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
*/
```

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.

Типы данных SQL

Типы данных SQL разделяются на три группы:
— строковые;
— с плавающей точкой (дробные числа);
— целые числа, дата и время.

Типы данных SQL строковые

Типы данных SQL	Описание
CHAR(size)	Строки фиксированной длиной (могут содержать буквы, цифры и специальные символы). Фиксированный размер указан в скобках. Можно записать до 255 символов
VARCHAR(size)	Может хранить не более 255 символов.
TINYTEXT	Может хранить не более 255 символов.
TEXT	Может хранить не более 65 535 символов.
BLOB	Может хранить не более 65 535 символов.
MEDIUMTEXT	Может хранить не более 16 777 215 символов.
MEDIUMBLOB	Может хранить не более 16 777 215 символов.
LONGTEXT	Может хранить не более 4 294 967 295 символов.
LONGBLOB	Может хранить не более 4 294 967 295 символов.
ENUM(x,y,z,etc.)	Позволяет вводить список допустимых значений. Можно ввести до 65535 значений в SQL Тип данных ENUM список. Если при вставке значения не будет присутствовать в списке ENUM, то мы получим пустое значение. Ввести возможные значения можно в таком формате: ENUM ('X', 'Y', 'Z')
SET	SQL Тип данных SET напоминает ENUM за исключением того, что SET может содержать до 64 значений.

Типы данных SQL с плавающей точкой (дробные числа) и целые числа

Типы данных SQL	Описание
TINYINT(size)	Может хранить числа от -128 до 127
SMALLINT(size)	Диапазон от -32 768 до 32 767
MEDIUMINT(size)	Диапазон от -8 388 608 до 8 388 607

INT(size)	Диапазон от -2 147 483 648 до 2 147 483 647
BIGINT(size)	Диапазон от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
FLOAT(size,d)	Число с плавающей точкой небольшой точности.
DOUBLE(size,d)	Число с плавающей точкой двойной точности.
DECIMAL(size,d)	Дробное число, хранящееся в виде строки.

Типы данных SQL — Дата и время

Типы данных SQL	Описание
DATE()	Дата в формате ГГГГ-ММ-ДД
DATETIME()	Дата и время в формате ГГГГ-ММ-ДД ЧЧ:ММ:СС
TIMESTAMP()	Дата и время в формате timestamp. Однако при получении значения поля оно отображается не в формате timestamp, а в виде ГГГГ-ММ-ДД ЧЧ:ММ:СС
TIME()	Время в формате ЧЧ:ММ:СС
YEAR()	Год в двух значной или в четырехзначном формате.

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)
)
```

На заметку

Общая концепция языка SQL для большинства СУБД остается одинаковой. 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.

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

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).

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),
  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)
)
```

Если таблица создана, но обнаружилось, что необходимо внести изменения в ее структуру, используется оператор ALTER TABLE

Синтаксис ALTER TABLE на примере MS SQL Server

Рассмотрим общий формальный синтаксис на примере SQL Server от Microsoft:

```
   ALTER TABLE имя_таблицы [WITH CHECK | WITH NOCHECK]
{ ADD имя_столбца тип_данных_столбца [атрибуты_столбца] |
  DROP COLUMN имя_столбца |
  ALTER COLUMN имя_столбца тип_данных_столбца [NULL|NOT NULL] |
  ADD [CONSTRAINT] определение_ограничения |
```

DROP [CONSTRAINT] имя_ограничения}

Итак, используя SQL-оператор ALTER TABLE, мы сможем выполнить разные сценарии изменения таблицы. Далее будут рассмотрены некоторые из этих сценариев.

Добавляем новый столбец

Для примера добавим новый column Address в таблицу Customers:

```
ALTER TABLE Customers
ADD Address NVARCHAR(50) NULL;
```

В примере выше столбец Address имеет тип NVARCHAR, плюс для него определён NULL-атрибут. Если же в таблице уже существуют данные, команда ALTER TABLE не выполнится. Однако если надо добавить столбец, который не должен принимать NULL-значения, можно установить значение по умолчанию, используя атрибут DEFAULT:

```
ALTER TABLE Customers
ADD Address NVARCHAR(50) NOT NULL DEFAULT 'Неизвестно';
```

Тогда, если в таблице существуют данные, для них для column Address добавится значение "Неизвестно".

Удаляем столбец

Теперь можно удалить column Address:

```
ALTER TABLE Customers
DROP COLUMN Address;
```

Меняем тип

Продолжим манипуляции с таблицей Customers: теперь давайте поменяем тип данных столбца FirstName на NVARCHAR(200).

```
ALTER TABLE Customers
ALTER COLUMN FirstName NVARCHAR(200);
```

Добавляем ограничения CHECK

Если добавлять ограничения, SQL Server автоматически проверит существующие данные на предмет их соответствия добавляемым ограничениям. В случае несоответствия, они не добавятся. Давайте ограничим Age по возрасту.

```
ALTER TABLE Customers
ADD CHECK (Age > 21);
```

При наличии в таблице строк со значениями, которые не соответствуют ограничению, sql-команда не выполнится. Если надо избежать проверки и добавить ограничение всё равно, используют выражение WITH NOCHECK:

```
ALTER TABLE Customers WITH NOCHECK
ADD CHECK (Age > 21);
```

По дефолту применяется значение WITH CHECK, проверяющее на соответствие ограничениям.

Добавляем внешний ключ

Представим, что изначально в базу данных будут добавлены 2 таблицы, которые между собой не связаны:

Теперь добавим к столбцу CustomerId ограничение внешнего ключа (таблица Orders):

```
ALTER TABLE Orders
ADD FOREIGN KEY(CustomerId) REFERENCES Customers(Id);
```

Добавляем первичный ключ

Применяя определенную выше таблицу Orders, можно добавить к ней для столбца Id первичный ключ:

```
ALTER TABLE Orders
ADD PRIMARY KEY (Id);
```

Добавляем ограничения с именами

Добавляя ограничения, можно указать имя для них — для этого пригодится оператор CONSTRAINT (имя прописывается после него).

Ограничения могут применяться либо на уровне столбца, либо на уровне таблицы. Ограничения на уровне столбца применяются только к одному столбцу, тогда как ограничения на уровне таблицы применяются ко всей таблице. Вот некоторые из наиболее часто используемых ограничений, доступных в SQL:

NOT NULL Constraint — столбец не может содержать значение NULL.

DEFAULT Constraint — задает значение по умолчанию для столбца, если оно не указано.

UNIQUE Constraint — все значения в столбце могут быть разными.

PRIMARY Key — уникальная идентификация каждой строки/записи в таблице базы данных.

FOREIGN Key — уникальная идентификация строки/записи в любой другой таблице базы данных.

CHECK Constraint — ограничение CHECK обеспечивает, чтобы все значения в столбце удовлетворяли определенным условиям.

INDEX — используется для быстрого создания данных базы данных.

ADD CONSTRAINT

Команда ADD CONSTRAINT используется для создания ограничения после того, как таблица уже создана.

ALTER TABLE <table_name>
ADD [CONSTRAINT <constraint_name>]
type (<column_name>);

Для существующих таблиц можно добавить ограничение, используя инструкцию ALTER TABLE с предложением ADD.

Элементы синтаксиса:

- table имя таблицы;
- constraint имя ограничения;
- type тип ограничения;
- column имя столбца, на который распространяется ограничение.

Синтаксис имени ограничения является необязательным, хотя рекомендуется его придерживаться. Если ограничениям не присваивать имена, имена ограничений создаются системой.

DROP CONSTRAINT

Команда DROP CONSTRAINT используется для удаления уникального первичного ключа, внешнего ключа или ограничения проверки.

Чтобы удалить уникальное ограничение, используйте следующий SQL код:

```
ALTER TABLE Persons
DROP CONSTRAINT UC_Person;
```

Чтобы удалить ограничение первичного ключа, используйте следующий SQL:

```
ALTER TABLE Persons
DROP CONSTRAINT PK_Person;
```

Чтобы удалить ограничение внешнего ключа, используйте следующий SQL:

```
ALTER TABLE Orders
DROP CONSTRAINT FK_PersonOrder;
```

Чтобы удалить ограничение проверки, используйте следующий SQL:

```
ALTER TABLE Persons
DROP CONSTRAINT CHK_PersonAge;
```

Чтобы удалить ограничения, следует знать их имя. Если с этим проблема, имя всегда можно определить с помощью SQL Server Management Studio.

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 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 Employees
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

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:

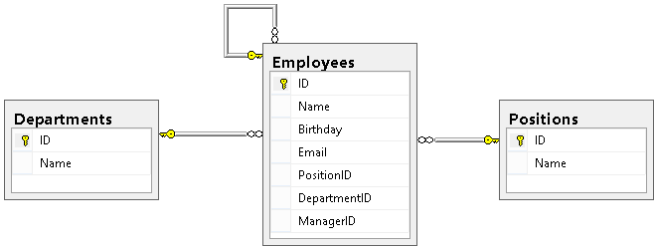
```
ALTER TABLE Employees ADD ManagerID int
```

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):



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.

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 не указано имя ограничения, т.к. в случае DEFAULT это не столь критично. Но если делать по-хорошему, то 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.

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

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	Position ID	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.

Создание самостоятельных индексов

Под самостоятельностью здесь имеются 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.

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.

Список источников

https://schoolsw3.com/sql/sql_ref_constraint.php
https://html5css.ru/sql/sql_foreignkey.php
https://webformyself.com/sql-koncepcii-rdbms
https://habr.com/ru/post/255361
https://otus.ru/nest/post/1684