Язык запросов SQL:

SQL переводят на русский как *Структурированный Язык Запросов* . С помощью SQL-запросов можно создавать и работать с реляционными базами данных. Этот язык стал стандартом, поэтому если ты хочешь работать с базами данных, то ты должен знать этот язык как каждую дырку в своих зубах.

SQL определяется *Американским Национальным Институтом Стандартов* и *Международной Организацией по стандартизации (ISO)* . Несмотря на это, некоторые производители баз данных вносят изменения и дополнения в этот язык. Эти изменения незначительны и основа остаётся совместимой со стандартом. Мы с тобой будем учить язык в чистом виде без изменений.

Что такое реляционная база данных? Это таблица, в которой в качестве столбцов выступают поля данных, а каждая строка хранит данные. В каждой таблице должно быть одно уникальное поле, которое однозначно будет идентифицировать строку. Это поле называется ключевым. С ним мы уже познакомились в разделе "Delphi и базы данных", я его называл *Key1* . Эти поля очень часто используются для связывания таблиц (как это мы уже делали). Но даже если у тебя таблица не связана, ключевое поле всё равно обязательно. Представь, что ты пишешь телефонную базу данных. Сколько у тебя будет "Ивановых"? Как ты будешь отличать их? Вот тут тебе поможет ключ. В качестве ключа желательно использовать численный тип и если позволяет база данных, то будет лучше, если он будет типа "autoincrement" (автоматически увеличивающееся/уменьшающееся число).

Столбцы в базе данных, также должны быть уникальными, но в этом случае не обязательно числовыми. Их можно называть как угодно, лишь бы было уникально и тебе понятно, а остальное никого не касается.

SQL может быть двух типов: *интерактивный* и *вложенный* . Первый - это отдельный язык, он сам выполняет запросы и сразу показывает результат работы. Второй - это когда SQL язык вложен в другой, как например в C++ или Delphi. В этом разделе я буду представлять, что у нас интерактивный SQL, а в практической части мы будем работать с вложенным (SQL будет вложен в Delphi).

Интерактивный SQL более близок к стандартному, а во вложенном очень часто встречаются отклонения и дополнения. Например, в стандартном SQL различаются только два типа данных: строки и числа, но некоторые производители добавляют свои типы

(Date, Time, Binary и т.д.). Числа в SQL делятся на два типа: целые (INTEGER или INT) и дробные (DECIMAL или DEC). Строки ограничены размером в 254 символа.

Для дальнейшего понимания материала, нам понадобятся наглядные таблицы. Представим, что на твою прогу набралось много покупателей. В этом случае тебе понадобится база данных, чтоб эффективно вести их учёт. Прежде чем создавать поля, нужно обязательно продумать, какие данные нужны, и как они будут представлены. В нашем случае понадобиться:

- Наименование проги (ProgName), которую купили (у тебя может быть несколько прог).
- Цена (Cost).
- Фамилия покупателя (LastName).
- Имя покупателя (FirstName).
- Адрес электронной почты (Email).
- Страна проживания (Country). Неплохо знать, где живёт твой пользователь.
-
- Количество купленных лицензий (LecNumber).
- Регистрационный код (RegNum).

На первый взгляд всё достаточно легко. Но это только на первый. Представим, что у тебя 2 проги, тогда у каждого второго в таблице будет одинаковое первое поле. Главная проблема, что это поле громоздкое, как минимум 20 символов. Чтобы избавится от этого недостатка необходимо вынести названия и цену прог в отдельную таблицу. То же самое со странами. Большинство твоих пользователей сосредоточено в трёх или четырёх странах, поэтому будет очень много повторений. Страны я тоже вынесу отдельно, и буду использовать как справочник. В этом случае, в основной таблице будут храниться только ссылки на справочник, что значительно уменьшит базу данных.

Связь между этими таблицами будет сложная. Ну а кому сейчас легко? Посмотри на рис 1. Там показана эта связь. В качестве "Key1" выступают первичные ключи, а "Key2" - вторичные. Что такое вторичный ключ ты поймёшь в процессе чтения моих статей.

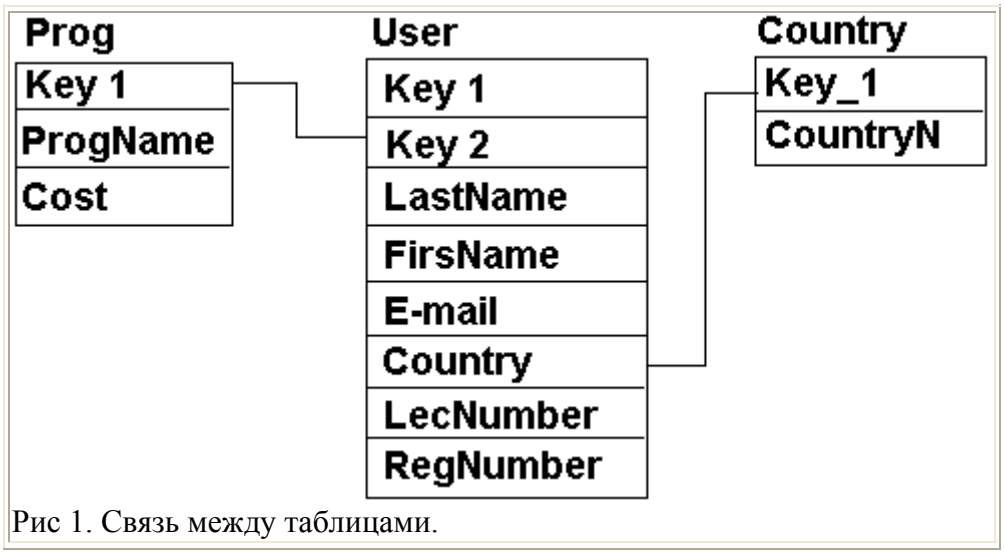

Рис 1. Связь между таблицами.

Теперь мы переходим к изучению команд SQL языка. Для этого мы создадим простой запрос.

```
SELECT ProgName, Cost
```

```
FROM  Prog;
```

SELECT все запросы начинаются с этой команды, и означает - "выбрать". После команды перечисляются имена полей, которые необходимо выбрать из базы данных.

FROM - эта команда также присутствует во всех запросов и указывает на таблицу (или несколько таблиц) к которой направлен этот запрос.

Точка с запятой ";" используется в интерактивном SQL в качестве индекатора конца запроса. В некоторых системах эту роль выполняет "\". Во вложенном такого индикатора нет.

В итоге мы получаем, что наш запрос выберет все строки с полями "ProgName" и "Cost" из таблицы "Prog". Как видишь, ничего сложного тут нет. Такой маленький запрос возвращает нам все поля из таблицы, кроме ключевого. Если нужно вывести всё, то можно перечислить все поля или воспользоваться значком "*". Следующий запрос выведет все поля из таблицы "prog":

```
SELECT  *
FROM  Prog;
```

Если вы используете звёздочку, то выведутся все поля в том порядке, в каком они находятся в таблице. Если нужно вывести в другом порядке, то перечисли все поля в том порядке, в каком твое душе угодно.

В SQL существует два ключевых слова, которые характеризуют параметры вывода информации: *ALL* и *DISTINCT* . Параметр *All* означает, что выводить нудно все строки, а *DISTINCT* означает, что ненужно выводить повторяющиеся строки. Следующий запрос выведет все возможные имена программ. Если в таблице встретится две строки с одинаковым именем программы, то он выведет это имя только один раз.

```
SELECT  DISTINCT ProgName
FROM  Prog;
```

Если не задать параметр *DISTINCT* , то считается, что надо использовать параметр *ALL* .

Сегодня мы начнём писать полноценные запросы. Если ты захочешь их отлаживать на какой-нибудь базе данных, то будь внимателен, тебя подстерегают некоторые опасности:

- Некоторые базы данных чувствительны к регистру.
- Возможно, что твоя база данных может воспринимать текст только в верхнем регистре.
- Некоторые производители отклоняются от стандартов, и возможны отклонения от того, что я говорю.

Я уже предостерегал тебя в прошлом номере, но всё же мне захотелось повториться (повторенье - мать мученья).

Следующий оператор, с которым я хочу тебя познакомить, будет *WHERE* . Этот оператор задаёт критерии поиска. Например, мне надо выбрать все записи из таблицы User.db, где в поле *Country* содержится значение "USA". В этом случае я должен написать следующий запрос:

```
SELECT *
FROM User
WHERE Country LIKE "USA"
```

Результатом запроса будут все строки содержащие в поле *Country* значение *"USA"* . Если проговорить последнюю строку запроса, то она будет звучать так: "Где поле *Country* равно USA". Ещё одно замечание: в запросам, строки выделяются кавычками. В зависимости от базы данных, кавычки могут быть одинарными или двойными. В Delphi это одинарные кавычки, но я буду использовать в этих статьях двойные, как это предусмотрено стандартом.

Я немного поторопился и использовал в запросе ключевое слово *LIKE* . Это слово идентично знаку "=" (равно), только используется для сравнения строк. Если тебе надо сравнивать числа, то ты должен использовать знак равно (=), а если строки, то оператор *LIKE* .

Давай теперь рассмотрим, как этот запрос будет выглядеть при поиске по числам. Для этого найдём все строки из той же базы, где количество лицензий равно 1.

```
SELECT *
FROM User
WHERE LecNumber =1
```

В этом случае мы производим поиск по числовому полю, поэтому используем знак равно (=). Результатом запроса будут все строки, содержащие в поле *LecNumber* значение 1.

Во всех запросах я использовал единственный оператор "=" (равно). Но это не значит, что больше ничего нет. Стандарт разрешает использовать следующие операторы:

- "=" - Равный
- ">" - Больше
- "<" - Меньше
- ">=" - Больше или равно
- "<=" - Меньше или равно
- "<>" - Неравно

Операторы "больше", "меньше" и др. можно использовать не только с числами, но и со строками. В этом случае буква "А" будет меньше чем "Р". При сравнении строк разного регистра, меньшим оказывается строка в верхнем регистре, например, "А" будет меньше "а" и "Р" будет меньше "а". При сравнении строк, тебе также необходимо использовать кавычки.

```
SELECT *
FROM User
WHERE LecNumber >1
```

Результатом этого запроса будет строки, в которых количество лицензий больше 1.

Теперь усложним запрос с помощью булевых операторов. В стандарте предусмотрено три булевых оператора: AND (логическое "и"), OR (логическое "или"), NOT (логическое "не").Сразу же рассмотрим пример:

```
SELECT *
FROM User
```

```
WHERE Country LIKE "USA"
AND LecNumber >1
```

Результат запроса - все строки содержащие в поле "Country" значение "USA" и в поле LecNumber значение больше "1". Если какое-то из этих условий не выполнится, то строка не будет выбрана.

```
SELECT *
FROM User
WHERE Country LIKE "USA"
OR LecNumber =1
```

Результат запроса - все строки содержащие в поле "Country" значение "USA" или в поле LecNumber значение "1". Для того, чтобы строка была выбрана, необходимо чтобы хотя бы одно из этих условий выполнилось.

```
SELECT *
FROM User
WHERE Country LIKE "USA"
AND NOT LecNumber =1
```

Результат запроса - все строки содержащие в поле "Country" значение "USA" и в поле LecNumber значение не равное "1". Для того, чтобы строка была выбрана, необходимо чтобы оба условия были верны (потому что мы используем оператор "AND").

Обрати внимание, что во втором условии "NOT LecNumber =1" оператор "NOT" стоит вначале. Вот именно там он и должен стоять, и не вздумай его совать в середину или даже в конец условия. В принципе, это условие идентично условию "LecNumber <>1", но это не значит, что этот оператор не нужен. Иногда он действительно очень удобен.

При использовании булевых операторов ты можешь использовать скобки:

```
SELECT *
FROM User
WHERE Country LIKE "USA"
AND (LecNumber =1 OR LecNumber =2)
```

Результат запроса - все строки содержащие в поле "Country" значение "USA" и в поле LecNumber значение не равное "1" или "2". Для того, чтобы строка была выбрана, необходимо чтобы условие Country LIKE "USA" и условие в скобках были верны.

Продолжаем изучать язык запросов SQL. На прошлом занятии мы покончили с булевыми операторами. Сегодня мы познакомимся ещё с несколькими операторами, упрощающими и усиливающими поиск необходимой информации. Сегодня это будут операторы IN, BETWEEN, IS NULL. А также , мы познакомимся с шаблонами.

Начну я с оператора IN. В принципе, можно создавать запросы и без него, но он упрощает SQL код и делает его более наглядным. Сразу рассмотрим пример:

```
SELECT *
FROM User
WHERE Country LIKE "USA" or Country LIKE "RUSSIA" or
      Country LIKE "GERMANY";
```

Этот запрос вполне работающий, но представь, что тебе так надо перечислить 20 или более стран. В этом случае запрос раздуется как земной шар. Вот как этот же запрос будет выглядеть с оператором IN.

```
SELECT *
FROM User
WHERE Country IN ("USA", "RUSSIA", "GERMANY");
```

Намного проще и меньше места. Если внимательно осмотреть два предыдущих запроса, то можно и без моих объяснений понять, что делает оператор IN. В данном случае он выведет все записи, в которых *Counrty* имеет одно из значений указанных в скобках.

Если используется числовое поле, то кавычки надо убрать:

```
SELECT *
FROM User
WHERE LecNumber IN (1, 2, 3)
```

В этом случае мы увидим все записи, где поле *LecNumber* равно 1 или 2 или 3. Вместе с оператором IN и со всеми последующими операторами ты можешь смело использовать булевы операторы, например:

```
SELECT *
FROM User
WHERE LecNumber NOT IN (1, 2, 3)
```

Теперь перейдём к рассмотрению оператора BETWEEN. Он также нужен только для удобства и вы можете спокойно обойтись без него. Снова перейдём к примеру. Допустим, что надо вывести все строки, где поле *LecNumber* больше либо равно 1 и меньше либо равно 5. Этот запрос будет выглядеть так:

```
SELECT *
FROM User
WHERE LecNumber >= 1 AND LecNumber <= 5
```

С оператором BETWEEN та же запись будет выглядеть так:

```
SELECT *
FROM User
WHERE LecNumber BETWEEN 1 AND 5
```

Как ты уже понял, этот оператор задаёт диапазон чисел. Ты должен помнить, что конечные точки включаются в результат запроса.

Этот оператор можно использовать и с строковыми полями:

```
SELECT *
FROM User
WHERE Country BETWEEN "A" AND "R";
```

В результат этого запроса войдут такие страны как GERMANY, AUSTRY, но не войдёт RUSSIA. Почему? Да потому что "GERMANY" меньше чем "R", а "RUSSIA" больше, потому что в слове "Russia" больше букв.

Оператор IS NULL означает нулевое значение. Сразу отмечу, что нулевое значение не равно 0 или "" (пустой строке). Ноль - это ноль, то есть не заполненное значение. В принципе, этот оператор прост, да и не очень важен, потому что ты очень редко будешь с ним работать. Поэтому я просто приведу пример и мы пойдём дальше:

```
SELECT *
FROM User
WHERE Country IS NULL;
```

Теперь рассмотрим ещё два оператора - % (процент) и _ (подчёркивание). Точнее сказать это не операторы, это управляющие символы используемые в шаблонах. Короче, ты их будешь использовать вместе с оператором LIKE. Представь, что тебе нужно вывести все записи, которые в поле Country содержат значение начинающееся на "R". Твой запрос будет выглядеть так:

```
SELECT *
FROM User
WHERE Country LIKE "R%";
```

Значок % (процент) означает любое количество символов, т.е. в результат войдут все слова начинающиеся на R и содержащие потом любое количество любых символов. Рассмотрим ещё пример: "A%C". В результат такого шаблона войдут слова: ананас, атас и другие, главное, чтобы они начинались на "A" и заканчивались на "C".

Теперь познакомимся с _ (подчёркивание) - оно означает один любой символ. Если % означал любое количество символов, то _ означает только один. Рассмотрим пример: "T_K". В результат такого шаблона войдут слова: ТОК, ТУК, ТИК, но не войдёт ТУПИК и другие слова, содержащие между буквами *Т* и *К* больше чем одну букву.

Этот символ очень удобен, особенно для тех, кто прекрасно знает русский язык. Представь, что тебе надо найти слово "корова", а ты не знаешь, как писать "корова" или "карова". В этом случае просто пиши "к_рова" и ты не ошибёшься. Или вообще "к_р%". Красиво? А в принципе, это то же самое. Вот если бы в школе диктанты можно было бы так писать.

Работа с несколькими таблицами:

С одной таблицей мы научились работать. Теперь пора научиться связывать несколько таблиц, как мы это уже делали на уроках программирования Delphi и базы данных. Помимо этого, нас ждёт несколько приёмов, украшающих вывод.

Ещё на первом занятии я нарисовал схему из трёх баз данных. Если ты забыл, то взгляни на рисунок 1, чтобы освежить ту схему в своей оперативной памяти.

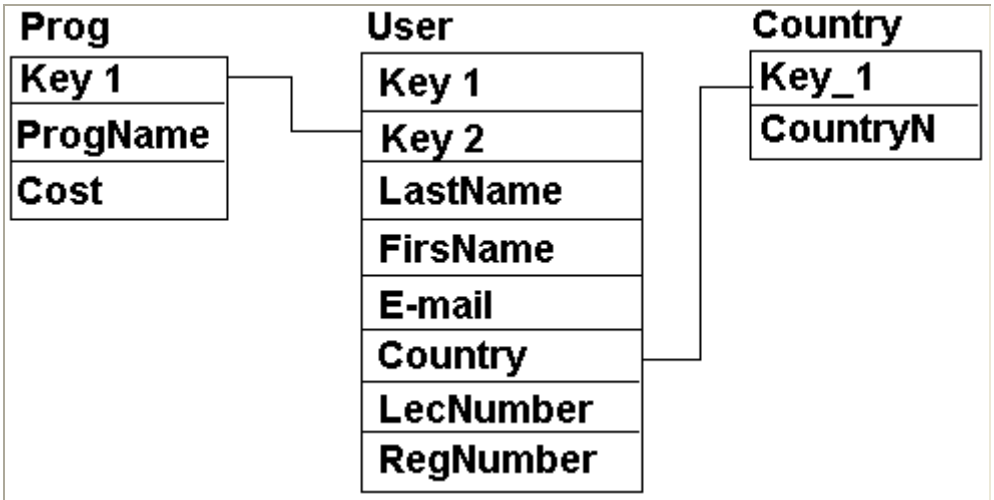

Рис 1. Связь между таблицами.

До сегодняшнего дня, мы обращались только к одной таблице. Сейчас мы выведем сразу две, причём с учётом связи. Сначала я покажу пример, а потом мы рассмотрим всё построчно.

```
SELECT *
FROM Prog, User
WHERE Prog.Key1=User.Key2
```

Первая строка, как всегда говорит, что надо вывести все поля (SELECT *). Вторая строка говорит, из каких баз данных надо это сделать (FROM Prog, User). На этот раз у нас здесь указано сразу две базы. Третья строка показывает связь (Prog.Key1=User.Key2). Для того, чтобы указать к какой базе относиться ключ Key1, я записываю полное имя поля как ИмяБазы.ИмяПоля. Если имя поля уникально для обеих таблиц (как Key1, которое есть только в таблице User), то можно имя таблицы опускать. Например, запрос мог выглядеть так:

```
SELECT *
FROM Prog, User
WHERE Prog.Key1= Key2
```

А вот Key1 записать без имени таблицы нельзя, потому что такое имя поля есть в обеих таблицах. Поэтому необходимо явно указать, Key1 какой именно таблицы интересует нас. Представим, что у нас есть две таблицы:

```
Prog.db
Key1  ProgName     Cost
-----------------------------------
1     Windows 95    100
2     Windows 98    120

User.db
Key1 Key2  LastName
--------------------------------
1    1     Иванов
2    1     Петров
3    2     Сидоров
```

Тогда результатом такого запроса может быть:

```
Prog.db                          User.db
```

```
Key1  ProgName     Cost       Key1 Key2  LastName
-------------------------------------------------
1     Windows 95    100        1    1    Иванов
1     Windows 95    100        2    1    Петров
2     Windows 98    120        3    2    Сидоров
```

Рассмотрим ещё пример:

```
SELECT *
FROM Prog, User
WHERE Prog.Key1= Key2
       AND ProgName LIKE 'Windows 95'
```

Результатом этого запроса, при тех же таблицах будет:

```
Prog.db                        User.db
Key1  ProgName     Cost       Key1 Key2  LastName
-------------------------------------------------------------------------
1     Windows 95    100        1   1     Иванов
1     Windows 95    100        2   1     Петров
```

Как видишь, всё очень просто. Ты можешь использовать с такими запросами всё, что мы уже изучали и практически всё, что ещё изучим. Давай теперь приукрасим наши запросы. Например, представим, что цена указана в долларах :), а мы хотим перевести в рубли. Корректируем немного запрос:

```
SELECT Prog.Key1, Prog.ProgName, Prog.Cost*2,
  Cost.Key1, Cost.Key2, Cost.LastName
FROM Prog, User
WHERE Prog.Key1= Key2
```

Под цифрой 2 внутри запроса (Prog.Cost*2) я подразумеваю курс доллара (размечтался я). Результат запроса будет:

```
Prog.db                        User.db
Key1  ProgName     Cost       Key1 Key2  LastName
-------------------------------------------------------------------------
1     Windows 95    200        1    1     Иванов
1     Windows 95    200        2    1     Петров
```

Давай ещё немного украсим:

```
SELECT Prog.Key1, Prog.ProgName, Prog.Cost*2 'руб',
  Cost.Key1, Cost.Key2, Cost.LastName
FROM Prog, User
WHERE Prog.Key1= Key2
```

*Prog.Cost*2 'руб'* - эта запись говорит, что к каждому значению полю надо прибавить строку 'руб'. Результат этого запроса:

```
Prog.db                            User.db
Key1  ProgName     Cost            Key1 Key2  LastName
-----------------------------------------------------
1     Windows 95   200 руб         1    1     Иванов
1     Windows 95   200 руб         2    1     Петров
```

Здесь есть маленький недостаток, приходиться перечислять поля, которые нужно вывести и * уже не работает. Так что если понадобиться вывести все поля, то придётся их все писать после команды SELECT.

Вот стало и ещё красивее. Теперь остаётся мне выполнить обещание, и показать, как сортируется вывод. Для сортировки используется команда *ORDER BY* . После этого пишутся поля, по которым надо отсортировать. В самом конце нужно поставить *ASC* (сортировать в порядке возрастания) или *DESC* (в порядке убывания). Если ты не ставишь *ASC* или *DESC* , то таблица сортируется по возрастанию и подразумевается параметр *ASC* . Например:

```
SELECT *
FROM Prog
ORDER BY ProgName
```

Результатом будет таблица Prog, отсортированная по полю ProgName в порядке возрастания . Если ты хочешь отсортировать в обратном порядке, то нужно обязательно поставить *DESC* :

```
SELECT *
FROM Prog
ORDER BY ProgName DESC
```

Связанные таблицы сортируются также:

```
SELECT *
FROM Prog, User
WHERE Prog.Key1= Key2
ORDER BY ProgName, Country ASK
```

SQL калькулятор:

До сих пор мы просто выводили данные из таблицы, но сегодня ты увидишь, что SQL может производить некоторые математические действия. Эти действия очень слабенькие, но в большинстве случаев даже этого бывает достаточно. Если тебе нужны более сложные расчёты, то придётся их выполнять вручную. Зачем нужно вставлять математику в запросы, ведь это можно сделать программно? Ответ в этой статье.

Ты можешь получить данные из базы, и потом выполнять математику, но если есть возможность вставить эту математику в запрос, то лучше это сделать. В этом случае ты освободишь клиентскую машину от лишней загрузки, и тем более сервер будет выполнять эти расчёты намного быстрее. Когда ты делаешь расчёты программно, то выполняется два просмотра всей базы. Во время первого выбираются данные, а во время второго идёт расчёт. Когда математика вставлена в запрос, то все действия выполняются за один проход - выбираются данные и одновременно происходит математика.

Давай переходить к изучению SQL-математики. Как я уже сказал, она не сложнее школьной программы для второклассника, поэтому особых мозгов сегодня не понадобиться. Выкладывай свои извилины в холодильник, пускай они охладятся после тяжёлого дня.

Для полного счастья нам доступны несколько функций:

- *COUNT* - подсчёт количества строк.
- *SUM* - подсчёт суммы.
- *AVG* - подсчёт среднего значения.
- *MAX* - поиск максимального значения.
- *MIN* - поиск минимального значения.

Всё очень просто. Теперь рассмотрим пару примерчиков:

```
SELECT COUNT(LecNumber)
FROM User
```

Этот запрос просто подсчитывает количество строк в базе. Не надо обращать внимания на параметр в скобках (LecNumber) у COUNT. Вроде всё просто, и не обязательно было для этого создавать запрос. Ладно, усложняем.

```
SELECT COUNT(LecNumber)
FROM User
WHERE LecNumber=1
```

Вот это уже интересней. Этот запрос опять подсчитывает количество строк, но теперь результатом будет количество народу, у которых поле LecNumber = 1. Тоже просто? Ещё усложняем:

```
SELECT COUNT(DISTINCT Country)
FROM User
```

В результате будет количество разных стран присутствующих в базе. Например, если в твоей базе четыре строки и поле Country для строк равны: Россия, Америка, Украина, Россия, то результатом будет 4 (четыре вида стран).Очень интересных эффектов можно добиться, если использовать математику вместе с GROP_BY. Если ты всё же используешь то ярр

Теперь рассмотрим что-нибудь более возбуждающее.

```
SELECT SUM(LecNumber)
FROM User
```

В результате этого запроса мы получим общее количество лицензий. Это количество не то, что было в случае с COUNT, это сумма данного поля для всех строк. Если немного напрячь то, что ты ещё не успел выложить в холодильник, то становиться видно одно правило: поле должно быть числовым (нельзя производить суммирование строковых полей).

Остальные функции работают также, поэтому я не вижу смысла приводить для них примеры. Уж лучше я продвинусь дальше и расскажу кое-что ещё про математику в SQL. Давай взглянём на следующий пример:

```
SELECT LecNumber+'шт.'
FROM User
```

Этот запрос выводить количество лицензий и единицу измерения в одном столбце. Здесь к числу прибавляется некий текст. Прибавлять можно и числовые значения:

```
SELECT LecNumber+1
```

```
FROM User
```

И самое интересное - прибавлять можно и другое поле:

```
SELECT LecNumber+Cost
FROM User
```

Cost - это поле в базе, но не надо его искать. В той базе, что я использую, нет такого имени, я его придумал только для удобства. И Наглядности.

Ещё можно вычитать умножать и делить значение полей. Потренируйся на эту тему сам. Должно тебе помочь.

Объединения:

Потихоньку мы созреваем до создания сложных запросов. Сегодня мы научимся формировать выходные данные в группы. Это очень удобная вещь, и если ты её поймёшь, то будешь использовать очень часто, потому что она очень сильно облегчает жизнь.

Представим, что у нас есть две таблицы User1 и User2:

```
User1.db
Key1  OC        LastName
-------------------------
1     Win       Иванов
2     Unix      Петров
3     Win       Яковлев
4     Win       Сидоров
5     Win       Ковалёв
6     Unix      Амаров

User2.db
Key1  OC        LastName
-------------------------
2     Unix      Богров
3     Win       Сидоров
4     OS2       Ковалёв
```

Мы хотим получить список всех пользователей Unix из двух таблиц сразу. Для этого нужно выполнить запрос выбора к первой таблице, а потом ко второй. Результатом будет две выходные таблицы. А если мы хотим получить одну? Для этого можно воспользоваться объединением - оператор UNION. Вот как это будет выглядеть:

```
SELECT *
FROM User1.db
WHERE OC LIKE 'Unix'
 UNION
  SELECT *
  FROM User2.db
  WHERE OC LIKE 'Unix';
```

результатом будет одна таблица:

```
User1.db
Key1  OC        LastName
-------------------------
2     Unix      Петров
```

```
6     Unix      Амаров
2     Unix      Богров
```

Не плохо?

Чтобы запрос не завершился ошибкой, он должен удовлетворять следующим условиям:

- Количество и типы полей должны быть одинаковыми.
- Символьные поля должны иметь одинаковое число символов.

Если одного поля в одном из запросов нет, то его можно заменить. Например:

```
SELECT Key1, OC, LastName
FROM User1.db
WHERE OC LIKE 'Unix'
 UNION
  SELECT Key1, OC, 'NO FOUND'
  FROM User2.db
  WHERE OC LIKE 'Unix';
```

результатом будет одна таблица:

```
User1.db
Key1  OC        LastName
-------------------------
2     Unix      Петров
6     Unix      Амаров
2     Unix      NO NAME
```

Здесь мы вместо поля *LastName* подсовываем текст 'NO FOUND', чтобы количество и тип полей совпадали.

Таким образом можно украсить запрос вот до такого вида:

```
SELECT Key1, OC, LastName, 'Table 1'
FROM User1.db
WHERE OC LIKE 'Unix'
 UNION
  SELECT Key1, OC, 'NO FOUND', 'Table 2'
  FROM User2.db
  WHERE OC LIKE 'Unix';
```

Результатом будет одна таблица:

```
User1.db
Key1  OC        LastName
--------------------------------
2     Unix      Петров    Table 1
6     Unix      Амаров    Table 1
2     Unix      NO NAME   Table 2
```

И на последок упорядочим наш вывод:

```
SELECT Key1, OC, LastName, 'Table 1'
FROM User1.db
WHERE OC LIKE 'Unix'
 UNION
  SELECT Key1, OC, 'NO FOUND', 'Table 2'
```

```
  FROM User2.db
  WHERE OC LIKE 'Unix';
   ORDER BY 3
```

ORDER BY 3 - говорит, что надо упорядочить вывод по третьему столбцу. Результатом будет одна таблица:

```
User1.db
Key1  OC        LastName
---------------------------------
2     Unix      NO NAME   Table 2
6     Unix      Амаров    Table 1
2     Unix      Петров    Table 1
```

Запросы с подзапросами:

В прошлом месяце я немного поторопился, когда начал тебе рассказывать про объединения, потому что сегодняшние знания я должен был дать немного раньше. В принципе, особой разницы нет, но всё же было бы удобнее сначала воспринять этот материал, а потом уже объединения. В дальнейшем я постараюсь давать материал более последовательно, не забегая вперёд.

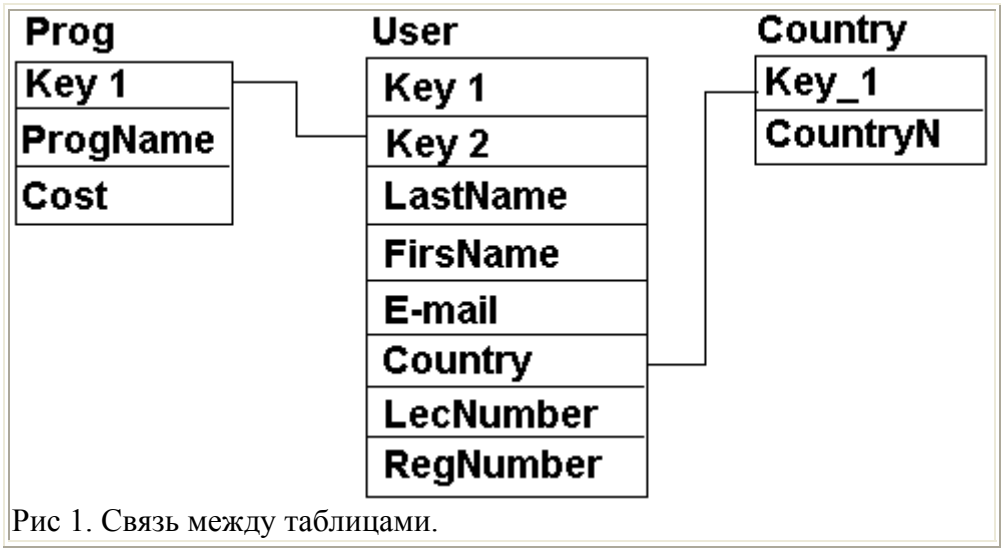

Рис 1. Связь между таблицами.

Итак, мы переходим к дальнейшему изучению SQL. Сегодня нам предстоит изучить подзапросы. Что это значит? SQL позволяет вставлять одни запросы внутрь других. Сейчас мы рассмотрим пример, но сначала я напомню тебе структуру наших таблиц, смотри рисунок 1.

Представим, что нам надо вывести все записи из таблицы User.db, которые соответствуют программе MyProg.exe. На первый взгляд запрос очень простой. Нужно написать:

```
SELECT *
FROM User1.db
WHERE Key2 = Key1 для программы MyProg.exe;
```

Возникает вопрос: "Какой ключ 1 у программы?". Для того, чтобы это узнать мы можем воспользоваться вложенным запросом. Теперь посмотрим на этот запрос:

```
SELECT *
FROM User1.db
WHERE Key2 =
   (SELECT Key1 FROM Prog
   WHERE ProgName LIKE 'MyProg.exe');
```

Сначала SQL выполнит внутренний запрос, который расположен в скобках и результат подставит в внешний запрос.

Всё прекрасно, но должно выполнятся два условия: у внутреннего запроса, в качестве результата должен быть только один столбец. Это значит, что ты не можешь написать во внутреннем запросе *SELECT* * , а можно только *SELECT ИмяОдногоПоля*. Помни, что имя должно быть только одно и тип его должен совпадать с типом сравниваемого значения (в нашем случае с типом key2 из таблицы User1).

Результатом внутреннего запроса, должна быть только одна строка. Если внутренний запрос вернёт несколько строк или вообще ничего не вернёт, то могут возникнуть проблемы. Так что ты должен очень аккуратно использовать подзапросы, потому что они могут привести к ошибке. Для большей надёжности используй с подзапросом оператор DISTINCT. Единственный случай, когда подзапрос может выдавать в результате несколько строк, это когда в основном запросе используется оператор IN:

```
SELECT *
FROM User1.db
WHERE Key2 IN
   (SELECT Key1 FROM Prog
   WHERE ProgName LIKE 'MyProg.exe');
```

Здесь, вместо знака равно я использовал оператор IN (key2 IN (подзапрос)). Так что для надёжности запроса можно использовать не только DISTINCT, но и оператор IN. Это желательно делать даже в тех случаях, когда ты уверен, что результатом будет только одна строка.

Хочу ещё обратить твоё внимание, что я написал запрос так: key2= (подзапрос). Ты должен писать также. Ни в коем случае нельзя эту запись написать как (подзапрос)=Key2. Подзапрос должен идти после знака = или любого другого, но никак ни перед.

Самое сильно, что ты можешь обращаться из внутреннего запроса к внешнему. Как это делать?

```
SELECT *
FROM User1.db outer
WHERE Key2 =
   (SELECT Key1
    FROM Prog inner
   WHERE key1 = outer.Key2);
```

Слова outer и inner - псевдонимы, которые назначаются таблицам user1 и prog соответственно. Это значит, что когда мы пишем outer, это то же самое, что и написать User1. Такой запрос будет выполнятся по следующему алгоритму:

- Выбрать строку из таблицы User1.db в внешнем запросе. Это будет текущая строка-кандидат.
- Сохранить значения из этой строки-кандидата в псевдониме с именем outer.

- Выполнить подзапрос. Везде, где псевдоним данный для внешнего запроса найден (в этом случае "outer"), использовать значение для текущей строки-кандидата. key1 = outer.Key2. Использование значения из строки- кандидата внешнего запроса в подзапросе называется - внешней ссылкой.
- Оценить предикат внешнего запроса на основе результатов подзапроса выполняемого в предыдущем шаге. Он определяет - выбирается ли строка-кандидат для вывода.

Пример не совсем удачный, но возможно ты придумаешь лучше. Попробуй остановится на секунду и подумать над предложенным мной запросом. Если ты сможешь с эти разобраться, то ты поймёшь всю силу программирования на SQL

Ещё несколько операторов:

Продолжаем знакомится с операторами SQL упрощающих вывод данных из таблиц. Первое, с чем нам предстоит сегодня познакомится – оператор *EXISTS* . Это простая проверка на существование. Этот оператор относится к выражениям Буля.

Сразу рассмотрим пример:

```
SELECT cnum, cname, city
FROM User1
WHERE EXISTS
   ( SELECT *
   FROM User2
   WHERE OC = “Unix” );
```

Если ты читал предыдущую статью, то наверно заметил здесь подзапрос. Обломися бабка, мы на пароходе. Это не подзапрос, потому что он выполняется только один раз. Внутренний запрос выбирает все записи, где ОС равна “Unix”. Оператор EXISTS проверяет, если был какой-то результат, то генерирует True, а значит выполниться условие EXISTS. После этого выполнится внешний запрос:

```
SELECT cnum, cname, city
FROM User1
```

Так как EXISTS Булев оператор, его можно использовать с другими Булями. Вот тебе пример с NOT:

```
SELECT cnum, cname, city
FROM User1
WHERE EXISTS
   ( SELECT *
   FROM User2
   WHERE OC = “Unix” );
```

В этом случае внешний запрос выполниться только если внутренний не выведет ни одной строки.

Теперь нам предстоит познакомится с операторами ANY, SOME, и ALL. Первые два оператора абсолютно одинаковы. Оба они дают один и тот же результат, поэтому ты можешь использовать тот, который больше подходить под твои пальцы. Мне больше нравиться ANY, потому что это слово короче аж на одну букву :).

```
SELECT cnum, cname, city
FROM User1
```

```
WHERE OC=ANY
   ( SELECT OC
   FROM User2
   );
```

Здесь сначала выполняется внутренний запрос, выбирая все ОС из базы User2. Затем выполняется внешний запрос, который выберет все строки где встретилась любая (ANY) из ОС внутреннего запроса. То есть, результатом будут все строки из User1, в которых встречаются ОС такие же как и в User2.

Результат следующего запроса будет абсолютно таким же:

```
SELECT cnum, cname, city
FROM User1
WHERE OC= SOME
   ( SELECT OC
   FROM User2
   );
```

Аналогично будет работать и следующий запрос:

```
SELECT cnum, cname, city
FROM User1
WHERE OC=IN
   ( SELECT OC
   FROM User2
   );
```

Я везде использую знак «=», но ANY и SOME могут работать и с операторами < или >.

Теперь нам предстоит познакомится с ALL

```
SELECT cnum, cname, city
FROM User1
WHERE NumberLesens>ALL
   ( SELECT NumberLesens
   FROM User2
   );
```

Результатом этого запроса будут все строки, в которых количество лицензий (NumberLesens) больше чем у всех из таблицы User2.

Работа с полями:

До сегодняшнего дня мы занимались выборками из базы данных. Я рассказал тебе практически все возможные приёмы для формировании данных из таблиц. Теперь пора научиться вставлять, удалять и модифицировать строки.

Для всего этого в SQL есть три магических оператора: INSERT (вставить), UPDATE (модифицировать), DELETE (удалить). Рассмотрим вставку строки, для этого используется простейшая конструкция:

```
INSERT INTO Имя Таблицы
VALUES (Значение 1, Значение 2, и т.д.);
```

Это общий вид. После оператора VALUES идёт перечисление всех полей строки. Теперь взглянём на конкретный пример:

```
INSERT INTO User1
VALUES ('Иванов', 'Сергей', 34);
```

Этой командой мы вставили строку и присвоили значения полям. В моей таблицы три поля: первые два поля строковые (Фамилия и Имя), последнее поле - целое число (возраст). Типы данных обязаны совпадать с теми, что установлены в таблицы, иначе секир башка твоему запросу.

А если ты не хочешь задавать все поля? Тогда ты можешь оставить их пустыми с помощью NULL:

```
INSERT INTO User1
VALUES ('Иванов', NULL, 34);
```

Как видишь, второе поле я оставил пустым и в него не будет заноситься значение.

А если у тебя таблица с большим количеством полей и ты хочешь заполнить только два из них? Неужели придется всем остальным полям ставить значение NULL? Нет. SQL - это достаточно продуманный язык и в нём есть на этот случай удобная вещичка:

```
INSERT INTO User1 (Family, Age)
VALUES ('Иванов', 35);
```

После конструкции *INSERT INTO* и имени базы я поставил скобки, где перечислил поля, которые необходимо заполнить (Фамилия и Возраст). В скобках после слова *VALUES* я перечисляю эти поля в той же последовательности, в которой перечислил перед этим (сначала фамилия, а потом возраст).

Теперь представь, что ты хочешь сохранить результат запроса SELECT в отдельной таблице. Для этого в SQL всё уже предусмотрено. Тебе нужно только написать:

```
INSERT INTO User1
SELECT *
FROM User2
WHERE Age=10
```

В этом примере сначала выполнится запрос SELECT:

```
SELECT *
FROM User2
WHERE Age=10
```

После его выполнения, результат будет занесён в таблицу User1. Ну как? Только не забудь, что количество столбцов в запросе и результирующей таблицы должно быть одинаково. А самое главное, это чтобы тип данных совпадал, иначе Гитлер капут.

Усложняем задачу. Теперь рассмотрим такой запрос:

```
INSERT INTO User1(Name,Age)
SELECT Name,Age
FROM User2
WHERE Age=10
```

Теперь в таблицу User1 будут перенесены только два столбца (имя и возраст). Здесь действуют те же ограничения - количество полей должно быть одинаково. Но есть и ещё одно - поля должны быть перечислены в таком порядке, чтобы типы и длина полей совпадали. У меня они перечислены так, что первое поле строковое, а второе целое число.

Двигаемся дальше. Мы смогли добавить строки, но надо и научиться изменять данные. Для этого нам доступна команда *UPDATE* . Сразу же попробуем взглянуть на пример:

```
UPDATE User1
SET age=65
```

Первая строка говорит о том, что нам надо обновить базу User1. Вторая строка начинается с оператора *SET* (установить). После этого я пишу поле, которое хочу обновить и присваиваю ему значение.

Этот маленький пример установит поле *age* у всех строк в значение 65. Если тебе нужно обновить только определённые строки, то ты должен написать так:

```
UPDATE User1
SET age=65
WHERE Name LIKE 'Вася'
```

Этот запрос установит значение 65 в поле *AGE* только тем строкам, в которых поле *Name* равно "Вася".

И снова усложняем себе жизнь.

```
UPDATE User1
SET age=age+1
```

Этот запрос увеличит во всех строках таблицы поле *Age* на единицу.

И наконец, обновление сразу нескольких полей:

```
UPDATE User1
SET age=age+1, Name='Иван'
WHERE Family LIKE 'Сидоров'
```

Этот запрос увеличит поле *Age* на единицу и установит поле *Name* в *"Иван"* во всех строках, где поле *Family* равно *"Сидоров"*.

С обновлением полей покончено, теперь мы переходим к удалению строк из таблицы. Для этого есть команда *DELETE* :

```
DELETE FROM User1
```

Всё очень просто, эта конструкция удаляет абсолютно все строки из таблицы User1. Можно сказать, что этим мы очищаем таблицу.

Теперь рассмотрим другой пример:

```
DELETE FROM User1
WHERE Age=10
```

Этот пример удаляет только те строки, в которых поле Age равно 10.

Создание, изменение и удаление таблиц:

Я уже рассказал практически все основы языка SQL. Теперь, с каждым разом материал становится всё сложнее и сложнее, хотя я стараюсь преподнести его как можно проще. Вот таким образом мы подобрались к вопросу создания изменения и удаления таблиц с помощью SQL.

Для создания таблицы используется команда CREATE TABLE, которая создаёт пустую таблицу. После этой команды нужно определить столбцы, их типы и размер.

```
CREATE TABLE
 ( <Имя столбца > <Тип> [(<Размер>)],
   <Имя столбца > <Тип> [(<Размер>)] ... );
```

Обрати внимание, что <Размер> я показал в квадратных скобках, потому что не все типы данных требуют указания размера поля.

Давай посмотрим, какие бывают типы данных:

- CHAR или CHARACTER - строковое поле. В качестве размера используется длина строки.
- DEC или DECIMAL - Десятичное число, т.е. число с дробной частью. Размер состоит из двух частей - точность и масштаб. Эти параметры нужно указывать через запятую. Точность показывает сколько значащих цифр имеет число. Масштаб показывает количество знаков после запятой. Если масштаб равен нулю, то число становится эквивалентом целого.
- NUMERIC - Такое же как DECIMAL. Разница только в том, что максимальное десятичное не может превышать аргумента точности.
- FLOAT - Опять число с плавающей точной, только в этом случае размер указывается одним числом, которое указывает на минимальную точность.
- REAL - то же, что и FLOAT, только размер не указывается, а берётся из системы по умолчанию.
- DOUBLE - то же, что и REAL, только размер побольше (чаще всего в два раза).
- INT или INTEGER - целое число. Размер указывать не надо, он подставляется автоматически.
- SMALLINT - то же, что и INTEGER, только его размер меньше. В большинстве случаев, если точность INTEGER равна 2 байтам, то точность SMALLINT равна 1 байту.

Сразу врываемся в реальный пример:

```
CREATE TABLE NovayaTablica
 (id    integer,
  name  char (10),
  city  char (10),
  Metr  declmal);
```

Этот запрос создаёт таблицу с именем NovayaTablica и четырмя полями.

Теперь я хочу открыть один маленький секрет. Когда ты работаешь с такими базами, как Access, то ты можешь использовать в качестве имён таблиц или полей русские слова. А

самое главное, что эти слова могут включать пробелы, например, ты можешь назвать таблицу как "Новая таблицы". Как же тогда обратится из запроса к такой таблице? Все названия включающие в себя пробелы должны заключатся в квадратные скобки.

```
CREATE TABLE [Новая таблица]
 (Идентификатор    integer,
  Имя  char (10),
  [Место расположения]  char (10),
  Метраж  declmal);
```

Обрати внимание, что все составные имена заключены в квадратные скобки. При работе с запросом, ты так же должен использовать такие скобки:

```
Select *
From [Новая таблица]
WHERE
  Идентификатор=10 and
  [Место расположения] LIKE 'Москва';
```

Всё прекрасно, но не все базы данных позволяют использовать русские и составные имена.

Теперь поговорим о создании индексов. В общем виде это выглядит так:

```
CREATE INDEX <Имя индекса> ON <Имя таблицы>
            (<Имя поля> [,<Имя поля>]...);
```

А вот и реальный пример:

```
CREATE INDEX NewIndex ON NewTable (name);
```

Здесь создаётся новый индекс с именем NewIndex в таблице NewTable для name. Если ты захочешь создать уникальный индекс, то ты должен написать так:

```
CREATE UNIQUE INDEX NewIndex ON NewTable (name);
```

При создании простых индексов ты можешь указать в скобках несколько полей, например:

```
CREATE INDEX NewIndex ON NewTable (name, city);
```

Это создаст составной индекс и проиндексирует таблицу сразу по двум полям name и city. ПОМНИ: При создании уникальных индексов такого делать нельзя. Поле должно быть одно.

Теперь поговорим о добавлении новых столбцов. Для этого существует команда ALTER TABLE. Вот так она выглядит в общем виде:

```
ALTER TABLE <Имя таблицы> ADD <Имя поля> <Тип> <Размер>;
```

Например:

```
ALTER TABLE Справочник ADD [Новое поле] INTEGER
```

Итак, мы так долго старались, а теперь одним махом уничтожим всё, что тут натворили. Для удаления полей используй:

```
ALTER TABLE [Имя таблицы] DROP [Имя поля]
```

А для удаления таблиц используется команда DROP TABLE:

```
DROP TABLE < Имя таблицы >;
```

Настройка таблиц:

Я уже рассказал о создании таблиц, изменении количества столбцов, удалении таблиц. Теперь подошло время рассказать про настройку полей в таблице, например, как указывать диапазон доступных значений для таблицы или нечто подобное.

При создании таблицы указывать диапазон допустимых значений. Если пользователь ввёл недопустимое значение, то программа просто отклонит это значение или сгенерирует ошибку. До сегодняшнего дня мы создавали таблицы, где мы опускали этот параметр и наши столбцы не имели ограничений кроме типа и размера поля. Рассмотрим общий случай:

```
CREATE TABLE Имя_Таблицы
 (
  Имя_Поля_1 Тип Ограничения,
  Имя_Поля_2 Тип Ограничения,
 );
```

Давай сразу создадим таблицу с двумя полями. Первое должно быть уникальным, а второе не должно содержать нулей:

```
CREATE TABLE NewTable
 (
  Name char (10) UNIQUE,
  email char (10) NOT NULL
 );
```

Таким образом мы создали таблицу с уникальным первым полем. Это позволяет нам использовать это поле в качестве уникального ключа, и за уникальностью будет следить сама база данных.

Для первичных ключей не нужно указывать уникальность, они уникальны от природы:

```
CREATE TABLE NewTable
 (
  Name char (10) NOT NULL PRIMARY KEY,
  email char (10) NOT NULL UNIQUE
 );
```

Единственное ограничение, которое я оставил - это NOT NULL. Его желательно оставить, потому что первичные ключи не могут быть пустыми. Пускай база данных следит за этим, хотя она и так будет следить. То, что второе поле я сделал уникальным и ненулевым, так это просто для красоты.

А если нам нужно создать таблицу, где комбинация из двух полей должна давать уникальность? Это значит, что в двух строках не может быть одинаковых значений двух определённых полей, например:

```
Поле1 Поле2
  3     3
  4     2
  5     3
  4     2
```

Если мы поставим уникальность по обоим полям, то во второй и четвёртой строке будет ошибка, оба поля имею одно и то же значение. Первая и третья строка не даст ошибки, потому что у них разное Поле1, хотя и Поле2 совпадает. Так как же задать ограничение сразу по двум столбцам?

```
CREATE TABLE NewTable
 (
  Name char (10) NOT NULL,
  email char (10) NOT NULL,
  Phone char (10),
  UNIQUE  (Name, email)
 );
```

В этом примере я создаю таблицу с тремя полями и первые два из них являются уникальными. Точно так же ты можешь создать таблицу с первичным ключом из двух первых столбцов:

```
CREATE TABLE NewTable
 (
  Name char (10) NOT NULL,
  email char (10) NOT NULL,
  Phone char (10),
  PRIMARY KEY (Name, email)
 );
```

Помни, что большинство баз данных накладывает ограничение на первичный ключ - только первые и подряд идущие поля могут входить в первичный ключ.

Есть ещё одно интересное ключевое слово - CHECK, которое позволяет задать диапазон допустимых значений в поле, например:

```
CREATE TABLE NewTable
 (
  Name char (10) NOT NULL PRIMARY KEY,
  email char (10) NOT NULL UNIQUE,
  Age decimal CHECK (Age<110)
 );
```

В этом примере я добавил поле Age (Возраст) типа decimal. После этого я поставил ключевое слово *CHECK* , которое задаёт ограничение. После этого я в скобках указал это ограничение, что поле Age должно быть меньше 110 (Я не думаю, что кто-то проживёт больше 110 лет, поэтому ограничил от ошибки). Теперь я усложню пример, сделав двойную проверку:

```
CREATE TABLE NewTable
 (
  Name char (10) NOT NULL PRIMARY KEY,
  email char (10) NOT NULL UNIQUE,
  Age decimal CHECK (Age<110 and Age>0)
 );
```

В этом примере я сделал дополнительное ограничение. Теперь значение в поле Age должно быть больше 0 и меньше 100.

Есть ещё один тип ограничений - типа SQL оператора IN, который мы изучили уже давным давно.

```
CREATE TABLE NewTable
 (
  Name char (10) NOT NULL PRIMARY KEY,
  email char (10) NOT NULL UNIQUE,
  Age decimal CHECK (Age<110),
  Town char(15) CHECK (Town in ('Moscow', 'Piter', 'Brest'))
 );
```

Здесь мы используем знакомый оператор IN, который перечисляет допустимые значения для поля Town. Так можно использовать любые допустимые операторы сравнения SQL. Например, можно использовать знаки множественного выбора (маски) такие как _ (подчёркивание - заменяет одну любую букву) и % (процент - заменяет множество букв). Эти маски мы уже изучали, так что давай посмотрим пример:

```
CREATE TABLE NewTable
 (
  Name char (10) NOT NULL PRIMARY KEY,
  email char (10) NOT NULL UNIQUE,
  Age decimal CHECK (Age<110),
  Town char(15) CHECK (Town in ('Moscow', 'Piter', 'Brest')),
  DateB char(10) CHECK (DateB LIKE '__/__/____'),
  Password char(10) CHECK (DateB LIKE 'RTY%')
 );
```

Здесь я добавил два поля:

- DateB (Дата рождения) - любое значение этого поля должно удовлетворять маске __/__/____
- Password (Пароль) - пароль - слово, которое должно начинаться с трёх букв 'RTY'

Помимо ограничений, нам доступны для настройки и значения по умолчанию, за это отвечает слово DEFAULT:

```
CREATE TABLE NewTable
 (
  Name char (10) NOT NULL PRIMARY KEY,
  email char (10) DEFAULT='@mail.ru',
  Age decimal CHECK(Age<110),
  Town char (15) DEFAULT='Moscow'
 );
```

В этом примере я сразу двум полям задал значения по умолчанию.

masteram.us 2013