In [1]:

```
import pandas as pd
import numpy as np
```

Подключение к бд

In [2]:

```
import sqlalchemy
# sqlalchemy.__version__
```

In [3]:

```
# !pip install pyodbc
import pyodbc
import warnings
warnings.filterwarnings('ignore')
```

In [4]:

```
conn = pyodbc.connect('DSN=TestDB;Trusted_Connection=yes;')
```

In [5]:

```
def select(sql):
  return pd.read_sql(sql,conn)
```

01-connect

Создание базы данных

```
--031 CreateDatabase
CREATE DATABASE TestDB
```

In [6]:

```
# 043 CreateTable
cur = conn.cursor()
sql = '''
drop table if exists TestTable;
drop table if exists TestTable2;
drop table if exists TestTable3;

CREATE TABLE TestTable(
    [ProductId] [INT] IDENTITY(1,1) NOT NULL,
    [CategoryId] [INT] NOT NULL,
    [ProductName] [VARCHAR](100) NOT NULL,
    [Price] [Money] NULL
)

CREATE TABLE TestTable2(
    [CategoryId] [INT] IDENTITY(1,1) NOT NULL,
    [CategoryName] [VARCHAR](100) NOT NULL
)

CREATE TABLE TestTable3(
    [ProductId] [INT] IDENTITY(1,1) NOT NULL,
    [ProductName] [VARCHAR](100) NOT NULL,
    [Weight] [DECIMAL](18, 2) NULL,
    [Price] [Money] NULL,
    [Summa] AS ([Weight] * [Price]) PERSISTED
)

'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [7]:

```
# 044 AlterTable
cur = conn.cursor()
sql = '''
ALTER TABLE TestTable3 ADD [SummaDop] AS ([Weight] * [Price]) PERSISTED;
ALTER TABLE TestTable ALTER COLUMN [Price] [Money] NOT NULL;
ALTER TABLE TestTable DROP COLUMN [Price];
ALTER TABLE TestTable ADD [Price] [Money] NULL;
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [8]:

```
# 047 InsertTable
cur = conn.cursor()
sql = '''
INSERT INTO TestTable VALUES
    (1,'Клавиатура', 100),
    (1, 'Мышь', 50),
    (2, 'Телефон', 300)

INSERT INTO TestTable2 VALUES
    ('Комплектующие компьютера'),
    ('Мобильные устройства')
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [9]:

```
sql = 'SELECT * FROM TestTable'
select(sql)
```

Out[9]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [10]:

```
sql = 'SELECT * FROM TestTable2'
select(sql)
```

Out[10]:

	CategoryId	CategoryName
0	1	Комплектующие компьютера
1	2	Мобильные устройства

02-TOP-operator

49 Оператор TOP

In [11]:

```
cur = conn.cursor()
sql = '''
delete from TestTable;
INSERT INTO TestTable
       VALUES (1,'Системный блок', 300),
              (1,'Монитор', 200),
              (1,'Клавиатура', 100),
              (1,'Мышь', 50),
              (3,'Принтер', 200),
              (3,'Сканер', 150),
              (2,'Телефон', 250),
              (2,'Планшет', 300)
'''
cur.execute(sql)
conn.commit()
cur.close()
sql = 'SELECT * FROM TestTable'
select(sql)
```

Out[11]:

	ProductId	CategoryId	ProductName	Price
0	4	1	Системный блок	300.0
1	5	1	Монитор	200.0
2	6	1	Клавиатура	100.0
3	7	1	Мышь	50.0
4	8	3	Принтер	200.0
5	9	3	Сканер	150.0
6	10	2	Телефон	250.0
7	11	2	Планшет	300.0

Возвращает только 20 процентов итогового результата:

In [12]:

```
sql = '''
   SELECT TOP (20) PERCENT ProductId, ProductName, Price
   FROM TestTable
   ORDER BY Price DESC;
'''
select(sql)
```

Out[12]:

	ProductId	ProductName	Price
0	4	Системный блок	300.0
1	11	Планшет	300.0

WITH TIES

Допустим, Вам нужно определить 5 самых дорогих товаров.
Вы, соответственно, отсортируете данные по столбцу с ценой и укажете оператор TOP 5, но,
если товаров с одинаковой ценой (эта цена входит в число самых больших) несколько, например, 7,
Вам все равно вернется 5 Tlgm: @it_boooks 50 Глава 4 - Выборка данных – оператор SELECT строк, что,
как Вы понимаете, неправильно, так как самых дорогих товаров на самом деле 7.
Чтобы это узнать или, как говорят, определить, есть ли «хвосты», т.е. строки с таким же значением,
которые не попали в выборку, за счет ограничения TOP, можно использовать параметр **WITH TIES**.

In [13]:

```
sql = '''
SELECT TOP 4 ProductID, ProductName, Price
FROM TestTable ORDER BY Price DESC;
'''
select(sql)
```

Out[13]:

	ProductID	ProductName	Price
0	4	Системный блок	300.0
1	11	Планшет	300.0
2	10	Телефон	250.0
3	5	Монитор	200.0

In [14]:

```
sql = '''
SELECT TOP 4 WITH TIES ProductID, ProductName, Price
FROM TestTable ORDER BY Price DESC;
'''
select(sql)
```

Out[14]:

	ProductID	ProductName	Price
0	4	Системный блок	300.0
1	11	Планшет	300.0
2	10	Телефон	250.0
3	8	Принтер	200.0
4	5	Монитор	200.0

Как в SQL получить первые (или последние) строки запроса? TOP или OFFSET?

In [15]:

```
sql = 'SELECT * FROM TestTable'
select(sql)
```

Out[15]:

	ProductId	CategoryId	ProductName	Price
0	4	1	Системный блок	300.0
1	5	1	Монитор	200.0
2	6	1	Клавиатура	100.0
3	7	1	Мышь	50.0
4	8	3	Принтер	200.0
5	9	3	Сканер	150.0
6	10	2	Телефон	250.0
7	11	2	Планшет	300.0

Как в SQL получить первые (или последние) строки запроса? TOP или OFFSET? | Info-Comp.ru - IT-блог для начинающих (https://info-comp.ru/obucheniest/672-get-first-query-records-sql.html)

In [16]:

```
sql = '''
SELECT TOP 5 ProductID, ProductName, Price FROM TestTable;
'''
select(sql)
```

Out[16]:

	ProductID	ProductName	Price
0	4	Системный блок	300.0
1	5	Монитор	200.0
2	6	Клавиатура	100.0
3	7	Мышь	50.0
4	8	Принтер	200.0

In [17]:

```
sql = '''
SELECT TOP 4 ProductID, ProductName, Price
FROM TestTable
ORDER BY Price DESC;
'''
select(sql)
```

Out[17]:

	ProductID	ProductName	Price
0	4	Системный блок	300.0
1	11	Планшет	300.0
2	10	Телефон	250.0
3	5	Монитор	200.0

In [18]:

```
sql = '''
SELECT ProductId, ProductName, Price
FROM TestTable
ORDER BY Price DESC
OFFSET 0 ROWS FETCH NEXT 4 ROWS ONLY;
'''
select(sql)
```

Out[18]:

	ProductId	ProductName	Price
0	4	Системный блок	300.0
1	11	Планшет	300.0
2	10	Телефон	250.0
3	5	Монитор	200.0

Получаем последние строки SQL запроса с помощью TOP

In [19]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[19]:

	ProductId	CategoryId	ProductName	Price
0	4	1	Системный блок	300.0
1	5	1	Монитор	200.0
2	6	1	Клавиатура	100.0
3	7	1	Мышь	50.0
4	8	3	Принтер	200.0
5	9	3	Сканер	150.0
6	10	2	Телефон	250.0
7	11	2	Планшет	300.0

In [20]:

```
sql = '''
WITH
SRC AS (
    --Получаем 5 последних строк в таблице
    SELECT TOP (5) ProductId, ProductName, Price
    FROM TestTable
    ORDER BY ProductId DESC
)
SELECT * FROM SRC
ORDER BY Price; --Применяем нужную нам сортировку
'''
select(sql)
```

Out[20]:

	ProductId	ProductName	Price
0	7	Мышь	50.0
1	9	Сканер	150.0
2	8	Принтер	200.0
3	10	Телефон	250.0
4	11	Планшет	300.0

Получаем последние строки SQL запроса с помощью OFFSET-FETCH

In [21]:

```
sql = '''
--Объявляем переменную
DECLARE @CNT INT;

--Узнаем количество строк в таблице
SELECT @CNT = COUNT(*)
FROM TestTable;

--Получаем 5 последних строк
SELECT ProductId, ProductName, Price
FROM TestTable
ORDER BY ProductId
OFFSET @CNT - 5 ROWS FETCH NEXT 5 ROWS ONLY;
'''
select(sql)
```

Out[21]:

	ProductId	ProductName	Price
0	7	Мышь	50.0
1	8	Принтер	200.0
2	9	Сканер	150.0
3	10	Телефон	250.0
4	11	Планшет	300.0

03-GroupBy-OrderBy

57 Группировка – GROUP BY

In [22]:

```
cur = conn.cursor()
sql = '''
delete from TestTable;
INSERT INTO TestTable
       VALUES (1,'Системный блок', 300),
              (1,'Монитор', 200),
              (1,'Клавиатура', 100),
              (1,'Мышь', 50),
              (3,'Принтер', 200),
              (3,'Сканер', 150),
              (2,'Телефон', 250),
              (2,'Планшет', 300)
'''
cur.execute(sql)
conn.commit()
cur.close()
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[22]:

	ProductId	CategoryId	ProductName	Price
0	12	1	Системный блок	300.0
1	13	1	Монитор	200.0
2	14	1	Клавиатура	100.0
3	15	1	Мышь	50.0
4	16	3	Принтер	200.0
5	17	3	Сканер	150.0
6	18	2	Телефон	250.0
7	19	2	Планшет	300.0

In [23]:

```
sql = '''
SELECT
    COUNT(*) AS [Количество строк],
    SUM(Price) AS [Сумма по столбцу Price],
    MAX(Price) AS [Максимальное значение в столбце Price],
    MIN(Price) AS [Минимальное значение в столбце Price],
    AVG(Price) AS [Среднее значение в столбце Price]
FROM TestTable;
'''
select(sql)
```

Out[23]:

	Количество строк	Сумма по столбцу Price	Максимальное значение в столбце Price	Минимальное значение в столбце Price	Среднее значение в столбце Price
0	8	1550.0	300.0	50.0	193.75

In [24]:

```
sql = '''
SELECT CategoryId AS [Id категории],
    COUNT(*) AS [Количество строк],
    MAX(Price) AS [Максимальное значение в столбце Price],
    MIN(Price) AS [Минимальное значение в столбце Price],
    AVG(Price) AS [Среднее значение в столбце Price]
FROM TestTable
GROUP BY CategoryId;
'''
select(sql)
```

Out[24]:

	Id категории	Количество строк	Максимальное значение в столбце Price	Минимальное значение в столбце Price	Среднее значение в столбце Price
0	1	4	300.0	50.0	162.5
1	2	2	300.0	250.0	275.0
2	3	2	200.0	150.0	175.0

In [25]:

```
sql = '''
SELECT CategoryId AS [Id категории],
    COUNT(*) AS [Количество строк],
    MAX(Price) AS [Максимальное значение в столбце Price],
    MIN(Price) AS [Минимальное значение в столбце Price],
    AVG(Price) AS [Среднее значение в столбце Price]
FROM TestTable
WHERE CategoryId <> 1
GROUP BY CategoryId;
'''
select(sql)
```

Out[25]:

	Id категории	Количество строк	Максимальное значение в столбце Price	Минимальное значение в столбце Price	Среднее значение в столбце Price
0	2	2	300.0	250.0	275.0
1	3	2	200.0	150.0	175.0

In [26]:

```
sql = '''
SELECT CategoryId AS [Id категории],
    COUNT(*) AS [Количество строк]
FROM TestTable
GROUP BY CategoryId
HAVING COUNT(*) > 2;
'''
select(sql)
```

Out[26]:

	Id категории	Количество строк
0	1	4

60 Сортировка - ORDER BY

In [27]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[27]:

	ProductId	CategoryId	ProductName	Price
0	12	1	Системный блок	300.0
1	13	1	Монитор	200.0
2	14	1	Клавиатура	100.0
3	15	1	Мышь	50.0
4	16	3	Принтер	200.0
5	17	3	Сканер	150.0
6	18	2	Телефон	250.0
7	19	2	Планшет	300.0

In [28]:

```
sql = '''
SELECT ProductID, ProductName, Price
FROM TestTable
ORDER BY Price DESC;
'''
select(sql)
```

Out[28]:

	ProductID	ProductName	Price
0	12	Системный блок	300.0
1	19	Планшет	300.0
2	18	Телефон	250.0
3	16	Принтер	200.0
4	13	Монитор	200.0
5	17	Сканер	150.0
6	14	Клавиатура	100.0
7	15	Мышь	50.0

Следующий запрос возвращает все строки, начиная со второй, т.е. первая строка будет пропущена:

In [29]:

```
sql = '''
SELECT ProductID, ProductName, Price
FROM TestTable
ORDER BY Price DESC
OFFSET 1 ROWS;
'''
select(sql)
```

Out[29]:

	ProductID	ProductName	Price
0	19	Планшет	300.0
1	18	Телефон	250.0
2	16	Принтер	200.0
3	13	Монитор	200.0
4	17	Сканер	150.0
5	14	Клавиатура	100.0
6	15	Мышь	50.0

In [30]:

```
sql = '''
SELECT ProductID, ProductName, Price
FROM TestTable
ORDER BY Price DESC
OFFSET 1 ROWS FETCH NEXT 3 ROWS ONLY;
'''
select(sql)
```

Out[30]:

	ProductID	ProductName	Price
0	19	Планшет	300.0
1	18	Телефон	250.0
2	13	Монитор	200.0

04-Join

62 Объединение JOIN

In [31]:

```
cur = conn.cursor()
sql = '''
truncate table TestTable;
INSERT INTO TestTable VALUES
    (1,'Клавиатура', 100),
    (1, 'Мышь', 50),
    (2, 'Телефон', 300)
'''
cur.execute(sql)
conn.commit()
cur.close()
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[31]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [32]:

```
sql = '''SELECT * FROM TestTable2'''
select(sql)
```

Out[32]:

	CategoryId	CategoryName
0	1	Комплектующие компьютера
1	2	Мобильные устройства

In [33]:

```
sql = '''
SELECT T1.ProductName, T2.CategoryName, T1.Price
FROM TestTable T1
INNER JOIN TestTable2 T2 ON T1.CategoryId = T2.CategoryId
ORDER BY T1.CategoryId;
'''
select(sql)
```

Out[33]:

	ProductName	CategoryName	Price
0	Клавиатура	Комплектующие компьютера	100.0
1	Мышь	Комплектующие компьютера	50.0
2	Телефон	Мобильные устройства	300.0

In [34]:

```
sql = '''
SELECT T1.ProductName, T2.CategoryName, T1.Price
FROM TestTable T1
LEFT JOIN TestTable2 T2 ON T1.CategoryId = T2.CategoryId
ORDER BY T1.CategoryId;
'''
select(sql)
```

Out[34]:

	ProductName	CategoryName	Price
0	Клавиатура	Комплектующие компьютера	100.0
1	Мышь	Комплектующие компьютера	50.0
2	Телефон	Мобильные устройства	300.0

68 Объединение UNION

In [35]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[35]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [36]:

```
sql = '''SELECT * FROM TestTable2'''
select(sql)
```

Out[36]:

	CategoryId	CategoryName
0	1	Комплектующие компьютера
1	2	Мобильные устройства

In [37]:

```
sql = '''
SELECT T1.ProductId, T1.ProductName, T1.Price
FROM TestTable T1
WHERE T1.ProductId = 1
UNION
SELECT T1.ProductId, T1.ProductName, T1.Price
FROM TestTable T1
WHERE T1.ProductId = 3;
'''
select(sql)
```

Out[37]:

	ProductId	ProductName	Price
0	1	Клавиатура	100.0
1	3	Телефон	300.0

In [38]:

```
sql = '''
SELECT T1.ProductId, T1.ProductName, T1.Price
FROM TestTable T1
WHERE T1.ProductId = 1
UNION
SELECT T1.ProductId, T1.ProductName, T1.Price
FROM TestTable T1
WHERE T1.ProductId = 1;
'''
select(sql)
```

Out[38]:

	ProductId	ProductName	Price
0	1	Клавиатура	100.0

In [39]:

```
sql = '''
SELECT T1.ProductId, T1.ProductName, T1.Price
FROM TestTable T1
WHERE T1.ProductId = 1
UNION ALL
SELECT T1.ProductId, T1.ProductName, T1.Price
FROM TestTable T1
WHERE T1.ProductId = 1;
'''
select(sql)
```

Out[39]:

	ProductId	ProductName	Price
0	1	Клавиатура	100.0
1	1	Клавиатура	100.0

70 Объединение INTERSECT и EXCEPT

In [40]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[40]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [41]:

```
sql = '''SELECT * FROM TestTable2'''
select(sql)
```

Out[41]:

	CategoryId	CategoryName
0	1	Комплектующие компьютера
1	2	Мобильные устройства

INTERSECT (пересечение) – данный оператор выводит одинаковые строки из первого, второго и последующих наборов данных.

In [42]:

```
sql = '''
SELECT T1.ProductId, T1.ProductName, T1.Price
FROM TestTable T1
WHERE T1.ProductId = 1
INTERSECT
SELECT T1.ProductId, T1.ProductName, T1.Price
FROM TestTable T1
WHERE T1.ProductId = 1;
'''
select(sql)
```

Out[42]:

	ProductId	ProductName	Price
0	1	Клавиатура	100.0

In [43]:

```
sql = '''
SELECT T1.ProductId, T1.ProductName, T1.Price
FROM TestTable T1
WHERE T1.ProductId = 1
UNION
SELECT T1.ProductId, T1.ProductName, T1.Price
FROM TestTable T1
WHERE T1.ProductId = 2;
'''
select(sql)
```

Out[43]:

	ProductId	ProductName	Price
0	1	Клавиатура	100.0
1	2	Мышь	50.0

In [44]:

```
sql = '''
SELECT T1.ProductId, T1.ProductName, T1.Price
FROM TestTable T1
WHERE T1.ProductId = 1
INTERSECT
SELECT T1.ProductId, T1.ProductName, T1.Price
FROM TestTable T1
WHERE T1.ProductId = 2;
'''
select(sql)
```

Out[44]:

	ProductId	ProductName	Price

EXCEPT (разность) – этот оператор выводит только те данные из первого набора строк, которых нет во втором наборе. Он полезен, например, тогда, когда необходимо сравнить две таблицы и вывести только те строки из первой таблицы, которых нет в другой таблице.

In [45]:

```
sql = '''
SELECT T1.ProductId, T1.ProductName, T1.Price
FROM TestTable T1
WHERE T1.ProductId = 1
EXCEPT
SELECT T1.ProductId, T1.ProductName, T1.Price
FROM TestTable T1
WHERE T1.ProductId = 2;
'''
select(sql)
```

Out[45]:

	ProductId	ProductName	Price
0	1	Клавиатура	100.0

74 Подзапросы (вложенные запросы)

In [46]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[46]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [47]:

```
sql = '''SELECT * FROM TestTable2'''
select(sql)
```

Out[47]:

	CategoryId	CategoryName
0	1	Комплектующие компьютера
1	2	Мобильные устройства

In [48]:

```
sql = '''
SELECT T2.CategoryName AS [Название категории],
(
    SELECT COUNT(*)
    FROM TestTable
    WHERE CategoryId = T2.CategoryId
) AS [Количество товаров]
FROM TestTable2 T2;
'''
select(sql)
```

Out[48]:

	Название категории	Количество товаров
0	Комплектующие компьютера	2
1	Мобильные устройства	1

In [49]:

```
sql = '''
SELECT ProductId, Price
FROM (
    SELECT ProductId, Price FROM TestTable
    ) AS Q1;
'''
select(sql)
```

Out[49]:

	ProductId	Price
0	1	100.0
1	2	50.0
2	3	300.0

In [50]:

```
sql = '''
SELECT Q1.ProductId, Q1.Price, Q2.CategoryName
FROM (
    SELECT ProductId, Price, CategoryId
    FROM TestTable
) AS Q1
LEFT JOIN (
    SELECT CategoryId, CategoryName
    FROM TestTable2
) AS Q2 ON Q1.CategoryId = Q2.CategoryId;
'''
select(sql)
```

Out[50]:

	ProductId	Price	CategoryName
0	1	100.0	Комплектующие компьютера
1	2	50.0	Комплектующие компьютера
2	3	300.0	Мобильные устройства

05-VIEW

78 Пользовательские представления

In [51]:

```
cur = conn.cursor()
sql = '''
truncate table TestTable;
INSERT INTO TestTable VALUES
    (1,'Клавиатура', 100),
    (1, 'Мышь', 50),
    (2, 'Телефон', 300);

truncate table TestTable2;
INSERT INTO TestTable2 VALUES
    ('Комплектующие компьютера'),
    ('Мобильные устройства')
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [52]:

```
cur = conn.cursor()
sql = '''
drop view if exists ViewCntProducts;
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [53]:

```
cur = conn.cursor()
sql = '''
CREATE VIEW ViewCntProducts
AS
SELECT T2.CategoryName AS CategoryName,
(
    SELECT COUNT(*)
    FROM TestTable
    WHERE CategoryId = T2.CategoryId
) AS CntProducts
FROM TestTable2 T2
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [54]:

```
cur = conn.cursor()
sql = '''
ALTER VIEW ViewCntProducts
AS
SELECT T2.CategoryId AS CategoryId,
T2.CategoryName AS CategoryName,
(
    SELECT COUNT(*)
    FROM TestTable
    WHERE CategoryId = T2.CategoryId
) AS CntProducts
FROM TestTable2 T2
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [55]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[55]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [56]:

```
sql = '''SELECT * FROM TestTable2'''
select(sql)
```

Out[56]:

	CategoryId	CategoryName
0	1	Комплектующие компьютера
1	2	Мобильные устройства

In [57]:

```
sql = '''
SELECT * FROM ViewCntProducts;
'''
select(sql)
```

Out[57]:

	CategoryId	CategoryName	CntProducts
0	1	Комплектующие компьютера	2
1	2	Мобильные устройства	1

80 Системные представления

In [58]:

```
sql = '''
SELECT * FROM sys.tables;
'''
select(sql)
```

Out[58]:

	name	object_id	principal_id	schema_id	parent_object_id	type	type_de
0	TestTable7	11147085	None	1	0	U	USER_TABI
1	TestTable8	59147256	None	1	0	U	USER_TABI
2	german_credit	66099276	None	1	0	U	USER_TABI
3	client_transactions	82099333	None	1	0	U	USER_TABI
4	greater_1000_credit	114099447	None	1	0	U	USER_TABI
5	null_test	146099561	None	1	0	U	USER_TABI
6	AutitTestTable	155147598	None	1	0	U	USER_TABI
7	purpose	194099732	None	1	0	U	USER_TABI
8	TestTable	203147769	None	1	0	U	USER_TABI
9	TestTable2	219147826	None	1	0	U	USER_TABI
10	TestTable3	235147883	None	1	0	U	USER_TABI
11	dupl_test	258099960	None	1	0	U	USER_TABI
12	dupls	274100017	None	1	0	U	USER_TABI
13	categories	290100074	None	1	0	U	USER_TABI
14	users	450100644	None	1	0	U	USER_TABI
15	items	466100701	None	1	0	U	USER_TABI
16	clients_task_name	482100758	None	1	0	U	USER_TABI
17	salary	770101784	None	1	0	U	USER_TABI
18	client_log	786101841	None	1	0	U	USER_TABI
19	revenue	802101898	None	1	0	U	USER_TABI
20	TestTableXML	1022626686	None	1	0	U	USER_TABI

	name	object_id	principal_id	schema_id	parent_object_id	type	type_de
21	tbl	1470628282	None	1	0	U	USER_TABL
22	Department	1630628852	None	1	0	U	USER_TABL
23	Employee	1646628909	None	1	0	U	USER_TABL
24	TestTableDop	1854629650	None	1	0	U	USER_TABL
25	TestTable4	1982630106	None	1	0	U	USER_TABL
26	TestTable5	2014630220	None	1	0	U	USER_TABL
27	TestTable6	2094630505	None	1	0	U	USER_TABL

28 rows × 41 columns

In [59]:

```
sql = '''
SELECT *
FROM sys.columns
WHERE object_id = object_id('TestTable');
'''
select(sql)
```

Out[59]:

	object_id	name	column_id	system_type_id	user_type_id	max_length	precision	s
0	203147769	ProductId	1	56	56	4	10	
1	203147769	CategoryId	2	56	56	4	10	
2	203147769	ProductName	3	167	167	100	0	
3	203147769	Price	5	60	60	8	19	

4 rows × 36 columns

In [60]:

```
cur = conn.cursor()
sql = '''
drop view if exists ViewCntProducts;
'''
cur.execute(sql)
conn.commit()
cur.close()
```

06-Update

In [61]:

```
cur = conn.cursor()
sql = '''
drop table if exists TestTable;
CREATE TABLE TestTable(
    [ProductId] [INT] IDENTITY(1,1) NOT NULL,
    [CategoryId] [INT] NOT NULL,
    [ProductName] [VARCHAR](100) NOT NULL,
    [Price] [Money] NULL
)

drop table if exists TestTable2;
CREATE TABLE TestTable2(
    [CategoryId] [INT] IDENTITY(1,1) NOT NULL,
    [CategoryName] [VARCHAR](100) NOT NULL
)
'''
cur.execute(sql)
conn.commit()
```

In [62]:

```
cur = conn.cursor()
sql = '''
truncate table TestTable;
INSERT INTO TestTable VALUES
    (1,'Клавиатура', 100),
    (1, 'Мышь', 50),
    (2, 'Телефон', 300);

INSERT INTO TestTable VALUES
    (1,'Клавиатура', 100),
    (1, 'Мышь', 50),
    (2, 'Телефон', 300);

truncate table TestTable2;
INSERT INTO TestTable2 VALUES
    ('Комплектующие компьютера'),
    ('Мобильные устройства');

INSERT INTO TestTable2 VALUES
    ('Комплектующие компьютера'),
    ('Мобильные устройства');
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [63]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[63]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0
3	4	1	Клавиатура	100.0
4	5	1	Мышь	50.0
5	6	2	Телефон	300.0

In [64]:

```
sql = '''SELECT * FROM TestTable2'''
select(sql)
```

Out[64]:

	CategoryId	CategoryName
0	1	Комплектующие компьютера
1	2	Мобильные устройства
2	3	Комплектующие компьютера
3	4	Мобильные устройства

85 INSERT INTO TestTable

In [65]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[65]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0
3	4	1	Клавиатура	100.0
4	5	1	Мышь	50.0
5	6	2	Телефон	300.0

In [66]:

```
cur = conn.cursor()
sql = '''
INSERT INTO TestTable (CategoryId, ProductName, Price)
    SELECT CategoryId, ProductName, Price
    FROM TestTable WHERE ProductId > 3;
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [67]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[67]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0
3	4	1	Клавиатура	100.0
4	5	1	Мышь	50.0
5	6	2	Телефон	300.0
6	7	1	Клавиатура	100.0
7	8	1	Мышь	50.0
8	9	2	Телефон	300.0

172 Конструкция SELECT INTO

In [68]:

```
cur = conn.cursor()
sql = '''
drop table if exists TestTableDop;

SELECT T1.ProductName, T2.CategoryName, T1.Price
INTO TestTableDop
FROM TestTable T1
LEFT JOIN TestTable2 T2 ON T1.CategoryId = T2.CategoryId
WHERE T1.CategoryId = 1
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [69]:

```
sql = '''
SELECT * FROM TestTableDop
'''
select(sql)
```

Out[69]:

	ProductName	CategoryName	Price
0	Клавиатура	Комплектующие компьютера	100.0
1	Мышь	Комплектующие компьютера	50.0
2	Клавиатура	Комплектующие компьютера	100.0
3	Мышь	Комплектующие компьютера	50.0
4	Клавиатура	Комплектующие компьютера	100.0
5	Мышь	Комплектующие компьютера	50.0

86 Обновление данных – UPDATE

In [70]:

```
cur = conn.cursor()
sql = '''
UPDATE TestTable SET
    Price = 120
WHERE ProductId = 1
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [71]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[71]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	120.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0
3	4	1	Клавиатура	100.0
4	5	1	Мышь	50.0
5	6	2	Телефон	300.0
6	7	1	Клавиатура	100.0
7	8	1	Мышь	50.0
8	9	2	Телефон	300.0

In [72]:

```
sql = '''
SELECT * FROM TestTable WHERE ProductId > 3
'''
select(sql)
```

Out[72]:

	ProductId	CategoryId	ProductName	Price
0	4	1	Клавиатура	100.0
1	5	1	Мышь	50.0
2	6	2	Телефон	300.0
3	7	1	Клавиатура	100.0
4	8	1	Мышь	50.0
5	9	2	Телефон	300.0

In [73]:

```
cur = conn.cursor()
sql = '''
UPDATE TestTable SET
    ProductName = 'Тестовый товар',
    Price = 150
WHERE ProductId > 3
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [74]:

```
sql = '''
SELECT * FROM TestTable WHERE ProductId > 3
'''
select(sql)
```

Out[74]:

	ProductId	CategoryId	ProductName	Price
0	4	1	Тестовый товар	150.0
1	5	1	Тестовый товар	150.0
2	6	2	Тестовый товар	150.0
3	7	1	Тестовый товар	150.0
4	8	1	Тестовый товар	150.0
5	9	2	Тестовый товар	150.0

достаточно часто требуется перекинуть данные из одной таблицу в другую,

т.е. обновить записи одной таблицы на значения, которые расположены в другой.

In [75]:

```
cur = conn.cursor()
sql = '''
UPDATE TestTable SET
    ProductName = T2.CategoryName,
    Price = 200
FROM TestTable2 T2 --источник
INNER JOIN TestTable T1 ON T1.CategoryId = T2.CategoryId
WHERE T1.ProductId > 3
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [76]:

```
sql = '''
SELECT * FROM TestTable WHERE ProductId > 3
'''
select(sql)
```

Out[76]:

	ProductId	CategoryId	ProductName	Price
0	4	1	Комплектующие компьютера	200.0
1	5	1	Комплектующие компьютера	200.0
2	6	2	Мобильные устройства	200.0
3	7	1	Комплектующие компьютера	200.0
4	8	1	Комплектующие компьютера	200.0
5	9	2	Мобильные устройства	200.0

90 Удаление данных – DELETE, TRUNCATE

In [77]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[77]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	120.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0
3	4	1	Комплектующие компьютера	200.0
4	5	1	Комплектующие компьютера	200.0
5	6	2	Мобильные устройства	200.0
6	7	1	Комплектующие компьютера	200.0
7	8	1	Комплектующие компьютера	200.0
8	9	2	Мобильные устройства	200.0

In [78]:

```
cur = conn.cursor()
sql = '''
DELETE TestTable WHERE ProductId > 3
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [79]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[79]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	120.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

95 MERGE

In [80]:

```
cur = conn.cursor()
sql = '''
drop table if exists TestTable3;
CREATE TABLE TestTable3(
    [ProductId] [INT] NOT NULL,
    [CategoryId] [INT] NOT NULL,
    [ProductName] [VARCHAR](100) NOT NULL,
    [Price] [Money] NULL
)
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [81]:

```
cur = conn.cursor()
sql = '''
INSERT INTO TestTable3 VALUES
    (1, 1, 'Клавиатура', 0),
    (2, 1, 'Мышь', 0),
    (4, 1, 'Тест', 0)
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [82]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[82]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	120.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [83]:

```
sql = '''SELECT * FROM TestTable3'''
select(sql)
```

Out[83]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	0.0
1	2	1	Мышь	0.0
2	4	1	Тест	0.0

In [84]:

```
sql = '''
MERGE TestTable3 AS T_Base
    USING TestTable AS T_Source
    ON (T_Base.ProductId = T_Source.ProductId)
    WHEN MATCHED THEN
        UPDATE SET
        ProductName = T_Source.ProductName,
        CategoryId = T_Source.CategoryId,
        Price = T_Source.Price
    WHEN NOT MATCHED THEN
        INSERT (ProductId, CategoryId, ProductName, Price) VALUES
        (T_Source.ProductId,
        T_Source.CategoryId,
        T_Source.ProductName,
        T_Source.Price)
    WHEN NOT MATCHED BY SOURCE THEN
        DELETE
    OUTPUT $action AS [Операция],
        Inserted.ProductId,
        Inserted.ProductName AS ProductNameNEW,
        Inserted.Price AS PriceNEW,
        Deleted.ProductName AS ProductNameOLD,
        Deleted.Price AS PriceOLD;
'''
select(sql)
```

Out[84]:

	Операция	ProductId	ProductNameNEW	PriceNEW	ProductNameOLD	PriceOLD
0	INSERT	3.0	Телефон	300.0	None	NaN
1	UPDATE	1.0	Клавиатура	120.0	Клавиатура	0.0
2	UPDATE	2.0	Мышь	50.0	Мышь	0.0
3	DELETE	NaN	None	NaN	Тест	0.0

In [85]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[85]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	120.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [86]:

```
sql = '''SELECT * FROM TestTable3'''
select(sql)
```

Out[86]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	120.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

96 Инструкция OUTPUT

In [87]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[87]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	120.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [88]:

```
sql = '''
INSERT INTO TestTable
    OUTPUT Inserted.ProductId,
        Inserted.CategoryId,
        Inserted.ProductName,
        Inserted.Price
    VALUES (1, 'Тестовый товар 1', 300),
        (1, 'Тестовый товар 2', 500),
        (2, 'Тестовый товар 3', 400);
'''
select(sql)
```

Out[88]:

	ProductId	CategoryId	ProductName	Price
0	10	1	Тестовый товар 1	300.0
1	11	1	Тестовый товар 2	500.0
2	12	2	Тестовый товар 3	400.0

In [89]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[89]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	120.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0
3	10	1	Тестовый товар 1	300.0
4	11	1	Тестовый товар 2	500.0
5	12	2	Тестовый товар 3	400.0

In [90]:

```
sql = '''
UPDATE TestTable SET
    Price = 0
    OUTPUT Inserted.ProductId AS [ProductId],
        Deleted.Price AS [Старое значение Price],
        Inserted.Price AS [Новое значение Price]
    WHERE ProductId > 3;
'''
select(sql)
```

Out[90]:

	ProductId	Старое значение Price	Новое значение Price
0	10	300.0	0.0
1	11	500.0	0.0
2	12	400.0	0.0

In [91]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[91]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	120.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0
3	10	1	Тестовый товар 1	0.0
4	11	1	Тестовый товар 2	0.0
5	12	2	Тестовый товар 3	0.0

In [92]:

```
sql = '''
DELETE TestTable
OUTPUT Deleted.*
WHERE ProductId > 3;
'''
select(sql)
```

Out[92]:

	ProductId	CategoryId	ProductName	Price
0	10	1	Тестовый товар 1	0.0
1	11	1	Тестовый товар 2	0.0
2	12	2	Тестовый товар 3	0.0

In [93]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[93]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	120.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

07-Index

In [94]:

```
cur = conn.cursor()
sql = '''
truncate table TestTable;
INSERT INTO TestTable VALUES
    (1,'Клавиатура', 100),
    (1, 'Мышь', 50),
    (2, 'Телефон', 300);

truncate table TestTable2;
INSERT INTO TestTable2 VALUES
    ('Комплектующие компьютера'),
    ('Мобильные устройства')
'''
cur.execute(sql)
conn.commit()
cur.close()
```

101 Создание индексов

In [95]:

```
cur = conn.cursor()
sql = '''
CREATE UNIQUE CLUSTERED INDEX IX_Clustered ON TestTable
(
    ProductId ASC
)
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [96]:

```
cur = conn.cursor()
sql = '''
CREATE NONCLUSTERED INDEX IX_NonClustered ON TestTable
(
    CategoryId ASC
)
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [97]:

```
cur = conn.cursor()
sql = '''
DROP INDEX IX_NonClustered ON TestTable;
'''
cur.execute(sql)
conn.commit()
cur.close()
```

Иногда требуется изменить эти индексы, например, добавить еще один ключевой столбец или добавить так называемые «Включенные столбцы» - это столбцы, которые не являются ключевыми, но включаются в индекс. За счет этого уменьшается количество дисковых операций ввода-вывода и скорость доступа к данным, соответственно, увеличивается

In [98]:

```
cur = conn.cursor()
sql = '''
CREATE NONCLUSTERED INDEX IX_NonClustered ON TestTable
(
    CategoryId ASC,
    ProductName ASC
)
INCLUDE (Price)
'''
cur.execute(sql)
conn.commit()
cur.close()
```

Изменить индекс:

In [99]:

```
cur = conn.cursor()
sql = '''
CREATE NONCLUSTERED INDEX IX_NonClustered ON TestTable
(
    CategoryId ASC,
    ProductName ASC
)
INCLUDE (Price)
WITH (DROP_EXISTING = ON);
'''
cur.execute(sql)
conn.commit()
cur.close()
```

105 Обслуживание индексов

Если степень фрагментации менее 5%, то реорганизацию или перестроение индекса вообще не стоит запускать;

Если степень фрагментации от 5 до 30%, то имеет смысл запустить реорганизацию индекса, так как данная операция использует минимальные системные ресурсы и не требует долговременных блокировок;

Если степень фрагментации более 30%, то необходимо выполнять перестроение индекса, так как данная операция, при значительной фрагментации, дает больший эффект чем операция реорганизации индекса.

In [100]:

```
sql = '''
SELECT OBJECT_NAME(T1.object_id) AS NameTable,
    T1.index_id AS IndexId,
    T2.name AS IndexName,
    T1.avg_fragmentation_in_percent AS Fragmentation
FROM sys.dm_db_index_physical_stats
    (DB_ID(), NULL, NULL, NULL, NULL) AS T1
LEFT JOIN sys.indexes AS T2 ON T1.object_id = T2.object_id AND T1.index_id = T2.index_id;
'''
select(sql)
```

Out[100]:

	NameTable	IndexId	IndexName	Fragmentation
0	TestTable7	0	None	0.000000
1	TestTable8	0	None	0.000000
2	german_credit	0	None	50.000000
3	client_transactions	0	None	66.666667
4	greater_1000_credit	0	None	25.000000
5	greater_1000_credit	0	None	0.000000
6	greater_1000_credit	0	None	0.000000
7	null_test	0	None	0.000000
8	AutitTestTable	1	PK_AutitTestTable	0.000000
9	purpose	0	None	0.000000
10	dupl_test	0	None	0.000000
11	TestTable	1	IX_Clustered	0.000000
12	TestTable	4	IX_NonClustered	0.000000
13	dupls	0	None	0.000000
14	TestTable2	0	None	0.000000
15	categories	0	None	0.000000
16	categories	0	None	0.000000
17	TestTableDop	0	None	0.000000
18	TestTable3	0	None	0.000000
19	users	0	None	0.000000
20	items	0	None	0.000000
21	clients_task_name	0	None	0.000000
22	salary	0	None	0.000000
23	client_log	0	None	0.000000
24	revenue	0	None	0.000000
25	TestTableXML	1	PK_TestTableXML	0.000000
26	tbl	0	None	0.000000
27	Department	0	None	0.000000
28	Employee	0	None	0.000000

	NameTable	IndexId	IndexName	Fragmentation
29	QueryNotificationErrorsQueue	1	None	0.000000
30	QueryNotificationErrorsQueue	2	None	0.000000
31	TestTable4	1	PK_CategoryId	0.000000
32	EventNotificationErrorsQueue	1	None	0.000000
33	EventNotificationErrorsQueue	2	None	0.000000
34	TestTable5	1	PK_TestTable5	0.000000
35	ServiceBrokerQueue	1	None	0.000000
36	ServiceBrokerQueue	2	None	0.000000
37	TestTable6	0	None	0.000000
38	TestTable6	2	PK_TestTable6_C2	0.000000
39	TestTable6	3	PK_TestTable6_C1	0.000000
40	TestTable6	4	PK_TestTable6_C3	0.000000

In [101]:

```
cur = conn.cursor()
sql = '''
ALTER INDEX IX_NonClustered ON TestTable
    REORGANIZE;
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [102]:

```
cur = conn.cursor()
sql = '''
ALTER INDEX IX_NonClustered ON TestTable
    REBUILD;
'''
cur.execute(sql)
conn.commit()
cur.close()
```

08-CONSTRAINT

In [103]:

```
cur = conn.cursor()
sql = '''
drop table if exists TestTable;
drop table if exists TestTable2;
drop table if exists TestTable3;

CREATE TABLE TestTable(
[ProductId] [INT] IDENTITY(1,1) NOT NULL,
[CategoryId] [INT] NOT NULL,
[ProductName] [VARCHAR](100) NOT NULL,
[Price] [Money] NULL
)

CREATE TABLE TestTable2(
[CategoryId] [INT] IDENTITY(1,1) NOT NULL,
[CategoryName] [VARCHAR](100) NOT NULL
)

CREATE TABLE TestTable3(
[ProductId] [INT] IDENTITY(1,1) NOT NULL,
[ProductName] [VARCHAR](100) NOT NULL,
[Weight] [DECIMAL](18, 2) NULL,
[Price] [Money] NULL,
[Summa] AS ([Weight] * [Price]) PERSISTED
)
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [104]:

```
cur = conn.cursor()
sql = '''
truncate table TestTable;
INSERT INTO TestTable VALUES
    (1,'Клавиатура', 100),
    (1, 'Мышь', 50),
    (2, 'Телефон', 300);

truncate table TestTable2;
INSERT INTO TestTable2 VALUES
    ('Комплектующие компьютера'),
    ('Мобильные устройства')
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [105]:

```
cur = conn.cursor()
sql = '''
drop table if exists TestTable8;
drop table if exists TestTable7;
drop table if exists TestTable6;
drop table if exists TestTable5;
drop table if exists TestTable4;
'''
cur.execute(sql)
conn.commit()
cur.close()
```

109 Создание ограничений

In [106]:

```
cur = conn.cursor()
sql = '''
ALTER TABLE TestTable
    ALTER COLUMN [Price] [Money] NOT NULL;
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [107]:

```
cur = conn.cursor()
sql = '''
ALTER TABLE TestTable
    ADD CONSTRAINT PK_TestTable PRIMARY KEY (ProductId);
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [108]:

```
cur = conn.cursor()
sql = '''
drop table if exists TestTable4
CREATE TABLE TestTable4(
    [CategoryId] [INT] IDENTITY(1,1) NOT NULL
        CONSTRAINT PK_CategoryId PRIMARY KEY,
    [CategoryName] [VARCHAR](100) NOT NULL
)
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [109]:

```
cur = conn.cursor()
sql = '''
drop table if exists TestTable4
CREATE TABLE TestTable4(
    [CategoryId] [INT] IDENTITY(1,1) NOT NULL,
    [CategoryName] [VARCHAR](100) NOT NULL,
    CONSTRAINT PK_CategoryId PRIMARY KEY (CategoryId)
)
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [110]:

```
cur = conn.cursor()
sql = '''
drop table if exists TestTable5
CREATE TABLE TestTable5(
    [ProductId] [INT] IDENTITY(1,1) NOT NULL,
    [CategoryId] [INT] NOT NULL,
    [ProductName] [VARCHAR](100) NOT NULL,
    [Price] [MONEY] NULL,
    CONSTRAINT PK_TestTable5 PRIMARY KEY (ProductId),
    CONSTRAINT FK_TestTable5 FOREIGN KEY (CategoryId)
        REFERENCES TestTable4 (CategoryId)
            ON DELETE CASCADE
            ON UPDATE NO ACTION
);
'''
cur.execute(sql)
conn.commit()
cur.close()
```

Для инструкций ON DELETE и ON UPDATE доступны следующие значения:
NO ACTION - ничего не делать, просто выводить ошибку,
CASCADE – каскадное изменение,
SET NULL - присвоить значение NULL,
SET DEFAULT - присвоить значение по умолчанию.
Эти инструкции необязательные, их можно и не указывать, тогда при изменении ключа, в случае наличия связанных записей,
будет выходить ошибка.

In [111]:

```
cur = conn.cursor()
sql = '''
ALTER TABLE TestTable2
    ADD CONSTRAINT PK_TestTable2 PRIMARY KEY (CategoryId);
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [112]:

```
cur = conn.cursor()
sql = '''
ALTER TABLE TestTable
    ADD CONSTRAINT FK_TestTable FOREIGN KEY (CategoryId)
        REFERENCES TestTable2 (CategoryId);
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [113]:

```
cur = conn.cursor()
sql = '''
drop table if exists TestTable6
CREATE TABLE TestTable6(
    [Column1] [INT] NOT NULL CONSTRAINT PK_TestTable6_C1 UNIQUE,
    [Column2] [INT] NOT NULL,
    [Column3] [INT] NOT NULL,
    CONSTRAINT PK_TestTable6_C2 UNIQUE (Column3)
);
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [114]:

```
cur = conn.cursor()
sql = '''
ALTER TABLE TestTable6
    ADD CONSTRAINT PK_TestTable6_C3 UNIQUE (Column3);
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [115]:

```
cur = conn.cursor()
sql = '''
drop table if exists TestTable7
CREATE TABLE TestTable7(
    [Column1] [INT] NOT NULL,
    [Column2] [INT] NOT NULL,
    CONSTRAINT CK_TestTable7_C1 CHECK (Column1 <> 0)
);
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [116]:

```
cur = conn.cursor()
sql = '''
ALTER TABLE TestTable7
    ADD CONSTRAINT CK_TestTable7_C2 CHECK (Column2 > Column1);
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [117]:

```
cur = conn.cursor()
sql = '''
drop table if exists TestTable8
CREATE TABLE TestTable8(
    [Column1] [INT] NULL CONSTRAINT DF_C1 DEFAULT (1),
    [Column2] [INT] NULL
);
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [118]:

```
cur = conn.cursor()
sql = '''
ALTER TABLE TestTable8
    ADD CONSTRAINT DF_C2 DEFAULT (2) FOR Column2;
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [119]:

```
cur = conn.cursor()
sql = '''
ALTER TABLE TestTable7 DROP CONSTRAINT CK_TestTable7_C1;
ALTER TABLE TestTable7 DROP CONSTRAINT CK_TestTable7_C2;
ALTER TABLE TestTable8 DROP CONSTRAINT DF_C1;
ALTER TABLE TestTable8 DROP CONSTRAINT DF_C2;
'''
cur.execute(sql)
conn.commit()
cur.close()
```

09-Programming

115 Переменные

```
DECLARE @TestTable TABLE (
    [ProductId] [INT] IDENTITY(1,1) NOT NULL,
    [CategoryId] [INT] NOT NULL,
    [ProductName][VARCHAR](100) NOT NULL,
    [Price] [Money] NULL
);

INSERT INTO @TestTable
    SELECT CategoryId, ProductName, Price
        FROM TestTable
        WHERE ProductId <= 3;

SELECT * FROM @TestTable;
```

In [120]:

```
sql = '''
SELECT @@SERVERNAME [Имя локального сервера], @@VERSION AS [Версия SQL сервера];
'''
select(sql)
```

Out[120]:

	Имя локального сервера	Версия SQL сервера
0	HPPROBOOK	Microsoft SQL Server 2019 (RTM-GDR) (KB5014356...

118 Команды условного выполнения

In [121]:

```
sql = '''
DECLARE @TestVar1 INT
DECLARE @TestVar2 VARCHAR(20)

SET @TestVar1 = 5

IF @TestVar1 > 0
    SET @TestVar2 = 'Больше 0'
ELSE
    SET @TestVar2 = 'Меньше 0'

SELECT @TestVar2 AS [Значение TestVar1]
'''
select(sql)
```

Out[121]:

	Значение TestVar1
0	Больше 0

In [122]:

```
sql = '''
DECLARE @TestVar1 INT
DECLARE @TestVar2 VARCHAR(20)

SET @TestVar1 = 0

IF @TestVar1 > 0
    SET @TestVar2 = 'Больше 0'

SELECT @TestVar2 AS [Значение TestVar1]
'''
select(sql)
```

Out[122]:

	Значение TestVar1
0	None

119 IF EXISTS

In [123]:

```
sql = '''
DECLARE @TestVar VARCHAR(20)

IF EXISTS(SELECT * FROM TestTable)
    SET @TestVar = 'Записи есть'
ELSE
    SET @TestVar = 'Записей нет'

SELECT @TestVar AS [Наличие записей]
'''
select(sql)
```

Out[123]:

	Наличие записей
0	Записи есть

120 CASE

In [124]:

```
sql = '''
DECLARE @TestVar1 INT
DECLARE @TestVar2 VARCHAR(20)

SET @TestVar1 = 1

SELECT @TestVar2 =
                CASE @TestVar1 WHEN 1 THEN 'Один'
                    WHEN 2 THEN 'Два'
                    ELSE 'Неизвестное'
                END

SELECT @TestVar2 AS [Число]
'''
select(sql)
```

Out[124]:

	Число
0	Один

121 BEGIN...END

In [125]:

```
sql = '''
DECLARE @TestVar1 INT
DECLARE @TestVar2 VARCHAR(20), @TestVar3 VARCHAR(20)

SET @TestVar1 = 5

IF @TestVar1 NOT IN (0, 1, 2)
BEGIN
    SET @TestVar2 = 'Первая инструкция';
    SET @TestVar3 = 'Вторая инструкция';
END

SELECT @TestVar2 AS [Значение TestVar1], @TestVar3 AS [Значение TestVar2]
'''
select(sql)
```

Out[125]:

	Значение TestVar1	Значение TestVar2
0	Первая инструкция	Вторая инструкция

121 Циклы

In [126]:

```
sql = '''
DECLARE @CountAll INT = 0 --Запускаем цикл

WHILE @CountAll < 10
BEGIN
    SET @CountAll = @CountAll + 1
END

SELECT @CountAll AS [Результат]
'''
select(sql)
```

Out[126]:

	Результат
0	10

In [127]:

```
sql = '''
DECLARE @CountAll INT
SET @CountAll = 0
--Запускаем цикл
WHILE @CountAll < 10
BEGIN
    SET @CountAll = @CountAll + 1
    IF @CountAll = 5 BREAK
END
SELECT @CountAll AS [Результат];
'''
select(sql)
```

Out[127]:

	Результат
0	5

In [128]:

```
sql = '''
DECLARE @Cnt INT = 0
DECLARE @CountAll INT = 0
--Запускаем цикл
WHILE @CountAll < 10
BEGIN
    SET @CountAll += 1
    IF @CountAll = 5
        CONTINUE
    SET @Cnt += 1
END

SELECT @CountAll AS [CountAll], @Cnt AS [Cnt];
'''
select(sql)
```

Out[128]:

	CountAll	Cnt
0	10	9

124 Команда RETURN

In [129]:

```
sql = '''
DECLARE @TestVar INT = 1

IF @TestVar < 0
    RETURN

SELECT @TestVar AS [Результат]
'''
select(sql)
```

Out[129]:

	Результат
0	1

125 Команда GOTO

In [130]:

```
sql = '''
DECLARE @TestVar INT = 0

METKA: --Устанавливаем метку
SET @TestVar += 1 --Увеличиваем значение переменной.
--Проверяем значение переменной
    IF @TestVar < 10 --Если оно меньше 10, то возвращаемся назад к метке
    GOTO METKA

SELECT @TestVar AS [Результат];
'''
select(sql)
```

Out[130]:

	Результат
0	10

In [131]:

```
sql = '''
DECLARE @TestVar INT = 2
DECLARE @Rez INT = 0

IF @TestVar <= 0
    GOTO METKA
SET @Rez = 10 / @TestVar

METKA: --Устанавливаем метку

SELECT @Rez AS [Результат];
'''
select(sql)
```

Out[131]:

	Результат
0	5

125 Команда WAITFOR

In [132]:

```
sql = '''
--Пауза на 5 секунд
WAITFOR DELAY '00:00:05'
SELECT 'Продолжение выполнение инструкции' AS [Test];
'''
select(sql)
```

Out[132]:

	Test
0	Продолжение выполнение инструкции

```
sql = '''
--Пауза до 10 часов
WAITFOR TIME '19:07:00'
SELECT 'Продолжение выполнение инструкции' AS [Test];
'''
select(sql)
```

126 Обработка ошибок

In [133]:

```
sql = '''
--Начало блока обработки ошибок
BEGIN TRY
    --Инструкции, в которых могут возникнуть ошибки
    DECLARE @TestVar1 INT = 10,
            @TestVar2 INT = 0,
            @Rez INT
    SET @Rez = @TestVar1 / @TestVar2
END TRY
--Начало блока CATCH
BEGIN CATCH
    -- Действия, которые будут выполняться в случае возникновения ошибки
    SELECT ERROR_NUMBER() AS [Номер ошибки],
            ERROR_MESSAGE() AS [Описание ошибки]
    SET @Rez = 0
END CATCH

SELECT @Rez AS [Результат];
'''
select(sql)
```

Out[133]:

	Номер ошибки	Описание ошибки
0	8134	Обнаружена ошибка: деление на ноль.

10-Function

129 Пользовательские функции

```
CREATE FUNCTION TestFunction (
    @ProductId INT --Объявление входящих параметров
    )
RETURNS VARCHAR(100) --Тип возвращаемого результата
AS
BEGIN
    --Объявление переменных внутри функции
    DECLARE @ProductName VARCHAR(100);
    --Получение наименования товара по его идентификатору
    SELECT @ProductName = ProductName
```

```
    FROM TestTable
    WHERE ProductId = @ProductId

    --Возвращение результата
    RETURN @ProductName
END
```

Вызов функции. Получение наименования конкретного товара:

In [134]:

```
sql = '''
SELECT dbo.TestFunction(1) AS [Наименование товара]
'''
select(sql)
```

Out[134]:

	Наименование товара
0	Комплектующие компьютера

Вызов функции. Передача в функцию параметра в виде столбца:

In [135]:

```
sql = '''
SELECT ProductId,
    ProductName,
    dbo.TestFunction(ProductId) AS [Наименование товара]
FROM TestTable
'''
select(sql)
```

Out[135]:

	ProductId	ProductName	Наименование товара
0	1	Клавиатура	Комплектующие компьютера
1	2	Мышь	Комплектующие компьютера
2	3	Телефон	Мобильные устройства

Пример обращения к табличной функции.

```
CREATE FUNCTION FT_TestFunction (
    @CategoryId INT --Объявление входящих параметров
)
RETURNS TABLE
AS
RETURN(
    --Получение всех товаров в определѐнной категории
    SELECT ProductId,
        ProductName,
        Price,
        CategoryId
    FROM TestTable
    WHERE CategoryId = @CategoryId
```

```
    )
```

In [136]:

```
sql = '''
SELECT * FROM FT_TestFunction(2)
'''
select(sql)
```

Out[136]:

	ProductId	ProductName	Price	CategoryId
0	3	Телефон	300.0	2

```
CREATE FUNCTION FT_TestFunction2 (
    --Объявление входящих параметров
    @CategoryId INT,
    @Price MONEY
    )
--Определяем результирующую таблицу
RETURNS @TMPTable TABLE (
                        ProductId INT,
                        ProductName VARCHAR(100),
                        Price MONEY,
                        CategoryId INT
                        )
AS
BEGIN
    --Если указана отрицательная цена, то задаем цену равной 0
    IF @Price < 0
        SET @Price = 0
    --Заполняем данными результирующую таблицу
    INSERT INTO @TMPTable
        SELECT ProductId,
            ProductName,
            Price,
            CategoryId
        FROM TestTable
        WHERE CategoryId = @CategoryId
            AND Price <= @Price
    --Возвращаем результат и прекращаем выполнение функции
    RETURN
END
```

In [137]:

```
sql = '''
SELECT * FROM FT_TestFunction2(2, 300)
'''
select(sql)
```

Out[137]:

	ProductId	ProductName	Price	CategoryId
0	3	Телефон	300.0	2

Пример использования функции.

```
ALTER FUNCTION TestFunction (
    @ProductId INT --Объявление входящих параметров
    )
RETURNS VARCHAR(100) --Тип возвращаемого результата
AS
BEGIN
    --Объявление переменных внутри функции
    DECLARE @CategoryName VARCHAR(100);
    --Получение наименования категории товара по идентификатору товара
    SELECT @CategoryName = T2.CategoryName
    FROM TestTable T1
    INNER JOIN TestTable2 T2 ON T1.CategoryId = T2.CategoryId
    WHERE T1.ProductId = @ProductId

    --Возвращение результата
    RETURN @CategoryName
END
```

In [138]:

```
sql = '''
SELECT ProductId,
    ProductName,
    dbo.TestFunction(ProductId) AS [CategoryName]
FROM TestTable
'''
select(sql)
```

Out[138]:

	ProductId	ProductName	CategoryName
0	1	Клавиатура	Комплектующие компьютера
1	2	Мышь	Комплектующие компьютера
2	3	Телефон	Мобильные устройства

135 Строковые функции

In [139]:

```
sql = '''
DECLARE @TestVar VARCHAR(100),
    @TestVar2 VARCHAR(100)

SELECT @TestVar = 'ТеКст', @TestVar2 = 'ТЕкст'
--Без использования функции
SELECT @TestVar AS TestVar, @TestVar2 AS TestVar2
'''
select(sql)
```

Out[139]:

	TestVar	TestVar2
0	ТеКст	ТЕкст

In [140]:

```
sql = '''
DECLARE @TestVar VARCHAR(100),
    @TestVar2 VARCHAR(100)

SELECT @TestVar = 'ТеКст', @TestVar2 = 'ТЕкст'
--С использованием функций
SELECT UPPER(@TestVar) AS TestVar, LOWER(@TestVar2) AS TestVar2
'''
select(sql)
```

Out[140]:

	TestVar	TestVar2
0	ТЕКСТ	текст

In [141]:

```
sql = '''
DECLARE @TestVar VARCHAR(100),
    @TestVar2 VARCHAR(100)

SELECT @TestVar = '1234567890', @TestVar2 = '1234567890'
SELECT LEFT(@TestVar, 5) AS TestVar, RIGHT(@TestVar2, 5) AS TestVar2
'''
select(sql)
```

Out[141]:

	TestVar	TestVar2
0	12345	67890

In [142]:

```
sql = '''
DECLARE @TestVar VARCHAR(100),
    @TestVar2 VARCHAR(100)

SELECT @TestVar = '1234567890', @TestVar2 = '1234567890'
SELECT SUBSTRING(@TestVar, 3, 5) AS TestVar
'''
select(sql)
```

Out[142]:

	TestVar
0	34567

138 Функции для работы с датой и временем

In [143]:

```
sql = '''
DECLARE @TestDate DATETIME

SET @TestDate = GETDATE()

SELECT GETDATE() AS [Текущая дата],
    DATENAME(M, @TestDate) AS [[Название месяца],
    DATEPART(M, @TestDate) AS [[Номер месяца],
    DAY(@TestDate) AS [День],
    MONTH(@TestDate) AS [Месяц],
    YEAR(@TestDate) AS [Год],
    DATEDIFF(D, '01.01.2018', @TestDate) AS [Количество дней],
    DATEADD(D, 5, GETDATE()) AS [+ 5 Дней]
'''
select(sql)
```

Out[143]:

	Текущая дата	[Название месяца	[Номер месяца	День	Месяц	Год	Количество дней	+ 5 Дней
0	2022-12-29 09:17:23.727	Декабрь	12	29	12	2022	1823	2023-01-03 09:17:23.727

139 Математические функции

In [144]:

```
sql = '''
SELECT ABS(-100) AS [ABS],
    ROUND(1.567, 2) AS [ROUND],
    CEILING(1.6) AS [CEILING],
    FLOOR(1.6) AS [FLOOR],
    SQRT(16) AS [SQRT],
    SQUARE(4) AS [SQUARE],
    POWER(4, 2) AS [POWER],
    LOG(10) AS [LOG]
'''
select(sql)
```

Out[144]:

	ABS	ROUND	CEILING	FLOOR	SQRT	SQUARE	POWER	LOG
0	100	1.57	2.0	1.0	4.0	16.0	16	2.302585

140 Функции метаданных

In [145]:

```
sql = '''
SELECT DB_ID() AS [Идентификатор текущей БД],
    DB_NAME() AS [Имя текущей БД],
    OBJECT_ID ('TestTable') AS [Идентификатор таблицы TestTable],
    OBJECT_NAME(149575571) AS [Имя объекта с ИД 149575571]
'''
select(sql)
```

Out[145]:

	Идентификатор текущей БД	Имя текущей БД	Идентификатор таблицы TestTable	Имя объекта с ИД 149575571
0	8	TestDB	331148225	sqlagent_jobsteps_logs

141 Прочие функции

In [146]:

```
sql = '''
SELECT
    ISNULL(NULL, 5) AS [ISNULL],
    COALESCE (NULL, NULL, 5) AS [COALESCE],
    CAST(1.5 AS INT) AS [CAST],
    HOST_NAME() AS [HOST_NAME],
    SUSER_SNAME() AS [SUSER_SNAME],
    USER_NAME() AS [USER_NAME]
'''
select(sql)
```

Out[146]:

	ISNULL	COALESCE	CAST	HOST_NAME	SUSER_SNAME	USER_NAME
0	5	5	1	HPPROBOOK	HPPROBOOK\m.pryanikov	dbo

11-Procedure

143 Пользовательские процедуры

```
--Создаем процедуру
CREATE PROCEDURE TestProcedure (
    @CategoryId INT,
    @ProductName VARCHAR(100)
)
AS
BEGIN
    --Объявляем переменную
    DECLARE @AVG_Price MONEY
```

```
    --Определяем среднюю цену в категории
    SELECT @AVG_Price = ROUND(AVG(Price), 2)
    FROM TestTable
    WHERE CategoryId = @CategoryId

    --Добавляем новую запись
    INSERT INTO TestTable(CategoryId, ProductName, Price) VALUES
    (@CategoryId, LTRIM(RTRIM(@ProductName)),
    @AVG_Price)

    --Возвращаем данные
    SELECT * FROM TestTable WHERE CategoryId = @CategoryId
END
```

Вызываем процедуру:

```
EXEC TestProcedure @CategoryId = 1, @ProductName = 'Тестовый товар'
```

Создание

```
declare @ProcName sysname set @ProcName = 'TestProcedure'
if not exists (select * from dbo.sysobjects where id = object_id(@ProcName))
  exec ('create procedure ' + @ProcName + ' as return')
GO

ALTER PROCEDURE TestProcedure (
    @CategoryId INT,
    @ProductName VARCHAR(100)
)
AS
BEGIN
    --Объявляем переменную
    DECLARE @AVG_Price MONEY

    --Определяем среднюю цену в категории
    SELECT @AVG_Price = ROUND(AVG(Price), 2)
    FROM TestTable
    WHERE CategoryId = @CategoryId

    --Добавляем новую запись
    INSERT INTO TestTable(CategoryId, ProductName, Price) VALUES
    (@CategoryId, LTRIM(RTRIM(@ProductName)),
    @AVG_Price)

    --Возвращаем данные
    SELECT * FROM TestTable WHERE CategoryId = @CategoryId
END

GO
```

```
--Вызываем процедуру
EXEC TestProcedure @CategoryId = 1, @ProductName = 'Тестовый товар'
```

Изменение

```
declare @ProcName sysname set @ProcName = 'TestProcedure'
```

```
declare @ProcName sysname set @ProcName = 'TestProcedure'
if not exists (select * from dbo.sysobjects where id = object_id(@ProcName))
  exec ('create procedure ' + @ProcName + ' as return')
GO

ALTER PROCEDURE TestProcedure (
    @CategoryId INT,
    @ProductName VARCHAR(100),
    @Price MONEY = NULL -- Необязательный параметр
)
AS
BEGIN
    --Если цену не передали, то определяем среднюю цену
    IF @Price IS NULL
        SELECT @Price = ROUND(AVG(Price), 2)
        FROM TestTable
        WHERE CategoryId = @CategoryId

    --Добавляем новую запись
    INSERT INTO TestTable(CategoryId, ProductName, Price) VALUES
    (@CategoryId, LTRIM(RTRIM(@ProductName)), @Price)

    --Возвращаем данные
    SELECT * FROM TestTable WHERE CategoryId = @CategoryId
END

GO
```

```
--Вызываем процедуру
EXEC TestProcedure @CategoryId = 1, @ProductName = 'Тестовый товар', @Price = 100
```

146 Системные хранимые процедуры

```
EXEC sp_helpdb 'TestDB'
EXEC sp_tables @table_type = "'TABLE'"
```

```
EXEC sp_rename TestTable_OldName, TestTable_NewName
```

12-Trigger

In [147]:

```
cur = conn.cursor()
sql = '''
drop table if exists TestTable;
drop table if exists TestTable2;
drop table if exists TestTable3;

CREATE TABLE TestTable(
[ProductId] [INT] IDENTITY(1,1) NOT NULL,
[CategoryId] [INT] NOT NULL,
[ProductName] [VARCHAR](100) NOT NULL,
[Price] [Money] NULL
)

CREATE TABLE TestTable2(
[CategoryId] [INT] IDENTITY(1,1) NOT NULL,
[CategoryName] [VARCHAR](100) NOT NULL
)

CREATE TABLE TestTable3(
[ProductId] [INT] IDENTITY(1,1) NOT NULL,
[ProductName] [VARCHAR](100) NOT NULL,
[Weight] [DECIMAL](18, 2) NULL,
[Price] [Money] NULL,
[Summa] AS ([Weight] * [Price]) PERSISTED
)
'''
cur.execute(sql)
conn.commit()
```

In [148]:

```
cur = conn.cursor()
sql = '''
truncate table TestTable;
INSERT INTO TestTable VALUES
    (1,'Клавиатура', 100),
    (1, 'Мышь', 50),
    (2, 'Телефон', 300);

truncate table TestTable2;
INSERT INTO TestTable2 VALUES
    ('Комплектующие компьютера'),
    ('Мобильные устройства')
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [149]:

```
cur = conn.cursor()
sql = '''
drop table if exists AutitTestTable;

CREATE TABLE AutitTestTable(
    Id INT IDENTITY(1,1) NOT NULL,
    DtChange DATETIME NOT NULL,
    UserName VARCHAR(100) NOT NULL,
    SQL_Command VARCHAR(100) NOT NULL,
    ProductId_Old INT NULL,
    ProductId_New INT NULL,
    CategoryId_Old INT NULL,
    CategoryId_New INT NULL,
    ProductName_Old VARCHAR(100) NULL,
    ProductName_New VARCHAR(100) NULL,
    Price_Old MONEY NULL,
    Price_New MONEY NULL,
    CONSTRAINT PK_AutitTestTable PRIMARY KEY (Id)
)
'''
cur.execute(sql)
conn.commit()
cur.close()
```

148 Создание триггеров на T-SQL

In [150]:

```
cur = conn.cursor()
sql = '''
CREATE TRIGGER TRG_Audit_TestTable ON TestTable
    AFTER INSERT, UPDATE, DELETE
AS
BEGIN
    DECLARE @SQL_Command VARCHAR(100);
    /*
    Определяем, что это за операция на основе наличия записей в таблицах inserted и deleted
    конечно же, лучше делать отдельный триггер для каждой операции
    */
    IF EXISTS (SELECT * FROM inserted) AND NOT EXISTS (SELECT * FROM deleted)
        SET @SQL_Command = 'INSERT'
    IF EXISTS (SELECT * FROM inserted) AND EXISTS (SELECT * FROM deleted)
        SET @SQL_Command = 'UPDATE'
    IF NOT EXISTS (SELECT * FROM inserted) AND EXISTS (SELECT * FROM deleted)
        SET @SQL_Command = 'DELETE'

    -- Инструкция если происходит добавление или обновление записи
    IF @SQL_Command = 'UPDATE' OR @SQL_Command = 'INSERT'
    BEGIN
        INSERT INTO AutitTestTable(DtChange, UserName, SQL_Command, ProductId_Old, ProductI
                CategoryId_Old, CategoryId_New, ProductName_Old, ProductName_New, Price_Old
            SELECT GETDATE(), SUSER_SNAME(), @SQL_Command,
                D.ProductId, I.ProductId, D.CategoryId, I.CategoryId, D.ProductName, I.Prod
            FROM inserted I
            LEFT JOIN deleted D ON I.ProductId = D.ProductId
    END

    -- Инструкция если происходит удаление записи
    IF @SQL_Command = 'DELETE'
    BEGIN
        INSERT INTO AutitTestTable(DtChange, UserName, SQL_Command, ProductId_Old, ProductI
                CategoryId_Old, CategoryId_New, ProductName_Old, ProductName_New, Price_Old
            SELECT GETDATE(), SUSER_SNAME(), @SQL_Command, D.ProductId, NULL, D.CategoryId,
            D.Price, NULL FROM deleted D
    END
END
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [151]:

```
cur = conn.cursor()
sql = '''
--Добавляем запись
INSERT INTO TestTable VALUES (1, 'Новый товар', 0)
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [152]:

```
cur = conn.cursor()
sql = '''
--Изменяем запись
UPDATE TestTable SET
    ProductName = 'Наименование товара',
    Price = 200
WHERE ProductName = 'Новый товар'
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [153]:

```
cur = conn.cursor()
sql = '''
--Удаляем запись
DELETE TestTable
WHERE ProductName = 'Наименование товара'
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [154]:

```
sql = '''
SELECT * FROM TestTable
'''
select(sql)
```

Out[154]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [155]:

```
sql = '''
SELECT * FROM AutitTestTable
'''
select(sql)
```

Out[155]:

	Id	DtChange	UserName	SQL_Command	ProductId_Old	ProductId_New	C
0	1	2022-12-29 09:18:15.567	HPPROBOOK\m.pryanikov	INSERT	NaN	4.0	
1	2	2022-12-29 09:18:17.047	HPPROBOOK\m.pryanikov	UPDATE	4.0	4.0	
2	3	2022-12-29 09:18:19.923	HPPROBOOK\m.pryanikov	DELETE	4.0	NaN	

150 Включение и отключение триггеров

In [156]:

```
cur = conn.cursor()
sql = '''
DISABLE TRIGGER TRG_Audit_TestTable ON TestTable;
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [157]:

```
cur = conn.cursor()
sql = '''
ENABLE TRIGGER TRG_Audit_TestTable ON TestTable;
'''
cur.execute(sql)
conn.commit()
cur.close()
```

13-Cursor

In [158]:

```
cur = conn.cursor()
sql = '''
truncate table TestTable;
INSERT INTO TestTable VALUES
    (1,'Клавиатура', 100),
    (1, 'Мышь', 50),
    (2, 'Телефон', 300);

truncate table TestTable2;
INSERT INTO TestTable2 VALUES
    ('Комплектующие компьютера'),
    ('Мобильные устройства')
'''
cur.execute(sql)
conn.commit()
cur.close()
```

153 Работа с курсорами

In [159]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[159]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [160]:

```
cur = conn.cursor()
sql = '''
--1. Объявление переменных
DECLARE @ProductId INT,
    @ProductName VARCHAR(100),
    @Price MONEY

--2. Объявление курсора
DECLARE TestCursor CURSOR FOR
    SELECT ProductId, ProductName, Price
    FROM TestTable
    WHERE CategoryId = 1
--3. Открываем курсор
OPEN TestCursor
--4. Считываем данные из первой строки в курсоре --и записываем их в переменные
FETCH NEXT FROM TestCursor INTO @ProductId, @ProductName, @Price
-- Запускаем цикл, выйдем из него, когда закончатся строки в курсоре
WHILE @@FETCH_STATUS = 0
BEGIN
    --На каждую итерацию цикла выполняем необходимые нам SQL инструкции
    --Для примера изменяем цену по условию
    IF @Price < 100
        UPDATE TestTable SET
            Price = Price + 10
        WHERE ProductId = @ProductId
        --Считываем следующую строку курсора
        FETCH NEXT FROM TestCursor INTO @ProductId, @ProductName, @Price
END
--5. Закрываем курсор
CLOSE TestCursor
-- Освобождаем ресурсы
DEALLOCATE TestCursor

'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [161]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[161]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	60.0
2	3	2	Телефон	300.0

14-Transaction

In [162]:

```
cur = conn.cursor()
sql = '''
truncate table TestTable;
INSERT INTO TestTable VALUES
    (1,'Клавиатура', 100),
    (1, 'Мышь', 50),
    (2, 'Телефон', 300);

truncate table TestTable2;
INSERT INTO TestTable2 VALUES
    ('Комплектующие компьютера'),
    ('Мобильные устройства')
'''
cur.execute(sql)
conn.commit()
cur.close()
```

157 Команды управления транзакциями

In [163]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[163]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [164]:

```
cur = conn.cursor()
sql = '''
BEGIN TRY
--Начало транзакции
BEGIN TRANSACTION
    --Инструкция 1
    UPDATE TestTable SET CategoryId = 2
    WHERE ProductId = 1

    --Инструкция 2
    UPDATE TestTable SET CategoryId = NULL
    WHERE ProductId = 2

    --...Другие инструкции
END TRY
    BEGIN CATCH
        --В случае непредвиденной ошибки
        --Откат транзакции
        ROLLBACK TRANSACTION

        --Выводим сообщение об ошибке
        SELECT ERROR_NUMBER() AS [Номер ошибки],
            ERROR_MESSAGE() AS [Описание ошибки]

        --Прекращаем выполнение инструкции
        RETURN;
    END CATCH

--Если все хорошо. Сохраняем все изменения
COMMIT TRANSACTION
'''
cur.execute(sql)
conn.commit()
cur.close()
```

515 Не удалось вставить значение NULL в столбец "CategoryId", таблицы "TestDB.dbo.TestTable"; в столбце запрещены значения NULL. Ошибка в UPDATE.

In [165]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[165]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

158 Уровни изоляции

READ UNCOMMITTED — самый низкий уровень, при котором SQL сервер разрешает так называемое

«грязное чтение». Грязным чтением называют считывание неподтвержденных данных, иными словами, если транзакция, которая изменяет данные, не завершена, другая транзакция может получить уже измененные данные, хотя они еще не зафиксированы и могут отмениться.

READ COMMITTED — этот уровень уже запрещает грязное чтение, в данном случае все процессы, запросившие данные, которые изменяются в тот же момент в другой транзакции, будут ждать завершения этой транзакции и подтверждения фиксации данных. Данный уровень по умолчанию используется SQL сервером.

REPEATABLE READ – на данном уровне изоляции запрещается изменение данных между двумя операциями чтения в одной транзакции. Здесь происходит запрет на так называемое ***неповторяющееся чтение*** (или *несогласованный анализ*). Другими словами, если в одной транзакции есть несколько операций чтения, данные будут блокированы, и их нельзя будет изменить в другой транзакции. Таким образом, Вы избежите случаи, когда вначале транзакции Вы запросили данные, провели их анализ (некое вычисления), в конце транзакции запросили те же самые данные, а они уже отличаются от первоначальных, так как они были изменены другой транзакцией. Также уровень REPEATABLE READ запрещает «***Потерянное обновление***» - это когда две транзакции сначала считывают одни и те же данные, а затем изменяют их на основе неких вычислений, в результате обе транзакции выполнятся, но данные будут те, которая зафиксировала последняя операция обновления. Это происходит, потому что данные в операциях чтения в начале этих транзакций не были заблокированы. На данном уровне это исключено.

SERIALIZABLE – данный уровень исключает чтение «*фантомных*» записей. Фантомные записи – это те записи, которые появились между началом и завершением транзакции. Иными словами, в начала транзакции Вы запросили определенные данные, в конце транзакции Вы запрашиваете их снова (с тем же фильтром), но там уже есть и новые данные, которые добавлены другой транзакцией. Более низкие уровни изоляции не блокировали строки, которых еще нет в таблице, данный уровень блокирует все строки, соответствующие фильтру запроса, с которыми будет работать транзакция, как существующие, так и те, что могут быть добавлены.

Также существуют уровни изоляции, алгоритм которых основан на версиях строк, это: ***SNAPSHOT*** и ***READ COMMITTED SNAPSHOT***. Иными словами, SQL Server делает снимок, и, соответственно, хранит последние версии подтвержденных строк. В данном случае, клиенту не нужно ждать снятия блокировок, пока одна транзакция изменит данные, он сразу получает последнюю версию подтвержденных строк. Следует отметить, уровни изоляции, основанные на версиях строк, замедляют операции обновления и удаления,

SNAPSHOT – уровень хранит строки, подтверждѐнные на момент начала транзакции, соответственно, именно эти строки будут считаны в случае обращения к ним из другой транзакции. Данный уровень исключает повторяющееся и фантомное чтение (*примерно так же, как уровень SERIALIZABLE*).

READ COMMITTED SNAPSHOT – этот уровень изоляции работает практически так же, как уровень SNAPSHOT, с одним отличием, он хранит снимок строк, которые подтверждены на момент запуска команды, а не транзакции как в SNAPSHOT.

Также для уровней SNAPSHOT и READ COMMITTED SNAPSHOT предварительно необходимо включить параметр базы данных ALLOW_SNAPSHOT_ISOLATION для уровня изоляции SNAPSHOT и READ_COMMITTED_SNAPSHOT для уровня READ COMMITTED SNAPSHOT. Например

```
ALTER DATABASE TestDB SET ALLOW_SNAPSHOT_ISOLATION ON;
```

15-XML

[Документация Beautiful Soup — Beautiful Soup 4.9.0 documentation (https://www.crummy.com/software/BeautifulSoup/bs4/doc.ru/bs4ru.html)](https://www.crummy.com/software/BeautifulSoup/bs4/doc.ru/bs4ru.html)

In [166]:

```
from bs4 import BeautifulSoup
```

In [167]:

```
def xml(sql, FieldName, xml = True):
    df = select(sql)
    for i, row in df.iterrows():
        if xml:
            print(BeautifulSoup(row[FieldName], "xml").prettify())
        else:
            print(BeautifulSoup(row[FieldName]).prettify())
        print()
```

In [168]:

```
cur = conn.cursor()
sql = '''
Drop table if exists TestTableXML;
CREATE TABLE TestTableXML(
    Id INT IDENTITY(1,1) NOT NULL,
    NameColumn VARCHAR(100) NOT NULL,
    XMLData XML NULL
    CONSTRAINT PK_TestTableXML PRIMARY KEY (Id)
);
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [169]:

```
cur = conn.cursor()
sql = '''
truncate table TestTableXML;
INSERT INTO TestTableXML (NameColumn, XMLData) VALUES(
    'Текст',
    '<Catalog> <Name>Иван</Name> <LastName>Иванов</LastName> </Catalog>'
)
INSERT INTO TestTableXML (NameColumn, XMLData) VALUES(
    'Текст',
    '<Catalog> <Name>Иван</Name> <LastName>Петров</LastName> </Catalog>'
)
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [170]:

```
sql = '''SELECT * FROM TestTableXML'''
xml(sql, 'XMLData')
```

```
<?xml version="1.0" encoding="utf-8"?>
<Catalog>
 <Name>
  Иван
 </Name>
 <LastName>
  Иванов
 </LastName>
</Catalog>

<?xml version="1.0" encoding="utf-8"?>
<Catalog>
 <Name>
  Иван
 </Name>
 <LastName>
  Петров
 </LastName>
</Catalog>
```

160 Методы типа данных XML

Метод query

In [171]:

```
sql = '''
SELECT XMLData.query('/Catalog/Name') AS [Ter Name]
FROM TestTableXML
'''
select(sql)
```

Out[171]:

	Ter Name
0	<Name>Иван</Name>
1	<Name>Иван</Name>

In [172]:

```
xml(sql, 'Ter Name')
```

```
<?xml version="1.0" encoding="utf-8"?>
<Name>
 Иван
</Name>

<?xml version="1.0" encoding="utf-8"?>
<Name>
 Иван
</Name>
```

Метод value

In [173]:

```
sql = '''
SELECT XMLData,
    XMLData.value('/Catalog[1]/LastName[1]','VARCHAR(100)') AS [LastName]
FROM TestTableXML
'''
select(sql)
```

Out[173]:

	XMLData	LastName
0	<Catalog><Name>Иван</Name><LastName>Иванов</La...	Иванов
1	<Catalog><Name>Иван</Name><LastName>Петров</La...	Петров

Метод exist

данный метод используется для того, чтобы проверять наличие тех или иных значений, атрибутов или элементов в XML документе. Метод возвращает значения типа bit, такие как: 1 – если выражение на языке XQuery при запросе возвращает непустой результат, 0 – если возвращается пустой результат, NULL – если данные типа xml, к которым идет обращение, не содержат никаких данных, т.е. NULL.

In [174]:

```
sql = '''
SELECT * FROM TestTableXML WHERE XMLData.exist('/Catalog[1]/LastName') = 1
'''
select(sql)
```

Out[174]:

	Id	NameColumn	XMLData
0	1	Текст	<Catalog><Name>Иван</Name><LastName>Иванов</La...
1	2	Текст	<Catalog><Name>Иван</Name><LastName>Петров</La...

Метод modify

In [175]:

```
sql = '''SELECT * FROM TestTableXML'''
xml(sql, 'XMLData')
```

```
<?xml version="1.0" encoding="utf-8"?>
<Catalog>
 <Name>
  Иван
 </Name>
 <LastName>
  Иванов
 </LastName>
</Catalog>

<?xml version="1.0" encoding="utf-8"?>
<Catalog>
 <Name>
  Иван
 </Name>
 <LastName>
  Петров
 </LastName>
</Catalog>
```

In [176]:

```
cur = conn.cursor()
sql = '''
UPDATE TestTableXML SET
    XMLData.modify('delete /Catalog/LastName')
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [177]:

```
sql = '''SELECT * FROM TestTableXML'''
xml(sql, 'XMLData')
```

```
<?xml version="1.0" encoding="utf-8"?>
<Catalog>
 <Name>
  Иван
 </Name>
</Catalog>

<?xml version="1.0" encoding="utf-8"?>
<Catalog>
 <Name>
  Иван
 </Name>
</Catalog>
```

In [178]:

```
cur = conn.cursor()
sql = '''
UPDATE TestTableXML SET
    XMLData.modify('insert <LastName>Иванов</LastName> as last into (/Catalog)[1] ')
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [179]:

```
sql = '''SELECT * FROM TestTableXML'''
xml(sql, 'XMLData')
```

```
<?xml version="1.0" encoding="utf-8"?>
<Catalog>
 <Name>
  Иван
 </Name>
 <LastName>
  Иванов
 </LastName>
</Catalog>

<?xml version="1.0" encoding="utf-8"?>
<Catalog>
 <Name>
  Иван
 </Name>
 <LastName>
  Иванов
 </LastName>
</Catalog>
```

In [180]:

```
cur = conn.cursor()
sql = '''
UPDATE TestTableXML SET
    XMLData.modify('replace value of(/Catalog/Name[1]/text())[1] with "Сергей" ')
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [181]:

```
sql = '''SELECT * FROM TestTableXML'''
xml(sql, 'XMLData')
```

```
<?xml version="1.0" encoding="utf-8"?>
<Catalog>
 <Name>
  Сергей
 </Name>
 <LastName>
  Иванов
 </LastName>
</Catalog>

<?xml version="1.0" encoding="utf-8"?>
<Catalog>
 <Name>
  Сергей
 </Name>
 <LastName>
  Иванов
 </LastName>
</Catalog>
```

Метод nodes

In [182]:

```
XML_Doc = '''
<Root> <row id="1" Name="Иван"></row> <row id="2" Name="Сергей"></row> </Root>
'''
```

In [183]:

```
print(BeautifulSoup(XML_Doc, "xml").prettify())
```

```
<?xml version="1.0" encoding="utf-8"?>
<Root>
 <row Name="Иван" id="1"/>
 <row Name="Сергей" id="2"/>
</Root>
```

In [184]:

```
sql = f'''
DECLARE @XML_Doc XML;

SET @XML_Doc = '{XML_Doc}'
--'<Root> <row id="1" Name="Иван"></row> <row id="2" Name="Сергей"></row> </Root>';

SELECT TMP.col.value('@id','INT') AS Id,
    TMP.col.value('@Name','VARCHAR(10)') AS Name
FROM @XML_Doc.nodes('/Root/row') TMP(Col);
'''
select(sql)
```

Out[184]:

	Id	Name
0	1	Иван
1	2	Сергей

В данном случае метод nodes сформировал таблицу TMP и столбец Col для пути '/Root/row' в XML данных. Каждый элемент row здесь отдельная строка, методом value мы извлекаем значения атрибутов.

Конструкция FOR XML

Существуют следующие режимы:

RAW – в данном случае в XML документе создается одиночный элемент для каждой строки результирующего набора данных инструкции SELECT;
AUTO – в данном режиме структура XML документа создается автоматически, в зависимости от инструкции SELECT (объединений, вложенных запросов и так далее);
EXPLICIT – режим, при котором Вы сами формируете структуру итогового XML документа, это самый расширенный режим работы конструкции FOR XML и, в то же время, самый трудоемкий;
PATH – это своего рода упрощенный режим EXPLICIT, который хорошо справляется с множеством задач по формированию XML документов, включая формирование атрибутов для элементов. Если Вам нужно самим сформировать структуру XML данных, то рекомендовано использовать именно этот режим.

Несколько полезных параметров конструкции FOR XML:

TYPE – возвращает сформированные XML данные с типом XML, если параметр TYPE не указан, данные возвращаются с типом nvarchar(max). Параметр необходим в тех случаях, когда над итоговыми XML данными будут проводиться операции, характерные для XML данных, например, выполнение инструкций на языке XQuery;
ELEMENTS – если указать данный параметр, столбцы возвращаются в виде вложенных элементов;
ROOT – параметр добавляет к результирующему XML-документу один элемент верхнего уровня (корневой элемент), по умолчанию «root», однако название можно указать произвольное.

In [185]:

```
cur = conn.cursor()
sql = '''
truncate table TestTable;
INSERT INTO TestTable VALUES
    (1,'Клавиатура', 100),
    (1, 'Мышь', 50),
    (2, 'Телефон', 300);

truncate table TestTable2;
INSERT INTO TestTable2 VALUES
    ('Комплектующие компьютера'),
    ('Мобильные устройства')
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [186]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[186]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

Пример 1

В этом примере используется режим **RAW**, а также параметр **TYPE**, как видите, мы просто после основного запроса SELECT написали данную конструкцию. Запрос нам вернет XML данные, где каждая строка таблицы TestTable будет элементом row, а все столбцы отображены в виде атрибутов этого элемента.

In [187]:

```
sql = '''
with
xmlr as(
    select(
        SELECT ProductId, ProductName, Price
        FROM TestTable
        ORDER BY ProductId
        FOR XML RAW, TYPE
    )as XMLData
)
select * from xmlr

'''
select(sql)
```

Out[187]:

	XMLData
0	<row ProductId="1" ProductName="Клавиатура" Pr...

In [188]:

```
xml(sql, 'XMLData', False)
```

```
<html>
 <body>
  <row price="100.0000" productid="1" productname="Клавиатура">
  </row>
  <row price="50.0000" productid="2" productname="Мышь">
  </row>
  <row price="300.0000" productid="3" productname="Телефон">
  </row>
 </body>
</html>
```

Пример 2

В данном случае мы также используем режим **RAW**, только мы изменили название каждого элемента на '*Product*', для этого указали соответствующий параметр, добавили параметр **ELEMENTS**, для того чтобы столбцы были отображены в виде вложенных элементов, а также добавили корневой элемент '*Products*' с помощью параметра **ROOT**.

In [189]:

```
sql = '''
with
xmlr as(
    select(
        SELECT ProductId, ProductName, Price
        FROM TestTable
        ORDER BY ProductId
        FOR XML RAW ('Product'), TYPE, ELEMENTS, ROOT ('Products')
    )as XMLData
)
select * from xmlr

'''
xml(sql, 'XMLData')
```

```
<?xml version="1.0" encoding="utf-8"?>
<Products>
 <Product>
  <ProductId>
   1
  </ProductId>
  <ProductName>
   Клавиатура
  </ProductName>
  <Price>
   100.0000
  </Price>
 </Product>
 <Product>
  <ProductId>
   2
  </ProductId>
  <ProductName>
   Мышь
  </ProductName>
  <Price>
   50.0000
  </Price>
 </Product>
 <Product>
  <ProductId>
   3
  </ProductId>
  <ProductName>
   Телефон
  </ProductName>
  <Price>
   300.0000
  </Price>
 </Product>
</Products>
```

Пример 3

In [190]:

```
sql = '''
SELECT TestTable.ProductId, TestTable.ProductName, TestTable2.CategoryName, TestTable.Price
FROM TestTable
LEFT JOIN TestTable2 ON TestTable.CategoryId = TestTable2.CategoryId
ORDER BY TestTable.ProductId
'''
select(sql)
```

Out[190]:

	ProductId	ProductName	CategoryName	Price
0	1	Клавиатура	Комплектующие компьютера	100.0
1	2	Мышь	Комплектующие компьютера	50.0
2	3	Телефон	Мобильные устройства	300.0

Сейчас мы использовали режим **AUTO**, при этом мы модифицировали запрос, добавили в него объединение для наглядности, в данном режиме нам вернулись XML данные, где записи таблицы TestTable представлены в виде элементов, ее столбцы в виде атрибутов, а соответствующие записи (на основе объединения) таблицы TestTable2 в виде вложенных элементов с атрибутами.

In [191]:

```
sql = f'''
with
xmlr as(
    select({sql}
        FOR XML AUTO, TYPE
    )as XMLData
)
select * from xmlr

'''
xml(sql, 'XMLData', False)
```

```
<html>
 <body>
  <testtable price="100.0000" productid="1" productname="Клавиатура">
   <testtable2 categoryname="Комплектующие компьютера">
   </testtable2>
  </testtable>
  <testtable price="50.0000" productid="2" productname="Мышь">
   <testtable2 categoryname="Комплектующие компьютера">
   </testtable2>
  </testtable>
  <testtable price="300.0000" productid="3" productname="Телефон">
   <testtable2 categoryname="Мобильные устройства">
   </testtable2>
  </testtable>
 </body>
</html>
```

Пример 4

Пример 4

In [192]:

```
sql = '''
SELECT TestTable.ProductId, TestTable.ProductName, TestTable2.CategoryName, TestTable.Price
FROM TestTable
LEFT JOIN TestTable2 ON TestTable.CategoryId = TestTable2.CategoryId
ORDER BY TestTable.ProductId
'''
select(sql)
```

Out[192]:

	ProductId	ProductName	CategoryName	Price
0	1	Клавиатура	Комплектующие компьютера	100.0
1	2	Мышь	Комплектующие компьютера	50.0
2	3	Телефон	Мобильные устройства	300.0

Как видите, если мы добавим параметр **ELEMENTS**, то данные сформируются уже в виде элементов без атрибутов.

In [193]:

```
sql = f'''
with
xmlr as(
    select({sql}
    FOR XML AUTO, TYPE, ELEMENTS
    )as XMLData
)
select * from xmlr
'''
xml(sql, 'XMLData', False)
```

```
<html>
 <body>
  <testtable>
   <productid>
    1
   </productid>
   <productname>
    Клавиатура
   </productname>
   <price>
    100.0000
   </price>
   <testtable2>
    <categoryname>
     Комплектующие компьютера
    </categoryname>
   </testtable2>
  </testtable>
  <testtable>
   <productid>
    2
   </productid>
   <productname>
    Мышь
   </productname>
   <price>
    50.0000
   </price>
   <testtable2>
    <categoryname>
     Комплектующие компьютера
    </categoryname>
   </testtable2>
  </testtable>
  <testtable>
   <productid>
    3
   </productid>
   <productname>
    Телефон
   </productname>
   <price>
    300.0000
   </price>
   <testtable2>
    <categoryname>
     Мобильные устройства
    </categoryname>
```

```
   </testtable2>
  </testtable>
 </body>
</html>
```

Пример 5

In [194]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[194]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

В этом примере мы уже используем расширенный режим **PATH**, при котором мы можем сами указывать, что у нас будет элементами, а что атрибутами.
В запросе SELECT, при определении списка выборки, для столбца ProductId мы задали псевдоним Id с помощью инструкции "*@Id*", SQL сервер расценивает такую запись как инструкцию создания атрибута, таким образом, мы указали в результирующем XML документе, что у нас значение ProductId будет атрибутом элемента, а его название Id. Элементом у нас также выступает каждая строка таблицы TestTable, название элементов мы переопределили с помощью параметра, указав значение ('*Product*'), а с помощью параметра **ROOT** мы указали корневой элемент.

In [195]:

```
sql = f'''
with
xmlr as(
    select(
        SELECT ProductId AS "@Id", ProductName, Price
        FROM TestTable
        ORDER BY ProductId
        FOR XML PATH ('Product'), TYPE, ROOT ('Products')
    )as XMLData
)
select * from xmlr
'''
xml(sql, 'XMLData', False)
```

```
<html>
 <body>
  <products>
   <product id="1">
    <productname>
     Клавиатура
    </productname>
    <price>
     100.0000
    </price>
   </product>
   <product id="2">
    <productname>
     Мышь
    </productname>
    <price>
     50.0000
    </price>
   </product>
   <product id="3">
    <productname>
     Телефон
    </productname>
    <price>
     300.0000
    </price>
   </product>
  </products>
 </body>
</html>
```

167 Конструкция OPENXML

В следующем примере мы сформируем XML документ, а затем извлечем из него данные в табличном виде.

In [196]:

```
sql = '''
--Объявляем переменные
DECLARE @XML_Doc XML;
DECLARE @XML_Doc_Handle INT;

--Формируем XML документ
/*
<Products>
  <Product Id="1">
    <ProductName>Клавиатура</ProductName>
    <Price>100.0000</Price>
  </Product>
  <Product Id="2">
    <ProductName>Мышь</ProductName>
    <Price>60.0000</Price>
  </Product>
  ...
  */
SET @XML_Doc = (
    SELECT ProductId AS "@Id", ProductName, Price
    FROM TestTable
    ORDER BY ProductId
    FOR XML PATH ('Product'), TYPE, ROOT ('Products')
    );

--Подготавливаем XML документ
EXEC sp_xml_preparedocument @XML_Doc_Handle OUTPUT, @XML_Doc;

--Извлекаем данные из XML документа
SELECT *
FROM OPENXML (@XML_Doc_Handle, '/Products/Product', 2)
WITH (
    ProductId INT '@Id',
    ProductName VARCHAR(100),
    Price MONEY
    );

--Удаляем дескриптор XML документа
EXEC sp_xml_removedocument @XML_Doc_Handle;

'''
select(sql)
```

Out[196]:

	ProductId	ProductName	Price
0	1	Клавиатура	100.0
1	2	Мышь	50.0
2	3	Телефон	300.0

XML документ строкой (1):

In [197]:

```
sql = '''
--Объявляем переменные
DECLARE @XML_Str VARCHAR(max);
DECLARE @XML_Doc XML;
DECLARE @XML_Doc_Handle INT;

--Формируем XML документ
SET @XML_Str = '<Products>
  <Product Id="1">
    <ProductName>Клавиатура</ProductName>
    <Price>100.0000</Price>
  </Product>
  <Product Id="2">
    <ProductName>Мышь</ProductName>
    <Price>50.0000</Price>
  </Product>
  <Product Id="3">
    <ProductName>Телефон</ProductName>
    <Price>300.0000</Price>
  </Product>
</Products>'

SET @XML_Doc = @XML_Str;

--Подготавливаем XML документ
EXEC sp_xml_preparedocument @XML_Doc_Handle OUTPUT, @XML_Doc;

--Извлекаем данные из XML документа
SELECT *
FROM OPENXML (@XML_Doc_Handle, '/Products/Product', 2)
WITH (
    ProductId INT '@Id',
    ProductName VARCHAR(100),
    Price MONEY
    );

--Удаляем дескриптор XML документа
EXEC sp_xml_removedocument @XML_Doc_Handle;
'''
select(sql)
```

Out[197]:

	ProductId	ProductName	Price
0	1	Клавиатура	100.0
1	2	Мышь	50.0
2	3	Телефон	300.0

XML документ строкой (2):

In [198]:

```
xml = """<Products>
  <Product Id="1">
    <ProductName>Клавиатура</ProductName>
    <Price>100.0000</Price>
  </Product>
  <Product Id="2">
    <ProductName>Мышь</ProductName>
    <Price>50.0000</Price>
  </Product>
  <Product Id="3">
    <ProductName>Телефон</ProductName>
    <Price>300.0000</Price>
  </Product>
</Products>"""
```

In [199]:

```
sql = f'''
--Объявляем переменные
DECLARE @XML_Str VARCHAR(max);
DECLARE @XML_Doc XML;
DECLARE @XML_Doc_Handle INT;

--Формируем XML документ
SET @XML_Str = '{xml}';
SET @XML_Doc = @XML_Str;

--Подготавливаем XML документ
EXEC sp_xml_preparedocument @XML_Doc_Handle OUTPUT, @XML_Doc;

--Извлекаем данные из XML документа
SELECT *
FROM OPENXML (@XML_Doc_Handle, '/Products/Product', 2)
WITH (
    ProductId INT '@Id',
    ProductName VARCHAR(100),
    Price MONEY
    );

--Удаляем дескриптор XML документа
EXEC sp_xml_removedocument @XML_Doc_Handle;

'''
select(sql)
```

Out[199]:

	ProductId	ProductName	Price
0	1	Клавиатура	100.0
1	2	Мышь	50.0
2	3	Телефон	300.0

XML документ строкой (2):

In [200]:

```
xml = """<data>
    <student>
        <name>Alice</name>
        <major>Computer Science</major>
        <age>20</age>
    </student>
    <student>
        <name>Bob</name>
        <major>Philosophy</major>
        <age>22</age>
    </student>
    <student>
        <name>Mary</name>
        <major>Biology</major>
        <age>21</age>
    </student>
</data>"""
```

In [201]:

```
sql = f'''
--Объявляем переменные
DECLARE @XML_Str VARCHAR(max);
DECLARE @XML_Doc XML;
DECLARE @XML_Doc_Handle INT;

--Формируем XML документ
SET @XML_Str = '{xml}';
SET @XML_Doc = @XML_Str;

--Подготавливаем XML документ
EXEC sp_xml_preparedocument @XML_Doc_Handle OUTPUT, @XML_Doc;

--Извлекаем данные из XML документа
SELECT *
FROM OPENXML (@XML_Doc_Handle, '/data/student', 2)
WITH (
    name VARCHAR(100),
    major VARCHAR(100),
    age INT
    );

--Удаляем дескриптор XML документа
EXEC sp_xml_removedocument @XML_Doc_Handle;

'''
select(sql)
```

Out[201]:

	name	major	age
0	Alice	Computer Science	20
1	Bob	Philosophy	22
2	Mary	Biology	21

Reading and Writing XML with Pandas – Finxter (https://blog.finxter.com/reading-and-writing-xml-with-pandas/)

In [202]:

```
xml = """<?xml version='1.0' encoding='utf-8'?>
<data>
    <student>
        <name>Alice</name>
        <major>Computer Science</major>
        <age>20</age>
    </student>
    <student>
        <name>Bob</name>
        <major>Philosophy</major>
        <age>22</age>
    </student>
    <student>
        <name>Mary</name>
        <major>Biology</major>
        <age>21</age>
    </student>
</data>"""
```

In [203]:

```
df = pd.read_xml(xml)
print(df)
```

```
    name             major  age
0  Alice  Computer Science   20
1    Bob        Philosophy   22
2   Mary           Biology   21
```

In [204]:

```
print(df.to_xml())
```

```
<?xml version='1.0' encoding='utf-8'?>
<data>
  <row>
    <index>0</index>
    <name>Alice</name>
    <major>Computer Science</major>
    <age>20</age>
  </row>
  <row>
    <index>1</index>
    <name>Bob</name>
    <major>Philosophy</major>
    <age>22</age>
  </row>
  <row>
    <index>2</index>
    <name>Mary</name>
    <major>Biology</major>
    <age>21</age>
  </row>
</data>
```

Чтение и запись файлов XML с помощью Pandas (https://rukovodstvo.net/posts/id_659/)

Как перебирать строки в фрейме данных Pandas (https://dev-gang.ru/article/kak-perebirat-stroki-v-freime-dannyh-pandas-6kv1i4ayi8/)

16.1-WITH

In [205]:

```
cur = conn.cursor()
sql = '''
truncate table TestTable;
INSERT INTO TestTable VALUES
    (1,'Клавиатура', 100),
    (1, 'Мышь', 50),
    (2, 'Телефон', 300);

truncate table TestTable2;
INSERT INTO TestTable2 VALUES
    ('Комплектующие компьютера'),
    ('Мобильные устройства')
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [206]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[206]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [207]:

```
sql = '''SELECT * FROM TestTable2'''
select(sql)
```

Out[207]:

	CategoryId	CategoryName
0	1	Комплектующие компьютера
1	2	Мобильные устройства

170 Конструкция WITH – обобщенное табличное выражение

In [208]:

```
sql = '''
--Пишем CTE с названием TestCTE
WITH
TestCTE (ProductId, ProductName, Price) AS
(
    --Запрос, который возвращает определѐнные логичные данные
    SELECT ProductId, ProductName, Price
    FROM TestTable
    WHERE CategoryId = 1
)
--Запрос, в котором мы можем использовать CTE
SELECT * FROM TestCTE
'''
select(sql)
```

Out[208]:

	ProductId	ProductName	Price
0	1	Клавиатура	100.0
1	2	Мышь	50.0

Перечисление столбцов после названия CTE (в нашем случае после TestCTE) можно и опустить:

In [209]:

```
sql = '''
--Пишем CTE с названием TestCTE
WITH
TestCTE AS
(
    --Запрос, который возвращает определѐнные логичные данные
    SELECT ProductId, ProductName, Price
    FROM TestTable
    WHERE CategoryId = 1
)
--Запрос, в котором мы можем использовать CTE
SELECT * FROM TestCTE
'''
select(sql)
```

Out[209]:

	ProductId	ProductName	Price
0	1	Клавиатура	100.0
1	2	Мышь	50.0

Несколько именованных запросов:

In [210]:

```
sql = '''
WITH
TestCTE1 AS (
    --Представьте, что здесь запрос со своей сложной логикой
    SELECT ProductId, CategoryId, ProductName, Price
    FROM TestTable
),
TestCTE2 AS (
    --Здесь также сложный запрос
    SELECT CategoryId, CategoryName
    FROM TestTable2
)
--Работаем с результирующими наборами данных двух запросов
SELECT T1.ProductName, T2.CategoryName, T1.Price
FROM TestCTE1 T1
LEFT JOIN TestCTE2 T2 ON T1.CategoryId = T2.CategoryId
WHERE T1.CategoryId = 1
'''
select(sql)
```

Out[210]:

	ProductName	CategoryName	Price
0	Клавиатура	Комплектующие компьютера	100.0
1	Мышь	Комплектующие компьютера	50.0

Без WITH:

In [211]:

```
sql = '''
SELECT T1.ProductName, T2.CategoryName, T1.Price
FROM (
    SELECT ProductId, CategoryId, ProductName, Price
    FROM TestTable
    ) T1
LEFT JOIN (
    SELECT CategoryId, CategoryName
    FROM TestTable2
    ) T2 ON T1.CategoryId = T2.CategoryId
WHERE T1.CategoryId = 1
'''
select(sql)
```

Out[211]:

	ProductName	CategoryName	Price
0	Клавиатура	Комплектующие компьютера	100.0
1	Мышь	Комплектующие компьютера	50.0

16.2-OVER

In [212]:

```
cur = conn.cursor()
sql = '''
truncate table TestTable;
INSERT INTO TestTable VALUES
    (1,'Клавиатура', 100),
    (1, 'Мышь', 50),
    (2, 'Телефон', 300);

truncate table TestTable2;
INSERT INTO TestTable2 VALUES
    ('Комплектующие компьютера'),
    ('Мобильные устройства')
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [213]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[213]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [214]:

```
sql = '''SELECT * FROM TestTable2'''
select(sql)
```

Out[214]:

	CategoryId	CategoryName
0	1	Комплектующие компьютера
1	2	Мобильные устройства

174 Агрегатные оконные функции

In [215]:

```
sql = '''
SELECT ProductId, ProductName, CategoryId, Price,
    SUM(Price) OVER (PARTITION BY CategoryId) AS [SUM],
    AVG(Price) OVER (PARTITION BY CategoryId) AS [AVG],
    COUNT(Price) OVER (PARTITION BY CategoryId) AS [COUNT],
    MIN(Price) OVER (PARTITION BY CategoryId) AS [MIN],
    MAX(Price) OVER (PARTITION BY CategoryId) AS [MAX]
FROM TestTable
'''
select(sql)
```

Out[215]:

	ProductId	ProductName	CategoryId	Price	SUM	AVG	COUNT	MIN	MAX
0	1	Клавиатура	1	100.0	150.0	75.0	2	50.0	100.0
1	2	Мышь	1	50.0	150.0	75.0	2	50.0	100.0
2	3	Телефон	2	300.0	300.0	300.0	1	300.0	300.0

175 Ранжирующие оконные функции

ROW_NUMBER – функция нумерации строк в секции результирующего набора данных, которая возвращает просто номер строки.

In [216]:

```
sql = '''
SELECT
    ROW_NUMBER () OVER (PARTITION BY CategoryId ORDER BY ProductID) AS [ROW_NUMBER]
    ,*
FROM TestTable
'''
select(sql)
```

Out[216]:

	ROW_NUMBER	ProductId	CategoryId	ProductName	Price
0	1	1	1	Клавиатура	100.0
1	2	2	1	Мышь	50.0
2	1	3	2	Телефон	300.0

RANK – ранжирующая функция, которая возвращает ранг каждой строки. В данном случае, в отличие от ROW_NUMBER (), здесь уже идет анализ значений. В случае если в столбце, по которому происходит сортировка, есть одинаковые значения, для них возвращается также одинаковый ранг (*следующее значение ранга в этом случае пропускается*).

In [217]:

```
sql = '''
SELECT
    RANK () OVER (PARTITION BY CategoryId ORDER BY Price) AS [RANK]
    ,*
FROM TestTable
'''
select(sql)
```

Out[217]:

	RANK	ProductId	CategoryId	ProductName	Price
0	1	2	1	Мышь	50.0
1	2	1	1	Клавиатура	100.0
2	1	3	2	Телефон	300.0

DENSE_RANK - ранжирующая функция, которая возвращает ранг каждой строки, но в отличие от RANK в случае нахождения одинаковых значений возвращает ранг без пропуска следующего.

In [218]:

```
sql = '''
SELECT
    DENSE_RANK () OVER (PARTITION BY CategoryId ORDER BY Price) AS [DENSE_RANK]
    ,*
FROM TestTable
'''
select(sql)
```

Out[218]:

	DENSE_RANK	ProductId	CategoryId	ProductName	Price
0	1	2	1	Мышь	50.0
1	2	1	1	Клавиатура	100.0
2	1	3	2	Телефон	300.0

NTILE – ранжирующая оконная функция, которая делит результирующий набор на группы по определенному столбцу. Количество групп передается в качестве параметра. В случае если в группах получается не одинаковое количество строк, то в самой первой группе будет наибольшее количество, например, в случае если в источнике 10 строк, при этом мы поделим результирующий набор на три группы, то в первой будет 4 строки, а во второй и третьей по 3.

In [219]:

```
sql = '''
SELECT
    NTILE (3) OVER (ORDER BY ProductId) AS [NTILE]
    ,*
FROM TestTable
'''
select(sql)
```

Out[219]:

	NTILE	ProductId	CategoryId	ProductName	Price
0	1	1	1	Клавиатура	100.0
1	2	2	1	Мышь	50.0
2	3	3	2	Телефон	300.0

176 Оконные функции смещения

LEAD – функция обращается к данным из следующей строки набора данных. Ее можно использовать, например, для того чтобы сравнить текущее значение строки со следующим. Имеет три параметра: столбец, значение которого необходимо вернуть (*обязательный параметр*), количество строк для смещения (*по умолчанию 1*), значение, которое необходимо вернуть, если после смещения возвращается значение NULL;

LAG – функция обращается к данным из предыдущей строки набора данных. В данном случае функцию можно использовать для того, чтобы сравнить текущее значение строки с предыдущим. Имеет три параметра: столбец, значение, которого необходимо вернуть (*обязательный параметр*), количество строк для смещения (*по умолчанию 1*), значение, которое необходимо вернуть, если после смещения возвращается значение NULL;

FIRST_VALUE - функция возвращает первое значение из набора данных, в качестве параметра принимает столбец, значение которого необходимо вернуть;

LAST_VALUE - функция возвращает последнее значение из набора данных, в качестве параметра принимает столбец, значение которого необходимо вернуть.

In [220]:

```
sql = '''
SELECT *
FROM TestTable
ORDER BY ProductId
'''
select(sql)
```

Out[220]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [221]:

```
sql = '''
SELECT ProductId, ProductName, CategoryId, Price,
    LEAD(ProductId) OVER (PARTITION BY CategoryId ORDER BY ProductId) AS [LEAD],
    LEAD(ProductId, 2, 0) OVER (PARTITION BY CategoryId ORDER BY ProductId) AS [LEAD2],
    LAG(ProductId) OVER (PARTITION BY CategoryId ORDER BY ProductId) AS [LAG],
    LAG(ProductId, 2, 0) OVER (PARTITION BY CategoryId ORDER BY ProductId) AS [LAG2],
    FIRST_VALUE(ProductId) OVER (
                                PARTITION BY CategoryId
                                ORDER BY ProductId
                                ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
                                ) AS [FIRST_VALUE],
    LAST_VALUE (ProductId) OVER (
                                PARTITION BY CategoryId
                                ORDER BY ProductId
                                ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
                                ) AS [LAST_VALUE],
    LEAD(ProductId, 2) OVER (ORDER BY ProductId) AS [LEAD_2],
    LAG(ProductId, 2, 0) OVER (ORDER BY ProductId) AS [LAG_2]
FROM TestTable
ORDER BY ProductId
'''
select(sql)
```

Out[221]:

	ProductId	ProductName	CategoryId	Price	LEAD	LEAD2	LAG	LAG2	FIRST_VALUE	LAS
0	1	Клавиатура	1	100.0	2.0	0	NaN	0	1	
1	2	Мышь	1	50.0	NaN	0	1.0	0	1	
2	3	Телефон	2	300.0	NaN	0	NaN	0	3	

178 Аналитические оконные функции

CUME_DIST - вычисляет и возвращает интегральное распределение значений в наборе данных. Иными словами, она определяет относительное положение значения в наборе;

PERCENT_RANK - вычисляет и возвращает относительный ранг строки в наборе данных;

PERCENTILE_CONT - вычисляет процентиль на основе постоянного распределения значения столбца. В качестве параметра принимает процентиль, который необходимо вычислить;

PERCENTILE_DISC - вычисляет определенный процентиль для отсортированных значений в наборе данных. В качестве параметра принимает процентиль, который необходимо вычислить.

In [222]:

```
sql = '''
SELECT ProductId, ProductName, CategoryId, Price,
    CUME_DIST() OVER (PARTITION BY CategoryId ORDER BY Price) AS [CUME_DIST],
    PERCENT_RANK() OVER (PARTITION BY CategoryId ORDER BY Price) AS [PERCENT_RANK],
    PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY ProductId)
        OVER(PARTITION BY CategoryId) AS [PERCENTILE_DISC],
    PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY ProductId)
        OVER(PARTITION BY CategoryId) AS [PERCENTILE_CONT]
FROM TestTable
'''
select(sql)
```

Out[222]:

	ProductId	ProductName	CategoryId	Price	CUME_DIST	PERCENT_RANK	PERCENTILE_DIS
0	1	Клавиатура	1	100.0	1.0	1.0	
1	2	Мышь	1	50.0	0.5	0.0	
2	3	Телефон	2	300.0	1.0	0.0	

In [223]:

```
sql = '''
SELECT *
FROM TestTable
ORDER BY Price
'''
select(sql)
```

Out[223]:

	ProductId	CategoryId	ProductName	Price
0	2	1	Мышь	50.0
1	1	1	Клавиатура	100.0
2	3	2	Телефон	300.0

In [224]:

```
sql = '''
SELECT *
FROM TestTable
ORDER BY ProductId
'''
select(sql)
```

Out[224]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

16.3-PIVOT

In [225]:

```
cur = conn.cursor()
sql = '''
truncate table TestTable;
INSERT INTO TestTable VALUES
    (1,'Клавиатура', 100),
    (1, 'Мышь', 50),
    (2, 'Телефон', 300);

truncate table TestTable2;
INSERT INTO TestTable2 VALUES
    ('Комплектующие компьютера'),
    ('Мобильные устройства')
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [226]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[226]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [227]:

```
sql = '''SELECT * FROM TestTable2'''
select(sql)
```

Out[227]:

	CategoryId	CategoryName
0	1	Комплектующие компьютера
1	2	Мобильные устройства

178 Операторы PIVOT

Можно реализовать хранимую процедуру, с помощью которой можно формировать динамические запросы PIVOT (пример реализации можете найти на моем сайте info-comp.ru (http://info-comp.ru/obucheniest/631-dynamic-pivot-in-t-sql.html)).

Обычная группировка:

In [228]:

```
sql = '''
SELECT T2.CategoryName,
    AVG(T1.Price) AS AvgPrice
FROM TestTable T1
LEFT JOIN TestTable2 T2 ON T1.CategoryId = T2.CategoryId
GROUP BY T2.CategoryName
'''
select(sql)
```

Out[228]:

	CategoryName	AvgPrice
0	Комплектующие компьютера	75.0
1	Мобильные устройства	300.0

Группировка с использованием PIVOT :

In [229]:

```
sql = '''
SELECT T1.Price, T2.CategoryName
FROM TestTable T1
LEFT JOIN TestTable2 T2 ON T1.CategoryId = T2.CategoryId
'''
select(sql)
```

Out[229]:

	Price	CategoryName
0	100.0	Комплектующие компьютера
1	50.0	Комплектующие компьютера
2	300.0	Мобильные устройства

In [230]:

```
sql = '''
SELECT 'Средняя цена' AS AvgPrice, [Комплектующие компьютера], [Мобильные устройства]
FROM (
    SELECT T1.Price, T2.CategoryName
    FROM TestTable T1
    LEFT JOIN TestTable2 T2 ON T1.CategoryId = T2.CategoryId
    ) AS SourceTable
PIVOT (
    AVG(Price) FOR CategoryName IN ([Комплектующие компьютера],[Мобильные устройства])
) AS PivotTable;
'''
select(sql)
```

Out[230]:

	AvgPrice	Комплектующие компьютера	Мобильные устройства
0	Средняя цена	75.0	300.0

Где,

[Комплектующие компьютера], [Мобильные устройства] – это значения в столбце CategoryName, которые мы заранее должны знать;

SourceTable – псевдоним выражения, в котором мы указываем исходный источник данных, например, вложенный запрос;

PIVOT – вызов оператора PIVOT;

AVG – агрегатная функция, в которую мы передаем столбец для анализа, в нашем случае Price;

FOR – с помощью данного ключевого слова мы указываем столбец, содержащий значения, которые будут выступать именами столбцов, в нашем случае CategoryName;

IN – оператор, с помощью которого мы перечисляем значения в столбце CategoryName;

PivotTable - псевдоним сводной таблицы, его необходимо указывать обязательно.

179 Операторы UNPIVOT

In [231]:

```
cur = conn.cursor()
sql = '''
--Создаѐм временную таблицу с помощью SELECT INTO
SELECT
    'Город' AS NamePar,
    'Москва' AS Column1,
    'Калуга' AS Column2,
    'Тамбов' AS Column3
INTO #TestUnpivot
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [232]:

```
sql = '''
SELECT * FROM #TestUnpivot
'''
select(sql)
```

Out[232]:

	NamePar	Column1	Column2	Column3
0	Город	Москва	Калуга	Тамбов

In [233]:

```
sql = '''
--Применяем оператор UNPIVOT
SELECT NamePar, ColumnName, CityNameValue
FROM #TestUnpivot
UNPIVOT(
    CityNameValue FOR ColumnName IN ([Column1], [Column2], [Column3])
)AS UnpivotTable
'''
select(sql)
```

Out[233]:

	NamePar	ColumnName	CityNameValue
0	Город	Column1	Москва
1	Город	Column2	Калуга
2	Город	Column3	Тамбов

Где,

#TestUnpivot – таблица источник, в нашем случае временная таблица;

CityNameValue – псевдоним столбца, который будет содержать значения наших столбцов;

FOR – ключевое слово, с помощью которого мы указываем псевдоним для столбца, который будет содержать имена наших столбцов;

ColumnName - псевдоним столбца, который будет содержать имена наших столбцов;

IN – ключевое слово для указания имен столбцов.

16.4-ROLLUP

In [234]:

```
cur = conn.cursor()
sql = '''
truncate table TestTable;
INSERT INTO TestTable VALUES
    (1,'Клавиатура', 100),
    (1, 'Мышь', 50),
    (2, 'Телефон', 300);

truncate table TestTable2;
INSERT INTO TestTable2 VALUES
    ('Комплектующие компьютера'),
    ('Мобильные устройства')
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [235]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[235]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [236]:

```
sql = '''SELECT * FROM TestTable2'''
select(sql)
```

Out[236]:

	CategoryId	CategoryName
0	1	Комплектующие компьютера
1	2	Мобильные устройства

181 Аналитический оператор ROLLUP

Ооператор, который формирует промежуточные итоги для каждого указанного элемента и общий итог.

In [237]:

```
sql = '''
--Без использования ROLLUP
SELECT CategoryId,
    SUM(Price) AS Summa
FROM TestTable
GROUP BY CategoryId
'''
select(sql)
```

Out[237]:

	CategoryId	Summa
0	1	150.0
1	2	300.0

In [238]:

```
sql = '''
--С использованием ROLLUP
SELECT CategoryId,
    SUM(Price) AS Summa
FROM TestTable
GROUP BY ROLLUP (CategoryId)
'''
select(sql)
```

Out[238]:

	CategoryId	Summa
0	1.0	150.0
1	2.0	300.0
2	NaN	450.0

182 Аналитический оператор CUBE

In [239]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[239]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [240]:

```
sql = '''
--С использованием ROLLUP
SELECT ProductName, CategoryId,
    SUM(Price) AS Summa
FROM TestTable
GROUP BY ROLLUP (CategoryId, ProductName)
'''
select(sql)
```

Out[240]:

	ProductName	CategoryId	Summa
0	Клавиатура	1.0	100.0
1	Мышь	1.0	50.0
2	None	1.0	150.0
3	Телефон	2.0	300.0
4	None	2.0	300.0
5	None	NaN	450.0

Оператор, который формирует результаты для всех возможных перекрестных вычислений. Отличие от ROLLUP состоит в том, что, если мы укажем несколько столбцов для группировки, ROLLUP выведет строки подытогов высокого уровня, т.е. для каждого уникального сочетания перечисленных столбцов, CUBE выведет подытоги для всех возможных сочетаний этих столбцов.

In [241]:

```
sql = '''
--С использованием CUBE
SELECT ProductName, CategoryId,
    SUM(Price) AS Summa
FROM TestTable
GROUP BY CUBE (CategoryId, ProductName)
'''
select(sql)
```

Out[241]:

	ProductName	CategoryId	Summa
0	Клавиатура	1.0	100.0
1	Клавиатура	NaN	100.0
2	Мышь	1.0	50.0
3	Мышь	NaN	50.0
4	Телефон	2.0	300.0
5	Телефон	NaN	300.0
6	None	NaN	450.0
7	None	1.0	150.0
8	None	2.0	300.0

183 Аналитический оператор GROUPING SETS

In [242]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[242]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [243]:

```
sql = '''
--С использованием UNION ALL
SELECT ProductName, NULL AS CategoryId,
    SUM(Price) AS Summa
FROM TestTable
GROUP BY ProductName
UNION ALL
SELECT NULL AS ProductName, CategoryId,
    SUM(Price) AS Summa
FROM TestTable
GROUP BY CategoryId
'''
select(sql)
```

Out[243]:

	ProductName	CategoryId	Summa
0	Клавиатура	NaN	100.0
1	Мышь	NaN	50.0
2	Телефон	NaN	300.0
3	None	1.0	150.0
4	None	2.0	300.0

Оператор, который формирует результаты нескольких группировок в один набор данных, другими словами, в результирующий набор попадают только строки по группировкам. Данный оператор эквивалентен конструкции UNION ALL, если в нем указать запросы просто с GROUP BY по каждому указанному столбцу.

In [244]:

```
sql = '''
SELECT ProductName, CategoryId,
    SUM(Price) AS Summa
FROM TestTable
GROUP BY GROUPING SETS (CategoryId, ProductName)
'''
select(sql)
```

Out[244]:

	ProductName	CategoryId	Summa
0	Клавиатура	NaN	100.0
1	Мышь	NaN	50.0
2	Телефон	NaN	300.0
3	None	1.0	150.0
4	None	2.0	300.0

16.5-APPLY

In [245]:

```
cur = conn.cursor()
sql = '''
truncate table TestTable;
INSERT INTO TestTable VALUES
    (1,'Клавиатура', 100),
    (1, 'Мышь', 50),
    (2, 'Телефон', 300);

truncate table TestTable2;
INSERT INTO TestTable2 VALUES
    ('Комплектующие компьютера'),
    ('Мобильные устройства')
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [246]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[246]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [247]:

```
sql = '''SELECT * FROM TestTable2'''
select(sql)
```

Out[247]:

	CategoryId	CategoryName
0	1	Комплектующие компьютера
1	2	Мобильные устройства

```
CREATE FUNCTION FT_TestFunction (
    @CategoryId INT --Объявление входящих параметров
)
RETURNS TABLE
AS
RETURN(
    --Получение всех товаров в определѐнной категории
    SELECT ProductId,
        ProductName,
        Price,
        CategoryId
    FROM TestTable
    WHERE CategoryId = @CategoryId
```

```
    )
```

185 Оператор APPLY

In [248]:

```
sql = '''
SELECT * FROM FT_TestFunction(1)
'''
select(sql)
```

Out[248]:

	ProductId	ProductName	Price	CategoryId
0	1	Клавиатура	100.0	1
1	2	Мышь	50.0	1

In [249]:

```
sql = '''SELECT * FROM TestTable2'''
select(sql)
```

Out[249]:

	CategoryId	CategoryName
0	1	Комплектующие компьютера
1	2	Мобильные устройства

Существует два типа оператора APPLY:
CROSS APPLY - возвращает только строки из внешней таблицы, которые создает табличная функция;
OUTER APPLY - возвращает и строки, которые формирует табличная функция, и строки со значениями NULL в столбцах, созданные табличной функцией. Например, табличная функция может не возвращать никаких данных для определенных значений, CROSS APPLY в таких случаях подобные строки не выводит, а OUTER APPLY выводит (OUTER APPLY лично мне требуется редко).

В данном случае функция FT_TestFunction была вызвана для каждой строки таблицы TestTable2.

In [250]:

```
sql = '''
SELECT T2.CategoryName, FT1.*
FROM TestTable2 T2
CROSS APPLY FT_TestFunction(T2.CategoryId) AS FT1
'''
select(sql)
```

Out[250]:

	CategoryName	ProductId	ProductName	Price	CategoryId
0	Комплектующие компьютера	1	Клавиатура	100.0	1
1	Комплектующие компьютера	2	Мышь	50.0	1
2	Мобильные устройства	3	Телефон	300.0	2

In [251]:

```
cur = conn.cursor()
sql = '''
--Добавление новой строки в таблицу TestTable2
INSERT INTO TestTable2 VALUES ('Новая категория');
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [252]:

```
sql = '''SELECT * FROM TestTable2'''
select(sql)
```

Out[252]:

	CategoryId	CategoryName
0	1	Комплектующие компьютера
1	2	Мобильные устройства
2	3	Новая категория

In [253]:

```
sql = '''
SELECT T2.CategoryName, FT1.*
FROM TestTable2 T2
OUTER APPLY FT_TestFunction(T2.CategoryId) AS FT1
'''
select(sql)
```

Out[253]:

	CategoryName	ProductId	ProductName	Price	CategoryId
0	Комплектующие компьютера	1.0	Клавиатура	100.0	1.0
1	Комплектующие компьютера	2.0	Мышь	50.0	1.0
2	Мобильные устройства	3.0	Телефон	300.0	2.0
3	Новая категория	NaN	None	NaN	NaN

16.6-OPENDATA

In [254]:

```
cur = conn.cursor()
sql = '''
truncate table TestTable;
INSERT INTO TestTable VALUES
    (1,'Клавиатура', 100),
    (1, 'Мышь', 50),
    (2, 'Телефон', 300);

truncate table TestTable2;
INSERT INTO TestTable2 VALUES
    ('Комплектующие компьютера'),
    ('Мобильные устройства')
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [255]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[255]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [256]:

```
sql = '''SELECT * FROM TestTable2'''
select(sql)
```

Out[256]:

	CategoryId	CategoryName
0	1	Комплектующие компьютера
1	2	Мобильные устройства

```
sp_configure 'show advanced options', 1;
RECONFIGURE;
```

```
sp_configure 'Ad Hoc Distributed Queries', 1;
RECONFIGURE;
```

186 Получение данных из внешних источников

[Невозможно создать экземпляр поставщика OLE DB Microsoft.Jet.OLEDB.4.0 для связанного сервера null (https://www.stackfinder.ru/questions/36987636/cannot-create-an-instance-of-ole-db-provider-microsoft-jet-oledb-4-0-for-linked)](https://www.stackfinder.ru/questions/36987636/cannot-create-an-instance-of-ole-db-provider-microsoft-jet-oledb-4-0-for-linked)

```
--С помощью OPENDATASOURCE
SELECT * FROM OPENDATASOURCE('Microsoft.Jet.OLEDB.4.0',
    'Data Source=D:\TestExcel.xls; Extended Properties=Excel 8.0')...[Лист1$];
```

```
--С помощью OPENROWSET
SELECT * FROM OPENROWSET('Microsoft.Jet.OLEDB.4.0',
    'Excel 8.0; Database=D:\TestExcel.xls', [Лист1$]);
```

```
--С помощью OPENROWSET (с запросом)
SELECT * FROM OPENROWSET('Microsoft.Jet.OLEDB.4.0',
    'Excel 8.0; Database=D:\TestExcel.xls', 'SELECT ProductName, Price FROM [Лист1$]');
```

In [257]:

```
cur = conn.cursor()
sql = '''
SELECT * FROM OPENDATASOURCE('Microsoft.Jet.OLEDB.4.0',
    'Data Source=D:\TestExcel.xls; Extended Properties=Excel 8.0')...[Лист1$];
'''
cur.execute(sql)
conn.commit()
cur.close()
```

```
---------------------------------------------------------------------------
ProgrammingError                          Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5860\955509017.py in <module>
      4     'Data Source=D:\TestExcel.xls; Extended Properties=Excel 8.
0')...[Лист1$];
      5 '''
----> 6 cur.execute(sql)
      7 conn.commit()
      8 cur.close()

ProgrammingError: ('42000', '[42000] [Microsoft][ODBC SQL Server Driver][SQL
Server]Не удалось создать экземпляр поставщика OLE DB "Microsoft.Jet.OLEDB.
4.0" для связанного сервера "(null)". (7302) (SQLExecDirectW)')
```

OPENDATASOURCE (Transact-SQL) - SQL Server | Microsoft Learn (https://learn.microsoft.com/ru-ru/sql/t-sql/functions/opendatasource-transact-sql?view=sql-server-ver16)

Импорт данных из Excel в Microsoft SQL Server на языке T-SQL | Info-Comp.ru - IT-блог для начинающих (https://info-comp.ru/import-excel-in-ms-sql-server)

In [258]:

```
sql = """
SELECT @@VERSION;
"""
select(sql)
```

Out[258]:

0	Microsoft SQL Server 2019 (RTM-GDR) (KB5014356...

Шаг 1 – Проверяем наличие провайдера Microsoft.ACE.OLEDB.12.0 на SQL Server

```
EXEC sp_enum_oledb_providers;
```

Шаг 2 – Установка провайдера Microsoft.ACE.OLEDB.12.0 (32-bit)

Download Microsoft Access Database Engine 2010 Redistributable from Official Microsoft Download Center (https://www.microsoft.com/en-us/download/details.aspx?id=13255)

Выберите и скачайте файл, соответствующий архитектуре x86 (т.е. в названии без x64).

...

16.7-executesql

In [259]:

```
cur = conn.cursor()
sql = '''
truncate table TestTable;
INSERT INTO TestTable VALUES
    (1,'Клавиатура', 100),
    (1, 'Мышь', 50),
    (2, 'Телефон', 300);

truncate table TestTable2;
INSERT INTO TestTable2 VALUES
    ('Комплектующие компьютера'),
    ('Мобильные устройства')
'''
cur.execute(sql)
conn.commit()
cur.close()
```

In [260]:

```
sql = '''SELECT * FROM TestTable'''
select(sql)
```

Out[260]:

	ProductId	CategoryId	ProductName	Price
0	1	1	Клавиатура	100.0
1	2	1	Мышь	50.0
2	3	2	Телефон	300.0

In [261]:

```
sql = '''SELECT * FROM TestTable2'''
select(sql)
```

Out[261]:

	CategoryId	CategoryName
0	1	Комплектующие компьютера
1	2	Мобильные устройства

188 Выполнение динамических T-SQL инструкций

Пример с использованием команды EXEC.

```
--Объявляем переменные
DECLARE @SQL_QUERY VARCHAR(200), @Var1 INT;
--Присваиваем значение переменным
SET @Var1 = 1;
--Формируем SQL инструкцию
SET @SQL_QUERY = 'SELECT * FROM TestTable WHERE ProductID = ' + CAST(@Var1 AS
VARCHAR(10));
--Смотрим на итоговую строку
SELECT @SQL_QUERY AS [TEXT QUERY]
--Выполняем текстовую строку как SQL инструкцию
EXEC (@SQL_QUERY)
```

Пример с использованием хранимой процедуры sp_executesql.

```
--Объявляем переменные
DECLARE @SQL_QUERY NVARCHAR(200);
--Формируем SQL инструкцию
SELECT @SQL_QUERY = N'SELECT * FROM TestTable WHERE ProductID = @Var1;';
--Смотрим на итоговую строку
SELECT @SQL_QUERY AS [TEXT QUERY]
--Выполняем текстовую строку как SQL инструкцию
EXEC sp_executesql @SQL_QUERY,--Текст SQL инструкции
N'@Var1 AS INT', --Объявление переменных в процедуре
```

```
@Var1 = 1 --Передаем значение для переменных
```

17-admin

191 Администрирование сервера и базы данных

Безопасность

Создание имени входа

In [278]:

```
cur = conn.cursor()
sql = """
DROP LOGIN [TestLogin]
"""
cur.execute(sql)
conn.commit()
cur.close()
```

In [267]:

```
cur = conn.cursor()
sql = """
CREATE LOGIN [TestLogin]
    WITH PASSWORD='Pa$$w0rd',
    DEFAULT_DATABASE=[TestDB]
"""
cur.execute(sql)
conn.commit()
cur.close()
```

Назначение роли сервера

In [268]:

```
cur = conn.cursor()
sql = """
EXEC sp_addsrvrolemember
    @loginame = 'TestLogin',
    @rolename = 'sysadmin'
"""
cur.execute(sql)
conn.commit()
cur.close()
```

Создание пользователя базы данных и сопоставление с именем входа

In [277]:

```
cur = conn.cursor()
sql = """
DROP USER [TestUser]
"""
cur.execute(sql)
conn.commit()
cur.close()
```

In [271]:

```
cur = conn.cursor()
sql = """
CREATE USER [TestUser]
    FOR LOGIN [TestLogin]
"""
cur.execute(sql)
conn.commit()
cur.close()
```

Назначение пользователю роли базы данных (права доступа к объектам)

In [272]:

```
cur = conn.cursor()
sql = """
EXEC sp_addrolemember 'db_owner', 'TestUser'
"""
cur.execute(sql)
conn.commit()
cur.close()
```

Параметры базы данных

In [273]:

```
cur = conn.cursor()
sql = """
ALTER DATABASE [TestDB] SET READ_ONLY
"""
cur.execute(sql)
conn.commit()
cur.close()
```

In [274]:

```
cur = conn.cursor()
sql = """
ALTER DATABASE [TestDB] SET READ_WRITE
"""
cur.execute(sql)
conn.commit()
cur.close()
```

Создание архива базы данных

```
BACKUP DATABASE [TestDB]
    TO DISK = 'A:\BACKUP_DB\TestDB.bak'
    WITH
        NAME = N'База данных TestDB',
        STATS = 10
```

```
18 проц. обработано.
37 проц. обработано.
55 проц. обработано.
61 проц. обработано.
73 проц. обработано.
80 проц. обработано.
99 проц. обработано.
100 проц. обработано.
Обработано 704 страниц для базы данных "TestDB", файл "TestDB" для файла 1.
Обработано 2 страниц для базы данных "TestDB", файл "TestDB_log" для файла 1.
BACKUP DATABASE успешно обработал 706 страниц за 0.049 секунд (112.484 МБ/сек).

Completion time: 2022-12-25T16:58:06.7171007+03:00
```

Восстановление базы данных из архива

Существуют следующие модели восстановления:

Простая – используется для баз данных, данные в которых изменяются неинтенсивно, и потеря данных с момента создания последней копии базы не является критичной. Например, копии создаются каждую ночь, если произошел сбой в середине дня, то все данные, которые были сделаны в течение этого дня, будут потеряны. Копия журнала транзакций при такой модели восстановления не создается;

Полная – используется для баз данных, в которых необходима поддержка длительных транзакций. Это самая надежная модель восстановления, она позволяет восстановить базу данных до точки сбоя, в случае наличия заключительного фрагмента журнала транзакций. В данном случае копию журнала транзакций необходимо делать по возможности как можно чаще. В журнал транзакций записываются все операции;

С **неполным протоколированием** – данная модель похожа на «Полную» модель, однако в данном случае большинство массовых операций не протоколируется, и, в случае сбоя, их придѐтся повторить с момента создания последней копии журнал транзакций.

```
USE master
GO
RESTORE DATABASE [TestDB]
    FROM DISK = N'D:\BACKUP_DB\TestDB.bak'
    WITH
        FILE = 1,
        STATS = 10
```

Перемещение базы данных

```
USE master
GO
EXEC sp_detach_db @dbname = 'TestDB'
```

```
USE master
GO
```

```
CREATE DATABASE [TestDB] ON (
    FILENAME = 'D:\DataBase\TestDB.mdf'),
    (FILENAME = 'D:\DataBase\TestDB_log.ldf')
FOR ATTACH
GO
```

Сжатие базы данных

In [275]:

```
cur = conn.cursor()
sql = """
DBCC SHRINKDATABASE('TestDB')
"""
cur.execute(sql)
conn.commit()
cur.close()
```

In [276]:

```
cur = conn.cursor()
sql = """
DBCC SHRINKFILE ('TestDB_log', 5)
"""
cur.execute(sql)
conn.commit()
cur.close()
```

In [ ]:

```
conn.close()
```

In [ ]:

```

```