии студентов из определенного списка:

```
SELECT FirstName, LastName
FROM Students
GROUP BY LastName, FirstName
HAVING LastName IN ('Moore', 'Thomas', 'Doe');
```

В текущем запросе записи группируются по фамилии и имени, потом остаются только те строки, у которых фамилии совпадают с заданным в операторе **HAVING** списком и только эта информация попадает в результатирующую таблицу (рис. 3.3).

FirstName	LastName
Oliver	Moore
Charlie	Thomas

Рисунок 3.3. Информация по студентам с определенными фамилиями

Последний запрос демонстрирует возможность использования оператора **HAVING** при отсутствии опера-

тора **GROUP BY**. Вывести минимальные значения фамилий студентов, если средняя стипендия превышает 1100:

```
SELECT MIN(LastName) AS [Minimum last name]  
FROM Students  
HAVING AVG(Grants)>1100;
```

В операторе **SELECT** предыдущего SQL-запроса мы можем вызывать только функции агрегирования, если попытаться указать название любого столбца, то будет сгенерирована ошибка, с требованием использовать оператор **GROUP BY**. Результат выполнения этого запроса представлен на рисунке 3.4.

Minimum last name
Brown

Рисунок 3.4. Использование оператора **HAVING** без **GROUP BY**

Если в операторе **HAVING** изменить условие — увеличить сравниваемое значение, например, до 1400, то мы получим пустой результирующий набор, потому что таких средних значений в таблице **Students** нет.

Сравнительный анализ **HAVING** и **WHERE**

Операторы **WHERE** и **HAVING** похожи по способу их использования и тот, и другой содержит условия фильтрации данных, однако эти операторы отличаются своим назначением. Условия после оператора **WHERE** определяют, каким образом соединять используемые таблицы и фильтруют отдельные строки данных, тем самым ограничивая количество выводимых записей в резуль-

тирующей таблице. В свою очередь оператор **HAVING** предназначен для фильтрации сгруппированных данных или выполнения условий для функций агрегирования, именно поэтому использование оператора **WHERE** в данных ситуациях невозможно. Например, если мы напишем SQL-запрос с использованием оператора **WHERE** для того чтобы получить информацию о студентах, средняя стипендия которых не превышает 1200, то его выполнение приведет к ошибке (рис. 3.5).

The screenshot shows a SQL query window with the following code:

```
SQLQuery1.sql - DE...ESKTOP\Yuriy (51)* | SELECT LastName, Grants FROM Students WHERE AVG(Grants) <= 1200;
```

Below the code, the 'Messages' tab is selected, displaying the following error message:

```
Msg 147, Level 15, State 1, Line 1
An aggregate may not appear in the WHERE clause unless it is in a subquery contained in a HAVING clause or a select list, and the column being aggregated is an outer reference.
```

Рисунок 3.5. Ошибка: невозможно использовать агрегатную функцию

4. Подзапросы

В языке SQL реализована возможность создания подзапросов или другими словами вложенных запросов, то есть возможность создания такого запроса, который будет помещен внутрь другого запроса. Существуют ситуации, когда без такого подхода получить требуемую информацию будет очень сложно или вообще невозможно.

Необходимость создания и использования подзапросов

Обсудим необходимость использования подзапросов на простом примере, допустим нам необходимо вывести фамилию, имя и номер группы студентов, которые получают максимальную стипендию. Казалось бы, что написать такой SQL-запрос не составит особого труда, например, так:

```
SELECT LastName, FirstName, GroupName, Grants  
FROM Students AS S, Groups AS G  
WHERE G.Id = S.GroupId  
GROUP BY LastName, FirstName, GroupName, Grants  
HAVING Grants = MAX(Grants);
```

Однако полученный результат будет далек от ожидаемого, в данном случае мы получили максимальное значение стипендии для каждой группы данных (рис. 4.1).

Такой результат выполнения SQL-запроса легко объясним. В данном случае данные сгруппированы таким образом, что формируют уникальные группы на основании фамилии и имени студента, названии группы и сти-

пенсии, а в операторе **HAVING** сравнивается значение стипендии для каждой группы данных с максимальным значением стипендии той же группы, то есть само с собой, тем самым возвращая истину. Группы, у которых максимальная стипендия имеет неопределенное значение (**Grants = NULL**) отбрасываются, так как сравнение на **NULL**-значение возможно только с использованием ключевого слова **IS NULL**.

LastName	FirstName	GroupName	Grants
Brown	Jessica	32PPS11	1100.00
Johnson	Joshua	31PPS11	1100.00
Jones	Jack	29PR21	1256.00
Miller	Harry	29PR21	1100.00
Taylor	Emily	32PR31	1100.00
Thomas	Charlie	32PR31	1256.00

Рисунок 4.1. Результат неправильно написанного SQL-запроса

Для того чтобы наш SQL-запрос вернул требуемые результаты, необходимо сравнивать стипендии студентов с максимальным значением стипендии (в нашем случае оно равно 1256). Однако, как вы знаете из предыдущего раздела, использование функций агрегирования в операторе **WHERE** запрещено, поэтому запрос можно написать следующим образом:

```
SELECT LastName, FirstName, GroupName, Grants
FROM Students AS S, Groups AS G
WHERE G.Id = S.GroupId AND Grants = 1256;
```

Результат запроса представлен на рисунке 4.2.

LastName	FirstName	GroupName	Grants
Jones	Jack	29PR21	1256.00
Thomas	Charlie	32PR31	1256.00

Рисунок 4.2. Студенты, которые получают максимальную стипендию

Несмотря на то, что в результате выполнения этого SQL-запроса мы получили правильный результирующий набор, такой подход не выдерживает никакой критики, ведь максимальное значение стипендии может меняться, что в свою очередь приведет к необходимости изменения «магического числа».

Поэтому для получения максимального значения стипендии целесообразно использовать соответствующий подзапрос (`SELECT MAX(Grants) FROM Students`). Полученное в результате его выполнения значение будет сравниваться со стипендией в каждой записи студентов и в случае совпадения значений, именно эта запись будет добавляться в результирующую таблицу. Перешишем предыдущий SQL-запрос с использованием подзапроса, результат его выполнения будет таким же, как на рисунке 4.2:

```
SELECT LastName, FirstName, GroupName, Grants
FROM Students AS S, Groups AS G
WHERE G.Id = S.GroupId
AND Grants = (SELECT MAX(Grants)
FROM Students);
```

Дополняя комментарии к данному запросу, следует добавить, что в соответствии с синтаксисом языка SQL подзапросы необходимо помещать в круглые скобки.

В предыдущем примере мы использовали подзапрос, который возвращал скалярное значение, как результат выполнения функции агрегирования **MAX()**. Однако SQL-запросы в основном возвращают множество строк и в этом случае при использовании их в качестве подзапросов существует своя особенность. Например, нам необходимо получить информацию обо всех студентах, которые учатся в группах с номером 11 при этом нам неважно на каком потоке. В этом случае подзапрос выполнится правильно и вернет неопределенное количество значений одного столбца (**Id**), но неправильное сравнение возвращаемых им результатов приведет к ошибке на рисунке 4.3.

```
SQLQuery1.sql - DE...ESKTOP\Yuriy (51)* ↵ X
SELECT LastName, FirstName, Grants
FROM Students
WHERE GroupId = (SELECT Id FROM Groups WHERE GroupName LIKE '%11');

132 % ↵
Results Messages
Msg 512, Level 16, State 1, Line 1
Subquery returned more than 1 value. This is not permitted when the subquery
follows =, !=, <, <=, >, >= or when the subquery is used as an expression.
```

Рисунок 4.3. Ошибка: невозможно применять логические операторы в том случае если запрос возвращает больше одного значения

Данная ошибка связана с тем, что наш подзапрос возвращает уникальные идентификаторы нескольких групп, которые соответствуют заданному условию (**GroupName LIKE '%11'**), а в операторе **WHERE** возвращаемые результаты сравниваются с единственным значением столбца **GroupId**, что в принципе невозможно (подобная ситуация уже описывалась в текущем уроке при использовании функций агрегирования). Поэтому во всех случаях, когда

запрос может вернуть несколько значений, для их сравнения вместо логических операторов следует использовать ключевое слово **IN**:

```
SELECT LastName, FirstName, GroupId
FROM Students
WHERE GroupId IN (SELECT Id
FROM Groups WHERE GroupName LIKE '%11');
```

Результат выполнения текущего запроса представлен на рисунке 4.4.

LastName	FirstName	GroupId
Evans	Grace	2
Wilson	Lily	2
Johnson	Joshua	3
Brown	Jessica	5

Рисунок 4.4. Студенты, которые учатся в группах с номером 11

Сравнение подзапросов и многотабличных запросов

Как многотабличные запросы, так и подзапросы можно отнести к категории «сложных» запросов. Однако если многотабличные запросы возвращают данные, полученные в результате соединения нескольких реально существующих таблиц, то подзапросы формируют виртуальные значения, которые используются для сравнения с реально существующими данными. Подзапросы позволяют обеспечить значительную гибкость при выполнении запросов, потому что их можно использовать не только в операторе **WHERE** (что было уже продемонстрировано), но и в операторах **SELECT**, **FROM** и **HAVING**.

В качестве примера мы напишем подзапрос, который вернет данные (в виде виртуальной таблицы), на основании которых мы выведем информацию основного запроса. Такое использование подзапросов в операторе **FROM** может быть возможным, только если для результатов выполнения подзапроса будет указан псевдоним с использованием ключевого слова **AS**.

Для того чтобы вам было легче понять следующий SQL-запрос мы приведем часть диаграммы базы данных University из прошлого урока (рис. 4.5).

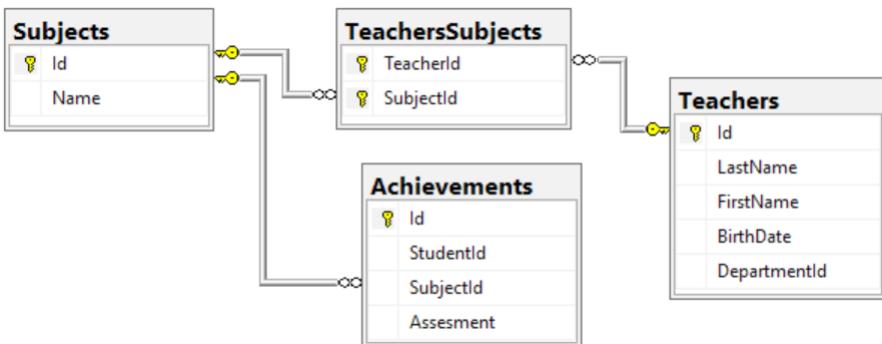


Рисунок 4.5. Диаграмма базы данных University (частично)

Предположим нам необходимо вывести фамилии преподавателей, читающих предметы, по которым студенты получают отличные оценки, для этого выполним следующий запрос:

```

SELECT T.LastName, M.SubjectName
FROM Teachers AS T, TeachersSubjects AS TS,
(SELECT S.Id AS SubId, S.Name AS SubjectName
FROM Subjects AS S, Achievements AS A
WHERE S.Id = A.SubjectId
  
```

```

GROUP BY S.Name, S.Id
HAVING MAX(A.Assessment) >= 10) AS M
WHERE T.Id = TS.TeacherId
AND TS.SubjectId = M.SubId;

```

В данном случае наш подзапрос формирует виртуальную таблицу, которая возвращает уникальный идентификатор и названия предметов, по которым студенты получают только отличные оценки (`MAX(A.Assessment) >= 10`). Для этой таблицы указан псевдоним (`AS M`), благодаря чему основной запрос может получать доступ к значениям ее столбцов: осуществлять соединение (`TS.SubjectId = M.SubId`) и выводить данные (`M.SubjectName`). Результат выполнения данного запроса представлен на рисунке 4.6.

Last Name	Subject Name
Nelson	C#
MacAlister	SQL Server
Cooper	ADO.NET

Рисунок 4.6. Преподаватели и предметы, по которым студенты получают отличные оценки

В следующем SQL-запросе мы продемонстрируем возможность использования подзапросов в операторе `HAVING`. При помощи этого запроса мы получим список преподавателей, среднее значение месяца рождения которых больше среднего значения месяца рождения студентов:

```

SELECT LastName, FirstName
FROM Teachers
GROUP BY LastName, FirstName
HAVING AVG(MONTH(BirthDate))>
(SELECT AVG(MONTH(BirthDate)) FROM Students);

```

В этом случае подзапрос возвращает среднее значение месяцев рождения студентов, которое в операторе **HAVING** сравнивается со средним значением месяца рождения каждого преподавателя. Результат выполнения данного запроса представлен на рисунке 4.7.

Last Name	First Name
Williams	Daniel
Cooper	Michael
Nelson	Sophia

Рисунок 4.7. Использование подзапроса
в операторе HAVING

Принцип работы подзапросов

В том случае если в SQL-запросе применяются подзапросы, то выполнение SQL-операторов в первую очередь происходит в подзапросе, а уже потом полученные результаты используются в основном запросе. В одном SQL-запросе может быть несколько подзапросов при этом они могут быть вложены друг в друга, в этом случае выполнение начинается с подзапроса, который имеет наиболее глубокое вложение.

Продемонстрируем это при помощи следующего SQL-запроса, который выведет название групп, студенты которых получают максимальную стипендию:

```
SELECT GroupName
FROM Groups
WHERE Id IN (SELECTGroupId
FROM Students
WHERE Grants = (SELECT MAX(Grants) FROM Students));
```

В данном случае первым выполнится подзапрос (`SELECT MAX(Grants) FROM Students`), который вернет размер максимальной стипендии. Потом его результаты будут использованы в подзапросе (`SELECT GroupId FROM Students WHERE Grants = (...)`) для определения уникальных идентификаторов групп, полученные результаты в свою очередь будут использоваться при выполнении основного SQL-запроса (рис. 4.8).

GroupName
29PR21
32PR31

Рисунок 4.8. Группы, в которых студенты получают максимальную стипендию

Следует заметить, что существует еще один вид подзапросов, которые называются связанными или коррелированными подзапросами. Особенностью выполнения таких подзапросов является их зависимость от значений в основном запросе и поэтому они не могут быть обработаны раньше, чем основной запрос, для лучшего понимания принципа их использования приведем пример. Допустим нам необходимо вывести максимальную оценку, полученную студентами по каждому предмету. Такую информацию можно получить, написав SQL-запрос и по-другому, но мы в данном случае решили продемонстрировать использование связанного подзапроса в операторе `SELECT`. Необходимо заметить, что обязательным условием такого использования является возврат подзапросом одного значения, для этих целей идеально подходят функции агрегирования:

```
SELECT Subjects.Name,(SELECT MAX(A.Assessment)
FROM Achievements AS A
WHERE Subjects.Id = A.SubjectId) AS Maximum
FROM Subjects;
```

Выполнение текущего связанного подзапроса зависит от сравнения уникального идентификатора предмета (`Subjects.Id = A.SubjectId`), значение которого изменяется по мере построчного прочтения записей в таблице `Subjects`, поэтому количество строк в полученной виртуальной таблице соответствует количеству записей в таблице `Subjects` (рис. 4.9).

Name	Maximum
C#	10
Discrete Math	NULL
SQL Server	10
ADO.NET	11
ITE1	NULL
JavaScript	NULL
WIN10	NULL

Рисунок 4.9. Максимальные оценки студентов по каждому предмету

5. Домашнее задание

1. Вам необходимо создать многотабличную базу данных, содержащую информацию о вымышленном фитнес клубе, которая должна содержать следующие таблицы: инструкторы, секции, посетители и т.д.
2. Используя полученные в этом уроке знания, написать к созданной вами базе данных следующие SQL-запросы:
 - вывести количество инструкторов по каждой секции;
 - показать количество людей, которые должны заниматься в определенный момент времени в каждой секции;
 - вывести количество посетителей фитнес клуба, которые пользуются услугами определенного мобильного оператора;
 - получить количество посетителей, у которых фамилия совпадает с фамилиями из определенного списка;
 - показать количество людей с одинаковыми именами, которых занимаются у определенного инструктора;
 - получить информацию о людях, которые посетили фитнес зал минимальное количество раз;
 - вывести количество посетителей, которые занимались в определенной секции за первую половину текущего года;
 - определить общее количество людей, посетивших фитнес зал за прошлый год.

Unit 6.

Объединения

1. Операторы для использования в подзапросах

В прошлом уроке вами были изучены подзапросы и возможность их использования при написании SQL-запросов с применением оператора **IN**, однако совместно с подзапросами можно использовать еще несколько логических операторов, именно их мы и рассмотрим в текущем разделе.

Для того чтобы продемонстрировать вам примеры выполнения SQL-запросов мы будем использовать, знакомую вам по прошлым занятиям, базу данных **University**, частичная диаграмма которой представлена на рисунке 1.1.

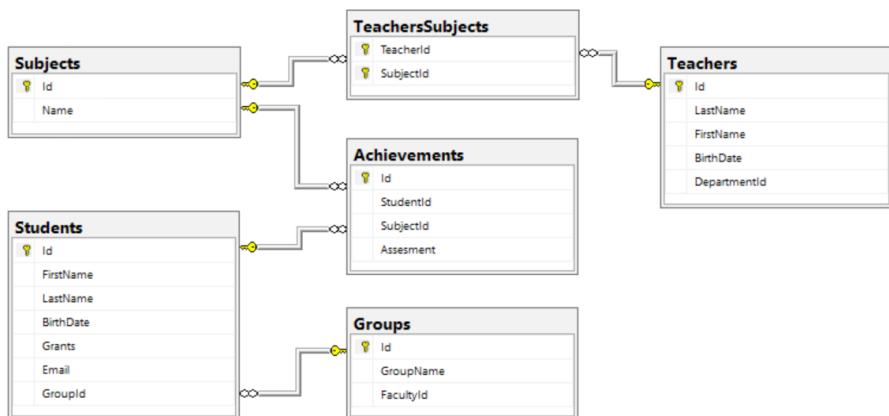


Рисунок 1.1. Диаграмма базы данных University
(частично)

Оператор EXISTS

Оператор **EXISTS** является в своем роде уникальным, потому что проверяет строки, а не сравнивает значения столбцов и предназначен для определения наличия строк, возвращаемых подзапросом. Если подзапрос содержит хотя бы одну строку, то оператор **EXISTS** возвращает истину и текущая запись помещается в результирующую таблицу, иначе оператор возвращает ложь и, соответственно, данные не записываются.

В качестве примера продемонстрируем SQL-запрос, возвращающий нам информацию о студентах, которые получали оценки по предметам:

```
SELECT FirstName, LastName, BirthDate, Email  
FROM Students  
WHERE EXISTS (SELECT *  
              FROM Achievements  
              WHERE Achievements.StudentId = Students.Id);
```

Следует обратить ваше внимание на ряд особенностей текущего запроса. Во-первых, вы, наверное, заметили, что в данном SQL-запросе подзапрос является связанным с основным запросом через уникальный идентификатор студента (**Achievements.StudentId = Students.Id**). Во-вторых, в подзапросе после оператора **SELECT** используется символ *****, такая запись характерна для любого подзапроса, с использованием оператора **EXISTS**, потому что он применяется к строкам и возвращаемые столбцы не имеют никакого значения.

При выполнении текущего SQL-запроса оператор **EXISTS** вернет истину, если в таблице **Achievements** су-

ществует хотя бы одна строка, содержащая уникальный идентификатор студента из основного запроса и именно информация по этому студенту будет записана в результирующую таблицу (рис. 1.2).

FirstName	LastName	BirthDate	Email
Jack	Jones	1997-11-05	jj@net.eu
Harry	Miller	1998-02-11	hm@net.eu
Grace	Evans	1997-06-24	eg@net.eu

Рисунок 1.2. Студенты, получившие оценки по предметам

Оператор **EXISTS** можно использовать совместно с оператором **NOT** для определения данных, которые не удовлетворяют условию. Например, при написании дуального SQL-запроса, который отобразит информацию о студентах, не получивших ни одной оценки по предметам:

```
SELECT FirstName, LastName, BirthDate, Email  
FROM Students  
WHERE NOT EXISTS (SELECT *  
                   FROM Achievements  
                   WHERE Achievements.StudentId = Students.Id);
```

В приведенном примере подзапрос также связан с основным запросом по идентификатору студента, однако в данном случае использование оператора **NOT** приведет к тому, что в результирующую таблицу будет записана информация только о тех студентах, для которых оператор **EXISTS** вернет ложь. То есть в том случае если в таблице **Achievements** не существует ни одной стро-

ки, содержащей уникальный идентификатор студента из основного запроса. Результат выполнения текущего запроса представлен на рисунке 1.3.

FirstName	LastName	BirthDate	Email
Lily	Wilson	1998-09-12	lw@net.eu
Joshua	Johnson	1997-05-23	jo@net.eu
Emily	Taylor	1997-12-27	et@net.eu
Charlie	Thomas	1998-01-31	ct@net.eu
Oliver	Moore	1997-07-05	om@net.eu
Jessica	Brown	1997-07-17	jb@net.eu

Рисунок 1.3. Студенты, которые не получили оценки по предметам

Оператор ANY/SOME

Операторы **ANY** и **SOME** являются синонимами и выполняют, как это ни странно, одно и то же действие – осуществляют проверку выполнения заданного условия сравнения хотя бы для одного значения из тех, которые возвращает подзапрос.

Следующий SQL-запрос позволит нам выяснить существуют ли в нашей базе данных **University** записи о студентах, которые получали оценку 10:

```
SELECT FirstName, LastName, BirthDate, Email
FROM Students
WHERE Id = ANY (SELECT StudentId
                 FROM Achievements
                 WHERE Assesment = 10);
```

Следует обратить ваше внимание на особенность использования оператора **ANY** — оператор сравнения

1. Операторы для использования в подзапросах

необходимо указывать между сравниваемым значением и оператором (**WHERE Id = ANY ...**).

В текущем запросе подзапрос возвращает уникальные идентификаторы тех студентов, которые получили оценку 10 и если идентификатор студента в записи из основного запроса совпадет с любым из них, то эта запись будет добавлена в результирующую таблицу. Результат выполнения данного запроса представлен на рисунке 1.4.

FirstName	LastName	BirthDate	Email
Jack	Jones	1997-11-05	jj@net.eu
Hamy	Miller	1998-02-11	hm@net.eu

Рисунок 1.4. Студенты, которые получали оценку 10

В том случае если вы замените в данном запросе оператор **ANY** на **SOME**, то вы получите такой же результат, мы предлагаем вам проверить это самостоятельно.

Внимательный студент может возразить, что для получения такого результата будет достаточно сравнить результаты выполнения подзапроса при помощи оператора **IN** и будет абсолютно прав. Следующий SQL-запрос вернет такой же результат, как и на рисунке 1.4:

```
SELECT FirstName, LastName, BirthDate, Email  
FROM Students  
WHERE Id IN (SELECT StudentId  
              FROM Achievements  
              WHERE Assesment = 10);
```

Однако преимущество операторов **ANY/SOME** заключается в том, что их можно использовать не только

с оператором сравнения на равенство (`=`), как оператор `IN`, но и с операторами сравнения на неравенство (`>`, `<`, `>=`, `<=`, `<>`).

В качестве примера приведем SQL-запрос, который позволит нам определить количество студентов, которые старше любого из преподавателей:

```
SELECT COUNT(*) AS [Count]
FROM Students
WHERE BirthDate < ANY (SELECT BirthDate
                           FROM Teachers);
```

В этом запросе дата рождения каждого студента сравнивается с каждой датой рождения преподавателей из списка дат, который возвращает нам подзапрос. Если дата рождения студента меньше чем любая из дат рождения преподавателей, то такой студент старше какого-то преподавателя и функция агрегирования `COUNT(*)` позволяет определить их количество. Результат выполнения текущего запроса представлен на рисунке 1.5.

Count
0

Рисунок 1.5. Количество студентов, которые старше любого из преподавателей

Оператор `ALL`

Оператор `ALL` используется при сравнении результатов подзапроса, таким образом, чтобы указанному условию удовлетворяли все результаты подзапроса без исключения.

Данный оператор так же, как и операторы ANY/SOME может применяться совместно с различными операторами сравнения. Например, выполнив следующий SQL-запрос, мы сможем получить список студентов, оценки которых больше, чем средние оценки каждого студента, результат этого запроса представлен на рисунке 1.6.

```
SELECT FirstName, LastName, Assesment  
FROM Students AS S, Achievements AS A  
WHERE StudentId = S.Id AND  
Assesment > ALL (SELECT AVG(Assesment)  
                  FROM Achievements  
                  GROUP BY StudentId);
```

FirstName	Last Name	Assesment
Jack	Jones	11

Рисунок 1.6. Студенты, оценки которых больше, чем средние оценки каждого студента

В данном случае подзапрос возвращает список средних оценок всех студентов, а в основном запросе осуществляется поиск студента, оценка которого больше, чем все оценки из этого списка.

Операторы ANY/SOME и ALL могут применяться не только совместно с оператором WHERE, что было показано во всех предыдущих примерах, но и с оператором HAVING. Для того чтобы продемонстрировать вам это мы, исключительно в учебных целях, напишем следующий SQL-запрос, который возвращает список студентов оценки, которых не равны любой из оценок студентов группы 30PR11:

```

SELECT FirstName, LastName, Assesment
FROM Students AS S,
(SELECT StudentId, Assesment
FROM Achievements
GROUP BY StudentId, Assesment
HAVING Assesment <> ALL (SELECT Assesment
FROM Groups AS G, Students AS S,
Achievements AS A
WHERE S.GroupId = G.Id
AND A.StudentId = S.Id
AND GroupName = '30PR11')) AS SA
WHERE SA.StudentId = S.Id;

```

В текущем SQL-запросе применяются два подзапроса. Первый подзапрос, с наибольшей вложенностью, возвращает список оценок, полученных студентами группы 30PR11. Результаты этого подзапроса используются вторым подзапросом для получения уникального идентификатора студента и его оценки при условии, что эта оценка не равна ни одной из оценок, возвращенных первым подзапросом. Результаты второго подзапроса формируются в виде таблицы с псевдонимом **SA**, которая соединяется с таблицей основного запроса по идентификатору студента (**SA.StudentId = S.Id**) (рис. 1.7).

FirstName	LastName	Assesment
Harry	Miller	9
Jack	Jones	10
Harry	Miller	10
Jack	Jones	11

Рисунок 1.7. Студенты оценки, которых не равны оценкам студентов группы 30PR11

2. Объединение результатов запроса

Принципы объединения

Объединение результатов запросов применяется в том случае если необходимо получить совокупность строк двух и более запросов, которые выполняются независимо друг от друга.

Для того чтобы результаты запросов можно было объединить они должны соответствовать определенным условиям совместимости:

- количество столбцов в каждом запросе должно быть одинаковым;
- типы данных соответствующих столбцов во всех запросах должны быть совместимы.

Также при объединении запросов существует ряд особенностей:

- в результирующем наборе, полученном при объединении, будут использоваться имена столбцов, которые были указаны в первом запросе;
- нельзя сортировать каждый запрос по отдельности, осуществить сортировку можно только всего составного запроса, указав по его окончанию оператор **ORDER BY**.

Объединение результатов запросов обеспечивается благодаря использованию ключевых слов **UNION** и **UNION ALL**. Общая форма записи SQL-запросов

с использованием этих операторов выглядит следующим образом:

```
SELECT columnName1, columnName2, ...
FROM tableName
UNION [ALL]
SELECT columnName1, columnName2, ...
FROM tableName;
```

Ключевое слово UNION

Ключевое слово **UNION** позволяет объединить результаты запросов и применяется в том случае, когда в результирующем наборе необходимо исключить повторяющиеся строки.

Продемонстрируем использование ключевого слова **UNION** на примере объединения двух запросов, в результате которого нам необходимо получить студентов и преподавателей, родившихся летом, при этом результирующий набор будет отсортирован по их дате рождения:

```
SELECT FirstName + ' ' + LastName
      AS FullName, BirthDate
   FROM Students
 WHERE MONTH(BirthDate) > 5
   AND MONTH(BirthDate) < 9
UNION
SELECT FirstName + ' ' + LastName, BirthDate
   FROM Teachers
 WHERE MONTH(BirthDate) > 5
   AND MONTH(BirthDate) < 9
 ORDER BY BirthDate;
```

Данный SQL-запрос будет выполняться следующим образом: вначале выполняются два запроса, которые вернут полное имя и дату рождения студентов и преподавателей, родившихся летом, соответственно. После этого результаты запросов объединяются вместе без повторяющихся записей при помощи ключевого слова **UNION**, полученный вследствие этого результатирующий набор будет отсортирован по возрастанию даты рождения с использованием оператора **ORDER BY** (рис. 2.1).

FullName	BirthDate
Daniel Williams	1979-07-30
Grace Evans	1997-06-24
Oliver Moore	1997-07-05
Jessica Brown	1997-07-17

Рисунок 2.1. Преподаватели и студенты, родившиеся летом

Ключевое слово UNION ALL

Ключевое слово **UNION ALL** также позволяет объединять результаты различных запросов, однако выполняется быстрее, чем **UNION**, потому что не тратит время на удаление дублирующих строк в объединяемых запросах.

Объединение результатов запросов можно также применять, в том случае если необходимо получить сводную информацию из различных SQL-запросов, которые связаны между собой определенным критерием. В следующем примере мы продемонстрируем возможность получения информации о количестве студентов, которые родились в разные времена года:

```

SELECT 'Spring' AS Seasons, COUNT(*)
      AS [Number of students]
FROM Students
WHERE MONTH(BirthDate) BETWEEN 3 AND 5
UNION ALL
SELECT 'Summer', COUNT(*)
FROM Students
WHERE MONTH(BirthDate) BETWEEN 6 AND 8
UNION ALL
SELECT 'Autumn', COUNT(*)
FROM Students
WHERE MONTH(BirthDate) BETWEEN 9 AND 11
UNION ALL
SELECT 'Winter', COUNT(*)
FROM Students
WHERE MONTH(BirthDate) IN (1, 2, 12);

```

В данном случае каждый из SQL-запросов возвращает количество студентов, родившихся весной, летом, осенью и зимой соответственно. Первым значением каждого запроса указано название времени года в виде строкового литерала, такой подход срабатывает, потому что используется, по сути, одинаковый тип данных.

Seasons	Number of students
Spring	1
Summer	3
Autumn	2
Winter	3

Рисунок 2.2. Количество студентов
в соответствии со временами года

Полученные результаты объединяются при помощи ключевого слова UNION ALL, которое еще обеспечивает

требуемый порядок объединения, что не позволяет сделать оператор **UNION** (рис. 2.2).

Еще один пример продемонстрирует возможность использования объединения при составлении статистических отчетов. Например, нам необходимо получить отдельно количество студентов и преподавателей, которые родились во втором квартале, а также их общее количество:

```
SELECT 'Students' AS [Second quarter], COUNT(*)  
AS [Count]  
FROM Students  
WHERE MONTH(BirthDate) BETWEEN 5 AND 8  
UNION ALL  
SELECT 'Teachers', COUNT(*)  
FROM Teachers  
WHERE MONTH(BirthDate) BETWEEN 5 AND 8  
UNION ALL  
SELECT 'All', SUM(AllSum.AllCount)  
FROM  
(  
    SELECT COUNT(*) AS AllCount  
    FROM Students  
    WHERE MONTH(BirthDate) BETWEEN 5 AND 8  
    UNION ALL  
    SELECT COUNT(*)  
    FROM Teachers  
    WHERE MONTH(BirthDate) BETWEEN 5 AND 8  
) AS AllSum
```

Данный SQL-запрос аналогичен предыдущему, только в этом случае два первых объединяемых запроса возвращают количество студентов и преподавателей, родившихся во втором квартале соответственно, а в третьем

запросе для подсчета общего количества записей в таблице с прототипом AllSum, которую мы получили как результат объединения двух запросов, используется функция **SUM(AllSum.AllCount)**. Результат выполнения этого SQL-запроса приведен на рисунке 2.3.

Second quarter	Count
Students	4
Teachers	2
All	6

Рисунок 2.3. Количество студентов и преподавателей, родившихся во втором квартале

3. Объединения JOIN

В прошлом разделе мы изучили возможность объединения результатов запросов при помощи ключевого слова **UNION**, такой вид объединения можно назвать объединением по вертикали. В свою очередь оператор **JOIN** позволяет объединять таблицы в пределах одного SQL-запроса и может считаться объединением по горизонтали. Существуют различные виды объединений, которые можно выполнить при помощи оператора **JOIN**: внутреннее (**INNER**) и внешние (**LEFT**, **RIGHT**, **FULL**).

В предыдущих уроках при написании SQL-запросов для объединения таблиц мы использовали синтаксис с применением оператора **WHERE**, который соответствует стандарту ANSI SQL-89. Синтаксис с использованием оператора **JOIN** соответствует стандарту ANSI SQL-92, который помимо других улучшений языка SQL обеспечивает, в отличие от SQL-89, поддержку выполнения внешних запросов и улучшает читабельность запроса, явно разделяя объединение таблиц и фактическую фильтрацию данных.

В данный момент времени вам может показаться, что предыдущие ваши усилия, потраченные на написание SQL-запросов по стандарту SQL-89, были напрасны, но нет. Дело в том, что оба синтаксиса и SQL-89, и SQL-92 повсеместно применяются различными программистами и для того, чтобы понимать запросы, написанные другими специалистами, вам необходимо научиться использовать оба этих синтаксиса.

В дальнейшем мы рекомендуем вам использовать синтаксис с использованием оператора JOIN, который представляет собой более современный подход при написании SQL-запросов и предоставляет большие возможности, описанные выше.

Понятие INNER JOIN

Внутреннее объединение двух таблиц возможно при помощи оператора INNER JOIN с указанием условия их объединения (предиката) после ключевого слова **ON**. Общая форма записи SQL-запросов такого вида будет выглядеть следующим образом:

```
SELECT columnName1, columnName2, ...
FROM tableName1 [INNER] JOIN tableName2
ON tableName1.columnName =
    tableName2.columnName;
```

При выполнении оператора INNER JOIN каждая запись из первой таблицы сопоставляется с каждой записью из другой таблицы по условию, указанному после оператора **ON**, если условие выполняется, тогда строки записываются в результирующую таблицу, которая формируется путем конкатенации строк первой и второй таблиц.

В качестве первого примера внутреннего объединения приведем запрос, при помощи которого мы получим всю информацию о группах и студентах в этих группах:

```
SELECT *
FROM Groups INNER JOIN Students
ON Groups.Id = Students.GroupId;
```

В данном SQL-запросе после оператора `SELECT` специально прописан символ `*`, для того чтобы мы смогли увидеть все полученные столбцы (рис. 3.1).

Groups			Students						
Id	GroupName	FacultyId	Id	FirstName	LastName	BirthDate	Grants	Email	GroupId
1	29PR21	1	1	Jack	Jones	1997-11-05	1256.00	jj@net.eu	1
1	29PR21	1	2	Harry	Miller	1998-02-11	1100.00	hm@net.eu	1
2	30PR11	2	3	Grace	Evans	1997-06-24	NULL	eg@net.eu	2
2	30PR11	2	4	Lily	Wilson	1998-09-12	NULL	lw@net.eu	2
3	31PPS11	1	5	Joshua	Johnson	1997-05-23	1100.00	jo@net.eu	3
4	32PR31	2	6	Emily	Taylor	1997-12-27	1100.00	et@net.eu	4
4	32PR31	2	7	Charlie	Thomas	1998-01-31	1256.00	ct@net.eu	4
4	32PR31	2	8	Oliver	Moore	1997-07-05	NULL	om@net.eu	4
5	32PPS11	3	9	Jessica	Brown	1997-07-17	1100.00	jb@net.eu	5

Рисунок 3.1. Результат использования внутреннего объединения в SQL-запросе

Как вы можете заметить, мы получили результирующую таблицу, которая фактически состоит из значений двух таблиц `Groups` и `Students`, связанных между собой уникальным идентификатором группы (`Groups.Id`).

Естественно, что использовать результаты запроса в таком виде не имеет никакого смысла, однако мы можем указать в операторе `SELECT` только необходимые нам столбцы, и в результирующей таблице будут именно их значения в указанной нами последовательности:

```
SELECT LastName, FirstName, Email, GroupName
FROM Groups INNER JOIN Students
ON Groups.Id = Students.GroupId;
```

Результат выполнения текущего SQL-запроса представлен на рисунке 3.2.

LastName	FirstName	Email	GroupName
Jones	Jack	jj@net.eu	29PR21
Miller	Harry	hm@net.eu	29PR21
Evans	Grace	eg@net.eu	30PR11
Wilson	Lily	lw@net.eu	30PR11
Johnson	Joshua	jo@net.eu	31PPS11
Taylor	Emily	et@net.eu	32PR31
Thomas	Charlie	ct@net.eu	32PR31
Moore	Oliver	om@net.eu	32PR31
Brown	Jessica	jb@net.eu	32PPS11

Рисунок 3.2. Информация о студентах

При использовании оператора INNER JOIN допускается не писать служебное слово INNER, в этом случае JOIN будет расцениваться как внутреннее объединение.

Как вы, наверное, догадались при помощи оператора INNER JOIN можно осуществлять объединение нескольких таблиц, продемонстрируем это на следующем примере. Допустим нам необходимо получить информацию об успеваемости всех студентов:

```
SELECT FirstName, LastName, Name AS Subject,
       Assesment, GroupName
  FROM Groups AS G JOIN Students AS S
    ON G.Id = S.GroupId
   JOIN Achievements AS A ON S.Id = A.StudentId
   JOIN Subjects AS Sb ON Sb.Id = A.SubjectId;
```

При выполнении данного SQL-запроса нам потребовалось объединить таблицы **Groups** и **Students** по идентификатору группы (**G.Id = S.GroupId**), **Achievements** и **Students** связав их по идентификатору студента (**S.Id = A.StudentId**) и таблицы **Subjects** и **Achievements** – по идентификатору предмета (**Sb.Id = A.SubjectId**) (рис. 3.3).

3. Объединения JOIN

FirstName	LastName	Subject	Assesment	GroupName
Jack	Jones	ADO.NET	11	29PR21
Harry	Miller	C#	10	29PR21
Jack	Jones	SQL Server	10	29PR21
Grace	Evans	C#	8	30PR11
Harry	Miller	ADO.NET	9	29PR21
Grace	Evans	SQL Server	7	30PR11

Рисунок 3.3. Информация об успеваемости студентов

При осуществлении внутреннего объединения вы, как и раньше, можете использовать все операторы языка SQL. Например, если потребуется получить оценки по предметам только у студентов 29 потока и отсортировать полученную информацию по фамилиям студентов, тогда нам достаточно применить операторы **WHERE** и **ORDER BY** к написанному ранее запросу (рис. 3.4):

```
SELECT FirstName, LastName, Name AS Subject,  
       Assesment, GroupName  
  FROM Groups AS G JOIN Students AS S  
    ON G.Id = S.GroupId  
 JOIN Achievements AS A ON S.Id = A.StudentId  
 JOIN Subjects AS Sb ON Sb.Id = A.SubjectId  
 WHERE GroupName LIKE '29%'  
 ORDER BY LastName;
```

FirstName	LastName	Subject	Assesment	GroupName
Jack	Jones	ADO.NET	11	29PR21
Jack	Jones	SQL Server	10	29PR21
Harry	Miller	C#	10	29PR21
Harry	Miller	ADO.NET	9	29PR21

Рисунок 3.4. Успеваемость студентов 29 потока

Необходимость использования внешнего объединения

При выполнении SQL-запросов, которые связаны с получением полной информации из таблиц, использование внутреннего объединения не даст желаемого результата. Например, необходимо вывести информацию обо всех студентах и оценках, которые ими были получены. Запрос к базе данных *University* с применением оператора **INNER JOIN** будет выглядеть следующим образом:

```
SELECT FirstName + ' ' + LastName
      AS FullName, Assesment
   FROM Students AS S INNER JOIN Achievements AS A
     ON S.Id = A.StudentId;
```

В качестве результата выполнения данного запроса мы получим следующий набор данных (рис. 3.5).

FullName	Assesment
Jack Jones	11
Harry Miller	10
Jack Jones	10
Grace Evans	8
Harry Miller	9
Grace Evans	7

Рисунок 3.5. Результат выполнения внутреннего объединения

Однако полученный нами результат не удовлетворяет требуемому условию, ведь нам необходимо было получить данные обо всех студентах, а не только о тех из них кто получил оценки. Такое поведение характерно при использовании оператора **INNER JOIN**, потому что

в результирующую таблицу будут помещены только те записи, которые удовлетворяют условию, указанному после ключевого слова **ON**, в нашем случае соответствие уникального идентификатора студента (**S.Id = A.StudentId**).

Для того, чтобы получать правильные результаты при написании подобных SQL-запросов необходимо применять внешние объединения (**OUTER JOIN**), которые позволяют получить все данные из одной таблицы независимо от того существует ли для них совпадения строк в другой таблице. Существует три типа внешних объединений: левое (**LEFT**), правое (**RIGHT**) и полное (**FULL**). Общая форма записи внешних объединений выглядит следующим образом:

```
SELECT columnName1, columnName2, ...
FROM leftTable LEFT | RIGHT | FULL [OUTER]
JOIN rightTable
ON tableName1.columnName = tableName2.columnName;
```

При написании запросов с использованием внешних объединений ключевое слово **OUTER** можно опустить, потому что ключевые слова **LEFT**, **RIGHT** и **FULL** однозначным образом определят его тип.

Понятие LEFT JOIN

Оператор **LEFT JOIN** (**LEFT OUTER JOIN**) позволяет создать, так называемое левое внешнее объединение, при выполнении которого в результирующий набор помимо строк, удовлетворяющих условию после ключевого слова **ON**, будут записаны даже те строки из таблицы, расположенной слева от оператора, которые не удовлетворяют указанному условию.

Попробуем решить задачу обо всех студентах и их оценках поставленную в предыдущем разделе при помощи оператора **LEFT JOIN** и напишем для этого соответствующий SQL-запрос:

```
SELECT FirstName + ' ' + LastName
    AS FullName, Assesment
FROM Students AS S LEFT JOIN Achievements AS A
ON S.Id = A.StudentId;
```

В данном случае в результирующий набор попадут данные обо всех студентах из таблицы **Students**, независимо от того получали они оценки или нет. В случае если в таблице **Achievements** отсутствует оценка для соответствующего студента, результат будет равен **NULL** (Рис. 3.6).

FullName	Assesment
Jack Jones	11
Jack Jones	10
Harry Miller	10
Harry Miller	9
Grace Evans	8
Grace Evans	7
Lily Wilson	NULL
Joshua Johnson	NULL
Emily Taylor	NULL
Charlie Thomas	NULL
Oliver Moore	NULL
Jessica Brown	NULL

Рисунок 3.6. Результат выполнения левого внешнего объединения

Для того чтобы окончательно убедить вас в полезности внешних объединений, продемонстрируем вам два варианта решения еще одной задачи. Допустим, нам

необходимо получить информацию о студентах, которые пока не получали оценок.

Одним из вариантов получения требуемых данных является SQL-запрос с обязательным использованием подзапроса:

```
SELECT FirstName + ' ' + LastName AS FullName
FROM Students
WHERE Id NOT IN (SELECT StudentId
                  FROM Achievements);
```

В данном случае подзапрос возвращает уникальные идентификаторы всех студентов, которые получили оценки, и информация о которых находится в таблице **Achievements**. В основном запросе выводятся данные только о тех студентах, уникальный идентификатор которых отсутствует во множестве, сформированном подзапросом, то есть запись о них отсутствует в таблице **Achievements**, а значит они еще не получили ни одной оценки (Рис. 3.7).

FullName
Lily Wilson
Joshua Johnson
Emily Taylor
Charlie Thomas
Oliver Moore
Jessica Brown

Рисунок 3.7. Студенты, которые не получили оценки

При написании подзапросов многие студенты сталкиваются с проблемой понимания их выполнения. Поэтому более изящным решением поставленной задачи

без использования подзапроса является возможность применения в данном SQL-запросе оператора **LEFT JOIN**, который образует более понятную синтаксическую конструкцию. К тому же такой запрос выполнится быстрее, потому что для каждого **LEFT JOIN** создается отдельной поток и его выполнение происходит параллельно в отличие от подзапроса, выполнение которого осуществляется последовательно с основным запросом в одном потоке.

Для того чтобы понять, как решить поставленную задачу при помощи внешнего объединения, вам необходимо внимательно присмотреться к результату выполнения SQL-запроса на рисунке 3.6. Как вы заметили, у некоторых студентов оценка имеет неопределенное значение (**NULL**), то есть если отфильтровать записи по этому признаку, мы получим требуемый результат, который полностью совпадает с результатом, который был получен при выполнении предыдущего запроса (рис. 3.7):

```
SELECT FirstName + ' ' + LastName AS FullName
FROM Students AS S LEFT JOIN Achievements AS A
ON S.Id = A.StudentId
WHERE Assesment IS NULL;
```

Понятие **RIGHT JOIN**

При помощи оператора **RIGHT JOIN (RIGHT OUTER JOIN)** возможно создать, так называемое правое внешнее объединение, при выполнении которого в результате получающий набор помимо строк, удовлетворяющих условию после ключевого слова **ON**, будут записаны все строки из таблицы, расположенной справа от оператора, независимо от того удовлетворяют они указанному условию или нет.

В качестве первого примера использования оператора **RIGHT JOIN** продемонстрируем вам решение уже известной вам задачи — вывод информации обо всех студентах и полученных ими оценках:

```
SELECT FirstName + ' ' + LastName
      AS FullName, Assesment
   FROM Achievements AS A RIGHT JOIN Students AS S
     ON S.Id = A.StudentId;
```

В данном случае мы поменяли местами таблицы **Achievements** и **Students** и в результате выполнения данного запроса мы получили результат идентичный результату на рисунке 3.6. Таким образом, при помощи правого внешнего объединения можно получить результаты аналогичные результатам левого внешнего объединения, если изменить порядок следования таблиц в SQL-запросе.

В следующем примере мы продемонстрируем возможность совместного использования правого и левого объединений. Предположим, что нам необходимо получить список всех предметов и информацию о преподавателях, которые их читают, ниже представлен возможный вариант запроса:

```
SELECT [Name] AS [Subject], LastName, FirstName
   FROM TeachersSubjects AS TS RIGHT JOIN Subjects
     AS S
    ON S.Id = TS.SubjectId LEFT JOIN Teachers AS T
    ON T.Id = TS.TeacherId
 ORDER BY [Name];
```

Таблица **TeachersSubjects** содержит информацию о том, кто из преподавателей какой предмет читает. В таблице

Subjects находится список всех предметов, то есть именно строки из этой таблицы нам необходимо получить, независимого от того, имеются ли соответствия в других связанных с ней таблицах. Поэтому мы объединили таблицы **TeachersSubjects** и **Subjects** при помощи правого внешнего соединения. Для того чтобы сохранились результаты первого объединения мы, при помощи левого внешнего объединения, связываем их с таблицей **Teachers**, в которой содержится информация о преподавателях. Из полученного результата видно, что предмет **Unity** на данный момент не читает ни один из преподавателей (рис. 3.8).

Subject	LastName	FirstName
ADO.NET	Cooper	Michael
C#	Nelson	Sophia
Discrete Math	Kirk	Emma
ITE1	Williams	Daniel
JavaScript	Nelson	Sophia
SQL Server	MacAlister	Henry
Unity	NULL	NULL
WIN10	Williams	Daniel

Рисунок 3.8. Список всех предметов и данные о преподавателях, которые их читают

Понятие FULL JOIN

Оператор **FULL JOIN (FULL OUTER JOIN)** позволяет создать полное внешнее объединение, при котором в результирующий набор включаются записи из левой таблицы даже в том случае, если для них отсутствуют соответствующие записи в правой таблице, а записи из правой таблицы будут помещены в результирующую таблицу даже тогда,

когда для них нет записей соответствия в левой таблице. Таким образом, полное внешнее соединение, по сути, является комбинацией левого и правого внешних объединений.

Приведем пример выполнения полного внешнего объединения для получения информации обо всех студентах и группах:

```
SELECT FirstName, LastName, GroupName
FROM Students AS S FULL JOIN Groups AS G
ON G.Id = S.GroupId
ORDER BY FirstName;
```

FirstName	LastName	GroupName
NULL	NULL	33GR12
Charlie	Thomas	32PR31
Charlotte	Becker	NULL
Emily	Taylor	32PR31
Grace	Evans	30PR11
Harry	Miller	29PR21
Jack	Jones	29PR21
Jessica	Brown	32PPS11
Joshua	Johnson	31PPS11
Lily	Wilson	30PR11
Oliver	Moore	32PR31

Рисунок 3.9. Информация обо всех студентах и группах

В полученной результирующей таблице находятся все записи из таблицы **Students** и все записи из таблицы **Groups** там, где отсутствует соответствующая запись из другой таблицы, возвращается неопределенное значение (**NULL**). В данном случае в группе **33GR12** нет ни одного студента и студентка **Charlotte Becker** пока не зачислена ни в одну из групп (рис. 3.9).

4. Домашнее задание

1. Вам необходимо создать многотабличную базу данных [Airport](#), в которой должна содержаться информация, связанная с работой аэропорта. В этой базе данных обязательно должны быть следующие таблицы: рейсы самолетов, билеты на рейсы (бизнес и эконом класс), пассажиры.
2. Используя полученные в этом уроке знания, вам необходимо написать к созданной базе [Airport](#) следующие SQL-запросы:
 - вывести все рейсы в определенный город на произвольную дату, упорядочив их по времени вылета;
 - вывести информацию о рейсе с наибольшей длительностью полета по времени;
 - показать все рейсы, длительность полета которых превышает два часа;
 - получить количество рейсов в каждый город;
 - показать город, в который наиболее часто осуществляются полеты;
 - получить информацию о количестве рейсов в каждый город и общее количество рейсов за определенный месяц;
 - вывести список рейсов, вылетающих сегодня, на которые есть свободные места в бизнес классе;
 - получить информацию о количестве проданных билетов на все рейсы за указанный день и их общую сумму;

4. Домашнее задание

- вывести информацию о предварительной продаже билетов на определенную дату с указанием всех рейсов и количестве проданных на них билетов;
- вывести номера всех рейсов и названия всех городов, в которые совершаются полеты из данного аэропорта.



Теория баз данных. Часть 2. Unit 4-6.

© Юрий Задерей.

© Компьютерная Академия «Шаг»
www.itstep.org.

Все права на охраняемые авторским правом фото-, аудио- и видеопрограммные изделия, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объеме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объем и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.