SimbirSoft

SQL

версия 3.1. от 05.07.2022 г.

ООО "СимбирСофт"
432071, г. Ульяновск,
Проспект Нариманова, 1, строение 2
Телефон:: +7 /8422/ 44-66-91
E-mail: info@simbirsoft.com
Web: www.simbirsoft.com

Содержание

Цели Блока

1. Изучить основы SQL
2. Ознакомиться с базовыми конструкциями и операторами SQL
3. Показать возможности использования SQL при работе QA
4. Научиться составлять базовые запросы, наиболее часто применяемые в деятельности QA.

Маркеры-иконки в блоке

Информация для запоминания

Информация для изучения

Информация для сведения

Введение

Одним из важнейших элементов любого программного продукта является такой компонент как база данных.

Каждый раз, когда мы что-то ищем в поисковике, используется база данных. При авторизации после ввода логина и пароля для входа на какой-нибудь сервис, они сравниваются со значениями, которые хранятся в базе данных этого сервиса. При загрузке страницы с динамическими данными используется база данных.

Таким образом, невозможно утверждать, соответствует ли наш продукт тем стандартам качества, к которым мы стремимся, без понимания, тестирования и использования базы данных.

Помимо проверки качества непосредственно самой базы, как элемента нашей системы, понимание и умение работать с ней здорово облегчает жизнь при тестировании всех остальных элементов.

Наиболее яркими примерами такого использования может являться настройка и создание клиентов под которыми мы проводим тестирование, установка необходимых параметров для запуска сценариев, создание и изменение тестовых данных и т.д.

Так как тема базы данных в общем и SQL, как средства общения с этой базой данных, в частности являются объемными по содержанию материала, в этом блоке мы рассмотрим основные базовые знания, наиболее часто используемые в работе QA.

Для начала давайте разберемся с терминами и понятиями:

Существует множество определений понятия «база данных», которые предлагаются как международными и национальными стандартами, так и различными монографиями.

Как пример можно привести определение из **ISO/IEC 2382:2015:**

1. База данных — совокупность данных, организованных в соответствии с концептуальной структурой, описывающей характеристики этих данных и взаимоотношения между ними, причём такое собрание данных, которое поддерживает одну или более областей применения.

2. Или можно проще описать базу данных как информационную модель, позволяющую упорядоченно хранить данные о группе объектов, обладающих одинаковым набором свойств.

3. Или еще проще - База данных — набор сведений, хранящихся некоторым упорядоченным способом.

Фактически единственной функцией базы данных является хранение данных.

Но если целью базы данных является простое хранение информации, то для управления и взаимодействия с базой данных используется уже система управления базами данных (СУБД).

Система управления базами данных – это совокупность языковых и программных средств, которая осуществляет доступ к данным, позволяет их создавать, менять и удалять, обеспечивает безопасность данных и т.д. В общем, СУБД – это система, позволяющая создавать базы данных и манипулировать сведениями из них.

Исходя из определения, функциями СУБД являются:

1. Определение структуры данных
2. Изменение данных
3. Получение данных
4. Администрирование

По характеру использования СУБД делят на однопользовательские (предназначенные для создания и использования БД на персональном компьютере) и многопользовательские (предназначенные для работы с единой БД нескольких компьютеров, объединенных в локальные сети).

Многопользовательские СУБД (MS SQL, Oracle, MySQL, Firebird и т.д.) различаются ценой, железом, производительностью, но при этом объединены тем, что используют одинаковый механизм – механизм запросов к базе данных SQL.

SQL – язык структурированных запросов, основной задачей которого является предоставление простого способа считывания и записи информации в базу данных.

В конкретной СУБД, язык SQL может иметь специфичную реализацию (свой диалект – в данном блоке изучение СУБД будет проходить на основе **MS SQL**).

Подмножества языка SQL:

- **Язык DDL** (Data Definition Language) служит для создания и модификации структуры БД, т.е. для создания/изменения/удаления таблиц и связей.

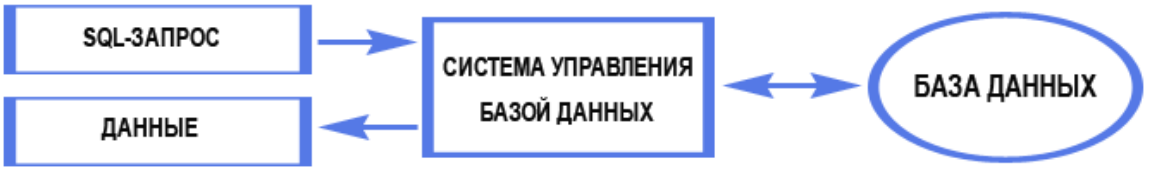
- **Язык DML** (Data Manipulation Language) позволяет осуществлять манипуляции с данными таблиц, т.е. с ее строками. Он позволяет делать выборку данных из таблиц, добавлять новые данные в таблицы, а также обновлять и удалять существующие данные.
- **Язык DCL** (Data Control Language) предназначен для осуществления административных операций, присваивающих или отменяющих право (привилегию) использовать базу данных, таблицы и другие объекты базы данных, а также выполнять те или иные операторы SQL.
- **Язык TCL (Transaction Control Language)** – используется для обработки транзакций.

В соответствии подмножеством языка SQL, операторы SQL делятся на:

- операторы определения данных (DDL):
  - CREATE создаёт объект базы данных (саму базу, таблицу, представление, пользователя и так далее),
  - ALTER изменяет объект,
  - DROP удаляет объект;
- операторы манипуляции данными (DML):
  - SELECT выбирает данные, удовлетворяющие заданным условиям,
  - INSERT добавляет новые данные,
  - UPDATE изменяет существующие данные,
  - DELETE удаляет данные;
- операторы определения доступа к данным (DCL):
  - GRANT предоставляет пользователю (группе) разрешения на определённые операции с объектом,
  - REVOKE отзывает ранее выданные разрешения,
  - DENY задаёт запрет, имеющий приоритет над разрешением;
- операторы управления транзакциями (TCL):
  - COMMIT применяет транзакцию,

- ROLLBACK откатывает все изменения, сделанные в контексте текущей транзакции,
- SAVEPOINT делит транзакцию на более мелкие участки.

Простейшей схемой работы с базой можно изобразить в следующем виде:



Структура Базы данных

Существуют множество видов классификаций баз данных – по модели данных, по среде постоянного хранения, по степени распределенности и т.д.

Наиболее распространенной является классификация по модели данных:

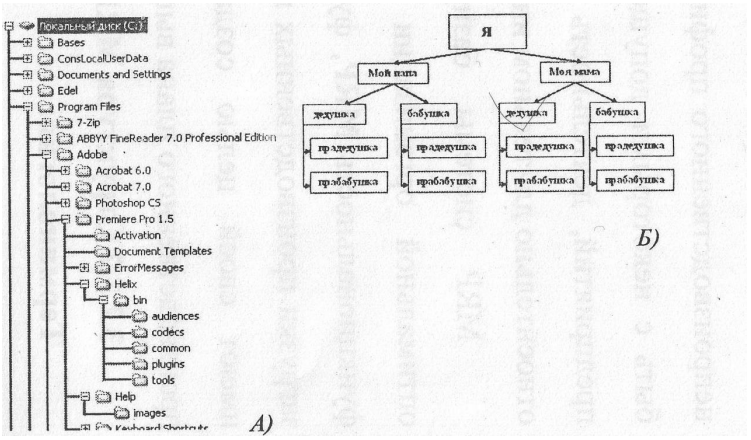
- Иерархическая
- Объектная и объектно-ориентированная
- Объектно-реляционная
- Реляционная
- Сетевая
- Функциональная.

Ознакомиться более подробно с ними можно по ссылкам, а в качестве примера рассмотрим такие как:

- Иерархическая
- Сетевая
- Реляционная

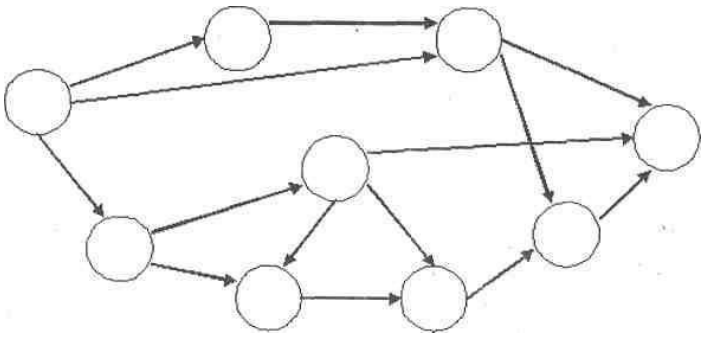
Иерархическая БД

В иерархической БД существует упорядоченность элементов в записи, один элемент считается главным, остальные — подчиненными. Данные в записи упорядочены в определенную последовательность, как ступеньки лестницы, и поиск данных может осуществляться лишь последовательным "спуском" со ступеньки на ступеньку. Поиск какого-либо элемента данных в такой системе может оказаться довольно трудоемким из-за необходимости последовательно проходить несколько предшествующих иерархических уровней. Принцип связи – “Один ко многим”.



Сетевая БД

Эта база данных отличается большей гибкостью, так как в ней существует возможность устанавливать дополнительно к вертикальным иерархическим связям горизонтальные связи. Это облегчает процесс поиска требуемых элементов данных, так как уже не требует обязательного прохождения всех предшествующих ступеней. Недостатком сетевой модели является сложность разработки серьезных приложений.



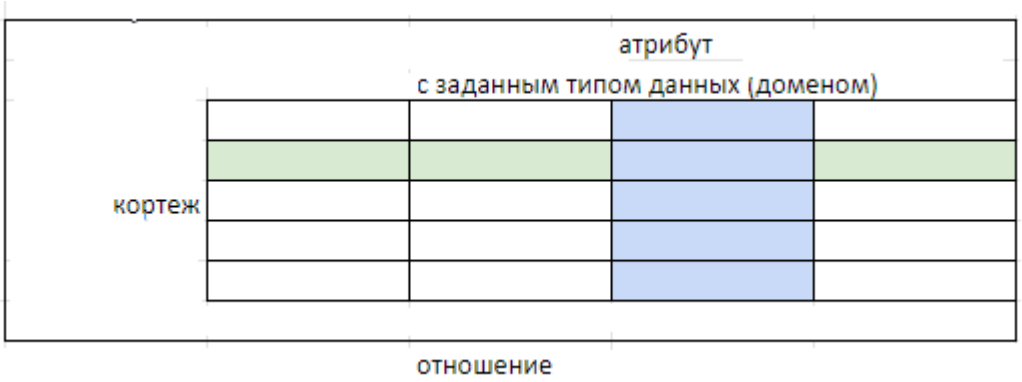
Реляционная БД

Наиболее распространенным способом организации данных является реляционная БД.

Реляционная база данных – это база данных, организованная по принципам реляционной модели данных.

Чтобы понять, что такое Реляционная база данных, можно представить обычную таблицу в Excel, в которой будут столбцы и строки. Вот совокупность нескольких таблиц и то, как они между собой взаимодействуют, и будет условной реляционной базой данных.

Relation (отношение), математ. термин – набор кортежей, каждый элемент в котором является членом определенного домена данных.



Где под кортежем понимается строка, каждый элемент является столбцом, т. е. можно считать синонимами термины таблица и отношение; строка и кортеж; столбец, атрибут и поле.

Все элементы в столбце имеют одинаковый тип (числовой, символьный), а каждый столбец — неповторяющееся имя. Одинаковые строки в таблице отсутствуют.

Преимущество таких БД — наглядность и понятность организации данных, скорость поиска нужной информации. Примером реляционной БД служит таблица на странице классного журнала, в которой записью является строка с данными о конкретном ученике, а имена полей (столбцов) указывают, какие данные о каждом ученике должны быть записаны в ячейках таблицы.

Так как порядка 90% от всех используемых баз данных являются реляционными базами данных, то дальнейшее описание будет в первую очередь касаться именно них.

Чуть выше кратко упоминались основные свойства таблиц, но для лучшего понимания повторим основные:

1. Данные внутри одного столбца имеют одинаковый тип
2. Каждый столбец имеет уникальное имя
3. Столбцы не зависят друг от друга
4. Порядок столбцов не важен
5. Поля содержат только данные
6. Строка должна быть уникальной
7. Строки не зависят друг от друга
8. Порядок строк не важен

Таким образом, при создании таблицы желательно, чтобы она имела уникальный столбец или же совокупность столбцов, которая уникальна для каждой ее строки – по данному уникальному значению можно однозначно идентифицировать запись. Такое значение называется **первичным ключом** таблицы (*primary key*).

Если говорить научным языком, ***Первичный ключ*** *– это подмножество атрибутов отношений, удовлетворяющих требованиям уникальности и минимальности.*

Где под уникальностью понимается отсутствие строк с одинаковым значением первичного ключа, а под минимальностью – такое состояние, при котором первичный ключ нельзя разбить на меньшее количество атрибутов (столбцов) без нарушения его уникальности.

Таким образом, первичный ключ является обязательным элементом каждой таблицы в базе данных. Именно по первичному ключу есть возможность идентифицировать нужную нам строку и обратиться к ней.

Помимо возможности идентификации строк, на основе первичного ключа строится взаимодействие между таблицами в базе данных. Для этого во второй таблице, которая должна обладать связью с первой таблицей, создается столбец идентичный первичному ключу из первой таблицы. Данный столбец уже не может в полном объеме применяться для определения уникальности строк во второй таблице (то есть не является первичным ключом), но с помощью него устанавливается связи между 1 и 2 таблицей. Такой столбец получил название - **внешний ключ** (*foreign key*)

***Внешний ключ SQL** — это ключ, используемый для объединения двух таблиц. Иногда его также называют ссылочным ключом. Внешний ключ — это столбец или комбинация столбцов, значения которых соответствуют Первичному ключу в другой таблице.*

Типы данных

Указанная в этой части информация содержит цель ознакомить с наиболее часто используемыми форматами в MS SQL.

У каждого столбца, локальной переменной, выражения и параметра есть определенный тип данных.

Тип данных — атрибут, определяющий, какого рода данные могут храниться в объекте: целые числа, символы, данные денежного типа, метки времени и даты, двоичные строки и так далее.

При объединении одним оператором двух выражений с разными типами данных, параметрами сортировки, точностями, масштабами или длинами, результат определяется следующим образом:

- Тип данных результата определяется применением правил очередности типов данных к входным выражениям. Дополнительные сведения.
- Параметры сортировки результата определяются правилами очередности параметров сортировки, если тип данных результата — char, varchar, text, nchar, nvarchar или ntext. Дополнительные сведения
- Точность, масштаб и длина результата зависят от точности, масштаба и длины входных выражений. Дополнительные сведения

Наиболее часто используемые параметры:

Значение	Обозначение в MS SQL	Описание

Строка переменной длины	varchar(N) и nvarchar(N)	При помощи числа N, мы можем указать максимально возможную длину строки для соответствующего столбца. Например, если мы хотим сказать, что значение столбца «ФИО» может содержать максимум 30 символов, то необходимо задать ей тип nvarchar(30). Отличие varchar от nvarchar заключается в том, что varchar позволяет хранить строки в формате ASCII, где один символ занимает 1 байт, а nvarchar хранит строки в формате Unicode, где каждый символ занимает 2 байта. Тип varchar стоит использовать только в том случае, если вы на 100% уверены, что в данном поле не потребуется хранить Unicode символы. Например, varchar можно использовать для хранения адресов электронной почты, т.к. они обычно содержат только ASCII символы.
Строка фиксированной длины	char(N) и nchar(N)	От строки переменной длины данный тип отличается тем, что если длина строки меньше N символов, то она всегда дополняется справа до длины N пробелами и сохраняется в БД в таком виде, т.е. в базе данных она занимает ровно N символов (где один символ занимает 1 байт для char и 2 байта для типа nchar). На моей практике данный тип очень редко находит применение, а если и используется, то он используется в основном в формате char(1), т.е. когда поле определяется одним символом.
Целое число	int	Данный тип позволяет нам использовать в столбце только целые числа, как положительные, так и отрицательные. Для справки (сейчас это не так актуально для нас) – диапазон чисел который позволяет тип int от -2 147 483 648 до 2 147 483 647. Обычно это основной тип, который используется для задания идентификаторов.
Вещественное или действительное число	float	Если говорить простым языком, то это числа, в которых может присутствовать десятичная точка (запятая).
Дата	date	Если в столбце необходимо хранить только Дату, которая состоит из трех составляющих: Числа, Месяца и Года. Например, 15.02.2014 (15 февраля 2014 года). Данный тип можно использовать для столбца «Дата приема», «Дата рождения» и т.п., т.е. в тех случаях, когда нам важно зафиксировать только дату, или, когда составляющая времени нам не важна и ее можно отбросить или если она не известна.
Время	time	Данный тип можно использовать, если в столбце необходимо хранить только данные о времени, т.е. Часы, Минуты, Секунды и Миллисекунды. Например, 17:38:31.3231603 Например, ежедневное «Время отправления рейса».

Дата и время	datetime	Данный тип позволяет одновременно сохранить и Дату, и Время. Например, 15.02.2014 17:38:31.323 Для примера это может быть дата и время какого-нибудь события.
Флаг	bit	Данный тип удобно применять для хранения значений вида «Да»/«Нет», где «Да» будет сохраняться как 1, а «Нет» будет сохраняться как 0.

При работе с базой данных ***обязательно*** следует учитывать какой тип данных применяется:

- Необходимо понимать, чем больше диапазон значений у типа данных, тем больше памяти он занимает. Например, если значения в столбце не будут превышать 30 символов, не нужно указывать тип данных для 1000 символов. Правильный выбор типа данных позволяет сэкономить место для хранения этих данных.
- Также указание неверного типа данных может привести к возникновению дефектов, например, нужна десятичная точность на 2-3 пункта в столбце, в котором хранится общая стоимость заказа. Если назначить этот столбец как целое число, база данных сможет хранить только целые числа без десятичных значений. Что произойдет с десятичными знаками, зависит от платформы базы данных – автоматически обрезать значения или выдать ошибку. Любая альтернатива может создать серьезную ошибку в приложении. Всегда нужно учитывать то, что нужно хранить при разработке таблиц.

Операторы манипуляции данными (DML)

В данной части блока рассмотрим теоретическую и практическую часть, связанную с манипуляциями (просмотр, добавление,изменение, удаление) данных в таблицах.

Для дальнейшего изучения необходимо настроить базу данных **MS SQL** 2017 в соответствии с инструкцией - ссылка на инструкцию по установке базы данных.

Обучение дальше будет продолжаться в режиме Step by Step, т.е. при чтении нужно сразу же своими руками пытаться выполнить пример. После делаете анализ полученного результата и пытаетесь понять его интуитивно.

Язык DML содержит следующие конструкции:

- SELECT – выборка данных

- INSERT – вставка новых данных
- UPDATE – обновление данных
- DELETE – удаление данных
- MERGE – слияние данных

SELECT – общая конструкция

Пожалуй, можно сказать, что конструкция SELECT является самой главной конструкцией языка DML, т.к. за счет нее или ее частей осуществляется выборка необходимых данных из БД.

SELECT *- возвращает строки из базы данных и позволяет делать выборку одной или нескольких строк или столбцов из одной или нескольких таблиц.*

Полный синтаксис инструкции SELECT сложен, однако основные предложения можно вкратце описать следующим образом.

Базовый синтаксис команды:

```
SELECT [DISTINCT] список_столбцов или *
FROM источник
WHERE фильтр
ORDER BY выражение_сортировки
```

Начнем с самой элементарной формы SELECT:

```
SELECT *
  FROM [AdventureWorks2017].[HumanResources].[Department]
```

В данном запросе мы просим вернуть все столбцы (на это указывает «*») из таблицы [HumanResources].[Department] – можно прочесть это как «ВЫБЕРИ все_поля ИЗ таблицы_департамент».

Значение [AdventureWorks2017].[HumanResources].[Department] - *означает имя_базы.имя_схемы.таблица.*

Такое уточнение бывает полезным, например, если:

- в одном запросе мы обращаемся к объектам, расположенным в разных схемах или базах данных
- требуется сделать перенос данных из одной схемы или БД в другую
- находясь в одной БД, требуется запросить данные из другой БД

Схема – очень удобное средство, которое полезно использовать при разработке архитектуры БД, а особенно крупных БД.

В нашем случае, если база данных уже была предварительно выбрана, то это значение базы данных можно опустить, оставив только имя схемы и таблицы.

Также можно опустить [], они являются необязательной конструкцией. То есть запрос может быть представлен в виде:

```
SELECT  *
 FROM HumanResources.Department
```

Если столбцов в таблице очень много, то предпочтительней будет выборка с непосредственным перечислением необходимых вам полей через запятую:

```
SELECT
DepartmentID
 ,Name
 FROM HumanResources.Department
```

Пояснение к форме Select и логике расчета данных

Непосредственно в MS SQL (в отличие, например, от БД ORACLE) Select может не содержать блока FROM, в этом случае вы можете использовать ее для получения каких-то значений:

```
SELECT
3570/100*15,
SYSDATETIME(),  -- получение системной даты БД
```

Обратите внимание на то, что расчет 3570/100*15 выдал результат 525, хотя если посчитать эти операции на калькуляторе, то значение получится – 535,5.

Такой результат связан с тем, что сработало целочисленное деление 3570/100 – так как в выражении все числа целые, поэтому и результат – целое число.

Нужно запомнить, что в MS SQL работает следующая логика:

- Целое / Целое = Целое
- Вещественное / Целое = Вещественное
- Целое / Вещественное = Вещественное

Выполните запрос ниже и сравните результат:

```
 SELECT
3570/100
, 3570./100
, 3570/100.
```

Поэтому необходимо обращать внимание на тип данных числовых столбцов. В том случае если он целый, а результат вам нужно получить вещественный, то нужно использовать преобразование, либо просто ставьте точку после числа указанного в виде константы.

О применение знаков комментария в запросе

В тексте запроса мы можем использовать как однострочные «-- …», так и многострочные «/* … */» комментарии. Если запрос большой и сложный, то комментарии могут очень помочь вам или кому-то другому через некоторое время вспомнить или разобраться в его структуре.

Кроме того, знаки комментария позволяют отсечь те данные, которые нам в данный момент не нужны, но их требуется сохранить для дальнейшего использования:

```
SELECT
DepartmentID
-- ,Name
 FROM HumanResources.Department
```

Задание псевдонимов(Alias) в таблице

При перечислении колонок их можно предварять именем таблицы, находящейся в блоке FROM:

```
SELECT
HumanResources.Department.DepartmentID
, HumanResources.Department.Name
 FROM HumanResources.Department
```

Но такой синтаксис обычно использовать неудобно, т.к. имя таблицы может быть длинным. Для этих целей обычно задаются и применяются более короткие имена – псевдонимы (alias):

```
SELECT
dep.DepartmentID
, dep.Name
 FROM HumanResources.Department as dep
```

Или опустив ключевое слово AS:

```
SELECT
dep.DepartmentID
, dep.Name
 FROM HumanResources.Department  dep
```

Конечно, при создании запроса к одной таблице, задание псевдонима не обязательно, да и не нужно, но при создании запроса сразу к нескольким таблицам (рассмотрим далее) здорово облегчает жизнь.

Кроме того, псевдонимы (alias) можно использовать и для вывода столбцов, например:

```
SELECT
dep.DepartmentID as "1"
```

```
, dep.Name as "2"
 FROM HumanResources.Department  dep
```

DISTINCT – отброс строк дубликатов

Следует отметить, что вертикальная выборка может содержать дубликаты строк в том случае, если она не содержит потенциального ключа, однозначно определяющего запись, например:

```
SELECT
GroupName
 FROM HumanResources.Department
```

Если требуется получить только уникальные строки, то можно использовать ключевое слово **DISTINCT**:

```
SELECT  distinct
GroupName
 FROM HumanResources.Department
```

То есть, по результатам запросов мы получили список “подразделений”, в которые входят все департаменты без их задвоения.

Конкатенация - объединение значений в один результат

Существует несколько вариантов объединения строк в одну. Для понимания того, как это происходит выполните запрос ниже:

```
Select
  FirstName
  ,MiddleName
  ,LastName
  ,FirstName+' '+MiddleName+' '+LastName as ФИО1
  ,ISNULL(FirstName,'')+' '+ISNULL(MiddleName,'')+' '+ISNULL(LastName,'') as ФИО2
  ,CONCAT (FirstName,' ',MiddleName,' ',LastName) as ФИО3
    FROM Person.Person
```

Для соединения (сложения, конкатенации) строк в MS SQL используется символ «+».

Обратите внимание, т.к. у некоторых сотрудников не указано отчество MiddleName (NULL значение), то результат выражения «FirstName+' '+MiddleName+' '+LastName» также вернул нам NULL.То есть необходимо запомнить, что все

выражения в которых участвует NULL (например, деление на NULL, сложение с NULL) будут возвращать NULL.

Функция ISNULL позволяет отследить выражение NULL и заменить его на то значение которое вы выберете. Так запись ISNULL(MiddleName,'') буквально звучит как “если значение в строке равно NULL, то заменить его на “” (пустое значение), иначе указываем значение”. Попробуйте в запросе заменить ISNULL(MiddleName,'') на ISNULL(MiddleName,'qwerty').Функция ISNULL применяется не только в вышеуказанном примере, ее применение полезно во всех случаях когда в поле может присутствовать значение NULL.

Функция CONCAT возвращает строку, возникающую в результате объединения двух или более строковых значений в сквозной форме, при этом аргументы со всеми значениями NULL она возвращает как пустую строку типа varchar(1).То есть в данном конкретном примере делает то же самое, что и объединение по «+» с функцией ISNULL, таким образом являясь предпочтительным вариантом объединения строк.

Основные арифметические операторы SQL

Оператор	Действие
+	Сложение (x+y) или унарный плюс (+x)
-	Вычитание (x-y) или унарный минус (-x)
*	Умножение (x*y)
/	Деление (x/y)
%	Остаток от деления (x%y). Для примера 15%10 даст 5

Приоритет выполнения арифметических операторов такой же, как и в математике. Если необходимо, то порядок применения операторов можно изменить используя круглые скобки ().

Еще раз - любая операция с NULL дает NULL, например: 10+NULL, NULL*10/3, 50/NULL – все это даст в результате NULL. Это необходимо учитывать при составлении запроса и при необходимости, делайте обработку NULL значений функциями ISNULL или другими подходящими, например - CASE, COALESCE.

Сортировка результата запроса

Чтобы упорядочить строки результирующего набора, можно выполнить сортировку по любому количеству полей, указанных в предложении **SELECT или присутствовавших на выходе предложения FROM**. Для этого используется

предложение **ORDER BY**, являющийся всегда последним предложением в операторе **SELECT**.

Сравним результат 2 запросов:

1.
```
SELECT
DepartmentID
 ,Name
 FROM HumanResources.Department
```

2.
```
SELECT
DepartmentID
 ,Name
 FROM HumanResources.Department
 order by DepartmentID    -- упорядочить результат по 1 столбцу DepartmentID
```

После имени столбца в предложении ORDER BY можно задать опцию ASC, которая служит для сортировки этого поля в порядке возрастания или DESC, которая служит для сортировки этого поля в порядке убывания. Но так как указание order by без опций сортирует в порядке возрастания, то опция ASC практически не применяется.

```
SELECT
DepartmentID
 ,Name
 FROM HumanResources.Department
 order by DepartmentID desc    -- упорядочить результат по 1 столбцу DepartmentID в порядке убывания.
```

Стоит отметить еще раз, что в предложении ORDER BY можно использовать и поля, которые не перечислены в предложении SELECT, но **присутствуют на выходе предложения FROM** (кроме случая, когда используется DISTINCT). Выполните следующие запросы:

1.
```
SELECT  *
 FROM HumanResources.Department  -- определяет какие поля присутствуют на выходе предложения FROM
```

2.
```
SELECT
DepartmentID
 ,Name
 FROM HumanResources.Department
 order by GroupName        -- упорядочить результат по 1 столбцу GroupName отсутствующему в предложении SELECT.
```

3.

```
SELECT
DepartmentID
 ,Name
,GroupName
 FROM HumanResources.Department
  order by GroupName        -- упорядочить результат по 1 столбцу GroupName
```
присутствующему в предложении SELECT.

Подобная сортировка по столбцу, который не перечислен в предложении SELECT оправдан, когда необходимо отобрать результаты, не показывая критерий сортировки - например, отобрать сотрудников у которых самая высокая ЗП, с учетом того, что саму ЗП в целях конфиденциальности показывать нельзя.

При этом если вы использовали DISTINCT, то невозможно осуществить сортировку по столбцу отсутствующему в предложении SELECT:

```
SELECT  DISTINCT
DepartmentID
 ,Name
 FROM HumanResources.Department
  order by GroupName        -- упорядочить результат по 1 столбцу GroupName
```
отсутствующему в предложении SELECT.

Кроме того, при использовании псевдонимов (Alias), можно указывать не только наименование столбца, но и его псевдоним:

```
SELECT
DepartmentID as ID
 ,Name
 FROM HumanResources.Department
 order by ID    -- упорядочить результат по 1 столбцу DepartmentID
```

Также при сортировке можно использовать порядковые номера столбцов, *но лучше забыть и никогда не использовать такой вариант сортировки, так как если кто-то, поменяет в таблице порядок столбцов, или удалит столбцы, запрос может также работать, но уже неправильно. Сортировка уже может идти по другим столбцам и это коварно тем, что данная ошибка может обнаружиться очень нескоро.*

Напоследок упомяну, что нет ограничения по скольки столбцам будет осуществляться сортировка - `order by DepartmentID, Name` - будет означать, что сначала результат сортируется по колонке `DepartmentID`, а затем по колонке `Name`.

Ограничение числа строк при выводе результата

Оператор TOP – ограничивает число строк, возвращаемых в результирующем наборе запроса до заданного числа или процентного значения.

Если предложение TOP используется совместно с предложением ORDER BY, то результирующий набор ограничен первыми N строками отсортированного результата. В противном случае возвращаются первые N строк в неопределенном порядке.

1.
```
SELECT  TOP 3
DepartmentID
 ,Name
 FROM HumanResources.Department
```

2.
```
SELECT  TOP 3
DepartmentID
 ,Name
 FROM HumanResources.Department
order by  DepartmentID
```

Помимо указания количества строк, можно указать каким процентом от выведенных строк будет ограничена выборка. Достигается это добавлением слова PERCENT:

```
SELECT  TOP 3 percent
DepartmentID
 ,Name
 FROM HumanResources.Department
order by  DepartmentID
```

Хочу обратить внимание, что ограничение вывода строк может привести к ошибкам, если присутствует неоднозначная сортировка, например, при сортировке на 4 и 5 позиции оказались одинаковые значения. И если мы ограничим выборку 4 наибольшими значениями - TOP 4, то хоть значение по 5 строке и будет равно 4 позиции, но в выборку не попадет.

Для исключения подобных ситуаций необходимо использовать опцию WITH TIES:

1.
```
-- проверяем какие значения будут при сортировки по полю GroupName
SELECT
DepartmentID
 ,Name
,GroupName
 FROM HumanResources.Department
order by  GroupName
```

2. -- ограничиваем выборку первыми 3 строками - 2 значения из группы Executive General and Administration не попадают в выборку

```
SELECT  TOP 3
DepartmentID
 ,Name
,GroupName
 FROM HumanResources.Department
order by  GroupName
```

3. -- добавляем опцию WITH TIES

```
SELECT  TOP 3 WITH TIES
DepartmentID
 ,Name
,GroupName
 FROM HumanResources.Department
order by  GroupName
```

WHERE – условие выборки строк

Помимо создания запросов с простой выборкой данных, примеры которых были рассмотрены выше, существует возможность задавать параметры выборки для фильтрации записей по заданному условию. Такой возможностью нас обеспечивает предложение WHERE.

Например, нам нужно узнать департаменты, входящие в группу - Sales and Marketing.

```
SELECT
Name, *
 FROM HumanResources.Department  as Dep
WHERE  GroupName='Sales and Marketing'
order by  Dep.Name --Desc
```

Прежде чем читать дальше попробуйте сами изменить запрос таким образом, чтобы выбирались только столбцы DepartmentID, Name и ModifiedDate, входящие в группу Sales and Marketing, при этом необходимо отсортировать значения по убыванию и вывести уникальное значение с выводом только первой записи.

Теперь, когда у вас получилось изменить запрос, можно сказать что WHERE является третьим волшебным словом наиболее часто используемым при создании запросов наряду с SELECT, FROM.

Предложение WHERE пишется до команды ORDER BY.

Порядок применения команд к исходному набору следующий:

1. FROM - идет отбор значения из таблицы Department

```
SELECT
```

```
DepartmentID
,Name
,ModifiedDate
FROM HumanResources.Department  as Dep
```

2. WHERE - если указано, то первым делом из всего набора идет отбор только удовлетворяющих условию записей

```
SELECT
DepartmentID
,Name
,ModifiedDate
FROM HumanResources.Department  as Dep
WHERE  GroupName='Sales and Marketing'
```

3. DISTINCT – если указано, то отбрасываются все дубликаты

```
SELECT  distinct
DepartmentID
,Name
,ModifiedDate
FROM HumanResources.Department  as Dep
WHERE  GroupName='Sales and Marketing'
```

4. ORDER BY – если указано, то делается сортировка результата

```
SELECT  distinct
DepartmentID
,Name
,ModifiedDate
FROM HumanResources.Department  as Dep
WHERE  GroupName='Sales and Marketing'
order by  Dep.Name Desc
```

5. TOP – если указано, то из отсортированного результата возвращается только указанное число записей

```
SELECT  distinct top 1
DepartmentID
,Name
,ModifiedDate
FROM HumanResources.Department  as Dep
WHERE  GroupName='Sales and Marketing'
order by  Dep.Name Desc
```

В результате у нас получается одна строка со значением:

DepartmentID	Name	ModifiedDate
3	Sales	2008-04-30 00:00:00.000

Булевы операторы и простые операторы сравнения

Для последующих примеров нам понадобится другая таблица Product. Запрос на выборку всех данных в таблице будет выглядеть так:

```
SELECT *
FROM Production.Product
```

В этой таблице нам будет необходимо выбрать строки по следующим параметрам:

1. ProductID должен быть меньше 350 и цвет продукта- Color должен быть черным - 'Black'
2. ProductID должен быть меньше 550 и цвет продукта- Color должен быть серебряным - 'Silver'
3. ProductID должен быть больше 750 и цвет продукта- Color не должен быть указан - NULL

При выполнении всех этих операций мы должны будем использовать Булевы операторы, простые операторы сравнения а также операторы для проверки значения/выражения на NULL. На теоретических моментах по всем этим операторам мы остановимся чуть позднее, а пока нам просто необходимо создать запрос выбирающий по всем условиям.

Для этого мы можем пойти, например, следующим путем - написав 3 запроса с использованием булева оператора - **AND,** простых операторов сравнения - =, < , > и оператора **IS NULL.**

Запросы будут выглядеть так:

1.

```
SELECT *
FROM Production.Product
Where ProductID<350 and Color='Black'   -- выбираем значение у которого ID меньшее (<) 350 И (AND) цвет равный (=) черному
```

2.

```
SELECT *
FROM Production.Product
Where ProductID<550 and Color='Silver'   -- выбираем значение у которого ID меньшее (<) 550 И (AND) цвет равный (=) серебряному
```

3.

```
SELECT *
FROM Production.Product
Where ProductID>750 and Color is NULL   -- выбираем значение у которого ID больше (>) 750 И (AND) цвет не указан (IS NULL)
```

Но для того чтобы не выполнять каждый из этих запросов по отдельности и потом копипастом объединять результаты - мы добавлением еще одного булево оператора ИЛИ - **OR,** перепишем запросы объединяя все условия в одно:

```
SELECT *
FROM Production.Product
Where ProductID<350 and Color='Black'
or ProductID<550 and Color='Silver'
or ProductID>750 and Color is NULL
```

Тут хочу обратить внимание, если вы сомневаетесь, как отработает запрос с учетом группировки операторов сравнения - не стесняйтесь пользоваться скобками:

```
SELECT *
FROM Production.Product
Where (ProductID<350 and Color='Black')
or (ProductID<550 and Color='Silver')
or (ProductID>750 and Color is NULL)
```

результат будет тот же, зато вы сами будете лучше ориентироваться в своем запросе.

Булевых операторов в языке SQL всего 3 – AND, OR и NOT:

AND логическое И. Ставится между двумя условиями (условие1 AND условие2). Чтобы выражение вернуло True, нужно, чтобы истинными были оба условия.

То есть если хотя бы одно из условий не будет истинным, значение не попадет в результат. Пример - выполните 2 запроса:

1.
```
SELECT *
FROM Production.Product
Where ProductID<5 and Color='Black'
```
2.
```
SELECT *
FROM Production.Product
Where ProductID<5
```

OR логическое ИЛИ. Ставится между двумя условиями (условие1 OR условие2). Чтобы выражение вернуло True, достаточно, чтобы истинным было только одно условие.

Пример:
```
SELECT *
FROM Production.Product
Where ProductID<5 or Color='Black'
```

NOT инвертирует условие/логическое_выражение. Накладывается на другое выражение (NOT логическое_выражение) и возвращает True, если логическое_выражение = False и возвращает False, если логическое_выражение = True

Пример:

```
SELECT *
FROM Production.Product
Where NOT(ProductID<323 and Color='Black')
```

В указанном примере Выражение в скобках «(ProductID<323 and Color='Black')» проверяет, что есть продукты с ID меньше 323 и черным цветом, а NOT инвертирует это значение, т.е. говорит «верни все продукты за исключением тех, у которых ID меньше 323 и цвет черный».

Для каждого булева оператора можно привести таблицы истинности, где дополнительно показано какой будет результат, когда условия могут быть равны NULL:

		Условие 1		
	AND	**TRUE**	**FALSE**	**NULL**
Условие 2	**TRUE**	TRUE	FALSE	NULL
	FALSE	FALSE	FALSE	FALSE
	NULL	NULL	FALSE	NULL

		Условие 1		
	OR	**TRUE**	**FALSE**	**NULL**
Условие 2	**TRUE**	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	NULL
	NULL	TRUE	NULL	NULL

	Условие		
NOT	**TRUE**	**FALSE**	**NULL**
-	FALSE	TRUE	NULL

Есть следующие простые операторы сравнения, которые используются для формирования условий:

Условие	Значение
=	Равно
<	Меньше
>	Больше
<=	Меньше или равно

>=	Больше или равно
<> !=	Не равно

Плюс имеются 2 оператора для проверки значения/выражения на NULL:

IS NULL Проверка на равенство NULL

IS NOT NULL Проверка на неравенство NULL

Примеры:

1.
```
SELECT *
FROM Production.Product
Where ProductID<323
```
2.
```
SELECT *
FROM Production.Product
Where ProductID<323 and Color IS NULL
```
3.
```
SELECT *
FROM Production.Product
Where ProductID<323 and Color IS NOT NULL
```

Приоритет: 1) Все операторы сравнения; 2) NOT; 3) AND; 4) OR.

При построении сложных логических выражений используются круглые скобки:

((условие1 AND условие2) OR NOT(условие3 AND условие4 AND условие5)) OR (…)

Также при помощи использования круглых скобок можно изменить стандартную последовательность вычислений.

BETWEEN – проверка на вхождение в диапазон

Помимо операторов AND, OR и NOT существуют и иные логические операторы. Ниже поговорим о наиболее часто всего применяемых для работы с массивами данных BETWEEN, IN и LIKE.

Представим пример когда из таблицы Product нам необходимо выбрать товары ID с 500 по 710. Это можно выполнить с помощью операторов сравнения:

```
SELECT *
FROM Production.Product
Where ProductID>=500 and ProductID<=710
```

или

```
SELECT *
FROM Production.Product
Where ProductID between 500 and 710  -- диапазон с 500 по 710
```

то есть BETWEEN это упрощенная запись вида ProductID>=500 and ProductID<=710.

Синтаксис оператора BETWEEN выглядит следующим образом:
проверяемое_значение [NOT] BETWEEN начальное_ значение AND конечное_ значение

Перед словом BETWEEN может использоваться слово NOT, которое будет осуществлять проверку значения на не вхождение в указанный диапазон:

```
SELECT *
FROM Production.Product
Where ProductID not between 500 and 710 -- в этом случае мы выбираем все значение которые не входят в диапазон с 500 по 710
```

IN – проверка на вхождение в перечень значений

Теперь рассмотрим пример когда нам необходимо выбрать строки только по определенным значениям, например - 1, 4, 329.

Это можно выполнить с помощью простого оператора OR:

```
SELECT *
FROM Production.Product
Where ProductID=1 OR ProductID=4 OR ProductID=329
```

Если у вас возник вопрос почему тоже самое не сделать через выборку с помощью оператора AND - выбрать значения где ProductID равен 1 и 4 и 329, попробуйте выполнить запрос:

```
SELECT *
```

```
FROM Production.Product
Where ProductID=1 AND ProductID=4 AND ProductID=329 -- и вспомнить, что
```

значения с оператором AND должны выполняться сразу все условия - ProductID должен быть равен и 1, и одновременно 4, и одновременно 329. Что в нормализованной таблице в принципе быть не может.

Однако возвращаясь к нашему примеру с оператором OR - запрос рабочий и в принципе нам ничего больше не нужно. Но представим ситуацию когда нам нужно указать не 3 числа, а допустим 7 или 10 или 1000 значений. И каждый раз писать ProductID=1 OR не очень приятно. Тут нам на помощь приходит оператор IN. Этот оператор имеет следующий вид:

проверяемое_значение [NOT] IN (значение1, значение2, …)

То есть наш запрос можно переписать следующим образом:

```
SELECT *
FROM Production.Product
Where ProductID IN (1,4,329)  -- сравните с количеством знаков в строчке
```

ProductID=1 OR ProductID=4 OR ProductID=329

В случае использования NOT - получим все те продукты, чей ID не равен 1.4.329:

```
SELECT *
FROM Production.Product
Where ProductID NOT IN (1,4,329)
```

Теперь чуть усложним задачу - нам необходимо выбрать все товары с цветом Black. White. Yellow. Silver, а также те товары у которых нет цвета.

Для этого мы также воспользуемся запросом с использованием оператора IN:

```
SELECT *
FROM Production.Product
Where Color IN ('Black','White','Yellow','Silver',NULL) -- NULL записи не войдут в
```

результат

Это связано с тем, что искать NULL значения при помощи конструкции IN не получится, т.к. проверка NULL=NULL вернет так же NULL, а не True. В этом случае необходимо разбить проверку на несколько условий:

```
SELECT *
FROM Production.Product
Where Color IN ('Black','White','Yellow','Silver')
        OR Color IS NULL
```

Есть еще более коварная ошибка, связанная с NULL, которую можно допустить при использовании конструкции NOT IN:

```
SELECT *
FROM Production.Product
Where Color NOT IN ('Black','White','Yellow','Silver',NULL) -- по логике из предыдущего примера должен выдать результат - иные цвета. кроме вышеперечисленных. а также NULL записи.
```

Однако запрос нам выдал пустой результат. Это связано с тем, что конструкция Color NOT IN ('Black','White','Yellow','Silver',NULL) раскладывается следующим образом:

```
SELECT *
FROM Production.Product
Where Color <>'Black'
AND Color <>'White'
AND Color <>'Yellow'
AND Color <>'Silver'
AND Color <>NULL -- проблема из-за этой проверки на NULL - это условие всегда вернет NULL
```

Крайнее правое условие (Color <>NULL) нам всегда даст неопределенность, т.е. NULL. Теперь вспомним таблицу истинности для оператора AND, где (TRUE AND NULL) дает NULL.

При выполнении левых условий, из-за неопределенного крайнего правого условия, в результате мы получим неопределенное значение всего выражения, поэтому строка не войдет в результат.

Для того, чтобы получить нужный нам результат, надо переписать запрос следующим образом:

```
SELECT *
FROM Production.Product
Where Color NOT IN ('Black','White','Yellow','Silver')
        AND Color IS NOT NULL
```

LIKE – проверка строки по шаблону

Сейчас в самом общем виде рассмотрим один очень полезный оператор, используемый для поиск каких либо строк - оператор LIKE.

Этот оператор имеет следующий вид:

проверяемая_строка [NOT] LIKE строка_шаблон [ESCAPE отменяющий_символ]

Рассмотрим 2 наиболее часто применяемых специальных символа в «строке_шаблон» (с полный перечнем можно ознакомиться тут):

1. Знак подчеркивания «_» — говорит, что на его месте может стоять любой единичный символ
2. Знак процента «%» — говорит, что на его месте может стоять сколько угодно символов, в том числе и ни одного

Давайте рассмотри на примере как это выглядит:

```
SELECT *
FROM Production.Product
Where Name Like '%Road%'-- у кого наименование содержит сочетание "Road"

SELECT *
FROM Production.Product
Where Name Like '%Caps'- у кого наименование оканчивается на "Caps"

SELECT *
FROM Production.Product
Where Name Like 'Flat%'-- у кого наименование начинается с букв "Flat"
```

Примеры с символом «_»:

```
SELECT *
FROM Production.Product
Where Name Like '_____s'- у кого наименование состоит из 6 букв и оканчивается на "s"

SELECT *
FROM Production.Product
Where Name Like '_lat%'- у кого наименование начинается с любого символа потом идут символы "lat" и потом наименование может продолжаться.
```

Наиболее часто применяется именно поиск со значением «%» - особенно это нужно чтобы найти какие логи или ключи - например найти значение по части известного ключа:

```
SELECT *
FROM Production.Product
Where rowguid Like '%8151%'
```

В заключение об операторе LIKE упомяну, что изредка специально для поиска значений в которых содержится знак процента или знак подчеркивания используется

ключевое слово ESCAPE. Оно задает отменяющий символ, который отменяет проверяющее действие специальных символов «_» и «%».

```
LIKE '%!%%' ESCAPE '!' -- строка содержит знак "%"
LIKE '%!_%' ESCAPE '!' -- строка содержит знак "_"
```

Поиск по дате

При проверке на дату можно использовать как и со строками одинарные кавычки '...'.

Вне зависимости от региональных настроек в MS SQL можно использовать следующий синтаксис дат 'YYYYMMDD' (год, месяц, день слитно без пробелов). Такой формат даты MS SQL поймет всегда (в таблице Product обратите внимание на столбец SellStartDate) :

```
SELECT *
FROM Production.Product
Where SellStartDate between '20080101' and '20090101'
```

Кроме того, если формат даты в столбце предусматривает не только дату, но и время, то можно таким образом указать поиск по дате

```
SELECT *
FROM Production.Product
WhereSellStartDate between '20090101 00:00:00.000' and '20160101 23:59:59.999'
```

Кроме того, при необходимости возможно задавать формат даты с помощью функции DATEFROMPARTS или DATETIMEFROMPARTS.

Или можно использовать функцию CONVERT, если требуется преобразовать строку в значение типа date или datetime:

```
SELECT
 CONVERT(date,'22.08.2019',104),
 CONVERT(datetime,'2019-08-21 22:40:15',120)
```

Значения 104 и 120 указывают какой формат даты используется в строке. Более подробно об этом можно почитать - тут.

Агрегатные функции

Немного теории:

Агрегатная функция выполняет вычисление на наборе значений и возвращает одиночное значение. Агрегатные функции, за исключением COUNT, не учитывают значения NULL.

Все агрегатные функции являются детерминированными. Другими словами, агрегатные функции возвращают одну и ту же величину при каждом их вызове на одном и том же наборе входных значений.

Рассмотрим только основные и наиболее часто используемые агрегатные функции:

Название	**Описание**
COUNT(*)	Возвращает количество строк полученных оператором «SELECT … WHERE …». В случае отсутствии WHERE, количество всех записей таблицы.
COUNT(столбец/выражение)	Возвращает количество значений (не равных NULL), в указанном столбце/выражении
COUNT(DISTINCT столбец/выражение)	Возвращает количество уникальных значений, не равных NULL в указанном столбце/выражении
AVG(столбец/выражение)	Возвращает среднее значение по значениям столбца/выражения. NULL значения для подсчета не учитываются.
AVG(DISTINCT столбец/выражение)	Возвращает среднее значение по уникальным значениям столбца/выражения. NULL значения для подсчета не учитываются.
SUM(столбец/выражение)	Возвращает сумму по значениям столбца/выражения
MIN(столбец/выражение)	Возвращает минимальное значение по значениям столбца/выражения
MAX(столбец/выражение)	Возвращает максимальное значение по значениям столбца/выражения

Агрегатные функции позволяют нам сделать расчет итогового значения для набора строк полученных при помощи оператора SELECT.

Рассмотрим каждую функцию на примере:

```
SELECT
 COUNT(*)          as 'Общее кол-во продуктов'
 , COUNT(Color)      as 'Кол-во продуктов у которых указан цвет'
 , COUNT(distinct Color)       as  'Кол-во используемых цветов'
 , COUNT(distinct ProductModelID)  as  'Кол-во уникальных моделей продуктов'
 , COUNT(distinct ReorderPoint)     as  'Кол-во точек заказов'
 ,SUM(ListPrice)       as  'Общая сумма цен по всем продуктам'
 ,MIN(ListPrice)       as  'Минимальная цена'
 ,MAX(ListPrice)      as  'Максимальная цена'
 , AVG (ListPrice)         as 'Средняя цена по всем продуктам'
 , AVG  (distinct ListPrice)          as  'Средняя цена по всем уникальным значениям
цен'
 , AVG  (NULLIF(ListPrice,0))          as  'Средняя цена по всем продуктам у которых
цена больше 0'
```

```
 ,COUNT(NULLIF(ListPrice,0))      as  'Кол-во продуктов у которых цена больше 0'
 ,COUNT(Case
when ListPrice>0 then NULL
else ListPrice
END )       as 'Кол-во продуктов у которых цена равна 0'
FROM Production.Product
```

Обратите внимание, что в запросе не использовались условия - WHERE, следовательно итоги будут считаться для всех данных, которые получаются запросом:

```
SELECT *  FROM Production.Product
```

Или для наглядности:

```
SELECT
 Color
,ProductModelID
,ReorderPoint
,ListPrice
 FROM Production.Product
```

Это исходные данные, по которым и будут считаться итоги агрегированного запроса.

Теперь разберем каждое агрегированное значение:

```
 COUNT(*)          as 'Общее кол-во продуктов'     = 504
```

COUNT(*) – т.к. мы не задали в запросе условия фильтрации в блоке WHERE, то COUNT(*) дало нам общее количество записей в таблице, т.е. это количество строк, которое возвращает запрос: `SELECT *  FROM Production.Product`

```
COUNT(Color)      as 'Кол-во продуктов у которых указан цвет'        = 256
```

COUNT(Color) возвращает количество строк, у которых указано значение Color(цвет), т.е. подсчитывается количество записей, у которых Color IS NOT NULL (в строке указано не NULL значение).

Здесь достаточно просто отбросить записи с NULL значениями. Берем значения колонки Color и вычеркиваем все NULL значения:

```
SELECT Color
FROM Production.Product
WHERE Color IS NOT NULL
```

```
 COUNT(distinct Color)       as  'Кол-во используемых цветов'          = 9
```

COUNT(DISTINCT Color) – вернуло нам значение 9, т.е. это число соответствует числу уникальных значений указанных в столбце Color без учета NULL значений.

При выполнении запроса ниже мы увидим какие же это значение были:

```
SELECT distinct
Color
FROM Production.Product
```

Тут нужно еще раз обратить внимание на тот факт, что по этому запросу у нас 10 уникальных значений включая - NULL.

	NULL
	Black
	Blue
	Grey
	Multi
	Red
	Silver
	Silver/Black
	White
10	Yellow

Но при запросе COUNT(distinct Color) NULL не учитывается.
Таким образом для повторения результат, мы должны отбросить NULL :

```
SELECT distinct
Color
FROM Production.Product
WHERE Color IS NOT NULL
```

```
COUNT(distinct ProductModelID) as 'Кол-во уникальных моделей продуктов'    =119
COUNT(distinct ReorderPoint)   as 'Кол-во точек заказов'                   =6
```

То же самое, что было сказано по COUNT(distinct Color).

Для понимания попробуйте не заглядывая в разбор выше расписать эти строчки другим способом.

```
SUM(ListPrice)    as 'Общая сумма цен по всем продуктам'    =221087,79
```

SUM(ListPrice) возвращает сумму всех значений в выражении. Функция SUM может быть использована только для числовых столбцов. Значения NULL пропускаются.

Кроме того, в случае если необходимости просуммировать не просто все значения в выражение, а именно уникальные значения, то можно как и в COUNT(distinct ReorderPoint) реализовать путем добавления distinct в выражение SUM(ListPrice).

Попробуйте проделать это самостоятельно и проверить результат, сумма должна получиться другой - 45052,64.

```
MIN(ListPrice)    as 'Минимальная цена'    =0,00
MAX(ListPrice)    as 'Максимальная цена'   =3578,27
```

MAX(ListPrice) возвращает максимальное значение ListPrice, опять же без учета NULL значений.

MIN(ListPrice) возвращает минимальное значение ListPrice. Как в случае с MAX, только ищем минимальное значение, игнорируя NULL.

distinct также можно указать для учета уникального значения, но не имеет смысла при использовании функцией MIN,MAX и доступен только для совместимости со стандартом ISO.

Выражения MAX(ListPrice) и MIN(ListPrice) можно вычислить по другому:

```
SELECT TOP 1 ListPrice
FROM Production.Product
WHERE ListPrice IS NOT NULL
ORDER BY ListPrice                    -- для выражения MIN(ListPrice)
```

Для выражения MAX(ListPrice) постройте альтернативное выражение самостоятельно.

```
AVG (ListPrice)        as 'Средняя цена по всем продуктам'                =438,6662
```

AVG (ListPrice) возвращает среднее значений. NULL-выражения при этом не учитываются, т.е. это соответствует больше второму выражению SUM(ListPrice)/COUNT(ListPrice):

```
SELECT
SUM( ListPrice)
COUNT(ListPrice)
AVG (ListPrice)
SUM( ListPrice)/COUNT(ListPrice)
SUM( ListPrice)/COUNT(*)
FROM Production.Product
```

Исправьте ошибки в запросе и выполните его для сравнения результатов.

Хотя тут результат и будет одинаков для всех 3 расчетов, но только потому что в столбце отсутствуют NULL-выражения.

Напомню, что COUNT(ListPrice) и COUNT(*) отличаются между собой тем, что COUNT(*) берет все строчки в таблице, а COUNT(ListPrice) только строки не содержащие NULL в конкретном столбце.

```
 AVG (distinct ListPrice)      as  'Средняя цена по всем уникальным значениям цен'
=437,4042
```

Выражение AVG (distinct ListPrice) соответствует тому что и делает AVG (ListPrice) за исключением того, что выборка производиться по уникальным значениям.

При выполнении этих двух выражений (AVG (ListPrice) и AVG (distinct ListPrice)) показалось странным что значения очень близко по выражению 438,6662 против 437,4042.

Давайте проверим - а правильно ли вычисляется результирующий набор по уникальным значениям.

Мы можем, конечно, прописать выражения SUM(distinct ListPrice) и COUNT(distinct ListPrice), но давайте на этом примере разберем как это можно сделать альтернативно, чтобы наглядно посмотреть на результаты выборки.

С помощью вот этого запроса мы выберем все уникальные значения:

```
SELECT distinct
ListPrice
FROM Production.Product
```

Но вот мы их выбрали - их у нас 103 значения. Что дальше? Суммировать на калькуляторе или может перенести в Excel и там просуммировать? Тоже выход.

Можно конечно попробовать добавить агрегатные функции SUM(ListPrice) и COUNT(ListPrice):

```
SELECT distinct
SUM( ListPrice)      = 221087,79
,COUNT(ListPrice)   = 504
FROM Production.Product
```

Но результат будет отличаться от тех 103 строк, которые были в запросе выше.

Причина в очередности выполнения операций.

Сначала у нас формируется таблица FROM Production.Product, потом выполняются агрегатные функции SUM(ListPrice) и COUNT(ListPrice) и лишь затем к этому результату мы применяем отображение только уникальных значений. Но так как строчка одна, то distinct отрабатывает в пустую.

То есть для получения нужного нам результата, нам надо чтобы SUM и COUNT выполнялись сразу на уникальных значениях. Для этого нам придется забежать чуть вперед и построить запрос с использованием подзапроса в конструкции FROM.

```
SELECT *
FROM (
SELECT distinct
ListPrice
FROM Production.Product) as qwe  -- сначала выполняется запрос выборки уникальных значений. Потом результату этой выборки мы присваиваем имя, т.е. создаем производную таблицу.
```

Помимо производных таблиц есть возможность создавать и временные таблицы. В этом блоке не будем останавливаться на них. Укажу только главное различие между ними:

*Временные таблицы существуют на протяжении **сессии** базы данных. Если такая таблица создается в редакторе запросов (Query Editor) в SQL Server Management Studio, то таблица будет существовать пока **открыт редактор запросов**. Таким образом, к временной таблице можно обращаться из разных скриптов внутри редактора запросов.*

*В отличие от временных таблиц, производные хранятся в **оперативной памяти** и существуют **только во время первого выполнения запроса**, который представляет эту таблицу.*

Таким образом, после того как мы выполнили запрос, то данные из производной таблице нигде не сохраняются.

Теперь когда у нас есть уникальные значения мы можем использоваться SUM и COUNT:

```
SELECT
```

```
SUM (qwe.ListPrice)  -- к наименованию столбца ListPrice хорошим тоном считается указывать наименование произвольной таблицы. это позволит избежать ошибок при крупных запросах.
,COUNT(qwe.ListPrice)
,SUM (qwe.ListPrice)/COUNT(qwe.ListPrice)
,AVG (qwe.ListPrice)
FROM (
SELECT distinct
ListPrice
FROM Production.Product) as qwe
```

Таким образом, мы убедились, что расчет уникальных средних значений был выполнен верно.

CASE – условный оператор языка SQL

Нам осталось разобрать 3 строки из запроса по агрегатным функциям. Вынесем их отдельно для наглядности:

```
SELECT
AVG (NULLIF(ListPrice,0))  as  'Средняя цена по всем продуктам у которых цена больше 0'
 ,COUNT(NULLIF(ListPrice,0))  as  'Кол-во продуктов у которых цена больше 0'
 ,COUNT(Case
when ListPrice>0 then NULL
else ListPrice
END )  as 'Кол-во продуктов у которых цена равна 0'
FROM Production.Product
```

И если операторы COUNT и AVG нам уже знакомы и больше не пугают (надеюсь), то вот все остальное довольно необычно.

Начнем разбор с конца, а именно с условного оператора CASE. Данный оператор позволяет осуществить проверку условий и возвратить, в зависимости от выполнения того или иного условия, тот или иной результат.

Оператор CASE имеет 2 формы:

Первая форма: простое выражение. Сравнивает первое выражение с выражением в каждом предложении WHEN	**Вторая форма: поисковое выражение. Вычисляет в указанном порядке выражения для каждого предложения WHEN**

CASE WHEN условие_1 THEN возвращаемое_значение_1 … WHEN условие_N THEN возвращаемое_значение_N [ELSE возвращаемое_значение] END	CASE проверяемое_значение WHEN сравниваемое_значение_1 THEN возвращаемое_значение_1 … WHEN сравниваемое_значение_N THEN возвращаемое_значение_N [ELSE возвращаемое_значение] END

При вычислении данных в 'Кол-во продуктов у которых цена равна 0' используется запись первой формы. Разберем ее более подробно, чуть модифицировав запрос для объяснения:

```
SELECT
ProductID
,name
,ListPrice
,Case
when ListPrice=0 then '100500'
when ListPrice>0 then NULL
else ListPrice
END       as 'Кол-во продуктов у которых цена равна 0'
FROM Production.Product
```

WHEN-условия проверяются последовательно, сверху-вниз. При достижении первого удовлетворяющего условия, дальнейшая проверка прерывается и возвращается значение, указанное после слова THEN, относящегося к данному блоку WHEN.

Если ни одно из WHEN-условий не выполняется, то возвращается значение, указанное после слова ELSE (что в данном случае означает «ИНАЧЕ ВЕРНИ …»).

Если ELSE-блок не указан и не выполняется ни одно WHEN-условие, то возвращается NULL.

И в первой, и во второй форме ELSE-блок идет в самом конце конструкции CASE, т.е. после всех WHEN-условий.

То есть можно прочитать так: если значение в строке равно 0, то укажи 100500. Если значение не 0, переходим к следующему условию: если значение в строке больше 0, то укажи NULL. Если значение и этому условию не равно, то ИНАЧЕ ВЕРНИ просто это значение. Конец.

С учетом всего сказанного можно по-разному прописывать условия для получения нужного нам результата:

```
SELECT
Case
when ListPrice>0 then NULL
else ListPrice
END       as '1:Кол-во продуктов у которых цена равна 0'
```

```
,Case
when ListPrice=0 then '0,00'
END       as '2:Кол-во продуктов у которых цена равна 0'
,Case
when ListPrice=0 then '0.00'
when ListPrice>0 then NULL
else '-1'
END       as '3:Кол-во продуктов у которых цена равна 0'
FROM Production.Product
```

Все зависит от ваших предпочтений и удобства.

Единственное, что нужно обязательно учитывать при использовании оператора CASE, это именно обработка ошибок. То есть, если вы при написании условий не учли какое-то условие, то обязательно нужно указывать ELSE-блок. Это поможет вам не терять какие-то значения, так как если этот блок не указан, у вас по этому выражению будет проставлен NULL.

Рассмотрим теперь вторую форму:

```
SELECT
ProductID
,name
,ListPrice
,Case ListPrice
when 0 then '100500'
when 337.22 then '3'
when 1364.50 then '4'
else NULL
END       as 'Кол-во продуктов у которых цена равна 0'
FROM Production.Product
```

Здесь делается последовательная проверка значения ListPrice с WHEN-значениями. При достижении первого равенства с WHEN-значением, проверка прерывается и возвращается значение, указанное после слова THEN, относящегося к данному блоку WHEN.

Соответственно, значение блока ELSE возвращается в случае, если ListPrice не совпал ни с одним WHEN-значением.

Если блок ELSE отсутствует, то в случае несовпадения ListPrice ни с одним WHEN-значением будет возвращено NULL.

Условно говоря вторая форма - это всего лишь упрощенная запись для тех случаев, когда нам нужно сделать сравнение на равенство, одного и того же проверяемого значения с каждым WHEN-значением/выражением.

Первая и вторая форма CASE входят в стандарт языка SQL, поэтому применимы во многих СУБД.

В языке MS SQL с 2012 версии появилась упрощенная форма записи IIF. Она может использоваться для упрощенной записи конструкции CASE, в том случае если возвращаются только 2 значения. Конструкция IIF имеет следующий вид:

```
IIF(условие, true_значение, false_значение)
```

Т.е. по сути это обертка для следующей CASE конструкции:

CASE WHEN условие THEN true_значение ELSE false_значение END

То есть с помощью этой конструкции мы можем представить

```
SELECT
ProductID
,name
,ListPrice
,Case
when ListPrice=0 then '100500'
when ListPrice>0 then NULL
else ListPrice
END       as 'Кол-во продуктов у которых цена равна 0'
FROM Production.Product
```

как

```
SELECT
ProductID
,name
,ListPrice
,IIF (ListPrice=0,'100500', null)   as 'Кол-во продуктов у которых цена равна 0'
FROM Production.Product
```

По точно такому же принципу раскладывается и NULLIF(ListPrice,0) из нашего запроса по агрегаторам. Данный оператор спрашивает если ListPrice равен 0, то указываем значение NULL, иначе берется значение ListPrice.

То есть строчки:

```
 SELECT
AVG (NULLIF(ListPrice,0))       as  'Средняя цена по всем продуктам у которых цена больше 0'
  ,COUNT(NULLIF(ListPrice,0))      as  'Кол-во продуктов у которых цена больше 0'
FROM Production.Product
```

можно преобразовать как:

```
  SELECT
AVG (NULLIF(ListPrice,0))       as  'Средняя цена по всем продуктам у которых цена больше 0'
,AVG (IIF(ListPrice=0,NULL,ListPrice))
,AVG (CASE when ListPrice=0 then NULL else ListPrice  END)
  ,COUNT(NULLIF(ListPrice,0))      as  'Кол-во продуктов у которых цена больше 0'
  ,COUNT (IIF(ListPrice=0,NULL,ListPrice))
  ,COUNT (CASE when ListPrice=0 then NULL else ListPrice  END)
  FROM Production.Product
```

Небольшое задание для закрепления результата - напишите запрос, который подсчитает количество строк с нулевым значением в колонке ListPrice 2 способами.

При рассмотрении агрегатных функций мы не использовали условия WHERE. Соответственно, при задании с агрегатными функциями дополнительного условия в блоке WHERE, будут подсчитаны только итоги, по строкам удовлетворяющих условию. Т.е. расчет агрегатных значений происходит для итогового набора, который получен при помощи конструкции SELECT.

Попробуйте самостоятельно прописывать условия к запросу к агрегатным функциям и посмотреть как будет изменяться итоговый результат.

И еще одна небольшая фишечка:

В случае, если агрегатная функция возвращает NULL, или в выборку не попало ни одной записи, а в отчете, для такого случая нам нужно показать 0, то функцией ISNULL можно обернуть агрегатное выражение:

```
SELECT
 AVG(ListPrice)      as 'Кол-во продуктов у которых цена больше 5000 '
 ,ISNULL (AVG(ListPrice),0) as 'Кол-во продуктов у которых цена больше 5000 '
 FROM Production.Product
WHERE ListPrice  >5000
```

GROUP BY – группировка данных

До текущего момента мы с вами вычисляли значения агрегатных функций для всей таблицы или по определенным условиям, но результат был в одну строчку.А что делать, если нам понадобится сделать выборку количества и средних цен, например, по каждому цвету?

Может попробовать сделать так:

```
SELECT
avg (ListPrice)
,COUNT (ListPrice)
FROM Production.Product
Where Color='Black'
```

И так по каждому цвету, а потом копировать и вставлять в Excel. Не удобно?

Тогда нам на помощь придет конструкция **GROUP BY** - *разделяет результат запроса на группы строк обычно с целью выполнения одного или нескольких статистических вычислений в каждой группе. Инструкция SELECT возвращает одну строку для каждой группы.*

Это полезная конструкция чаще всего применяемая именно для статистического анализа, хотя и не только для этого. Более подробно о ней можно почитать - тут.

Мы же рассмотрим самый основной базис. Вот так выглядит запрос по которому мы можем получить средние цены и количество видов товаров по разным цветам.

```
SELECT
Color as 'Цвет'
```

```
,avg (ListPrice) as 'Средняя цена'
,COUNT (ListPrice) as 'Количество разных продуктов в этом цвете'
FROM Production.Product
group by Color
```

Давайте теперь на этом примере попробуем разобраться как работает GROUP BY.

Для полей, перечисленных после GROUP BY из таблицы Production.Product определяются все уникальные значения по колонке Color , т.е. происходит примерно следующее:

```
SELECT distinct
Color as 'Цвет'
FROM Production.Product
```

	NULL
	Black
	Blue
	Grey
	Multi
	Red
	Silver
	Silver/Black
	White
10	Yellow

После чего по каждому значению происходит пробежка и делаются вычисления агрегатных функций:

```
SELECT
avg (ListPrice)
,COUNT (ListPrice)
FROM Production.Product
Where Color='Black'

SELECT
avg (ListPrice)
,COUNT (ListPrice)
FROM Production.Product
Where Color='Blue'
```

и т.д.

А потом все эти результаты объединяются вместе и отдаются нам в виде одного набора:

Цвет	Средняя цена	Количество разных продуктов в этом цвете
NULL	16,8641	248
Black	725,121	93

Blue	923,6792	26
Grey	125	1
Multi	59,865	8
Red	1401,95	38
Silver	850,3053	43
Silver/Black	64,0185	7
White	9,245	4
Yellow	959,0913	36

В предложении GROUP BY можно указывать несколько полей «GROUP BY поле1, поле2, …, полеN», в этом случае группировка произойдет по группам, которые образовывают значения данных полей «поле1, поле2, …, полеN».

Представим, что нам нужен не просто группировка по цвету, но еще и по пункту заказа - ReorderPoint:

```
SELECT
Color as 'Цвет'
,ReorderPoint as 'Точка заказа'
,avg (ListPrice) as 'Средняя цена'
,COUNT (ListPrice) as 'Количество разных продуктов в этом цвете'
FROM Production.Product
group by Color,ReorderPoint
```

Отличием этого запроса от первого будет то, что у нас вначале определяться не уникальные значения по колонке Color, а уникальные комбинации по 2 колонкам Color и ReorderPoint:

```
SELECT distinct
Color as 'Цвет'
,ReorderPoint as 'Точка заказа'
FROM Production.Product
```

И уже затем по каждой комбинации происходит пробежка и делаются вычисления агрегатных функций.

Из основного, стоит отметить, что в случае группировки (GROUP BY), в перечне колонок в блоке SELECT:

- Мы можем использовать только колонки, перечисленные в блоке GROUP BY
- Можно использовать выражения с полями из блока GROUP BY
- Можно использовать константы, т.к. они не влияют на результат группировки
- Все остальные поля (не перечисленные в блоке GROUP BY) можно использовать только с агрегатными функциями (COUNT, SUM, MIN, MAX, …)
- Не обязательно перечислять все колонки из блока GROUP BY в списке колонок SELECT

Пример для подтверждения:

```
SELECT
 'Color+ReorderPoint' Const1, -- константа в виде строки
  '31.08.2019' Const2, -- константа в виде даты
```

```
 CONCAT('ReorderPoint','-',ReorderPoint) as Point -- выражение с использованием полей участвующих в группировке

,Color as 'Цвет' -- поле из списка полей участвующих в группировке
--,ReorderPoint as 'Точка заказа' - поле участвующее в группировке, не обязательно дублировать здесь
-- остальные поля можно использовать только с агрегатными функциями: COUNT, SUM, MIN, MAX, …
,avg (ListPrice) as 'Средняя цена'
,COUNT (ListPrice) as 'Количество разных продуктов в этом цвете'
FROM Production.Product
group by Color,ReorderPoint
```

GROUP BY в скупе с агрегатными функциями, одно из основных средств, служащих для получения сводных данных из БД, ведь обычно данные в таком виде и используются. И конечно же все это крутится вокруг знания базовой конструкции, т.к. прежде чем что-то подытожить (агрегировать), нужно первым делом это правильно выбрать, используя «SELECT … WHERE …».

В заключение по GROUP BY давайте привнесем немного красоты и обработаем значение NULL в ‘no color’, а также отсортируем результат по цвету:

```
SELECT
 'Color+ReorderPoint' Const1,
 , '31.08.2019' Const2
, CONCAT('ReorderPoint','-',ReorderPoint) as Point
,case when Color is null then 'No color' else Color end as 'Цвет'
,avg (ListPrice) as 'Средняя цена'
,COUNT (ListPrice) as 'Количество разных продуктов в этом цвете'
FROM Production.Product
group by Color,ReorderPoint
order by Color
```

HAVING – наложение условия выборки к сгруппированным данным

Если вы поняли, что такое группировка и WHERE-условие, то с HAVING ничего сложного нет.

HAVING – чем-то подобен WHERE, только если WHERE-условие применяется к детальным данным, то HAVING-условие применяется к уже сгруппированным данным. По этой причине в условиях блока HAVING мы можем использовать либо выражения с полями, входящими в группировку, либо выражения, заключенные в агрегатные функции.

В HAVING-условии так же можно строить сложные условия, используя операторы AND, OR и NOT как и в WHERE-условие.

Разберем краткий пример такого условия:

```
SELECT
 'Color+ReorderPoint' Const1,
 , '31.08.2019' Const2
, CONCAT('ReorderPoint','-',ReorderPoint) as Point
,case when Color is null then 'No color' else Color end as 'Цвет'
,avg (ListPrice) as 'Средняя цена'
,COUNT (ListPrice) as 'Количество разных продуктов в этом цвете'
FROM Production.Product
where ReorderPoint=375  -- точка заказа равна 375
group by Color,ReorderPoint
having avg (ListPrice)>500 and COUNT (ListPrice)>10  -- средняя цена по цвету должна превышать 500 и количество таких товаров больше 10
order by Color
```

К итоговому примеру из темы про группировку мы добавили 2 условия, что выборка данных происходит по точке заказа №375, и что нам нужны цвета и номенклатура продуктов, у которых средняя стоимость превышает 500 и номенклатура превышает 10.

На протяжении всего предыдущего упоминался порядок выполнения тех или иных конструкций относительно друг друга. Ниже в таблице представлена сводная информация о порядке их выполнения:

Конструкция/Блок	**Порядок выполнения**	**Выполняемая функция**
SELECT возвращаемые выражения	4	Возврат данных полученных запросом
FROM источник	0	Формирование строк таблицы: из одной либо с нескольких
WHERE условие выборки из источника	1	Отбираются только строки, проходящие по условию
GROUP BY выражения группировки	2	Создание групп по указанному выражению группировки. Расчет агрегированных значений по этим группам, используемых в SELECT либо HAVING блоках
HAVING фильтр по сгруппированным данным	3	Фильтрация, накладываемая на сгруппированные данные
ORDER BY выражение сортировки результата	5	Сортировка данных по указанному выражению

Операции горизонтального соединения

Сначала немного теории.

Если говорить просто, то операции горизонтального соединения таблицы с другими таблицами используются для того, чтобы получить из них недостающие данные.

Есть пять типов соединения:

1. JOIN – левая_таблица JOIN правая_таблица ON условия_соединения

2. LEFT JOIN – левая_таблица LEFT JOIN правая_таблица ON условия_соединения

3. RIGHT JOIN – левая_таблица RIGHT JOIN правая_таблица ON условия_соединения

4. FULL JOIN – левая_таблица FULL JOIN правая_таблица ON условия_соединения

5. CROSS JOIN – левая_таблица CROSS JOIN правая_таблица

Краткий синтаксис	**Полный синтаксис**	**Описание**
JOIN	INNER JOIN	Из строк левой_таблицы и правой_таблицы **объединяются** и возвращаются **только те строки**, по которым **выполняются** условия_соединения.
LEFT JOIN	LEFT OUTER JOIN	Возвращаются **все строки левой_таблицы** (ключевое слово LEFT). **Данными правой_таблицы** дополняются только те строки левой_таблицы, для которых **выполняются условия_соединения**. Для **недостающих** данных вместо строк правой_таблицы **вставляются NULL-значения**.
RIGHT JOIN	RIGHT OUTER JOIN	Возвращаются **все строки правой_таблицы** (ключевое слово RIGHT). **Данными левой_таблицы** дополняются только те строки правой_таблицы, для которых **выполняются условия_соединения**. Для **недостающих** данных вместо строк левой_таблицы **вставляются NULL-значения.**

FULL JOIN	FULL OUTER JOIN	Возвращаются **все строки левой_таблицы и правой_таблицы**. Если для строк левой_таблицы и правой_таблицы **выполняются условия_соединения**, то они объединяются в одну строку. Для строк, для которых не выполняются условия_соединения, NULL-значения вставляются на место левой_таблицы, либо на место правой_таблицы, в зависимости от того данных какой таблицы в строке не имеется.
CROSS JOIN	-	Объединение каждой строки левой_таблицы со всеми строками правой_таблицы. Этот вид соединения иногда называют декартовым произведением.

Если не понятно описание в таблице, то просто вернитесь сюда после рассмотрения примеров.

При написании запросов далее используется только краткий синтаксис, так как это экономит время.

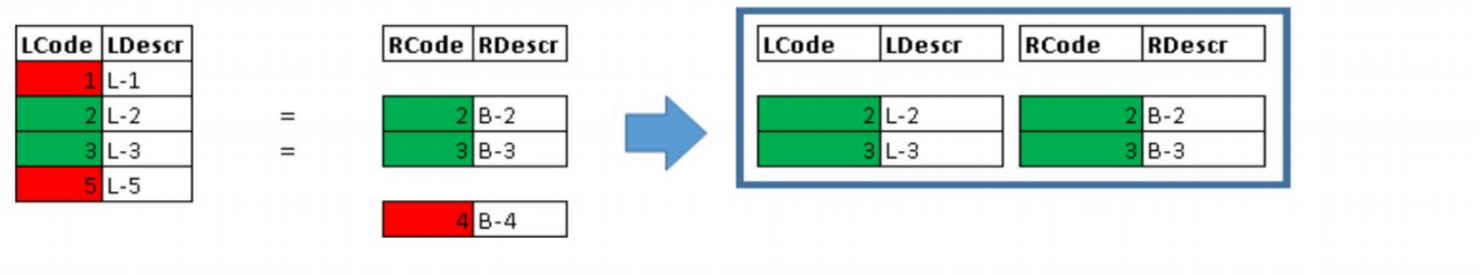
Разберем каждый вид горизонтального объединения:

1. JOIN

Синтаксис:

```
SELECT l.*,r.*
FROM LeftTable as l
JOIN RightTable as r ON l.LCode=r.RCode
```

Здесь были возвращены объединения строк для которых выполняется условие (l.LCode=r.RCode)



Для понимания того как происходит объединение, разберем это соединение подробнее. Для этого возьмем 2 таблицы:

1. Production.Product - уже нам знакомая по предыдущим запросам

```
SELECT *
FROM Production.Product -- результат 504 строки
```

2. Production.ProductReview - таблицу отзывов на товары из первой таблицы.

```
SELECT *
FROM Production.ProductReview -- результат 4 строки
```

Обратите внимание на одинаковой столбец ProductID у той и у другой таблицы. Это так называемый первичный для Production.Product и внешний ключ для Production.ProductReview.

Если подзабыли что такое первичный и внешний ключ, стоит вернуться к блоку с теоретической информацией и еще раз ее внимательно перечитать.

Далее по значениям из столбцов ProductID мы решаем объединить эти 2 таблицы.

```
SELECT *
  FROM Production.Product as P -- присваиваем псевдоним таблице
       JOIN Production.ProductReview as R -- присваиваем псевдоним таблице
         on P.Productid=R.ProductID -- прописываем условие соединения по столбцу ProductID
```

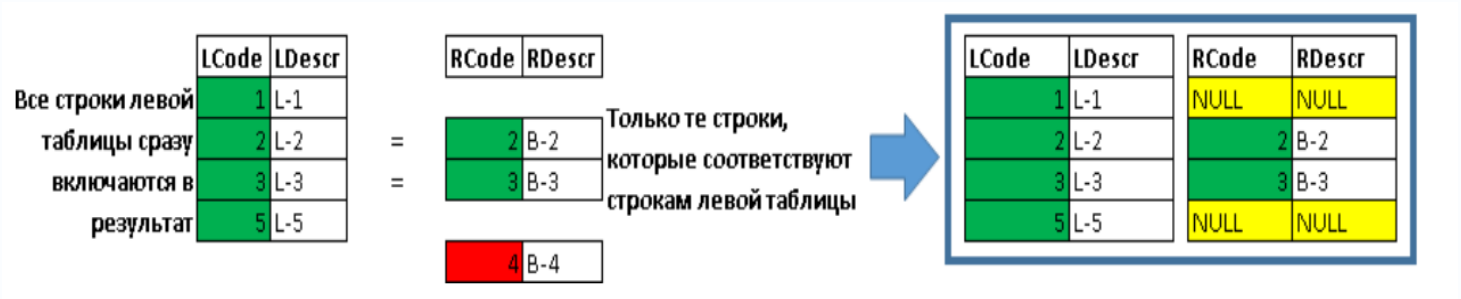
В результате объединения у нас получаются только 4 строки, данные по которым совпадают и в таблице Production.Product и в таблице Production.ProductReview. Все остальные значения не прошедшие условие объединения отброшены.

Обратите внимание, что 2 и 3 строчка имеют одинаковый ProductID. То есть на один и тот же товар было сделано 2 отзыва. В результате значение из таблицы Product повторилось при соединении дважды и все различия между этими 2 строками находятся в столбцах мигрированных из таблицы ProductReview.

Такие ситуации стоит учитывать при дальнейшей выборке.

2. **LEFT JOIN**

```
SELECT l.*,r.*
FROM LeftTable l
LEFT JOIN RightTable r ON l.LCode=r.RCode
```



Все также 2 таблицы:

1. Production.Product - таблица товаров

```
  SELECT *
  FROM Production.Product -- результат 504 строки
```

2. Production.ProductReview - таблицу отзывов на товары из первой таблицы.

```
SELECT *
  FROM Production.ProductReview -- результат 4 строки
```

Объединяем таблицы с использованием оператора LEFT JOIN:

```
SELECT *
```

```
  FROM Production.Product as P -- присваиваем псевдоним таблице
            LEFT JOIN Production.ProductReview as R -- присваиваем псевдоним
таблице
        on P.Productid=R.ProductID -- прописываем условие соединения по столбцу
ProductID
```

Здесь были возвращены все строки Product, которые были дополнены данными строк из ProductReview, для которых выполнилось условие (P.Productid=R.ProductID).

Отличием от INNER JOIN является то, что за основу берется левая таблица и к ней добавляются строки по которым выполняется условие соединения. При этом по строкам не прошедшим проверку на соединение проставляется значение NULL.

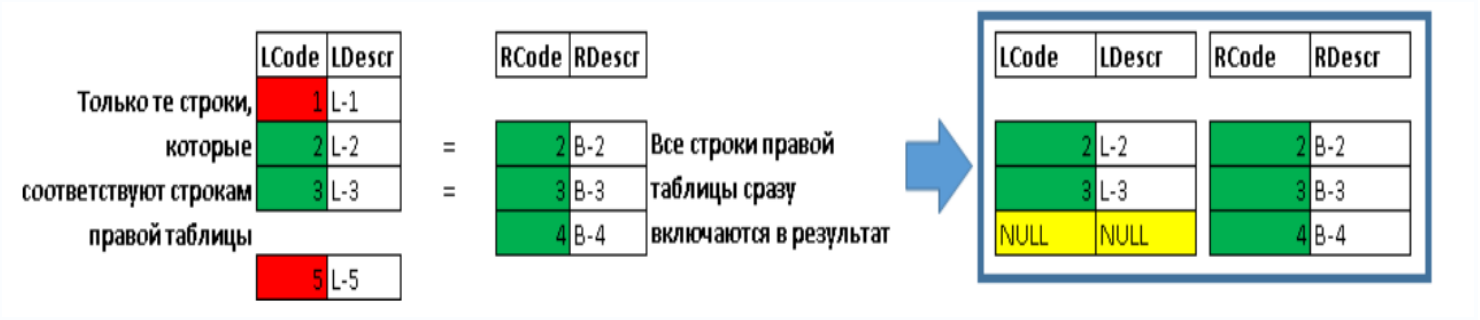
В результате объединения у нас получаются только 505 строк.

505 строчка все также появилась в результате 2 отзывов на один товар.

При сдвиге ползунка вправо, в столбцах из таблицы ProductReview, по которым не было выполнено условие соединения, указано значение NULL.

3. **RIGHT JOIN**

```
SELECT l.*,r.*
FROM LeftTable l
RIGHT JOIN RightTable r ON l.LCode=r.RCode
```



RIGHT JOIN запрос обратный LEFT JOIN. При выполнении этого оператора за основу берется правая таблица.

```
SELECT *
  FROM Production.ProductReview -- результат 4 строки
```

Объединяем таблицы с использованием оператора RIGHT JOIN:

```
SELECT *
  FROM Production.Product as P -- присваиваем псевдоним таблице
            RIGHT JOIN Production.ProductReview as R -- присваиваем псевдоним
таблице
        on P.Productid=R.ProductID -- прописываем условие соединения по столбцу
ProductID
```

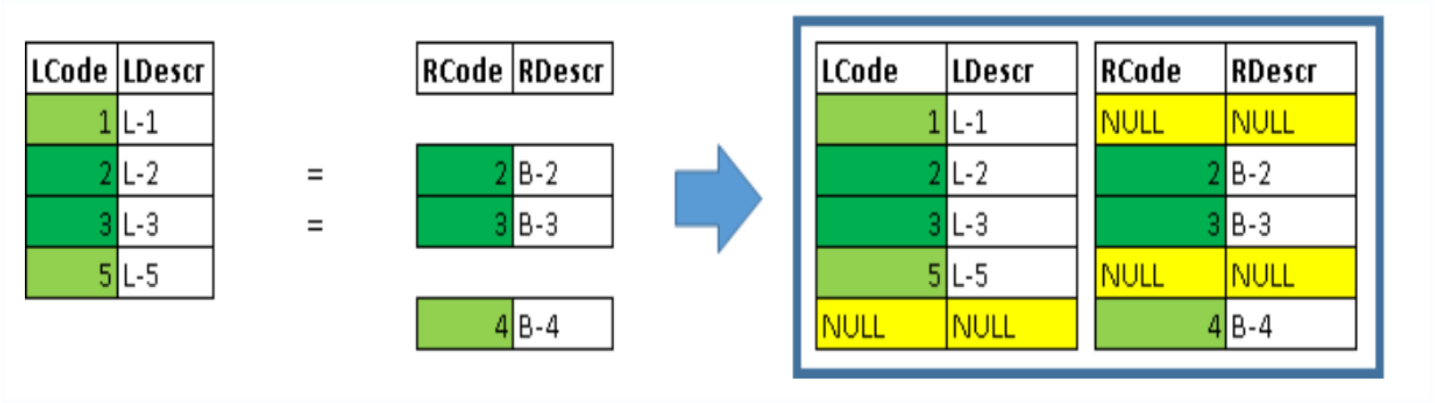
Здесь были возвращены все строки ProductReview, которые были дополнены данными строк из Product, для которых выполняется условие (P.Productid=R.ProductID).

В результате объединения у нас получаются только 4 строки. Так как в таблице ProductReview не содержится строк ProductID, которые не содержатся в таблице Product, то запрос фактически получился идентичным INNER JOIN. Так бывает не всегда, все зависит от конкретных таблиц, а также порядка их соединения.

Попробуйте поменять порядок соединения таблиц - первая будет ProductReview, вторая Product и проанализируйте результат.

4. **FULL JOIN** – это по сути одновременный LEFT JOIN + RIGHT JOIN

```
SELECT l.*,r.*
FROM LeftTable l
FULL JOIN RightTable r ON l.LCode=r.RCode
```



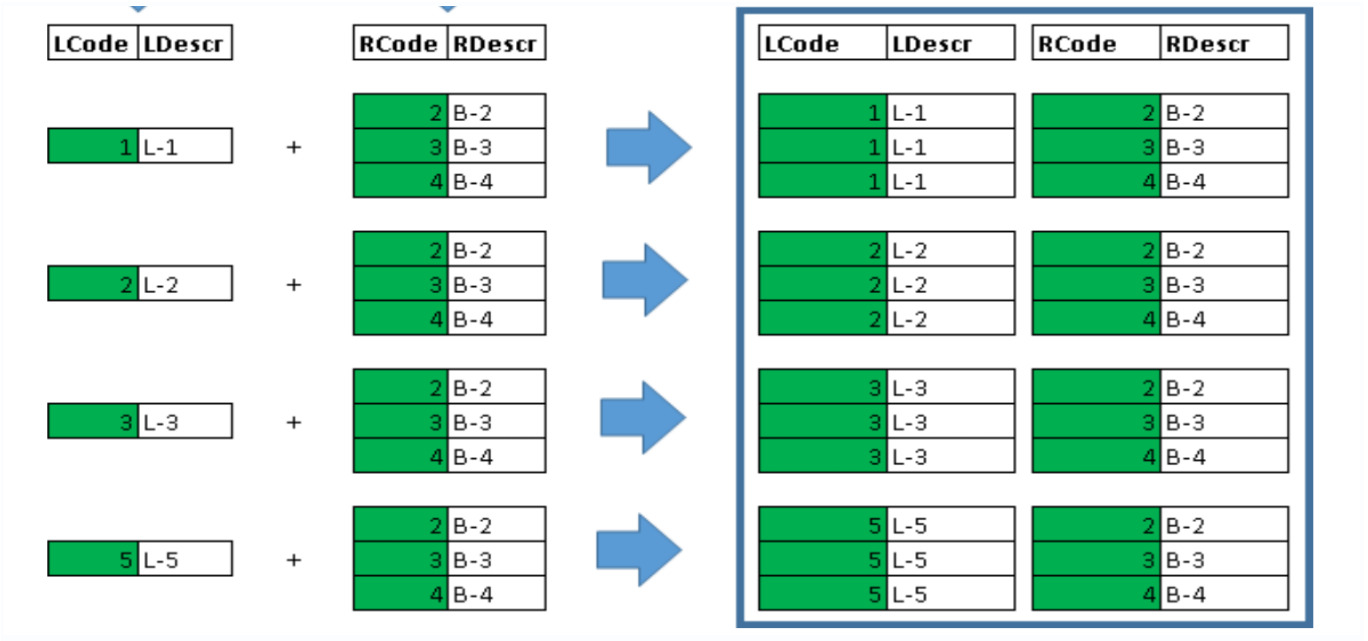

```
SELECT *
  FROM Production.Product as P -- присваиваем псевдоним таблице
            FULL JOIN Production.ProductReview as R -- присваиваем псевдоним таблице
        on P.Productid=R.ProductID -- прописываем условие соединения по столбцу ProductID
```

Вернулись все строки из Product и ProductReview. Строки для которых выполнилось условие (P.Productid=R.ProductID) были объединены в одну строку. Отсутствующие в строке данные с левой или правой стороны заполняются NULL-значениями.

Фактически выполняется сразу и LEFT JOIN, и RIGHT JOIN соединение.

5. **CROSS JOIN**

```
SELECT l.*,r.*
FROM LeftTable l
CROSS JOIN RightTable r
```




```
SELECT *
  FROM Production.Product
       CROSS JOIN Production.ProductReview
```

Каждая строка Product соединяется с данными всех строк ProductReview.
504 *4=2016 строк в результирующем наборе.

В заключение об операторах горизонтального поведение хочу отметить, что наиболее частыми для использования являются операторы JOIN и LEFT JOIN, все остальные операторы можно заменить этими предварительно обдумав в каком именно порядке и по каким условиям будут соединяться таблицы.

Кроме того, соединять можно не только 2 таблицы, но и больше. Примером такого соединения может быть пример, когда помимо данных по продуктам и отзывам, нам необходимо получить данные о категории и подкатегории товара.

(Бизнес требование - по каким товарам,с указанием категории и подкатегории товара приходили отзывы. Необходимо также вывести наименование товара, цвет если есть, цену, отзыв, подкатегорию товара и категорию товара).

В этом случае запрос будет строить так выглядеть так:

1. За основу берем таблицу Product - так как это таблица является основой с наиболее полной информацией:

```
SELECT *
  FROM Production.Product
```

2. К ней добавляем таблицу ProductReview. Соединить можно будет по 2 видам соединений по JOIN и LEFT JOIN. У каждого соединения есть свои плюсы и минусы, поэтому проведем сразу 2 ветки соединения:

1 вариант:

```
SELECT *
  FROM Production.Product as P
       JOIN Production.ProductReview as R on P.Productid=R.ProductID
```

2 вариант:

```
SELECT *
  FROM Production.Product as P
       LEFT JOIN Production.ProductReview as R on P.Productid=R.ProductID
```

3. Теперь нам к полученному результату необходимо добавить данные о подкатегории товара из таблицы ProductSubcategory. При анализе этой таблицы (попробуйте сами построить запрос, чтобы просмотреть какие колонки в ней присутствуют), можем заметить что у Product и ProductSubcategory есть общая колонка ProductSubcategoryID. По ней и проведем соединение с вариантами из 2 шага. Соединение будет только по LEFT JOIN, так как из этой таблицы нам может понадобится только 1 колонка - наименование подкатегории.

1 вариант:

```
SELECT *
  FROM Production.Product as P
       JOIN Production.ProductReview as R on P.Productid=R.ProductID
      LEFT JOIN Production.ProductSubcategory as S  -- присваиваем псевдоним
on P.ProductSubcategoryID=S.ProductSubcategoryID -- указываем соединение
```

2 вариант:

```
SELECT *
  FROM Production.Product as P
       LEFT JOIN Production.ProductReview as R on P.Productid=R.ProductID
      LEFT JOIN Production.ProductSubcategory as S  -- присваиваем псевдоним
on P.ProductSubcategoryID=S.ProductSubcategoryID -- указываем соединение
```

4. Теперь к полученному варианту нам необходимо добавить данные об общей категории товара из таблицы ProductCategory. Первичным ключом у этой таблицы является колонка ProductCategoryID. В таблице Product такого внешнего ключа нет, зато такой ключ есть в таблице ProductSubcategory. С ней и будем соединять по LEFT JOIN:

1 вариант:

```
SELECT *
  FROM Production.Product as P
       JOIN Production.ProductReview as R on P.Productid=R.ProductID
      LEFT JOIN Production.ProductSubcategory as S
on P.ProductSubcategoryID=S.ProductSubcategoryID
      LEFT JOIN Production.ProductCategory as C  --присваиваем псевдоним
on S.ProductCategoryID=C.ProductCategoryID -- соединяем 2 таблицы
ProductSubcategory и ProductCategory
```

2 вариант:

```
SELECT *
  FROM Production.Product as P
       LEFT JOIN Production.ProductReview as R on P.Productid=R.ProductID
      LEFT JOIN Production.ProductSubcategory as S
```

```
on P.ProductSubcategoryID=S.ProductSubcategoryID
    LEFT JOIN Production.ProductCategory as C  --присваиваем псевдоним
on  S.ProductCategoryID=C.ProductCategoryID  --  соединяем  2  таблицы
ProductSubcategory и ProductCategory
```

5. Теперь когда мы соединили таблицы - нужно прописать WHERE условия, если они необходимы:

1 вариант: - так как условием было вывести товары по которым были отзывы, то в первом варианте условие дополнительно прописывать не нужно. С помощью JOIN мы итак его выполнили.

```
SELECT *
  FROM Production.Product as P
    JOIN Production.ProductReview as R on P.Productid=R.ProductID
    LEFT JOIN Production.ProductSubcategory as S
on P.ProductSubcategoryID=S.ProductSubcategoryID
    LEFT JOIN Production.ProductCategory as C
on S.ProductCategoryID=C.ProductCategoryID
```

2 вариант: - а вот в этом варианте условие понадобится

```
SELECT *
  FROM Production.Product as P
    LEFT JOIN Production.ProductReview as R on P.Productid=R.ProductID
    LEFT JOIN Production.ProductSubcategory as S
on P.ProductSubcategoryID=S.ProductSubcategoryID
    LEFT JOIN Production.ProductCategory as C
on S.ProductCategoryID=C.ProductCategoryID
 WHERE  R.Comments is not null  --указываем условие, что нужны только
строчки, где в поле Comments из таблицы ProductReview (псевдоним R) ненулевое
значение.
```

6. И финишная операция, указываем колонки которые нам будут нужны (*наименование товара, цвет, цену, отзыв, подкатегорию товара и категорию товара):*

1 вариант:

```
SELECT
p.Name
,p.Color
,p.ListPrice
,R.Comments
,S.Name
,C.Name -- прописываем колонки с учетом тех псевдонимов таблиц которые мы
выбрали
  FROM Production.Product as P
    JOIN Production.ProductReview as R on P.Productid=R.ProductID
    LEFT JOIN Production.ProductSubcategory as S
on P.ProductSubcategoryID=S.ProductSubcategoryID
    LEFT JOIN Production.ProductCategory as C
```

```
on S.ProductCategoryID=C.ProductCategoryID
2 вариант:
SELECT
p.Name
,p.Color
,p.ListPrice
,R.Comments
,S.Name
,C.Name -- прописываем колонки с учетом тех псевдонимов таблиц которые мы выбрали
  FROM Production.Product as P
      LEFT JOIN Production.ProductReview as R on P.Productid=R.ProductID
    LEFT JOIN Production.ProductSubcategory as S
on P.ProductSubcategoryID=S.ProductSubcategoryID
    LEFT JOIN Production.ProductCategory as C
on S.ProductCategoryID=C.ProductCategoryID
 WHERE    R.Comments is not null
```

Как видно из результат 2 запросов - результат идентичен. При этом у LEFT JOIN -соединении нам понадобилось дополнительно прописывать условие. Однако первый вариант сложнее будет изменить, если вдруг понадобится изменить выборку.

Типы связи между таблицами

Отвлечемся от конкретного написания запросов и остановимся еще на одном важном моменте, с которым мы обязательно столкнемся при пользовании базы данных - типы связи между таблицами.

Этот тема напрямую связана с предыдущей темой - “Операции горизонтального соединения”, но вынесена отдельно, так как является одной из важнейших с точки зрения аналитики, как на момент продумывания и создания логики базы данных (на момент проектирования), так и при составлении самого запроса и последующего анализа результата.

Существуют три типа связи между таблицами:

1. Один к одному
2. Один ко многим
3. Многие ко многим

Для раскрытия этой темы представим, что нам необходимо спроектировать базу данных, где будут храниться Отделы и Сотрудники. Это 2 отдельных таблицы, в каждой из которых есть некий список.

Отдел	
ID	Name
1	IT
2	Presale
3	QA

Сотрудники	
ID	Name
1	Иванов
2	Петров
3	Сидоров

Эти таблицы пока никак не связаны между собой. И нам необходимо установить эту связь.

Сделаем упрощение и представим, что каждый из сотрудников работает только в одном отделе:

В этом случае нам достаточно просто в одну из таблиц добавить новый столбец, где укажем ID из другой таблицы. Этот столбец добавим в таблицу “Сотрудники”.

Отдел	
ID	Name
1	IT
2	Presale
3	QA

Сотрудники		
ID	Name	ID Отдел
1	Иванов	1
2	Петров	3
3	Сидоров	2

Такой тип связи называется “один к одному”. Суть тут в том, что каждая строчка может соединиться только к одной строке другой таблицы.

В результате запроса:

```
SELECT
*
  FROM Сотрудники as P
      JOIN Отдел as R on P.ID_Отдел=R.ID
```

Получим результат в 3 строки.

ID	Name	ID Отдел	ID	Name
1	Иванов	1	1	IT
2	Петров	3	3	QA
3	Сидоров	2	2	Presale

Но понятно, что количество сотрудников в отделе может быть больше чем 1 человек. Добавим еще одного сотрудника:

Отдел	
ID	Name
1	IT
2	Presale
3	QA

Сотрудники		
ID	Name	ID Отдел
1	Иванов	1
2	Петров	3
3	Сидоров	2
4	Валуев	1

В этом случае одна строка одной таблицы будет стыковать сразу к нескольким строкам другой таблицы:

Отдел	
ID	Name
1	IT
2	Presale
3	QA

Сотрудники		
ID	Name	ID Отдел
1	Иванов	1
2	Петров	3
3	Сидоров	2
4	Валуев	1

В результате запроса:

```
SELECT
*
  FROM Сотрудники as P
      JOIN Отдел as R on P.ID_Отдел=R.ID
```

Получим вот такой результат.

ID	Name	ID Отдел	ID	Name
1	Иванов	1	1	IT
2	Петров	3	3	QA
3	Сидоров	2	2	Presale
4	Валуев	1	1	IT

Такой тип связи называется “один ко многим”. Опасность тут может представлять тот момент, что данные из одной таблицы задваиваются (серая область).

И если после установления связи в запросе, мы выполняем еще какие-нибудь манипуляции (например, суммирование цифровых показателей в таблице Отдел) и для подсчета выбрали колонки, в которых произошли задвоение, то результат будет ошибочный.

У нас остался последний тип связи “многие ко многим”. Вернемся к нашим 2 таблицам.

Отдел	
ID	Name
1	IT
2	Presale
3	QA

Сотрудники		
ID	Name	ID Отдел
1	Иванов	1
2	Петров	3
3	Сидоров	2
4	Валуев	1

Неожиданно возник жизненный кейс - один из сотрудников (Сидоров) начал работать на полставки сразу в 2 отделах (Presale и QA).

Отдел	
ID	Name
1	IT
2	Presale
3	QA

Сотрудники		
ID	Name	ID Отдел
1	Иванов	1
2	Петров	3
3	Сидоров	2
4	Валуев	1

Как нам организовать эту таблицу? Добавить значение в уже существующий столбец?

Сотрудники		
ID	Name	ID Отдел
1	Иванов	1
2	Петров	3
3	Сидоров	2,3
4	Валуев	1

Так нельзя!

Или добавить еще один столбец в таблицу Отдел?

Отдел		
ID	Name	ID Сотрудники
1	IT	1
2	Presale	3
3	QA	2
4	QA	3

Сотрудники		
ID	Name	ID Отдел
1	Иванов	1
2	Петров	3
3	Сидоров	2
4	Валуев	1

Так тоже категорически!! нельзя.

В этом и заключается особенность такого типа связи. Для решения этого кейса, необходимо создать отдельную таблицу. В новой таблице нужно прописать все варианты соединений строк при типе связи - “многие со многими”:

ID	ID Отдел	ID Сотрудники
1	1	1
2	3	2
3	2	3
4	3	3
5	1	4

Полным запрос сразу к 2 таблицам (“Сотрудники” и “Отдел”), нужно делать через 3 таблицу:

```
SELECT
*
  FROM Новая as N

JOIN Сотрудники as P on N.ID_Сотрудники=P.ID
JOIN Отдел as R on N.ID_Отдел=R.ID
```

ID	ID Отдел	ID Сотрудники	ID	Name	ID Отдел	ID	Name
1	1	1	1	Иванов	1	1	IT
2	3	2	2	Петров	3	3	QA
3	2	3	3	Сидоров	2	2	Presale
4	3	3	3	Сидоров	3	3	QA
5	1	4	4	Валуев	1	1	IT

Как видите вероятность задвоения при этом типе возрастает еще больше.

Операции вертикального соединения

Немного теории:

В MS SQL реализованы следующие виды вертикального объединения:

Операция	**Описание**
UNION ALL	В результат включаются все строки из обоих наборов. (A+B)
UNION	В результат включаются только уникальные строки двух наборов. DISTINCT(A+B)
EXCEPT	В результат попадают уникальные строки верхнего набора, которые отсутствуют в нижнем наборе. Разница 2-х множеств. DISTINCT(A-B)
INTERSECT	В результат включаются только уникальные строки,

	присутствующие в обоих наборах. Пересечение 2-х множеств. DISTINCT(A&B)

Отличием горизонтальных соединений от вертикальных является то, что при горизонтальных соединениях мы добавляем колонки из другой таблицы (справа или слева). А при вертикальном соединении, условно говоря, выполняем 2 независимых запроса, а потом строки из 2 запроса добавляем под строками из первого запроса - количество столбцов при этом не меняется.

Такой вид соединения накладывает ряд ограничений:

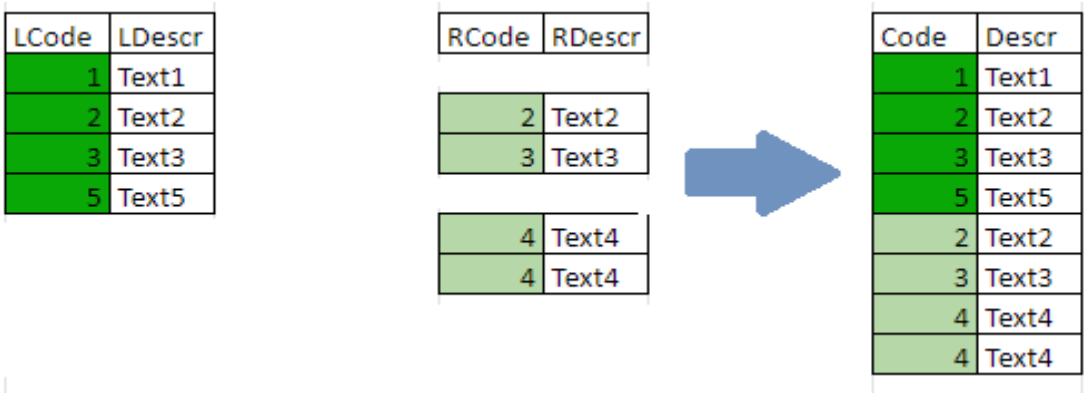
- количество колонок в каждом запросе должно быть одинаковым
- в колонках должны быть совместимыми типы этих колонок, т.е. строка под строкой, число под числом, дата под датой и т.п.

Наиболее часто используемыми запросами являются оператор UNION ALL или UNION. В принципе вертикальные объединения намного проще, чем JOIN-соединения.

UNION ALL

UNION ALL позволяет склеить результаты, полученные разными запросами в один общий результат. При этом в результирующий набор попадают как строки из первого запроса так и строки из второго запроса.

```
Select LCode, LDescr
From LeftTable
Union all
Select RCode, RDescr
From RightTable
```



Пример:

```
SELECT
Color as 'Цвет'
,ReorderPoint as 'Точка заказа'
,avg (ListPrice) as 'Средняя цена'
,COUNT (ListPrice) as 'Количество разных продуктов в этом цвете'
```

```
FROM Production.Product
Where Color = 'BLACK' and ReorderPoint = 3
group by Color,ReorderPoint -- выбираем данные по товарам с черным цветом из точки заказов №3

UNION ALL    -- объединяем результаты

SELECT
Color as 'Цвет'
,ReorderPoint as 'Точка заказа'
,avg (ListPrice) as 'Средняя цена'
,COUNT (ListPrice) as 'Количество разных продуктов в этом цвете'
FROM Production.Product
Where Color = 'silver' and ReorderPoint = 75
group by Color,ReorderPoint -- выбираем данные по товарам с серебряным цветом из  точки заказов №75
```

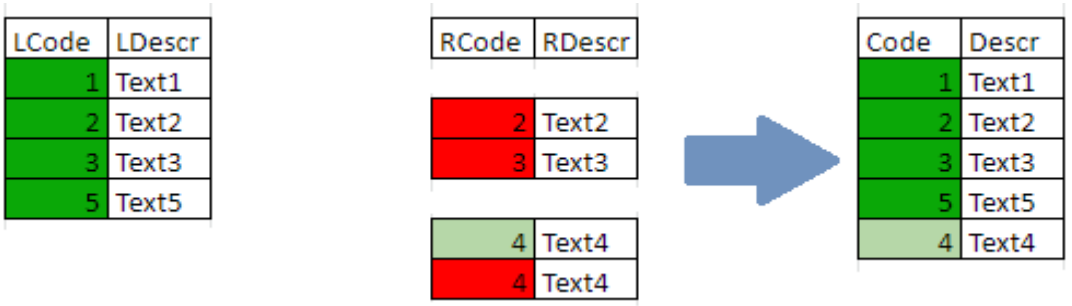
Вот еще один пример:

```
SELECT
Color as 'Цвет'
,ReorderPoint as 'Точка заказа'
,ListPrice as 'Цена'
FROM Production.Product
Where ListPrice between  10 and 100 -- получаем товары с ценой от 10 до 100

UNION ALL    -- объединяем результат

SELECT
Color as 'Цвет'
,ReorderPoint as 'Точка заказа'
,ListPrice as 'Цена'
FROM Production.Product
Where ListPrice between  50 and 150 -- получаем товары с ценой от 50 до 150
```

Следующий оператор - **UNION**

По сути UNION можно представить, как UNION ALL, к которому применена операция DISTINCT: склеивает результаты, и выводит только уникальные значения полученные разными запросами в один общий результат.

```
Select LCode, LDescr
From LeftTable
Union
Select RCode, RDescr
From RightTable
```



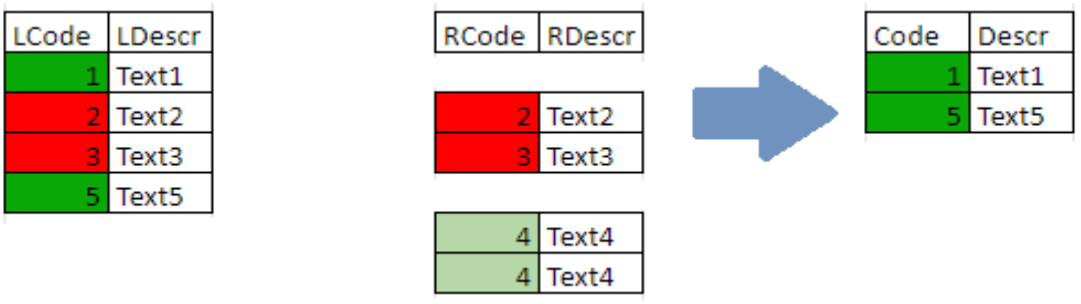
Пример:

```
SELECT
Color as 'Цвет'
,ReorderPoint as 'Точка заказа'
,ListPrice as 'Цена'
FROM Production.Product
Where ListPrice between  10 and 100 -- получаем товары с ценой от 10 до 100

UNION    -- объединяем результат и выводим только уникальные строки

SELECT
Color as 'Цвет'
,ReorderPoint as 'Точка заказа'
,ListPrice as 'Цена'
FROM Production.Product
Where ListPrice between  50 and 150 -- получаем товары с ценой от 50 до 150
```

EXCEPT

Оператор EXCEPT возвращает уникальные строки из первого запроса, которые не выводятся во втором запросе.

```
Select LCode, LDescr
From LeftTable
EXCEPT
Select RCode, RDescr
From RightTable
```



Пример:

```
SELECT
Color as 'Цвет'
,ReorderPoint as 'Точка заказа'
,ListPrice as 'Цена'
FROM Production.Product
Where ListPrice between  10 and 100 -- получаем товары с ценой от 10 до 100
```

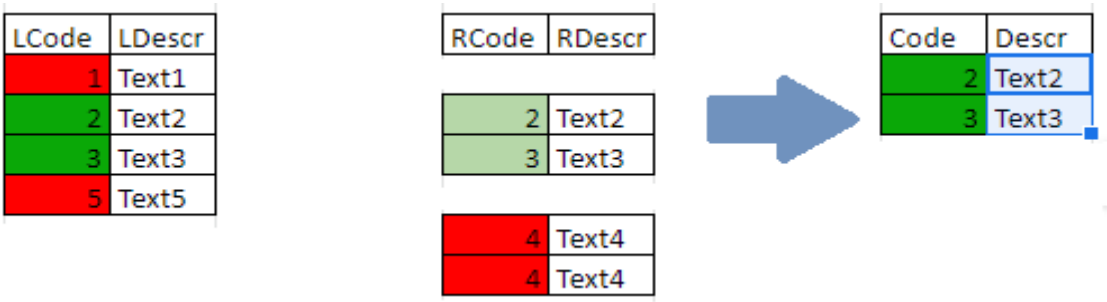
EXCEPT -- объединяем результат и выводим только строки с ценой от 10 до 100 которые не были получены 2 запросом. Фактически товары с ценой с 10 до 50(не включая 50)

```
SELECT
Color as 'Цвет'
,ReorderPoint as 'Точка заказа'
,ListPrice as 'Цена'
FROM Production.Product
Where ListPrice between  50 and 150 -- получаем товары с ценой от 50 до 150
```

INTERSECT

Оператор INTERSECT возвращает уникальные строки, выводимые и первым, и вторым запросом.

```
Select LCode, LDescr
From LeftTable
INTERSECT
Select RCode, RDescr
From RightTable
```



Пример:

```
SELECT
Color as 'Цвет'
,ReorderPoint as 'Точка заказа'
,ListPrice as 'Цена'
FROM Production.Product
Where ListPrice between  10 and 100 -- получаем товары с ценой от 10 до 100
```

INTERSECT -- объединяем результат и выводим только строки которые были и в первом и во втором запросе. Фактически товары с ценой с 50 до 100.

```
SELECT
Color as 'Цвет'
,ReorderPoint as 'Точка заказа'
,ListPrice as 'Цена'
FROM Production.Product
Where ListPrice between  50 and 150 -- получаем товары с ценой от 50 до 150
```

Подзапросы

Переходим к заключительной части разбора инструкции Select, а именно к подзапросам.

Подзапросы были специально оставлены в последнюю очередь, т.к. прежде чем их использовать, нужно научиться правильно строить запросы. К тому же в некоторых случаях можно вообще избежать использования подзапросов и обойтись базовыми конструкциями.

Ранее уже использовали подзапросы в блоке FROM.

```
SELECT *
FROM (
SELECT distinct
ListPrice
FROM  Production.Product) as qwe  -- сначала выполняется запрос выборки уникальных значений. Потом результату этой выборки мы присваиваем имя, т.е. создаем производную таблицу.
```

Тут результат, возвращаемый подзапросом по сути играет роль новой таблицы.

Но все равно, давайте на простом примере разберем когда подзапрос может быть нам полезен.

Представим ситуацию что у нас есть 2 таблицы.

1.
```
 SELECT *
 FROM Production.ProductReview
```

2.
```
 SELECT *
 FROM Production.Product
```

Из первой таблицы нам известно имя человека который оставил отзыв - David. Нам нужно определить наименование товара к которому оставлен этот отзыв.

Можно пойти несколькими путями -
1. Это объединение через JOIN -- это уже умеем.
2. Можем выполнить запрос

```
SELECT
ProductID
FROM Production.ProductReview
WHERE ReviewerName='David'
```

Так мы определяем ID товара - 937
Теперь в таблице Production.Product с помощью этого ID находим имя товара:

```
SELECT
Name
FROM Production.Product
WHERE ProductID=937  -- результат HL Mountain Pedal
```

3. А можно написать запрос с использованием вложенного запроса (подзапроса), объединив эти 2 запроса:

```
SELECT
Name
FROM Production.Product
WHERE ProductID=(SELECT
ProductID
FROM Production.ProductReview
WHERE ReviewerName='David') -- результат HL Mountain Pedal
```

Третий вариант - это простой вложенный запрос в блоке WHERE.
Немного теории:
Вложенный запрос — это запрос, который используется внутри инструкции SELECT, INSERT, UPDATE или DELETE или внутри другого вложенного запроса. Подзапрос может быть использован везде, где разрешены выражения.
Вложенный запрос по-другому называют внутренним запросом или внутренней операцией выбора, в то время как инструкцию, содержащую вложенный запрос, называют внешним запросом или внешней операцией выбора.
В языке MS SQL обычно не бывает разницы в производительности между подзапросом и версией без подзапроса, например с использованием JOIN. Однако в некоторых случаях, когда проверяется существование, соединения по JOIN показывают лучшую производительность.
В противном случае для устранения дубликатов вложенный запрос должен обрабатываться для получения каждого результата внешнего запроса. В таких случаях метод работы соединений дает лучшие результаты.
То есть иными словами - если есть возможность сделать запрос без вложенного с использованием JOIN, делайте именно так. Но иногда без подзапросов не обойтись.
Вложенный во внешнюю инструкцию SELECT запрос имеет обычный запрос:
- SELECT,
- обычное предложение FROM,
- необязательное предложение WHERE,
- необязательное предложение GROUP BY,

- необязательное предложение HAVING.

То есть по факту практически ничем не отличается от обычной инструкции SELECT.

Запрос SELECT вложенного запроса всегда заключен в скобки.

Существуют независимые и связанные подзапросы или по другому простые и связанные подзапросы.

Список выбора вложенного запроса, начинающийся с оператора сравнения, может включать только одно выражение или имя столбца.

Разберем поподробнее 3 последних предложения.

Когды будете прописывать подзапрос в любом блоке SELECT, он обязательно должен быть заключен в скобки.

Существуют простые (независимые) подзапросы - именно такой пример мы рассматривали выше. При выполнении таких запросов, внешний запрос никак не влияет на подзапрос, то есть он независим. И только его результат оказывает влияние на внешний запрос - нашли ID и подали его в условие внешнего запроса.

Существуют связанные запросы - при такой конструкции выполнение результат в подзапросе зависит от выполнения каких то операций в внешнем запросе, после чего результат подзапроса подается во внешний запрос с целью формирования итогового результата.

Или другими словами: В SQL можно создавать подзапросы со ссылкой на таблицу из внешнего запроса. В этом случае подзапрос выполняется многократно, по одному разу для каждой строки таблицы из внешнего запроса.

Если внешний запрос возвращает относительно небольшое число строк, то связанный подзапрос будет работать быстрее несвязанного.

Если подзапрос возвращает большое число строк, то связанный запрос выполняется медленнее несвязанного.

Звучит очень страшно и непонятно поэтому разберем это на примере:

Воспользуемся опять нашими таблицами

1.
```
SELECT *
FROM Production.ProductSubcategory
```

2.
```
SELECT *
FROM Production.Product
```

и напишем связанные запрос который по всем подкатегориям товара подсчитает среднюю цену.

```
SELECT
S.ProductSubcategoryID
,S.Name
, (SELECT avg(P.ListPrice)
FROM Production.Product as P
WHERE P.ProductSubcategoryID=S.ProductSubcategoryID) as 'средняя цена'
 FROM Production.ProductSubcategory as s
```

Как работает механизм такого запроса - сначала формируется таблица из FROM Production.ProductSubcategory as s, после чего результат построчно подается в подзапрос, где по условию равенства ID подсчитывается результат и он уже подается в разбивке на подкатегории во внешний запрос.

Хочу обратить внимание на очень частые ошибки при использовании подзапросов в блоке SELECT. Попробуйте выполнить такой запрос:

```
SELECT
S.ProductSubcategoryID
,S.Name
, (SELECT avg(P.ListPrice) , MAX(P.ListPrice)
 FROM Production.Product as P
  WHERE P.ProductSubcategoryID=S.ProductSubcategoryID) as 'средняя цена'
   FROM Production.ProductSubcategory as s
```

- будет получена ошибка, что в строчке выбора должно выводиться одно значение.

То есть не может быть нескольких столбцов в одном подзапросе. Чтобы получить нужный нам результат нужно MAX(P.ListPrice) вынести в отдельный подзапрос.

```
SELECT
S.ProductSubcategoryID
,S.Name
, (SELECT avg(P.ListPrice)
 FROM Production.Product as P
  WHERE P.ProductSubcategoryID=S.ProductSubcategoryID) as 'средняя цена'
, (SELECT MAX(P.ListPrice)
 FROM Production.Product as P
     WHERE  P.ProductSubcategoryID=S.ProductSubcategoryID)  as  'максимальная
цена'
   FROM Production.ProductSubcategory as s
```

Разберем еще один момент. Для этого изменим запрос:

```
SELECT
 Name
  FROM Production.Product
  WHERE ProductID=(SELECT
 ProductID
  FROM Production.ProductReview
  WHERE ReviewerName='David')
```

так чтобы были найден все наименования продукта, по которым отзывы поступили “2013.11.15”

```
SELECT
 Name
  FROM Production.Product
```

```
 WHERE ProductID=(SELECT
ProductID
 FROM Production.ProductReview
 WHERE CONVERT(date, ReviewDate, 104)='2013.11.15')
```

Получаем ошибку “Вложенный запрос вернул больше одного значения”. Если выполнить запрос:

```
SELECT
 ProductID
 FROM Production.ProductReview
 WHERE CONVERT(date, ReviewDate, 104)='2013.11.15'
```

- то увидим, что получено 2 ID: 937 и 798

То есть, если в запросе используются операторы сравнения =, !=, <, <=, >, >= для взаимодействия с подзапросом, то соответственно подзапрос должен возвращать максимум одну строку и одно значение.

Но если нам все же надо получить итоговое значение по 2 строкам? То тут нам на помощь придет уже известный оператор IN:

```
SELECT
 Name
 FROM Production.Product
 WHERE ProductID IN    -- заменяем оператор ‘=’ на ‘IN’
(SELECT
ProductID
 FROM Production.ProductReview
 WHERE CONVERT(date, ReviewDate, 104)='2013.11.15')
```

На этом заканчиваем базовое изучение инструкции SELECT. Помимо изучения базовых конструкций для самостоятельного изучения можно порекомендовать изучить:

1. Конструкция WITH
2. APPLY
3. PIVOT, UNPIVOT
4. Предложение OVER, которое дает возможность использовать:
   - Агрегатные функции (COUNT, SUM, MIN, MAX, AVG) без использования GROUP BY;
   - Ранжирующие функции: ROW_NUMBER(),RANK() и DENSE_RANK();
   - Аналитические функции: LAG() и LEAD(),FIRST_VALUE() и LAST_VALUE();
5. Изучить конструкции позволяющие вычислить под итоги: GROUP BY ROLLUP(…), GROUP BY GROUPING SETS(…) и т. д. А также вспомогательные функции используемые для этих целей: GROUPING_ID() и GROUPING();

Кроме того, без закрепления практикой невозможно обойтись для уверенного написания запросов, мало понимать теорию, нужно еще много практиковаться.

Для этой цели можно использовать сайт «SQL-EX.RU – Практическое владение языком SQL» который содержит несколько демонстрационных баз данных и предоставляет возможность попрактиковаться в написании самых каверзных запросов, начиная с решения самых простых задач. Там тоже есть много учебного материала по языку SQL.

Операторы манипуляции данными

Здесь мы в общих чертах рассмотрим работу с операторами модификации данных:

- INSERT – вставка новых данных
- UPDATE – обновление данных
- DELETE – удаление данных

Операции модификации данных очень сильно связаны с конструкциями оператора SELECT, т.к. по сути выборка модифицируемых данных идет при помощи них. Поэтому для понимания данного материала, важное место имеет уверенное владение конструкциями оператора SELECT.

Данная часть будет больше обзорная. Здесь будут описаны только основные формы операторов модификации данных. Полную информацию при необходимости можно будет изучить самостоятельно.

Прямая модификация информации в РБД требует от человека большой ответственности, так как может привести к необратимым последствиям. Однако данные операторы могут быть полезны допустим при необходимости установки параметра, при объемном тестировании или при удалении ненужной информации.

Если вы смогли освоить оператор SELECT, то думаю, и операторы модификации вам будут под силу, т.к. после оператора SELECT здесь нет ничего сверхсложного, и по большей части должно восприниматься на интуитивном уровне.

Но порой сложность представляют не сами операторы модификации, а то что они должны выполняться группами, в рамках одной транзакции, т.е. когда дополнительно нужно учитывать целостность данных

INSERT – вставка новых данных

Данный оператор имеет 2 основные формы:

1. INSERT INTO таблица(перечень_полей) VALUES(перечень_значений) – вставка в таблицу новой строки значения полей которой формируются из перечисленных значений

2. INSERT INTO таблица(перечень_полей) SELECT перечень_значений FROM … – вставка в таблицу новых строк, значения которых формируются из значений строк возвращенных запросом.

В диалекте MS SQL слово INTO можно отпускать.

К тому же стоит отметить, что первая форма в диалекте MS SQL с версии 2008, позволяет вставить в таблицу сразу несколько строк:

```
INSERT таблица(перечень_полей) VALUES
(перечень_значений1),
(перечень_значений2),
…
(перечень_значенийN)
```

Для рассмотрения INSERT на практике воспользуемся таблицей ProductReview:

```
SELECT  *
 FROM Production.ProductReview
```

Нам поставлена задача добавить новый отзыв по клиенту. Обратите внимание на количество колонок в этой таблице - их 8:

```
    ProductReviewID
     ,ProductID
     ,ReviewerName
     ,ReviewDate
     ,EmailAddress
     ,Rating
     ,Comments
     ,ModifiedDate
```

Теперь начинаем прописывать наш запрос:

```
 INSERT  --INTO  можно опустить
Production.ProductReview  -- наименование таблицы куда хотим вставить
      (
     ProductID  -- колонку ProductReviewID пропускаю так как это первичный ключ и он сам укажет новое значение
```

```
        ,ReviewerName   -- далее перечисляем все остальные поля, можно их скопировать не теряя запятых
    ,ReviewDate
    ,EmailAddress
    ,Rating
    ,Comments
    ,ModifiedDate)
    Values      -- тут мы можем начинаем прописывать, а какие собственно говоря значения мы хотим вставить
    (
     709     -- это значение соответствует ProductID и товару на который будет добавлен отзыв
    ,'John Smith'
    ,'20190901'
    ,'john@fourthcoffee.com'
    ,3
    ,'Maybe its just because I'
    ,'20190901')     -- количество колонок указанное в первой части запроса должно совпадать тому что мы указываем
```

Запрос успешно выполнен, для того чтобы посмотреть его результат еще раз воспользуемся запросом

```
SELECT   *
 FROM Production.ProductReview
```

Как видите ничего сложного.

Несколько заметок про INSERT:

- Порядок перечисления полей не имеет значения, вы можете написать как угодно. Здесь важно только то, чтобы он совпадал с порядком значений, которые вы перечисляете в скобках после ключевого слова VALUES.

- Так же важно, чтобы при вставке были заданы значения для всех обязательных полей, которые помечены в таблице как NOT NULL.

- Можно не указывать поля у которых была указана опция IDENTITY (наш случай с первичным ключом) или же поля у которых было задано значение по умолчанию при помощи DEFAULT, т.к. в качестве их значения подставится либо значение из счетчика, либо значение, указанное по умолчанию. Такие вставки мы уже делали в первой части.

- В случаях, когда значение поля со счетчиком нужно задать явно используйте опцию IDENTITY_INSERT.

На как видно из запроса, были изменены всего 3 колонки, а все остальные были скопированы из 1 строки.

В этом случае для того чтобы не переписывать все значения руками или для того чтобы добавить новую запись идентичную из ранее уже имеющихся, нам на помощь приходит 2 форма INSERT.

Она позволяет использовать SELECT внутри конструкции INSERT:

```
 INSERT   Production.ProductReview
(
    ProductID
    ,ReviewerName
    ,ReviewDate
    ,EmailAddress
    ,Rating
    ,Comments
    ,ModifiedDate)  -- в первой части запроса ничего не меняется
SELECT       -- а вот тут мы указываем простой запрос SELECT с выборкой тех
  колонок что были в в первой части (копипаст никто не отменял)
 ProductID
    ,ReviewerName
    ,ReviewDate
    ,EmailAddress
    ,Rating
    ,Comments
    ,ModifiedDate
 FROM Production.ProductReview
 WHERE ProductReviewID=1    --  и условие что все данные будут идентичны 1
  строке.
```

Запрос, написанный выше, позволяет создать строку идентичную той, что уже была, а если нам необходимо что-то изменить? То вместо наименование столбца указываем тот параметр которой нам нужен:

```
 INSERT   Production.ProductReview
(
    ProductID
    ,ReviewerName
    ,ReviewDate
    ,EmailAddress
    ,Rating
    ,Comments
    ,ModifiedDate)
SELECT
 ProductID
    ,ReviewerName
    ,ReviewDate
    ,EmailAddress
    ,Rating
    ,Comments
    ,'20190902'  -- например дату
 FROM Production.ProductReview
 WHERE ProductReviewID=1
```

UPDATE – обновление данных

Данный оператор в MS SQL имеет 2 формы:

1. UPDATE таблица SET … WHERE условие_выборки – обновлении строк таблицы, для которых выполняется условие_выборки. Если предложение WHERE не указано, то будут обновлены все строки. Это можно сказать классическая форма оператора UPDATE.

2. UPDATE псевдоним SET … FROM … – обновление данных таблицы участвующей в предложении FROM, которая задана указанным псевдонимом. Конечно, здесь можно и не использовать псевдонимов, используя вместо них имена таблиц, но с псевдонимом удобнее.

Давайте возьмем и в таблице ProductReview изменим оценку по отзывы которая была проставлена раньше, столбец Rating.

```
SELECT  *
 FROM Production.ProductReview
```

Для этого воспользуемся первой формой оператора:

```
 update  Production.ProductReview  --задаем таблицу в которой будем изменять данные
 set Rating=2 -- прописываем какой столбец должен быть изменен и какое в нем должно быть значение
 where ProductID=709 -- указываем условие чтобы определить- в каких  строчках в будет изменено значение по столбцу Rating
```

Выполним запрос SELECT чтобы посмотреть на получившийся результат:

```
SELECT  *
 FROM Production.ProductReview
```

И обнаружим, что изменения коснулись не одной строчки, а сразу всех где ProductID=709. Это легко перепроверить переписав запрос

```
 update Production.ProductReview  set Rating=2  where ProductID=709
в запрос
SELECT  Rating  FROM Production.ProductReview where ProductID=709
```

То есть перед выполнением нашего запроса мы не перепроверили - а какие строчки затронет наш запрос.

Именно в этом большая опасность данного запроса. А если вдруг не будет указано условие как в запросе ниже, то может случиться катастрофа:

```
update Production.ProductReview
set Rating=2
```

Выполните этот запрос и оцените, что случилось в базе данных.

Как говорилось выше если предложение WHERE не указано, то будут обновлены все строки.

Именно поэтому перед каждым выполнением запроса на обновление данных необходимо переделывать запрос в SELECT, чтобы убедиться какие же именно строки будут затронуты.

Попробуйте самостоятельно изменить запрос таким образом чтобы он изменил всего одну строчку.

Рассмотрим теперь вторую форму запроса UPDATE, причем не простую с указанием таблицы, а чуть более усложненную с подзапросом.

Так например, мы хотим изменить не просто оценку, но еще и дату изменения - ModifiedDate.

Для этого выполним следующие действия:

1. Перепишем запрос

```
SELECT  Rating  FROM Production.ProductReview where ProductID=709
```

таким образом, чтобы у нас остались только строки и столбцы по которым требуются изменения:

```
SELECT
ProductReviewID
,Rating
,ModifiedDate
FROM Production.ProductReview
where ProductReviewID=1
or ReviewerName='David'
```

В результате у нас получаются 2 строчки и 3 столбца

2. А теперь этот запрос включаем в качестве подзапроса во вторую форму UPDATE:

```
update Review  --задаем имя таблицы, псевдоним которой указан в FROM
set
Rating=4
,ModifiedDate ='20190901' -- прописываем какие столбцы будут изменены
 FROM
(
    SELECT
        ProductReviewID
,Rating
,ModifiedDate
     FROM Production.ProductReview
     where ProductReviewID=1 or ReviewerName='David'
) as Review-- данные из подзапроса указываем в качестве таблицы и присваиваем им имя
```

Смотрим полученный результат и видим что у нас все получилось.

DELETE – удаление данных

Принцип работы DELETE похож на принцип работы UPDATE, и так же в MS SQL можно использовать 2 формы:

1. DELETE таблица WHERE условие_выборки – удаление строк таблицы, для которых выполняется условие_выборки. Если предложение WHERE не указано, то будут удалены все строки. Это можно сказать классическая форма оператора DELETE (только в некоторых СУБД нужно писать DELETE FROM таблица WHERE условие_выборки).

2. DELETE псевдоним FROM … – удаление данных таблицы участвующей в предложения FROM, которая задана указанным псевдонимом. Конечно, здесь можно и не использовать псевдонимов, используя вместо них имена таблиц, но с псевдонимом удобнее.

Используя 1 форму удалим одну из строк созданных ранее с помощью INSERT:

```
Delete Production.ProductReview -- задаем имя таблицы
WHERE ProductReviewID=6 --задаем номер строки которая будет удалена
```

Выполняем и проверяем результат, что у нас все получилось.

При использовании DELETE можно (нужно) также как и при UPDATE проверять на запросе SELECT какие строки будут удалены.

Воспользуемся 2 формой DELETE, чтобы удалить последние 2 добавленные строчки:

1.

```
SELECT
*
FROM Production.ProductReview
where ProductReviewID=5
or ProductReviewID=7    -- разница от запроса в UPDATE заключается в том чтобы
  по итогу выборки должны попадать полные строчки, а не отдельные  столбцы
```

2.

```
Delete Review
FROM
(SELECT
*
FROM Production.ProductReview
where ProductReviewID=5
or ProductReviewID=7
```

```
) as Review    -- при необходимости после присвоения подзапросу псевдонима, можно указывать дополнительные условия WHERE  при необходимости.
```

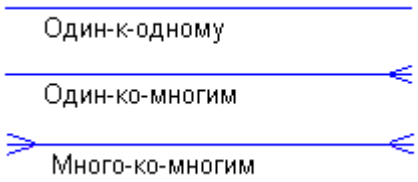
INSERT, UPDATE и DELETE очень легко понять интуитивно, когда умеешь пользоваться конструкциями оператора SELECT. Поэтому рассказ о операторе SELECT растянулся на гораздо больший объем, чем было написано об операторах модификации.

Диаграмма базы данных

Диаграмма базы данных (или ER-диаграмма) — это графическое представление таблиц БД с визуальным отображением связей между таблицами по внешним ключам (foreign key). По большей части это разновидность блок-схемы, где показано, как разные «сущности» (люди, объекты, концепции и так далее) связаны между собой внутри системы. ER-диаграммы чаще всего применяются для проектирования и отладки реляционных баз данных.
В сфере разработки программного обеспечения ER-диаграмма, как правило, служит первым шагом в определении требований проекта по созданию информационных систем. На дальнейших этапах работы ER-диаграммы также применяются для моделирования конкретных баз данных.
Кроме того, наличие ER-диаграммы помогает быстрее разобраться в связях между таблицами. Как та или иная таблица связана с другой таблицей, через какие столбцы осуществляется эта связь, а также какой тип связи используется при соединении таблиц (один-к-одному, один-ко- многим, многие-ко-многим).

При построении ER-диаграммы используются различные графические элементы. Они признаны показать, какой тип связи используется при построении. Мы не будем подробно разбирать эти элементы остановимся

только на трех(обозначения могут немного различаться в разных инструментах построения диаграмма, но они все равно интуитивно различаются):



Связь типа один-к-одному означает, что один экземпляр первой сущности (левой) связан с одним экземпляром второй сущности (правой). Связь один-к-одному чаще всего свидетельствует о том, что на самом деле мы имеем всего одну сущность, разделенную на две.

Отдел	
ID	Name
1	IT
2	Presale
3	QA

Сотрудники	
ID	Name
1	Иванов
2	Петров
3	Сидоров

Связь типа один-ко-многим означает, что один экземпляр первой сущности (левой) связан с несколькими экземплярами второй сущности (правой). Это наиболее часто используемый тип связи. Левая сущность (со стороны "один") называется родительской, правая (со стороны "много") - дочерней.

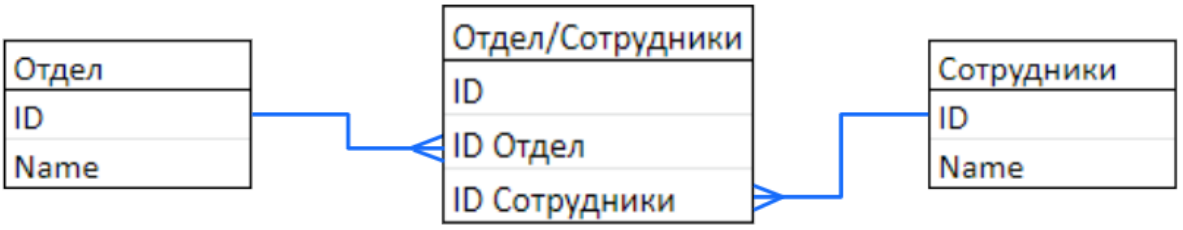
Отдел	
ID	Name
1	IT
2	Presale
3	QA

Сотрудники		
ID	Name	ID Отдел
1	Иванов	1
2	Петров	3
3	Сидоров	2
4	Валуев	1

Связь типа много-ко-многим означает, что каждый экземпляр первой сущности может быть связан с несколькими экземплярами второй сущности, и каждый экземпляр второй сущности может быть связан с несколькими экземплярами первой сущности. Тип связи много-ко-многим является временным типом связи, допустимым на ранних этапах разработки модели. В дальнейшем этот тип связи должен быть заменен двумя связями типа один-ко-многим путем создания промежуточной сущности.

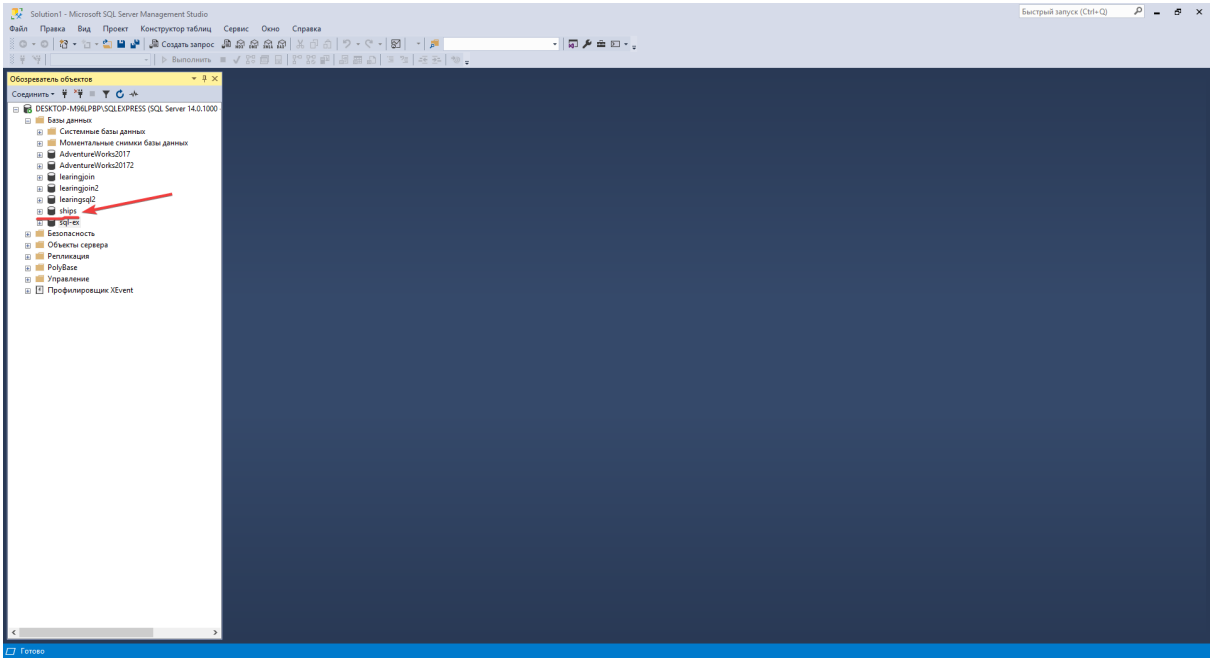
Временный вариант для отображения связи много-ко-многим на проектной диаграмме.

Фактическая реализация связи много-ко-многим через промежуточную таблицу.

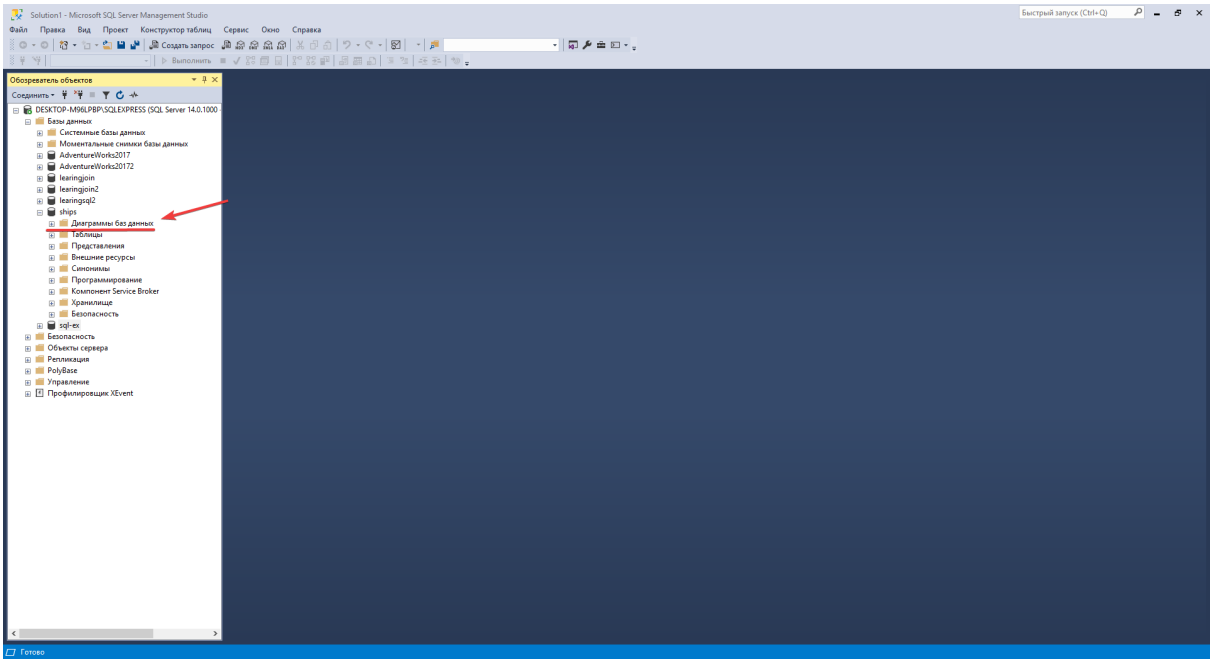


Рассмотрим как можно создать диаграмму в MS SQL, используя клиент Management Studio:

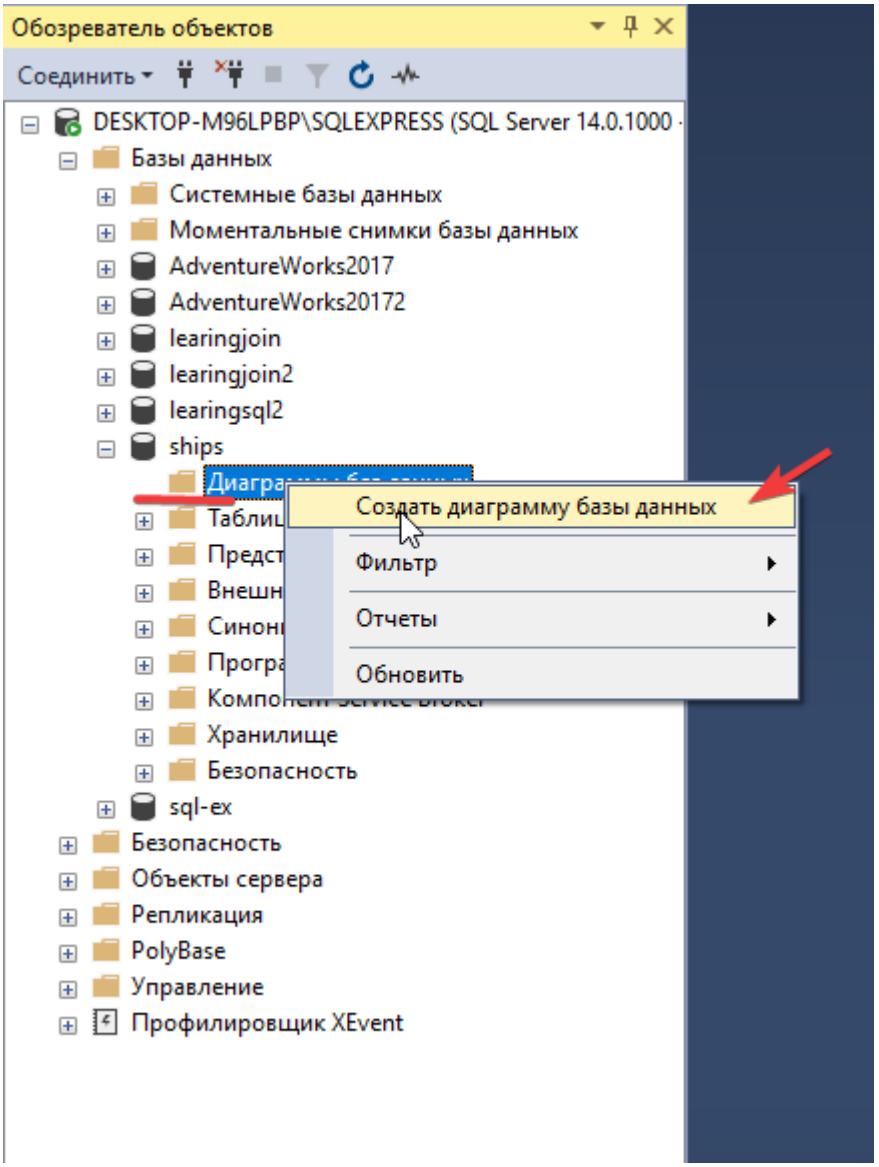
1. Авторизуемся в Management Studio, выбираем базу данных по которой будем строить диаграмму. Например, база данных **ships.**



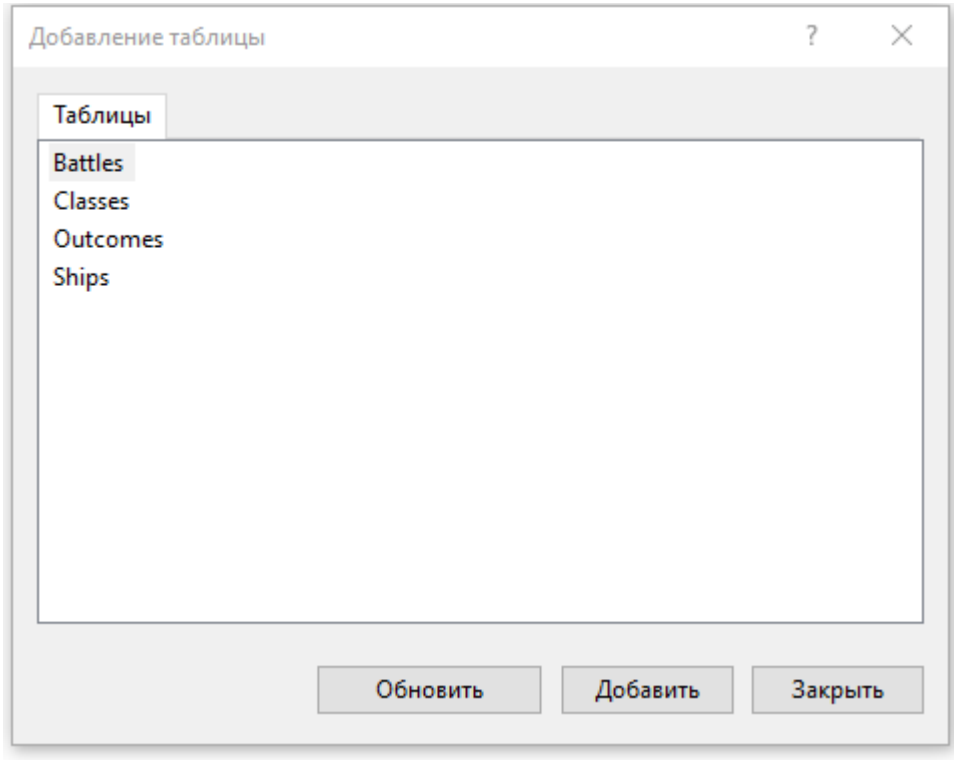
2. Выбираем пункт “Диаграмма баз данных”

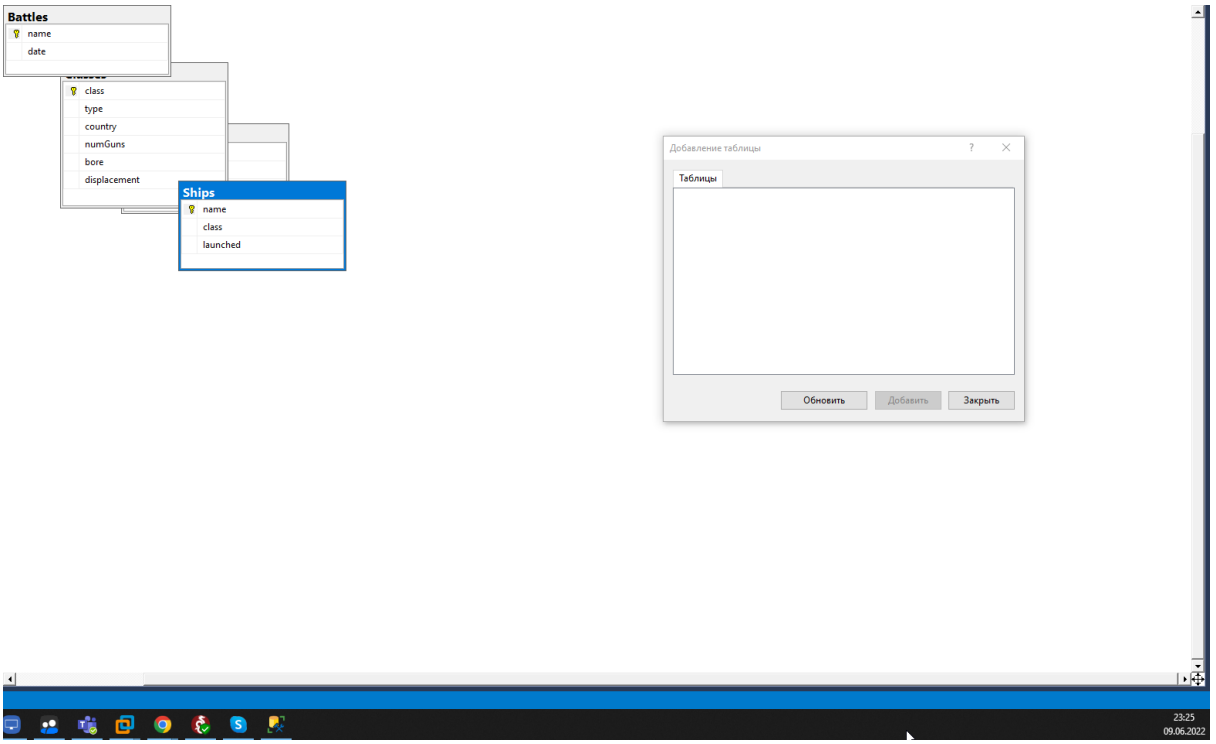


3. Кликаем правой кнопкой мыши по пункту “Диаграмма баз данных” и выбираем пункт “Создать диаграмму базы данных”

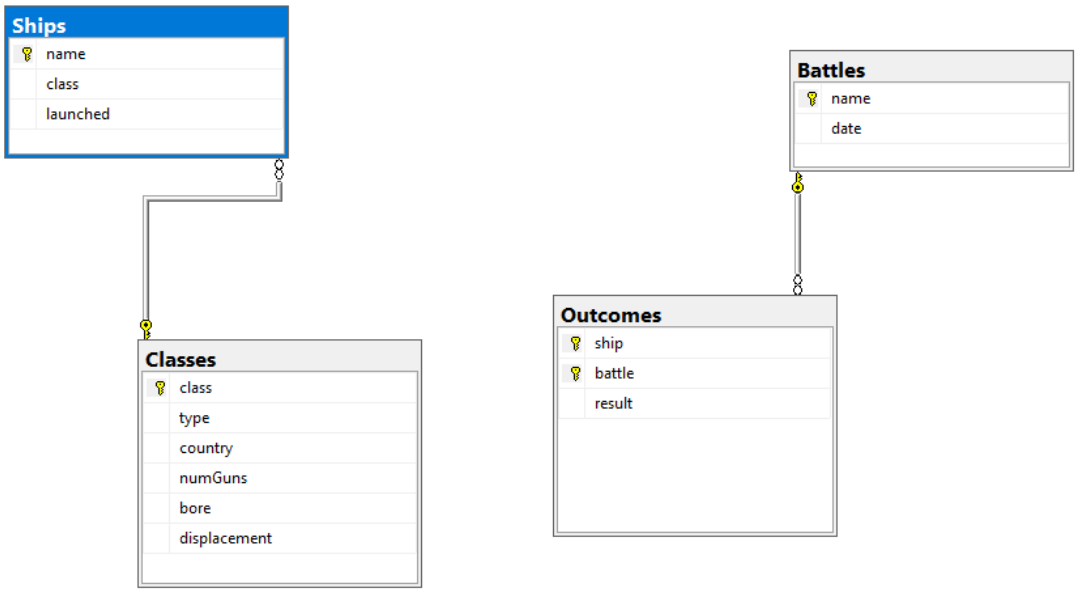


4. Добавляем все таблицы



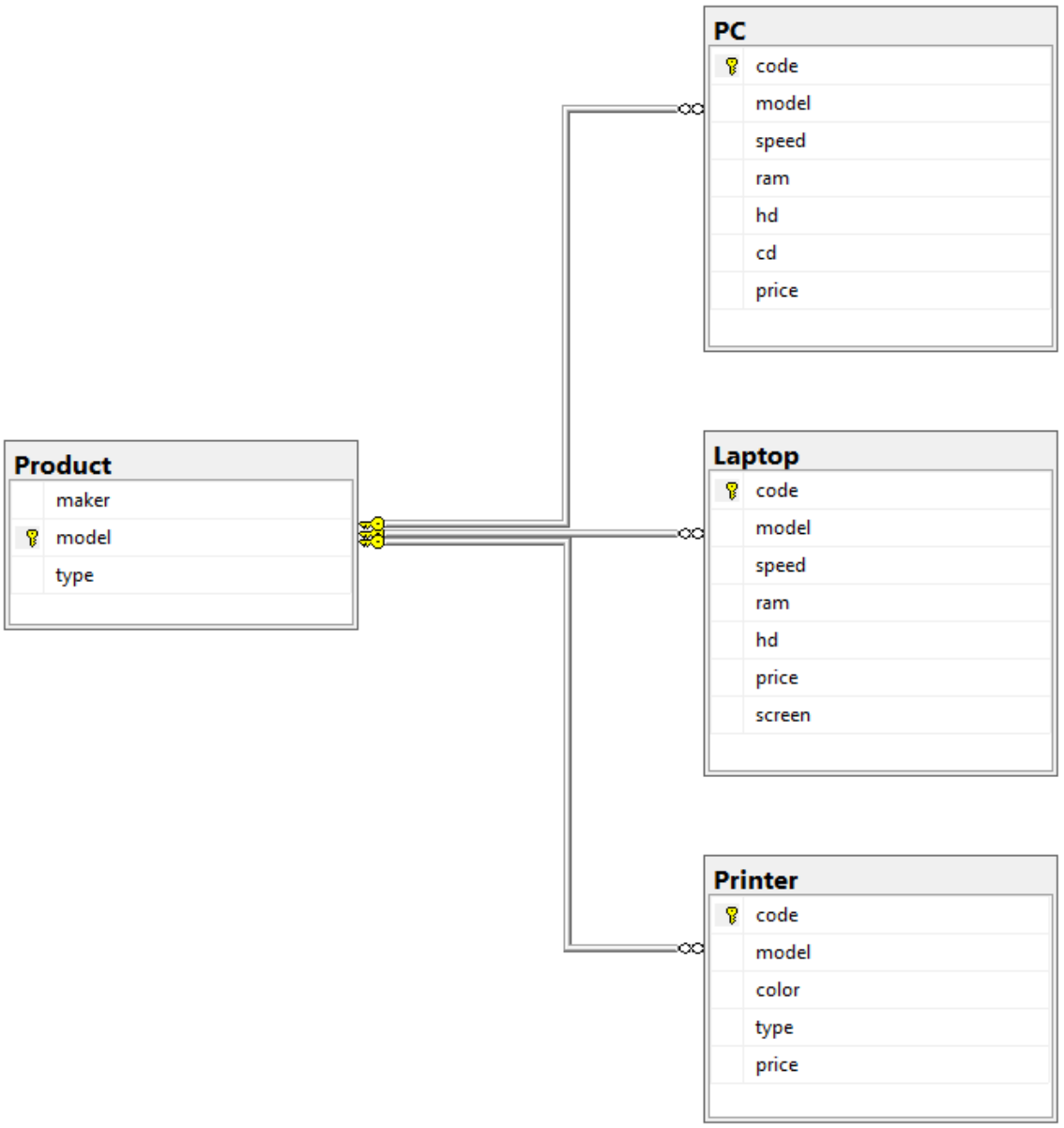


5. Закрываем окно добавления

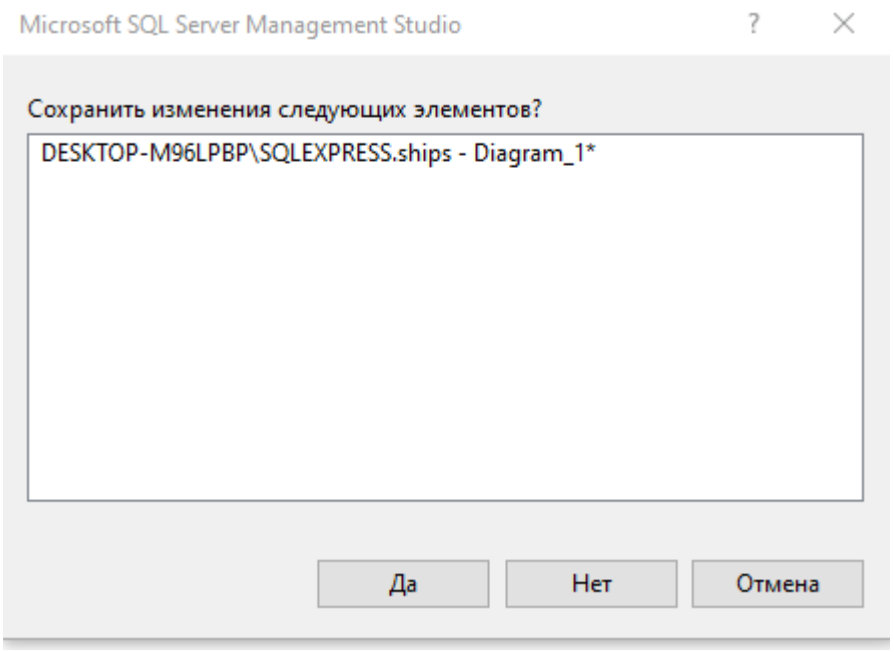


У нас отображаются таблицы со связями между таблицами.

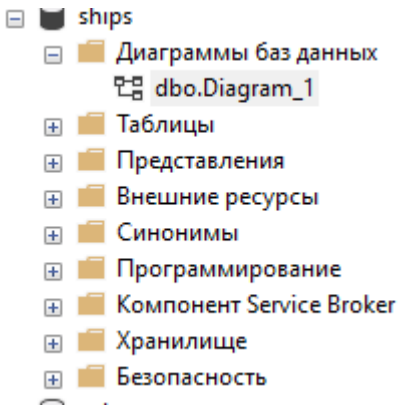
6. После этого необходимо перенести линии соединения между таблицами, так чтобы была связь между конкретными столбцами в разных таблицах. Для примера, вот например диаграмма между таблицами PC, Laptop, Printer и Product. Связь осуществляется между столбцами model.



7. После настройки связей, сохраняем диаграмму



8. Сохраненная диаграмма теперь отображается в пункте “Диаграмма баз данных”

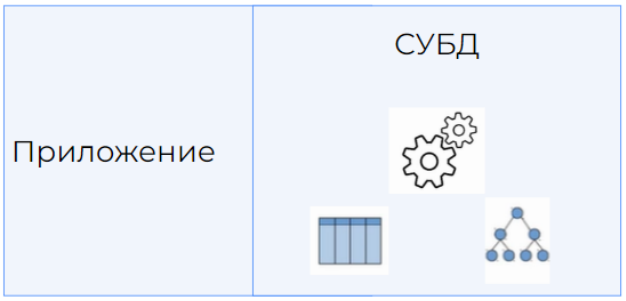


Способы взаимодействия с базой данных

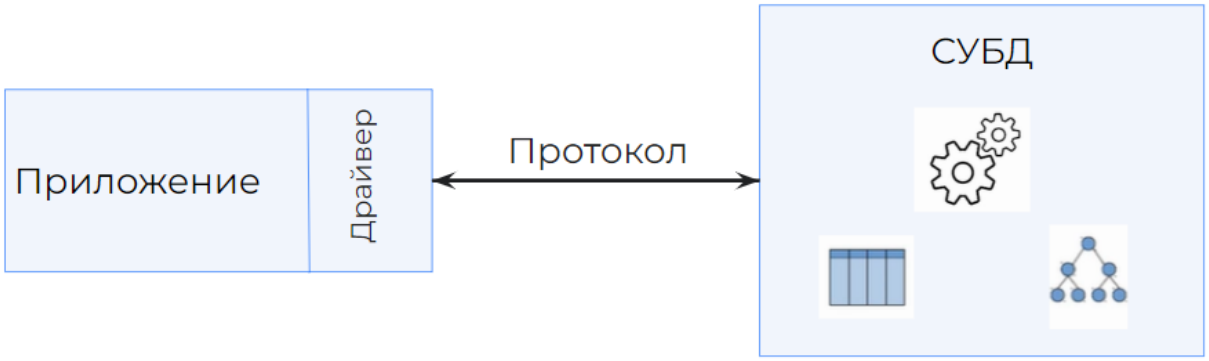
Стоит сказать несколько слов о способах взаимодействия нами с базой данных (СУБД). Для начала небольшая ремарка для лучшего понимания как работает СУБД.

СУБД можно подразделить на 2 актуальные категории:

1. Встраиваемые СУБД



2. Клиент - серверные СУБД



Встраиваемые СУБД интегрируются в приложение и по факту выступаю вместе с ним как единое приложение. Как правило это библиотека, которая подключается в код программы и позволяет использовать функции СУБД прямо внутри программы.

Примером такой СУБД может быть SQLite и большинство браузеров в который встроена данная СУБД.
Главным плюсом таких СУБД является более быстрое взаимодействие, минусом является ограничения по размеру, безопасности и т.д.

Клиент - серверные СУБД в отличие от встраиваемых, можно сказать является отдельным (обособленным) приложением, которое выполняет функция хранения данных. В клиент-серверных СУБД все основные компоненты СУБД выполняются на отдельном сервере, и на нём же хранится база данных. На клиенте находится интерфейсная (клиентская) часть СУБД и выполняется код приложения.

Плюсом такой архитектуры является оптимизации на стороне клиента: всю работу с БД осуществляет сервер, по сети передаётся только обработанный ответ небольшого размера. Недостатком такой архитектуры является зависимость клиентов от сервера . В случае неисправности которого, ни один клиент не сможет работать с БД. Также существенным недостатком клиент-серверных СУБД является необходимость установления прямого соединения между клиентским компьютером и сервером БД.

Существует трёхзвенная клиент-серверная архитектура, которая добавляет дополнительный вспомогательный сервер приложений - прослойку между клиентами и сервером БД, который выполняет код приложения. Это позволяет полностью изолировать клиентов от конкретной БД и вся логика работы с БД ложится на плечи вспомогательного сервера. Трёхзвенная архитектура помимо повышения уровня безопасности позволяет более гибко модернизировать приложения. При модернизации не нужно закачивать обновления в клиенты. Достаточно обновить только сервер приложений.

Клиент- серверные СУБД в настоящее время являются самыми популярными.

Думаю понятно, что если СУБД разделено на 2 части - основную часть и интерфейс, то между ними должен быть организован какой способ взаимодействия, что то вроде API. Роль API в этом случае выполняет **драйвер** и **протокол**.

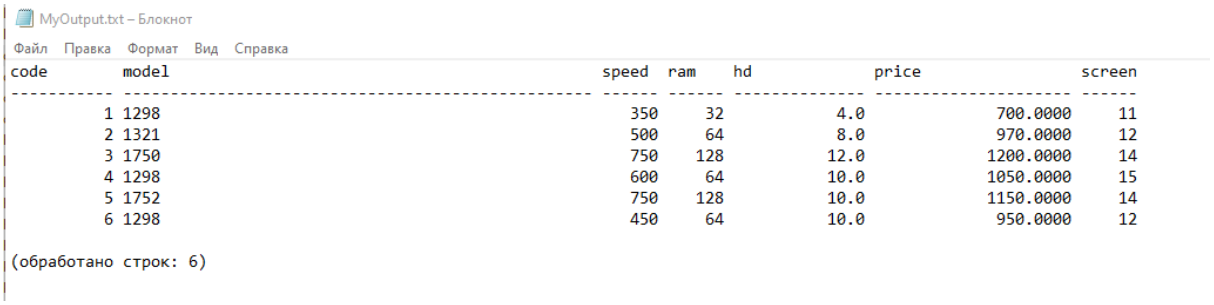
Таким образом для того чтобы нам написать и выполнить запрос к базе данных, мы должны обратиться к основной части СУБД. И выступить в роли обычного клиента.

И тут на первое место выходит вопрос, а каким способом и с помощью чего мы будем обращаться к движку СУБД (основной части).
Условно можно выделить 3 способа взаимодействия:

1. **Через терминал/командную строку.**

То есть через UI (интерфейс взаимодействия), не обладающей графической оболочкой. Так как на всем протяжении блока мы использовали СУБД MSSQL, то и пример подключения приведу именно для этой СУБД:

```
C:\Users\user>sqlcmd -S DESKTOP-[redacted]\SQLEXPRESS -Q "SELECT * FROM [AdventureWorks20172].[dbo].[Laptop]"
 -o C:\1\MyOutput.txt
```



MyOutput.txt – Блокнот

Файл Правка Формат Вид Справка

```
code        model                                              speed  ram    hd             price                screen
----------- -------------------------------------------------- ------ ------ -------------- -------------------- ------
          1 1298                                                 350     32            4.0              700.0000     11
          2 1321                                                 500     64            8.0              970.0000     12
          3 1750                                                 750    128           12.0             1200.0000     14
          4 1298                                                 600     64           10.0             1050.0000     15
          5 1752                                                 750    128           10.0             1150.0000     14
          6 1298                                                 450     64           10.0              950.0000     12

(обработано строк: 6)
```

Так для этого в командной строке Windows вводим:
`sqlcmd` - это служебная программа командной строки для нерегламентированного интерактивного выполнения инструкций и скриптов.

`-S <ComputerName>` - Серверный параметр (-S) определяет экземпляр Microsoft SQL Server, к которому подключается программа sqlcmd. Так как MSSQL развернут у меня на компьютере, то это просто наименование компьютера.

`-Q "SELECT * FROM [AdventureWorks20172].[dbo].[Laptop]"` - Параметр входа (-Q) определяет расположение входных данных для программы sqlcmd. То есть с этого момента начинается инструкция SQL для выполнения.

`-o C:\1\MyOutput.txt` - Параметр выходных данных (-o) определяет файл, в который программа sqlcmd помещает выходные данные. То есть куда будут записаны результаты выполнения запроса.

Некоторые СУБД позволяют выводить результат сразу в командную строку/терминал

Как понятно из описания, данный способ рекомендуется использовать не на постоянной основе, так как для длительного взаимодействия удобнее клиенты с графическим отображением (GUI). Но плюсом подобного способа является скорость, так как не требует времени на загрузку графической части клиента.

2. **Через клиентское программное приложение SQL, разработанное производителем СУБД.**

В нашем примере для MSSQL подобным приложением является SQL Server Management Studio (SSMS).
SSMS является основным инструментом управления базами данных для серверов баз данных SQL Server. Он представляет собой графический пользовательский интерфейс (GUI) и интерфейс сценариев Transact-SQL для управления компонентом ядра базы данных и базами данных.

На всем протяжении изучении блока мы пользовались именно им.

Минусом подобного приложения от разработчиков СУБД, является то, что с помощью этого инструмента можно взаимодействовать только с одним конкретным типом СУБД.

А если у нас сразу несколько баз данных, и все они на разных СУБД? Большое количество однотипных программ для каждой СУБД, мягко говоря неудобно…

3. Мульти-платформенные клиентские приложения

Мульти-платформенные клиентские приложения это инструменты для работы с различными типами баз данных. То есть в рамках данных приложений мы может подключать одновременно нужные базы данных вместе с драйверами (MySQL, PostgreSQL, SQLite, Oracle, DB2, SQL Server, Sybase, MS Access, Teradata, Firebird, Apache Hive, Phoenix, Presto и др.)

Ярким примером подобных программ является программа DBeaver.

На этом изучение блока заканчивается, хотя при желании есть еще много интересной информации, которую можно изучить по SQL. И опять же, на первое место выходит практика.

Вопросы для самоподготовки:

1. Что такое первичный ключ?
2. Что такое внешний ключ?
3. Для чего используется SQL?
4. Что такое СУБД?
5. Назвать основные группы операторов
6. Для чего предназначен оператор Select?
7. Какие блоки оператора оператор Select вы знаете? Для чего они предназначены?
8. Что проверяет предикат BETWEEN?
9. Как с помощью LIKE и части сообщения найти полное сообщение?
10. Можно ли несколько условий по AND (1 условие and 2 условие and 3 условие) заменить оператором IN?Если можно то как?
11. Какие агрегатные функции вы знаете?
12. Какая функция возвращает среднее значение в указанном столбце?
13. Можно ли использовать Group by для сбора статистики?
14. Для чего используются JOIN?
15. Что такое подзапросы и для чего могут пригодиться?