Учебник по языку SQL (DDL, DML) на примере диалекта MS SQL Server

Оглавление

Часть первая
О чем данный учебник

Данный учебник представляет собой что-то типа «штампа моей памяти» по языку SQL (DDL, DML), т.е. это информация, которая накопилась по ходу профессиональной деятельности и постоянно хранится в моей голове. Это для меня достаточный минимум, который применяется при работе с базами данных наиболее часто. Если встает необходимость применять более полные конструкции SQL, то я обычно обращаюсь за помощью в библиотеку MSDN расположенную в интернет. На мой взгляд, удержать все в голове очень сложно, да и нет особой необходимости в этом. Но знать основные конструкции очень полезно, т.к. они применимы практически в таком же виде во многих реляционных базах данных, таких как Oracle, MySQL, Firebird. Отличия в основном состоят в типах данных, которые могут отличаться в деталях. Основных конструкций языка SQL не так много, и при постоянной практике они быстро запоминаются. Например, для создания объектов (таблиц, ограничений, индексов и т.п.) достаточно иметь под рукой текстовый редактор среды (IDE) для работы с базой данных, и нет надобности изучать визуальный инструментарий заточенный для работы с конкретным типом баз данных (MS SQL, Oracle, MySQL, Firebird, …). Это удобно и тем, что весь текст находится перед глазами, и не нужно бегать по многочисленным вкладкам для того чтобы создать, например, индекс или ограничение. При постоянной работе с базой данных, создать, изменить, а особенно пересоздать объект при помощи скриптов получается в разы быстрее, чем если это делать в визуальном режиме. Так же в скриптовом режиме (соответственно, при должной аккуратности), проще задавать и контролировать правила наименования объектов (мое субъективное мнение). К тому же скрипты удобно использовать в случае, когда изменения, делаемые в одной базе данных (например, тестовой), необходимо перенести в таком же виде в другую базу (продуктивную).

Язык SQL подразделяется на несколько частей, здесь я рассмотрю 2 наиболее важные его части:

- DDL – Data Definition Language (язык описания данных)
- DML – Data Manipulation Language (язык манипулирования данными), который содержит следующие конструкции:
  - SELECT – выборка данных
  - INSERT – вставка новых данных
  - UPDATE – обновление данных
  - DELETE – удаление данных
  - MERGE – слияние данных

Т.к. я являюсь практиком, как таковой теории в данном учебнике будет мало, и все конструкции будут объясняться на практических примерах. К тому же я считаю, что язык программирования, а особенно SQL, можно освоить только на практике, самостоятельно пощупав его и поняв, что происходит, когда вы выполняете ту или иную конструкцию.

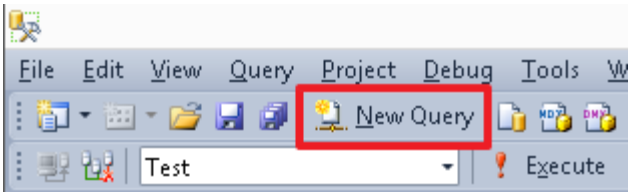
Данный учебник создан по принципу Step by Step, т.е. необходимо читать его последовательно и желательно сразу же выполняя примеры. Но если по ходу у вас возникает потребность узнать о какой-то команде более детально, то используйте конкретный поиск в интернет, например, в библиотеке MSDN.

При написании данного учебника использовалась база данных MS SQL Server версии 2014, для выполнения скриптов я использовал MS SQL Server Management Studio (SSMS).

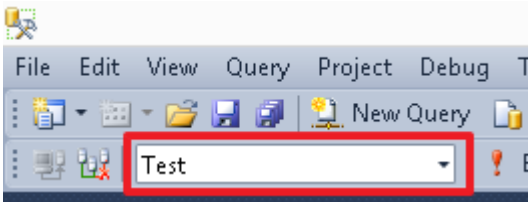
Кратко о MS SQL Server Management Studio (SSMS)

SQL Server Management Studio (SSMS) — утилита для Microsoft SQL Server для конфигурирования, управления и администрирования компонентов базы данных. Данная утилита содержит редактор скриптов (который в основном и будет нами использоваться) и графическую программу, которая работает с объектами и настройками сервера. Главным инструментом SQL Server Management Studio является Object Explorer, который позволяет пользователю просматривать, извлекать объекты сервера, а также управлять ими. ***Данный текст частично позаимствован с википедии.***

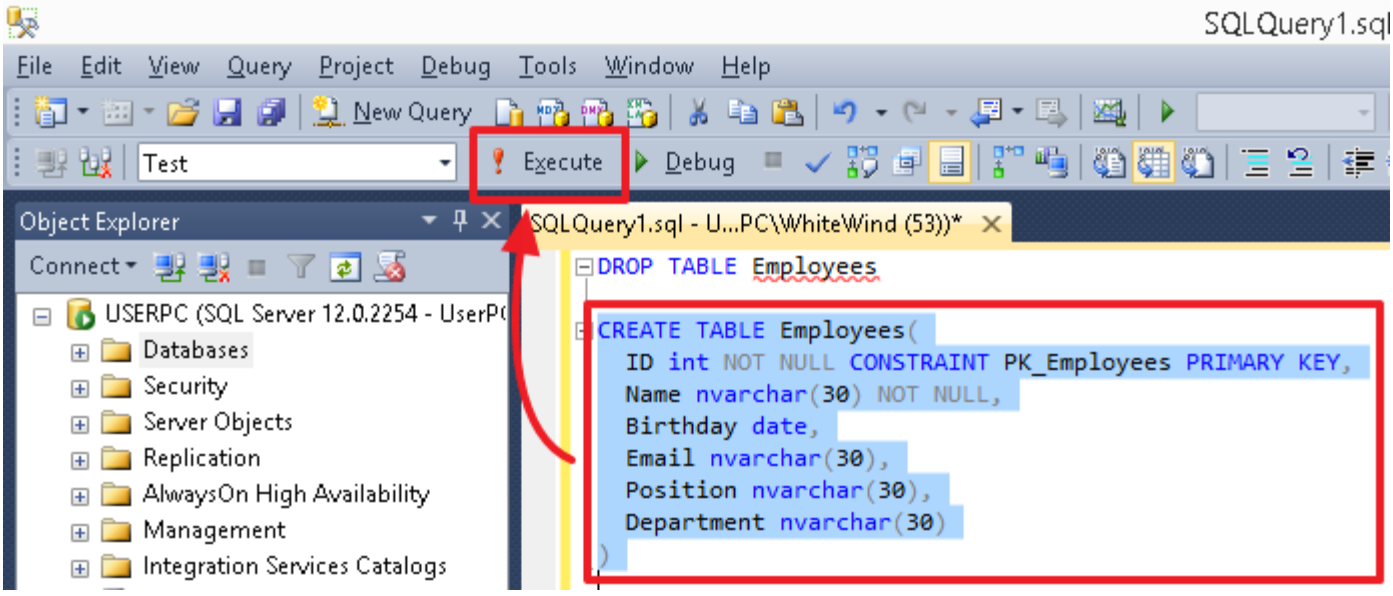
Для создания нового редактора скрипта используйте кнопку «New Query/Новый запрос»:



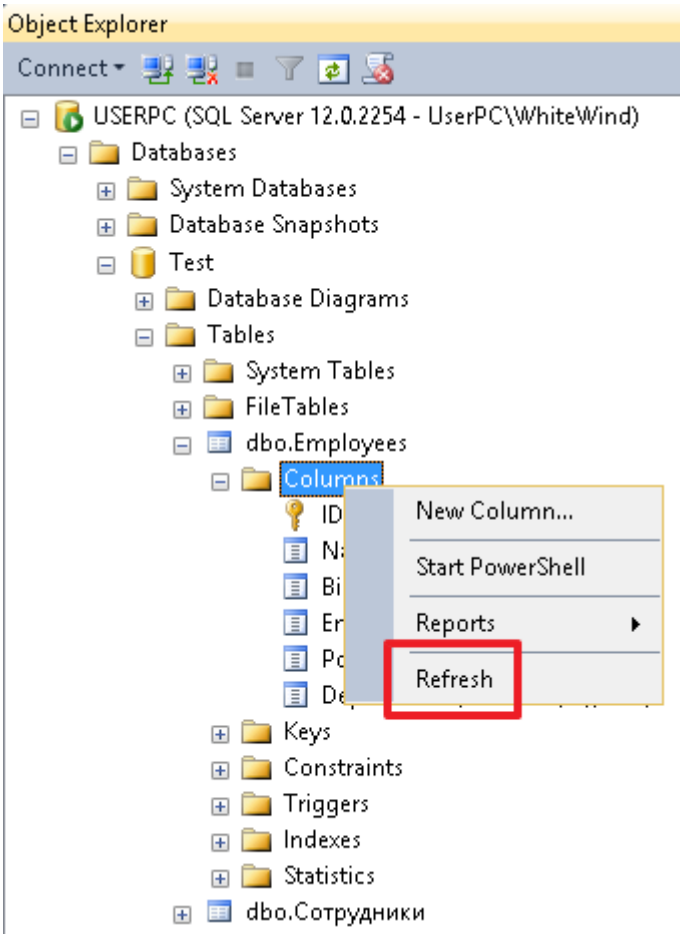
Для смены текущей базы данных можно использовать выпадающий список:



Для выполнения определенной команды (или группы команд) выделите ее и нажмите кнопку «Execute/Выполнить» или же клавишу «F5». Если в редакторе в текущий момент находится только одна команда, или же вам необходимо выполнить все команды, то ничего выделять не нужно.



После выполнения скриптов, в особенности создающих объекты (таблицы, столбцы, индексы), чтобы увидеть изменения, используйте обновление из контекстного меню, выделив соответствующую группу (например, Таблицы), саму таблицу или группу Столбцы в ней.



Собственно, это все, что нам необходимо будет знать для выполнения приведенных здесь примеров. Остальное по утилите SSMS несложно изучить самостоятельно.

Немного теории

Реляционная база данных (РБД, или далее в контексте просто БД) представляет из себя совокупность таблиц, связанных между собой. Если говорить грубо, то БД – файл в котором данные хранятся в структурированном виде.

СУБД – Система Управления этими Базами Данных, т.е. это комплекс инструментов для работы с конкретным типом БД (MS SQL, Oracle, MySQL, Firebird, …).

Примечание. *Т.к. в жизни, в разговорной речи, мы по большей части говорим: «БД Oracle», или даже просто «Oracle», на самом деле подразумевая «СУБД Oracle», то в контексте данного учебника иногда будет употребляться термин БД. Из контекста, я думаю, будет понятно, о чем именно идет речь.*

Таблица представляет из себя совокупность столбцов. Столбцы, так же могут называть полями или колонками, все эти слова будут использоваться как синонимы, выражающие одно и тоже.

Таблица – это главный объект РБД, все данные РБД хранятся построчно в столбцах таблицы. Строки, записи – тоже синонимы.

Для каждой таблицы, как и ее столбцов задаются наименования, по которым впоследствии к ним идет обращение.
Наименование объекта (имя таблицы, имя столбца, имя индекса и т.п.) в MS SQL может иметь максимальную длину 128 символов.

Для справки *– в БД ORACLE наименования объектов могут иметь максимальную длину 30 символов. Поэтому для конкретной БД нужно вырабатывать свои правила для наименования объектов, чтобы уложиться в лимит по количеству символов.*

SQL — язык позволяющий осуществлять запросы в БД посредством СУБД. В конкретной СУБД, язык SQL может иметь специфичную реализацию (свой диалект).

DDL и DML — подмножество языка SQL:

- Язык DDL служит для создания и модификации структуры БД, т.е. для создания/изменения/удаления таблиц и связей.
- Язык DML позволяет осуществлять манипуляции с данными таблиц, т.е. с ее строками. Он позволяет делать выборку данных из таблиц, добавлять новые данные в таблицы, а так же обновлять и удалять существующие данные.

В языке SQL можно использовать 2 вида комментариев (однострочный и многострочный):

```
-- однострочный комментарий
```

и

```
/*
  многострочный
  комментарий
*/
```

Собственно, все для теории этого будет достаточно.

DDL – Data Definition Language (язык описания данных)

Для примера рассмотрим таблицу с данными о сотрудниках, в привычном для человека не являющимся программистом виде:

Табельный номер	ФИО	Дата рождения	E-mail	Должность	Отдел
1000	Иванов И.И.	19.02.1955	i.ivanov@test.tt	Директор	Администрация
1001	Петров П.П.	03.12.1983	p.petrov@test.tt	Программист	ИТ
1002	Сидоров С.С.	07.06.1976	s.sidorov@test.tt	Бухгалтер	Бухгалтерия
1003	Андреев А.А.	17.04.1982	a.andreev@test.tt	Старший программист	ИТ

В данном случае столбцы таблицы имеют следующие наименования: Табельный номер, ФИО, Дата рождения, E-mail, Должность, Отдел.

Каждый из этих столбцов можно охарактеризовать по типу содержащемся в нем данных:
Табельный номер – целое число

- ФИО – строка
- Дата рождения – дата
- E-mail – строка
- Должность – строка
- Отдел – строка

Тип столбца – характеристика, которая говорит о том какого рода данные может хранить данный столбец.

Для начала будет достаточно запомнить только следующие основные типы данных используемые в MS SQL:

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 строка 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.

Так же значение поля, в том случае если это не запрещено, может быть не указано, для этой цели используется ключевое слово NULL.

Для выполнения примеров создадим тестовую базу под названием Test.

Простую базу данных (без указания дополнительных параметров) можно создать, выполнив следующую команду:

```
CREATE DATABASE Test
```

Удалить базу данных можно командой (стоит быть очень осторожным с данной командой):

```
DROP DATABASE Test
```

Для того, чтобы переключиться на нашу базу данных, можно выполнить команду:

```
USE Test
```

Или же выберите базу данных Test в выпадающем списке в области меню SSMS. При работе мною чаще используется именно этот способ переключения между базами.

Теперь в нашей БД мы можем создать таблицу используя описания в том виде как они есть, используя пробелы и символы кириллицы:

```
CREATE TABLE [Сотрудники](
  [Табельный номер] int,
  [ФИО] nvarchar(30),
  [Дата рождения] date,
  [E-mail] nvarchar(30),
  [Должность] nvarchar(30),
  [Отдел] nvarchar(30)
)
```

В данном случае нам придется заключать имена в квадратные скобки […].

Но в базе данных для большего удобства все наименования объектов лучше задавать на латинице и не использовать в именах пробелы. В MS SQL обычно в данном случае каждое слово начинается с прописной буквы, например, для поля «Табельный номер», мы могли бы задать имя PersonnelNumber. Так же в имени можно использовать цифры, например, PhoneNumber1.

На заметку. *В некоторых СУБД более предпочтительным может быть следующий формат наименований «PHONE_NUMBER», например, такой формат часто используется в БД ORACLE. Естественно при задании имя поля желательно чтобы оно не совпадало с ключевыми словами используемые в СУБД.*

По этой причине можете забыть о синтаксисе с квадратными скобками и удалить таблицу [Сотрудники]:

```
DROP TABLE [Сотрудники]
```

Например, таблицу с сотрудниками можно назвать «Employees», а ее полям можно задать следующие наименования:

- ID – Табельный номер (Идентификатор сотрудника)
- Name – ФИО
- Birthday – Дата рождения
- Email – E-mail
- Position – Должность
- Department – Отдел

Очень часто для наименования поля идентификатора используется слово ID.

Теперь создадим нашу таблицу:

```
CREATE TABLE Employees(
  ID int,
  Name nvarchar(30),
  Birthday date,
  Email nvarchar(30),
  Position nvarchar(30),
  Department nvarchar(30)
)
```

Для того, чтобы задать обязательные для заполнения столбцы, можно использовать опцию NOT NULL.

Для уже существующей таблицы поля можно переопределить при помощи следующих команд:

```
-- обновление поля ID
ALTER TABLE Employees ALTER COLUMN ID int NOT NULL

-- обновление поля Name
ALTER TABLE Employees ALTER COLUMN Name nvarchar(30) NOT NULL
```

На заметку

Общая концепция языка SQL для большинства СУБД остается одинаковой (по крайней мере, об этом я могу судить по тем СУБД, с которыми мне довелось поработать). Отличие DDL в разных СУБД в основном заключаются в типах данных (здесь могут отличаться не только их наименования, но и детали их реализации), так же может немного отличаться и сама специфика реализации языка SQL (т.е. суть команд одна и та же, но могут быть небольшие различия в диалекте, увы, но одного стандарта нет). Владея основами SQL вы легко сможете перейти с одной СУБД на другую, т.к. вам в данном случае нужно будет только разобраться в деталях реализации команд в новой СУБД, т.е. в большинстве случаев достаточно будет просто провести аналогию.

Чтобы не быть голословным, приведу несколько примеров тех же команд для СУБД ORACLE:

```
-- создание таблицы
CREATE TABLE Employees(
  ID int, -- в ORACLE тип int - это эквивалент(обертка) для number(38)
  Name nvarchar2(30), -- nvarchar2 в ORACLE эквивалентен nvarchar в MS SQL
  Birthday date,
  Email nvarchar2(30),
  Position nvarchar2(30),
  Department nvarchar2(30)
);

-- обновление полей ID и Name (здесь вместо ALTER COLUMN используется MODIFY(…))
ALTER TABLE Employees MODIFY(ID int NOT NULL,Name nvarchar2(30) NOT NULL);

-- добавление PK (в данном случае конструкция выглядит как и в MS SQL, она будет показана ниже)
ALTER TABLE Employees ADD CONSTRAINT PK_Employees PRIMARY KEY(ID);
```

Для ORACLE есть отличия в плане реализации типа varchar2, его кодировка зависит настроек БД и текст может сохраняться, например, в кодировке UTF-8. Помимо этого длину поля в ORACLE можно задать как в байтах, так и в символах, для этого используются дополнительные опции BYTE и CHAR, которые указываются после длины поля, например:

```
NAME varchar2(30 BYTE) -- вместимость поля будет равна 30 байтам
NAME varchar2(30 CHAR) -- вместимость поля будет равна 30 символов
```

Какая опция будет использоваться по умолчанию BYTE или CHAR, в случае простого указания в ORACLE типа varchar2(30), зависит от настроек БД, так же она иногда может задаваться в настройках IDE. В общем порой можно легко запутаться, поэтому в случае ORACLE, если используется тип varchar2 (а это здесь порой оправдано, например, при использовании кодировки UTF-8) я предпочитаю явно прописывать CHAR (т.к. обычно длину строки удобнее считать именно в символах).

Но в данном случае если в таблице уже есть какие-нибудь данные, то для успешного выполнения команд необходимо, чтобы во всех строках таблицы поля ID и Name были обязательно заполнены. Продемонстрируем это на примере, вставим в таблицу данные в поля ID, Position и Department, это можно сделать следующим скриптом:

```
INSERT Employees(ID,Position,Department) VALUES
(1000,N'Директор',N'Администрация'),
(1001,N'Программист',N'ИТ'),
(1002,N'Бухгалтер',N'Бухгалтерия'),
(1003,N'Старший программист',N'ИТ')
```

В данном случае, команда INSERT также выдаст ошибку, т.к. при вставке мы не указали значения обязательного поля Name.
В случае, если бы у нас в первоначальной таблице уже имелись эти данные, то команда «ALTER TABLE Employees ALTER COLUMN ID int NOT NULL» выполнилась бы успешно, а команда «ALTER TABLE Employees ALTER COLUMN Name int NOT NULL» выдала сообщение об ошибке, что в поле Name имеются NULL (не указанные) значения.

Добавим значения для полю Name и снова зальем данные:

```
INSERT Employees(ID,Position,Department,Name) VALUES
(1000,N'Директор',N'Администрация',N'Иванов И.И.'),
(1001,N'Программист',N'ИТ',N'Петров П.П.'),
(1002,N'Бухгалтер',N'Бухгалтерия',N'Сидоров С.С.'),
(1003,N'Старший программист',N'ИТ',N'Андреев А.А.')
```

Так же опцию NOT NULL можно использовать непосредственно при создании новой таблицы, т.е. в контексте команды CREATE TABLE.

Сначала удалим таблицу при помощи команды:

```
DROP TABLE Employees
```

Теперь создадим таблицу с обязательными для заполнения столбцами ID и Name:

```
CREATE TABLE Employees(
  ID int NOT NULL,
  Name nvarchar(30) NOT NULL,
  Birthday date,
  Email nvarchar(30),
  Position nvarchar(30),
  Department nvarchar(30)
)
```

Можно также после имени столбца написать NULL, что будет означать, что в нем будут допустимы NULL-значения (не указанные), но этого делать не обязательно, так как данная характеристика подразумевается по умолчанию.

Если требуется наоборот сделать существующий столбец необязательным для заполнения, то используем следующий синтаксис команды:

```
ALTER TABLE Employees ALTER COLUMN Name nvarchar(30) NULL
```

Или просто:

```
ALTER TABLE Employees ALTER COLUMN Name nvarchar(30)
```

Так же данной командой мы можем изменить тип поля на другой совместимый тип, или же изменить его длину. Для примера давайте расширим поле Name до 50 символов:

```
ALTER TABLE Employees ALTER COLUMN Name nvarchar(50)
```

Первичный ключ

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. Для нашей таблицы Employees таким уникальным значением может быть столбец ID (который содержит «Табельный номер сотрудника» — пускай в нашем случае данное значение уникально для каждого сотрудника и не может повторяться).

Создать первичный ключ к уже существующей таблице можно при помощи команды:

```
ALTER TABLE Employees ADD CONSTRAINT PK_Employees PRIMARY KEY(ID)
```

Где «PK_Employees» это имя ограничения, отвечающего за первичный ключ. Обычно для наименования первичного ключа используется префикс «PK_» после которого идет имя таблицы.

Если первичный ключ состоит из нескольких полей, то эти поля необходимо перечислить в скобках через запятую:

```
ALTER TABLE имя_таблицы ADD CONSTRAINT имя_ограничения PRIMARY KEY(поле1,поле2,…)
```

Стоит отметить, что в MS SQL все поля, которые входят в первичный ключ, должны иметь характеристику NOT NULL.

Так же первичный ключ можно определить непосредственно при создании таблицы, т.е. в контексте команды CREATE TABLE. Удалим таблицу:

```
DROP TABLE Employees
```

А затем создадим ее, используя следующий синтаксис:

```
CREATE TABLE Employees(
  ID int NOT NULL,
  Name nvarchar(30) NOT NULL,
  Birthday date,
  Email nvarchar(30),
  Position nvarchar(30),
  Department nvarchar(30),
  CONSTRAINT PK_Employees PRIMARY KEY(ID) -- описываем PK после всех полей, как ограничение
)
```

После создания зальем в таблицу данные:

```
INSERT Employees(ID,Position,Department,Name) VALUES
(1000,N'Директор',N'Администрация',N'Иванов И.И.'),
(1001,N'Программист',N'ИТ',N'Петров П.П.'),
(1002,N'Бухгалтер',N'Бухгалтерия',N'Сидоров С.С.'),
(1003,N'Старший программист',N'ИТ',N'Андреев А.А.')
```

Если первичный ключ в таблице состоит только из значений одного столбца, то можно использовать следующий синтаксис:

```
CREATE TABLE Employees(
  ID int NOT NULL CONSTRAINT PK_Employees PRIMARY KEY, -- указываем как характеристику поля
  Name nvarchar(30) NOT NULL,
  Birthday date,
  Email nvarchar(30),
  Position nvarchar(30),
  Department nvarchar(30)
)
```

На самом деле имя ограничения можно и не задавать, в этом случае ему будет присвоено системное имя (наподобие «PK__Employee__3214EC278DA42077»):

```
CREATE TABLE Employees(
  ID int NOT NULL,
  Name nvarchar(30) NOT NULL,
  Birthday date,
  Email nvarchar(30),
  Position nvarchar(30),
  Department nvarchar(30),
  PRIMARY KEY(ID)
)
```

Или:

```
CREATE TABLE Employees(
  ID int NOT NULL PRIMARY KEY,
  Name nvarchar(30) NOT NULL,
  Birthday date,
  Email nvarchar(30),
  Position nvarchar(30),
  Department nvarchar(30)
)
```

Но я бы рекомендовал для постоянных таблиц всегда явно задавать имя ограничения, т.к. по явно заданному и понятному имени с ним впоследствии будет легче проводить манипуляции, например, можно произвести его удаление:

```
ALTER TABLE Employees DROP CONSTRAINT PK_Employees
```

Но такой краткий синтаксис, без указания имен ограничений, удобно применять при создании временных таблиц БД (имя временной таблицы начинается с # или ##), которые после использования будут удалены.

Подытожим

На данный момент мы рассмотрели следующие команды:

- **CREATE TABLE** имя_таблицы (перечисление полей и их типов, ограничений) – служит для создания новой таблицы в текущей БД;
- **DROP TABLE** имя_таблицы – служит для удаления таблицы из текущей БД;
- **ALTER TABLE** имя_таблицы **ALTER COLUMN** имя_столбца … – служит для обновления типа столбца или для изменения его настроек (например для задания характеристики NULL или NOT NULL);
- **ALTER TABLE** имя_таблицы **ADD CONSTRAINT** имя_ограничения **PRIMARY KEY**(поле1, поле2,…) – добавление первичного ключа к уже существующей таблице;
- **ALTER TABLE** имя_таблицы **DROP CONSTRAINT** имя_ограничения – удаление ограничения из таблицы.

Немного про временные таблицы

Вырезка из MSDN. *В MS SQL Server существует два вида временных таблиц: локальные (#) и глобальные (##). Локальные временные таблицы видны только их создателям до завершения сеанса соединения с экземпляром SQL Server, как только они впервые созданы. Локальные временные таблицы автоматически удаляются после отключения пользователя от экземпляра SQL Server. Глобальные временные таблицы видны всем пользователям в течение любых сеансов соединения после создания этих таблиц и удаляются, когда все пользователи, ссылающиеся на эти таблицы, отключаются от экземпляра SQL Server.*

Временные таблицы создаются в системной базе tempdb, т.е. создавая их мы не засоряем основную базу, в остальном же временные таблицы полностью идентичны обычным таблицам, их так же можно удалить при помощи команды DROP TABLE. Чаще используются локальные (#) временные таблицы.

Для создания временной таблицы можно использовать команду CREATE TABLE:

```
CREATE TABLE #Temp(
  ID int,
  Name nvarchar(30)
)
```

Так как временная таблица в MS SQL аналогична обычной таблице, ее соответственно так же можно удалить самому командой DROP TABLE:

```
DROP TABLE #Temp
```

Так же временную таблицу (как собственно и обычную таблицу) можно создать и сразу заполнить данными возвращаемые запросом используя синтаксис SELECT … INTO:

```
SELECT ID,Name
INTO #Temp
FROM Employees
```

На заметку. *В разных СУБД реализация временных таблиц может отличаться. Например, в СУБД ORACLE и Firebird структура временных таблиц должна быть определена заранее командой CREATE GLOBAL TEMPORARY TABLE с указанием специфики хранения в ней данных, дальше уже пользователь видит ее среди основных таблиц и работает с ней как с обычной таблицей.*

Нормализация БД – дробление на подтаблицы (справочники) и определение связей

Наша текущая таблица Employees имеет недостаток в том, что в полях Position и Department пользователь может ввести любой текст, что в первую очередь чревато ошибками, так как он у одного сотрудника может указать в качестве отдела просто «ИТ», а у второго сотрудника, например, ввести «ИТ-отдел», у третьего «IT». В итоге будет непонятно, что имел ввиду пользователь, т.е. являются ли данные сотрудники работниками одного отдела, или же пользователь описался и это 3 разных отдела? А тем более, в этом случае, мы не сможем правильно сгруппировать данные для какого-то отчета, где, может требоваться показать количество сотрудников в разрезе каждого отдела.

Второй недостаток заключается в объеме хранения данной информации и ее дублированием, т.е. для каждого сотрудника указывается полное наименование отдела, что требует в БД места для хранения каждого символа из названия отдела.

Третий недостаток – сложность обновления данных полей, в случае если изменится название какой-то должности, например, если потребуется переименовать должность «Программист», на «Младший программист». В данном случае нам придется вносить изменения в каждую строчку таблицы, у которой Должность равняется «Программист».

Чтобы избежать данных недостатков и применяется, так называемая, нормализация базы данных – дробление ее на подтаблицы, таблицы справочники. Не обязательно лезть в дебри теории и изучать что из себя представляют нормальные формы, достаточно понимать суть нормализации.

Давайте создадим 2 таблицы справочники «Должности» и «Отделы», первую назовем Positions, а вторую соответственно Departments:

```
CREATE TABLE Positions(
  ID int IDENTITY(1,1) NOT NULL CONSTRAINT PK_Positions PRIMARY KEY,
  Name nvarchar(30) NOT NULL
)

CREATE TABLE Departments(
  ID int IDENTITY(1,1) NOT NULL CONSTRAINT PK_Departments PRIMARY KEY,
  Name nvarchar(30) NOT NULL
)
```

Заметим, что здесь мы использовали новую опцию IDENTITY, которая говорит о том, что данные в столбце ID будут нумероваться автоматически, начиная с 1, с шагом 1, т.е. при добавлении новых записей им последовательно будут присваиваться значения 1, 2, 3, и т.д. Такие поля обычно называют автоинкрементными. В таблице может быть определено только одно поле со свойством IDENTITY и обычно, но необязательно, такое поле является первичным ключом для данной таблицы.

На заметку. *В разных СУБД реализация полей со счетчиком может делаться по своему. В MySQL, например, такое поле определяется при помощи опции AUTO_INCREMENT. В ORACLE и Firebird раньше данную функциональность можно было съэмулировать при помощи использования последовательностей (SEQUENCE). Но насколько я знаю в ORACLE сейчас добавили опцию GENERATED AS IDENTITY.*

Давайте заполним эти таблицы автоматически, на основании текущих данных записанных в полях Position и Department таблицы Employees:

```
-- заполняем поле Name таблицы Positions, уникальными значениями из поля Position таблицы Employ
ees
INSERT Positions(Name)
SELECT DISTINCT Position
FROM Employees
WHERE Position IS NOT NULL -- отбрасываем записи у которых позиция не указана
```

То же самое проделаем для таблицы Departments:

```
INSERT Departments(Name)
SELECT DISTINCT Department
FROM Employees
WHERE Department IS NOT NULL
```

Если теперь мы откроем таблицы Positions и Departments, то увидим пронумерованный набор значений по полю ID:

```
SELECT * FROM Positions
```

ID	Name
1	Бухгалтер
2	Директор
3	Программист
4	Старший программист

```
SELECT * FROM Departments
```

ID	Name
1	Администрация
2	Бухгалтерия
3	ИТ

Данные таблицы теперь и будут играть роль справочников для задания должностей и отделов. Теперь мы будем ссылаться на идентификаторы должностей и отделов. В первую очередь создадим новые поля в таблице Employees для хранения данных идентификаторов:

```
-- добавляем поле для ID должности
ALTER TABLE Employees ADD PositionID int
-- добавляем поле для ID отдела
ALTER TABLE Employees ADD DepartmentID int
```

Тип ссылочных полей должен быть каким же, как и в справочниках, в данном случае это int.

Так же добавить в таблицу сразу несколько полей можно одной командой, перечислив поля через запятую:

```
ALTER TABLE Employees ADD PositionID int, DepartmentID int
```

Теперь пропишем ссылки (ссылочные ограничения — FOREIGN KEY) для этих полей, для того чтобы пользователь не имел возможности записать в данные поля, значения, отсутствующие среди значений ID находящихся в справочниках.

```
ALTER TABLE Employees ADD CONSTRAINT FK_Employees_PositionID
FOREIGN KEY(PositionID) REFERENCES Positions(ID)
```

И то же самое сделаем для второго поля:

```
ALTER TABLE Employees ADD CONSTRAINT FK_Employees_DepartmentID
FOREIGN KEY(DepartmentID) REFERENCES Departments(ID)
```

Теперь пользователь в данные поля сможет занести только значения ID из соответствующего справочника. Соответственно, чтобы использовать новый отдел или должность, он первым делом должен будет добавить новую запись в соответствующий справочник. Т.к. должности и отделы теперь хранятся в справочниках в одном единственном экземпляре, то чтобы изменить название, достаточно изменить его только в справочнике.

Имя ссылочного ограничения, обычно является составным, оно состоит из префикса «FK_», затем идет имя таблицы и после знака подчеркивания идет имя поля, которое ссылается на идентификатор таблицы-справочника.

Идентификатор (ID) обычно является внутренним значением, которое используется только для связей и какое значение там хранится, в большинстве случаев абсолютно безразлично, поэтому не нужно пытаться избавиться от дырок в последовательности чисел, которые возникают по ходу работы с таблицей, например, после удаления записей из справочника.

Так же в некоторых случаях ссылку можно организовать по нескольким полям:

```
ALTER TABLE таблица ADD CONSTRAINT имя_ограничения
FOREIGN KEY(поле1,поле2,…) REFERENCES таблица_справочник(поле1,поле2,…)
```

В данном случае в таблице «таблица_справочник» первичный ключ представлен комбинацией из нескольких полей (поле1, поле2,…).

Собственно, теперь обновим поля PositionID и DepartmentID значениями ID из справочников. Воспользуемся для этой цели DML командой UPDATE:

```
UPDATE e
SET
  PositionID=(SELECT ID FROM Positions WHERE Name=e.Position),
  DepartmentID=(SELECT ID FROM Departments WHERE Name=e.Department)
FROM Employees e
```

Посмотрим, что получилось, выполнив запрос:

```
SELECT * FROM Employees
```

ID	Name	Birthday	Email	Position	Department	PositionID	DepartmentID
1000	Иванов И.И.	NULL	NULL	Директор	Администрация	2	1
1001	Петров П.П.	NULL	NULL	Программист	ИТ	3	3
1002	Сидоров С.С.	NULL	NULL	Бухгалтер	Бухгалтерия	1	2
1003	Андреев А.А.	NULL	NULL	Старший программист	ИТ	4	3

Всё, поля PositionID и DepartmentID заполнены соответствующие должностям и отделам идентификаторами надобности в полях Position и Department в таблице Employees теперь нет, можно удалить эти поля:

```
ALTER TABLE Employees DROP COLUMN Position,Department
```

Теперь таблица у нас приобрела следующий вид:

```
SELECT * FROM Employees
```

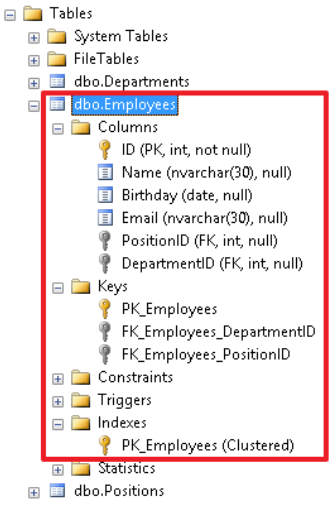
ID	Name	Birthday	Email	PositionID	DepartmentID
1000	Иванов И.И.	NULL	NULL	2	1
1001	Петров П.П.	NULL	NULL	3	3
1002	Сидоров С.С.	NULL	NULL	1	2
1003	Андреев А.А.	NULL	NULL	4	3

Т.е. мы в итоге избавились от хранения избыточной информации. Теперь, по номерам должности и отдела можем однозначно определить их названия, используя значения в таблицах-справочниках:

```
SELECT e.ID,e.Name,p.Name PositionName,d.Name DepartmentName
FROM Employees e
LEFT JOIN Departments d ON d.ID=e.DepartmentID
LEFT JOIN Positions p ON p.ID=e.PositionID
```

ID	Name	PositionName	DepartmentName
1000	Иванов И.И.	Директор	Администрация
1001	Петров П.П.	Программист	ИТ
1002	Сидоров С.С.	Бухгалтер	Бухгалтерия
1003	Андреев А.А.	Старший программист	ИТ

В инспекторе объектов мы можем увидеть все объекты, созданные для в данной таблицы. Отсюда же можно производить разные манипуляции с данными объектами – например, переименовывать или удалять объекты.



Так же стоит отметить, что таблица может ссылаться сама на себя, т.е. можно создать рекурсивную ссылку. Для примера добавим в нашу таблицу с сотрудниками еще одно поле ManagerID, которое будет указывать на сотрудника, которому подчиняется данный сотрудник. Создадим поле:

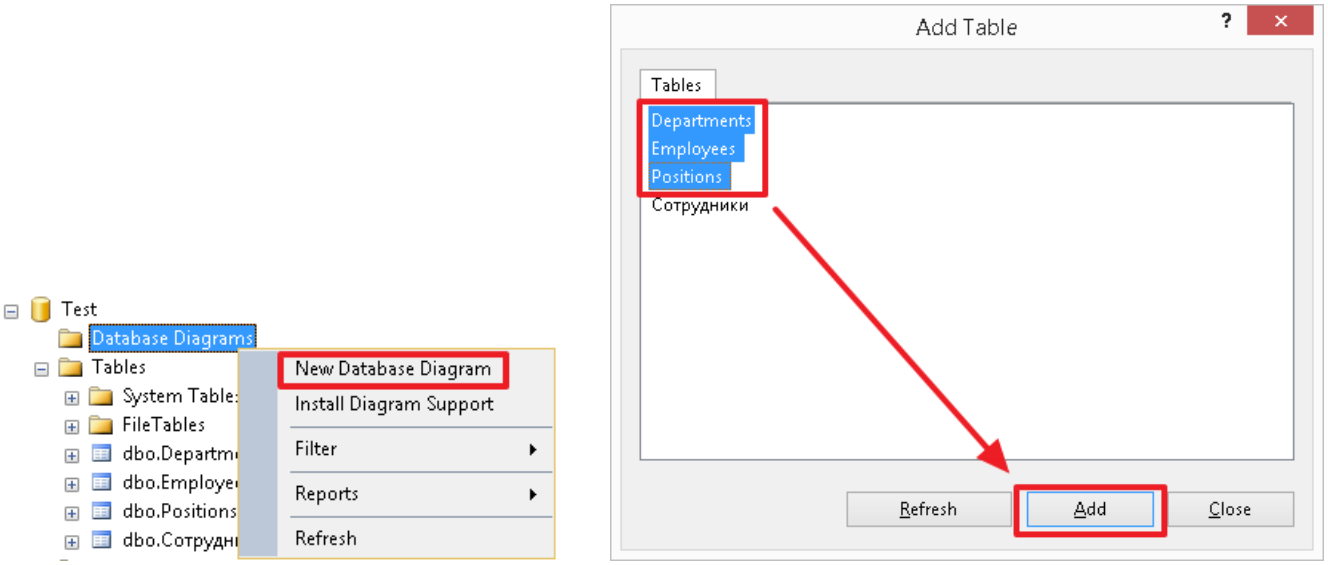
```
ALTER TABLE Employees ADD ManagerID int
```

В данном поле допустимо значение NULL, поле будет пустым, если, например, над сотрудником нет вышестоящих.

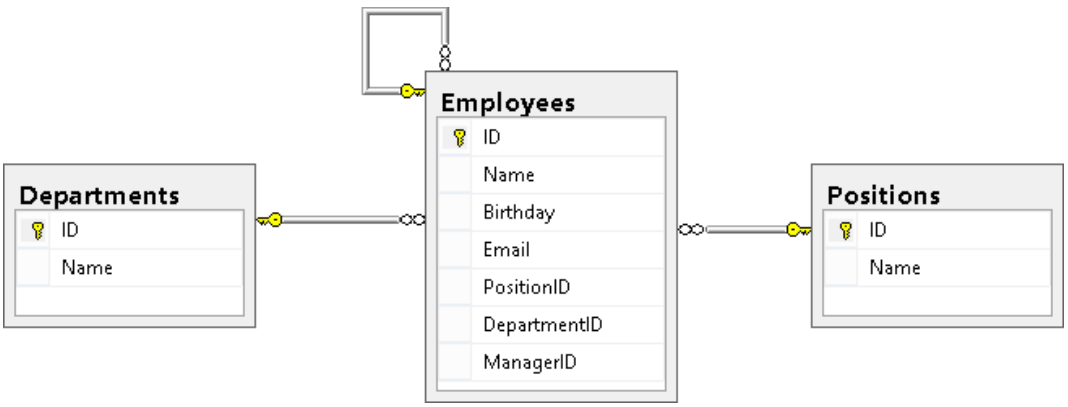
Теперь создадим FOREIGN KEY на таблицу Employees:

```
ALTER TABLE Employees ADD CONSTRAINT FK_Employees_ManagerID
FOREIGN KEY (ManagerID) REFERENCES Employees(ID)
```

Давайте, теперь создадим диаграмму и посмотрим, как выглядят на ней связи между нашими таблицами:



В результате мы должны увидеть следующую картину (таблица Employees связана с таблицами Positions и Depertments, а так же ссылается сама на себя):



Напоследок стоит сказать, что ссылочные ключи могут включать дополнительные опции ON DELETE CASCADE и ON UPDATE CASCADE, которые говорят о том, как вести себя при удалении или обновлении записи, на которую есть ссылки в таблице-справочнике. Если эти опции не указаны, то мы не можем изменить ID в таблице справочнике у той записи, на которую есть ссылки из другой таблицы, так же мы не сможем удалить такую запись из справочника, пока не удалим все строки, ссылающиеся на эту запись или, же обновим в этих строках ссылки на другое значение.

Для примера пересоздадим таблицу с указанием опции ON DELETE CASCADE для FK_Employees_DepartmentID:

```
DROP TABLE Employees

CREATE TABLE Employees(
  ID int NOT NULL,
  Name nvarchar(30),
  Birthday date,
  Email nvarchar(30),
  PositionID int,
  DepartmentID int,
  ManagerID int,
CONSTRAINT PK_Employees PRIMARY KEY (ID),
CONSTRAINT FK_Employees_DepartmentID FOREIGN KEY(DepartmentID) REFERENCES Departments(ID)
ON DELETE CASCADE,
CONSTRAINT FK_Employees_PositionID FOREIGN KEY(PositionID) REFERENCES Positions(ID),
CONSTRAINT FK_Employees_ManagerID FOREIGN KEY (ManagerID) REFERENCES Employees(ID)
)

INSERT Employees (ID,Name,Birthday,PositionID,DepartmentID,ManagerID)VALUES
(1000,N'Иванов И.И.','19550219',2,1,NULL),
(1001,N'Петров П.П.','19831203',3,3,1003),
(1002,N'Сидоров С.С.','19760607',1,2,1000),
(1003,N'Андреев А.А.','19820417',4,3,1000)
```

Удалим отдел с идентификатором 3 из таблицы Departments:

```
DELETE Departments WHERE ID=3
```

Посмотрим на данные таблицы Employees:

```
SELECT * FROM Employees
```

ID	Name	Birthday	Email	PositionID	DepartmentID	ManagerID
1000	Иванов И.И.	1955-02-19	NULL	2	1	NULL
1002	Сидоров С.С.	1976-06-07	NULL	1	2	1000

Как видим, данные по отделу 3 из таблицы Employees так же удалились.

Опция ON UPDATE CASCADE ведет себя аналогично, но действует она при обновлении значения ID в справочнике. Например, если мы поменяем ID должности в справочнике должностей, то в этом случае будет производиться обновление DepartmentID в таблице Employees на новое значение ID которое мы задали в справочнике. Но в данном случае это продемонстрировать просто не получится, т.к. у колонки ID в таблице Departments стоит опция IDENTITY, которая не позволит нам выполнить следующий запрос (сменить идентификатор отдела 3 на 30):

```
UPDATE Departments
SET
  ID=30
WHERE ID=3
```

Главное понять суть этих 2-х опций ON DELETE CASCADE и ON UPDATE CASCADE. Я применяю эти опции очень в редких случаях и рекомендую хорошо подумать, прежде чем указывать их в ссылочном ограничении, т.к. при нечаянном удалении записи из таблицы справочника это может привести к большим проблемам и создать цепную реакцию.

Восстановим отдел 3:

```
-- даем разрешение на добавление/изменение IDENTITY значения
SET IDENTITY_INSERT Departments ON

INSERT Departments(ID,Name) VALUES(3,N'ИТ')

-- запрещаем добавление/изменение IDENTITY значения
SET IDENTITY_INSERT Departments OFF
```

Полностью очистим таблицу Employees при помощи команды TRUNCATE TABLE:

```
TRUNCATE TABLE Employees
```

И снова перезальем в нее данные используя предыдущую команду INSERT:

```
INSERT Employees (ID,Name,Birthday,PositionID,DepartmentID,ManagerID)VALUES
(1000,N'Иванов И.И.','19550219',2,1,NULL),
(1001,N'Петров П.П.','19831203',3,3,1003),
(1002,N'Сидоров С.С.','19760607',1,2,1000),
(1003,N'Андреев А.А.','19820417',4,3,1000)
```

Подытожим

На данным момент к нашим знаниям добавилось еще несколько команд DDL:

- Добавление свойства IDENTITY к полю – позволяет сделать это поле автоматически заполняемым (полем-счетчиком) для таблицы;
- **ALTER TABLE** имя_таблицы **ADD** перечень_полей_с_характеристиками – позволяет добавить новые поля в таблицу;
- **ALTER TABLE** имя_таблицы **DROP COLUMN** перечень_полей – позволяет удалить поля из таблицы;
- **ALTER TABLE** имя_таблицы **ADD CONSTRAINT** имя_ограничения **FOREIGN KEY**(поля) **REFERENCES**таблица_справочник(поля) – позволяет определить связь между таблицей и таблицей справочником.

Прочие ограничения – UNIQUE, DEFAULT, CHECK

При помощи ограничения UNIQUE можно сказать что значения для каждой строки в данном поле или в наборе полей должно быть уникальным. В случае таблицы Employees, такое ограничение мы можем наложить на поле Email. Только предварительно заполним Email значениями, если они еще не определены:

```
UPDATE Employees SET Email='i.ivanov@test.tt' WHERE ID=1000
UPDATE Employees SET Email='p.petrov@test.tt' WHERE ID=1001
UPDATE Employees SET Email='s.sidorov@test.tt' WHERE ID=1002
UPDATE Employees SET Email='a.andreev@test.tt' WHERE ID=1003
```

А теперь можно наложить на это поле ограничение-уникальности:

```
ALTER TABLE Employees ADD CONSTRAINT UQ_Employees_Email UNIQUE(Email)
```

Теперь пользователь не сможет внести один и тот же E-Mail у нескольких сотрудников.

Ограничение уникальности обычно именуется следующим образом – сначала идет префикс «UQ_», далее название таблицы и после знака подчеркивания идет имя поля, на которое накладывается данное ограничение.

Соответственно если уникальной в разрезе строк таблицы должна быть комбинация полей, то перечисляем их через запятую:

```
ALTER TABLE имя_таблицы ADD CONSTRAINT имя_ограничения UNIQUE(поле1,поле2,…)
```

При помощи добавления к полю ограничения DEFAULT мы можем задать значение по умолчанию, которое будет подставляться в случае, если при вставке новой записи данное поле не будет перечислено в списке полей команды INSERT. Данное ограничение можно задать непосредственно при создании таблицы.

Давайте добавим в таблицу Employees новое поле «Дата приема» и назовем его HireDate и скажем что значение по умолчанию у данного поля будет текущая дата:

```
ALTER TABLE Employees ADD HireDate date NOT NULL DEFAULT SYSDATETIME()
```

Или если столбец HireDate уже существует, то можно использовать следующий синтаксис:

```
ALTER TABLE Employees ADD DEFAULT SYSDATETIME() FOR HireDate
```

Здесь я не указал имя ограничения, т.к. в случае DEFAULT у меня сложилось мнение, что это не столь критично. Но если делать по-хорошему, то, думаю, не нужно лениться и стоит задать нормальное имя. Делается это следующим образом:

```
ALTER TABLE Employees ADD CONSTRAINT DF_Employees_HireDate DEFAULT SYSDATETIME() FOR HireDate
```

Та как данного столбца раньше не было, то при его добавлении в каждую запись в поле HireDate будет вставлено текущее значение даты.

При добавлении новой записи, текущая дата так же будет вставлена автоматом, конечно если мы ее явно не зададим, т.е. не укажем в списке столбцов. Покажем это на примере, не указав поле HireDate в перечне добавляемых значений:

```
INSERT Employees(ID,Name,Email)VALUES(1004,N'Сергеев С.С.','s.sergeev@test.tt')
```

Посмотрим, что получилось:

```
SELECT * FROM Employees
```

ID	Name	Birthday	Email	PositionID	DepartmentID	ManagerID	HireDate
1000	Иванов И.И.	1955-02-19	i.ivanov@test.tt	2	1	NULL	2015-04-08
1001	Петров П.П.	1983-12-03	p.petrov@test.tt	3	4	1003	2015-04-08
1002	Сидоров С.С.	1976-06-07	s.sidorov@test.tt	1	2	1000	2015-04-08
1003	Андреев А.А.	1982-04-17	a.andreev@test.tt	4	3	1000	2015-04-08
1004	Сергеев С.С.	NULL	s.sergeev@test.tt	NULL	NULL	NULL	2015-04-08

Проверочное ограничение CHECK используется в том случае, когда необходимо осуществить проверку вставляемых в поле значений. Например, наложим данное ограничение на поле табельный номер, которое у нас является идентификатором сотрудника (ID). При помощи данного ограничения скажем, что табельные номера должны иметь значение от 1000 до 1999:

```
ALTER TABLE Employees ADD CONSTRAINT CK_Employees_ID CHECK(ID BETWEEN 1000 AND 1999)
```

Ограничение обычно именуется так же, сначала идет префикс «CK_», затем имя таблицы и имя поля, на которое наложено это ограничение.

Попробуем вставить недопустимую запись для проверки, что ограничение работает (мы должны получить соответствующую ошибку):

```
INSERT Employees(ID,Email) VALUES(2000,'test@test.tt')
```

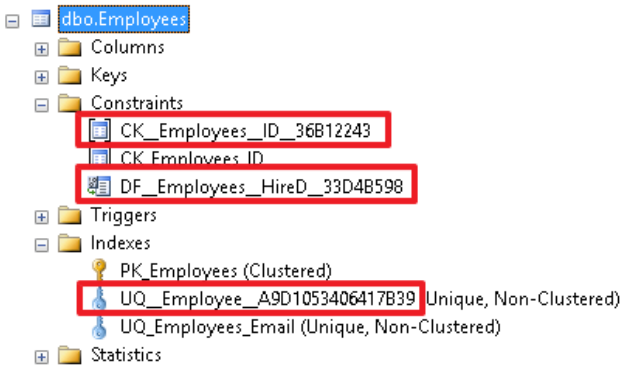
А теперь изменим вставляемое значение на 1500 и убедимся, что запись вставится:

```
INSERT Employees(ID,Email) VALUES(1500,'test@test.tt')
```

Можно так же создать ограничения UNIQUE и CHECK без указания имени:

```
ALTER TABLE Employees ADD UNIQUE(Email)
ALTER TABLE Employees ADD CHECK(ID BETWEEN 1000 AND 1999)
```

Но это не очень хорошая практика и лучше задавать имя ограничения в явном виде, т.к. чтобы разобраться потом, что будет сложнее, нужно будет открывать объект и смотреть, за что он отвечает.



При хорошем наименовании много информации об ограничении можно узнать непосредственно по его имени.

И, соответственно, все эти ограничения можно создать сразу же при создании таблицы, если ее еще нет. Удалим таблицу:

```
DROP TABLE Employees
```

И пересоздадим ее со всеми созданными ограничениями одной командой CREATE TABLE:

```
CREATE TABLE Employees(
  ID int NOT NULL,
  Name nvarchar(30),
  Birthday date,
  Email nvarchar(30),
  PositionID int,
  DepartmentID int,
  HireDate date NOT NULL DEFAULT SYSDATETIME(), -- для DEFAULT я сделаю исключение
CONSTRAINT PK_Employees PRIMARY KEY (ID),
CONSTRAINT FK_Employees_DepartmentID FOREIGN KEY(DepartmentID) REFERENCES Departments(ID),
CONSTRAINT FK_Employees_PositionID FOREIGN KEY(PositionID) REFERENCES Positions(ID),
CONSTRAINT UQ_Employees_Email UNIQUE (Email),
CONSTRAINT CK_Employees_ID CHECK (ID BETWEEN 1000 AND 1999)
)
```

Напоследок вставим в таблицу наших сотрудников:

```
INSERT Employees (ID,Name,Birthday,Email,PositionID,DepartmentID)VALUES
(1000,N'Иванов И.И.','19550219','i.ivanov@test.tt',2,1),
(1001,N'Петров П.П.','19831203','p.petrov@test.tt',3,3),
(1002,N'Сидоров С.С.','19760607','s.sidorov@test.tt',1,2),
(1003,N'Андреев А.А.','19820417','a.andreev@test.tt',4,3)
```

Немного про индексы, создаваемые при создании ограничений PRIMARY KEY и UNIQUE

Как можно увидеть на скриншоте выше, при создании ограничений PRIMARY KEY и UNIQUE автоматически создались индексы с такими же названиями (PK_Employees и UQ_Employees_Email). По умолчанию индекс для первичного ключа создается как CLUSTERED, а для всех остальных индексов как NONCLUSTERED. Стоит сказать, что понятие кластерного индекса есть не во всех СУБД. Таблица может иметь только один кластерный (CLUSTERED) индекс. CLUSTERED – означает, что записи таблицы будут сортироваться по этому индексу, так же можно сказать, что этот индекс имеет непосредственный доступ ко всем данным таблицы. Это так сказать главный индекс таблицы. Если сказать еще грубее, то это индекс, прикрученный к таблице. Кластерный индекс – это очень мощное средство, которое может помочь при оптимизации запросов, пока просто запомним это. Если мы хотим сказать, чтобы кластерный индекс использовался не в первичном ключе, а для другого индекса, то при создании первичного ключа мы должны указать опцию NONCLUSTERED:

```
ALTER TABLE имя_таблицы ADD CONSTRAINT имя_ограничения
PRIMARY KEY NONCLUSTERED(поле1,поле2,…)
```

Для примера сделаем индекс ограничения PK_Employees некластерным, а индекс ограничения UQ_Employees_Email кластерным. Первым делом удалим данные ограничения:

```
ALTER TABLE Employees DROP CONSTRAINT PK_Employees
ALTER TABLE Employees DROP CONSTRAINT UQ_Employees_Email
```

А теперь создадим их с опциями CLUSTERED и NONCLUSTERED:

```
ALTER TABLE Employees ADD CONSTRAINT PK_Employees PRIMARY KEY NONCLUSTERED (ID)
ALTER TABLE Employees ADD CONSTRAINT UQ_Employees_Email UNIQUE CLUSTERED (Email)
```

Теперь, выполнив выборку из таблицы Employees, мы увидим, что записи отсортировались по кластерному индексу UQ_Employees_Email:

```
SELECT * FROM Employees
```

ID	Name	Birthday	Email	PositionID	DepartmentID	HireDate
1003	Андреев А.А.	1982-04-17	a.andreev@test.tt	4	3	2015-04-08
1000	Иванов И.И.	1955-02-19	i.ivanov@test.tt	2	1	2015-04-08
1001	Петров П.П.	1983-12-03	p.petrov@test.tt	3	3	2015-04-08
1002	Сидоров С.С.	1976-06-07	s.sidorov@test.tt	1	2	2015-04-08

До этого, когда кластерным индексом был индекс PK_Employees, записи по умолчанию сортировались по полю ID.

Но в данном случае это всего лишь пример, который показывает суть кластерного индекса, т.к. скорее всего к таблице Employees будут делаться запросы по полю ID и в каких-то случаях, возможно, она сама будет выступать в роли справочника.

Для справочников обычно целесообразно, чтобы кластерный индекс был построен по первичному ключу, т.к. в запросах мы часто ссылаемся на идентификатор справочника для

получения, например, наименования (Должности, Отдела). Здесь вспомним, о чем я писал выше, что кластерный индекс имеет прямой доступ к строкам таблицы, а отсюда следует, что мы можем получить значение любого столбца без дополнительных накладных расходов.

Кластерный индекс выгодно применять к полям, по которым выборка идет наиболее часто.

Иногда в таблицах создают ключ по суррогатному полю, вот в этом случае бывает полезно сохранить опцию CLUSTERED индекс для более подходящего индекса и указать опцию NONCLUSTERED при создании суррогатного первичного ключа.

Подытожим

На данном этапе мы познакомились со всеми видами ограничений, в их самом простом виде, которые создаются командой вида «ALTER TABLE имя_таблицы ADD CONSTRAINT имя_ограничения …»:

- **PRIMARY KEY** – первичный ключ;
- **FOREIGN KEY** – настройка связей и контроль ссылочной целостности данных;
- **UNIQUE** – позволяет создать уникальность;
- **CHECK** – позволяет осуществлять корректность введенных данных;
- **DEFAULT** – позволяет задать значение по умолчанию;
- Так же стоит отметить, что все ограничения можно удалить, используя команду «**ALTER TABLE** имя_таблицы **DROP CONSTRAINT** имя_ограничения».

Так же мы частично затронули тему индексов и разобрали понятие кластерный (**CLUSTERED**) и некластерный (**NONCLUSTERED**) индекс.

Создание самостоятельных индексов

Под самостоятельностью я здесь имеется в виду индексы, которые создаются не для ограничения PRIMARY KEY или UNIQUE.

Индексы по полю или полям можно создавать следующей командой:

```
CREATE INDEX IDX_Employees_Name ON Employees(Name)
```

Так же здесь можно указать опции CLUSTERED, NONCLUSTERED, UNIQUE, а так же можно указать направление сортировки каждого отдельного поля ASC (по умолчанию) или DESC:

```
CREATE UNIQUE NONCLUSTERED INDEX UQ_Employees_EmailDesc ON Employees(Email DESC)
```

При создании некластерного индекса опцию NONCLUSTERED можно отпустить, т.к. она подразумевается по умолчанию, здесь она показана просто, чтобы указать позицию опции CLUSTERED или NONCLUSTERED в команде.

Удалить индекс можно следующей командой:

```
DROP INDEX IDX_Employees_Name ON Employees
```

Простые индексы так же, как и ограничения, можно создать в контексте команды CREATE TABLE.

Для примера снова удалим таблицу:

```
DROP TABLE Employees
```

И пересоздадим ее со всеми созданными ограничениями и индексами одной командой CREATE TABLE:

```
CREATE TABLE Employees(
  ID int NOT NULL,
  Name nvarchar(30),
  Birthday date,
  Email nvarchar(30),
  PositionID int,
  DepartmentID int,
  HireDate date NOT NULL CONSTRAINT DF_Employees_HireDate DEFAULT SYSDATETIME(),
  ManagerID int,
CONSTRAINT PK_Employees PRIMARY KEY (ID),
CONSTRAINT FK_Employees_DepartmentID FOREIGN KEY(DepartmentID) REFERENCES Departments(ID),
CONSTRAINT FK_Employees_PositionID FOREIGN KEY(PositionID) REFERENCES Positions(ID),
CONSTRAINT FK_Employees_ManagerID FOREIGN KEY (ManagerID) REFERENCES Employees(ID),
CONSTRAINT UQ_Employees_Email UNIQUE(Email),
CONSTRAINT CK_Employees_ID CHECK(ID BETWEEN 1000 AND 1999),
INDEX IDX_Employees_Name(Name)
)
```

Напоследок вставим в таблицу наших сотрудников:

```
INSERT Employees (ID,Name,Birthday,Email,PositionID,DepartmentID,ManagerID)VALUES
(1000,N'Иванов И.И.','19550219','i.ivanov@test.tt',2,1,NULL),
(1001,N'Петров П.П.','19831203','p.petrov@test.tt',3,3,1003),
(1002,N'Сидоров С.С.','19760607','s.sidorov@test.tt',1,2,1000),
(1003,N'Андреев А.А.','19820417','a.andreev@test.tt',4,3,1000)
```

Дополнительно стоит отметить, что в некластерный индекс можно включать значения при помощи указания их в INCLUDE. Т.е. в данном случае INCLUDE-индекс чем-то будет напоминать кластерный индекс, только теперь не индекс прикручен к таблице, а необходимые значения прикручены к индексу. Соответственно, такие индексы могут очень повысить производительность запросов на выборку (SELECT), если все перечисленные поля имеются в индексе, то возможно обращений к таблице вообще не понадобится. Но это естественно повышает размер индекса, т.к. значения перечисленных полей дублируются в индексе.

Вырезка из MSDN. *Общий синтаксис команды для создания индексов*

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
    ON <object> ( column [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( column_name [ ,...n ] ) ]
```

Подытожим

Индексы могут повысить скорость выборки данных (SELECT), но индексы уменьшают скорость модификации данных таблицы, т.к. после каждой модификации системе будет необходимо перестроить все индексы для конкретной таблицы.

Желательно в каждом случае найти оптимальное решение, золотую середину, чтобы и производительность выборки, так и модификации данных была на должном уровне. Стратегия по созданию индексов и их количества может зависеть от многих факторов, например, насколько часто изменяются данные в таблице.

Заключение по DDL

Как можно увидеть, язык DDL не так сложен, как может показаться на первый взгляд. Здесь я смог показать практически все его основные конструкции, оперируя всего тремя таблицами.

Главное — понять суть, а остальное дело практики.

Удачи вам в освоении этого замечательного языка под названием SQL.

Часть вторая

DML – Data Manipulation Language (язык манипулирования данными)

В первой части мы уже немного затронули язык DML, применяя почти весь набор его команд, за исключением команды MERGE.

Рассказывать про DML я буду по своей последовательности выработанной на личном опыте. По ходу, так же постараюсь рассказать про «скользкие» места, на которые стоит акцентировать внимание, эти «скользкие» места, схожи во многих диалектах языка SQL.

Т.к. учебник посвящается широкому кругу читателей (не только программистам), то и объяснение, порой будет соответствующее, т.е. долгое и нудное. Это мое видение материала, которое в основном получено на практике в результате профессиональной деятельности.

Основная цель данного учебника, шаг за шагом, выработать полное понимание сути языка SQL и научить правильно применять его конструкции. Профессионалам в этой области, может тоже будет интересно пролистать данный материал, может и они смогут вынести для себя что-то новое, а может просто, будет полезно почитать в целях освежить память. Надеюсь, что всем будет интересно.

Т.к. DML в диалекте БД MS SQL очень сильно связан с синтаксисом конструкции SELECT, то я начну рассказывать о DML именно с нее. На мой взгляд конструкция SELECT является самой главной конструкцией языка DML, т.к. за счет нее или ее частей осуществляется выборка необходимых данных из БД.

Язык DML содержит следующие конструкции:

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

В данной части, мы рассмотрим, только базовый синтаксис команды SELECT, который выглядит следующим образом:

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

Тема оператора SELECT очень обширная, поэтому в данной части я и остановлюсь только на его базовых конструкциях. Я считаю, что, не зная хорошо базы, нельзя приступать к изучению более сложных конструкций, т.к. дальше все будет крутиться вокруг этой базовой конструкции (подзапросы, объединения и т.д.).

Также в рамках этой части, я еще расскажу о предложении TOP. Это предложение я намерено не указал в базовом синтаксисе, т.к. оно реализуется по-разному в разных диалектах языка SQL.

Если язык DDL больше статичен, т.е. при помощи него создаются жесткие структуры (таблицы, связи и т.п.), то язык DML носит динамический характер, здесь правильные результаты вы можете получить разными путями.

Обучение так же 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. Если что-то остается непонятным, например, значение какой-нибудь функции, то обращайтесь за помощью в интернет.

Примеры будут показываться на БД Test, которая была создана при помощи DDL+DML в первой части.

Для тех, кто не создавал БД в первой части (т.к. не всех может интересовать язык DDL), может воспользоваться следующим скриптом:

```
-- создание БД
CREATE DATABASE Test
GO

-- сделать БД Test текущей
USE Test
GO

-- создаем таблицы справочники
CREATE TABLE Positions(
  ID int IDENTITY(1,1) NOT NULL CONSTRAINT PK_Positions PRIMARY KEY,
  Name nvarchar(30) NOT NULL
)

CREATE TABLE Departments(
  ID int IDENTITY(1,1) NOT NULL CONSTRAINT PK_Departments PRIMARY KEY,
  Name nvarchar(30) NOT NULL
)
GO

-- заполняем таблицы справочники данными
SET IDENTITY_INSERT Positions ON
INSERT Positions(ID,Name)VALUES
(1,N'Бухгалтер'),
(2,N'Директор'),
(3,N'Программист'),
(4,N'Старший программист')
SET IDENTITY_INSERT Positions OFF
GO
```

```
SET IDENTITY_INSERT Departments ON
INSERT Departments(ID,Name)VALUES
(1,N'Администрация'),
(2,N'Бухгалтерия'),
(3,N'ИТ')
SET IDENTITY_INSERT Departments OFF
GO

-- создаем таблицу с сотрудниками
CREATE TABLE Employees(
  ID int NOT NULL,
  Name nvarchar(30),
  Birthday date,
  Email nvarchar(30),
  PositionID int,
  DepartmentID int,
  HireDate date NOT NULL CONSTRAINT DF_Employees_HireDate DEFAULT SYSDATETIME(),
  ManagerID int,
CONSTRAINT PK_Employees PRIMARY KEY (ID),
CONSTRAINT FK_Employees_DepartmentID FOREIGN KEY(DepartmentID) REFERENCES Departments(ID),
CONSTRAINT FK_Employees_PositionID FOREIGN KEY(PositionID) REFERENCES Positions(ID),
CONSTRAINT FK_Employees_ManagerID FOREIGN KEY (ManagerID) REFERENCES Employees(ID),
CONSTRAINT UQ_Employees_Email UNIQUE(Email),
CONSTRAINT CK_Employees_ID CHECK(ID BETWEEN 1000 AND 1999),
INDEX IDX_Employees_Name(Name)
)
GO

-- заполняем ее данными
INSERT Employees (ID,Name,Birthday,Email,PositionID,DepartmentID,ManagerID)VALUES
(1000,N'Иванов И.И.','19550219','i.ivanov@test.tt',2,1,NULL),
(1001,N'Петров П.П.','19831203','p.petrov@test.tt',3,3,1003),
(1002,N'Сидоров С.С.','19760607','s.sidorov@test.tt',1,2,1000),
(1003,N'Андреев А.А.','19820417','a.andreev@test.tt',4,3,1000)
```

Все, теперь мы готовы приступить к изучению языка DML.

SELECT – оператор выборки данных

Первым делом, для активного редактора запроса, сделаем текущей БД Test, выбрав ее в выпадающем списке или же командой «USE Test».

Начнем с самой элементарной формы SELECT:

```
SELECT *
FROM Employees
```

В данном запросе мы просим вернуть все столбцы (на это указывает «*») из таблицы Employees – можно прочесть это как «ВЫБЕРИ все_поля ИЗ таблицы_сотрудники». В случае наличия кластерного индекса, возвращенные данные, скорее всего будут отсортированы по нему, в данном случае по колонке ID (но это не суть важно, т.к. в большинстве случаев сортировку мы будем указывать в явном виде сами при помощи ORDER BY …):

ID	Name	Birthday	Email	PositionID	DepartmentID	HireDate	ManagerID
1000	Иванов И.И.	1955-02-19	i.ivanov@test.tt	2	1	2015-04-08	NULL
1001	Петров П.П.	1983-12-03	p.petrov@test.tt	3	3	2015-04-08	1003
1002	Сидоров С.С.	1976-06-07	s.sidorov@test.tt	1	2	2015-04-08	1000
1003	Андреев А.А.	1982-04-17	a.andreev@test.tt	4	3	2015-04-08	1000

Вообще стоит сказать, что в диалекте MS SQL самая простая форма запроса SELECT может не содержать блока FROM, в этом случае вы можете использовать ее, для получения каких-то значений:

```
SELECT
    5550/100*15,
    SYSDATETIME(), -- получение системной даты БД
    SIN(0)+COS(0)
```

(No column name)	(No column name)	(No column name)
825	2015-04-11 12:12:36.0406743	1

Обратите внимание, что выражение (5550/100*15) дало результат 825, хотя если мы посчитаем на калькуляторе получится значение (832.5). Результат 825 получился по той причине, что в нашем выражении все числа целые, поэтому и результат целое число, т.е. (5550/100) дает нам 55, а не (55.5).

Запомните следующее, что в MS SQL работает следующая логика:

- Целое / Целое = Целое (т.е. в данном случае происходит целочисленное деление)
- Вещественное / Целое = Вещественное
- Целое / Вещественное = Вещественное

Т.е. результат преобразуется к большему типу, поэтому в 2-х последних случаях мы получаем вещественное число (рассуждайте как в математике – диапазон вещественных чисел больше диапазона целых, поэтому и результат преобразуется к нему):

```
SELECT
  123/10, -- 12
  123./10, -- 12.3
  123/10. -- 12.3
```

Здесь (123.) = (123.0), просто в данном случае 0 можно отбросить и оставить только точку.

При других арифметических операциях действует та же самая логика, просто в случае деления этот нюанс более актуален.

Поэтому обращайте внимание на тип данных числовых столбцов. В том случае если он целый, а результат вам нужно получить вещественный, то используйте преобразование, либо просто ставьте точку после числа указанного в виде константы (123.).

Для преобразования полей можно использовать функцию CAST или CONVERT. Для примера воспользуемся полем ID, оно у нас типа int:

```
SELECT
  ID,
  ID/100, -- здесь произойдет целочисленное деление
  CAST(ID AS float)/100, -- используем функцию CAST для преобразования в тип float
  CONVERT(float,ID)/100, -- используем функцию CONVERT для преобразования в тип float
  ID/100. -- используем преобразование за счет указания что знаменатель вещественное число
FROM Employees
```

ID	(No column name)	(No column name)	(No column name)	(No column name)
1000	10	10	10	10.000000
1001	10	10.01	10.01	10.010000
1002	10	10.02	10.02	10.020000
1003	10	10.03	10.03	10.030000

На заметку. *В БД ORACLE синтаксис без блока FROM недопустим, там для этой цели используется системная таблица DUAL, которая содержит одну строку:*

```
SELECT
    5550/100*15, -- а в ORACLE результат будет равен 832.5
    sysdate,
    sin(0)+cos(0)
FROM DUAL
```

Примечание. *Имя таблицы во многих РБД может предваряться именем схемы:*

```
SELECT *
FROM dbo.Employees -- dbo – имя схемы
```

Схема – это логическая единица БД, которая имеет свое наименование и позволяет сгруппировать внутри себя объекты БД такие как таблицы, представления и т.д.

Определение схемы в разных БД может отличатся, где-то схема непосредственно связанна с пользователем БД, т.е. в данном случае можно сказать, что схема и пользователь – это синонимы и все создаваемые в схеме объекты по сути являются объектами данного пользователя. В MS SQL схема – это независимая логическая единица, которая может быть создана сама по себе (см. CREATE SCHEMA).

По умолчанию в базе MS SQL создается одна схема с именем dbo (Database Owner) и все создаваемые объекты по умолчанию создаются именно в данной схеме. Соответственно, если мы в запросе указываем просто имя таблицы, то она будет искаться в схеме dbo текущей БД. Если мы хотим создать объект в конкретной схеме, мы должны будем так же предварить имя объекта именем схемы, например, «CREATE TABLE имя_схемы.имя_таблицы(…)».

В случае MS SQL имя схемы может еще предваряться именем БД, в которой находится данная схема:

```
SELECT *
FROM Test.dbo.Employees -- имя_базы.имя_схемы.таблица
```

Такое уточнение бывает полезным, например, если:

- *в одном запросе мы обращаемся к объектам расположенных 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*
- *и т.п.*

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

Так же не забываем, что 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.

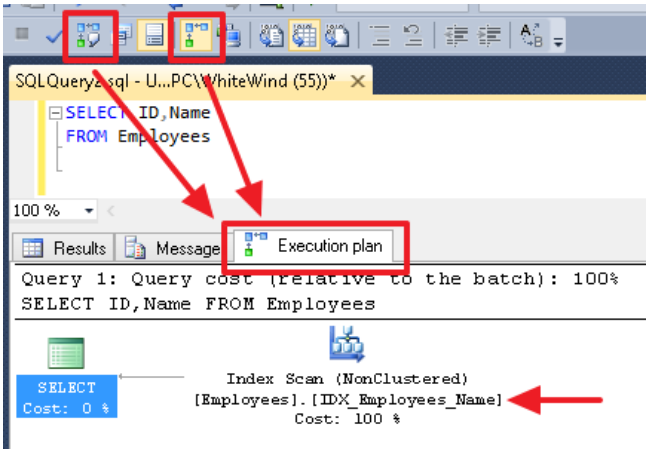
Если столбцов в таблице очень много, а особенно, если в таблице еще очень много строк, плюс к тому если мы делаем запросы к БД по сети, то предпочтительней будет выборка с непосредственным перечислением необходимых вам полей через запятую:

```
SELECT ID,Name
FROM Employees
```

Т.е. здесь мы говорим, что нам из таблицы нужно вернуть только поля ID и Name. Результат будет следующим (кстати оптимизатор здесь решил воспользоваться индексом, созданным по полю Name):

ID	Name
1003	Андреев А.А.
1000	Иванов И.И.
1001	Петров П.П.
1002	Сидоров С.С.

На заметку. *Порой бывает полезным посмотреть на то как осуществляется выборка данных, например, чтобы выяснить какие индексы используются. Это можно сделать если нажать кнопку «Display Estimated Execution Plan – Показать расчетный план» или установить «Include Actual Execution Plan – Включить в результат актуальный план выполнения запроса» (в данном случае мы сможем увидеть уже реальный план, соответственно, только после выполнения запроса):*



Анализ плана выполнения очень полезен при оптимизации запроса, он позволяет выяснить каких индексов не хватает или же какие индексы вообще не используются и их можно удалить.

Если вы только начали осваивать DML, то сейчас для вас это не так важно, просто возьмите на заметку и можете спокойно забыть об этом (может это вам никогда и не пригодится) – наша первоначальная цель изучить основы языка DML и научится правильно применять их, а оптимизация это уже отдельное искусство. Порой важнее, чтобы на руках просто был

правильно написанный запрос, который возвращает правильные результат с предметной точки зрения, а его оптимизацией уже занимаются отдельные люди. Для начала вам нужно научиться просто правильно писать запросы, используя любые средства для достижения цели. Главная цель которую вы сейчас должны достичь – чтобы ваш запрос возвращал правильные результаты.

Задание псевдонимов для таблиц

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

```
SELECT Employees.ID,Employees.Name
FROM Employees
```

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 emp.ID,emp.Name
FROM Employees AS emp
```

или

```
SELECT emp.ID,emp.Name
FROM Employees emp -- ключевое слово AS можно отпустить (я предпочитаю такой вариант)
```

Здесь emp – псевдоним для таблицы Employees, который можно будет использоваться в контексте данного оператора SELECT. Т.е. можно сказать, что в контексте этого оператора SELECT мы задаем таблице новое имя.

Конечно, в данном случае результаты запросов будут точно такими же как и для «SELECT ID,Name FROM Employees». Для чего это нужно будет понятно дальше (даже не в этой части), пока просто запоминаем, что имя колонки можно предварять (уточнять) либо непосредственно именем таблицы, либо при помощи псевдонима. Здесь можно использовать одно из двух, т.е. если вы задали псевдоним, то и пользоваться нужно будет им, а использовать имя таблицы уже нельзя.

На заметку. *В ORACLE допустим только вариант задания псевдонима таблицы без ключевого слова AS.*

DISTINCT – отброс строк дубликатов

Ключевое слово DISTINCT используется для того чтобы отбросить из результата запроса строки дубликаты. Грубо говоря представьте, что сначала выполняется запрос без опции DISTINCT, а затем из результата выбрасываются все дубликаты. Продемонстрируем это для большей наглядности на примере:

```
-- создадим для демонстрации временную таблицу
CREATE TABLE #Trash(
  ID int NOT NULL PRIMARY KEY,
  Col1 varchar(10),
  Col2 varchar(10),
  Col3 varchar(10)
)

-- наполним данную таблицу всяким мусором
INSERT #Trash(ID,Col1,Col2,Col3)VALUES
(1,'A','A','A'), (2,'A','B','C'), (3,'C','A','B'), (4,'A','A','B'),
(5,'B','B','B'), (6,'A','A','B'), (7,'A','A','A'), (8,'C','A','B'),
(9,'C','A','B'), (10,'A','A','B'), (11,'A',NULL,'B'), (12,'A',NULL,'B')


-- посмотрим что возвращает запрос без опции DISTINCT
SELECT Col1,Col2,Col3
FROM #Trash


-- посмотрим что возвращает запрос с опцией DISTINCT
SELECT DISTINCT Col1,Col2,Col3
FROM #Trash


-- удалим временную таблицу
DROP TABLE #Trash
```

Наглядно это будет выглядеть следующим образом (все дубликаты помечены одним цветом):

```
SELECT Col1,Col2,Col3
FROM #Trash
```

Col1	Col2	Col3
A	A	A
A	B	C
C	A	B
A	A	B
B	B	B
A	A	B
A	A	A
C	A	B
C	A	B
A	A	B
A	NULL	B
A	NULL	B

```
SELECT DISTINCT Col1,Col2,Col3
FROM #Trash
```

Col1	Col2	Col3
A	NULL	B
A	A	A
A	A	B
A	B	C
B	B	B
C	A	B

Теперь давайте рассмотрим где это можно применить, на более практичном примере – вернем из таблицы Employees только уникальные идентификаторы отделов (т.е. узнаем ID отделов в которых числятся сотрудники):

```
SELECT DISTINCT DepartmentID
FROM Employees
```

DepartmentID
1
2
3

Здесь мы получили три строки, т.к. 2 сотрудника у нас числятся в одном отделе (ИТ).

Теперь узнаем в каких отделах, какие должности фигурируют:

```
SELECT DISTINCT DepartmentID,PositionID
FROM Employees
```

DepartmentID	PositionID
1	2
2	1
3	3
3	4

Здесь мы получили 4 строчки, т.к. повторяющихся комбинаций (DepartmentID, PositionID) в нашей таблице нет.

Ненадолго вернемся к DDL

Так как данных для демонстрационных примеров начинает не хватать, а рассказать хочется более обширно и понятно, то давайте чуть расширим нашу таблицу Employess. К тому же немного вспомним DDL, как говорится «повторение – мать учения», и плюс снова немного забежим вперед и применим оператор UPDATE:

```
-- создаем новые колонки
ALTER TABLE Employees ADD
  LastName nvarchar(30), -- фамилия
  FirstName nvarchar(30), -- имя
  MiddleName nvarchar(30), -- отчество
  Salary float, -- и конечно же ЗП в каких-то УЕ
  BonusPercent float -- процент для вычисления бонуса от оклада
GO

-- наполняем их данными (некоторые данные намерено пропущены)
UPDATE Employees
SET
  LastName=N'Иванов',FirstName=N'Иван',MiddleName=N'Иванович',
  Salary=5000,BonusPercent= 50
WHERE ID=1000 -- Иванов И.И.

UPDATE Employees
SET
  LastName=N'Петров',FirstName=N'Петр',MiddleName=N'Петрович',
  Salary=1500,BonusPercent= 15
WHERE ID=1001 -- Петров П.П.

UPDATE Employees
SET
  LastName=N'Сидоров',FirstName=N'Сидор',MiddleName=NULL,
  Salary=2500,BonusPercent=NULL
WHERE ID=1002 -- Сидоров С.С.

UPDATE Employees
SET
  LastName=N'Андреев',FirstName=N'Андрей',MiddleName=NULL,
  Salary=2000,BonusPercent= 30
WHERE ID=1003 -- Андреев А.А.
```

Убедимся, что данные обновились успешно:

```
SELECT *
FROM Employees
```

ID	Name	...	LastName	FirstName	MiddleName	Salary	BonusPercent
1000	Иванов И.И.		Иванов	Иван	Иванович	5000	50
1001	Петров П.П.		Петров	Петр	Петрович	1500	15
1002	Сидоров С.С.		Сидоров	Сидор	NULL	2500	NULL
1003	Андреев А.А.		Андреев	Андрей	NULL	2000	30

Задание псевдонимов для столбцов запроса

Думаю, здесь будет проще показать, чем написать:

```
SELECT
  -- даем имя вычисляемому столбцу
  LastName+' '+FirstName+' '+MiddleName AS ФИО,
  -- использование двойных кавычек, т.к. используется пробел
  HireDate AS "Дата приема",
  -- использование квадратных скобок, т.к. используется пробел
  Birthday AS [Дата рождения],
  -- слово AS не обязательно
  Salary ZP
FROM Employees
```

ФИО	Дата приема	Дата рождения	ZP
Иванов Иван Иванович	2015-04-08	1955-02-19	5000
Петров Петр Петрович	2015-04-08	1983-12-03	1500
NULL	2015-04-08	1976-06-07	2500
NULL	2015-04-08	1982-04-17	2000

Как видим заданные нами псевдонимы столбцов, отразились в заголовке результирующей таблицы. Собственно, это и есть основное предназначение псевдонимов столбцов.

Обратите внимание, т.к. у последних 2-х сотрудников не указано отчество (NULL значение), то результат выражения «LastName+' '+FirstName+' '+MiddleName» так же вернул нам NULL.

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

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

На заметку. *В случае ORACLE для объединения строк используется оператор «||» и конкатенация будет выглядеть как «LastName||' '||FirstName||' '||MiddleName». Для ORACLE стоит отметить, что у него для строковых типов есть исключение, для них NULL и пустая строка '' это одно и тоже, поэтому в ORACLE такое выражение вернет для последних 2-х сотрудников «Сидоров Сидор » и «Андреев Андрей ». На момент версии ORACLE 12c, насколько я знаю, опции которая изменяет такое поведение нет (если не прав, прошу поправить меня). Здесь мне сложно судить хорошо это или плохо, т.к. в одних случаях удобнее поведение NULL-строки как в MS SQL, а в других как в ORACLE.*

В ORACLE тоже допустимы все перечисленные выше псевдонимы столбцов, кроме […].

Для того чтобы не городить конструкцию с использованием функции ISNULL, в MS SQL мы можем применить функцию CONCAT. Рассмотрим и сравним 3 варианта:

```
SELECT
  LastName+' '+FirstName+' '+MiddleName FullName1,
  -- 2 варианта для замены NULL пустыми строками '' (получаем поведение как и в ORACLE)
  ISNULL(LastName,'')+' '+ISNULL(FirstName,'')+' '+ISNULL(MiddleName,'') FullName2,
  CONCAT(LastName,' ',FirstName,' ',MiddleName) FullName3
FROM Employees
```

FullName1	FullName2	FullName3
Иванов Иван Иванович	Иванов Иван Иванович	Иванов Иван Иванович
Петров Петр Петрович	Петров Петр Петрович	Петров Петр Петрович
NULL	Сидоров Сидор	Сидоров Сидор
NULL	Андреев Андрей	Андреев Андрей

В MS SQL псевдонимы еще можно задавать при помощи знака равенства:

```
SELECT
  'Дата приема'=HireDate, -- помимо "…" и […] можно использовать '…'
  [Дата рождения]=Birthday,
  ZP=Salary
FROM Employees
```

Использовать для задания псевдонима ключевое слово AS или же знак равенства, наверное, больше дело вкуса. Но при разборе чужих запросов, данные знания могут пригодиться.

Напоследок скажу, что для псевдонимов имена лучше задавать, используя только символы латиницы и цифры, избегая применения '…', "…" и […], то есть использовать те же правила, что мы использовали при наименовании таблиц. Дальше, в примерах я буду использовать только такие наименования и никаких '…', "…" и […].

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 — (a+b)*(x/(y-z)).

И еще раз повторюсь, что любая операция с NULL дает NULL, например: 10+NULL, NULL*15/3, 100/NULL – все это даст в результате NULL. Т.е. говоря просто неопределенное значение не может дать определенный результат. Учитывайте это при составлении запроса и при необходимости делайте обработку NULL значений функциями ISNULL, COALESCE:

```
SELECT
  ID,Name,
  Salary/100*BonusPercent AS Result1, -- без обработки NULL значений
  Salary/100*ISNULL(BonusPercent,0) AS Result2, -- используем функцию ISNULL
  Salary/100*COALESCE(BonusPercent,0) AS Result3 -- используем функцию COALESCE
FROM Employees
```

ID	Name	Result1	Result2	Result3
1000	Иванов И.И.	2500	2500	2500
1001	Петров П.П.	225	225	225
1002	Сидоров С.С.	NULL	0	0
1003	Андреев А.А.	600	600	600
1004	Николаев Н.Н.	NULL	0	0
1005	Александров А.А.	NULL	0	0

Немного расскажу о функции COALESCE:

```
COALESCE (expr1, expr2, ..., exprn) - Возвращает первое не NULL значение из списка значений.
```

Пример:

```
SELECT COALESCE(f1, f1*f2, f2*f3) val -- в данном случае вернется третье значение
FROM (SELECT null f1, 2 f2, 3 f3) q
```

В основном, я сосредоточусь на рассказе конструкций языка DML и по большей части не буду рассказывать о функциях, которые будут встречаться в примерах. Если вам непонятно, что делает та или иная функция поищите ее описание в интернет, можете даже поискать информацию сразу по группе функций, например, задав в поиске Google «MS SQL строковые функции», «MS SQL математические функции» или же «MS SQL функции обработки NULL». Информации по функциям очень много, и вы ее сможете без труда найти. Для примера, в библиотеке MSDN, можно узнать больше о функции COALESCE:

Вырезка из MSDN *Сравнение COALESCE и CASE*

Выражение COALESCE — синтаксический ярлык для выражения CASE. Это означает, что код COALESCE(expression1,...n) переписывается оптимизатором запросов как следующее выражение CASE:

```
CASE
   WHEN (expression1 IS NOT NULL) THEN expression1
   WHEN (expression2 IS NOT NULL) THEN expression2
   ...
   ELSE expressionN
END
```

Для примера рассмотрим, как можно воспользоваться остатком от деления (%). Данный оператор очень полезен, когда требуется разбить записи на группы. Например, вытащим всех сотрудников, у которых четные табельные номера (ID), т.е. те ID, которые делятся на 2:

```
SELECT ID,Name
FROM Employees
WHERE ID%2=0 -- остаток от деления на 2 равен 0
```

ID	Name
1000	Иванов И.И.
1004	Николаев Н.Н.
1002	Сидоров С.С.

ORDER BY – сортировка результата запроса

Предложение ORDER BY используется для сортировки результата запроса.

```
SELECT
  LastName,
  FirstName,
  Salary
FROM Employees
ORDER BY LastName,FirstName -- упорядочить результат по 2-м столбцам - по Фамилии, и после по Им
ени
```

LastName	FirstName	Salary
Андреев	Андрей	2000
Иванов	Иван	5000
Петров	Петр	1500
Сидоров	Сидор	2500

После имя поля в предложении ORDER BY можно задать опцию DESC, которая служит для сортировки этого поля в порядке убывания:

```
SELECT LastName,FirstName,Salary
FROM Employees
ORDER BY -- упорядочить в порядке
  Salary DESC, -- 1. убывания Заработной Платы
  LastName, -- 2. по Фамилии
  FirstName -- 3. по Имени
```

LastName	FirstName	Salary
Иванов	Иван	5000
Сидоров	Сидор	2500
Андреев	Андрей	2000
Петров	Петр	1500

Для заметки. *Для сортировки по возрастанию есть ключевое слово ASC, но так как сортировка по возрастанию применяется по умолчанию, то про эту опцию можно забыть (я не помню случая, чтобы я когда-то использовал эту опцию).*

Стоит отметить, что в предложении ORDER BY можно использовать и поля, которые не перечислены в предложении SELECT (кроме случая, когда используется DISTINCT, об этом случае я расскажу ниже). Для примера забегу немного вперед используя опцию TOP и покажу, как например, можно отобрать 3-х сотрудников у которых самая высокая ЗП, с учетом что саму ЗП в целях конфиденциальности я показывать не должен:

```
SELECT TOP 3 -- вернуть только 3 первые записи из всего результата

  ID,LastName,FirstName
FROM Employees
ORDER BY Salary DESC -- сортируем результат по убыванию Заработной Платы
```

ID	LastName	FirstName
1000	Иванов	Иван
1002	Сидоров	Сидор

Конечно здесь есть случай, что у нескольких сотрудников может быть одинаковая ЗП и тут сложно сказать каких именно трех сотрудников вернет данный запрос, это уже нужно решать с постановщиком задачи. Допустим, после обсуждения с постановщиком данной задачи, вы согласовали и решили использовать следующий вариант – сделать дополнительную сортировку по полю даты рождения (т.е. молодым у нас дорога), а если и дата рождения у нескольких сотрудников может совпасть (ведь такое тоже не исключено), то можно сделать третью сортировку по убыванию значений ID (в последнюю очередь под выборку попадут те, у кого ID окажется максимальным – например, те кто был принят последним, допустим табельные номера у нас выдаются последовательно):

```
SELECT TOP 3 -- вернуть только 3 первые записи из всего результата
  ID,LastName,FirstName
FROM Employees
ORDER BY
  Salary DESC, -- 1. сортируем результат по убыванию Заработной Платы
  Birthday, -- 2. потом по Дате рождения
  ID DESC -- 3. и для полной однозначности результата добавляем сортировку по ID
```

Т.е. вы должны стараться чтобы результат запроса был предсказуемым, чтобы вы могли в случае разбора полетов объяснить почему в «черный список» попали именно эти люди, т.е. все было выбрано честно, по утверждённым правилам.

Сортировать можно так же используя разные выражения в предложении ORDER BY:

```
SELECT LastName,FirstName
FROM Employees
ORDER BY CONCAT(LastName,' ',FirstName) -- используем выражение
```

Так же в ORDER BY можно использовать псевдонимы заданные для колонок:

```
SELECT CONCAT(LastName,' ',FirstName) fi
FROM Employees
ORDER BY fi -- используем псевдоним
```

Стоит отметить что в случае использования предложения DISTINCT, в предложении ORDER BY могут использоваться только колонки, перечисленные в блоке SELECT. Т.е. после применения операции DISTINCT мы получаем новый набор данных, с новым набором колонок. По этой причине, следующий пример не отработает:

```
SELECT DISTINCT
  LastName,FirstName,Salary
FROM Employees
ORDER BY ID -- ID отсутствует в итоговом наборе, который мы получили при помощи DISTINCT
```

Т.е. предложение ORDER BY применяется уже к итоговому набору, перед выдачей результата пользователю.

Примечание 1. *Так же в предложении ORDER BY можно использовать номера столбцов, перечисленных в SELECT:*

```
SELECT LastName,FirstName,Salary
FROM Employees
ORDER BY -- упорядочить в порядке
  3 DESC, -- 1. убывания Заработной Платы
  1, -- 2. по Фамилии
  2 -- 3. по Имени
```

Для начинающих выглядит удобно и заманчиво, но лучше забыть и никогда не использовать такой вариант сортировки.

Если в данном случае (когда поля явно перечислены), такой вариант еще допустим, то для случая с использованием «» такой вариант лучше никогда не применять. Почему – потому что, если кто-то, например, поменяет в таблице порядок столбцов, или удалит столбцы (и это нормальная ситуация), ваш запрос может так же работать, но уже неправильно, т.к. сортировка уже может идти по другим столбцам, и это коварно тем что данная ошибка может обнаружиться очень нескоро.*

В случае, если бы столбы были явно перечислены, то в вышеуказанной ситуации, запрос либо бы продолжал работать, но также правильно (т.к. все явно определено), либо бы он просто выдал ошибку, что данного столбца не существует.

Так что можете смело забыть, о сортировке по номерам столбцов.

Примечание 2.
В MS SQL при сортировке по возрастанию NULL значения будут отображаться первыми.

```
SELECT BonusPercent FROM Employees ORDER BY BonusPercent
```

Соответственно при использовании DESC они будут в конце

```
SELECT BonusPercent FROM Employees ORDER BY BonusPercent DESC
```

Если необходимо поменять логику сортировки NULL значений, то используйте выражения, например:

```
SELECT BonusPercent FROM Employees ORDER BY ISNULL(BonusPercent,100)
```

В ORACLE для этой цели предусмотрены 2 опции NULLS FIRST и NULLS LAST (применяется по умолчанию). Например:

```
SELECT BonusPercent FROM Employees ORDER BY BonusPercent DESC NULLS LAST
```

Обращайте на это внимание при переходе на ту или иную БД.

TOP – возврат указанного числа записей

Вырезка из MSDN. 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.

Обычно данное выражение используется с предложением ORDER BY и мы уже смотрели примеры, когда нужно было вернуть N-первых строк из результирующего набора.

Без ORDER BY обычно данное предложение применяется, когда нужно просто посмотреть на неизвестную нам таблицу, в которой может быть очень много записей, в этом случае мы можем, для примера, попросить вернуть нам только первые 10 строк, но для наглядности мы скажем только 2:

```
SELECT TOP 2
  *
FROM Employees
```

Так же можно указать слово PERCENT, для того чтобы вернулось соответствуй процент строк из результирующего набора:

```
SELECT TOP 25 PERCENT
  *
FROM Employees
```

На моей практике чаше применяется именно выборка по количеству строк.

Так же с TOP можно использовать опцию WITH TIES, которая поможет вернуть все строки в случае неоднозначной сортировки, т.е. это предложение вернет все строки, которые равны по составу строкам, которые попадают в выборку TOP N, в итоге строк может быть выбрано больше чем N. Давайте для демонстрации добавим еще одного «Программиста» с окладом 1500:

```
INSERT Employees(ID,Name,Email,PositionID,DepartmentID,ManagerID,Salary)
VALUES(1004,N'Николаев Н.Н.','n.nikolayev@test.tt',3,3,1003,1500)
```

и введем еще одного сотрудника без указания должности и отдела с окладом 2000:

```
INSERT Employees(ID,Name,Email,PositionID,DepartmentID,ManagerID,Salary)
VALUES(1005,N'Александров А.А.','a.alexandrov@test.tt',NULL,NULL,1000,2000)
```

Теперь давайте выберем при помощи опции WITH TIES всех сотрудников, у которых оклад совпадает с окладами 3-х сотрудников, с самым маленьким окладом (надеюсь дальше будет понятно, к чему я клоню):

```
SELECT TOP 3 WITH TIES
  ID,Name,Salary
FROM Employees
ORDER BY Salary
```

Здесь хоть и указано TOP 3, но запрос вернул 4 записи, т.к. значение Salary которое вернуло TOP 3 (1500 и 2000) оказалось у 4-х сотрудников. Наглядно это работает примерно следующим образом:

```
SELECT TOP 3
  ID,Name,Salary
FROM Employees
ORDER BY Salary
```

ID	Name	Salary
1001	Петров П.П.	1500
1004	Николаев Н.Н.	1500
1003	Андреев А.А.	2000

запрос записей с Salary=(1500,2000)

```
SELECT
  ID,Name,Salary
FROM Employees
ORDER BY Salary
```

ID	Name	Salary
1001	Петров П.П.	1500
1004	Николаев Н.Н.	1500
1005	Александров А.А.	2000
1003	Андреев А.А.	2000
1002	Сидоров С.С.	2500
1000	Иванов И.И.	5000

```
SELECT TOP 3 WITH TIES
  ID,Name,Salary
FROM Employees
ORDER BY Salary
```

ID	Name	Salary
1001	Петров П.П.	1500
1004	Николаев Н.Н.	1500
1005	Александров А.А.	2000
1003	Андреев А.А.	2000

На заметку.

В разных БД TOP реализуется разными способами, в MySQL для этого есть предложение LIMIT, в котором дополнительно можно задать начальное смещение.

В ORACLE 12c, тоже ввели свой аналог совмещающий функциональность TOP и LIMIT – ищите по словам «ORACLE OFFSET FETCH». До версии 12c для этой цели обычно использовался псевдостолбец ROWNUM.

А что же будет если применить одновременно предложения DISTINCT и TOP? На такие вопросы легко ответить, проводя эксперименты. В общем, не бойтесь и не ленитесь экспериментировать, т.к. большая часть познается именно на практике. Порядок слов в операторе SELECT следующий, первым идет DISTINCT, а после него идет TOP, т.е. если рассуждать логически и читать слева-направо, то первым применится отброс дубликатов, а потом уже по этому набору будет сделан TOP. Что-ж проверим и убедимся, что так и есть:

```
SELECT DISTINCT TOP 2
  Salary
FROM Employees
ORDER BY Salary
```

Salary
1500
2000

Т.е. в результате мы получили 2 самые маленькие зарплаты из всех. Конечно может быть случай что ЗП для каких-то сотрудников может быть не указанной (NULL), т.к. схема нам это позволяет. Поэтому в зависимости от задачи принимаем решение либо обработать NULL значения в предложении ORDER BY, либо просто отбросить все записи, у которых Salary равна NULL, а для этого переходим к изучению предложения WHERE.

WHERE – условие выборки строк

Данное предложение служит для фильтрации записей по заданному условию. Например, выберем всех сотрудников работающих в «ИТ» отделе (его ID=3):

```
SELECT ID,LastName,FirstName,Salary
FROM Employees
WHERE DepartmentID=3 -- ИТ
ORDER BY LastName,FirstName
```

ID	LastName	FirstName	Salary
1004	NULL	NULL	1500
1003	Андреев	Андрей	2000
1001	Петров	Петр	1500

Предложение WHERE пишется до команды ORDER BY.

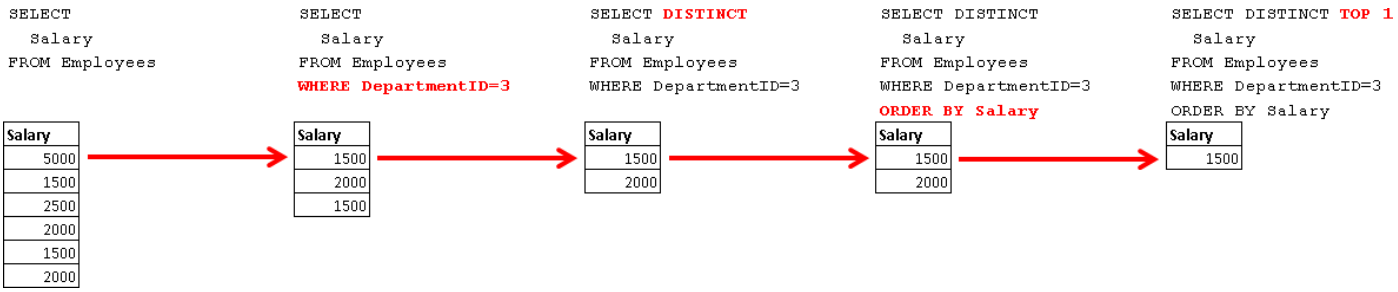
Порядок применения команд к исходному набору Employees следующий:

1. WHERE – если указано, то первым делом из всего набора Employees идет отбор только удовлетворяющих условию записей
2. DISTINCT – если указано, то отбрасываются все дубликаты
3. ORDER BY – если указано, то делается сортировка результата
4. TOP – если указано, то из отсортированного результата возвращается только указанное число записей

Рассмотрим для наглядности пример:

```
SELECT DISTINCT TOP 1
  Salary
FROM Employees
WHERE DepartmentID=3
ORDER BY Salary
```

Наглядно это будет выглядеть следующим образом:



Стоит отметить, что проверка на NULL делается не знаком равенства, а при помощи операторов IS NULL и IS NOT NULL. Просто запомните, что на NULL при помощи оператора «=» (знак равенства) сравнивать нельзя, т.к. результат выражения будет так же равен NULL.

Например, выберем всех сотрудников, у которых не указан отдел (т.е. DepartmentID IS NULL):

```
SELECT ID,Name
FROM Employees
WHERE DepartmentID IS NULL
```

ID	Name
1005	Александров А.А.

Теперь для примера посчитаем бонус для всех сотрудников у которых указано значение BonusPercent (т.е. BonusPercent IS NOT NULL):

```
SELECT ID,Name,Salary/100*BonusPercent AS Bonus
FROM Employees
WHERE BonusPercent IS NOT NULL
```

Да, кстати, если подумать, то значение BonusPercent может равняться нулю (0), а так же значение может быть внесено со знаком минус, ведь мы не накладывали на данное поле никаких ограничений.

Хорошо, рассказав о проблеме, нам пока сказали считать, что если (BonusPercent<=0 или BonusPercent IS NULL), то это означает что у сотрудника так же нет бонуса. Для начала, как нам сказали, так и сделаем, реализуем это при помощи логического оператора OR и NOT:

```
SELECT ID,Name,Salary/100*BonusPercent AS Bonus
FROM Employees
WHERE NOT(BonusPercent<=0 OR BonusPercent IS NULL)
```

Т.е. здесь мы начали изучать булевы операторы. Выражение в скобках «(BonusPercent<=0 OR BonusPercent IS NULL)» проверяет на то что у сотрудника нет бонуса, а NOT инвертирует это значение, т.е. говорит «верни всех сотрудников которые не сотрудники у которых нет бонуса».

Так же данное выражение можно переписать и сразу сказав сразу «верни всех сотрудников, у которых есть бонус» выразив это выражением (BonusPercent>0 и BonusPercent IS NOT NULL):

```
SELECT ID,Name,Salary/100*BonusPercent AS Bonus
FROM Employees
WHERE BonusPercent>0 AND BonusPercent IS NOT NULL
```

Также в блоке WHERE можно делать проверку разного рода выражений с применением арифметических операторов и функций. Например, аналогичную проверку можно сделать, использовав выражение с функцией ISNULL:

```
SELECT ID,Name,Salary/100*BonusPercent AS Bonus
FROM Employees
WHERE ISNULL(BonusPercent,0)>0
```

Булевы операторы и простые операторы сравнения

Да, без математики здесь не обойтись, поэтому сделаем небольшой экскурс по булевым и простым операторам сравнения.

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

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

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 (…)
```

Так же при помощи использования круглых скобок, можно изменить стандартную последовательность вычислений.

Здесь я постарался дать представление о булевой алгебре в достаточном для работы объеме. Как видите, чтобы писать условия посложнее без логики уже не обойтись, но ее здесь немного (AND, OR и NOT) и придумывали ее люди, так что все достаточно логично.

Идем к завершению второй части

Как видите даже про базовый синтаксис оператора SELECT можно говорить очень долго, но, чтобы остаться в рамках статьи, напоследок я покажу дополнительные логических операторы – BETWEEN, IN и LIKE.

BETWEEN – проверка на вхождение в диапазон

Этот оператор имеет следующий вид:

```
проверяемое_значение [NOT] BETWEEN начальное_ значение AND конечное_ значение
```

В роли значений могут выступать выражения.

Разберем на примере:

```
SELECT ID,Name,Salary
FROM Employees
WHERE Salary BETWEEN 2000 AND 3000 -- у кого ЗП в диапазоне 2000-3000
```

ID	Name	Salary
1002	Сидоров С.С.	2500
1003	Андреев А.А.	2000
1005	Александров А.А.	2000

Собственно, BETWEEN это упрощенная запись вида:

```
SELECT ID,Name,Salary
FROM Employees
WHERE Salary>=2000 AND Salary<=3000 -- все у кого ЗП в диапозоне 2000-3000
```

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

```
SELECT ID,Name,Salary
FROM Employees
WHERE Salary NOT BETWEEN 2000 AND 3000 -- аналогично выражению NOT(Salary>=2000 AND Salary<=3000
)
```

Соответственно, в случае использования BETWEEN, IN, LIKE вы можете так же объединять их с другими условиями при помощи AND и OR:

```
SELECT ID,Name,Salary
FROM Employees
WHERE Salary BETWEEN 2000 AND 3000 -- у кого ЗП в диапазоне 2000-3000
  AND DepartmentID=3 -- учитывать сотрудников только отдела 3
```

IN – проверка на вхождение в перечень значений

Этот оператор имеет следующий вид:

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

Думаю, проще показать на примере:

```
SELECT ID,Name,Salary
FROM Employees
WHERE PositionID IN(3,4) -- у кого должность равна 3 или 4
```

ID	Name	Salary
1001	Петров П.П.	1500
1003	Андреев А.А.	2000
1004	Николаев Н.Н.	1500

Т.е. по сути это аналогично следующему выражению:

```
SELECT ID,Name,Salary
FROM Employees
WHERE PositionID=3 OR PositionID=4 -- у кого должность равна 3 или 4
```

В случае NOT это будет аналогично (получим всех кроме тех, кто из отдела 3 и 4):

```
SELECT ID,Name,Salary
FROM Employees
WHERE PositionID NOT IN(3,4) -- аналогично выражению NOT(PositionID=3 OR PositionID=4)
```

Так же запрос с NOT IN можно выразить и через AND:

```
SELECT ID,Name,Salary
FROM Employees
WHERE PositionID<>3 AND PositionID<>4 -- равносильно PositionID NOT IN(3,4)
```

Учтите, что искать NULL значения при помощи конструкции IN не получится, т.к. проверка NULL=NULL вернет так же NULL, а не True:

```
SELECT ID,Name,DepartmentID
FROM Employees
WHERE DepartmentID IN(1,2,NULL) -- NULL записи не войдут в результат
```

В этом случае разбивайте проверку на несколько условий:

```
SELECT ID,Name,DepartmentID
FROM Employees
WHERE DepartmentID IN(1,2) -- 1 или 2
   OR DepartmentID IS NULL -- или NULL
```

Или же можно написать что-то вроде:

```
SELECT ID,Name,DepartmentID
FROM Employees
WHERE ISNULL(DepartmentID,-1) IN(1,2,-1) -- если вы уверены, что в нет и не будет департамента с
ID=-1
```

Думаю, первый вариант, в данном случае будет более правильным и надежным. Ну ладно, это всего лишь пример, для демонстрации того какие еще конструкции можно строить.

Так же стоит упомянуть еще более коварную ошибку, связанную с NULL, которую можно допустить при использовании конструкции NOT IN. Для примера, давайте попробуем выбрать всех сотрудников, кроме тех, у которых отдел равен 1 или у которых отдел вообще не указан, т.е. равен NULL. В качестве решения напрашивается вариант:

```
SELECT ID,Name,DepartmentID
FROM Employees
WHERE DepartmentID NOT IN(1,NULL)
```

Но выполнив запрос, мы не получим ни одной строки, хотя мы ожидали увидеть следующее:

ID	Name	DepartmentID
1001	Петров П.П.	3
1002	Сидоров С.С.	2
1003	Андреев А.А.	3
1004	Николаев Н.Н.	3

Опять же шутку здесь сыграло NULL указанное в списке значений.

Разберем почему в данном случае возникла логическая ошибка. Разложим запрос при помощи AND:

```
SELECT ID,Name,DepartmentID
FROM Employees
WHERE DepartmentID<>1
  AND DepartmentID<>NULL -- проблема из-за этой проверки на NULL - это условие всегда вернет NUL
L
```

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

Т.е. при выполнении левого условия (DepartmentID<>1) из-за неопределенного правого условия в результате мы получим неопределенное значение всего выражения (DepartmentID<>1 AND DepartmentID<>NULL), поэтому строка не войдет в результат.

Переписать условие правильно можно следующим образом:

```
SELECT ID,Name,DepartmentID
FROM Employees
WHERE DepartmentID NOT IN(1) -- или в данном случае просто DepartmentID<>1
  AND DepartmentID IS NOT NULL -- и отдельно проверяем на NOT NULL
```

IN еще можно использовать с подзапросами, но к такой форме мы вернемся, уже в последующих частях данного учебника.

LIKE – проверка строки по шаблону

Про данный оператор я расскажу только в самом простом виде, который является стандартом и поддерживается большинством диалектов языка SQL. Даже в таком виде при помощи него можно решить много задач, которые требуют выполнить проверку по содержимому строки.

Этот оператор имеет следующий вид:

```
проверяемая_строка [NOT] LIKE строка_шаблон [ESCAPE отменяющий_символ]
```

В «строке_шаблон» могут применятся следующие специальные символы:

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

Рассмотрим примеры с символом «%» (на практике, кстати он чаще применяется):

```
SELECT ID,Name
FROM Employees
WHERE Name LIKE 'Пет%' -- у кого имя начинается с букв "Пет"

SELECT ID,LastName
FROM Employees
WHERE LastName LIKE '%ов' -- у кого фамилия оканчивается на "ов"

SELECT ID,LastName
FROM Employees
WHERE LastName LIKE '%ре%' -- у кого фамилия содержит сочетание "ре"
```

Рассмотрим примеры с символом «_»:

```
SELECT ID,LastName
FROM Employees
WHERE LastName LIKE '_етров' -- у кого фамилия состоит из любого первого символа и последующих б
укв "етров"

SELECT ID,LastName
FROM Employees
WHERE LastName LIKE '____ов' -- у кого фамилия состоит из четырех любых символов и последующих б
укв "ов"
```

При помощи ESCAPE можно задать отменяющий символ, который отменяет проверяющее действие специальных символов «_» и «%». Данное предложение используется, когда в строке нужно непосредственно проверить наличие знака процента или знака подчеркивания.

Для демонстрации ESCAPE давайте занесем в одну запись мусор:

```
UPDATE Employees
SET
  FirstName='Это_мусор, содержащий %'
WHERE ID=1005
```

И посмотрим, что вернут следующие запросы:

```
SELECT *
FROM Employees
WHERE FirstName LIKE '%!%%' ESCAPE '!' -- строка содержит знак "%"

SELECT *
FROM Employees
WHERE FirstName LIKE '%!_%' ESCAPE '!' -- строка содержит знак "_"
```

В случае, если требуется проверить строку на полное совпадение, то вместо LIKE лучше использовать просто знак «=»:

```
SELECT *
FROM Employees
WHERE FirstName='Петр'
```

На заметку.
В MS SQL в шаблоне оператора LIKE так же можно задать поиск по регулярным выражениям, почитайте о нем в интернете, в том случае, если вам станет недостаточно стандартных возможностей данного оператора.

В ORACLE для поиска по регулярным выражениям применяется функция REGEXP_LIKE.

Немного о строках

В случае проверки строки на наличие Unicode символов, нужно будет ставить перед кавычками символ N, т.е. N'…'. Но так как у нас в таблице все символьные поля в формате Unicode (тип nvarchar), то для этих полей можно всегда использовать такой формат. Пример:

```
SELECT ID,Name
FROM Employees
WHERE Name LIKE N'Пет%'

SELECT ID,LastName
FROM Employees
WHERE LastName=N'Петров'
```

Если делать правильно, при сравнении с полем типа varchar (ASCII) нужно стараться использовать проверки с использованием '…', а при сравнении поля с типом nvarchar (Unicode) нужно стараться использовать проверки с использованием N'…'. Это делается для того, чтобы избежать в процессе выполнения запроса неявных преобразований типов. То же самое правило используем при вставке (INSERT) значений в поле или их обновлении (UPDATE).

При сравнении строк стоит учесть момент, что в зависимости от настройки БД (collation), сравнение строк может быть, как регистро-независимым (когда 'Петров'='ПЕТРОВ'), так и регистро-зависимым (когда 'Петров'<>'ПЕТРОВ').
В случае регистро-зависимой настройки, если требуется сделать поиск без учета регистра, то можно, например, сделать предварительное преобразование правого и левого выражения в один регистр – верхний или нижний:

```
SELECT ID,Name
FROM Employees
WHERE UPPER(Name) LIKE UPPER(N'Пет%') -- или LOWER(Name) LIKE LOWER(N'Пет%')

SELECT ID,LastName
FROM Employees
WHERE UPPER(LastName)=UPPER(N'Петров') -- или LOWER(LastName)=LOWER(N'Петров')
```

Немного о датах

При проверке на дату, вы можете 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:

```
SELECT ID,Name,Birthday
FROM Employees
WHERE Birthday BETWEEN '19800101' AND '19891231' -- сотрудники 80-х годов
ORDER BY Birthday
```

В некоторых случаях, дату удобнее задавать при помощи функции DATEFROMPARTS:

```
SELECT ID,Name,Birthday
FROM Employees
WHERE Birthday BETWEEN DATEFROMPARTS(1980,1,1) AND DATEFROMPARTS(1989,12,31)
ORDER BY Birthday
```

Так же есть аналогичная функция DATETIMEFROMPARTS, которая служит для задания Даты и Времени (для типа datetime).

Еще вы можете использовать функцию CONVERT, если требуется преобразовать строку в значение типа date или datetime:

```
SELECT
  CONVERT(date,'12.03.2015',104),
  CONVERT(datetime,'2014-11-30 17:20:15',120)
```

Значения 104 и 120, указывают какой формат даты используется в строке. Описание всех допустимых форматов вы можете найти в библиотеке MSDN задав в поиске «MS SQL CONVERT».

Функций для работы с датами в MS SQL очень много, ищите «ms sql функции для работы с датами».

Примечание. *Во всех диалектах языка SQL свой набор функций по работе с датами и применяется свой подход по работе с ними.*

Немного о числах и их преобразованиях

Информация этого раздела наверно больше будет полезна ИТ-специалистам. Если вы таковым не являетесь, а ваша цель просто научится писать запросы для получения из БД необходимой вам информации, то такие тонкости вам возможно и не понадобятся, но в любом случае можете бегло пройтись по тексту и взять что-то на заметку, т.к. если вы взялись за изучение SQL, то вы уже приобщаетесь к ИТ.

В отличие от функции преобразования CAST, в функции CONVERT можно задать третий параметр, который отвечает за стиль преобразования (формат). Для разных типов данных может использоваться свой набор стилей, которые могут повлиять на возвращаемый результат. Использование стилей мы уже затрагивали при рассмотрении преобразования строки функцией CONVERT в типы date и datetime.

Подробней про функции CAST, CONVERT и стили можно почитать в MSDN – «Функции CAST и CONVERT (Transact-SQL)»:msdn.microsoft.com/ru-ru/library/ms187928.aspx

Для упрощения примеров здесь будут использованы инструкции языка Transact-SQL – DECLARE и SET.

Конечно, в случае преобразования целого числа в вещественное (которое я привел вначале данного урока, в целях демонстрации разницы между целочисленным и вещественным делением), знание нюансов преобразования не так критично, т.к. там мы делали преобразование целого числа в вещественное (диапазон которого намного больше диапазона целых):

```
DECLARE @min_int int SET @min_int=-2147483648
DECLARE @max_int int SET @max_int=2147483647

SELECT
  -- (-2147483648)
  @min_int,CAST(@min_int AS float),CONVERT(float,@min_int),

  -- 2147483647
  @max_int,CAST(@max_int AS float),CONVERT(float,@max_int),

  -- numeric(16,6)
  @min_int/1., -- (-2147483648.000000)
  @max_int/1. -- 2147483647.000000
```

Возможно не стоило указывать способ неявного преобразования, получаемого делением на (1.), т.к. желательно стараться делать явные преобразования, для большего контроля типа получаемого результата. Хотя, в случае, если мы хотим получить результат типа numeric, с указанным количеством цифр после запятой, то мы можем в MS SQL применить трюк с умножением целого значения на (1., 1.0, 1.00 и т.д):

```
DECLARE @int int  SET @int=123

SELECT
  @int*1., -- numeric(12, 0) - 0 знаков после запятой
  @int*1.0, -- numeric(13, 1) - 1 знак
  @int*1.00, -- numeric(14, 2) - 2 знака

  -- хотя порой лучше сделать явное преобразование
  CAST(@int AS numeric(20, 0)), -- 123
  CAST(@int AS numeric(20, 1)), -- 123.0
  CAST(@int AS numeric(20, 2))  -- 123.00
```

В некоторых случаях детали преобразования могут быть действительно важны, т.к. они влияют на правильность полученного результата, например, в случае, когда делается преобразование числового значения в строку (varchar). Рассмотрим примеры по преобразованию значений типа money и float в varchar:

```
-- поведение при преобразовании money в varchar
DECLARE @money money
SET @money = 1025.123456789 -- произойдет неявное преобразование в 1025.1235, т.к. тип money хра
нит только 4 цифры после запятой

SELECT
  @money, -- 1025.1235
  -- по умолчанию CAST и CONVERT ведут себя одинаково (т.е. грубо говоря применяется стиль 0)
  CAST(@money as varchar(20)), -- 1025.12
  CONVERT(varchar(20), @money), -- 1025.12
  CONVERT(varchar(20), @money, 0), -- 1025.12 (стиль 0 - без разделителя тысячных и 2 цифры посл
е запятой (формат по умолчанию))

  CONVERT(varchar(20), @money, 1), -- 1,025.12 (стиль 1 - используется разделитель тысячных и 2
цифры после запятой)
  CONVERT(varchar(20), @money, 2) -- 1025.1235 (стиль 2 - без разделителя и 4 цифры после запято
й)
```

```
-- поведение при преобразовании float в varchar
DECLARE @float1 float SET @float1 = 1025.123456789
DECLARE @float2 float SET @float2 = 1231025.123456789

SELECT
  @float1, -- 1025.123456789
  @float2, -- 1231025.12345679
  -- по умолчанию CAST и CONVERT ведут себя одинаково (т.е. грубо говоря применяется стиль 0)
  -- стиль 0 - Не более 6 разрядов. По необходимости используется экспоненциальное представление чисел
  -- при преобразовании в varchar здесь творятся действительно страшные вещи
  CAST(@float1 as varchar(20)), -- 1025.12
  CONVERT(varchar(20), @float1), -- 1025.12
  CONVERT(varchar(20), @float1, 0), -- 1025.12

  CAST(@float2 as varchar(20)), -- 1.23103e+006
  CONVERT(varchar(20), @float2), -- 1.23103e+006
  CONVERT(varchar(20), @float2, 0), -- 1.23103e+006

  -- стиль 1 - Всегда 8 разрядов. Всегда используется экспоненциальное представление чисел.
  -- этот стиль для float тоже не очень точен
  CONVERT(varchar(20), @float1, 1), -- 1.0251235e+003
  CONVERT(varchar(20), @float2, 1), -- 1.2310251e+006

  -- стиль 2 - Всегда 16 разрядов. Всегда используется экспоненциальное представление чисел.
  -- здесь с точностью уже получше
  CONVERT(varchar(30), @float1, 2), -- 1.025123456789000e+003 - OK
  CONVERT(varchar(30), @float2, 2) -- 1.231025123456789e+006 - OK
```

Как видно из примера, плавающие типы float, real в некоторых случаях действительно могут создать большую погрешность, особенно при перегонке в строку и обратно (такое может быть при разного рода интеграциях, когда данные, например, передаются в текстовых файлах из одной системы в другую).

Если нужно явно контролировать точность до определенного знака, более 4-х, то для хранения данных, порой лучше использовать тип decimal/numeric. Если хватает 4-х знаков, то можно использовать и тип money – он примерно соотвествует numeric(20,4).

```
-- decimal и numeric
DECLARE @money money SET @money = 1025.123456789 -- 1025.1235
DECLARE @float1 float SET @float1 = 1025.123456789
DECLARE @float2 float SET @float2 = 1231025.123456789

DECLARE @numeric numeric(28,9) SET @numeric = 1025.123456789

SELECT
  CAST(@numeric as varchar(20)), -- 1025.12345679
  CONVERT(varchar(20), @numeric), -- 1025.12345679

  CAST(@money as numeric(28,9)), -- 1025.123500000
  CAST(@float1 as numeric(28,9)), -- 1025.123456789
  CAST(@float2 as numeric(28,9)) -- 1231025.123456789
```

Примечание.

С версии MS SQL 2008, можно использовать вместо конструкции:

```
DECLARE @money money
SET @money = 1025.123456789
```

Более короткий синтаксис инициализации переменных:

```
DECLARE @money money = 1025.123456789
```

Заключение второй части

В этой части, я постарался вспомнить и отразить наиболее важные моменты, касающиеся базового синтаксиса. Базовая конструкция – это костяк, без которого нельзя приступать к изучению более сложных конструкций языка SQL.

Надеюсь, данный материал поможет людям, делающим первые шаги в изучении языка SQL.

Удачи в изучении и применении на практике данного языка.

Часть третья

О чем будет рассказано в этой части

В этой части мы познакомимся:

1. с выражением CASE, которое позволяет включить условные выражения в запрос;
2. с агрегатными функциями, которые позволяют получить разного рода итоги (агрегированные значения) рассчитанные на основании детальных данных, полученных оператором «SELECT … WHERE …»;
3. с предложением GROUP BY, которое в скупе с агрегатными функциями позволяет получить итоги по детальным данным в разрезе групп;
4. с предложением HAVING, которое позволяет произвести фильтрацию по сгруппированным данным.

Выражение CASE – условный оператор языка SQL

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:	**Вторая форма:**
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

В качестве значений здесь могут выступать и выражения.

Разберем на примере первую форму CASE:

```
SELECT
  ID,Name,Salary,

  CASE
    WHEN Salary>=3000 THEN 'ЗП >= 3000'
    WHEN Salary>=2000 THEN '2000 <= ЗП < 3000'
    ELSE 'ЗП < 2000'
  END SalaryTypeWithELSE,

  CASE
    WHEN Salary>=3000 THEN 'ЗП >= 3000'
    WHEN Salary>=2000 THEN '2000 <= ЗП < 3000'
  END SalaryTypeWithoutELSE

FROM Employees
```

ID	Name	Salary	SalaryTypeWithELSE	SalaryTypeWithoutELSE
1000	Иванов И.И.	5000	ЗП >= 3000	ЗП >= 3000
1001	Петров П.П.	1500	ЗП < 2000	NULL
1002	Сидоров С.С.	2500	2000 <= ЗП < 3000	2000 <= ЗП < 3000
1003	Андреев А.А.	2000	2000 <= ЗП < 3000	2000 <= ЗП < 3000
1004	Николаев Н.Н.	1500	ЗП < 2000	NULL
1005	Александров А.А.	2000	2000 <= ЗП < 3000	2000 <= ЗП < 3000

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.

Разберем на примере вторую форму CASE:

Допустим, на новый год решили премировать всех сотрудников и попросили вычислить сумму бонусов по следующей схеме:

- Сотрудникам ИТ-отдела выдать по 15% от ЗП;
- Сотрудникам Бухгалтерии по 10% от ЗП;
- Всем остальным по 5% от ЗП.

Используем для данной задачи запрос с выражением CASE:

```
SELECT
  ID,Name,Salary,DepartmentID,

  -- для наглядности выведем процент в виде строки
  CASE DepartmentID -- проверяемое значение
    WHEN 2 THEN '10%' -- 10% от ЗП выдать Бухгалтерам
    WHEN 3 THEN '15%' -- 15% от ЗП выдать ИТ-шникам
    ELSE '5%' -- всем остальным по 5%
  END NewYearBonusPercent,

  -- построим выражение с использованием CASE, чтобы увидеть сумму бонуса
  Salary/100*
  CASE DepartmentID
    WHEN 2 THEN 10 -- 10% от ЗП выдать Бухгалтерам
    WHEN 3 THEN 15 -- 15% от ЗП выдать ИТ-шникам
    ELSE 5 -- всем остальным по 5%
  END BonusAmount

FROM Employees
```

ID	Name	Salary	DepartmentID	NewYearBonusPercent	BonusAmount
1000	Иванов И.И.	5000	1	5%	250
1001	Петров П.П.	1500	3	15%	225
1002	Сидоров С.С.	2500	2	10%	250
1003	Андреев А.А.	2000	3	15%	300
1004	Николаев Н.Н.	1500	3	15%	225
1005	Александров А.А.	2000	NULL	5%	100

Здесь делается последовательная проверка значения DepartmentID с WHEN-значениями. При достижении первого равенства DepartmentID 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 DepartmentID не совпал ни с одним WHEN-значением.

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

Вторую форму CASE несложно представить при помощи первой формы:

```
SELECT
  ID,Name,Salary,DepartmentID,

  CASE
    WHEN DepartmentID=2 THEN '10%' -- 10% от ЗП выдать Бухгалтерам
    WHEN DepartmentID=3 THEN '15%' -- 15% от ЗП выдать ИТ-шникам
    ELSE '5%' -- всем остальным по 5%
  END NewYearBonusPercent,

  -- построим выражение с использованием CASE, чтобы увидеть сумму бонуса
  Salary/100*
  CASE
    WHEN DepartmentID=2 THEN 10 -- 10% от ЗП выдать Бухгалтерам
    WHEN DepartmentID=3 THEN 15 -- 15% от ЗП выдать ИТ-шникам
    ELSE 5 -- всем остальным по 5%
  END BonusAmount

FROM Employees
```

Так что, 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.

Примечание. *Первая и вторая форма CASE входят в стандарт языка SQL, поэтому скорее всего они должны быть применимы во многих СУБД.*

С MS SQL версии 2012 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
```

Посмотрим на примере:

```
SELECT
  ID,Name,Salary,

  IIF(Salary>=2500,'ЗП >= 2500','ЗП < 2500') DemoIIF,

  CASE WHEN Salary>=2500 THEN 'ЗП >= 2500' ELSE 'ЗП < 2500' END DemoCASE

FROM Employees
```

Конструкции CASE, IIF могут быть вложенными друг в друга. Рассмотрим абстрактный пример:

```
SELECT
  ID,Name,Salary,

  CASE
    WHEN DepartmentID IN(1,2) THEN 'A'
    WHEN DepartmentID=3 THEN
                          CASE PositionID -- вложенный CASE
                            WHEN 3 THEN 'B-1'
                            WHEN 4 THEN 'B-2'
                          END
    ELSE 'C'
  END Demo1,

  IIF(DepartmentID IN(1,2),'A',
    IIF(DepartmentID=3,CASE PositionID WHEN 3 THEN 'B-1' WHEN 4 THEN 'B-2' END,'C')) Demo2

FROM Employees
```

Так как конструкция CASE и IIF представляют из себя выражение, которые возвращают результат, то мы можем использовать их не только в блоке SELECT, но и в остальных блоках, допускающих использование выражений, например, в блоках WHERE или ORDER BY.

Для примера, пускай перед нами поставили задачу – создать список на выдачу ЗП на руки, следующим образом:

- Первым делом ЗП должны получить сотрудники у кого оклад меньше 2500
- Те сотрудники у кого оклад больше или равен 2500, получают ЗП во вторую очередь
- Внутри этих двух групп нужно упорядочить строки по ФИО (поле Name)

Попробуем решить эту задачу при помощи добавления CASE-выражение в блок ORDER BY:

```
SELECT
  ID,Name,Salary
FROM Employees
ORDER BY
  CASE WHEN Salary>=2500 THEN 1 ELSE 0 END, -- выдать ЗП сначала тем у кого она ниже 2500
  Name -- дальше упорядочить список в порядке ФИО
```

ID	Name	Salary
1005	Александров А.А.	2000
1003	Андреев А.А.	2000
1004	Николаев Н.Н.	1500
1001	Петров П.П.	1500
1000	Иванов И.И.	5000
1002	Сидоров С.С.	2500

Как видим, Иванов и Сидоров уйдут с работы последними.

И абстрактный пример использования CASE в блоке WHERE:

```
SELECT
  ID,Name,Salary
FROM Employees
WHERE CASE WHEN Salary>=2500 THEN 1 ELSE 0 END=1 -- все записи у которых выражение равно 1
```

Можете попытаться самостоятельно переделать 2 последних примера с функцией IIF.
И напоследок, вспомним еще раз о NULL-значениях:

```
SELECT
  ID,Name,Salary,DepartmentID,

  CASE
    WHEN DepartmentID=2 THEN '10%' -- 10% от ЗП выдать Бухгалтерам
    WHEN DepartmentID=3 THEN '15%' -- 15% от ЗП выдать ИТ-шникам
    WHEN DepartmentID IS NULL THEN '-' -- внештатникам бонусов не даем (используем IS NULL)
    ELSE '5%' -- всем остальным по 5%
  END NewYearBonusPercent1,

  -- а так проверять на NULL нельзя, вспоминаем что говорилось про NULL во второй части
  CASE DepartmentID -- проверяемое значение
    WHEN 2 THEN '10%'
    WHEN 3 THEN '15%'
    WHEN NULL THEN '-' -- !!! в данном случае использование второй формы CASE не подходит
    ELSE '5%'
  END NewYearBonusPercent2

FROM Employees
```

ID	Name	Salary	DepartmentID	NewYearBonusPercent1	NewYearBonusPercent2
1000	Иванов И.И.	5000	1	5%	5%
1001	Петров П.П.	1500	3	15%	15%
1002	Сидоров С.С.	2500	2	10%	10%
1003	Андреев А.А.	2000	3	15%	15%
1004	Николаев Н.Н.	1500	3	15%	15%
1005	Александров А.А.	2000	NULL	-	5%

Конечно можно было переписать и как-то так:

```
SELECT
  ID,Name,Salary,DepartmentID,

  CASE ISNULL(DepartmentID,-1) -- используем замену в случае NULL на -1
    WHEN 2 THEN '10%'
    WHEN 3 THEN '15%'
    WHEN -1 THEN '-' -- если мы уверены, что отдела с ID равным (-1) нет и не будет
    ELSE '5%'
  END NewYearBonusPercent3

FROM Employees
```

В общем, полет фантазии в данном случае не ограничен.

Для примера посмотрим, как при помощи CASE и IIF можно смоделировать функцию ISNULL:

```
SELECT
  ID,Name,LastName,

  ISNULL(LastName,'Не указано') DemoISNULL,
  CASE WHEN LastName IS NULL THEN 'Не указано' ELSE LastName END DemoCASE,
  IIF(LastName IS NULL,'Не указано',LastName) DemoIIF
FROM Employees
```

Конструкция CASE очень мощное средство языка SQL, которое позволяет наложить дополнительную логику для расчета значений результирующего набора. В данной части владение CASE-конструкцией нам еще пригодится, поэтому в этой части в первую очередь внимание уделено именно ей.

Агрегатные функции

Здесь мы 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
SUM(столбец/выражение)	Возвращает сумму по значениям столбца/выражения
AVG(столбец/выражение)	Возвращает среднее значение по значениям столбца/выражения. NULL значения для подсчета не учитываются.
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(*) [Общее кол-во сотрудников],
  COUNT(DISTINCT DepartmentID) [Число уникальных отделов],
  COUNT(DISTINCT PositionID) [Число уникальных должностей],
  COUNT(BonusPercent) [Кол-во сотрудников у которых указан % бонуса],
  MAX(BonusPercent) [Максимальный процент бонуса],
  MIN(BonusPercent) [Минимальный процент бонуса],
  SUM(Salary/100*BonusPercent) [Сумма всех бонусов],
  AVG(Salary/100*BonusPercent) [Средний размер бонуса],
  AVG(Salary) [Средний размер ЗП]
FROM Employees
```

Общее кол-во сотрудников	Число уникальных отделов	Число уникальных должностей	Кол-во сотрудников у которых указан % бонуса	Максимальный процент бонуса	Минимальный процент бонуса	Сумма всех бонусов	Средний размер бонуса	Средний размер ЗП
6	3	4	3	50	15	3325	1108.33333333333	2416.66666666667

Для большей наглядности я решил здесь сделать исключение и воспользовался синтаксисом […] для задания псевдонимов колонок.

Разберем каким образом получилось каждое возвращенное значение, а за одно вспомним конструкции базового синтаксиса оператора SELECT.

Во-первых, т.к. мы в запросе не указали WHERE-условия, то итоги будут считаться для детальных данных, которые получаются запросом:

```
SELECT * FROM Employees
```

т.е. для всех строк таблицы Employees.

Для наглядности выберем только поля и выражения, которые используются в агрегатных функциях:

```
SELECT
  DepartmentID,
  PositionID,
  BonusPercent,
  Salary/100*BonusPercent [Salary/100*BonusPercent],
  Salary
FROM Employees
```

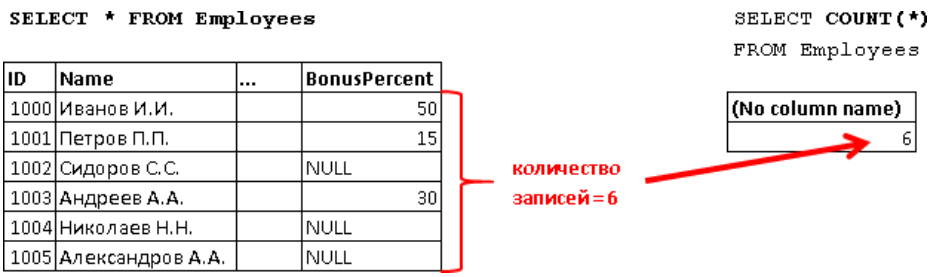
DepartmentID	PositionID	BonusPercent	Salary/100*BonusPercent	Salary
1	2	50	2500	5000
3	3	15	225	1500
2	1	NULL	NULL	2500
3	4	30	600	2000
3	3	NULL	NULL	1500
NULL	NULL	NULL	NULL	2000

Это исходные данные (детальные строки), по которым и будут считаться итоги агрегированного запроса.

Теперь разберем каждое агрегированное значение:

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 Employees
```



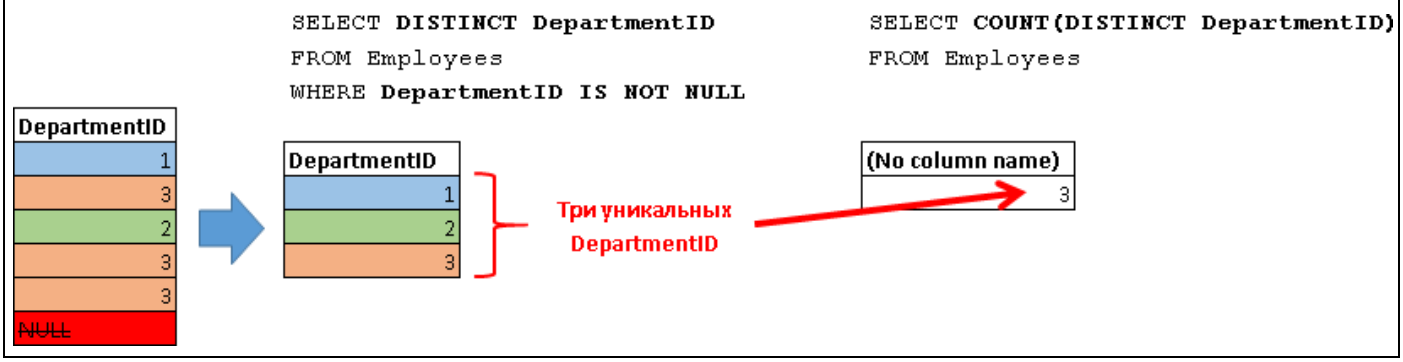
ID	Name	...	BonusPercent
1000	Иванов И.И.		50
1001	Петров П.П.		15
1002	Сидоров С.С.		NULL
1003	Андреев А.А.		30
1004	Николаев Н.Н.		NULL
1005	Александров А.А.		NULL

COUNT(DISTINCT DepartmentID) – вернуло нам значение 3, т.е. это число соответствует числу уникальных значений департаментов указанных в столбце DepartmentID без учета NULL значений. Пройдемся по значениям колонки DepartmentID и раскрасим одинаковые значения в один цвет (не стесняйтесь, для обучения все методы хороши):

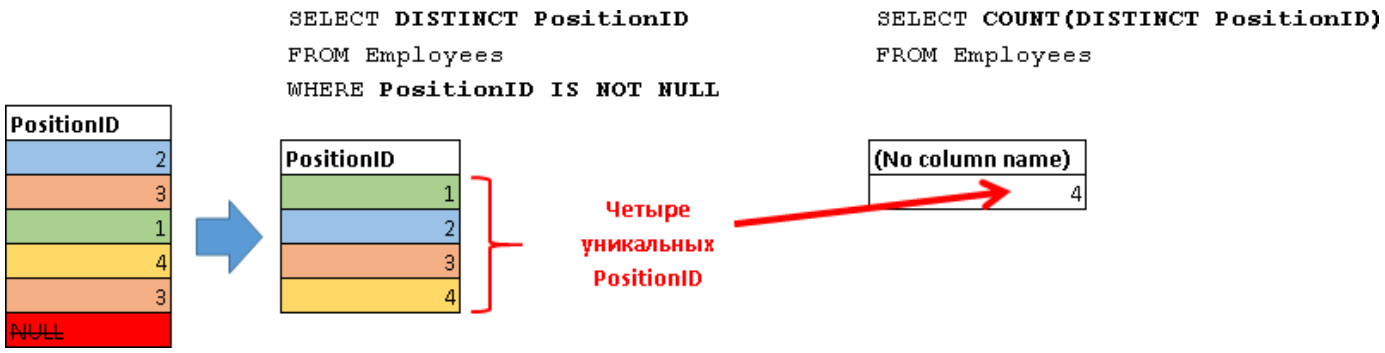
DepartmentID
1
3
2
3
3
~~NULL~~

Отбрасываем NULL, после чего, мы получили 3 уникальных значения (1, 2 и 3). Т.е. значение получаемое COUNT(DISTINCT DepartmentID), в развернутом виде можно представить следующей выборкой:

```
SELECT DISTINCT DepartmentID -- 2. берем только уникальные значения
FROM Employees
WHERE DepartmentID IS NOT NULL -- 1. отбрасываем NULL значения
```



COUNT(DISTINCT PositionID) – то же самое, что было сказано про COUNT(DISTINCT DepartmentID), только полю PositionID. Смотрим на значения колонки PositionID и не жалеем красок:



COUNT(BonusPercent) – возвращает количество строк, у которых указано значение BonusPercent, т.е. подсчитывается количество записей, у которых BonusPercent IS NOT NULL. Здесь нам будет проще, т.к. не нужно считать уникальные значения, достаточно просто отбросить записи с NULL значениями. Берем значения колонки BonusPercent и вычеркиваем все NULL значения:

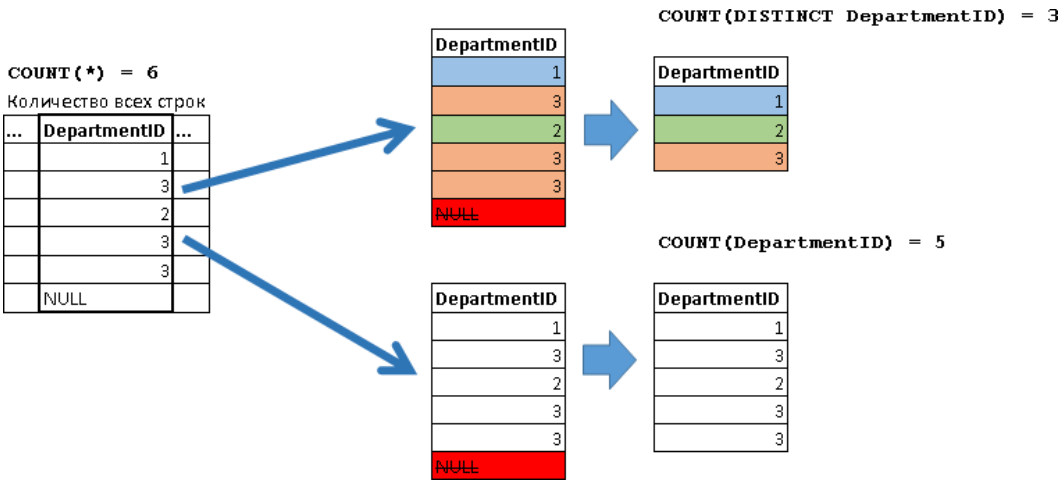
BonusPercent
50
15
~~NULL~~
30
~~NULL~~
~~NULL~~

Остается 3 значения. Т.е. в развернутом виде выборку можно представить так:

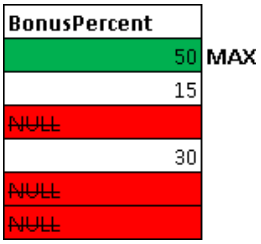
```
SELECT BonusPercent -- 2. берем все значения
FROM Employees
WHERE BonusPercent IS NOT NULL -- 1. отбрасываем NULL значения
```

Т.к. мы не использовали слова DISTINCT, то посчитаются и повторяющиеся BonusPercent в случае их наличия, без учета BonusPercent равных NULL. Для примера давайте сделаем сравнение результата с использованием DISTINCT и без него. Для большей наглядности воспользуемся значениями поля DepartmentID:

```
SELECT
  COUNT(*), -- 6
  COUNT(DISTINCT DepartmentID), -- 3
  COUNT(DepartmentID) -- 5
FROM Employees
```



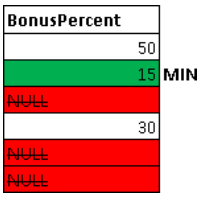
MAX(BonusPercent) – возвращает максимальное значение BonusPercent, опять же без учета NULL значений.
Берем значения колонки BonusPercent и ищем среди них максимальное значение, на NULL значения не обращаем внимания:



Т.е. мы получаем следующее значение:

```
SELECT TOP 1 BonusPercent
FROM Employees
WHERE BonusPercent IS NOT NULL
ORDER BY BonusPercent DESC -- сортируем по убыванию
```

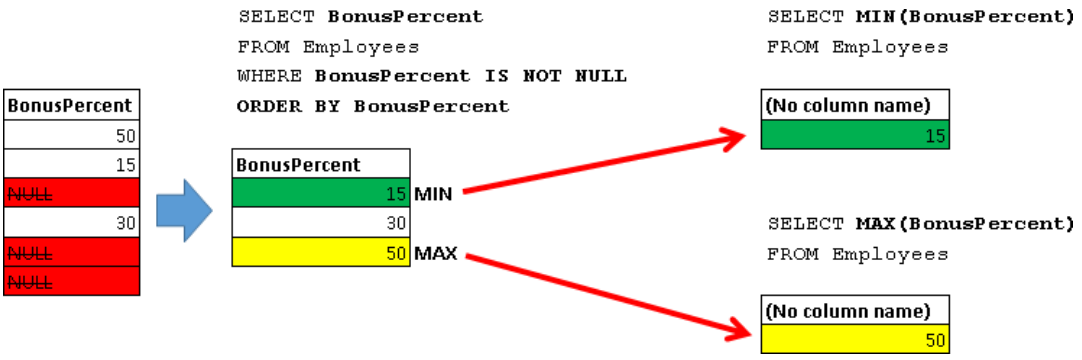
MIN(BonusPercent) – возвращает минимальное значение BonusPercent, опять же без учета NULL значений. Как в случае с MAX, только ищем минимальное значение, игнорируя NULL:



Т.е. мы получаем следующее значение:

```
SELECT TOP 1 BonusPercent
FROM Employees
WHERE BonusPercent IS NOT NULL
ORDER BY BonusPercent -- сортируем по возрастанию
```

Наглядное представление MIN(BonusPercent) и MAX(BonusPercent):

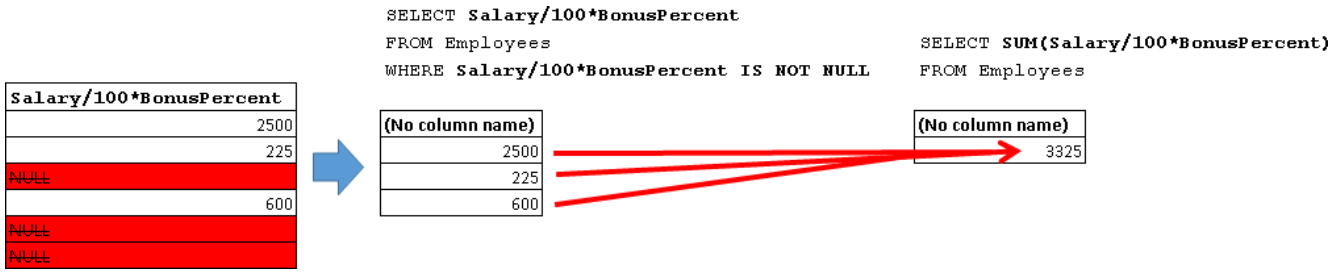


SUM(Salary/100*BonusPercent) – возвращает сумму всех не NULL значений. Разбираем значения выражения (Salary/100*BonusPercent):

Salary/100*BonusPercent
2500
225
~~NULL~~
600
~~NULL~~
~~NULL~~

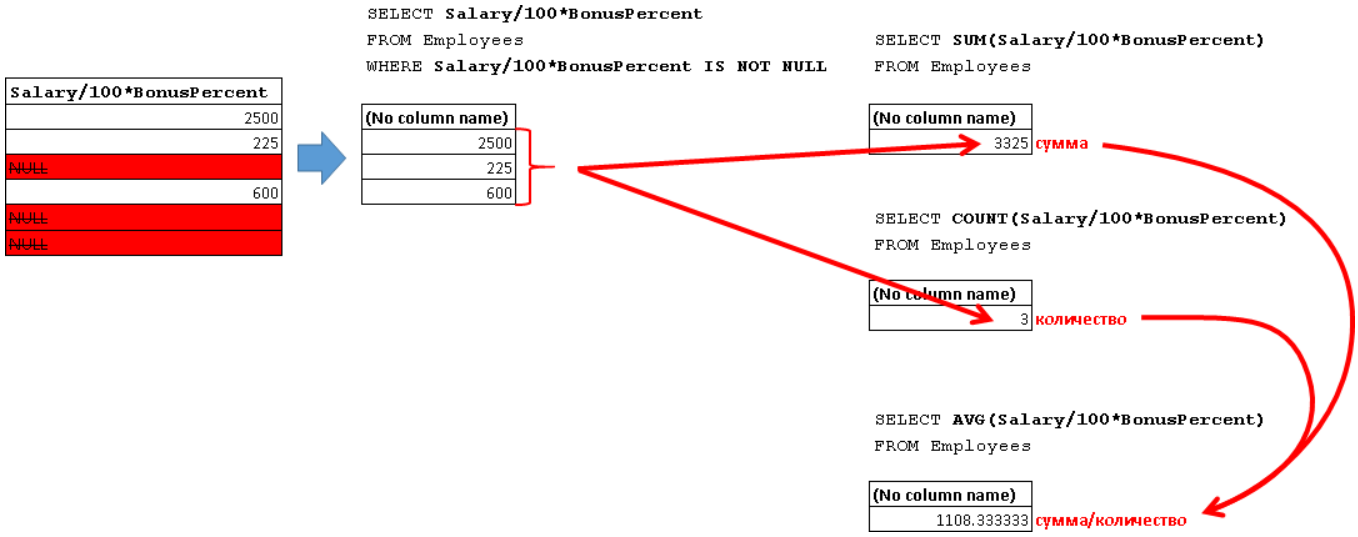
Т.е. происходит суммирование следующих значений:

```
SELECT Salary/100*BonusPercent
FROM Employees
WHERE Salary/100*BonusPercent IS NOT NULL
```



AVG(Salary/100*BonusPercent) – возвращает среднее значений. NULL-выражения не учитываются, т.е. это соответствует второму выражению:

```
SELECT
  AVG(Salary/100*BonusPercent), -- 1108.33333333333
  SUM(Salary/100*BonusPercent)/COUNT(Salary/100*BonusPercent), -- 1108.33333333333
  SUM(Salary/100*BonusPercent)/COUNT(*) -- 554.166666666667
FROM Employees
```



Т.е. опять же NULL-значения не учитываются при подсчете количества.

Если же вам необходимо вычислить среднее по всем сотрудникам, как в третьем выражении, которое дает 554.166666666667, то используйте предварительное преобразование NULL значений в ноль:

```
SELECT
  AVG(ISNULL(Salary/100*BonusPercent,0)), -- 554.166666666667
  SUM(Salary/100*BonusPercent)/COUNT(*) -- 554.166666666667
FROM Employees
```

AVG(Salary) – собственно, здесь все то же самое что и в предыдущем случае, т.е. если у сотрудника Salary равен NULL, то он не учтется. Чтобы учесть всех сотрудников, соответственно делаете предварительное преобразование NULL значений AVG(ISNULL(Salary,0))

Подведем некоторые итоги:

- COUNT(*) – служит для подсчета общего количества строк, которые получены оператором «SELECT … WHERE …»
- во всех остальных вышеперечисленных агрегатных функциях при расчете итога, NULL-значения не учитываются
- если нам нужно учесть все строки, это больше актуально для функции AVG, то предварительно необходимо осуществить обработку NULL значений, например, как было показано выше «AVG(ISNULL(Salary,0))»

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. Например, сделаем все тоже самое, но только в разрезе ИТ-отдела:

```
SELECT
  COUNT(*) [Общее кол-во сотрудников],
  COUNT(DISTINCT DepartmentID) [Число уникальных отделов],
  COUNT(DISTINCT PositionID) [Число уникальных должностей],
  COUNT(BonusPercent) [Кол-во сотрудников у которых указан % бонуса],
  MAX(BonusPercent) [Максимальный процент бонуса],
  MIN(BonusPercent) [Минимальный процент бонуса],
  SUM(Salary/100*BonusPercent) [Сумма всех бонусов],
  AVG(Salary/100*BonusPercent) [Средний размер бонуса],
  AVG(Salary) [Средний размер ЗП]
FROM Employees
WHERE DepartmentID=3 -- учесть только ИТ-отдел
```

Общее кол-во сотрудников	Число уникальных отделов	Число уникальных должностей	Кол-во сотрудников у которых указан % бонуса	Максимальный процент бонуса	Минимальный процент бонуса	Сумма всех бонусов	Средний размер бонуса	Средний размер ЗП
3	1	2	2	30	15	825	412.5	1666.66666666667

Предлагаю вам, для большего понимания работы агрегатных функций, самостоятельно проанализировать каждое полученное значение. Расчеты здесь ведем, соответственно, по детальным данным полученным запросом:

```
SELECT
  DepartmentID,
  PositionID,
  BonusPercent,
  Salary/100*BonusPercent [Salary/100*BonusPercent],
  Salary
FROM Employees
WHERE DepartmentID=3 -- учесть только ИТ-отдел
```

DepartmentID	PositionID	BonusPercent	Salary/100*BonusPercent	Salary
3	3	15	225	1500
3	4	30	600	2000
3	3	NULL	NULL	1500

Идем, дальше. В случае, если агрегатная функция возвращает NULL (например, у всех сотрудников не указано значение Salary), 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
  SUM(Salary),
  AVG(Salary),

  -- обрабатываем итог при помощи ISNULL
  ISNULL(SUM(Salary),0),
  ISNULL(AVG(Salary),0)
FROM Employees
WHERE DepartmentID=10 -- здесь специально указан несуществующий отдел, чтобы запрос не вернул за
писей
```

(No column name)	(No column name)	(No column name)	(No column name)
NULL	NULL	0	0

Я считаю, что очень важно понимать назначение каждой агрегатной функции и то каким образом они производят расчет, т.к. в SQL это главный инструмент, который служит для расчета итоговых значений.

В данном случае мы рассмотрели, как каждая агрегатная функция ведет себя самостоятельно, т.е. она применялась к значениям всего набора записей полученным командой SELECT. Дальше мы рассмотрим, как эти же функции применяются для вычисления итогов по группам, при помощи конструкции GROUP BY.

GROUP BY – группировка данных

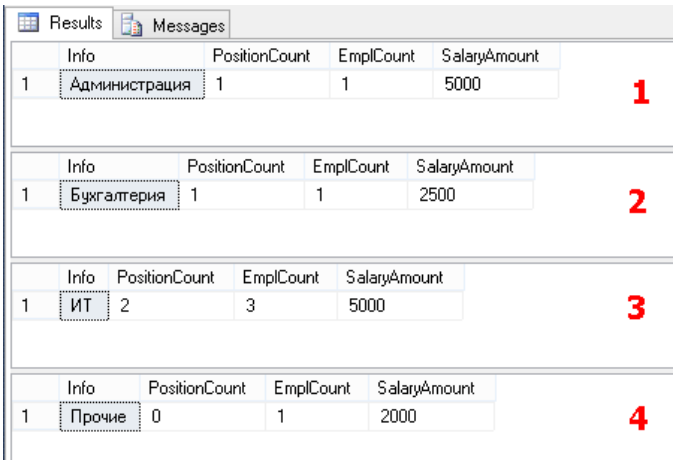
До этого мы уже вычисляли итоги для конкретного отдела, примерно следующим образом:

```
SELECT
  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount
FROM Employees
WHERE DepartmentID=3 -- данные только по ИТ отделу
```

А теперь представьте, что нас попросили получить такие же цифры в разрезе каждого отдела. Конечно мы можем засучить рукава и выполнить этот же запрос для каждого отдела. Итак, сказано-сделано, пишем 4 запроса:

```
SELECT
  'Администрация' Info,
  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount
FROM Employees
WHERE DepartmentID=1 -- данные по Администрации

SELECT
  'Бухгалтерия' Info,
  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount
FROM Employees
WHERE DepartmentID=2 -- данные по Бухгалтерии

SELECT
  'ИТ' Info,
  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount
FROM Employees
WHERE DepartmentID=3 -- данные по ИТ отделу

SELECT
  'Прочие' Info,
  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount
FROM Employees
WHERE DepartmentID IS NULL -- и еще не забываем данные по внештатникам
```

В результате мы получим 4 набора данных:



Обратите внимание, что мы можем использовать поля, заданные в виде констант – 'Администрация', 'Бухгалтерия', …

В общем все цифры, о которых нас просили, мы добыли, объединяем все в Excel и отдаем директору.

Отчет директору понравился, и он говорит: «а добавьте еще колонку с информацией по среднему окладу». И как всегда это нужно сделать очень срочно.

Мда, что делать?! Вдобавок представим еще что отделов у нас не 3, а 15.

Вот как раз то примерно для таких случаев служит конструкция GROUP BY:

```
SELECT
  DepartmentID,
  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount,
  AVG(Salary) SalaryAvg -- плюс выполняем пожелание директора
FROM Employees
GROUP BY DepartmentID
```

DepartmentID	PositionCount	EmplCount	SalaryAmount	SalaryAvg
NULL	0	1	2000	2000
1	1	1	5000	5000
2	1	1	2500	2500
3	2	3	5000	1666.66666666667

Мы получили все те же самые данные, но теперь используя только один запрос!

Пока не обращайте внимание, на то что департаменты у нас вывелись в виде цифр, дальше мы научимся выводить все красиво.

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».

Для примера, сделаем группировку данных в разрезе Отделов и Должностей:

```
SELECT
  DepartmentID,PositionID,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount
FROM Employees
GROUP BY DepartmentID,PositionID
```

DepartmentID	PositionID	EmplCount	SalaryAmount
NULL	NULL	1	2000
2	1	1	2500
1	2	1	5000
3	3	2	3000
3	4	1	2000

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

Для полей, перечисленных после GROUP BY из таблицы Employees определяются все уникальные комбинации по значениям DepartmentID и PositionID, т.е. происходит примерно следующее:

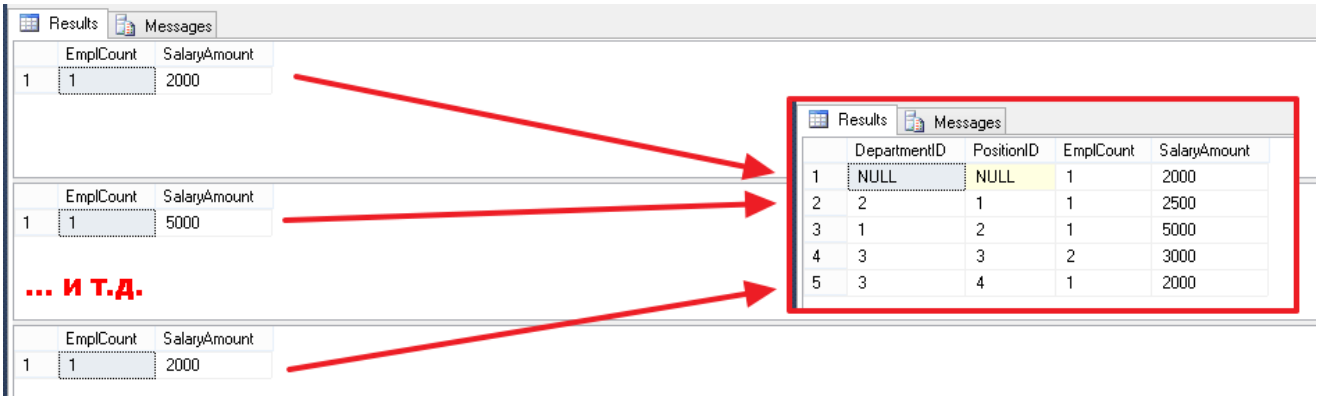
```
SELECT DISTINCT DepartmentID,PositionID
FROM Employees
```

DepartmentID	PositionID
NULL	NULL
1	2
2	1
3	3
3	4

После чего делается пробежка по каждой комбинации и делаются вычисления агрегатных функций:

```
SELECT
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount
FROM Employees
WHERE  DepartmentID IS NULL AND PositionID IS NULL

SELECT
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount
FROM Employees
WHERE  DepartmentID=1 AND PositionID=2

-- ...

SELECT
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount
FROM Employees
WHERE  DepartmentID=3 AND PositionID=4
```

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



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

И демонстрация всего сказанного:

```
SELECT
  'Строка константа' Const1, -- константа в виде строки
  1 Const2, -- константа в виде числа

  -- выражение с использованием полей участвующих в группировке
  CONCAT('Отдел № ',DepartmentID) ConstAndGroupField,
  CONCAT('Отдел № ',DepartmentID,', Должность № ',PositionID) ConstAndGroupFields,

  DepartmentID, -- поле из списка полей участвующих в группировке
  -- PositionID, -- поле учавствующее в группировке, не обязательно дублировать здесь

  COUNT(*) EmplCount, -- кол-во строк в каждой группе

  -- остальные поля можно использовать только с агрегатными функциями: COUNT, SUM, MIN, MAX, …
  SUM(Salary) SalaryAmount,
  MIN(ID) MinID
FROM Employees
GROUP BY DepartmentID,PositionID -- группировка по полям DepartmentID,PositionID
```

Const1	Const2	ConstAndGroupField	ConstAndGroupFields	DepartmentID	EmplCount	SalaryAmount	MinID
Строка константа	1	Отдел №	Отдел №, Должность №	NULL	1	2000	1005
Строка константа	1	Отдел № 2	Отдел № 2, Должность № 1	2	1	2500	1002
Строка константа	1	Отдел № 1	Отдел № 1, Должность № 2	1	1	5000	1000
Строка константа	1	Отдел № 3	Отдел № 3, Должность № 3	3	2	3000	1001
Строка константа	1	Отдел № 3	Отдел № 3, Должность № 4	3	1	2000	1003

Так же стоит отметить, что группировку можно делать не только по полям, но также и по выражениям. Для примера сгруппируем данные по сотрудникам, по годам рождения:

```
SELECT
  CONCAT('Год рождения - ',YEAR(Birthday)) YearOfBirthday,
  COUNT(*) EmplCount
FROM Employees
GROUP BY YEAR(Birthday)
```

Рассмотрим пример с более сложным выражением. Для примера, получим градацию сотрудников по годам рождения:

```
SELECT
  CASE
    WHEN YEAR(Birthday)>=2000 THEN 'от 2000'
    WHEN YEAR(Birthday)>=1990 THEN '1999-1990'
    WHEN YEAR(Birthday)>=1980 THEN '1989-1980'
    WHEN YEAR(Birthday)>=1970 THEN '1979-1970'
    WHEN Birthday IS NOT NULL THEN 'ранее 1970'
    ELSE 'не указано'
  END RangeName,
  COUNT(*) EmplCount
FROM Employees
GROUP BY
  CASE
    WHEN YEAR(Birthday)>=2000 THEN 'от 2000'
    WHEN YEAR(Birthday)>=1990 THEN '1999-1990'
    WHEN YEAR(Birthday)>=1980 THEN '1989-1980'
    WHEN YEAR(Birthday)>=1970 THEN '1979-1970'
    WHEN Birthday IS NOT NULL THEN 'ранее 1970'
    ELSE 'не указано'
  END
```

RangeName	EmplCount
1979-1970	1
1989-1980	2
не указано	2
ранее 1970	1

Т.е. в данном случае группировка делается по предварительно вычисленному для каждого сотрудника CASE-выражению:

```
SELECT
  ID,
  CASE
    WHEN YEAR(Birthday)>=2000 THEN 'от 2000'
    WHEN YEAR(Birthday)>=1990 THEN '1999-1990'
    WHEN YEAR(Birthday)>=1980 THEN '1989-1980'
    WHEN YEAR(Birthday)>=1970 THEN '1979-1970'
    WHEN Birthday IS NOT NULL THEN 'ранее 1970'
    ELSE 'не указано'
  END
FROM Employees
```

ID	(No column name)
1000	ранее 1970
1001	1989-1980
1002	1979-1970
1003	1989-1980
1004	не указано
1005	не указано

RangeName	EmplCount
1979-1970	1
1989-1980	2
не указано	2
ранее 1970	1

Ну и конечно же вы можете объединять в блоке GROUP BY выражения с полями:

```
SELECT
  DepartmentID,
  CONCAT('Год рождения - ',YEAR(Birthday)) YearOfBirthday,
  COUNT(*) EmplCount
FROM Employees
GROUP BY YEAR(Birthday),DepartmentID -- порядок может не совпадать с порядком их использования в
блоке SELECT
ORDER BY DepartmentID,YearOfBirthday -- напоследок мы можем применить к результату сортировку
```

Вернемся к нашей изначальной задаче. Как мы уже знаем, отчет очень понравился директору, и он попросил нас делать его еженедельно, дабы он мог мониторить изменения по компании. Чтобы, не перебивать каждый раз в Excel цифровое значение отдела на его наименование, воспользуемся знаниями, которые у нас уже есть, и усовершенствуем наш запрос:

```
SELECT
  CASE DepartmentID
    WHEN 1 THEN 'Администрация'
    WHEN 2 THEN 'Бухгалтерия'
    WHEN 3 THEN 'ИТ'
    ELSE 'Прочие'
  END Info,
  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount,
  AVG(Salary) SalaryAvg -- плюс выполняем пожелание директора
FROM Employees
GROUP BY DepartmentID
ORDER BY Info -- добавим для большего удобства сортировку по колонке Info
```

Info	PositionCount	EmplCount	SalaryAmount	SalaryAvg
Администрация	1	1	5000	5000
Бухгалтерия	1	1	2500	2500
ИТ	2	3	5000	1666.66666666667
Прочие	0	1	2000	2000

Хоть со стороны может выглядит и страшно, но все равно это получше чем было изначально. Недостаток в том, что если заведут новый отдел и его сотрудников, то выражение CASE нам нужно будет дописывать, дабы сотрудники нового отдела не попали в группу «Прочие».
Но ничего, со временем, мы научимся делать все красиво, чтобы выборка у нас не зависела от появления в БД новых данных, а была динамической. Немного забегу вперед, чтобы показать к написанию каких запросов мы стремимся прийти:

```
SELECT
  ISNULL(dep.Name,'Прочие') DepName,
  COUNT(DISTINCT emp.PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(emp.Salary) SalaryAmount,
  AVG(emp.Salary) SalaryAvg -- плюс выполняем пожелание директора
FROM Employees emp
LEFT JOIN Departments dep ON emp.DepartmentID=dep.ID
GROUP BY emp.DepartmentID,dep.Name
ORDER BY DepName
```

В общем, не переживайте – все начинали с простого. Пока вам просто нужно понять суть конструкции GROUP BY.

Напоследок, давайте посмотрим каким образом можно строить сводные отчеты при помощи GROUP BY.

Для примера выведем сводную таблицу, в разрезе отделов, так чтобы была подсчитана суммарная заработная плата, получаемая сотрудниками в разбивке по должностям:

```
SELECT
  DepartmentID,
  SUM(CASE WHEN PositionID=1 THEN Salary END) [Бухгалтера],
  SUM(CASE WHEN PositionID=2 THEN Salary END) [Директора],
  SUM(CASE WHEN PositionID=3 THEN Salary END) [Программисты],
  SUM(CASE WHEN PositionID=4 THEN Salary END) [Старшие программисты],
  SUM(Salary) [Итого по отделу]
FROM Employees
GROUP BY DepartmentID
```

DepartmentID	Бухгалтера	Директора	Программисты	Старшие программисты	Итого по отделу
NULL	NULL	NULL	NULL	NULL	2000
1	NULL	5000	NULL	NULL	5000
2	2500	NULL	NULL	NULL	2500
3	NULL	NULL	3000	2000	5000

Т.е. мы свободно можем использовать любые выражения внутри агрегатных функций.

Можно конечно переписать и при помощи IIF:

```
SELECT
  DepartmentID,
  SUM(IIF(PositionID=1,Salary,NULL)) [Бухгалтера],
  SUM(IIF(PositionID=2,Salary,NULL)) [Директора],
  SUM(IIF(PositionID=3,Salary,NULL)) [Программисты],
  SUM(IIF(PositionID=4,Salary,NULL)) [Старшие программисты],
  SUM(Salary) [Итого по отделу]
FROM Employees
GROUP BY DepartmentID
```

Но в случае с IIF нам придется явно указывать NULL, которое возвращается в случае невыполнения условия.

В аналогичных случаях мне больше нравится использовать CASE без блока ELSE, чем лишний раз писать NULL. Но это конечно дело вкуса, о котором не спорят.

И давайте вспомним, что в агрегатных функциях при агрегации не учитываются NULL значения.
Для закрепления, сделайте самостоятельный анализ полученных данных по развернутому запросу:

```
SELECT
  DepartmentID,
  CASE WHEN PositionID=1 THEN Salary END [Бухгалтера],
  CASE WHEN PositionID=2 THEN Salary END [Директора],
  CASE WHEN PositionID=3 THEN Salary END [Программисты],
  CASE WHEN PositionID=4 THEN Salary END [Старшие программисты],
  Salary [Итого по отделу]
FROM Employees
```

DepartmentID	Бухгалтера	Директора	Программисты	Старшие программисты	Итого по отделу
1	NULL	5000	NULL	NULL	5000
3	NULL	NULL	1500	NULL	1500
2	2500	NULL	NULL	NULL	2500
3	NULL	NULL	NULL	2000	2000
3	NULL	NULL	1500	NULL	1500
NULL	NULL	NULL	NULL	NULL	2000

И еще давайте вспомним, что если вместо NULL мы хотим увидеть нули, то мы можем обработать значение, возвращаемое агрегатной функцией. Например:

```
SELECT
  DepartmentID,
  ISNULL(SUM(IIF(PositionID=1,Salary,NULL)),0) [Бухгалтера],
  ISNULL(SUM(IIF(PositionID=2,Salary,NULL)),0) [Директора],
  ISNULL(SUM(IIF(PositionID=3,Salary,NULL)),0) [Программисты],
  ISNULL(SUM(IIF(PositionID=4,Salary,NULL)),0) [Старшие программисты],
  ISNULL(SUM(Salary),0) [Итого по отделу]
FROM Employees
GROUP BY DepartmentID
```

DepartmentID	Бухгалтера	Директора	Программисты	Старшие программисты	Итого по отделу
NULL	0	0	0	0	2000
1	0	5000	0	0	5000
2	2500	0	0	0	2500
3	0	0	3000	2000	5000

Теперь в целях практики, вы можете:

- вывести названия департаментов вместо их идентификаторов, например, добавив выражение CASE обрабатывающее DepartmentID в блоке SELECT
- добавьте сортировку по имени отдела при помощи ORDER BY

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, т.к. обычно от нас требуют предоставления сводных отчетов, а не детальных данных (простыней). И конечно же все это крутится вокруг знания базовой конструкции, т.к. прежде чем что-то подытожить (агрегировать), вам нужно первым делом это правильно выбрать, используя «SELECT … WHERE …».

Важное место здесь имеет практика, поэтому, если вы поставили целью понять язык SQL, не изучить, а именно понять – практикуйтесь, практикуйтесь и практикуйтесь, перебирая самые разные варианты, которые только сможете придумать.

На начальных порах, если вы не уверены в правильности полученных агрегированных данных, делайте детальную выборку, включающую все значения, по которым идет агрегация. И проверяйте правильность расчетов вручную по этим детальным данным. В этом случае очень сильно может помочь использование программы Excel.

Допустим, что вы дошли до этого момента

Допустим, что вы бухгалтер Сидоров С.С., который решил научиться писать SELECT-запросы. Допустим, что вы уже успели дочитать данный учебник до этого момента, и уже уверено пользуетесь всеми вышеперечисленными базовыми конструкциями, т.е. вы умеете:

- Выбирать детальные данные по условию WHERE из одной таблицы
- Умеете пользоваться агрегатными функциями и группировкой из одной таблицы

Так как на работе посчитали, что вы уже все умеете, то вам предоставили доступ к БД (и такое порой бывает), и теперь вы разработали и вытаскиваете тот самый еженедельный отчет для директора.

Да, но они не учли, что вы пока не умеете строить запросы из нескольких таблиц, а только из одной, т.е. вы не умеете делать что-то вроде такого:

```
SELECT
  emp.*, -- вернуть все поля таблицы Employees
  dep.Name DepartmentName, -- к этим полям добавить поле Name из таблицы Departments
  pos.Name PositionName -- и еще добавить поле Name из таблицы Positions
FROM Employees emp
LEFT JOIN Departments dep ON emp.DepartmentID=dep.ID
LEFT JOIN Positions pos ON emp.PositionID=pos.ID
```

ID	Name	Birthday	…	Salary	BonusPercent	DepartmentName	PositionName
1000	Иванов И.И.	19.02.1955		5000	50	Администрация	Директор
1001	Петров П.П.	03.12.1983		1500	15	ИТ	Программист
1002	Сидоров С.С.	07.06.1976		2500	NULL	Бухгалтерия	Бухгалтер
1003	Андреев А.А.	17.04.1982		2000	30	ИТ	Старший программист
1004	Николаев Н.Н.	NULL		1500	NULL	ИТ	Программист
1005	Александров А.А.	NULL		2000	NULL	NULL	NULL

Несмотря на то, что вы этого не умеете, поверьте, вы молодец, и уже, и так много достигли.

И так, как же можно воспользоваться вашими текущими знаниями и получить при этом еще более продуктивные результаты?! Воспользуемся силой коллективного разума – идем к программистам, которые работают у вас, т.е. к Андрееву А.А., Петрову П.П. или Николаеву Н.Н., и попросим кого-нибудь из них написать для вас представление (VIEW или просто «Вьюха», так они даже, думаю, быстрее поймут вас), которое помимо основных полей из таблицы Employees, будет еще возвращать поля с «Названием отдела» и «Названием должности», которых вам так недостает сейчас для еженедельного отчета, которым вас загрузил Иванов И.И.

Т.к. вы все грамотно объяснили, то ИТ-шники, сразу же поняли, что от них хотят и создали, специально для вас, представление с названием ViewEmployeesInfo.

Представляем, что вы следующей команды не видите, т.к. это делают ИТ-шники:

```
CREATE VIEW ViewEmployeesInfo
AS
SELECT
  emp.*, -- вернуть все поля таблицы Employees
  dep.Name DepartmentName, -- к этим полям добавить поле Name из таблицы Departments
  pos.Name PositionName -- и еще добавить поле Name из таблицы Positions
FROM Employees emp
LEFT JOIN Departments dep ON emp.DepartmentID=dep.ID
LEFT JOIN Positions pos ON emp.PositionID=pos.ID
```

Т.е. для вас весь этот, пока страшный и непонятный, текст остается за кадром, а ИТ-шники дают вам только название представления «ViewEmployeesInfo», которое возвращает все вышеуказанные данные (т.е. то что вы у них просили).

Вы теперь можете работать с данным представлением, как с обычной таблицей:

```
SELECT *
FROM ViewEmployeesInfo
```

ID	Name	Birthday	…	Salary	BonusPercent	DepartmentName	PositionName
1000	Иванов И.И.	19.02.1955		5000	50	Администрация	Директор
1001	Петров П.П.	03.12.1983		1500	15	ИТ	Программист
1002	Сидоров С.С.	07.06.1976		2500	NULL	Бухгалтерия	Бухгалтер
1003	Андреев А.А.	17.04.1982		2000	30	ИТ	Старший программист
1004	Николаев Н.Н.	NULL		1500	NULL	ИТ	Программист
1005	Александров А.А.	NULL		2000	NULL	NULL	NULL

Т.к. теперь все необходимые для отчета данные есть в одной «таблице» (а-ля вьюха), то вы с легкостью сможете переделать свой еженедельный отчет:

```
SELECT
  DepartmentName,
  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount,
  AVG(Salary) SalaryAvg
FROM ViewEmployeesInfo emp
GROUP BY DepartmentID,DepartmentName
ORDER BY DepartmentName
```

DepartmentName	PositionCount	EmplCount	SalaryAmount	SalaryAvg
NULL	0	1	2000	2000
Администрация	1	1	5000	5000
Бухгалтерия	1	1	2500	2500
ИТ	2	3	5000	1666.66666666667

Теперь все названия отделов на местах, плюс к тому же запрос стал динамическим, и будет изменяться при добавлении новых отделов и их сотрудников, т.е. вам теперь ничего переделывать не нужно, а достаточно раз в неделю выполнить запрос и отдать его результат директору.

Т.е. для вас в данном случае, будто бы ничего и не поменялось, вы продолжаете так же работать с одной таблицей (только уже правильнее сказать с представлением ViewEmployeesInfo), которое возвращает все необходимые вам данные. Благодаря помощи ИТ-шников, детали по добыванию DepartmentName и PositionName остались для вас в черном ящике. Т.е. представление для вас выглядит так же, как и обычная таблица, считайте, что это расширенная версия таблицы Employees.

Давайте для примера еще сформируем ведомость, чтобы вы убедились, что все действительно так как я и говорил (что вся выборка идет из одного представления):

```
SELECT
  ID,
  Name,
  Salary
FROM ViewEmployeesInfo
WHERE Salary IS NOT NULL
  AND Salary>0
ORDER BY Name
```

ID	Name	Salary
1005	Александров А.А.	2000
1003	Андреев А.А.	2000
1000	Иванов И.И.	5000
1004	Николаев Н.Н.	1500
1001	Петров П.П.	1500
1002	Сидоров С.С.	2500

Надеюсь, что данный запрос вам понятен.

Использование представлений в некоторых случаях, дает возможность значительно расширить границы пользователей, владеющих написанием базовых SELECT-запросов. В данном случае представление, представляет собой плоскую таблицу со всеми необходимыми пользователю данными (для тех, кто разбирается в OLAP, это можно сравнить с приближенным подобием OLAP-куба с фактами и измерениями).

Вырезка с википедии. *Хотя SQL и задумывался как средство работы конечного пользователя, в конце концов он стал настолько сложным, что превратился в инструмент программиста.*

Как видите, уважаемые пользователи, язык SQL изначально задумывался, как инструмент для вас. Так что, все в ваших руках и желании, не отпускайте руки.

HAVING – наложение условия выборки к сгруппированным данным

Собственно, если вы поняли, что такое группировка, 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.

Рассмотрим пример:

```
SELECT
  DepartmentID,
  SUM(Salary) SalaryAmount
FROM Employees
GROUP BY DepartmentID
HAVING SUM(Salary)>3000
```

DepartmentID	SalaryAmount
1	5000
3	5000

Т.е. данный запрос вернул нам сгруппированные данные только по тем отделам, у которых сумма ЗП всех сотрудников превышает 3000, т.е. «SUM(Salary)>3000».

```
SELECT
  DepartmentID,
  SUM(Salary) SalaryAmount
FROM Employees
GROUP BY DepartmentID
```

DepartmentID	SalaryAmount
NULL	2000
1	5000
2	2500
3	5000

```
SELECT
  DepartmentID,
  SUM(Salary) SalaryAmount
FROM Employees
GROUP BY DepartmentID
HAVING SUM(Salary)>3000
```

DepartmentID	SalaryAmount
NULL	2000
1	5000 >3000
2	2500
3	5000 >3000

Результат

DepartmentID	SalaryAmount
1	5000
3	5000

Т.е. здесь в первую очередь происходит группировка и вычисляются данные по всем отделам:

```
SELECT
  DepartmentID,
  SUM(Salary) SalaryAmount
FROM Employees
GROUP BY DepartmentID -- 1. получаем сгруппированные данные по всем отделам
```

А уже к этим данным применяется условие указанно в блоке HAVING:

```
SELECT
  DepartmentID,
  SUM(Salary) SalaryAmount
FROM Employees
GROUP BY DepartmentID -- 1. получаем сгруппированные данные по всем отделам
HAVING SUM(Salary)>3000 -- 2. условие для фильтрации сгруппированных данных
```

В HAVING-условии так же можно строить сложные условия используя операторы AND, OR и NOT:

```
SELECT
  DepartmentID,
  SUM(Salary) SalaryAmount
FROM Employees
GROUP BY DepartmentID
HAVING SUM(Salary)>3000 AND COUNT(*)<2 -- и число людей меньше 2-х
```

```
SELECT
  DepartmentID,
  SUM(Salary) SalaryAmount,
  COUNT(*)
FROM Employees
GROUP BY DepartmentID
```

```
SELECT
  DepartmentID,
  SUM(Salary) SalaryAmount
FROM Employees
GROUP BY DepartmentID
HAVING SUM(Salary)>3000 AND COUNT(*)<2
```

DepartmentID	SalaryAmount	(No column name)
NULL	2000	1
1	5000	1
2	2500	1
3	5000	3

DepartmentID	SalaryAmount	(No column name)
NULL	2000	1
1	5000	1
2	2500	1
3	5000	3

OK

Результат

DepartmentID	SalaryAmount
1	5000

Как можно здесь заметить агрегатная функция (см. «COUNT(*)») может быть указана только в блоке HAVING.

Соответственно мы можем отобразить только номер отдела, подпадающего под HAVING-условие:

```
SELECT
  DepartmentID
FROM Employees
GROUP BY DepartmentID
HAVING SUM(Salary)>3000 AND COUNT(*)<2 -- и число людей меньше 2-х
```

Пример использования HAVING-условия по полю включенного в GROUP BY:

```
SELECT
  DepartmentID,
  SUM(Salary) SalaryAmount
FROM Employees
GROUP BY DepartmentID -- 1. сделать группировку
HAVING DepartmentID=3 -- 2. наложить фильтр на результат группировки
```

Это только пример, т.к. в данном случае проверку логичнее было бы сделать через WHERE-условие:

```
SELECT
  DepartmentID,
  SUM(Salary) SalaryAmount
FROM Employees
WHERE DepartmentID=3 -- 1. провести фильтрацию детальных данных
GROUP BY DepartmentID -- 2. сделать группировку только по отобранным записям
```

Т.е. сначала отфильтровать сотрудников по отделу 3, и только потом сделать расчет.

Примечание. *На самом деле, несмотря на то, что эти два запроса выглядят по-разному оптимизатор СУБД может выполнить их одинаково.*

Думаю, на этом рассказ о HAVING-условиях можно окончить.

Подведем итоги

Сведем данные полученные во второй и третьей части и рассмотрим конкретное месторасположение каждой изученной нами конструкции и укажем порядок их выполнения:

Конструкция/Блок	Порядок выполнения	Выполняемая функция
SELECT возвращаемые выражения	4	Возврат данных полученных запросом
FROM источник	0	В нашем случае это пока все строки таблицы
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

Конечно же, вы так же можете применить к сгруппированным данным предложения DISTINCT и TOP, изученные во второй части.

Эти предложения в данном случае применятся к окончательному результату:

```
SELECT
  TOP 1 -- 6. применится в последнюю очередь
    SUM(Salary) SalaryAmount
FROM Employees
GROUP BY DepartmentID
HAVING SUM(Salary)>3000
ORDER BY DepartmentID -- 5. сортировка результата
```

SalaryAmount
5000

```
SELECT
  DISTINCT -- показать только уникальные значения SalaryAmount
    SUM(Salary) SalaryAmount
FROM Employees
GROUP BY DepartmentID
```

SalaryAmount
2000
2500
5000

Как получились данные результаты проанализируйте самостоятельно.

Заключение

Основная цель которую я ставил в данной части – раскрыть для вас суть агрегатных функций и группировок.

Если базовая конструкция позволяла нам получить необходимые детальные данные, то применение агрегатных функций и группировок к этим детальным данным, дало нам возможность получить по ним сводные данные. Так что, как видите здесь все важно, т.к. одно опирается на другое – без знания базовой конструкции мы не сможем, например, правильно отобрать данные, по которым нам нужно просчитать итоги.

Здесь я намеренно стараюсь показывать только основы, чтобы сосредоточить внимание начинающих на самых главных конструкциях и не перегружать их лишней информацией. Твердое понимание основных конструкций (о которых я еще продолжу рассказ в последующих частях) даст вам возможность решить практически любую задачу по выборке данных из РБД. Основные конструкции оператора SELECT применимы в таком же виде практически во всех СУБД (отличия в основном состоят в деталях, например, в реализации функций – для работы со строками, временем, и т.д.).

В последующем, твердое знание базы даст вам возможность самостоятельно легко изучить разные расширения языка SQL, такие как:

- GROUP BY ROLLUP(…), GROUP BY GROUPING SETS(…), …
- PIVOT, UNPIVOT
- и т.п.

В рамках данного учебника я решил не рассказывать об этих расширениях, т.к. и без их знания, владея только базовыми конструкциями языка SQL, вы сможете решать очень большой спектр задач. Расширения языка SQL по сути служат для решения какого-то определенного круга задач, т.е. позволяют решить задачу определенного класса более изящно (но не всегда эффективней в плане скорости или затраченных ресурсов).

Если вы делаете первые шаги в SQL, то сосредоточьтесь в первую очередь, именно на изучении базовых конструкций, т.к. владея базой, все остальное вам понять будет гораздо легче, и к тому же самостоятельно. Вам в первую очередь, как бы нужно объемно понять возможности языка SQL, т.е. какого рода операции он вообще позволяет совершить над данными. Донести до начинающих информацию в объемном виде – это еще одна из причин, почему я буду показывать только самые главные (железные) конструкции.

Удачи вам в изучении и понимании языка SQL.

Часть четвертая

В данной части мы рассмотрим

Многотабличные запросы:

- Операции горизонтального соединения таблиц – JOIN
- Связь таблиц при помощи WHERE-условия
- Операции вертикального объединения результатов запросов – UNION

Работу с подзапросами:

- Подзапросы в блоках FROM, SELECT
- Подзапрос в конструкции APPLY
- Использование предложения WITH
- Подзапросы в блоке WHERE:
  - Групповое сравнение — ALL, ANY
  - Условие EXISTS
  - Условие IN

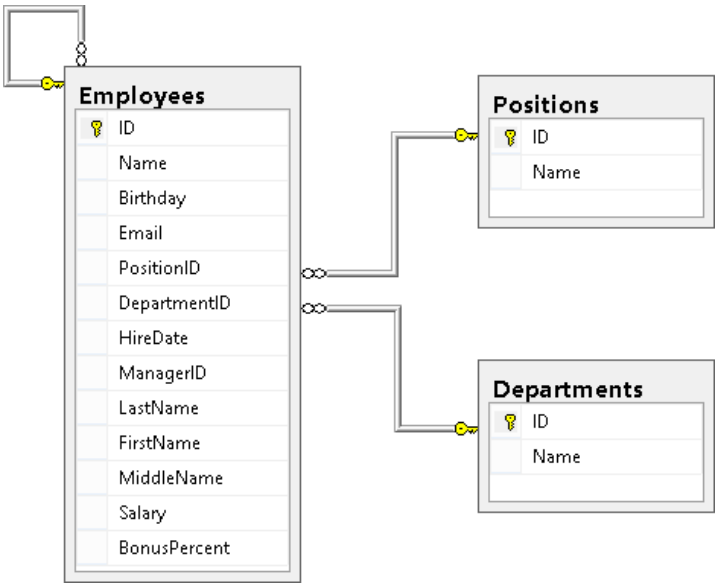
Добавим немного новых данных

Для демонстрационных целей добавим несколько отделов и должностей:

```
SET IDENTITY_INSERT Departments ON
INSERT Departments(ID,Name) VALUES(4,N'Маркетинг и реклама')
INSERT Departments(ID,Name) VALUES(5,N'Логистика')
SET IDENTITY_INSERT Departments OFF


SET IDENTITY_INSERT Positions ON
INSERT Positions(ID,Name) VALUES(5,N'Маркетолог')
INSERT Positions(ID,Name) VALUES(6,N'Логист')
INSERT Positions(ID,Name) VALUES(7,N'Кладовщик')
SET IDENTITY_INSERT Positions OFF
```

JOIN-соединения – операции горизонтального соединения данных

Здесь нам очень пригодится знание структуры БД, т.е. какие в ней есть таблицы, какие данные хранятся в этих таблицах и по каким полям таблицы связаны между собой. Первым делом всегда досконально изучайте структуру БД, т.к. нормальный запрос можно написать только тогда, когда ты знаешь, что откуда берется. У нас структура состоит из 3-х таблиц Employees, Departments и Positions. Приведу здесь диаграмму из первой части:



Если суть РДБ – разделяй и властвуй, то суть операций объединений снова склеить разбитые по таблицам данные, т.е. привести их обратно в человеческий вид.

Если говорить просто, то операции горизонтального соединения таблицы с другими таблицами используются для того, чтобы получить из них недостающие данные. Вспомните пример с еженедельным отчетом для директора, когда при запросе из таблицы Employees, нам для получения окончательного результата недоставало поля «Название отдела», которое находится в таблице Departments.

Начнем с теории. 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 (Это не всегда всем сразу понятно. Так что, если не понятно, то просто вернитесь сюда после рассмотрения примеров.)**
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.

Как видно из таблицы полный синтаксис от краткого отличается только наличием слов INNER или OUTER.

Лично я всегда при написании запросов использую только краткий синтаксис, по той причине:

1. Это короче и не засоряет запрос лишними словами;
2. По словам LEFT, RIGHT, FULL и CROSS и так понятно о каком соединении идет речь, так же и в случае просто JOIN;
3. Считаю слова INNER и OUTER в данном случае ненужными рудиментами, которые больше путают начинающих.

Но конечно, это мое личное предпочтение, возможно кому-то нравится писать длинно, и он видит в этом свои прелести.

Понимание каждого вида соединения очень важно, т.к. от применения того или иного вида, результат запроса может отличаться. Сравните результаты одного и того же запроса с применением разного типа соединения, попробуйте пока просто увидеть разницу и идите дальше (мы сюда еще вернемся):

```
-- JOIN вернет 5 строк
SELECT emp.ID,emp.Name,emp.DepartmentID,dep.ID,dep.Name
FROM Employees emp
JOIN Departments dep ON emp.DepartmentID=dep.ID
```

ID	Name	DepartmentID	ID	Name
1000	Иванов И.И.	1	1	Администрация
1001	Петров П.П.	3	3	ИТ
1002	Сидоров С.С.	2	2	Бухгалтерия
1003	Андреев А.А.	3	3	ИТ
1004	Николаев Н.Н.	3	3	ИТ

```
-- LEFT JOIN вернет 6 строк
SELECT emp.ID,emp.Name,emp.DepartmentID,dep.ID,dep.Name
FROM Employees emp
LEFT JOIN Departments dep ON emp.DepartmentID=dep.ID
```

ID	Name	DepartmentID	ID	Name
1000	Иванов И.И.	1	1	Администрация
1001	Петров П.П.	3	3	ИТ
1002	Сидоров С.С.	2	2	Бухгалтерия
1003	Андреев А.А.	3	3	ИТ
1004	Николаев Н.Н.	3	3	ИТ
1005	Александров А.А.	NULL	NULL	NULL

```
-- RIGHT JOIN вернет 7 строк
SELECT emp.ID,emp.Name,emp.DepartmentID,dep.ID,dep.Name
FROM Employees emp
RIGHT JOIN Departments dep ON emp.DepartmentID=dep.ID
```

ID	Name	DepartmentID	ID	Name
1000	Иванов И.И.	1	1	Администрация
1002	Сидоров С.С.	2	2	Бухгалтерия
1001	Петров П.П.	3	3	ИТ
1003	Андреев А.А.	3	3	ИТ
1004	Николаев Н.Н.	3	3	ИТ
NULL	NULL	NULL	4	Маркетинг и реклама
NULL	NULL	NULL	5	Логистика

```
-- FULL JOIN вернет 8 строк
SELECT emp.ID,emp.Name,emp.DepartmentID,dep.ID,dep.Name
FROM Employees emp
FULL JOIN Departments dep ON emp.DepartmentID=dep.ID
```

ID	Name	DepartmentID	ID	Name
1000	Иванов И.И.	1	1	Администрация
1001	Петров П.П.	3	3	ИТ
1002	Сидоров С.С.	2	2	Бухгалтерия
1003	Андреев А.А.	3	3	ИТ
1004	Николаев Н.Н.	3	3	ИТ
1005	Александров А.А.	NULL	NULL	NULL
NULL	NULL	NULL	4	Маркетинг и реклама
NULL	NULL	NULL	5	Логистика

```
-- CROSS JOIN вернет 30 строк - (6 строк таблицы Employees) * (5 строк таблицы Departments)
SELECT emp.ID,emp.Name,emp.DepartmentID,dep.ID,dep.Name
FROM Employees emp
CROSS JOIN Departments dep
```

ID	Name	DepartmentID	ID	Name
1000	Иванов И.И.	1	1	Администрация
1001	Петров П.П.	3	1	Администрация
1002	Сидоров С.С.	2	1	Администрация
1003	Андреев А.А.	3	1	Администрация
1004	Николаев Н.Н.	3	1	Администрация
1005	Александров А.А.	NULL	1	Администрация
1000	Иванов И.И.	1	2	Бухгалтерия
1001	Петров П.П.	3	2	Бухгалтерия
1002	Сидоров С.С.	2	2	Бухгалтерия
1003	Андреев А.А.	3	2	Бухгалтерия
1004	Николаев Н.Н.	3	2	Бухгалтерия
1005	Александров А.А.	NULL	2	Бухгалтерия
1000	Иванов И.И.	1	3	ИТ
1001	Петров П.П.	3	3	ИТ
1002	Сидоров С.С.	2	3	ИТ
1003	Андреев А.А.	3	3	ИТ
1004	Николаев Н.Н.	3	3	ИТ
1005	Александров А.А.	NULL	3	ИТ
1000	Иванов И.И.	1	4	Маркетинг и реклама
1001	Петров П.П.	3	4	Маркетинг и реклама
1002	Сидоров С.С.	2	4	Маркетинг и реклама
1003	Андреев А.А.	3	4	Маркетинг и реклама
1004	Николаев Н.Н.	3	4	Маркетинг и реклама
1005	Александров А.А.	NULL	4	Маркетинг и реклама
1000	Иванов И.И.	1	5	Логистика
1001	Петров П.П.	3	5	Логистика
1002	Сидоров С.С.	2	5	Логистика
1003	Андреев А.А.	3	5	Логистика
1004	Николаев Н.Н.	3	5	Логистика
1005	Александров А.А.	NULL	5	Логистика

Настало время вспомнить про псевдонимы таблиц

Пришло время вспомнить про псевдонимы таблиц, о которых я рассказывал в начале второй части.

В многотабличных запросах, псевдоним помогает нам явно указать из какой именно таблицы берется поле. Посмотрим на пример:

```
SELECT emp.ID,emp.Name,emp.DepartmentID,dep.ID,dep.Name
FROM Employees emp
JOIN Departments dep ON emp.DepartmentID=dep.ID
```

В нем поля с именами ID и Name есть в обоих таблицах и в Employees, и в Departments. И чтобы их различать, мы предваряем имя поля псевдонимом и точкой, т.е. «emp.ID», «emp.Name», «dep.ID», «dep.Name».

Вспоминаем почему удобнее пользоваться именно короткими псевдонимами – потому что, без псевдонимов наш запрос бы выглядел следующим образом:

```
SELECT Employees.ID,Employees.Name,Employees.DepartmentID,Departments.ID,Departments.Name
FROM Employees
JOIN Departments ON Employees.DepartmentID=Departments.ID
```

По мне, стало очень длинно и хуже читаемо, т.к. имена полей визуально потерялись среди повторяющихся имен таблиц.

В многотабличных запросах, хоть и можно указать имя без псевдонима, в случае если имя не дублируется во второй таблице, но я бы рекомендовал всегда использовать псевдонимы в случае соединения, т.к. никто не гарантирует, что поле с таким же именем со временем не добавят во вторую таблицу, а тогда ваш запрос просто сломается, ругаясь на то что он не может понять к какой таблице относится данное поле.

Только используя псевдонимы, мы сможем осуществить соединения таблицы самой с собой. Предположим встала задача, получить для каждого сотрудника, данные сотрудника, который был принят прямо до него (табельный номер отличается на единицу меньше). Допустим, что у нас табельные номера выдаются последовательно и без дырок, тогда мы можем это сделать примерно следующим образом:

```
SELECT
  e1.ID EmpID1,
  e1.Name EmpName1,
  e2.ID EmpID2,
  e2.Name EmpName2
FROM Employees e1
LEFT JOIN Employees e2 ON e1.ID=e2.ID+1 -- получить данные предыдущего сотрудника
```

Т.е. здесь одной таблице Employees, мы дали псевдоним «e1», а второй «e2».

Разбираем каждый вид горизонтального соединения

Для этой цели рассмотрим 2 небольшие абстрактные таблицы, которые так и назовем LeftTable и RightTable:

```
CREATE TABLE LeftTable(
  LCode int,
  LDescr varchar(10)
)
GO

CREATE TABLE RightTable(
  RCode int,
  RDescr varchar(10)
)
GO

INSERT LeftTable(LCode,LDescr)VALUES
(1,'L-1'),
(2,'L-2'),
(3,'L-3'),
(5,'L-5')

INSERT RightTable(RCode,RDescr)VALUES
(2,'B-2'),
(3,'B-3'),
(4,'B-4')
```

Посмотрим, что в этих таблицах:

```
SELECT * FROM LeftTable
```

LCode	LDescr
1	L-1
2	L-2
3	L-3
5	L-5

```
SELECT * FROM RightTable
```

RCode	RDescr
2	B-2
3	B-3
4	B-4

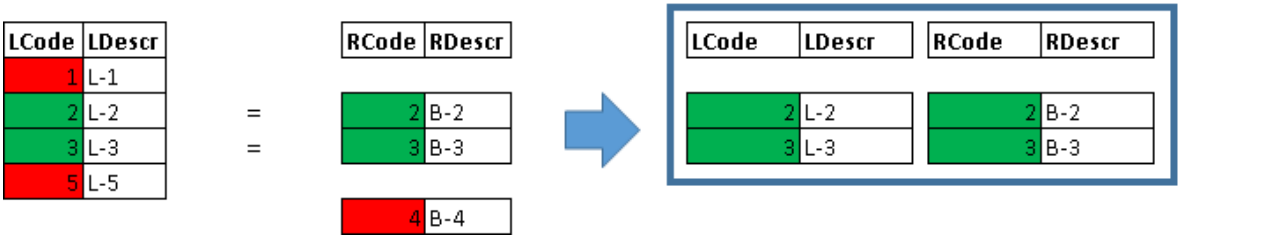
JOIN

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

LCode	LDescr	RCode	RDescr
2	L-2	2	B-2
3	L-3	3	B-3

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

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



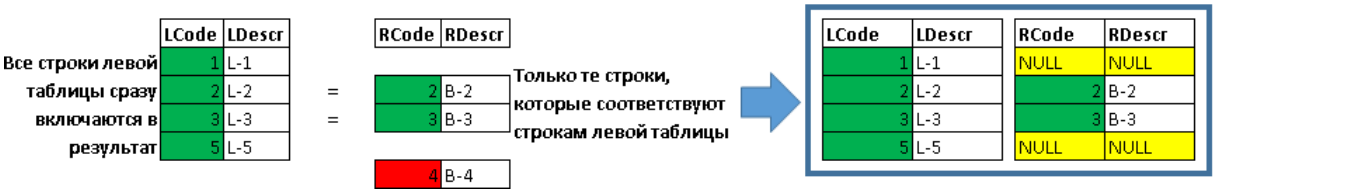
LEFT JOIN

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

LCode	LDescr	RCode	RDescr
1	L-1	NULL	NULL
2	L-2	2	B-2
3	L-3	3	B-3
5	L-5	NULL	NULL

Здесь были возвращены все строки LeftTable, которые были дополнены данными строк из RightTable, для которых выполнилось условие (l.LCode=r.RCode)

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

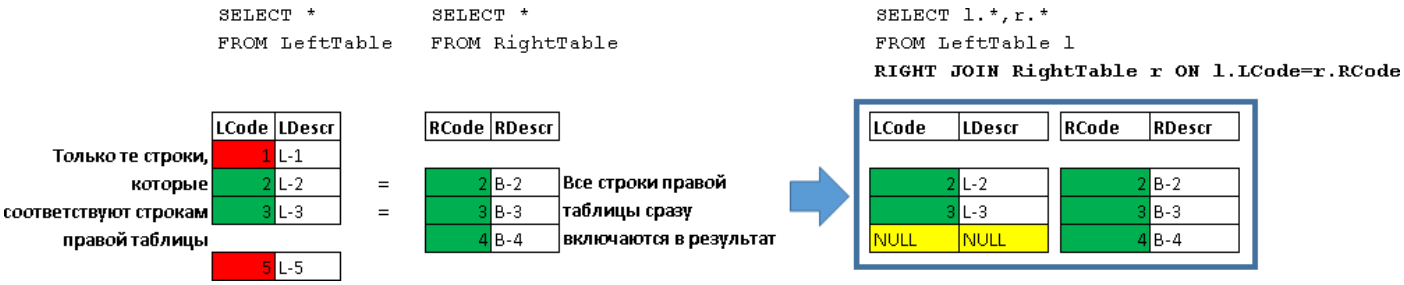


RIGHT JOIN

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

LCode	LDescr	RCode	RDescr
2	L-2	2	B-2
3	L-3	3	B-3
NULL	NULL	4	B-4

Здесь были возвращены все строки RightTable, которые были дополнены данными строк из LeftTable, для которых выполнилось условие (l.LCode=r.RCode)



По сути если мы переставим LeftTable и RightTable местами, то аналогичный результат мы получим при помощи левого соединения:

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

LCode	LDescr	RCode	RDescr
2	L-2	2	B-2
3	L-3	3	B-3
NULL	NULL	4	B-4

Я за собой заметил, что я чаще применяю именно LEFT JOIN, т.е. я сначала думаю, данные какой таблицы мне важны, а потом думаю, какая таблица/таблицы будет играть роль дополняющей таблицы.

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
```

LCode	LDescr	RCode	RDescr
1	L-1	NULL	NULL
2	L-2	2	B-2
3	L-3	3	B-3
5	L-5	NULL	NULL
NULL	NULL	4	B-4

Вернулись все строки из LeftTable и RightTable. Строки для которых выполнилось условие (l.LCode=r.RCode) были объединены в одну строку. Отсутствующие в строке данные с левой или правой стороны заполняются NULL-значениями.

```
SELECT *
FROM LeftTable
```

LCode	LDescr
1	L-1
2	L-2
3	L-3
5	L-5

=
=

```
SELECT *
FROM RightTable
```

RCode	RDescr
2	B-2
3	B-3
4	B-4

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

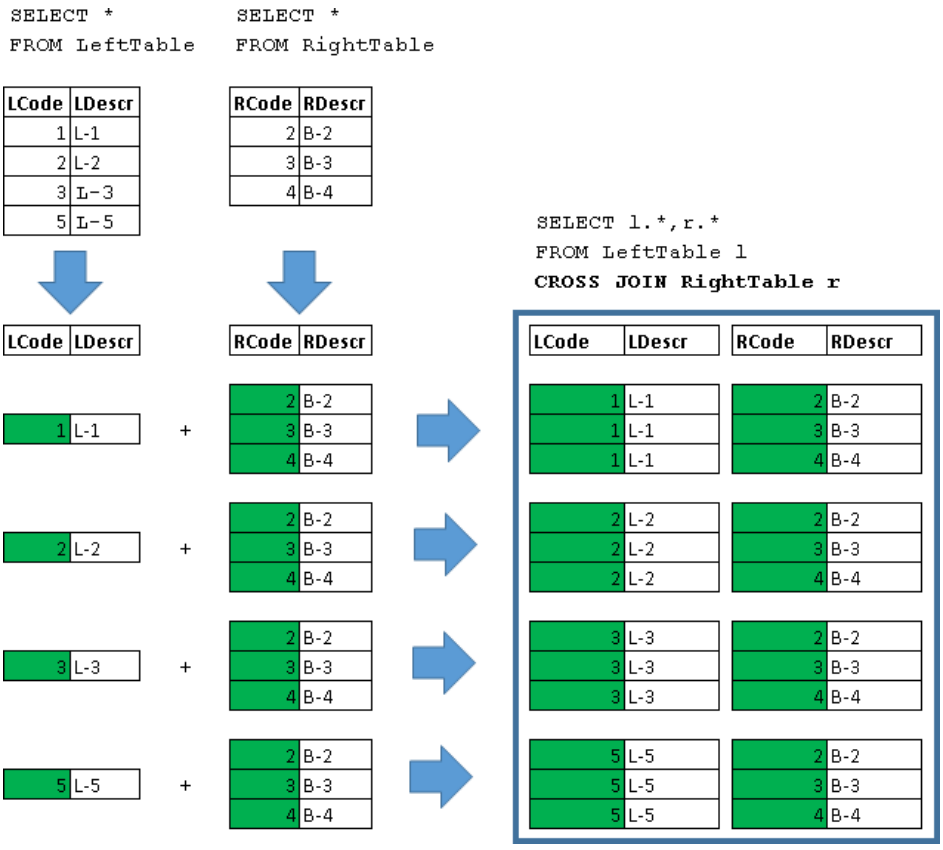
LCode	LDescr	RCode	RDescr
1	L-1	NULL	NULL
2	L-2	2	B-2
3	L-3	3	B-3
5	L-5	NULL	NULL
NULL	NULL	4	B-4

CROSS JOIN

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

LCode	LDescr	RCode	RDescr
1	L-1	2	B-2
2	L-2	2	B-2
3	L-3	2	B-2
5	L-5	2	B-2
1	L-1	3	B-3
2	L-2	3	B-3
3	L-3	3	B-3
5	L-5	3	B-3
1	L-1	4	B-4
2	L-2	4	B-4
3	L-3	4	B-4
5	L-5	4	B-4

Каждая строка LeftTable соединяется с данными всех строк RightTable.



Возвращаемся к таблицам Employees и Departments

Надеюсь вы поняли принцип работы горизонтальных соединений. Если это так, то возвратитесь на начало раздела «JOIN-соединения – операции горизонтального соединения данных» и попробуйте самостоятельно понять примеры с объединением таблиц Employees и Departments, а потом снова возвращайтесь сюда, обсудим это вместе.

Давайте попробуем вместе подвести резюме для каждого запроса:

Запрос	Резюме

`-- JOIN вернет 5 строк`
`SELECT emp.ID,emp.Name,emp.DepartmentID,dep.ID,dep.Name`
`FROM Employees emp`
`JOIN Departments dep ON emp.DepartmentID=dep.ID`
	По сути данный запрос вернет только сотрудников, у которых указано значение DepartmentID.
	Т.е. мы можем использовать данное соединение, в случае, когда нам нужны данные по сотрудникам числящихся за каким-нибудь отделом (без учета внештаткиков).

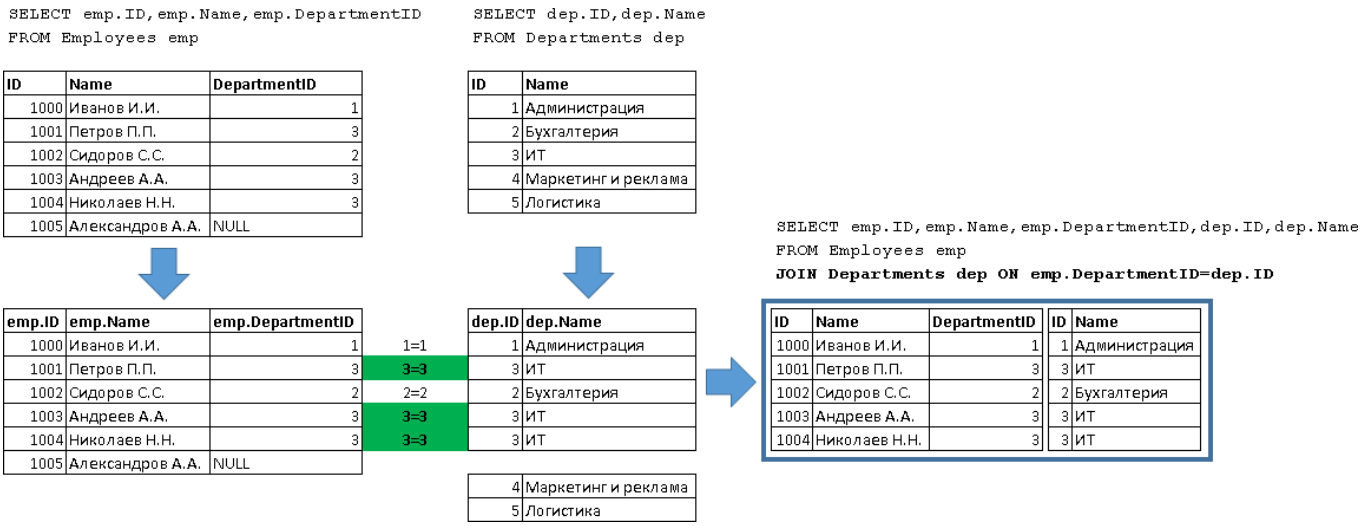
`-- LEFT JOIN вернет 6 строк`
`SELECT emp.ID,emp.Name,emp.DepartmentID,dep.ID,dep.Name`
`FROM Employees emp`
`LEFT JOIN Departments dep ON emp.DepartmentID=dep.ID`
	Вернет всех сотрудников. Для тех сотрудников у которых не указан DepartmentID, поля «dep.ID» и «dep.Name» будут содержать NULL.
	Вспоминайте, что NULL значения в случае необходимости можно обработать, например, при помощи ISNULL(dep.Name,'вне штата').
	Этот вид соединения можно использовать, когда нам важно получить данные по всем сотрудникам, например, чтобы получить список для начисления ЗП.

`-- RIGHT JOIN вернет 7 строк`
`SELECT emp.ID,emp.Name,emp.DepartmentID,dep.ID,dep.Name`
`FROM Employees emp`
`RIGHT JOIN Departments dep ON emp.DepartmentID=dep.ID`
	Здесь мы получили дырки слева, т.е. отдел есть, но сотрудников в этом отделе нет.
	Такое соединение можно использовать, например, когда нужно выяснить, какие отделы и кем у нас заняты, а какие еще не сформированы. Эту информацию можно использовать для поиска и приема новых работников из которых будет формироваться отдел.

`-- FULL JOIN вернет 8 строк`
`SELECT emp.ID,emp.Name,emp.DepartmentID,dep.ID,dep.Name`
`FROM Employees emp`
`FULL JOIN Departments dep ON emp.DepartmentID=dep.ID`
	Этот запрос важен, когда нам нужно получить все данные по сотрудникам и все данные по имеющимся отделам. Соответственно получаем дырки (NULL-значения) либо по сотрудникам, либо по отделам (внештатники).
	Данный запрос, например, может использоваться в целях проверки, все ли сотрудники сидят в правильных отделах, т.к. может у некоторых сотрудников, которые числятся как внештатники, просто забыли указать отдел.

`-- CROSS JOIN вернет 30 строк - (6 строк таблицы Employees) * (5 строк таблицы Departments)`
`SELECT emp.ID,emp.Name,emp.DepartmentID,dep.ID,dep.Name`
`FROM Employees emp`
`CROSS JOIN Departments dep`
	В таком виде даже сложно придумать где это можно применить, поэтому пример с CROSS JOIN я покажу ниже.

Обратите внимание, что в случае повторения значений DepartmentID в таблице Employees, произошло соединение каждой такой строки со строкой из таблицы Departments с таким же ID, то есть данные Departments объединились со всеми записями для которых выполнилось условие (emp.DepartmentID=dep.ID):



ID	Name	DepartmentID
1000	Иванов И.И.	1
1001	Петров П.П.	3
1002	Сидоров С.С.	2
1003	Андреев А.А.	3
1004	Николаев Н.Н.	3
1005	Александров А.А.	NULL

ID	Name
1	Администрация
2	Бухгалтерия
3	ИТ
4	Маркетинг и реклама
5	Логистика

emp.ID	emp.Name	emp.DepartmentID
1000	Иванов И.И.	1
1001	Петров П.П.	3
1002	Сидоров С.С.	2
1003	Андреев А.А.	3
1004	Николаев Н.Н.	3
1005	Александров А.А.	NULL

dep.ID	dep.Name
1	Администрация
3	ИТ
2	Бухгалтерия
3	ИТ
3	ИТ
4	Маркетинг и реклама
5	Логистика

ID	Name	DepartmentID	ID	Name
1000	Иванов И.И.	1	1	Администрация
1001	Петров П.П.	3	3	ИТ
1002	Сидоров С.С.	2	2	Бухгалтерия
1003	Андреев А.А.	3	3	ИТ
1004	Николаев Н.Н.	3	3	ИТ

В нашем случае все получилось правильно, т.е. мы дополнили таблицу Employees, данными таблицы Departments. Я специально заострил на этом внимание, т.к. бывают случаи, когда такое поведение нам не нужно. Для демонстрации поставим задачу – для каждого отдела вывести последнего принятого сотрудника, если сотрудников нет, то просто вывести название отдела. Возможно напрашивается такое решение – просто взять предыдущий запрос и поменять условие соединение на RIGHT JOIN, плюс переставить поля местами:

```
SELECT dep.ID,dep.Name,emp.ID,emp.Name
FROM Employees emp
RIGHT JOIN Departments dep ON emp.DepartmentID=dep.ID
```

ID	Name	ID	Name
1	Администрация	1000	Иванов И.И.
2	Бухгалтерия	1002	Сидоров С.С.
3	ИТ	1001	Петров П.П.
3	ИТ	1003	Андреев А.А.
3	ИТ	1004	Николаев Н.Н.
4	Маркетинг и реклама	NULL	NULL
5	Логистика	NULL	NULL

Но мы для ИТ-отдела получили три строчки, когда нам нужна была только строчка с последним принятым сотрудником, т.е. Николаевым Н.Н.

Задачу такого рода, можно решить, например, при помощи использования подзапроса:

```
SELECT dep.ID,dep.Name,emp.ID,emp.Name
FROM Employees emp

/*
  объединяем с подзапросом возвращающим последний (максимальный - MAX(ID))
  идентификатор сотрудника для каждого отдела (GROUP BY DepartmentID)
*/
JOIN
   (
     SELECT MAX(ID) MaxEmployeeID
     FROM Employees
     GROUP BY DepartmentID
   ) lastEmp
ON emp.ID=lastEmp.MaxEmployeeID

RIGHT JOIN Departments dep ON emp.DepartmentID=dep.ID -- все данные Departments
```

ID	Name	ID	Name
1	Администрация	1000	Иванов И.И.
2	Бухгалтерия	1002	Сидоров С.С.
3	ИТ	1004	Николаев Н.Н.
4	Маркетинг и реклама	NULL	NULL
5	Логистика	NULL	NULL

При помощи предварительного объединения Employees с данными подзапроса, мы смогли оставить только нужных нам для соединения с Departments сотрудников.

Здесь мы плавно переходим к использованию подзапросов. Я думаю использование их в таком виде должно быть для вас понятно на интуитивном уровне. То есть подзапрос подставляется на место таблицы и играет ее роль, ничего сложного. К теме подзапросов мы еще вернемся отдельно.

Посмотрите отдельно, что возвращает подзапрос:

```
SELECT MAX(ID) MaxEmployeeID
FROM Employees
GROUP BY DepartmentID
```

MaxEmployeeID
1005
1000
1002
1004

Т.е. он вернул только идентификаторы последних принятых сотрудников, в разрезе отделов.

Соединения выполняются последовательно сверху-вниз, наращиваясь как снежный ком, который катится с горы. Сначала происходит соединение «Employees emp JOIN (Подзапрос) lastEmp», формируя новый выходной набор:

```
SELECT emp.ID,emp.Name,emp.DepartmentID
FROM Employees emp
```

emp.ID	emp.Name	emp.DepartmentID
1000	Иванов И.И.	1
1001	Петров П.П.	3
1002	Сидоров С.С.	2
1003	Андреев А.А.	3
1004	Николаев Н.Н.	3
1005	Александров А.А.	NULL

```
SELECT MAX(ID) MaxEmployeeID
FROM Employees
GROUP BY DepartmentID
```

	lastEmp.MaxEmployeeID
=	1000
=	1002
=	1004
=	1005

```
SELECT …
FROM Employees emp
JOIN
  (
    SELECT MAX(ID) MaxEmployeeID
    FROM Employees
    GROUP BY DepartmentID
  ) lastEmp
ON emp.ID=lastEmp.MaxEmployeeID
```

emp.ID	emp.Name	emp.DepartmentID	lastEmp.MaxEmployeeID
1000	Иванов И.И.	1	1000
1002	Сидоров С.С.	2	1002
1004	Николаев Н.Н.	3	1004
1005	Александров А.А.	NULL	1005

Потом идет объединение набора, полученного «Employees emp JOIN (Подзапрос) lastEmp» (назовем его условно «ПоследнийРезультат») с Departments, т.е. «ПоследнийРезультат RIGHT JOIN Departments dep»:

```
ПоследнийРезультат
```

emp.ID	emp.Name	emp.DepartmentID	lastEmp.MaxEmployeeID
1000	Иванов И.И.	1	1000
1002	Сидоров С.С.	2	1002
1004	Николаев Н.Н.	3	1004
1005	Александров А.А.	NULL	1005

```
SELECT *
FROM Departments
```

	ID	Name
=	1	Администрация
=	2	Бухгалтерия
=	3	ИТ
	4	Маркетинг и реклама
	5	Логистика

```
SELECT …
FROM ПоследнийРезультат
RIGHT JOIN Departments dep ON emp.DepartmentID=dep.ID
```

emp.ID	emp.Name	emp.DepartmentID	lastEmp.MaxEmployeeID
1000	Иванов И.И.	1	1000
1002	Сидоров С.С.	2	1002
1004	Николаев Н.Н.	3	1004
NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL

ID	Name
1	Администрация
2	Бухгалтерия
3	ИТ
4	Маркетинг и реклама
5	Логистика

Самостоятельная работа для закрепления материала

Если вы новичок, то вам обязательно нужно прорабатывать каждую JOIN-конструкцию, до тех пор, пока вы на 100% не будете понимать, как работает каждый вид соединения и правильно представлять результат какого вида будет получен в итоге.

Для закрепления материала про JOIN-соединения сделаем следующее:

```
-- очистим таблицы LeftTable и RightTable
TRUNCATE TABLE LeftTable
TRUNCATE TABLE RightTable
GO

-- и зальем в них другие данные
INSERT LeftTable(LCode,LDescr)VALUES
(1,'L-1'),
(2,'L-2a'),
(2,'L-2b'),
(3,'L-3'),
(5,'L-5')

INSERT RightTable(RCode,RDescr)VALUES
(2,'B-2a'),
(2,'B-2b'),
(3,'B-3'),
(4,'B-4')
```

Посмотрим, что в таблицах:

```
SELECT *
FROM LeftTable
```

LCode	LDescr
1	L-1
2	L-2a
2	L-2b
3	L-3
5	L-5

```
SELECT *
FROM RightTable
```

RCode	RDescr
2	B-2a
2	B-2b
3	B-3
4	B-4

А теперь попытайтесь сами разобрать, каким образом получилась каждая строчка запроса с каждым видом соединения (Excel вам в помощь):

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

LCode	LDescr	RCode	RDescr
2	L-2a	2	B-2a
2	L-2a	2	B-2b
2	L-2b	2	B-2a
2	L-2b	2	B-2b
3	L-3	3	B-3

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

LCode	LDescr	RCode	RDescr
1	L-1	NULL	NULL
2	L-2a	2	B-2a
2	L-2a	2	B-2b
2	L-2b	2	B-2a
2	L-2b	2	B-2b
3	L-3	3	B-3
5	L-5	NULL	NULL

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

LCode	LDescr	RCode	RDescr
2	L-2a	2	B-2a
2	L-2b	2	B-2a
2	L-2a	2	B-2b
2	L-2b	2	B-2b
3	L-3	3	B-3
NULL	NULL	4	B-4

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

LCode	LDescr	RCode	RDescr
1	L-1	NULL	NULL
2	L-2a	2	B-2a
2	L-2a	2	B-2b
2	L-2b	2	B-2a
2	L-2b	2	B-2b
3	L-3	3	B-3
5	L-5	NULL	NULL
NULL	NULL	4	B-4

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

LCode	LDescr	RCode	RDescr
1	L-1	2	B-2a
2	L-2a	2	B-2a
2	L-2b	2	B-2a
3	L-3	2	B-2a
5	L-5	2	B-2a
1	L-1	2	B-2b
2	L-2a	2	B-2b
2	L-2b	2	B-2b
3	L-3	2	B-2b
5	L-5	2	B-2b
1	L-1	3	B-3
2	L-2a	3	B-3
2	L-2b	3	B-3
3	L-3	3	B-3
5	L-5	3	B-3
1	L-1	4	B-4
2	L-2a	4	B-4
2	L-2b	4	B-4
3	L-3	4	B-4
5	L-5	4	B-4

Еще раз про JOIN-соединения

Еще один пример с использованием нескольких последовательных операций соединении. Здесь повтор получился не специально, так получилось – не выбрасывать же материал. ;) Но ничего «повторение – мать учения».

Если используется несколько операций соединения, то в таком случае они применяются последовательно сверху-вниз. Грубо говоря, после каждого соединения создается новый набор и следующее соединение уже происходит с этим расширенным набором. Рассмотрим простой пример:

```
SELECT
  e.ID,
  e.Name EmployeeName,
  p.Name PositionName,
  d.Name DepartmentName
FROM Employees e
LEFT JOIN Departments d ON e.DepartmentID=d.ID
LEFT JOIN Positions p ON e.PositionID=p.ID
```

Первым делом выбрались все записи таблицы Employees:

```
SELECT
  e.*
FROM Employees e -- 1
```

Дальше произошло соединение с таблицей Departments:

```
SELECT
  e.*, -- к полям Employees
  d.*  -- добавились соответствующие (e.DepartmentID=d.ID) поля Departments
FROM Employees e -- 1
LEFT JOIN Departments d ON e.DepartmentID=d.ID -- 2
```

Дальше уже идет соединение этого набора с таблицей Positions:

```
SELECT
  e.*, -- к полям Employees
  d.*, -- добавились соответствующие (e.DepartmentID=d.ID) поля Departments
  p.*  -- добавились соответствующие (e.PositionID=p.ID) поля Positions
FROM Employees e -- 1
LEFT JOIN Departments d ON e.DepartmentID=d.ID -- 2
LEFT JOIN Positions p ON e.PositionID=p.ID -- 3
```

Т.е. это выглядит примерно так:

```
SELECT e.*
FROM Employees e
```

e.ID	e.Name	...	e.BonusPercent
1000	Иванов И.И.		50
1001	Петров П.П.		15
1002	Сидоров С.С.		NULL
1003	Андреев А.А.		30
1004	Николаев Н.Н.		NULL
1005	Александров А.А.		NULL

```
SELECT e.*,d.*
FROM Employees e
LEFT JOIN Departments d ON e.DepartmentID=d.ID
```

e.ID	e.Name	...	e.BonusPercent	d.ID	d.Name
1000	Иванов И.И.		50	1	Администрация
1001	Петров П.П.		15	3	ИТ
1002	Сидоров С.С.		NULL	2	Бухгалтерия
1003	Андреев А.А.		30	3	ИТ
1004	Николаев Н.Н.		NULL	3	ИТ
1005	Александров А.А.		NULL	NULL	NULL

```
SELECT e.*,d.*,p.*
FROM Employees e
LEFT JOIN Departments d ON e.DepartmentID=d.ID
LEFT JOIN Positions p ON e.PositionID=p.ID
```

e.ID	e.Name	...	e.BonusPercent	d.ID	d.Name	p.ID	p.Name
1000	Иванов И.И.		50	1	Администрация	2	Директор
1001	Петров П.П.		15	3	ИТ	3	Программист
1002	Сидоров С.С.		NULL	2	Бухгалтерия	1	Бухгалтер
1003	Андреев А.А.		30	3	ИТ	4	Старший программист
1004	Николаев Н.Н.		NULL	3	ИТ	3	Программист
1005	Александров А.А.		NULL	NULL	NULL	NULL	NULL

И в последнюю очередь идет возврат тех данных, которые мы просим вывести:

```
SELECT

  e.ID, -- 1. идентификатор сотрудника
  e.Name EmployeeName, -- 2. имя сотрудника
  p.Name PositionName, -- 3. название должности
  d.Name DepartmentName -- 4. название отдела
FROM Employees e
LEFT JOIN Departments d ON e.DepartmentID=d.ID
LEFT JOIN Positions p ON e.PositionID=p.ID
```

Соответственно, ко всему этому полученному набору можно применить фильтр WHERE и сортировку ORDER BY:

```
SELECT
  e.ID, -- 1. идентификатор сотрудника
  e.Name EmployeeName, -- 2. имя сотрудника
  p.Name PositionName, -- 3. название должности
  d.Name DepartmentName -- 4. название отдела
FROM Employees e
LEFT JOIN Departments d ON e.DepartmentID=d.ID
LEFT JOIN Positions p ON e.PositionID=p.ID
WHERE d.ID=3 -- используем поля из поле ID из Departments
  AND p.ID=3 -- используем для фильтрации поле ID из Positions
ORDER BY e.Name -- используем для сортировки поле Name из Employees
```

ID	EmployeeName	PositionName	DepartmentName
1004	Николаев Н.Н.	Программист	ИТ
1001	Петров П.П.	Программист	ИТ

То есть последний полученный набор – представляет собой такую же таблицу, над которой можно выполнять базовый запрос:

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

То есть если раньше в роли источника выступала только одна таблица, то теперь на это место мы просто подставляем наше выражение:

```
Employees e
LEFT JOIN Departments d ON e.DepartmentID=d.ID
LEFT JOIN Positions p ON e.PositionID=p.ID
```

В результате чего получаем тот же самый базовый запрос:

```
SELECT
  e.ID,
  e.Name EmployeeName,
  p.Name PositionName,
  d.Name DepartmentName
FROM

      /* источник - начало */
      Employees e
      LEFT JOIN Departments d ON e.DepartmentID=d.ID
      LEFT JOIN Positions p ON e.PositionID=p.ID
      /* источник - конец */

WHERE d.ID=3
  AND p.ID=3
ORDER BY e.Name
```

А теперь, применим группировку:

```
SELECT
  ISNULL(dep.Name,'Прочие') DepName,
  COUNT(DISTINCT emp.PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(emp.Salary) SalaryAmount,
  AVG(emp.Salary) SalaryAvg -- плюс выполняем пожелание директора
FROM

    /* источник - начало */
    Employees emp
    LEFT JOIN Departments dep ON emp.DepartmentID=dep.ID
    /* источник - конец */

GROUP BY emp.DepartmentID,dep.Name
ORDER BY DepName
```

Видите, мы все так же крутимся вокруг да около базовых конструкций, теперь надеюсь понятно, почему очень важно в первую очередь хорошо понять их.

И как мы увидели, в запросе на месте любой таблицы может стоять подзапрос. В свою очередь подзапросы могут быть вложены в подзапросы. И все эти подзапросы тоже представляют из себя базовые конструкции. То есть базовая конструкция, это кирпичики, из которых строится любой запрос.

Обещанный пример с CROSS JOIN

Давайте используем соединение CROSS JOIN, чтобы подсчитать сколько сотрудников, в каком отделе и на каких должностях числится. Для каждого отдела перечислим все существующие должности:

```
SELECT
  d.Name DepartmentName,
  p.Name PositionName,
  e.EmplCount
FROM Departments d
CROSS JOIN Positions p
LEFT JOIN
  (
    /*
      здесь я использовал подзапрос для подсчета сотрудников
      в разрезе групп (DepartmentID,PositionID)
    */
    SELECT DepartmentID,PositionID,COUNT(*) EmplCount
    FROM Employees
    GROUP BY DepartmentID,PositionID
  ) e
ON e.DepartmentID=d.ID AND e.PositionID=p.ID
ORDER BY DepartmentName,PositionName
```

Results | Messages

	DepartmentName	PositionName	EmplCount
1	Администрация	Бухгалтер	NULL
2	Администрация	Директор	1
3	Администрация	Кладовщик	NULL
4	Администрация	Логист	NULL
5	Администрация	Маркетолог	NULL
6	Администрация	Программист	NULL
7	Администрация	Старший программист	NULL
8	Бухгалтерия	Бухгалтер	1
9	Бухгалтерия	Директор	NULL
10	Бухгалтерия	Кладовщик	NULL
11	Бухгалтерия	Логист	NULL
12	Бухгалтерия	Маркетолог	NULL
13	Бухгалтерия	Программист	NULL
14	Бухгалтерия	Старший программист	NULL
15	ИТ	Бухгалтер	NULL
16	ИТ	Директор	NULL
17	ИТ	Кладовщик	NULL
18	ИТ	Логист	NULL

Results | Messages

	DepartmentName	PositionName	EmplCount
19	ИТ	Маркетолог	NULL
20	ИТ	Программист	2
21	ИТ	Старший программист	1
22	Логистика	Бухгалтер	NULL
23	Логистика	Директор	NULL
24	Логистика	Кладовщик	NULL
25	Логистика	Логист	NULL
26	Логистика	Маркетолог	NULL
27	Логистика	Программист	NULL
28	Логистика	Старший программист	NULL
29	Маркетинг и ре...	Бухгалтер	NULL
30	Маркетинг и ре...	Директор	NULL
31	Маркетинг и ре...	Кладовщик	NULL
32	Маркетинг и ре...	Логист	NULL
33	Маркетинг и ре...	Маркетолог	NULL
34	Маркетинг и ре...	Программист	NULL
35	Маркетинг и ре...	Старший программист	NULL

В данном случае сначала выполнилось соединение при помощи CROSS JOIN, а затем к полученному набору сделалось соединение с данными из подзапроса при помощи LEFT JOIN. Вместо таблицы в LEFT JOIN мы использовали подзапрос.

Подзапрос заключается в скобки и ему присваивается псевдоним, в данном случае это «e».

То есть в данном случае объединение происходит не с таблицей, а с результатом следующего запроса:

```
SELECT DepartmentID,PositionID,COUNT(*) EmplCount
FROM Employees
GROUP BY DepartmentID,PositionID
```

DepartmentID	PositionID	EmplCount
NULL	NULL	1
2	1	1
1	2	1
3	3	2
3	4	1

Вместе с псевдонимом «e» мы можем использовать имена DepartmentID, PositionID и EmplCount. По сути дальше подзапрос ведет себя так же, как если на его месте стояла таблица. Соответственно, как и у таблицы,
все имена колонок, которые возвращает подзапрос, должны быть заданы явно и не должны повторяться.

Связь при помощи WHERE-условия

Для примера перепишем следующий запрос с JOIN-соединением:

```
SELECT emp.ID,emp.Name,emp.DepartmentID,dep.ID,dep.Name
FROM Employees emp
JOIN Departments dep ON emp.DepartmentID=dep.ID -- условие соединения таблиц
WHERE emp.DepartmentID=3 -- условие фильтрации данных
```

Через WHERE-условие он примет следующую форму:

```
SELECT emp.ID,emp.Name,emp.DepartmentID,dep.ID,dep.Name
FROM
  Employees emp,
  Departments dep
WHERE emp.DepartmentID=dep.ID -- условие соединения таблиц
  AND emp.DepartmentID=3 -- условие фильтрации данных
```

Здесь плохо то, что происходит смешивание условий соединения таблиц (emp.DepartmentID=dep.ID) с условием фильтрации (emp.DepartmentID=3).

Теперь посмотрим, как сделать CROSS JOIN:

```
SELECT emp.ID,emp.Name,emp.DepartmentID,dep.ID,dep.Name
FROM Employees emp
CROSS JOIN Departments dep -- декартово соединение (соединение без условия)
WHERE emp.DepartmentID=3 -- условие фильтрации данных
```

Через WHERE-условие он примет следующую форму:

```
SELECT emp.ID,emp.Name,emp.DepartmentID,dep.ID,dep.Name
FROM
  Employees emp,
  Departments dep
WHERE emp.DepartmentID=3 -- условие фильтрации данных
```

Т.е. в этом случае мы просто не указали условие соединения таблиц Employees и Departments. Чем плох этот запрос? Представьте, что кто-то другой смотрит на ваш запрос и думает «кажется тот, кто писал запрос забыл здесь дописать условие (emp.DepartmentID=dep.ID)» и с радостью, что обнаружил косяк, дописывает это условие. В результате чего задуманное вами может сломаться, т.к. вы подразумевали CROSS JOIN. Так что, если вы делаете декартово соединение, то лучше явно укажите, что это именно оно, используя конструкцию CROSS JOIN.

Для оптимизатора запроса может быть и без разницы как вы реализуете соединение (при помощи WHERE или JOIN), он их может выполнить абсолютно одинаково. Но из соображения понимаемости кода, я бы рекомендовал в современных СУБД стараться никогда не делать соединение таблиц при помощи WHERE-условия. Использовать WHERE-условия для соединения, в том случае, если в СУБД реализованы конструкции JOIN, я бы назвал сейчас моветоном. WHERE-условия служат для фильтрации набора, и не нужно перемешивать условия служащие для соединения, с условиями отвечающими за фильтрацию. Но если вы пришли к выводу, что без реализации соединения через WHERE не обойтись, то конечно приоритет за решеной задачей и «к черту все устои».

UNION-объединения – операции вертикального объединения результатов запросов

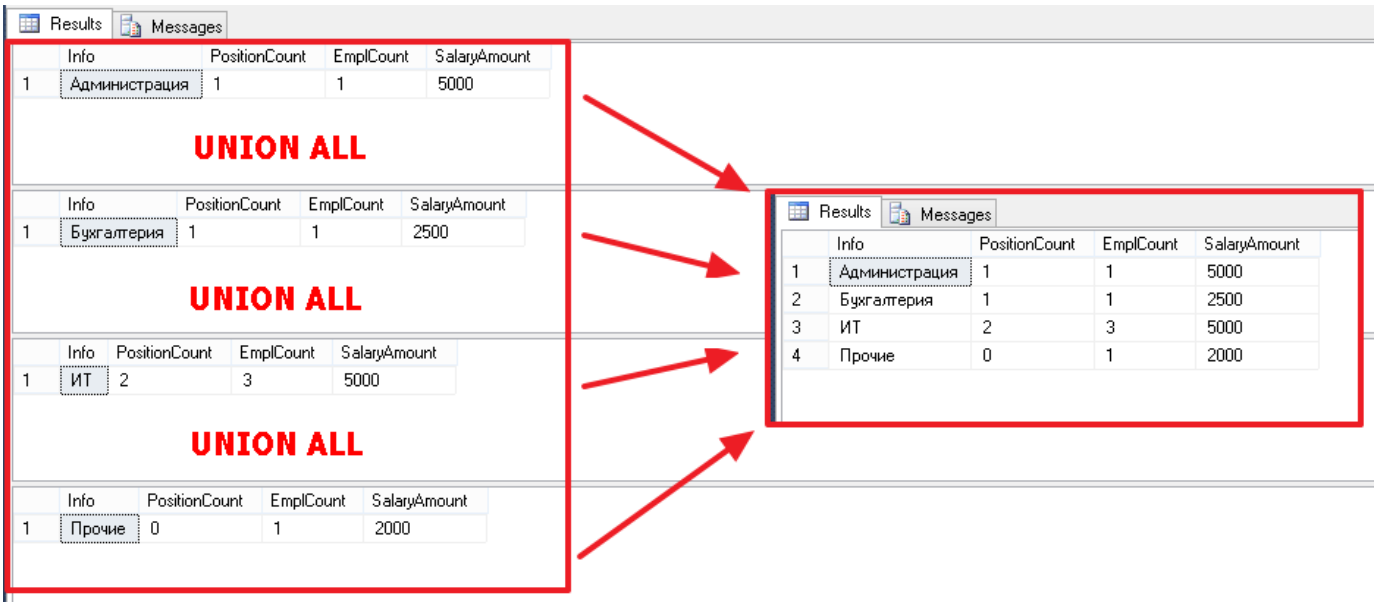
Я специально использую словосочетания горизонтальное соединение и вертикальное объединение, т.к. заметил, что новички часто недопонимают и путают суть этих операций.

Давайте первым делом вспомним как мы делали первую версию отчета для директора:

```
SELECT
  'Администрация' Info,
  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount
FROM Employees
WHERE DepartmentID=1 -- данные по Администрации

SELECT
  'Бухгалтерия' Info,
  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount
FROM Employees
WHERE DepartmentID=2 -- данные по Бухгалтерии

SELECT
  'ИТ' Info,
  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount
FROM Employees
WHERE DepartmentID=3 -- данные по ИТ отделу

SELECT
  'Прочие' Info,
  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount
FROM Employees
WHERE DepartmentID IS NULL -- и еще не забываем данные по внештатникам
```

Так вот, если бы мы не знали, что существует операция группировки, но знали бы, что существует операция объединения результатов запроса при помощи UNION ALL, то мы могли бы склеить все эти запросы следующим образом:

```
SELECT
  'Администрация' Info,
  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount
FROM Employees
WHERE DepartmentID=1 -- данные по Администрации
UNION ALL
SELECT
  'Бухгалтерия' Info,
  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount
FROM Employees
WHERE DepartmentID=2 -- данные по Бухгалтерии
UNION ALL
SELECT
  'ИТ' Info,
  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount
FROM Employees
WHERE DepartmentID=3 -- данные по ИТ отделу
UNION ALL
SELECT
  'Прочие' Info,
  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount
FROM Employees
WHERE DepartmentID IS NULL -- и еще не забываем данные по внештатникам
```



Т.е. UNION ALL позволяет склеить результаты, полученные разными запросами в один общий результат.

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

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)

Все это проще понять на наглядном примере.

Создадим 2 таблицы и наполним их данными:

```
CREATE TABLE TopTable(
  T1 int,
  T2 varchar(10)
)
GO

CREATE TABLE BottomTable(
  B1 int,
  B2 varchar(10)
)
GO

INSERT TopTable(T1,T2)VALUES
(1,'Text 1'),
(1,'Text 1'),
(2,'Text 2'),
(3,'Text 3'),
(4,'Text 4'),
(5,'Text 5')


INSERT BottomTable(B1,B2)VALUES
(2,'Text 2'),
(3,'Text 3'),
(6,'Text 6'),
(6,'Text 6')
```

Посмотрим на содержимое:

```
SELECT *
FROM TopTable
```

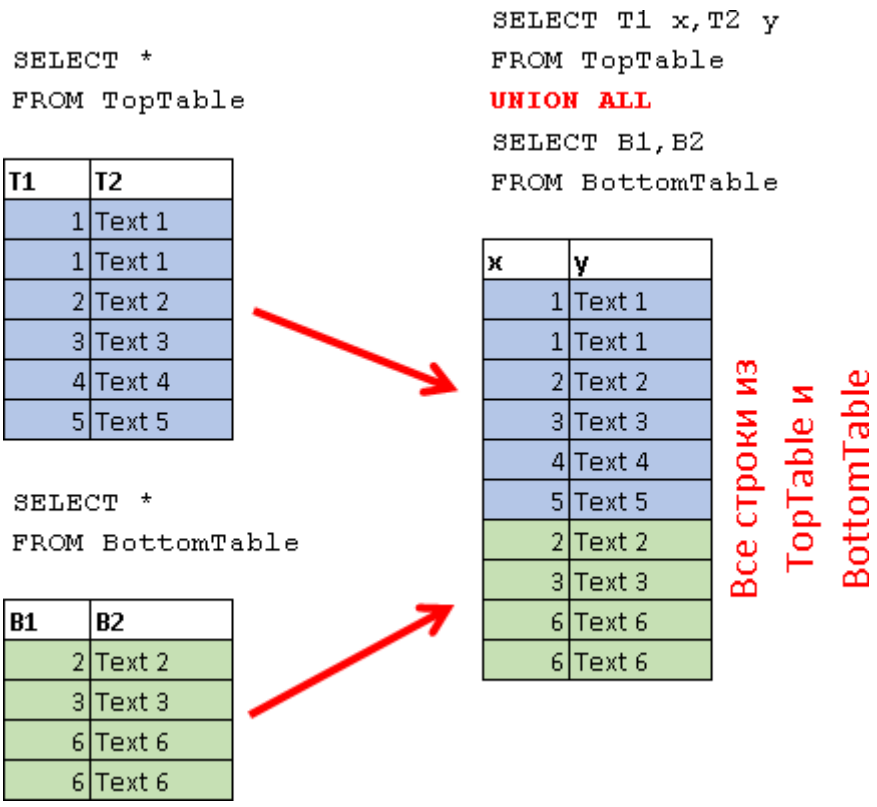
T1	T2
1	Text 1
1	Text 1
2	Text 2
3	Text 3
4	Text 4
5	Text 5

```
SELECT *
FROM BottomTable
```

B1	B2
2	Text 2
3	Text 3
6	Text 6
6	Text 6

UNION ALL

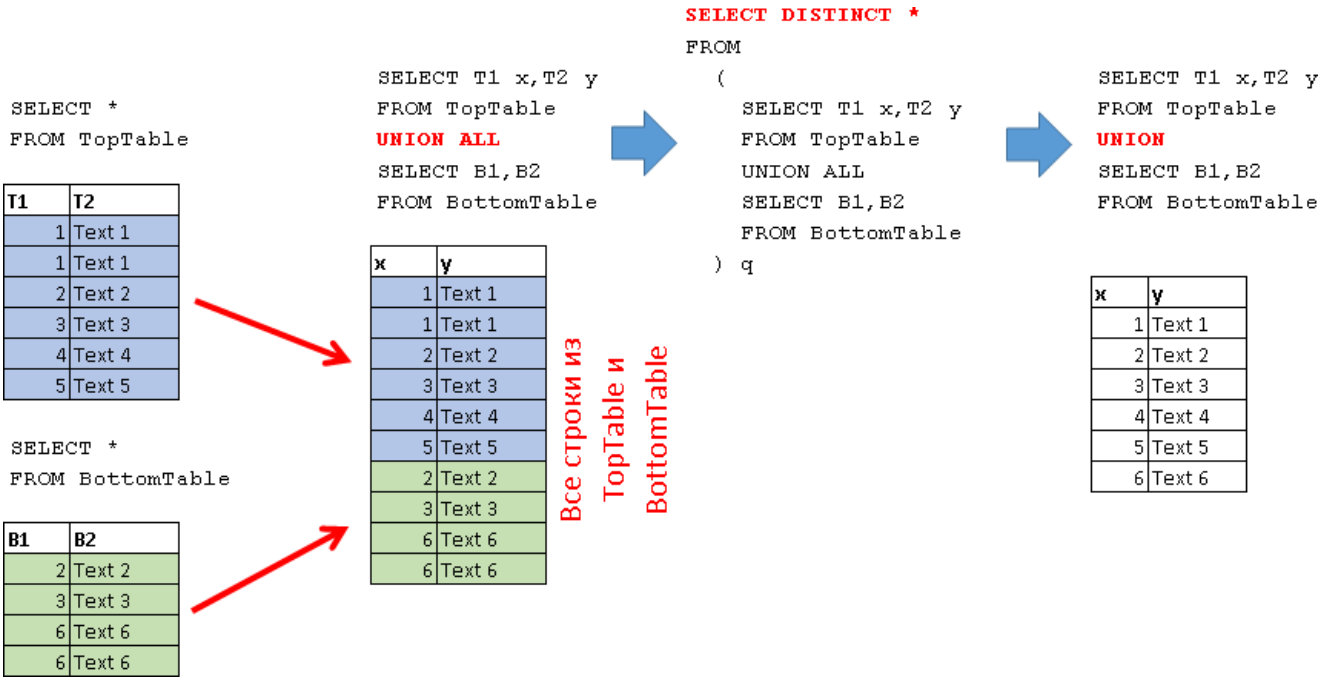
```
SELECT T1 x,T2 y
FROM TopTable

UNION ALL

SELECT B1,B2
FROM BottomTable
```



UNION

```
SELECT T1 x,T2 y
FROM TopTable

UNION

SELECT B1,B2
FROM BottomTable
```

По сути UNION можно представить, как UNION ALL, к которому применена операция DISTINCT:



EXCEPT

```
SELECT T1 x,T2 y
FROM TopTable

EXCEPT

SELECT B1,B2
FROM BottomTable
```

```
SELECT DISTINCT *
FROM TopTable
```

T1	T2
1	Text 1
2	Text 2
3	Text 3
4	Text 4
5	Text 5

```
SELECT T1 x,T2 y
FROM TopTable
EXCEPT
SELECT B1,B2
FROM BottomTable
```

x	y
1	Text 1
4	Text 4
5	Text 5

Уникальные строки из TopTable, которых нет в BottomTable

```
SELECT DISTINCT *
FROM BottomTable
```

B1	B2
2	Text 2
3	Text 3
6	Text 6

INTERSECT

```
SELECT T1 x,T2 y
FROM TopTable

INTERSECT

SELECT B1,B2
FROM BottomTable
```

```
SELECT DISTINCT *
FROM TopTable
```

T1	T2
1	Text 1
2	Text 2
3	Text 3
4	Text 4
5	Text 5

```
SELECT DISTINCT *
FROM BottomTable
```

B1	B2
2	Text 2
3	Text 3
6	Text 6

```
SELECT T1 x,T2 y
FROM TopTable
INTERSECT
SELECT B1,B2
FROM BottomTable
```

x	y
2	Text 2
3	Text 3

Одинаковые строки из TopTable и BottomTable. Без дубликатов.

Завершаем разговор о UNION-соединениях

Вот в принципе и все, что касается вертикальных объединений, это намного проще, чем JOIN-соединения.

Чаще всего в моей в практике находит применение UNION ALL, но и другие виды вертикальных объединений находят свое применение.

При нескольких операциях вертикально объединения, не гарантируется, что они будут выполняться последовательно сверху-вниз. Создадим еще одну таблицу и рассмотрим это на примере:

```
CREATE TABLE NextTable(
  N1 int,
  N2 varchar(10)
)
GO

INSERT NextTable(N1,N2)VALUES
(1,'Text 1'),
(4,'Text 4'),
(6,'Text 6')
```

Например, если мы напишем просто:

```
SELECT T1 x,T2 y
FROM TopTable

EXCEPT

SELECT B1,B2
FROM BottomTable

INTERSECT

SELECT N1,N2
FROM NextTable
```

То мы получим:

x	y
1	Text 1
2	Text 2
3	Text 3
4	Text 4
5	Text 5

Т.е. получается сначала выполнился INTERSECT, а после EXCEPT. Хотя логически будто должно было быть наоборот, т.е. идти сверху-вниз.
Я редко использую эти операции объединений, а тем более в таком виде, поэтому, чтобы не думать не гадать, в какой очередности он выполняет объединения, можно просто при помощи скобок явно указать последовательность объединений, давайте скажем, что сначала нужно сделать EXCEPT, а потом INTERSECT:

```
(
   SELECT T1 x,T2 y
   FROM TopTable

   EXCEPT

   SELECT B1,B2
   FROM BottomTable
)

INTERSECT

SELECT N1,N2
FROM NextTable
```

x	y
1	Text 1
4	Text 4

Вот теперь я получил то, что и хотел.

Я не знаю работает ли такой синтаксис в других СУБД, но если что используйте подзапрос:

```
SELECT x,y
FROM
   (
      SELECT T1 x,T2 y
      FROM TopTable

      EXCEPT

      SELECT B1,B2
      FROM BottomTable
   ) q

INTERSECT

SELECT N1,N2
FROM NextTable
```

При использовании ORDER BY сортировка применяется к окончательному набору:

```
SELECT T1 x,T2 y
FROM TopTable

UNION ALL

SELECT B1,B2
FROM BottomTable

UNION ALL

SELECT B1,B2
FROM BottomTable

ORDER BY x DESC
```

Для задания сортировки здесь удобней использовать псевдоним колонки, заданный в первом запросе.

Самое главное про UNION-объединения я вроде написал, если что поиграйте с UNION-объединениями самостоятельно.

Примечание. В СУБД Oracle тоже есть такие же виды соединения, разница только в операции EXCEPT, там она называется MINUS.

Использование подзапросов

Подзапросы я оставил на 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.

Косвенно мы уже использовали подзапросы в блоке FROM. Там результат, возвращаемый подзапросом по сути играет роль новой таблицы. Думаю, большого смысла останавливаться здесь нет смысла. Просто рассмотрим абстрактный пример с объединением 2-х подзапросов:

```
SELECT q1.x1,q1.y1,q2.x2,q2.y2
FROM
  (
    SELECT T1 x1,T2 y1
    FROM TopTable

    EXCEPT

    SELECT B1,B2
    FROM BottomTable
  ) q1
JOIN
  (
    SELECT T1 x2,T2 y2
    FROM TopTable

    EXCEPT

    SELECT N1,N2
    FROM NextTable
  ) q2
ON q1.x1=q2.x2
```

Если не понятно, сразу, то разбирайте такие запросы по частям. Т.е. сначала посмотрите, что возвращает первый подзапрос «q1», потом, что возвращает второй подзапрос «q2», а затем выполните операцию JOIN над результатами подзапросов «q1» и «q2».

Конструкция WITH

Это достаточно полезная конструкция особенно в случае работы с большими подзапросами.

Сравним:

```
SELECT q1.x1,q1.y1,q2.x2,q2.y2
FROM
  (
    SELECT T1 x1,T2 y1
    FROM TopTable

    EXCEPT

    SELECT B1,B2
    FROM BottomTable
  ) q1
JOIN
  (
    SELECT T1 x2,T2 y2
    FROM TopTable

    EXCEPT

    SELECT N1,N2
    FROM NextTable
  ) q2
ON q1.x1=q2.x2
```

То же самое написанное при помощи WITH:

```
WITH q1 AS(
    SELECT T1 x1,T2 y1
    FROM TopTable

    EXCEPT

    SELECT B1,B2
    FROM BottomTable
),
q2 AS(
    SELECT T1 x2,T2 y2
    FROM TopTable

    EXCEPT

    SELECT N1,N2
    FROM NextTable
)

-- основной запрос становится более прозрачным
SELECT q1.x1,q1.y1,q2.x2,q2.y2
FROM q1
JOIN q2 ON q1.x1=q2.x2
```

Как видим большие подзапросы вынесены и поименованы в блоке WITH, что дало возможность разгрузить текст основного запроса и сделать его понятным.

Вспомним так же пример из предыдущей части, где использовалось представление ViewEmployeesInfo:

```
CREATE VIEW ViewEmployeesInfo
AS
SELECT
  emp.*, -- вернуть все поля таблицы Employees
  dep.Name DepartmentName, -- к этим полям добавить поле Name из таблицы Departments
  pos.Name PositionName -- и еще добавить поле Name из таблицы Positions
FROM Employees emp
LEFT JOIN Departments dep ON emp.DepartmentID=dep.ID
LEFT JOIN Positions pos ON emp.PositionID=pos.ID
```

И запрос, который использовал данное представление:

```
SELECT
  DepartmentName,
  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount,
  AVG(Salary) SalaryAvg
FROM ViewEmployeesInfo emp
GROUP BY DepartmentID,DepartmentName
ORDER BY DepartmentName
```

По сути WITH дает нам возможность разместить текст из представления непосредственно в запросе, т.е. смысл один и тот же:

```
WITH cteEmployeesInfo AS(
   SELECT
     emp.*, -- вернуть все поля таблицы Employees
     dep.Name DepartmentName, -- к этим полям добавить поле Name из таблицы Departments
     pos.Name PositionName -- и еще добавить поле Name из таблицы Positions
  FROM Employees emp
  LEFT JOIN Departments dep ON emp.DepartmentID=dep.ID
  LEFT JOIN Positions pos ON emp.PositionID=pos.ID
)
SELECT
  DepartmentName,
  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount,
  AVG(Salary) SalaryAvg
FROM cteEmployeesInfo emp
GROUP BY DepartmentID,DepartmentName
ORDER BY DepartmentName
```

Только в случае созданного представления мы можем использовать его из разных запросов, т.к. представление создается на уровне БД. Тогда как подзапрос оформленный в блоке WITH виден только в рамках этого запроса.

Использование WITH по-другому называет CTE-выражениями:

Общие табличные выражения (CTE — Common Table Expressions) позволяют существенно уменьшить объем кода, если многократно приходится обращаться к одним и тем же запросам. CTE играет роль представления, которое создается в рамках одного запроса и, не сохраняется как объект схемы.

У CTE есть еще одно важное назначение, с его помощью можно написать рекурсивный запрос.

Приведу только небольшой пример рекурсивного запроса. Отобразим сотрудников с учетом их подчинения другому сотруднику (если помните, у нас в таблице Employees ключ, ссылающийся на эту же таблицу).

```
WITH cteEmpl AS(
   SELECT ID,CAST(Name AS nvarchar(300)) Name,1 EmpLevel
   FROM Employees
   WHERE ManagerID IS NULL -- все сотрудники у которых нет вышестоящего

   UNION ALL

   SELECT emp.ID,CAST(SPACE(cte.EmpLevel*5)+emp.Name AS nvarchar(300)),cte.EmpLevel+1
   FROM Employees emp
   JOIN cteEmpl cte ON emp.ManagerID=cte.ID
)
SELECT *
FROM cteEmpl
```

ID	Name	EmpLevel
1000	Иванов И.И.	1
1002	_____Сидоров С.С.	2
1003	_____Андреев А.А.	2
1005	_____Александров А.А.	2
1001	__________Петров П.П.	3
1004	__________Николаев Н.Н.	3

Для наглядности пробелы заменены знаками подчеркивания.

Рассматривать, как строятся рекурсивные запросы, в рамках данного учебника я не буду. Я считаю, это достаточно специфичная тема для начинающих, и она пока совсем ни к чему им. Прежде чем приступать к изучению рекурсивных запросов нужно уверено научиться пользоваться всеми основными конструкциями, о которых я рассказал, без данной базы дальше двигаться не стоит. В большинстве случаев, знание базовых конструкций, вам будет достаточно для написания запросов любой сложности.

Продолжаем разговор про подзапросы

Давайте теперь рассмотрим, как можно использовать подзапросы еще, а также передавать в них параметры при помощи псевдонима из основного запроса.

Здесь я уже не буду сильно углубляться в разъяснение, т.к. к этому этапу вы уже должны были научиться мыслить и понимать принцип работы с данными. Обязательно практикуйтесь, выполняйте примеры и пытайтесь и понять полученный результат. Чтобы понять, нужно прочувствовать каждый пример себе.

Подзапрос можно использовать в блоке SELECT

Вернемся к нашему отчету:

```
SELECT
  DepartmentID,
  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount,
  AVG(Salary) SalaryAvg -- плюс выполняем пожелание директора
FROM Employees
GROUP BY DepartmentID
```

Здесь название отдела можно так же достать при помощи подзапроса с параметром:

```
SELECT
  /*
    используем подзапрос с параметром
    в данном случае подзапрос должен возвращать одну строку
    и только одно значение
  */
  (SELECT Name FROM Departments dep WHERE dep.ID=emp.DepartmentID) DepartmentName,

  COUNT(DISTINCT PositionID) PositionCount,
  COUNT(*) EmplCount,
  SUM(Salary) SalaryAmount,
  AVG(Salary) SalaryAvg
FROM Employees emp -- задаем псевдоним
GROUP BY DepartmentID
ORDER BY DepartmentName
```

В данном случае подзапрос (SELECT Name FROM Departments dep WHERE dep.ID=emp.DepartmentID) выполнится 4 раза, т.е. для каждого значения emp.DepartmentID

Подзапрос в данном случае должен возвращать только одну строку и одну колонку. Если в подзапросе получается много строк, то используйте в нем либо TOP, либо какую-нибудь агрегатную функцию, чтобы в итоге получилась одна строка. Например, получим для каждого отдела ID последнего принятого сотрудника:

```
SELECT
  ID,
  Name,
  -- подзапрос 1а - получаем ID сотрудника
  (SELECT MAX(ID) FROM Employees emp WHERE emp.DepartmentID=dep.ID) LastEmpID_var1,
  -- подзапрос 1б - получаем ID сотрудника
  (SELECT TOP 1 ID FROM Employees emp WHERE emp.DepartmentID=dep.ID ORDER BY ID DESC) LastEmpID_
var2,
  -- подзапрос 2 - получаем имя сотрудника
  (SELECT TOP 1 Name FROM Employees emp WHERE emp.DepartmentID=dep.ID ORDER BY ID DESC) LastEmpN
ame
FROM Departments dep
```

Не хорошо правда ведь? Т.к. каждый подзапрос выполнится по 4 раза, итого 12 выполнится 12 подзапросов.

Поэтому, я бы рекомендовал прибегать к использованию подзапросов с параметрами в самую последнюю очередь, т.к. когда вы не можете выразить запрос при помощи простых операций соединений, т.к. использование подзапросов в таких случаях может сильно понизить скорость выполнения запроса. Потому что подзапрос с параметром будет выполняться для каждого переданного ему параметра.

Подзапросы с конструкцией APPLY

В MS SQL для последнего примера:

```
SELECT
  ID,
  Name,
  -- подзапрос 1 - получаем ID сотрудника
  (SELECT TOP 1 ID FROM Employees emp WHERE emp.DepartmentID=dep.ID ORDER BY ID DESC) LastEmpID,
  -- подзапрос 2 - получаем имя сотрудника
  (SELECT TOP 1 Name FROM Employees emp WHERE emp.DepartmentID=dep.ID ORDER BY ID DESC) LastEmpN
ame
FROM Departments dep
```

можно применить конструкцию APPLY, которая имеет 2 формы – CROSS APPLY и OUTER APPLY.

Конструкция APPLY позволяет избавиться от множества подзапросов, как в данном примере, когда требуется получить и ID и Name последнего принятого сотрудника для каждого отдела:

```
SELECT
  ID,
  Name,
  empInfo.LastEmpID,
  empInfo.LastEmpName
FROM Departments dep
CROSS APPLY
  (
    SELECT TOP 1 ID LastEmpID,Name LastEmpName
    FROM Employees emp
    WHERE emp.DepartmentID=dep.ID
    ORDER BY emp.ID DESC
  ) empInfo
```

ID	Name	LastEmpID	LastEmpName
1	Администрация	1000	Иванов И.И.
2	Бухгалтерия	1002	Сидоров С.С.
3	ИТ	1004	Николаев Н.Н.

Здесь подзапрос блока CROSS APPLY выполнится для каждого значения строки из таблицы Departments. Если подзапрос строки не вернет, то данный отдел исключается из результирующего списка.

Если требуется, чтобы были возвращены все строки таблицы Departments, то используйте следующую форму этого оператора OUTER APPLY:

```
SELECT
  ID,
  Name,
  empInfo.LastEmpID,
  empInfo.LastEmpName
FROM Departments dep
OUTER APPLY
  (
    SELECT TOP 1 ID LastEmpID,Name LastEmpName
    FROM Employees emp
    WHERE emp.DepartmentID=dep.ID
    ORDER BY emp.ID DESC
  ) empInfo
```

ID	Name	LastEmpID	LastEmpName
1	Администрация	1000	Иванов И.И.
2	Бухгалтерия	1002	Сидоров С.С.
3	ИТ	1004	Николаев Н.Н.
4	Маркетинг и реклама	NULL	NULL
5	Логистика	NULL	NULL

В общем, достаточно полезный оператор, который в некоторых ситуациях сильно упрощающий запрос. Данный подзапрос так же сработает для каждой строки результирующего набора, т.е. для каждого переданного параметра, но сработает он намного эффективнее, чем в случае использования множества подзапросов. С прочими деталями в случае применения APPLY, например, для случая, когда подзапрос возвращает несколько строк, я думаю вы сможете разобраться самостоятельно. Ладно, раз уж заговорил об этом, то приведу небольшой пример для самостоятельного разбора:

```
SELECT dep.ID,dep.Name,pos.PositionID,pos.PositionName
FROM Departments dep
CROSS APPLY
  (
    SELECT ID PositionID,Name PositionName
    FROM Positions
  ) pos
```

Использование подзапросов в блоке WHERE

Для примера получим отделы, в которых числится более двух сотрудников:

```
SELECT *
FROM Departments dep
WHERE (SELECT COUNT(*) FROM Employees emp WHERE emp.DepartmentID=dep.ID)>2
```

Так как здесь мы используем оператор сравнения, то соответственно подзапрос должен возвращать максимум одну строку и одно значение, т.е. так же когда подзапрос используется и в блоке SELECT.

Конструкции EXISTS и NOT EXISTS

Позволяют проверить есть ли соответствующие условию записи в подзапросе:

```
-- отделы в которых есть хотя бы один сотрудник
SELECT *
FROM Departments dep
WHERE EXISTS(SELECT * FROM Employees emp WHERE emp.DepartmentID=dep.ID)



-- отделы в которых нет ни одного сотрудника
SELECT *
FROM Departments dep
WHERE NOT EXISTS(SELECT * FROM Employees emp WHERE emp.DepartmentID=dep.ID)
```

Здесь все просто – EXISTS возвращает True, если подзапрос возвращает хотя бы одну строку, и False, если подзапрос не возвращает строк. NOT EXISTS – инверсия результата.

Конструкция IN и NOT IN с подзапросом

До этого мы рассматривали IN с перечислением значений. Так же можно использовать его с подзапросом, который возвращает перечень этих значений:

```
-- отделы где есть сотрудники
SELECT *
FROM Departments
WHERE ID IN(SELECT DISTINCT DepartmentID FROM Employees WHERE DepartmentID IS NOT NULL)



-- отделы где нет сотрудников
SELECT *
FROM Departments
WHERE ID NOT IN(SELECT DISTINCT DepartmentID FROM Employees WHERE DepartmentID IS NOT NULL)
```

Обратите внимание, что я исключил NULL значение используя условие (DepartmentID IS NOT NULL) в подзапросе. NULL значения в данном случае так же опасны – смотрите об этом в описании конструкции IN во второй части.

Операции группового сравнения ALL и ANY

Данные операторы, очень хитрые и их использовать нужно очень аккуратно. Вообще в своей практике я их применяю достаточно редко, предпочитая использовать в условиях операторы IN или EXISTS.

Операторы ALL и ANY используются в тех случаях, когда необходимо проверить условие на соответствие, с каждым значением которое вернул подзапрос. Они, как и оператор EXISTS работают только с подзапросами.

Для примера в каждом отделе выберем сотрудника, у которого ЗП больше ЗП всех сотрудников работающих в этом же отделе. Для этой цели применим ALL:

```
SELECT ID,Name,DepartmentID,Salary
FROM Employees e1
WHERE e1.Salary>ALL(
                        SELECT e2.Salary
                        FROM Employees e2
                        WHERE e2.DepartmentID=e1.DepartmentID -- учесть только сотрудников этого ж
e отдела
                          AND e2.ID<>e1.ID -- чтобы исключить сравнение со своей же ЗП
                          AND e2.Salary IS NOT NULL -- исключить NULL значения
                    )
```

ID	Name	DepartmentID	Salary
1000	Иванов И.И.	1	5000
1002	Сидоров С.С.	2	2500
1003	Андреев А.А.	3	2000
1005	Александров А.А.	NULL	2000

Здесь происходит проверка на то, что e1.Salary больше значений e2.Salary, которые вернул подзапрос.

Как думаете, почему здесь вернулись даже те сотрудники, для которых подзапрос не вернул ни одной строки? А потому что логика такая – нет записей, не с чем проверять, а значит я и так больше всех.))) Вот такая хитрость здесь скрыта.

Для большего понимания, давайте посмотрим, как можно здесь оператор ALL заменить оператором NOT EXISTS:

```
SELECT ID,Name,DepartmentID,Salary
FROM Employees e1
WHERE NOT EXISTS(
                    SELECT *
                    FROM Employees e2
                    WHERE e2.DepartmentID=e1.DepartmentID -- учесть только сотрудников этого же
отдела
                      AND e2.Salary>e1.Salary -- выбираем только ЗП больше ЗП этого сотрудника
                 )
```

Т.е. мы тут выразили то же самое только другими словами «Верни сотрудников для которых нет сотрудников из того же отдела с большей ЗП чем у него».

Здесь становится понятно почему ALL возвращает истинное значение в том случае если подзапрос не возвращает данных.

Так же обратите внимание, что для ALL важно исключить NULL-значения из подзапроса, иначе результат проверки на каждое значение может оказаться неопределенным. Сравнивайте в этом случае логика ALL логикой при использовании AND, т.е. выражение (Salary>1000 AND Salary>1500 AND Salary>NULL) вернет NULL.

А вот с ANY (он же SOME) будет по-другому:

```
SELECT ID,Name,DepartmentID,Salary
FROM Employees e1
WHERE e1.Salary>ANY( -- ANY = SOME
                      SELECT e2.Salary
                      FROM Employees e2
                      WHERE e2.DepartmentID=e1.DepartmentID -- учесть только сотрудников этого ж
е отдела
                        AND e2.ID<>e1.ID -- чтобы исключить сравнение со своей же ЗП
                    )
```

ID	Name	DepartmentID	Salary
1003	Андреев А.А.	3	2000

С оператором ANY важно, чтобы подзапрос вернул записи, с которыми можно сравнить на любое выполнение условия. Т.к. во всех отделах сидят только по одному сотруднику, кроме ИТ-отдела, то вернулся только Андреев А.А., чью ЗП удалось сравнить с ЗП других сотрудников этого же отдела. Т.е. мы вытащили здесь тех, чья ЗП больше любой ЗП сотрудника из этого же отдела.

Давайте для большего понимания, попробуем выразить здесь ANY при помощи EXISTS:

```
SELECT ID,Name,DepartmentID,Salary
FROM Employees e1
WHERE EXISTS(
                SELECT *
                FROM Employees e2
                WHERE e2.DepartmentID=e1.DepartmentID -- учесть только сотрудников этого же отдела
                  AND e2.Salary<e1.Salary -- проверяем есть ли сотрудники с меньшей ЗП чем у данно
го сотрудника
             )
```

Смысл здесь стал «есть ли хоть какой-то сотрудник из этого отделу у которого ЗП ниже ЗП данного сотрудника».

В таком виде становится понятно, почему ANY возвращает ложное значение, если подзапрос не возвращает данных.

Наличие NULL-значений в подзапросе здесь не так опасно, т.к. мы сравниваем на любое значение. Сравнивайте в этом случае логика ANY логикой при использовании OR, т.е. выражение (Salary>1000 OR Salary>1500 OR Salary>NULL) может вернуть истинное значение если выполнится хотя бы одно условие.

Если ANY используется для сравнения на равенство, то его можно представить при помощи IN:

```
SELECT *
FROM Departments
WHERE ID=ANY(SELECT DISTINCT DepartmentID FROM Employees)
```

Здесь мы возвращаем все отделы, в которых есть сотрудники. Соответственно это будет эквивалентно:

```
SELECT *
FROM Departments
WHERE ID IN(SELECT DISTINCT DepartmentID FROM Employees)
```

Как видите ALL и ANY можно выразить при помощи других операторов. Но в некоторых случаях их использование может сделать запрос более читабельным, поэтому для полноты картины их тоже стоит знать и применять в подходящих для этого случаях. Т.е. при написании запроса вы можете написать его так как вас попросили «выбери сотрудника у которого ЗП больше всех»:

```
SELECT *
FROM Employees e1
WHERE e1.Salary>ALL(SELECT e2.Salary FROM Employees e2 WHERE e2.ID<>e1.ID AND e2.Salary IS NOT N
ULL)
```

не заменяя смысл на аналогичный «выбери сотрудников для которых нет сотрудников с ЗП больше чем у него»:

```
SELECT *
FROM Employees e1
WHERE NOT EXISTS(SELECT * FROM Employees e2 WHERE e2.Salary>e1.Salary)
```

Это еще раз показывает, что язык SQL изначально задумывался как язык для обычных пользователей, чтобы они могли выражать свои мысли разными способами.

Еще пара слов про подзапросы

Подзапросы можно так же использовать во многих других блоках, например, в блоке HAVING, осуществлять проверку в конструкциях CASE. В общем, тут уже насколько хватит вашей фантазии.

Но я бы рекомендовал в первую очередь всегда пытаться решить задачу стандартными конструкциями оператора SELECT, и, если этого не получается, прибегать к помощи подзапросов.

Поэтому в данном учебнике я уделил целых три части на рассмотрение базовых конструкций и только один раздел выделил подзапросам. Я считаю, что нельзя начинать объяснение SELECT с подзапросов, т.к. зная, что есть подзапросы, но не владея базовыми конструкциями, новички могут нагородить такие трехэтажные конструкции (подзапросы в подзапросах-подзапросах), которые даже профессионалам потом бывает трудно разобрать. Но если разобраться, то в некоторых случаях, зная основы все эти трехэтажные конструкции можно было бы выразить при помощи одного запроса с использованием, например, соединений и группировок.

Я не говорю, что подзапросы – это плохо, т.к. иногда при помощи них конкретную задачу можно решить более изящно. Я говорю здесь о том, что в первую очередь нужно научится уверенно пользоваться базовыми конструкциями, ведь подзапросы тоже строятся их них. А так все конструкции хороши, когда они применяются по назначению.

Заключение

Вот мы и закончили разбираться со всеми основными конструкциями оператора SELECT. Если посчитать, то их не так много, но уверенное владение каждой из них и умение пользоваться ими сообща, дает вам огромные возможности для получения практически любой информации хранящейся в РБД.

Данный материал был создан, опираясь на свой собственный практический опыт работы с языком SQL, которому уже более 10 лет, в разных СУБД (начиная с СУБД Paradox). В данном учебнике, я постарался максимально простым образом объяснить суть всех основных конструкций языка SQL служащих для выборки данных. Я старался объяснять так, чтобы данный учебник был понятен широкому кругу людей, а не только ИТ-специалистам. Надеюсь, что это у меня получилось и что данный материал поможет вам сделать первые шаги или же поможет понять какую-нибудь отдельную конструкцию, которая возможно вам не давалась ранее. В любом случае, спасибо всем, кто уделил свое время на ознакомление с этим материалом.

В следующей части я уже в общих чертах расскажу о операторах модификации данных. В общих чертах, так как эта информация, как и знание DDL, нужна не всем (в основном ИТ-специалистам) – большинство людей изучают SQL именно в целях научиться делать выборку данных при помощи оператора SELECT. Думаю, следующая часть будет заключительной. Все знания, полученные до этого момента, нам так же пригодятся в следующей части, т.к. для правильного написания сложных конструкций по модификации данных, нужно уверено пользоваться конструкциями оператора SELECT. Например, перед тем как удалить или изменить группу строк таблицы, мы должны правильно выбрать эти данные. Поэтому следующая часть так же будет содержать конструкции SELECT и думаю будет интересна тем людям, которые изучают SQL именно из-за оператора выборки SELECT.

Для уверенного написания запросов, мало понимать теорию, нужно еще много практиковаться. Для этой цели я бы рекомендовал вам известный сайт под названием «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. К тому же вы можете потягаться в решении рейтинговых задач и в итоге получить сертификат, доказывающий именно ваши практические навыки, а не только знание теории.

После того как вы уверенно научитесь использовать базовые конструкции, я бы посоветовал вам в следующую очередь самостоятельно изучить:

- 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();
- Изучить конструкции позволяющие вычислить под итоги: GROUP BY ROLLUP(…), GROUP BY GROUPING SETS(…), … А так же вспомогательные функции используемые для этих целей: GROUPING_ID() и GROUPING();
- Конструкции PIVOT, UNPIVOT.

Краткая информация по всему этому дана в пятой части в «Приложение 1 – бонус по оператору SELECT» и «Приложение 2 – OVER и аналитические функции». Дополнительную информацию по всему этому вы легко сможете найти в интернет, в той же библиотеке MSDN.

Удачи в изучении.

Часть пятая

В данной части мы рассмотрим

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

- INSERT – вставка новых данных
- UPDATE – обновление данных
- DELETE – удаление данных
- SELECT … INTO … – сохранить результат запроса в новой таблице
- MERGE – слияние данных
- Использование конструкции OUTPUT
- TRUNCATE TABLE – DDL-операция для быстрой очистки таблицы

В самом конце вас ждут «Приложение 1 – бонус по оператору SELECT» и «Приложение 2 – OVER и аналитические функции», в которых будут показаны некоторые расширенные конструкции:

- PIVOT
- UNPIVOT
- GROUP BY ROLLUP
- GROUP BY GROUPING SETS
- использование приложения OVER

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, как я и говорил, будет больше обзорная. Здесь я буду описывать только те основные формы операторов модификации данных, которыми я сам регулярно пользуюсь. Поэтому на полноту изложения рассчитывать не стоит, здесь будут показан только необходимый минимум, который новички могут использовать как направление для более глубокого изучения. За более подробной информацией по каждому оператору обращайтесь в MSDN. Хотя кому-то возможно и в таком объеме информации будет вполне достаточно.

Т.к. прямая модификация информации в РБД требует от человека большой ответственности, а также потому что пользователи обычно модифицируют информацию БД посредством разных АРМ, и не имеют полного доступа к БД, то данная часть больше посвящается начинающим ИТ-специалистам, и я буду здесь очень краток. Но конечно, 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. В любом случае можете почитать и попытаться проделать примеры в ознакомительных целях, к тому же в итоге вы сможете получить более детальную базу, на которой можно будет отработать те или иные конструкции оператора SELECT.

Проведем изменения в структуре нашей БД

Давайте проведем небольшое обновление структуры и данных таблицы Employees:

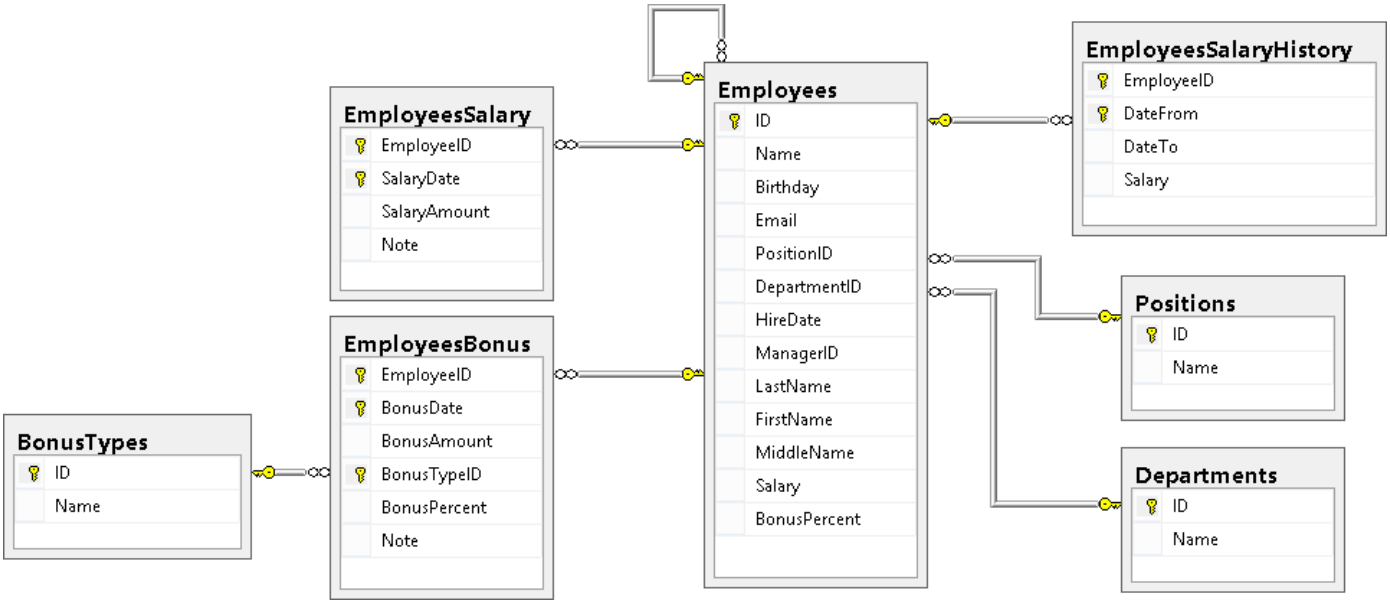
```
-- информацию по ЗП решено хранить до 2-х знаков после запятой
ALTER TABLE Employees ALTER COLUMN Salary numeric(20,2)
-- информацию по процентам решено хранить только в целых числах
ALTER TABLE Employees ALTER COLUMN BonusPercent tinyint
```

А также для демонстрационных целей расширим схему нашей БД, а за одно повторим DDL. Назначения таблиц и полей указаны в комментариях:

```
-- история изменений ЗП у сотрудников
CREATE TABLE EmployeesSalaryHistory(
  EmployeeID int NOT NULL, -- ссылка на ID сотрудника
  DateFrom date NOT NULL,  -- с какой даты
  DateTo date,             -- по какую дату. Содержит NULL если это последняя установленная ЗП.
  Salary numeric(20,2) NOT NULL, -- сумма ЗП за этот период
CONSTRAINT PK_EmployeesSalaryHistory PRIMARY KEY(EmployeeID,DateFrom),
CONSTRAINT FK_EmployeesSalaryHistory_EmployeeID FOREIGN KEY(EmployeeID) REFERENCES Employees(ID)
)
GO

-- таблица для хранения истории начислений по ЗП
CREATE TABLE EmployeesSalary(
  EmployeeID int NOT NULL,
  SalaryDate date NOT NULL, -- дата начисления
  SalaryAmount numeric(20,2) NOT NULL, -- сумма начисления
  Note nvarchar(50), -- примечание
-- здесь сумма ЗП может фиксироваться по человеку 1 раз в день
CONSTRAINT PK_EmployeesSalary PRIMARY KEY(EmployeeID,SalaryDate),
-- связь с таблицей Employees
CONSTRAINT FK_EmployeesSalary_EmployeeID FOREIGN KEY(EmployeeID) REFERENCES Employees(ID)
)
GO

-- справочник по типам бонусов
CREATE TABLE BonusTypes(
  ID int IDENTITY(1,1) NOT NULL,
  Name nvarchar(30) NOT NULL,
CONSTRAINT PK_BonusTypes PRIMARY KEY(ID)
)
GO
```

```
-- таблица для хранения истории начислений бонусов
CREATE TABLE EmployeesBonus(
  EmployeeID int NOT NULL,
  BonusDate date NOT NULL, -- дата начисления
  BonusAmount numeric(20,2) NOT NULL, -- сумма начисления
  BonusTypeID int NOT NULL,
  BonusPercent tinyint,
  Note nvarchar(50), -- примечание
-- бонус одного типа может фиксироваться по человеку 1 раз в день
CONSTRAINT PK_EmployeesBonus PRIMARY KEY(EmployeeID,BonusDate,BonusTypeID),
-- связь с таблицей Employees и BonusTypes
CONSTRAINT FK_EmployeesBonus_EmployeeID FOREIGN KEY(EmployeeID) REFERENCES Employees(ID),
CONSTRAINT FK_EmployeesBonus_BonusTypeID FOREIGN KEY(BonusTypeID) REFERENCES BonusTypes(ID)
)
GO
```

Вот такой полигон мы должны были получить в итоге:



Кстати, потом этот полигон (когда он будет наполнен данными) вы и можете использовать для того чтобы опробовать на нем разнообразные запросы – здесь можно опробовать и разнообразные JOIN-соединения, и UNION-объединения, и группировки с агрегированием 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, что мне очень нравится и я этим всегда пользуюсь.

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)
```

INSERT – форма 1. Переходим сразу к практике

Наполним таблицу EmployeesSalaryHistory предоставленными нам данными:

```
INSERT EmployeesSalaryHistory(EmployeeID,DateFrom,DateTo,Salary)
VALUES
  -- Иванов И.И.
  (1000,'20131101','20140531',4000),
  (1000,'20140601','20141230',4500),
  (1000,'20150101',NULL,5000),
  -- Петров П.П.
  (1001,'20131101','20140630',1300),
  (1001,'20140701','20140930',1400),
  (1001,'20141001',NULL,1500),
  -- Сидоров С.С.
  (1002,'20140101',NULL,2500),
  -- Андреев А.А.
  (1003,'20140601',NULL,2000),
  -- Николаев Н.Н.
  (1004,'20140701','20150131',1400),
  (1004,'20150201','20150131',1500),
  -- Александров А.А.
  (1005,'20150101',NULL,2000)
```

Таким образом мы вставили в таблицу EmployeesSalaryHistory 11 новых записей.

```
SELECT *
FROM EmployeesSalaryHistory
```

EmployeeID	DateFrom	DateTo	Salary
1000	2013-11-01	2014-05-31	4000.00
1000	2014-06-01	2014-12-30	4500.00
1000	2015-01-01	NULL	5000.00
1001	2013-11-01	2014-06-30	1300.00
1001	2014-07-01	2014-09-30	1400.00
1001	2014-10-01	NULL	1500.00
1002	2014-01-01	NULL	2500.00
1003	2014-06-01	NULL	2000.00
1004	2014-07-01	2015-01-31	1400.00
1004	2015-02-01	2015-01-31	1500.00
1005	2015-01-01	NULL	2000.00

Хоть мы в этом случае могли и не указывать перечень полей, т.к. мы вставляем данные всех полей и в таком же виде, как они перечислены в таблице, т.е. мы могли бы написать:

```
INSERT EmployeesSalaryHistory
VALUES
  -- Иванов И.И.
  (1000,'20131101','20140531',4000),
  (1000,'20140601','20141230',4500),
  (1000,'20150101',NULL,5000),
  …
```

Но я бы не рекомендовал использовать такой подход, особенно если данный запрос будет использоваться регулярно, например, вызываясь из какого-то АРМ. Опять же это чревато тем, что структура таблицы может изменяться, в нее могут быть добавлены новые поля, или же последовательность полей может быть изменена, что еще опасней, т.к. это может привести к появлению логических ошибок во вставленных данных. Поэтому лучше лишний раз не полениться и перечислить явно все поля, в которые вы хотите вставить значение.

Несколько заметок про INSERT:

- Порядок перечисления полей не имеет значения, вы можете написать и (EmployeeID,DateFrom,DateTo,Salary) и (DateFrom,DateTo, EmployeeID,Salary). 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 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.

В предыдущих частях мы периодически использовали опцию IDENTITY_INSERT. Давайте и здесь воспользуемся данной опцией для создания строк в таблице BonusTypes, у которой поле ID указано с опцией IDENTITY:

```
-- даем разрешение на добавление/изменение IDENTITY значения
SET IDENTITY_INSERT BonusTypes ON

INSERT BonusTypes(ID,Name)VALUES
(1,N'Ежемесячный'),
(2,N'Годовой'),
(3,N'Индивидуальный')

-- запрещаем добавление/изменение IDENTITY значения
SET IDENTITY_INSERT BonusTypes OFF
```

Давайте вставим информацию по начислению сотрудникам ЗП, любезно предоставленную нам бухгалтером:

```
-- Иванов И.И.
INSERT EmployeesSalary(EmployeeID,SalaryDate,Sa
laryAmount,Note)VALUES
(1000,'20131130',4000,NULL),
(1000,'20131231',4000,NULL),
(1000,'20140115',2000,N'Аванс'),
(1000,'20140131',2000,NULL),
(1000,'20140228',4000,NULL),
(1000,'20140331',4000,NULL),
(1000,'20140430',4000,NULL),
(1000,'20140531',4000,NULL),
(1000,'20140630',6500,N'ЗП + Аванс 2500 за 2014
.07'),
(1000,'20140731',2000,NULL),
(1000,'20140831',4500,NULL),
(1000,'20140930',4500,NULL),
(1000,'20141031',4500,NULL),
(1000,'20141130',4500,NULL),
(1000,'20141230',4500,NULL),
(1000,'20150131',5000,NULL),
(1000,'20150228',5000,NULL),
(1000,'20150331',5000,NULL)
```

```
-- Петров П.П.
INSERT EmployeesSalary(EmployeeID,SalaryDate,Sa
laryAmount,Note)VALUES
(1001,'20131130',2600,N'ЗП + ЗП за 2013.12'),
(1001,'20140228',2600,N'За 2 месяца 2014.01, 20
14.02'),
(1001,'20140331',1300,NULL),
(1001,'20140430',1300,NULL),
(1001,'20140510',300,N'Аванс'),
(1001,'20140520',500,N'Аванс'),
(1001,'20140531',500,NULL),
(1001,'20140630',1300,NULL),
(1001,'20140731',1400,NULL),
(1001,'20140831',1400,NULL),
(1001,'20140930',1400,NULL),
(1001,'20141031',1500,NULL),
(1001,'20141130',1500,NULL),
(1001,'20141230',3000,N'ЗП + ЗП за 2015.01'),
(1001,'20150228',1500,NULL),
(1001,'20150331',1500,NULL)
```

```
-- Сидоров С.С.
INSERT EmployeesSalary(EmployeeID,SalaryDate,Sa
laryAmount,Note)VALUES
(1002,'20140131',2500,NULL),
(1002,'20140228',2500,NULL),
(1002,'20140331',2500,NULL),
(1002,'20140430',2500,NULL),
(1002,'20140531',2500,NULL),
(1002,'20140630',2500,NULL),
(1002,'20140731',2500,NULL),
(1002,'20140831',2500,NULL),
(1002,'20140930',2500,NULL),
(1002,'20141031',2500,NULL),
(1002,'20141130',2500,NULL),
(1002,'20141230',2500,NULL),
(1002,'20150131',2500,NULL),
(1002,'20150228',2500,NULL),
(1002,'20150331',2500,NULL)
```

```
-- Андреев А.А.
INSERT EmployeesSalary(EmployeeID,SalaryDate,Sa
laryAmount,Note)VALUES
(1003,'20140630',2000,NULL),
(1003,'20140731',2000,NULL),
(1003,'20140831',2000,NULL),
(1003,'20140930',2000,NULL),
(1003,'20141031',2000,NULL),
(1003,'20141130',2000,NULL),
(1003,'20141230',2000,NULL),
(1003,'20150131',2000,NULL),
(1003,'20150228',2000,NULL),
(1003,'20150331',2000,NULL)
```

```
-- Николаев Н.Н.
INSERT EmployeesSalary(EmployeeID,SalaryDate,Sa
laryAmount,Note)VALUES
(1004,'20140731',1400,NULL),
(1004,'20140831',1400,NULL),
(1004,'20140930',1400,NULL),
(1004,'20141031',1400,NULL),
(1004,'20141130',1400,NULL),
(1004,'20141212',400,N'Аванс'),
(1004,'20141230',1400,NULL),
(1004,'20150131',1400,NULL),
(1004,'20150228',1500,NULL),
(1004,'20150331',1500,NULL)
```

```
-- Александров А.А.
INSERT EmployeesSalary(EmployeeID,SalaryDate,Sa
laryAmount,Note)VALUES
(1005,'20150131',2000,NULL),
(1005,'20150228',2000,NULL),
(1005,'20150331',2000,NULL)
```

Думаю, приводить содержимое таблицы уже нет смысла.

INSERT – форма 2

Данная форма позволяет вставить в таблицу данные полученные запросом.

Для демонстрации наполним таблицу с начислениями бонусов одним большим запросом:

```
INSERT EmployeesBonus(EmployeeID,BonusDate,BonusAmount,BonusTypeID,BonusPercent)
-- расчет ежемесячных бонусов
SELECT hist.EmployeeID,bdate.BonusDate,hist.Salary/100*emp.BonusPercent,1 BonusTypeID,emp.BonusP
ercent
FROM EmployeesSalaryHistory hist
JOIN
  (
    VALUES -- весь период работы компании - последние дни месяцев
      ('20131130'),
      ('20131231'),
      ('20140131'),
      ('20140228'),
      ('20140331'),
      ('20140430'),
      ('20140531'),
      ('20140630'),
      ('20140731'),
      ('20140831'),
      ('20140930'),
      ('20141031'),
      ('20141130'),
      ('20141230'),
      ('20150131'),
      ('20150228'),
      ('20150331')
  ) bdate(BonusDate)
ON bdate.BonusDate BETWEEN hist.DateFrom AND ISNULL(hist.DateTo,'20991231')
JOIN Employees emp ON hist.EmployeeID=emp.ID
WHERE emp.BonusPercent IS NOT NULL AND emp.BonusPercent>0
  AND NOT EXISTS( -- исключаем сотрудников, которым по какой-то причине не дали бонус в указанны
й период
               SELECT *
               FROM
                 (
                   VALUES
                     (1001,'20140115'),
                     (1001,'20140430'),
                     (1001,'20141031'),
                     (1001,'20141130'),
                     (1001,'20150228')
                 ) exclude(EmployeeID,BonusDate)
               WHERE exclude.EmployeeID=emp.ID
                 AND exclude.BonusDate=bdate.BonusDate
             )

UNION ALL
```

```
-- годовой бонус за 2014 год - всем кто проработал больше полугода
SELECT
  hist.EmployeeID,
  '20141231' BonusDate,
  hist.Salary/100*
  CASE DepartmentID
    WHEN 2 THEN 10 -- 10% от ЗП выдать Бухгалтерам
    WHEN 3 THEN 15 -- 15% от ЗП выдать ИТ-шникам
    ELSE 5 -- всем остальным по 5%
  END BonusAmount,
  2 BonusTypeID,
  CASE DepartmentID
    WHEN 2 THEN 10 -- 10% от ЗП выдать Бухгалтерам
    WHEN 3 THEN 15 -- 15% от ЗП выдать ИТ-шникам
    ELSE 5 -- всем остальным по 5%
  END BonusPercent
FROM EmployeesSalaryHistory hist
JOIN Employees emp ON hist.EmployeeID=emp.ID
WHERE CAST('20141231' AS date) BETWEEN hist.DateFrom AND ISNULL(hist.DateTo,'20991231')
  AND emp.HireDate<='20140601'

UNION ALL

-- индивидуальные бонусы
SELECT EmployeeID,BonusDate,BonusAmount,3 BonusTypeID,NULL BonusPercent
FROM
  (
    VALUES
      (1001,'20140930',300),
      (1002,'20140331',500),
      (1002,'20140630',500),
      (1002,'20140930',500),
      (1002,'20141230',500),
      (1002,'20150331',500),
      (1004,'20140831',200)
  ) indiv(EmployeeID,BonusDate,BonusAmount)
```

В таблицу EmployeesBonus должно было вставиться 50 записей.

Результат каждого запроса объединенных конструкциями UNION ALL вы можете проанализировать самостоятельно. Если вы хорошо изучили базовые конструкции, то вам должно быть все понятно, кроме возможно конструкции с VALUES (конструктор табличных значений), которая появилась с MS SQL 2008.

Пара слов про конструкцию VALUES

```
SELECT EmployeeID,BonusDate,BonusAmount,3 BonusTypeID,NULL BonusPercent
FROM
  (
    VALUES
      (1001,'20140930',300),
      (1002,'20140331',500),
      (1002,'20140630',500),
      (1002,'20140930',500),
      (1002,'20141230',500),
      (1002,'20150331',500),
      (1004,'20140831',200)
  ) indiv(EmployeeID,BonusDate,BonusAmount)
```

В случае необходимости, данную конструкцию можно заменить, аналогичным запросом, написанным через UNION ALL:

```
SELECT 1001 EmployeeID,'20140930' BonusDate,300 BonusAmount,3 BonusTypeID,NULL BonusPercent
UNION ALL
SELECT 1002,'20140331',500,3,NULL
UNION ALL
SELECT 1002,'20140630',500,3,NULL
UNION ALL
SELECT 1002,'20140930',500,3,NULL
UNION ALL
SELECT 1002,'20141230',500,3,NULL
UNION ALL
SELECT 1002,'20150331',500,3,NULL
UNION ALL
SELECT 1004,'20140831',200,3,NULL
```

Думаю, комментарии излишни и вам не составит большого труда разобраться с этим самостоятельно.

Так что, идем дальше.

INSERT + CTE-выражения

Совместно с INSERT можно применять CTE выражения. Для примера перепишем тот же запрос перенеся все подзапросы в блок WITH.

Для начала полностью очистим таблицу EmployeesBonus при помощи операции TRUNCATE TABLE:

```
TRUNCATE TABLE EmployeesBonus
```

Теперь перепишем запрос вынеся запросы в блок WITH:

```
WITH cteBonusType1 AS(
   -- расчет ежемесячных бонусов
   SELECT hist.EmployeeID,bdate.BonusDate,hist.Salary/100*emp.BonusPercent BonusAmount,1 BonusTyp
eID,emp.BonusPercent
  FROM EmployeesSalaryHistory hist
  JOIN
    (
      VALUES -- весь период работы компании - последние дни месяцев
        ('20131130'),
        ('20131231'),
        ('20140131'),
        ('20140228'),
        ('20140331'),
        ('20140430'),
        ('20140531'),
        ('20140630'),
        ('20140731'),
        ('20140831'),
        ('20140930'),
        ('20141031'),
        ('20141130'),
        ('20141230'),
        ('20150131'),
        ('20150228'),
        ('20150331')
    ) bdate(BonusDate)
  ON bdate.BonusDate BETWEEN hist.DateFrom AND ISNULL(hist.DateTo,'20991231')
  JOIN Employees emp ON hist.EmployeeID=emp.ID
  WHERE emp.BonusPercent IS NOT NULL AND emp.BonusPercent>0
    AND NOT EXISTS( -- исключаем сотрудников, которым по какой-то причине не дали бонус в указан
ный период
                  SELECT *
                  FROM
                    (
                      VALUES
                        (1001,'20140115'),
                        (1001,'20140430'),
                        (1001,'20141031'),
                        (1001,'20141130'),
                        (1001,'20150228')
                    ) exclude(EmployeeID,BonusDate)
                  WHERE exclude.EmployeeID=emp.ID
                    AND exclude.BonusDate=bdate.BonusDate
                )
),
```

```
cteBonusType2 AS(
  -- годовой бонус за 2014 год - всем кто проработал больше полугода
  SELECT
    hist.EmployeeID,
    '20141231' BonusDate,
    hist.Salary/100*
    CASE DepartmentID
      WHEN 2 THEN 10 -- 10% от ЗП выдать Бухгалтерам
      WHEN 3 THEN 15 -- 15% от ЗП выдать ИТ-шникам
      ELSE 5 -- всем остальным по 5%
    END BonusAmount,
    2 BonusTypeID,
    CASE DepartmentID
      WHEN 2 THEN 10 -- 10% от ЗП выдать Бухгалтерам
      WHEN 3 THEN 15 -- 15% от ЗП выдать ИТ-шникам
      ELSE 5 -- всем остальным по 5%
    END BonusPercent
  FROM EmployeesSalaryHistory hist
  JOIN Employees emp ON hist.EmployeeID=emp.ID
  WHERE CAST('20141231' AS date) BETWEEN hist.DateFrom AND ISNULL(hist.DateTo,'20991231')
    AND emp.HireDate<='20140601'
),
cteBonusType3 AS(
  -- индивидуальные бонусы
  SELECT EmployeeID,BonusDate,BonusAmount,3 BonusTypeID,NULL BonusPercent
  FROM
    (
      VALUES
        (1001,'20140930',300),
        (1002,'20140331',500),
        (1002,'20140630',500),
        (1002,'20140930',500),
        (1002,'20141230',500),
        (1002,'20150331',500),
        (1004,'20140831',200)
    ) indiv(EmployeeID,BonusDate,BonusAmount)
)

INSERT EmployeesBonus(EmployeeID,BonusDate,BonusAmount,BonusTypeID,BonusPercent)
SELECT *
FROM cteBonusType1
UNION ALL
SELECT *
FROM cteBonusType2
UNION ALL
SELECT *
FROM cteBonusType3
```

Как видим вынос больших подзапросов в блок WITH упростил основной запрос – сделал его более понятным.

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 на мой взгляд удобнее.

Давайте при помощи первой формы приведем даты приема каждого сотрудника в порядок. Выполним 6 отдельных операций UPDATE:

```
-- приведем даты приема в порядок
UPDATE Employees SET HireDate='20131101' WHERE ID=1000

UPDATE Employees SET HireDate='20131101' WHERE ID=1001

UPDATE Employees SET HireDate='20140101' WHERE ID=1002

UPDATE Employees SET HireDate='20140601' WHERE ID=1003

UPDATE Employees SET HireDate='20140701' WHERE ID=1004

-- а здесь еще почистим поле FirstName
UPDATE Employees SET HireDate='20150101',FirstName=NULL WHERE ID=1005
```

Вторую форму, где применялся псевдоним, мы уже тоже успели использовать в первой части, когда обновляли поля PositionID и DepartmentID, на значения возвращаемые подзапросами:

```
UPDATE e
SET
  PositionID=(SELECT ID FROM Positions WHERE Name=e.Position),
  DepartmentID=(SELECT ID FROM Departments WHERE Name=e.Department)
FROM Employees e
```

Сейчас конечно данный и следующий запрос не сработают, т.к. поля Position и Department мы удалили из таблицы Employees. Вот так можно было бы представить этот запрос при помощи операций соединений:

```
UPDATE e
SET
  PositionID=p.ID,
  DepartmentID=d.ID
FROM Employees e
LEFT JOIN Positions p ON p.Name=e.Position
LEFT JOIN Departments d ON d.Name=e.Department
```

Надеюсь суть обновления здесь понятна, тут обновляться будут строки таблицы Employees.

Сначала вы можете сделать выборку, чтобы посмотреть какие данные будут обновлены и на какие значения:

```
SELECT
  e.ID,
  e.PositionID,e.DepartmentID, -- старые значения
  e.Position,e.Department,
  p.ID,d.ID, -- новые значения
  p.Name,d.Name
FROM Employees e
LEFT JOIN Positions p ON p.Name=e.Position
LEFT JOIN Departments d ON d.Name=e.Department
```

А потом переписать это в UPDATE:

```
UPDATE e
SET
  PositionID=p.ID,
  DepartmentID=d.ID
FROM Employees e
LEFT JOIN Positions p ON p.Name=e.Position
LEFT JOIN Departments d ON d.Name=e.Department
```

Эх, не могу я так, все-таки давайте посмотрим, как это работает наглядно.

Для этого опять вспомним DDL и временно создадим поля Position и Department в таблице Employees:

```
ALTER TABLE Employees ADD Position nvarchar(30),Department nvarchar(30)
```

Зальем в них данные, предварительно посмотрев при помощи SELECT, что получится:

```
SELECT
  e.ID,
  e.Position,
  p.Name NewPosition,
  e.Department,
  d.Name NewDepartment
FROM Employees e
LEFT JOIN Positions p ON p.ID=e.PositionID
LEFT JOIN Departments d ON d.ID=e.DepartmentID
```

Теперь перепишем и выполним обновление:

```
UPDATE e
SET
  e.Position=p.Name,
  e.Department=d.Name
FROM Employees e
LEFT JOIN Positions p ON p.ID=e.PositionID
LEFT JOIN Departments d ON d.ID=e.DepartmentID
```

Посмотрите, что получилось (должны были появиться значения в 2-х полях – Position и Department, находящиеся в конце таблицы):

```
SELECT *
FROM Employees
```

Теперь и этот запрос:

```
UPDATE e
SET
  PositionID=(SELECT ID FROM Positions WHERE Name=e.Position),
  DepartmentID=(SELECT ID FROM Departments WHERE Name=e.Department)
FROM Employees e
```

И этот:

```
UPDATE e
SET
  PositionID=p.ID,
  DepartmentID=d.ID
FROM Employees e
LEFT JOIN Positions p ON p.Name=e.Position
LEFT JOIN Departments d ON d.Name=e.Department
```

Отработают успешно.

Не забудьте только предварительно посмотреть (это очень полезная привычка):

```
SELECT
  e.ID,
  e.PositionID,e.DepartmentID, -- старые значения
  e.Position,e.Department,
  p.ID,d.ID, -- новые значения
  p.Name,d.Name
FROM Employees e
LEFT JOIN Positions p ON p.Name=e.Position
LEFT JOIN Departments d ON d.Name=e.Department
```

И конечно же можете использовать здесь условие WHERE:

```
UPDATE e
SET
  PositionID=p.ID,
  DepartmentID=d.ID
FROM Employees e
LEFT JOIN Positions p ON p.Name=e.Position
LEFT JOIN Departments d ON d.Name=e.Department
WHERE d.ID=3 -- обновить только данные по ИТ-отделу
```

Все, убедились, что все работает. Если хотите, то можете снова удалить поля Position и Department.

Вторую форму можно так же использовать с подзапросом:

```
UPDATE e
SET
  HireDate='20131101',
  MiddleName=N'Иванович'
FROM (SELECT MiddleName,HireDate FROM Employees WHERE ID=1000) e
```

В данном случае подзапрос должен возвращать в явном виде строки таблицы Employees, которые будут обновлены. В подзапросе нельзя использовать группировки или предложения DISTINCT, т.к. в этом случае мы не получим явных строк таблицы Employees. И соответственно все обновляемые поля должны содержаться в предложении SELECT, если конечно вы не указали «SELECT *».

Так же с UPDATE вы можете использовать CTE-выражения. Для примера перенесем наш подзапрос в блок WITH:

```
WITH cteEmp AS(
   SELECT MiddleName,HireDate FROM Employees WHERE ID=1000
)
UPDATE cteEmp
SET
  HireDate='20131101',
  MiddleName=N'Иванович'
```

Идем дальше.

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 на мой взгляд удобнее.

Для примера при помощи первого варианта:

```
-- удалим неиспользуемые должности Логист и Кладовщик
DELETE Positions WHERE ID IN(6,7)
```

При помощи второго варианта удалим остальные неиспользуемые должности. В целях демонстрации запрос намеренно излишне усложнен. Сначала посмотрим, что именно удалиться (всегда старайтесь делать проверку, а то ненароком можно удалить лишнее, а то и всю информацию из таблицы):

```
SELECT pos.*
FROM
  (
    SELECT DISTINCT PositionID
    FROM Employees
  ) emp
RIGHT JOIN Positions pos ON pos.ID=emp.PositionID
WHERE emp.PositionID IS NULL -- нет среди должностей указанных в Employees
```

Убедились, что все нормально. Переписываем запрос на DELETE:

```
DELETE pos -- удалить из этой таблицы
FROM
  (
    SELECT DISTINCT PositionID
    FROM Employees
  ) emp
RIGHT JOIN Positions pos ON pos.ID=emp.PositionID
WHERE emp.PositionID IS NULL -- нет среди должностей указанных в Employees
```

В качестве таблицы Positions может выступать и подзапрос, главное, чтобы он однозначно возвращал строки, которые будут удаляться. Давайте добавим для демонстрации в таблицу Positions мусора:

```
INSERT Positions(Name) VALUES('Test 1'),('Test 2')
```

Теперь для демонстрации используем вместо таблицы Positions, подзапрос, в котором отбираются только определенные строки из таблицы Positions:

```
DELETE pos -- удалить из этой таблицы
FROM
  (
    SELECT DISTINCT PositionID
    FROM Employees
  ) emp
RIGHT JOIN
  (
    SELECT ID
    FROM Positions
    WHERE ID>4 -- отбираем должности по условию
  ) pos
ON pos.ID=emp.PositionID
WHERE emp.PositionID IS NULL -- нет среди должностей указанных в Employees
```

Так же мы можем использовать CTE выражения (подзапросы, оформленные в блоке WITH). Давайте снова добавим для демонстрации в таблицу Positions мусора:

```
INSERT Positions(Name) VALUES('Test 1'),('Test 2')
```

И посмотрим на тот же запрос с CTE-выражением:

```
WITH ctePositionc AS(
  SELECT ID
  FROM Positions
  WHERE ID>4 -- отбираем должности по условию
)
DELETE pos -- удалить из этой таблицы
FROM
  (
    SELECT DISTINCT PositionID
    FROM Employees
  ) emp
RIGHT JOIN ctePositionc pos ON pos.ID=emp.PositionID
WHERE emp.PositionID IS NULL -- нет среди должностей указанных в Employees
```

Заключение по INSERT, UPDATE и DELETE

Вот по сути и все, что я хотел рассказать вам про основные операторы модификации данных – INSERT, UPDATE и DELETE.

Я считаю, что данные операторы очень легко понять интуитивно, когда умеешь пользоваться конструкциями оператора SELECT. Поэтому рассказ о операторе SELECT растянулся на 3 части, а рассказ о операторах модификации был написан в такой беглой форме.

И как вы увидели, с операторами модификации тоже полет фантазии не ограничен. Но все же старайтесь писать, как можно проще и понятней, обязательно предварительно проверяя, какие записи будут обработаны при помощи SELECT, т.к. обычно модификация данных, это очень большая ответственность.

В дополнение скажу, что в диалекте MS SQL со всеми операциями модификации можно использовать предложение TOP (INSERT TOP …, UPDATE TOP …, DELETE TOP …), но мне пока ни разу не приходилось прибегать к такой форме, т.к. здесь непонятно какие именно TOP записей будут обработаны.

Если уж нужно обработать TOP записей, то я, наверное, лучше воспользуюсь указанием опции TOP в подзапросе и применю в нем нужным мне образом ORDER BY, чтобы явно знать какие именно TOP записей будут обработаны. Для примера снова добавим мусора:

```
INSERT Positions(Name) VALUES('Test 1'),('Test 2')
```

И удалим 2 последние записи:

```
DELETE emp
FROM
  (
    SELECT TOP 2 * -- 2. берем только 2 верхние записи
    FROM Positions
    ORDER BY ID DESC -- 1. сортируем по убыванию
  ) emp
```

Я здесь привожу примеры больше в целях демонстрации возможностей языка SQL. В реальных запросах старайтесь выражать свои намерения очень точно, дабы выполнение вашего запроса не привело к порче данных. Еще раз скажу – будьте очень внимательны, и не ленитесь делать предварительные проверки.

SELECT … INTO … – сохранить результат запроса в новой таблице

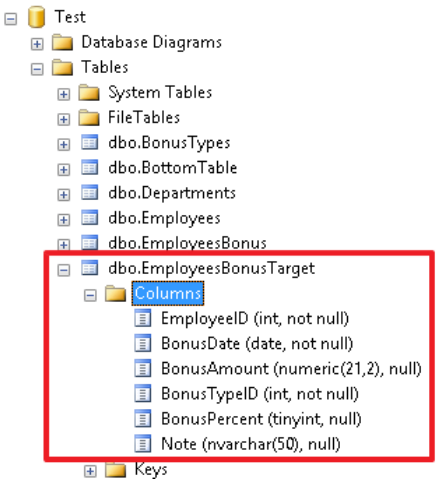
Данная конструкция позволяет сохранить результат выборки в новой таблице. Она представляет из себя что-то промежуточное между DDL и DML.

Типы колонок созданной таблицы будут определены на основании типов колонок набора, полученного запросом SELECT. Если в выборке присутствуют результаты выражений, то им должны быть заданы псевдонимы, которые будут служить в роли имен колонок.

Давайте отберем следующие данные и сохраним их в таблице EmployeesBonusTarget (перед FROM просто пишем INTO и указываем имя новой таблицы):

```
SELECT
  bonus.EmployeeID,
  bonus.BonusDate,
  bonus.BonusAmount-bonus.BonusAmount BonusAmount, -- обнулим значения
  bonus.BonusTypeID,
  bonus.BonusPercent,
  bonus.Note
INTO EmployeesBonusTarget -- сохраним результат в новой таблице EmployeesBonusTarget
FROM EmployeesBonus bonus
JOIN Employees emp ON bonus.EmployeeID=emp.ID
WHERE emp.DepartmentID=3
```

Можете обновить список таблиц в инспекторе объектов и увидеть новую таблицу EmployeesBonusTarget:



На самом деле я специально создал таблицу EmployeesBonusTarget, я ее буду использовать для демонстрации оператора MERGE.

Еще пара слов про конструкцию SELECT … INTO …

Данную конструкцию иногда удобно применять при формировании очень сложных отчетов, которые требуют выборки из множества таблиц. В этом случае данные обычно сохраняют во временных таблицах (#). Т.е. предварительно при помощи запросов, мы сбрасываем данные во временные таблицы, а затем используем эти временные таблицы в других запросах, которые формируют окончательный результат:

```
SELECT
  ID,
  CONCAT(LastName,' ',FirstName,' ',MiddleName) FullName, -- используем псевдоним FullName
  Salary,
  BonusPercent,
  Salary/100*ISNULL(BonusPercent,0) Bonus -- используем псевдоним Bonus
INTO #EmployeesBonus -- сохранить результат во временной таблице
FROM Employees



SELECT …
FROM #EmployeesBonus b
JOIN …
```

Иногда данную конструкцию удобно использовать, чтобы сделать полную копию всех данных текущей таблицы:

```
SELECT *
INTO EmployeesBackup
FROM Employees
```

Это можно использовать, например, для подстраховки, перед тем как вносить серьезные изменения в структуру таблицы Employees. Вы можете сохранить копию либо всех данных таблицы, либо только тех данных, которых коснется модификация. Т.е. если что-то пойдет не так, вы сможете восстановить данные таблицы Employees с этой копии. В таких случаях конечно хорошо сделать предварительный бэкап БД на текущий момент, но это бывает не всегда возможно из-за огромных объемов, срочности и т.п.

Чтобы не засорять основную базу, можно создать новую БД и сделать копию таблицы туда:

```
CREATE DATABASE TestTemp
GO

SELECT *
INTO TestTemp.dbo.EmployeesBackup -- используем префикс ИмяБаза.Схема.
FROM Employees
```

Для того чтобы увидеть новую БД TestTemp, соответственно, обновите в инспекторе объектов список баз данных, в ней и уже можете найти данную таблицу.

На заметку.
В БД Oracle так же есть конструкция для сохранения результата запроса в новую таблицу, выглядит она следующим образом:

```
CREATE TABLE EMPLOYEES_BACK -- сохранить результат в новой таблице с именем EMPLOYEES_BACK
AS
SELECT *
FROM EMPLOYEES
```

MERGE – слияние данных

Данный оператор хорошо подходит для синхронизации данных 2-х таблиц. Такая задача может понадобится при интеграции разных систем, когда данные передаются порциями из одной системы в другую.

В нашем случае, допустим, что стоит задача синхронизации таблицы EmployeesBonusTarget с таблицей EmployeesBonus.

Давайте добавим в таблицу EmployeesBonusTarget какого-нибудь мусора:

```
INSERT EmployeesBonusTarget(EmployeeID,BonusDate,BonusAmount,BonusTypeID,Note)VALUES
(9999,'20150101',9999.99,0,N'это мусор'),
(9999,'20150201',9999.99,0,N'это мусор'),
(9999,'20150301',9999.99,0,N'это мусор'),
(9999,'20150401',9999.99,0,N'это мусор'),
(9999,'20150501',9999.99,0,N'это мусор'),
(9999,'20150601',9999.99,0,N'это мусор')
```

Теперь при помощи оператора MERGE добьемся того, чтобы данные в таблице EmployeesBonusTarget стали такими же, как и в EmployeesBonus, т.е. сделаем синхронизацию данных.

Синхронизацию мы будем осуществлять на основании сопоставления данных входящих в первичный ключ таблицы EmployeesBonus (EmployeeID, BonusDate, BonusTypeID):

1. Если для строки таблицы EmployeesBonusTarget соответствия по ключу не нашлось, то нужно сделать удаление таких строк из EmployeesBonusTarget
2. Если соответствие нашлось, то нужно обновить строки EmployeesBonusTarget данными соответствующей строки из EmployeesBonus
3. Если строка есть в EmployeesBonus, но ее нет в EmployeesBonusTarget, то ее нужно добавить в EmployeesBonusTarget

Сделаем реализацию всей этой логики при помощи инструкции MERGE:

```
MERGE EmployeesBonusTarget trg -- таблица приемник
USING EmployeesBonus src -- таблица источник
ON trg.EmployeeID=src.EmployeeID AND trg.BonusDate=src.BonusDate AND trg.BonusTypeID=src.BonusTy
peID -- условие слияния

-- 1. Строка есть в trg но нет сопоставления со строкой из src
WHEN NOT MATCHED BY SOURCE THEN
   DELETE

-- 2. Есть сопоставление строки trg со строкой из источника src
WHEN MATCHED THEN
  UPDATE SET
     trg.BonusAmount=src.BonusAmount,
     trg.BonusPercent=src.BonusPercent,
     trg.Note=src.Note

-- 3. Строка не найдена в trg, но есть в src
WHEN NOT MATCHED BY TARGET THEN -- предложение BY TARGET можно отпускать, т.е. NOT MATCHED = NOT
MATCHED BY TARGET
  INSERT(EmployeeID,BonusDate,BonusAmount,BonusTypeID,BonusPercent,Note)
  VALUES(src.EmployeeID,src.BonusDate,src.BonusAmount,src.BonusTypeID,src.BonusPercent,src.Note)
;
```

Данная конструкция должна оканчиваться «;».

После выполнения запроса сравните 2 таблицы, их данные должны быть одинаковыми.

Конструкция MERGE чем-то напоминает условный оператор CASE, она так же содержит блоки WHEN, при выполнении условий которых происходит то или иное действие, в данном случае удаление (DELETE), обновление (UPDATE) или добавление (INSERT). Модификация данных производится в таблице приемнике.

В качестве источника может выступать запрос. Например, синхронизируем только данные по отделу 3 и для примера исключаем блок «NOT MATCHED BY SOURCE», чтобы данные не удались в случае не совпадения:

```
MERGE EmployeesBonusTarget trg -- таблица приемник
USING
        (
            SELECT bonus.*
            FROM EmployeesBonus bonus
            JOIN Employees emp ON bonus.EmployeeID=emp.ID
            WHERE emp.DepartmentID=3
        ) src -- источник
ON trg.EmployeeID=src.EmployeeID AND trg.BonusDate=src.BonusDate AND trg.BonusTypeID=src.BonusTy
peID -- условие слияния

-- 2. Есть сопоставление строки trg со строкой из источника src
WHEN MATCHED THEN
  UPDATE SET
    trg.BonusAmount=src.BonusAmount,
    trg.BonusPercent=src.BonusPercent,
    trg.Note=src.Note

-- 3. Строка не найдена в trg, но есть в src
WHEN NOT MATCHED BY TARGET THEN -- предложение BY TARGET можно отпускать, т.е. NOT MATCHED = NOT
MATCHED BY TARGET
  INSERT(EmployeeID,BonusDate,BonusAmount,BonusTypeID,BonusPercent,Note)
  VALUES(src.EmployeeID,src.BonusDate,src.BonusAmount,src.BonusTypeID,src.BonusPercent,src.Note)
;
```

Я показал работу конструкции MERGE в самом общем ее виде. При помощи нее можно реализовывать более разнообразные схемы для слияния данных, например, можно включать в блоки WHEN дополнительные условия (WHEN MATCHED AND … THEN). Это очень мощная конструкция, позволяющая в подходящих случаях сократить объем кода и совместить в рамках одного оператора функционал всех трех операторов – INSERT, UPDATE и DELETE.

И естественно с конструкцией MERGE так же можно применять CTE-выражения:

```
WITH cteBonus AS(
   SELECT bonus.*
  FROM EmployeesBonus bonus
  JOIN Employees emp ON bonus.EmployeeID=emp.ID
  WHERE emp.DepartmentID=3
)
MERGE EmployeesBonusTarget trg -- таблица приемник
USING cteBonus src -- источник
ON trg.EmployeeID=src.EmployeeID AND trg.BonusDate=src.BonusDate AND trg.BonusTypeID=src.BonusTy
peID -- условие слияния

-- 2. Есть сопоставление строки trg со строкой из источника src
WHEN MATCHED THEN
  UPDATE SET
    trg.BonusAmount=src.BonusAmount,
    trg.BonusPercent=src.BonusPercent,
    trg.Note=src.Note

-- 3. Строка не найдена в trg, но есть в src
WHEN NOT MATCHED BY TARGET THEN -- предложение BY TARGET можно отпускать, т.е. NOT MATCHED = NOT
MATCHED BY TARGET
  INSERT(EmployeeID,BonusDate,BonusAmount,BonusTypeID,BonusPercent,Note)
  VALUES(src.EmployeeID,src.BonusDate,src.BonusAmount,src.BonusTypeID,src.BonusPercent,src.Note)
;
```

В общем, я постарался вам задать направление, более подробнее, в случае необходимости, изучайте уже самостоятельно.

Использование конструкции OUTPUT

Конструкция OUTPUT дает возможность получить информацию по строкам, которые были добавлены, удалены или изменены в результате выполнения DML команд INSERT, DELETE, UPDATE и MERGE. Данная конструкция, представляет расширение для операций модификации данных и в каждой СУБД может быть реализовано по-своему, либо вообще отсутствовать.

Конструкция OUTPUT имеет 2 основные формы:

1. **OUTPUT перечень_выражений** – используется для возврата результата в виде набора
2. **OUTPUT перечень_выражений INTO принимающая_таблица(список_полей)** – используется для вставки результата в указанную таблицу

Рассмотрим первую форму

Добавим в таблицу Positions новые записи:

```
INSERT Positions(Name)
OUTPUT inserted.*
VALUES
  (N'Test 1'),
  (N'Test 2'),
  (N'Test 3')
```

После выполнения данной операции, записи будут вставлены в таблицу Positions и в добавок мы увидим информацию по добавленным строкам на экране.

Ключевое слово «inserted» дает нам доступ к значениям добавленных строк. В данном случае использование «inserted.*» вернет нам информацию по всем полям, которые есть в таблице Positions (ID и Name).

Так же после OUTPUT вы можете явно указать возвращаемый на экран перечень полей посредством «inserted.имя_поля», также вы можете использовать разные выражения:

```
INSERT Positions(Name)
OUTPUT inserted.ID,inserted.Name,'I'
VALUES
  (N'Test 4'),
  (N'Test 5'),
  (N'Test 6')
```

При использовании DML команды DELETE, доступ к значениям измененных строк получается при помощи ключевого слова «deleted»:

```
DELETE Positions
OUTPUT deleted.ID,deleted.Name,'D'
WHERE Name LIKE N'Test%'
```

При использовании DML команды UPDATE, мы можем использовать ключевое слово:

- deleted – для того, чтобы получить доступ к значениям строки, которые были до обновления (старые значения)
- inserted – для того, чтобы получить новые значения строки

Продемонстрируем на таблице Employees:

```
UPDATE Employees
SET
  LastName=N'Александров',
  FirstName=N'Александр'
OUTPUT
  deleted.ID,
  deleted.LastName [Старая Фамилия],
  deleted.FirstName [Старое Имя],
  inserted.ID,
  inserted.LastName [Новая Фамилия],
  inserted.FirstName [Новое Имя]
WHERE ID=1005
```

ID	Старая Фамилия	Старое Имя	ID	Новая Фамилия	Новое Имя
1005	NULL	NULL	1005	Александров	Александр

В случае MERGE мы можем так же использовать «inserted» и «deleted» для доступа к значениям обработанных строк.

Давайте для примера создадим таблицу PositionsTarget, на которой после будет показан пример с MERGE:

```
SELECT
  CAST(ID AS int) ID, -- чтобы поле создалось без опции IDENTITY
  Name+'-old' Name -- изменим название
INTO PositionsTarget
FROM Positions
WHERE ID=2 -- вставим только одну должность
```

Добавим в PositionsTarget мусора:

```
INSERT PositionsTarget(ID,Name)VALUES
(100,N'Qwert'),
(101,N'Asdf')
```

Выполним команду MERGE с конструкцией OUTPUT:

```
MERGE PositionsTarget trg -- таблица приемник
USING Positions src -- таблица источник
ON trg.ID=src.ID -- условие слияния

-- 1. Строка есть в trg но нет сопоставления со строкой из src
WHEN NOT MATCHED BY SOURCE THEN
   DELETE

-- 2. Есть сопоставление строки trg со строкой из источника src
WHEN MATCHED THEN
  UPDATE SET
    trg.Name=src.Name

-- 3. Строка не найдена в trg, но есть в src
WHEN NOT MATCHED BY TARGET THEN -- предложение BY TARGET можно отпускать, т.е. NOT MATCHED = NOT
MATCHED BY TARGET
  INSERT(ID,Name)
  VALUES(src.ID,src.Name)

OUTPUT
  deleted.ID Old_ID,
  deleted.Name Old_Name,
  inserted.ID New_ID,
  inserted.Name New_Name,
  CASE
    WHEN deleted.ID IS NOT NULL AND inserted.ID IS NOT NULL THEN 'U'
    WHEN deleted.ID IS NOT NULL THEN 'D'
    WHEN inserted.ID IS NOT NULL THEN 'I'
  END OperType;
```

Old_ID	Old_Name	New_ID	New_Name	OperType
NULL	NULL	1	Бухгалтер	I
2	Директор-old	2	Директор	U
NULL	NULL	3	Программист	I
NULL	NULL	4	Старший программист	I
100	Qwert	NULL	NULL	D
101	Asdf	NULL	NULL	D

Думаю, назначение первой формы понятно – сделать модификацию и получить результат в виде набора, который можно вернуть пользователю.

Рассмотрим вторую форму

У конструкции OUTPUT, есть и более важное предназначение – она позволяет не только получить, но и зафиксировать (OUTPUT … INTO …) информацию о том, что уже произошло по факту, то есть после выполнения операции модификации. Она может оказаться полезна в случае логированния произошедших действий. В некоторых случаях, ее можно использовать как хорошую альтернативу тригерам (для прозрачности действий).

Давайте создадим демонстрационную таблицу, для логирования изменений по таблице Positions:

```
CREATE TABLE PositionsLog(
  LogID int IDENTITY(1,1) NOT NULL CONSTRAINT PK_PositionsLog PRIMARY KEY,
  ID int,
  Old_Name nvarchar(30),
  New_Name nvarchar(30),
  LogType char(1) NOT NULL,
  LogDateTime datetime NOT NULL DEFAULT SYSDATETIME()
)
```

А теперь сделаем при помощи конструкции (OUTPUT … INTO …) запись в эту таблицу:

```
-- добавление
INSERT Positions(Name)
OUTPUT inserted.ID,inserted.Name,'I' INTO PositionsLog(ID,New_Name,LogType)
VALUES
  (N'Test 1'),
  (N'Test 2')

-- обновление
UPDATE Positions
SET
  Name+=' - new' -- обратите внимание на синтаксис "+=", аналогично Name=Name+' - new'
OUTPUT
  deleted.ID,
  deleted.Name,
  inserted.Name,
  'U'
INTO PositionsLog(ID,Old_Name,New_Name,LogType)
WHERE Name LIKE N'Test%'

-- удаление
DELETE Positions
OUTPUT deleted.ID,deleted.Name,'D' INTO PositionsLog(ID,Old_Name,LogType)
WHERE Name LIKE N'Test%'
```

Посмотрите, что получилось:

```
SELECT * FROM PositionsLog
```

TRUNCATE TABLE – DDL-операция для быстрой очистки таблицы

Данный оператор является DDL-операцией и служит для быстрой очистки таблицы – удаляет все строки из нее. За более детальными подробностями обращайтесь в MSDN.

Некоторые вырезки из MSDN. *TRUNCATE TABLE – удаляет все строки в таблице, не записывая в журнал удаление отдельных строк. Инструкция TRUNCATE TABLE похожа на инструкцию DELETE без предложения WHERE, однако TRUNCATE TABLE выполняется быстрее и требует меньших ресурсов системы и журналов транзакций.*

Если таблица содержит столбец идентификаторов (столбец с опцией IDENTITY), счетчик этого столбца сбрасывается до начального значения, определенного для этого столбца. Если начальное значение не задано, используется значение по умолчанию, равное 1. Чтобы сохранить столбец идентификаторов, используйте инструкцию DELETE.

Инструкцию TRUNCATE TABLE нельзя использовать если на таблицу ссылается ограничение FOREIGN KEY. Таблицу, имеющую внешний ключ, ссылающийся сам на себя, можно усечь.

Пример:

```
TRUNCATE TABLE EmployeesBonusTarget
```

Заключение по операциям модификации данных

Здесь я наверно повторю, все что писал ранее.

Старайтесь в первую очередь написать запрос на модификацию как можно проще, в первую очередь попытайтесь выразить свое намерение при помощи базовых конструкций и в последнюю очередь прибегайте к использованию подзапросов.

Прежде чем запустить запрос на модификацию данных по условию, убедитесь, что он выбирает именно необходимые записи, а не больше и не меньше. Для этой цели воспользуйтесь операцией SELECT.

Не забывайте перед очень серьезными изменениями делать резервные копии, хотя бы той информации, которая будет подвергнута модификации, это можно сделать при помощи SELECT … INTO …

Помните, что модификация данных это очень серьезно.

Приложение 1 – бонус по оператору SELECT

Подумав, я решил дописать этот раздел для тех, кто дошел до конца.

В данном разделе я дам примеры с использованием некоторых расширенных конструкций:

- PIVOT
- UNPIVOT
- GROUP BY ROLLUP
- GROUP BY GROUPING SETS

Попробуйте разобрать каждый из следующих примеров самостоятельно, анализируя результаты выполнения запросов. Обращайте внимание на комментарии, которые я указал в текстах запросов, некоторые важные вещи указаны в них.

Получение сводных отчетов при помощи GROUP BY+CASE и конструкции PIVOT

Для начала давайте посмотрим, как можно создать сводный отчет при помощи конструкции GROUP BY и CASE-условий. Можно сказать, это классический способ создания сводных отчетов:

```
-- получение сводной таблицы при помощи GROUP BY
SELECT
  EmployeeID,
  SUM(CASE WHEN MONTH(BonusDate)=1 THEN BonusAmount END) BonusAmount1,
  SUM(CASE WHEN MONTH(BonusDate)=2 THEN BonusAmount END) BonusAmount2,
  SUM(CASE WHEN MONTH(BonusDate)=3 THEN BonusAmount END) BonusAmount3,
  SUM(CASE WHEN MONTH(BonusDate)=4 THEN BonusAmount END) BonusAmount4,
  SUM(CASE WHEN MONTH(BonusDate)=5 THEN BonusAmount END) BonusAmount5,
  SUM(CASE WHEN MONTH(BonusDate)=6 THEN BonusAmount END) BonusAmount6,
  SUM(CASE WHEN MONTH(BonusDate)=7 THEN BonusAmount END) BonusAmount7,
  SUM(CASE WHEN MONTH(BonusDate)=8 THEN BonusAmount END) BonusAmount8,
  SUM(CASE WHEN MONTH(BonusDate)=9 THEN BonusAmount END) BonusAmount9,
  SUM(CASE WHEN MONTH(BonusDate)=10 THEN BonusAmount END) BonusAmount10,
  SUM(CASE WHEN MONTH(BonusDate)=11 THEN BonusAmount END) BonusAmount11,
  SUM(CASE WHEN MONTH(BonusDate)=12 THEN BonusAmount END) BonusAmount12,
  SUM(BonusAmount) TotalBonusAmount
FROM EmployeesBonus
WHERE BonusDate BETWEEN '20140101' AND '20141231' -- отберем данные за 2014 год
GROUP BY EmployeeID
```

Теперь рассмотрим, как получить эти же данные при помощи конструкции PIVOT:

```
-- получение сводной таблицы при помощи PIVOT
SELECT
  EmployeeID,
  [1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[12],
  ISNULL([1],0)+ISNULL([2],0)+ISNULL([3],0)+ISNULL([4],0)+
  ISNULL([5],0)+ISNULL([6],0)+ISNULL([7],0)+ISNULL([8],0)+
  ISNULL([9],0)+ISNULL([10],0)+ISNULL([11],0)+ISNULL([12],0) TotalBonusAmount
FROM
  (
    /*
      в данном подзапросе мы отберем только необходимые для свода данные:
        - поля BonusMonth и BonusAmount будут задействованы в конструкции PIVOT
        - прочие поля, в данном случае это только EmployeeID, будут использованны для группировки данных
    */
    SELECT
      EmployeeID,
      MONTH(BonusDate) BonusMonth,
      BonusAmount
    FROM EmployeesBonus
    WHERE BonusDate BETWEEN '20140101' AND '20141231'
  ) q
  PIVOT(SUM(BonusAmount) FOR BonusMonth IN([1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[12]))
p
```

В конструкции PIVOT кроме SUM, как вы думаю догадались, можно использовать и другие агрегатные функции (COUNT, AVG, MIN, MAX, …).

Конструкция UNPIVOT

Давайте теперь рассмотрим, как работает конструкция UNPIVOT. Для демонстрации сбросим сводный результат в таблицу DemoPivotTable:

```
SELECT
  EmployeeID,
  SUM(CASE WHEN MONTH(BonusDate)=1 THEN BonusAmount END) BonusAmount1,
  SUM(CASE WHEN MONTH(BonusDate)=2 THEN BonusAmount END) BonusAmount2,
  SUM(CASE WHEN MONTH(BonusDate)=3 THEN BonusAmount END) BonusAmount3,
  SUM(CASE WHEN MONTH(BonusDate)=4 THEN BonusAmount END) BonusAmount4,
  SUM(CASE WHEN MONTH(BonusDate)=5 THEN BonusAmount END) BonusAmount5,
  SUM(CASE WHEN MONTH(BonusDate)=6 THEN BonusAmount END) BonusAmount6,
  SUM(CASE WHEN MONTH(BonusDate)=7 THEN BonusAmount END) BonusAmount7,
  SUM(CASE WHEN MONTH(BonusDate)=8 THEN BonusAmount END) BonusAmount8,
  SUM(CASE WHEN MONTH(BonusDate)=9 THEN BonusAmount END) BonusAmount9,
  SUM(CASE WHEN MONTH(BonusDate)=10 THEN BonusAmount END) BonusAmount10,
  SUM(CASE WHEN MONTH(BonusDate)=11 THEN BonusAmount END) BonusAmount11,
  SUM(CASE WHEN MONTH(BonusDate)=12 THEN BonusAmount END) BonusAmount12,
  SUM(BonusAmount) TotalBonusAmount
INTO DemoPivotTable -- сбросим сводный результат в таблицу
FROM EmployeesBonus
WHERE BonusDate BETWEEN '20140101' AND '20141231'
GROUP BY EmployeeID
```

Первым делом посмотрите, как у нас выглядят данные в данной таблице:

```
SELECT *
FROM DemoPivotTable
```

Теперь применим к данной таблице конструкцию UNPIVOT:

```
-- демонстрация UNPIVOT
SELECT
  *,
  CAST(REPLACE(ColumnLabel,'BonusAmount','') AS int) BonusMonth
FROM DemoPivotTable
  UNPIVOT(BonusAmount FOR ColumnLabel IN(BonusAmount1,BonusAmount2,BonusAmount3,BonusAmount4,
                                         BonusAmount5,BonusAmount6,BonusAmount7,BonusAmount8,
                                         BonusAmount9,BonusAmount10,BonusAmount11,BonusAmount12)
) u
```

Обратите внимание, что NULL значения не войдут в результат.

Как вы наверно догадались, на месте таблицы может стоять и подзапрос с заданным для него псевдонимом.

GROUP BY ROLLUP и GROUP BY GROUPING SETS

Данные конструкции позволяют подбить промежуточные итоги по строкам.

Пример первый:

```
-- GROUP BY ROLLUP и функция GROUPING
SELECT
  --GROUPING(YEAR(bonus.BonusDate)) g1,
  --GROUPING(bonus.EmployeeID) g2,
  --GROUPING(emp.Name) g3,

  CASE
    WHEN GROUPING(YEAR(bonus.BonusDate))=1 THEN 'Общий итог'
    WHEN GROUPING(bonus.EmployeeID)=1 THEN 'Итого за '+CAST(YEAR(bonus.BonusDate) AS varchar(4))
+' год'
  END RowTitle,

  emp.Name,
  SUM(CASE WHEN DATEPART(QUARTER,bonus.BonusDate)=1 THEN bonus.BonusAmount END) BonusAmountQ1,
  SUM(CASE WHEN DATEPART(QUARTER,bonus.BonusDate)=2 THEN bonus.BonusAmount END) BonusAmountQ2,
  SUM(CASE WHEN DATEPART(QUARTER,bonus.BonusDate)=3 THEN bonus.BonusAmount END) BonusAmountQ3,
  SUM(CASE WHEN DATEPART(QUARTER,bonus.BonusDate)=4 THEN bonus.BonusAmount END) BonusAmountQ4,
  SUM(bonus.BonusAmount) TotalBonusAmount
FROM EmployeesBonus bonus
JOIN Employees emp ON bonus.EmployeeID=emp.ID
GROUP BY ROLLUP(YEAR(bonus.BonusDate),bonus.EmployeeID,emp.Name)
-- исключаем ненужный итог обрабатывая GROUPING
HAVING NOT(GROUPING(YEAR(bonus.BonusDate))=0 AND GROUPING(bonus.EmployeeID)=0 AND GROUPING(emp.N
ame)=1)
```

Чтобы понять, как работает функции GROUPING, раскомментируйте поля g1, g2 и g3, чтобы они попали в результирующий набор, а также закомментируйте предложение HAVING.

Пример второй:

```
-- GROUP BY ROLLUP и функция GROUPING_ID
SELECT
  /*
    GROUPING_ID (a, b, c) input = GROUPING(a) + GROUPING(b) + GROUPING(c)
    бинарное 001 = десятичное 1
    бинарное 011 = десятичное 3
    бинарное 111 = десятичное 7
  */

  --GROUPING_ID(YEAR(bonus.BonusDate),bonus.EmployeeID,emp.Name) gID,

  CASE GROUPING_ID(YEAR(bonus.BonusDate),bonus.EmployeeID,emp.Name)
    WHEN 7 THEN 'Общий итог'
    WHEN 3 THEN 'Итого за '+CAST(YEAR(bonus.BonusDate) AS varchar(4))+' год'
  END RowTitle,

  emp.Name,
  SUM(CASE WHEN DATEPART(QUARTER,bonus.BonusDate)=1 THEN bonus.BonusAmount END) BonusAmountQ1,
  SUM(CASE WHEN DATEPART(QUARTER,bonus.BonusDate)=2 THEN bonus.BonusAmount END) BonusAmountQ2,
  SUM(CASE WHEN DATEPART(QUARTER,bonus.BonusDate)=3 THEN bonus.BonusAmount END) BonusAmountQ3,
  SUM(CASE WHEN DATEPART(QUARTER,bonus.BonusDate)=4 THEN bonus.BonusAmount END) BonusAmountQ4,
  SUM(bonus.BonusAmount) TotalBonusAmount
FROM EmployeesBonus bonus
JOIN Employees emp ON bonus.EmployeeID=emp.ID
GROUP BY ROLLUP(YEAR(bonus.BonusDate),bonus.EmployeeID,emp.Name)
-- исключаем ненужный итог обрабатывая GROUPING_ID
HAVING GROUPING_ID(YEAR(bonus.BonusDate),bonus.EmployeeID,emp.Name)<>1
```

Здесь для понимания, можете так же раскомментировать поле gID и закомментировать предложение HAVING.

Пример третий:

```
-- GROUP BY GROUPING SETS и функция GROUPING_ID
SELECT
  /*
    GROUPING_ID (a, b, c) input = GROUPING(a) + GROUPING(b) + GROUPING(c)
    бинарное 001 = десятичное 1
    бинарное 011 = десятичное 3
    бинарное 111 = десятичное 7
  */

  --GROUPING_ID(YEAR(bonus.BonusDate),bonus.EmployeeID,emp.Name) gID,

  CASE GROUPING_ID(YEAR(bonus.BonusDate),bonus.EmployeeID,emp.Name)
    WHEN 7 THEN 'Общий итог'
    WHEN 3 THEN 'Итого за '+CAST(YEAR(bonus.BonusDate) AS varchar(4))+' год'
  END RowTitle,

  emp.Name,
  SUM(CASE WHEN DATEPART(QUARTER,bonus.BonusDate)=1 THEN bonus.BonusAmount END) BonusAmountQ1,
  SUM(CASE WHEN DATEPART(QUARTER,bonus.BonusDate)=2 THEN bonus.BonusAmount END) BonusAmountQ2,
  SUM(CASE WHEN DATEPART(QUARTER,bonus.BonusDate)=3 THEN bonus.BonusAmount END) BonusAmountQ3,
  SUM(CASE WHEN DATEPART(QUARTER,bonus.BonusDate)=4 THEN bonus.BonusAmount END) BonusAmountQ4,
  SUM(bonus.BonusAmount) TotalBonusAmount
FROM EmployeesBonus bonus
JOIN Employees emp ON bonus.EmployeeID=emp.ID
GROUP BY GROUPING SETS(
                  (YEAR(bonus.BonusDate),bonus.EmployeeID,emp.Name), -- Имя сотрудника
                  (YEAR(bonus.BonusDate)), -- Сумма по годам
                  () -- Общий итог
               )
```

При помощи GROUPING SET можно явно указать какие именно итоги нам нужны, поэтому здесь можно обойтись без предложения HAVING.

Т.е. можно сказать, что GROUP BY ROLLUP частный случай GROUP BY GROUPING SETS, когда делается вывод всех итогов.

Пример использования FULL JOIN

Здесь для примера выведем для каждого сотрудника сводные данные по начислениям бонусов и ЗП, поквартально:

```
-- пример использования FULL JOIN
WITH cteBonus AS(
   SELECT
      YEAR(BonusDate) BonusYear,
      EmployeeID,
      SUM(CASE WHEN DATEPART(QUARTER,BonusDate)=1 THEN BonusAmount END) BonusAmountQ1,
      SUM(CASE WHEN DATEPART(QUARTER,BonusDate)=2 THEN BonusAmount END) BonusAmountQ2,
      SUM(CASE WHEN DATEPART(QUARTER,BonusDate)=3 THEN BonusAmount END) BonusAmountQ3,
      SUM(CASE WHEN DATEPART(QUARTER,BonusDate)=4 THEN BonusAmount END) BonusAmountQ4,
      SUM(BonusAmount) TotalBonusAmount
   FROM EmployeesBonus
   GROUP BY YEAR(BonusDate),EmployeeID
),
cteSalary AS(
   SELECT
      YEAR(SalaryDate) SalaryYear,
      EmployeeID,
      SUM(CASE WHEN DATEPART(QUARTER,SalaryDate)=1 THEN SalaryAmount END) SalaryAmountQ1,
      SUM(CASE WHEN DATEPART(QUARTER,SalaryDate)=2 THEN SalaryAmount END) SalaryAmountQ2,
      SUM(CASE WHEN DATEPART(QUARTER,SalaryDate)=3 THEN SalaryAmount END) SalaryAmountQ3,
      SUM(CASE WHEN DATEPART(QUARTER,SalaryDate)=4 THEN SalaryAmount END) SalaryAmountQ4,
      SUM(SalaryAmount) TotalSalaryAmount
   FROM EmployeesSalary
   GROUP BY YEAR(SalaryDate),EmployeeID
)

SELECT
   ISNULL(s.SalaryYear,b.BonusYear) AccYear,
   ISNULL(s.EmployeeID,b.EmployeeID) EmployeeID,
   s.SalaryAmountQ1,s.SalaryAmountQ2,s.SalaryAmountQ3,s.SalaryAmountQ4,
   s.TotalSalaryAmount,
   b.BonusAmountQ1,b.BonusAmountQ2,b.BonusAmountQ3,b.BonusAmountQ4,
   b.TotalBonusAmount,
   ISNULL(s.TotalSalaryAmount,0)+ISNULL(b.TotalBonusAmount,0) TotalAmount
FROM cteSalary s
FULL JOIN cteBonus b ON s.EmployeeID=b.EmployeeID AND s.SalaryYear=b.BonusYear
```

Попробуйте самостоятельно разобрать, почему я здесь применил именно FULL JOIN. Посмотрите на результаты, которые дают запросы размещенные в блоке WITH.

Приложение 2 – OVER и аналитические функции

Предложение OVER служит для проведения дополнительных вычислений, на окончательном наборе, полученном оператором SELECT (в подзапросах или запросах). Поэтому предложения OVER может быть применено только в блоке SELECT, т.е. его нельзя использовать, например, в блоке WHERE.

Выражения с использованием OVER могут в некоторых ситуациях значительно сократить запрос. В данном приложении я постарался привести самые основные моменты с использованием данной конструкции. Надеюсь, что самостоятельная проработка каждого приведенного здесь запроса и их результатов, поможет вам разобраться с особенностями конструкции OVER и вы сможете применять ее по назначению (не злоупотребляя ими чрезмерно там, где можно обойтись без них и наоборот) при написании своих запросов.

Для демонстрационных целей, для получения более наглядных результатов, добавим немного новых данных:

```
-- добавим новые должности
SET IDENTITY_INSERT Positions ON
INSERT Positions(ID,Name)VALUES
(10,N'Маркетолог'),
(11,N'Логист')
SET IDENTITY_INSERT Positions OFF



-- новые сотрудники
INSERT Employees(ID,Name,DepartmentID,PositionID,HireDate,Salary,Email)VALUES
(1006,N'Антонов А.А.',4,10,'20150215',1800,'a.antonov@test.tt'),
(1007,N'Максимов М.М.',5,11,'20150405',1200,'m.maksimov@test.tt'),
(1008,N'Данилов Д.Д.',5,11,'20150410',1200,'d.danolov@test.tt'),
(1009,N'Остапов О.О.',5,11,'20150415',1200,'o.ostapov@test.tt')
```

Предложение OVER дает возможность делать агрегатные вычисления, без применения группировки

```
SELECT
  ID,
  Name,
  DepartmentID,
  Salary,
  -- получаем сумму ЗП всех сотрудников
  SUM(Salary) OVER() AllSalary,
  -- получаем сумму ЗП сотрудников этого же отдела
  SUM(Salary) OVER(PARTITION BY DepartmentID) DepartmentSalary,
  -- процент ЗП сотрудника от суммы ЗП всего отдела
  CAST(Salary/SUM(Salary) OVER(PARTITION BY DepartmentID)*100 AS numeric(20,3)) SalaryPercentOfDepSalary,
  -- кол-во всех сотрудников
  COUNT(*) OVER() AllEmplCount,
  -- кол-во сотрудников в отделе
  COUNT(*) OVER(PARTITION BY DepartmentID) DepEmplCount
FROM Employees
```

ID	Name	DepartmentID	Salary	AllSalary	DepartmentSalary	SalaryPercentOfDepSalary	AllEmplCount	DepEmplCount
1005	Александров А.А.	NULL	2000.00	19900.00	2000.00	100.000	10	1
1000	Иванов И.И.	1	5000.00	19900.00	5000.00	100.000	10	1
1002	Сидоров С.С.	2	2500.00	19900.00	2500.00	100.000	10	1
1003	Андреев А.А.	3	2000.00	19900.00	5000.00	40.000	10	3
1004	Николаев Н.Н.	3	1500.00	19900.00	5000.00	30.000	10	3
1001	Петров П.П.	3	1500.00	19900.00	5000.00	30.000	10	3
1006	Антонов А.А.	4	1800.00	19900.00	1800.00	100.000	10	1
1007	Максимов М.М.	5	1200.00	19900.00	3600.00	33.333	10	3
1008	Данилов Д.Д.	5	1200.00	19900.00	3600.00	33.333	10	3
1009	Остапов О.О.	5	1200.00	19900.00	3600.00	33.333	10	3

Предложение «PARTITION BY» позволяет сделать разбиение данных по группам, можно сказать выполняет здесь роль «GROUP BY».

Можно задать группировку по нескольким полям, использовать выражения, например, «PARTITION BY DepartmentID,PositionID», «PARTITION BY DepartmentID,YEAR(HireDate)».

Поэкспериментируйте и с другими агрегатными функциями, которые мы разбирали – AVG, MIN, MAX, COUNT с DISTINCT.

Нумерация и ранжирование строк

Для цели нумерации строк используется функция ROW_NUMBER.

Пронумеруем сотрудников по полю Name и по нескольким полям LastName,FirstName,MiddleName:

```
SELECT
  ID,
  Name,
  -- нумирация в порядке значений Name
  ROW_NUMBER() OVER(ORDER BY Name) EmpNoByName,
  -- нумирация в порядке значений LastName,FirstName,MiddleName
  ROW_NUMBER() OVER(ORDER BY LastName,FirstName,MiddleName) EmpNoByFullName
FROM Employees
ORDER BY Name
```

ID	Name	EmpNoByName	EmpNoByFullName
1005	Александров А.А.	1	6
1003	Андреев А.А.	2	7
1006	Антонов А.А.	3	1
1008	Данилов Д.Д.	4	2
1000	Иванов И.И.	5	8
1007	Максимов М.М.	6	3
1004	Николаев Н.Н.	7	4
1009	Остапов О.О.	8	5
1001	Петров П.П.	9	9
1002	Сидоров С.С.	10	10

Здесь для задания порядка в OVER используется предложение «ORDER BY».

Для разбиения на группы, здесь так же в OVER можно использовать предложение «PARTITION BY»:

```
SELECT
  emp.ID,
  emp.Name EmpName,
  dep.Name DepName,
  -- нумирация сотрудников в разрезе отделов, в порядке значений Name
  ROW_NUMBER() OVER(PARTITION BY dep.ID ORDER BY emp.Name) EmpNoInDepByName
FROM Employees emp
LEFT JOIN Departments dep ON emp.DepartmentID=dep.ID
ORDER BY dep.Name,emp.Name
```

ID	EmpName	DepName	EmpNoInDepByName
1005	Александров А.А.	NULL	1
1000	Иванов И.И.	Администрация	1
1002	Сидоров С.С.	Бухгалтерия	1
1003	Андреев А.А.	ИТ	1
1004	Николаев Н.Н.	ИТ	2
1001	Петров П.П.	ИТ	3
1008	Данилов Д.Д.	Логистика	1
1007	Максимов М.М.	Логистика	2
1009	Остапов О.О.	Логистика	3
1006	Антонов А.А.	Маркетинг и реклама	1

Ранжирование строк – это можно сказать нумерация, только группами. Есть 2 вида нумерации, с дырками (RANK) и без дырок (DENSE_RANK).

```
SELECT
  emp.ID,
  emp.Name EmpName,
  emp.PositionID,
  -- кол-во сотрудников в разрезе должностей
  COUNT(*) OVER(PARTITION BY emp.PositionID) EmpCountInPos,
  -- ранжирование с дырками - следующий номер зависит от кол-ва записей в предыдущей группе
  RANK() OVER(ORDER BY emp.PositionID) RankValue,
  -- ранжирование без дырок - плотная нумерация (последовательная)
  DENSE_RANK() OVER(ORDER BY emp.PositionID) DenseRankValue
FROM Employees emp
LEFT JOIN Positions pos ON emp.PositionID=pos.ID
```

ID	EmpName	PositionID	EmpCountInPos	RankValue	DenseRankValue
1005	Александров А.А.	NULL	1	1	1
1002	Сидоров С.С.	1	1	2	2
1000	Иванов И.И.	2	1	3	3
1001	Петров П.П.	3	2	4	4
1004	Николаев Н.Н.	3	2	4	4
1003	Андреев А.А.	4	1	6	5
1006	Антонов А.А.	10	1	7	6
1007	Максимов М.М.	11	3	8	7
1008	Данилов Д.Д.	11	3	8	7
1009	Остапов О.О.	11	3	8	7

Аналитические функции: LAG() и LEAD(), FIRST_VALUE() и LAST_VALUE()

Данные функции позволяют получить значения другой строки относительно текущей строки.

Рассмотрим LAG() и LEAD():

```
SELECT
  ID CurrEmpID,
  Name CurrEmpName,
  -- значения предыдущей строки
  LAG(ID) OVER(ORDER BY ID) PrevEmpID,
  LAG(Name) OVER(ORDER BY ID) PrevEmpName,
  LAG(ID,2) OVER(ORDER BY ID) PrevPrevEmpID,
  LAG(Name,2,'not found') OVER(ORDER BY ID) PrevPrevEmpName,
  -- значения следующей строки
  LEAD(ID) OVER(ORDER BY ID) NextEmpID,
  LEAD(Name) OVER(ORDER BY ID) NextEmpName,
  LEAD(ID,2) OVER(ORDER BY ID) NextNextEmpID,
  LEAD(Name,2,'not found') OVER(ORDER BY ID) NextNextEmpName
FROM Employees
ORDER BY ID
```

CurrEmpID	CurrEmpName	PrevEmpID	PrevEmpName	PrevPrevEmpID	PrevPrevEmpName	NextEmpID	NextEmpName	NextNextEmpID	NextNextEmpName
1000	Иванов И.И.	NULL	NULL	NULL	not found	1001	Петров П.П.	1002	Сидоров С.С.
1001	Петров П.П.	1000	Иванов И.И.	NULL	not found	1002	Сидоров С.С.	1003	Андреев А.А.
1002	Сидоров С.С.	1001	Петров П.П.	1000	Иванов И.И.	1003	Андреев А.А.	1004	Николаев Н.Н.
1003	Андреев А.А.	1002	Сидоров С.С.	1001	Петров П.П.	1004	Николаев Н.Н.	1005	Александров А.А.
1004	Николаев Н.Н.	1003	Андреев А.А.	1002	Сидоров С.С.	1005	Александров А.А.	1006	Антонов А.А.
1005	Александров А.А.	1004	Николаев Н.Н.	1003	Андреев А.А.	1006	Антонов А.А.	1007	Максимов М.М.
1006	Антонов А.А.	1005	Александров А.А.	1004	Николаев Н.Н.	1007	Максимов М.М.	1008	Данилов Д.Д.
1007	Максимов М.М.	1006	Антонов А.А.	1005	Александров А.А.	1008	Данилов Д.Д.	1009	Остапов О.О.
1008	Данилов Д.Д.	1007	Максимов М.М.	1006	Антонов А.А.	1009	Остапов О.О.	NULL	not found
1009	Остапов О.О.	1008	Данилов Д.Д.	1007	Максимов М.М.	NULL	NULL	NULL	not found

В данных функциях вторым параметром можно указать сдвиг относительно текущей строки, а третьим параметром можно указать возвращаемое значение для случая если для указанного смещения строки не существует.

Для разбиения данных по группам, попробуйте самостоятельно добавить предложение «PARTITION BY» в OVER, например, «OVER(PARTITION BY emp.DepartmentID ORDER BY emp.ID)».

Рассмотрим FIRST_VALUE() и LAST_VALUE():

```
SELECT
  ID CurrEmpID,
  Name CurrEmpName,
  DepartmentID,
  -- первое значение в группе
  FIRST_VALUE(ID) OVER(PARTITION BY DepartmentID ORDER BY ID) FirstEmpID,
  FIRST_VALUE(Name) OVER(PARTITION BY DepartmentID ORDER BY ID) FirstEmpName,
  -- последнее значение в группе
  LAST_VALUE(ID) OVER(PARTITION BY DepartmentID ORDER BY ID RANGE BETWEEN CURRENT ROW AND UNBOUN
DED FOLLOWING) LastEmpID,
  LAST_VALUE(Name) OVER(PARTITION BY DepartmentID ORDER BY ID RANGE BETWEEN CURRENT ROW AND UNBO
UNDED FOLLOWING) LastEmpName
FROM Employees
ORDER BY DepartmentID,ID
```

CurrEmpID	CurrEmpName	DepartmentID	FirstEmpID	FirstEmpName	LastEmpID	LastEmpName
1005	Александров А.А.	NULL	1005	Александров А.А.	1005	Александров А.А.
1000	Иванов И.И.	1	1000	Иванов И.И.	1000	Иванов И.И.
1002	Сидоров С.С.	2	1002	Сидоров С.С.	1002	Сидоров С.С.
1001	Петров П.П.	3	1001	Петров П.П.	1004	Николаев Н.Н.
1003	Андреев А.А.	3	1001	Петров П.П.	1004	Николаев Н.Н.
1004	Николаев Н.Н.	3	1001	Петров П.П.	1004	Николаев Н.Н.
1006	Антонов А.А.	4	1006	Антонов А.А.	1006	Антонов А.А.
1007	Максимов М.М.	5	1007	Максимов М.М.	1009	Остапов О.О.
1008	Данилов Д.Д.	5	1007	Максимов М.М.	1009	Остапов О.О.
1009	Остапов О.О.	5	1007	Максимов М.М.	1009	Остапов О.О.

Думаю, здесь все понятно. Стоит только объяснить, что такое RANGE.

Параметры RANGE и ROWS

При помощи дополнительных параметров «RANGE» и «ROWS», можно изменить область работы функции, которая работает с предложением OVER. У каждой функции по умолчанию используется какая-то своя область действия. Такая область обычно называется окном.

Важное замечание. В разных СУБД для одних и тех же функций область по умолчанию может быть разной, поэтому нужно быть внимательным и смотреть справку конкретной СУБД по каждой отдельной функции.

Можно создавать окна по двум критериям:

1. по диапазону (RANGE) значений данных
2. по смещению (ROWS) относительно текущей строки

Общий синтаксис этих опций выглядит следующим образом:

Вариант 1:

{ROWS | RANGE} {{UNBOUNDED | выражение} PRECEDING | CURRENT ROW}

Вариант 2:

{ROWS | RANGE}
BETWEEN
{{UNBOUNDED PRECEDING | CURRENT ROW |
{UNBOUNDED | выражение 1}{PRECEDING | FOLLOWING}}
AND
{{UNBOUNDED FOLLOWING | CURRENT ROW |
{UNBOUNDED | выражение 2}{PRECEDING | FOLLOWING}}

Здесь проще понять если проанализировать в Excel результат запроса:

```
SELECT
  ID,
  Salary,

  SUM(Salary) OVER() Sum1,
  -- сумма всех строк - "все предыдущие" и "все последующие"
  SUM(Salary) OVER(ORDER BY ID ROWS BETWEEN unbounded preceding AND unbounded following) Sum2,
  -- сумма строк до текущей строки включительно - "все предыдущие" и "текущая строка"
  SUM(Salary) OVER(ORDER BY ID ROWS BETWEEN unbounded preceding AND current row) Sum3,
  -- сумма всех последующих от текущей строки включительно - "текущая строка" и "все последующие
"
  SUM(Salary) OVER(ORDER BY ID ROWS BETWEEN current row AND unbounded following) Sum4,
  -- сумма следующих трех строк - "1 следующую" и "3 следующие"
  SUM(Salary) OVER(ORDER BY ID ROWS BETWEEN 1 following AND 3 following) Sum5,
  -- сумма трех строк - "1 предыдущая" и "1 следующую"
  SUM(Salary) OVER(ORDER BY ID ROWS BETWEEN 1 preceding AND 1 following) Sum6,
  -- сумма предыдущих "трех предыдущих" и "текущей"
  SUM(Salary) OVER(ORDER BY ID ROWS 3 preceding) Sum7,
  -- сумма "всех предыдущих" и "текущей"
  SUM(Salary) OVER(ORDER BY ID ROWS unbounded preceding) Sum8
FROM Employees
ORDER BY ID
```

ID	Salary	Sum1	Sum2	Sum3	Sum4	Sum5	Sum6	Sum7	Sum8
1000	5000.00	19900.00	19900.00	5000.00	19900.00	6000.00	6500.00	5000.00	5000.00
1001	1500.00	19900.00	19900.00	6500.00	14900.00	6000.00	9000.00	6500.00	6500.00
1002	2500.00	19900.00	19900.00	9000.00	13400.00	5500.00	6000.00	9000.00	9000.00
1003	2000.00	19900.00	19900.00	11000.00	10900.00	5300.00	6000.00	11000.00	11000.00
1004	1500.00	19900.00	19900.00	12500.00	8900.00	5000.00	5500.00	7500.00	12500.00
1005	2000.00	19900.00	19900.00	14500.00	7400.00	4200.00	5300.00	8000.00	14500.00
1006	1800.00	19900.00	19900.00	16300.00	5400.00	3600.00	5000.00	7300.00	16300.00
1007	1200.00	19900.00	19900.00	17500.00	3600.00	2400.00	4200.00	6500.00	17500.00
1008	1200.00	19900.00	19900.00	18700.00	2400.00	1200.00	3600.00	6200.00	18700.00
1009	1200.00	19900.00	19900.00	19900.00	1200.00	NULL	2400.00	5400.00	19900.00

С RANGE все тоже самое, только здесь смещения идут не относительно строк, а относительно их значений. Поэтому в данном случае в ORDER BY допустимы значения только типа дата или число.

```
SELECT
  PositionID,
  Salary,

  SUM(Salary) OVER(PARTITION BY PositionID) Sum1,
  -- сумма ЗП для всех значений PositionID - "все меньшие" и "все большие"
  SUM(Salary) OVER(ORDER BY PositionID RANGE BETWEEN unbounded preceding AND unbounded following
) Sum2,
  -- сумма ЗП для значений меньших PositionID до текущего значения включительно - "все меньшие"
и "текущее значение" (значения<=PositionID)
  SUM(Salary) OVER(ORDER BY PositionID RANGE BETWEEN unbounded preceding AND current row) Sum3,
  -- сумма ЗП для всех больших значений от текущего значения включительно - "текущее значение" и
"все большие" (значения>=PositionID)
  SUM(Salary) OVER(ORDER BY PositionID RANGE BETWEEN current row AND unbounded following) Sum4,

  /*
    Увы следующие комбинации для RANGE в MS SQL не работают, хотя в Oracle они работают.

    Вырезки из MSDN:

    Предложение RANGE не может использоваться со <спецификацией неподписанного значения> PRECEDI
NG или со <спецификацией неподписанного значения> FOLLOWING.

    <спецификация неподписанного значения> PRECEDING
      Указывается с <беззнаковым указанием значения> для обозначения числа строк или значений пе
ред текущей строкой.
      Эта спецификация не допускается в предложении RANGE.

    <спецификация неподписанного значения> FOLLOWING
      Указывается с <беззнаковым указанием значения> для обозначения числа строк или значений по
сле текущей строки.
      Эта спецификация не допускается в предложении RANGE.
  */
  -- сумма ЗП для трех значений - "+1" и "+3" (значение BETWEEN PositionID+1 AND PositionID+3)
  --SUM(Salary) OVER(ORDER BY PositionID RANGE BETWEEN 1 following AND 3 following) Sum5,
  -- сумма ЗП для трех значений  - "-1" и "+1" (значение BETWEEN PositionID-1 AND PositionID+1)
  --SUM(Salary) OVER(ORDER BY PositionID RANGE BETWEEN 1 preceding AND 1 following) Sum6,
  -- сумма ЗП для предыдущих трех значений - "-3" и "текущее" (значение BETWEEN PositionID-3 AND
PositionID)
  --SUM(Salary) OVER(ORDER BY PositionID RANGE 3 preceding) Sum7,

  -- сумма ЗП для "всех предыдущих значений" и "текущего" (значения<=PositionID)
  SUM(Salary) OVER(ORDER BY PositionID RANGE unbounded preceding) Sum8
FROM Employees
ORDER BY PositionID
```

PositionID	Salary	Sum1	Sum2	Sum3	Sum4	Sum8
NULL	2000.00	2000.00	19900.00	2000.00	19900.00	2000.00
1	2500.00	2500.00	19900.00	4500.00	17900.00	4500.00
2	5000.00	5000.00	19900.00	9500.00	15400.00	9500.00
3	1500.00	3000.00	19900.00	12500.00	10400.00	12500.00
3	1500.00	3000.00	19900.00	12500.00	10400.00	12500.00
4	2000.00	2000.00	19900.00	14500.00	7400.00	14500.00
10	1800.00	1800.00	19900.00	16300.00	5400.00	16300.00
11	1200.00	3600.00	19900.00	19900.00	3600.00	19900.00
11	1200.00	3600.00	19900.00	19900.00	3600.00	19900.00
11	1200.00	3600.00	19900.00	19900.00	3600.00	19900.00

Заключение

Вот и все, уважаемые читатели, на этом я оканчиваю свой учебник по SQL (DDL, DML).

Надеюсь, что вам было интересно провести время за прочтением данного материала, а главное надеюсь, что он принес вам понимание самых важных базовых конструкций языка SQL.

Учитесь, практикуйтесь, добивайтесь получения правильных результатов.

Спасибо за внимание! На этом пока все.

PS. Отдельное спасибо всем, кто помогал сделать данный материал лучше, указывая на опечатки или давая дельные советы!