Пловдивски университет "Паисий Хилендарски"

Факултет по Математика и Информатика

Катедра "Компютърни системи"

Самостоятелна работа по Бази от данни

Учебна година 09/10

Съдържание

1. ВЪВЕДЕНИЕ В MS SQL SERVER 2000 3

1.1. Бази данни и файлове на базите данни 3

2. ВЪВЕДЕНИЕ В TRANSACT-SQL 5

2.1. Език за дефиниране на данни (DDL) 5
2.2. Език за контрол на достъпа до данни (DCL) 6
2.3. Език за манипулиране на данни (DML) 7
2.4. Типове данни 8
2.5. Оператори 10
2.6. Вградени (стандартни) функции 11
2.7. Идентификатори 14
2.8. Коментари 14

3. СЪЗДАВАНЕ НА БАЗА ДАННИ ЗА БИБЛИОТЕКА 15

4. МАНИПУЛИРАНЕ НА ДАННИТЕ 24

4.1. Добавяне 24
4.2. Променяне 27
4.3. Изтриване 28

5. КОНСТРУКЦИИ ЗА ИЗВЛИЧАНЕ НА ДАННИ 30

5.1. SELECT 30
5.2. Оператор UNION 33
5.3. Оператор JOIN 36

6. ТРАНЗАКЦИИ 44

6.1. Нива на изолация на транзакциите в T-SQL 47
6.2. Deadlock ситуации 48

7. СЪХРАНЕНИ ПРОЦЕДУРИ И ФУНКЦИИ 51

7.1. Опростен синтаксис и примери 51
7.2. Вложени съхранени процедури 53
7.3. Рекурсия в съхранените процедури 53
7.4. Функции 55

8. ТРИГЕРИ 58

8.1. Тригерни събития 58
8.2. Изпълнение на тригер 58
8.3. Опростен синтаксис за създаване на тригер 59
8.4. Изтриване, промяна и забраняване 59
8.5. Създаване на тригери 60

9. КУРСОРИ 65

9.1. Въведение 65
9.2. Опростен синтаксис 66
9.3. Примери 66

1. Въведение в MS SQL Server 2000

По време на курса по бази данни за сървър ще бъде използван Microsoft SQL Server 2000. Конкретната реализация на SQL стандарта в този сървър за бази данни се нарича Transact-SQL, който представлява език, съдържащ командите за администриране и управление на една база данни в този сървър.

1.1. Бази данни и файлове на базите данни

Базата данни на MS SQL Server 2000 е набор от обекти, които съдържат и манипулират данни. Такива обекти са таблици, изгледи, съхранени процедури, ограничения и др. Базата данни също така поддържа собствено множество от потребителски регистрации и средства за защита.

1.1.1. Специални системни бази данни

Всяка нова инсталация на SQL Server включва автоматично няколко бази данни:

- **master –** състои се от системни таблици, които съдържат информация от инсталацията на сървъра и всички други бази данни, създавани след това. Въпреки, че всяка база данни има набор от системни каталози, съдържащи информация за обектите в нея, тази база данни има системни каталози с информация за дисковото пространство, разположението и използването на файловете, параметри на конфигурацията за цялата система, регистрации на потребители и други;
- **model –** база данни, използвана за модел при създаване на нова база данни. Ако желаем всяка нова база данни да бъде създавана с определени параметри (регистрации на потребители, таблици, съхранени процедури и др.), можем да ги поставим в тази база данни и всички нови бази данни ще ги наследяват;
- **tempdb –** временна база данни, работно пространство. При всяко рестартиране на сървъра тази база данни бива създавана наново, т.е. всички обекти в тази база данни, създадени от потребителите, ще бъдат загубени;
- **msdb –** тази база данни се използва от услугата SQL Server Agent, който изпълнява планирани действия – архивиране, репликации и др.

1.1.2. Файлове на базите данни

Базата данни обхваща поне два, а може и повече, файла и тези файлове се задават при създаване или модифициране на базата данни. Всяка база данни трябва да обхваща поне два файла – един за данни (с разширение MDF) и един

за дневника с транзакциите (с разширение LDF), но може да съдържа и нула или повече второстепенни файлове с данни (с разширение MDF).

1.1.3. Създаване на бази данни

Най-бързият начин за създаване на база данни е с помощта на Enterprise Manager. За създаване можем да използваме и Transact-SQL команда, която в най-прост вид е следната:

```
CREATE DATABASE mydb
```

където mydb е името на новата база данни. Но тази команда има и множество параметри – имена на файловете за данни и дневника с транзакциите, максимален техен размер и стъпка на нарастване и т.н., които са описани в системната документация, която може да бъде инсталирана заедно със сървъра и чието инсталиране е препоръчително – Books Online.

2. Въведение в Transact-SQL

Езикът Transact-SQL е разширение на стандарта SQL. Той съдържа конструкции от езика за дефиниране на данни (Data Definition Language), конструкции от езика за контрол на достъпа до данни (Data Control Language) и конструкции от езика за манипулиране на данни (Data Manipulation Language).

Всяко приложение, което комуникира с SQL Server, изпраща конструкции на T-SQL на сървъра, независимо от потребителския интерфейс на приложението.

SQL Server Books Online включва пълен справочник за конструкциите на T-SQL, описващ всеки елемент и предоставящ примери за използването им. Една конструкция представлява набор от символи, който изпълнява действия върху обекти или данни в една база данни.

Синтаксисът на T-SQL, който ще използваме, не е *case sensitive*, т.е. не разпознава малки и големи букви.

2.1. Език за дефиниране на данни (DDL)

Служи за дефиниране на обекти в базата данни – таблици, изгледи, съхранени процедури и др. Повечето DDL конструкции имат следната форма:

- CREATE *име_на_обект* – създаване на обект;
- ALTER *име_на_обект* – промяна на обект;
- DROP *име_на_обект* – премахване на обект.

Следващите примери демонстрират използването на тези конструкции.

CREATE

Създаване на таблица в съществуващата база данни mydb, която ще съдържа данни за студенти, разпределени в три колони – номер, име и телефон на студент. При добавяне на запис в тази таблица първите две колони ще изискват задължително въвеждане на данни – декларирани са като NOT NULL.

```
USE mydb

CREATE TABLE Students
(
  student_num int not null,
  student_name varchar(50) not null,
  phone varchar(40)
)
```

ALTER

Ще модифицираме таблицата като добавим една колона за адрес на студента.

```
USE mydb

ALTER TABLE Students
ADD address varchar(100) NULL
```

Добавената колона е декларирана така, че въвеждането на данни в нея да не е задължително.

DROP

Можем да премахнем таблица от базата данни. Ако таблицата, която се опитваме да премахнем, се използва от други обекти трябва първо да премахнем тях и чак след това нея.

```
USE mydb

DROP TABLE Students
```

Забележка: командата за изтриване на таблицата е добре да не бъде изпълнявана в момента, защото с тази таблица ще дадем някои примери по-късно.

Това бяха команди в техния най-прост вариант. За да видим всички опции, които те притежават, можем да използваме Books Online.

2.2. Език за контрол на достъпа до данни (DCL)

Конструкциите от този език служат за управление на правата за достъп до обекти от базата данни.

GRANT

Чрез тази конструкция можем да разрешим на определен потребител на текущата база данни да работи с данните или да изпълнява T-SQL конструкции.

Следващата конструкция дефинира разрешение на ролята Public (може и на конкретен потребител) за извличане на данни от таблицата STUDENTS в базата данни mydb.

```
USE mydb

GRANT SELECT ON Students TO Public
```

REVOKE

Чрез тази конструкция можем да премахнем предоставено или отказано преди разрешение на потребител или роля в текущата база данни. Следната конструкция отнема разрешението за извличане на данни от таблицата Students на ролята Public.

```
USE mydb

REVOKE SELECT ON Students TO Public
```

DENY

Чрез тази конструкция можем да откажем разрешение на определен потребител на базата данни като в същото време предотвратим възможността този потребител да наследи разрешението чрез членство в определена група, която има разрешение.

```
USE mydb

DENY SELECT ON Students TO Public
```

2.3. Език за манипулиране на данни (DML)

Използва се за извличане, добавяне, изтриване и актуализиране на данни от обекти, дефинирани чрез DDL.

INSERT

Конструкцията служи за добавяне на нов ред в таблица. Ще добавим един запис за студент в таблицата.

```
USE mydb

INSERT INTO Students(student_num, student_name, phone, address)
VALUES(951021, 'Мария Петрова', '032 67-34-53', 'гр. Пловдив')
```

SELECT

Тази конструкция служи за извличане на редове от таблици от базата данни като позволява избирането на определени колони.

```
USE mydb

SELECT student_num, student_name
FROM Students
WHERE student_name = 'Иван Иванов'
```

Изпълнението на тази конструкция извлича номер и име на всеки студент, записът, за който съдържа в полето student_name стойността 'Иван Иванов'.

UPDATE

Конструкцията служи за промяна на данните в една таблица. Ще променим адреса на добавената студентка в предния пример.

```
USE mydb

UPDATE Students
SET address = 'гр. Ямбол'
WHERE student_num = 951021
```

DELETE

Конструкцията служи за изтриване на редове от таблица. Ще премахнем записа за гореспоменатата студентка.

```
USE mydb

DELETE FROM Students
WHERE student_num = 951021
```

2.4. Типове данни

SQL доставя различни типове системни данни, както и възможност за дефиниране на потребителски типове, базирани на системните типове данни.

2.4.1. Целочислени

- **bigint –** интервал на допустимите стойности от -2^{63} (-9223372036854775808) до 2^{63} - 1 (9223372036854775807). Размерът на заеманата памет е 8 байта;
- **int –** интервал на допустимите стойности от -2^{31} (-2147483648) до 2^{31} - 1 (2147483647). Размерът на заеманата памет е 4 байта. SQL-92 синонимът е integer;

- **smallint -** интервал на допустимите стойности от -2^{15} (-32768) до $2^{15} - 1$ (32767). Размерът на заеманата памет е 2 байта;
- **tinyint -** интервал на допустимите стойности от 0 до 255. Размерът на заеманата памет е 1 байт.

2.4.2. Логически

- **bit –** целочислен тип със стойности 0 или 1. Размерът на заеманата памет е 1 бит.

2.4.3. Числа с плаваща запетая с фиксирана точност

- **decimal -** интервал на допустимите стойности от $-10^{38} + 1$ до $10^{38} - 1$. Формат: decimal(p, s), където p е точността (max 38 цифри), а s е броят на цифрите след десетичната запетая;
- **numeric –** като decimal.

2.4.4. Числа с приблизителна точност

- **float -** интервал на допустимите стойности от -1.79E+308 до 1.79E+308. Формат: float(n), където n определя точността и размера памет, в който ще бъде записано числото. Стойностите на n са от 1 до 53, като от 1 до 24 ще се използват 7 цифри, а от 25 до 53 – 15 цифри, съответно 4 и 8 байта;
- **real -** интервал на допустимите стойности от -3.40E+38 до 3.40E+38. Размерът памет е 4 байта, еквивалентно число на float(24).

2.4.5. Парични

- **money -** интервал на допустимите стойности от -2^{63} (-922337203685477.5808) до $2^{63} – 1$ (922337203685477.5807) с точност до десетохилядна. Размерът на заеманата памет е 8 байта;
- **smallmoney -** интервал на допустимите стойности от -214748.3648 до 214748.3647 с точност до десетохилядна. Размерът на заеманата памет е 4 байта.

2.4.6. Дата и час

- **datetime –** дата и час от 1 януари, 1753, до 31 декември, 9999, с точност до 3 милисекунди;
- **smalldatetime –** дата и час от 1 януари, 1900, до 6 юни, 2079, с точност до 1 минута.

2.4.7. Низове

- **char –** низ с фиксирана дължина. Максимална дължина – 8000 символа;
- **varchar -** низ с променлива дължина. Максимална дължина – 8000 символа;
- **text -** низ с променлива дължина. Размер до 2 GB.

2.4.8. Низове във формат на Unicode

- **nchar –** низ с фиксирана дължина. Максимална дължина – 4000 символа;
- **nvarchar -** низ с променлива дължина. Максимална дължина – 4000 символа;
- **ntext -** низ с променлива дължина. Размер до 1 GB.

2.4.9. Двоични данни

- **binary –** двоични данни с фиксирана дължина. Размер до 8000 байта;
- **varbinary –** двоични данни с променлива дължина. Размер до 8000 байта;
- **image –** двоични данни с променлива дължина. Размер до 2GB.

2.5. Оператори

2.5.1. Аритметични

- Сбор (**+**), ако аргументите са низове – конкатенация;
- Разлика (**–**);
- Умножение (*****);
- Деление (**/**). Ако операндите са цели числа – целочислено деление;
- Остатък от целочислено деление (**%**).

2.5.2. Оператори за сравнение

- Равно (**=**);
- Различно (**<>**, **!=**);
- По-малко или равно (**<=**);
- По-голямо или равно (**>=**);
- По-малко (**<**);

- По-голямо (**>**);
- Не по-голямо (**!>**);
- Не по-малко (**!<**)

2.5.3. Логически оператори

- И – **AND**;
- ИЛИ – **OR**;
- Отрицание – **NOT**.

2.6. Вградени (стандартни) функции

Програмният език T-SQL съдържа три вида стандартни функции: за набори от редове, агрегатни и скаларни.

2.6.1. Агрегатни функции

Агрегатните функции работят върху съвкупност от стойности от много редове, но връщат единична стойност. Характерно за агрегатните функции с изключение на COUNT(*) е, че не вземат предвид при изчислението редове със стойности NULL в изчисляваните колони.

- SUM ([ALL | DISTINCT] *expression*) – връща сумата от всички стойности или само от DISTINCT стойностите в израза;
- AVG ([ALL | DISTINCT] *expression*) – връща средно аритметичната стойност на всички стойности или само на различните (DISTINCT) стойности в израза (колоната);
- COUNT ([ALL | DISTINCT] *expression*) – връща броя различни от NULL стойности. Когато се използва DISTINCT връща броя на уникалните и различни от NULL стойности;
- COUNT (*) – връща броя редове, дори и тези с NULL стойности в израза. В този си вид функцията не приема параметри и не може да бъде използван DISTINCT;
- MAX (expression) – връща максималната стойност в израза;
- MIN (expression) – връща минималната стойност в израза.

Забележка: функциите MAX и MIN могат да използват за стойности на израза числа, низове и дати, а останалите функции – само за числа. Където го има като опция за параметрите, ALL е по подразбиране.

2.6.2. Скаларни функции

Скаларните функции приемат като параметър (или обработват) единична стойност и връщат като резултат единична стойност. Могат да бъдат използвани навсякъде, където е валидно използването на изрази.

1. Функции за преобразуване на типове

Служат за явно преобразуване на един тип в друг.

- CAST (expression AS data_type);
- CONVERT (*data_type* [(*length*)] , *expression* [, *style*]).

Където:

- expression – валиден израз, резултатът от който да бъде преобразуван;
- data_type – тип, към който се прави преобразуване;
- length – опционален параметър за символните типове данни (низове);
- style – стил, използван при преобразуване на дати в низове.

2. Функции за дата

- GETDATE() – връща системната дата на компютъра, на който работи сървъра;
- DATEADD(*datepart, number, date*) – връща дата, образувана чрез добавяне към датата *date*, *number* на брой части *datepart*;
- DATEDIFF(*datepart, date1, date2*) – връща число, разликата в части от вид *datepart*, между две дати;
- DATENAME(*datepart, date*) – връща като низ съдържанието на конкретна част от дата. Напр. February, ако месеца в *date* е февруари и *datepart* е mm;
- DATEPART(*datepart, date*) – връща като число съдържанието на конкретна част от дата. Напр. 2, ако месеца в *date* е февруари и *datepart* е mm.

Където стойностите на *datepart* са от следната таблица:

datepart	**Съкращение**	**Стойности**
year	yy, yyyy	1753-9999

quarter	qq, q	1-4
month	mm	1-12
dayofyear	dy, y	1-366
day	dd, d	1-31
week	wk, ww	1-53
weekday	dw	1-7 (Sun.-Sat.)
hour	hh	0-23
minute	mi, n	0-59
second	ss, s	0-59
millisecond	ms	0-999

3. Математически функции

Математическите функции са следните: ABS, ACOS, ASIN, ATAN, ATN2, CEILING, COS, COT, DEGREES, EXP, FLOOR, LOG, LOG10, PI, POWER, RADIANS, RAND, ROUND, SIGN, SIN, SQUARE, SQRT, TAN.

Някои от тях:

- PI() – връща числото π;
- POWER(number, y) – връща number на степен y;
- RAND([seed]) – генератор на случайни числа между 0 и 1 или цяло ако е указано цялото число seed.

4. Функции за работа с низове

Функциите за работа с низове са следните: ASCII, CHAR, CHARINDEX, DIFFERENCE, LEFT, RIGHT, LEN, LTRIM, RTRIM, LOWER, UPPER, NCHAR, PATINDEX, REPLACE, QUOTENAME, REPLICATE, REVERSE, SOUNDEX, SPACE, STR, STUFF, SUBSTRING, UNICODE.

Някои от тях:

- LTRIM(char_expr) – премахва началните (водещите) интервали;
- RTRIM(char_expr) – премахва завършващите интервали;
- LOWER (char_expr) – преобразува низа до малки букви;
- UPPER (char_expr) – преобразува низа до големи букви;
- REVERSE (char_expr) – обръща низа;

- SUBSTRING(*expression*, *start*, *length*) - връща част от символен или бинарен израз, започвайки от символ с индекс start и дължина length символа.

5. Системни функции

Връщат системна информация.

COALESCE, HOST_NAME, OBJECT_NAME, COL_LENGTH, IDENT_INCR, STATS_DATE, COL_NAME, IDENT_SEED, SUSER_ID, DATALENGTH, INDEX_COL, SUSER_NAME, DB_ID, ISNULL, USER_ID, DB_NAME, NULLIF, USER_NAME, GETANSINULL, HOST_ID, OBJECT_ID.

6. Функции за работа с типове TEXT и IMAGE

Следните функции са предназначени за работа със стойности от тип TEXT и IMAGE: PATINDEX, TEXTPTR, TEXTVALID.

2.7. Идентификатори

Идентификаторите се състоят от букви, цифри и някои специални символи - след първия знак може да има #, $, или _. Не трябва да съвпадат със запазена дума и започват с буква или един от символите _, #, @, като в тези случаи:

- Идентификатор, започващ с @ е локална променлива;
- Идентификатор, започващ с @@ е глобална променлива;
- Идентификатор, започващ с # е локален временен обект;
- Идентификатор, започващ с ## е глобален временен обект.

2.8. Коментари

Начало на коментар е последователността от символи /*.

Край на коментар е последователността от символи */.

Коментар на ред е --.

Забележка: в коментар не трябва да се среща ключовата дума go, която указва край на пакет команди за изпълнение.

3. Създаване на база данни за библиотека

Задача: Да се проектира схема на база данни за библиотека, в която да се съхранява информация за:

- Книгите в библиотеката – ISBN номер на книгата, заглавие, цена, година на публикуване;
- Автори – персонален номер, име, фамилия;
- Клиенти на библиотеката – идентификатор (ЕГН), име, фамилия, телефонни номера и адрес.

Изисквания:

- ✓ Всяка книга може да бъде написана от повече от един автор;
- ✓ Всяка книга може да бъде вземана от различни клиенти, но в различни моменти от времето;
- ✓ Всеки клиент може да вземе повече от една книга в един и същ момент;
- ✓ Клиент може да има повече от един телефонен номер;
- ✓ Трябва да може да се разбере всяка книга в кой клиент е в момента, кой клиент какви книги е взимал и кои в момента са у него.

Този пример представя част от организацията в библиотека относно наличните книги, техните автори и клиентите на библиотеката. Целта му е да демонстрира използването на основни конструкции от езика Transact-SQL, както и реализация на отношение от тип “много към много” чрез разлагането му на две от тип “едно към много”.

В началната фаза на проектирането е добре да бъдат обмислени относително независимите обекти. В случая такива изглеждат този, който ще съхранява информация за книгите, за авторите и за клиентите. Те няма да зависят от други обекти и затова ще е най-лесно да започнем тях като съобразим атрибутите им с изискванията от заданието.

Първо да си създадем базата данни.

```
CREATE DATABASE Library
GO
```

За съхраняване на информацията за книгите ще използваме обект, наречен BOOK.

Описание на атрибутите:

- isbn – идентификатор на книгата, използван за първичен ключ. Тип – низ, до 20 символа, задължително е наличието на стойност във всички атрибути на обекта;
- title – заглавие на книгата. Тип - низ до 50 символа;
- price – цена на книгата. Тип – число с плаваща запетая, дължина до 7 цифри за цялата част и до 2 след десетичната запетая (точка);
- year_pub – година на публикуване. Тип – цяло число.

Ограничения, наложени върху атрибутите:

- ✓ `PK_BOOK` – ограничение за първичен ключ на атрибута isbn;
- ✓ `CHK_YEAR_PRICE` – ограничение от тип CHECK, проверяващо годината на публикуване да е в интервала 1700 – 2008 и цената да има стойност минимум 1,50.

Схема на обекта:

BOOK			
ISBN	varchar(20)	<pk>	not null
TITLE	varchar(50)		not null
PRICE	decimal(9,2)		not null
YEAR_PUB	int		not null

За създаването на описаният обект използваме следния код:

```
create table BOOK
(
    ISBN        varchar(20)           not null,
    TITLE       varchar(50)           not null,
    PRICE       decimal(9,2)          not null,
    YEAR_PUB    int                   not null,

    constraint PK_BOOK primary key (ISBN),
    constraint CHK_YEAR_PRICE
      check ((YEAR_PUB between 1700 and 2008) and
             (PRICE >= 1.50))
)
```

За съхраняване на информацията за авторите ще използваме обект, който именуваме AUTHOR.

Описание на атрибутите:

- author_id – идентификатор на автора. Тип – цяло число, задължително е наличието на стойност във всички атрибути на обекта;
- fname – име на автора. Тип - низ до 15 символа;

- lname – фамилия на автора. Тип – низ до 15 символа.

Ограничения, наложени върху атрибутите:

✓ PK_AUTHOR – ограничение за първичен ключ на атрибута author_id.

```
create table AUTHOR
(
    AUTHOR_ID   int                   not null,
    FNAME       varchar(15)           not null,
    LNAME       varchar(15)           not null

    constraint PK_AUTHOR primary key (AUTHOR_ID)
)
```

След изпълнението на посоченият код схемата има следния вид:

За съхраняване на информацията за клиентите ще създадем обект CLIENT.

Описание на атрибутите:

- client_egn – ЕГН на клиента, използван за първичен ключ. Тип – низ от 10 символа, задължително е наличието на стойност;
- fname – име на клиента. Тип - низ до 15 символа, задължително е наличието на стойност;
- lname – фамилия на клиента. Тип – низ до 15 символа, задължително е наличието на стойност;
- address – адрес на клиента. Тип – низ с голяма дължина, задължително е наличието на стойност. В термините на Transact-SQL (MS SQL Server) този тип е **text** (размер – до 2,147,483,647 символа (или 2GB)), стандартно (SQL-92) - двоичен обект (BLOB – Binary Large Objects).

От изискванията към данните за клиентите се вижда, че всеки клиент може да притежава повече от един телефонен номер, т.е. отношението на клиентите и телефоните е от тип едно-към-много (1:n). Следователно телефонните номера не могат да присъстват в таблицата CLIENT, т.е. за тях ще трябва да създадем отделна таблица, която ще е зависима от CLIENT, защото тя е “слаб” обект, т.е. без наличие на CLIENT тя няма смисъл.

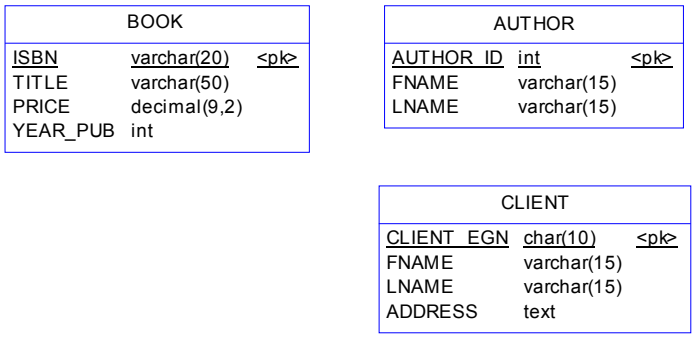
Ограничения, наложени върху атрибутите:

- ✓ PK_CLIENT – ограничение за първичен ключ на атрибута client_egn.

```
create table CLIENT
(
    CLIENT_EGN int                  not null,
    FNAME      varchar(15)          not null,
    LNAME      varchar(15)          not null,
    ADDRESS    text                     null,

    constraint PK_CLIENT primary key (CLIENT_EGN)
)
```

Основната структура на схемата вече е създадена и сега изглежда така:



Остава да създадем таблицата за телефонните номера, която ще има два атрибута:

- owner_id – външен ключ към обекта CLIENT и съответният му атрибут client_egn. Тип - цяло число, задължително е наличието на стойност в този атрибут;
- phone – телефонен номер на съответния клиент. Тъй като той може да съдържа водеща нула или знак + ще изберем за тип данни низ с дължина до 20 символа.

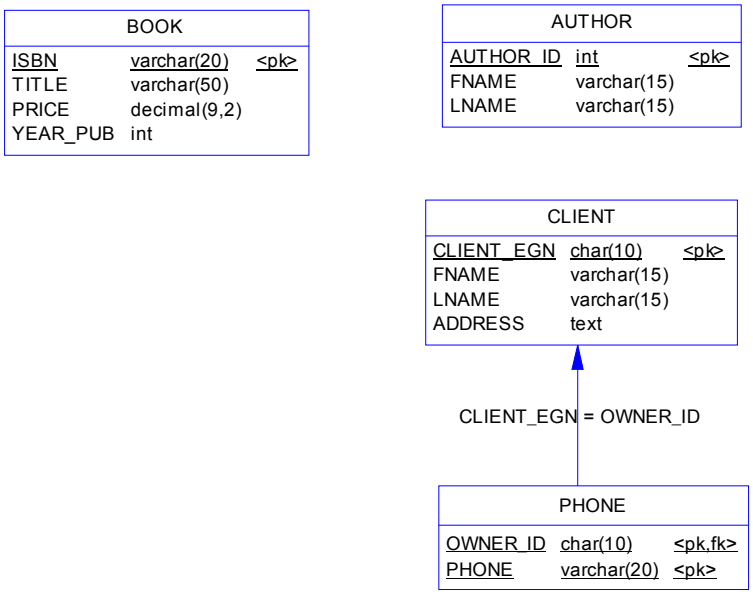
Ограничения, наложени върху атрибутите:

- ✓ PK_PHONE – ограничение за първичен ключ, включващ атрибутите owner_id и phone;
- ✓ FK_PHONE_CLIENT – ограничение за външен ключ върху атрибута owner_id към обекта CLIENT и съответния му атрибут client_egn. Ще използваме опцията за каскадно разширяване на операцията за изтриване и промяна на стойност на първичния ключ в таблицата CLIENT, т.е. при изтриване на клиент да се изтриват всички негови телефони, а при промяна на идентификатора му да се актуализират стойностите на външния ключ в таблицата с телефоните;

```
create table PHONE
(
   OWNER_ID             int                    not null,
   PHONE                varchar(20)            not null,
   constraint PK_PHONE primary key  (OWNER_ID, PHONE)
)
go

alter table PHONE
   add constraint FK_PHONE_CLIENT foreign key (OWNER_ID)
      references CLIENT (CLIENT_EGN)
on delete cascade
on update cascade
go
```

След изпълнението на този код схемата изглежда така:



Вижда се, че трябва да свържем авторите и книгите по някакъв начин, за да имаме данни за това коя книга от кой (кои) автори е написана и съответно кой автор кои книги е написал (или участвал в написването им). Тъй като една книга може да бъде писана от повече от един автор, а и един автор може да е написал повече от една книга, то тук е налице релация от тип “много към много”. За да избегнем излишно повтаряне на информация в обектите BOOK и AUTHOR ще добавим обект BOOK_AUTHORS, чрез който ще разложим релацията на две от тип “едно към много”.

Описание на атрибутите:

- isbn – външен ключ към обекта BOOK и съответният му атрибут isbn. Тип - цяло число, задължително е наличието на стойност във всички атрибути на обекта;
- author_id – външен ключ към обекта AUTHOR и съответния му атрибут author_id.

Ограничения, наложени върху атрибутите:

- ✓ `PK_BOOK_AUTHORS` – ограничение за първичен ключ, наложено върху комбинацията от атрибути isbn и author_id с цел предотвратяване повторното въвеждане на факта (дублиране), че определен автор е писал дадена книга;
- ✓ `FK_BOOK_BOOK` – ограничение за външен ключ към обекта BOOK и съответният му атрибут isbn;
- ✓ `FK_BOOK_AUTHOR` – ограничение за външен ключ към обекта AUTHOR и съответният му атрибут author_id.

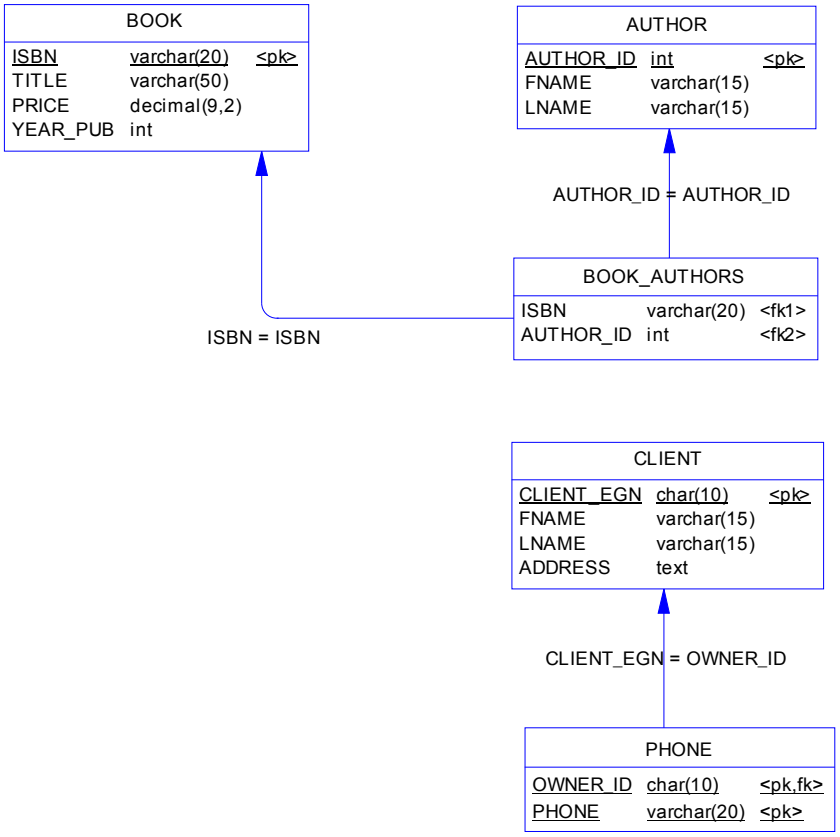
```
create table BOOK_AUTHORS
(
   ISBN            varchar(20)          not null,
   AUTHOR_ID       char(10)             not null
)
go

alter table BOOK_AUTHORS
   add constraint FK_BOOK_AUT_REFERENCE_BOOK foreign key (ISBN)
      references BOOK (ISBN)
go

alter table BOOK_AUTHORS
   add constraint FK_BOOK_AUT_REFERENCE_AUTHOR foreign key (AUTHOR_ID)
      references AUTHOR (AUTHOR_ID)
go
```

<u>Забележка:</u> От кода се вижда, че таблицата няма първичен ключ. Целта на тази реализация е да демонстрира, че на практика в много случаи една таблица може да съществува в схемата и да работи без да има първичен ключ, но това не е препоръчително, защото при невнимателно боравене с такава таблица могат да възникнат усложнения (в зависимост от СУБД), особено ако тя няма и уникален индекс или ключ. Затова се препоръчва подобна схема да бъде избягвана, дори единствената причина за това да е добрият стил.

Схемата вече изглежда така:



За да имаме информация за това кой клиент каква книга е вземал и дали в момента има книги в него и кои са те трябва да създадем последният обект, който да свърже клиентите с книгите. Тук ситуацията много прилича на тази в предишната стъпка. А именно – всеки клиент може да вземе повече от една книга и всяка книга може да бъде вземана от много хора (не е едновременно, разбира се). Затова отново представяме тази релация като две от типа “едно към много” чрез допълнителен обект, именуван CLIENT_BOOKS.

Описание на атрибутите:

- client_egn – външен ключ към обекта CLIENT и съответния му атрибут client_egn. Тип - цяло число, задължително е наличието на стойност;
- isbn - външен ключ към обекта BOOK и съответния му атрибут. Тип - цяло число, задължително е наличието на стойност;
- taken – дата, на която е взета книгата. Тип – дата, задължително е наличието на стойност;
- returned – дата, на която е върната книгата. Тип – дата. Стойността на този атрибут не е задължителна, защото тя се попълва когато книгата бъде върната, а записът се създава при вземането ѝ, т.е. при връщането има само модифициране. Липсата на стойност в този атрибут означава, че книгата е още в съответния клиент.

Ограничения, наложени върху атрибутите:

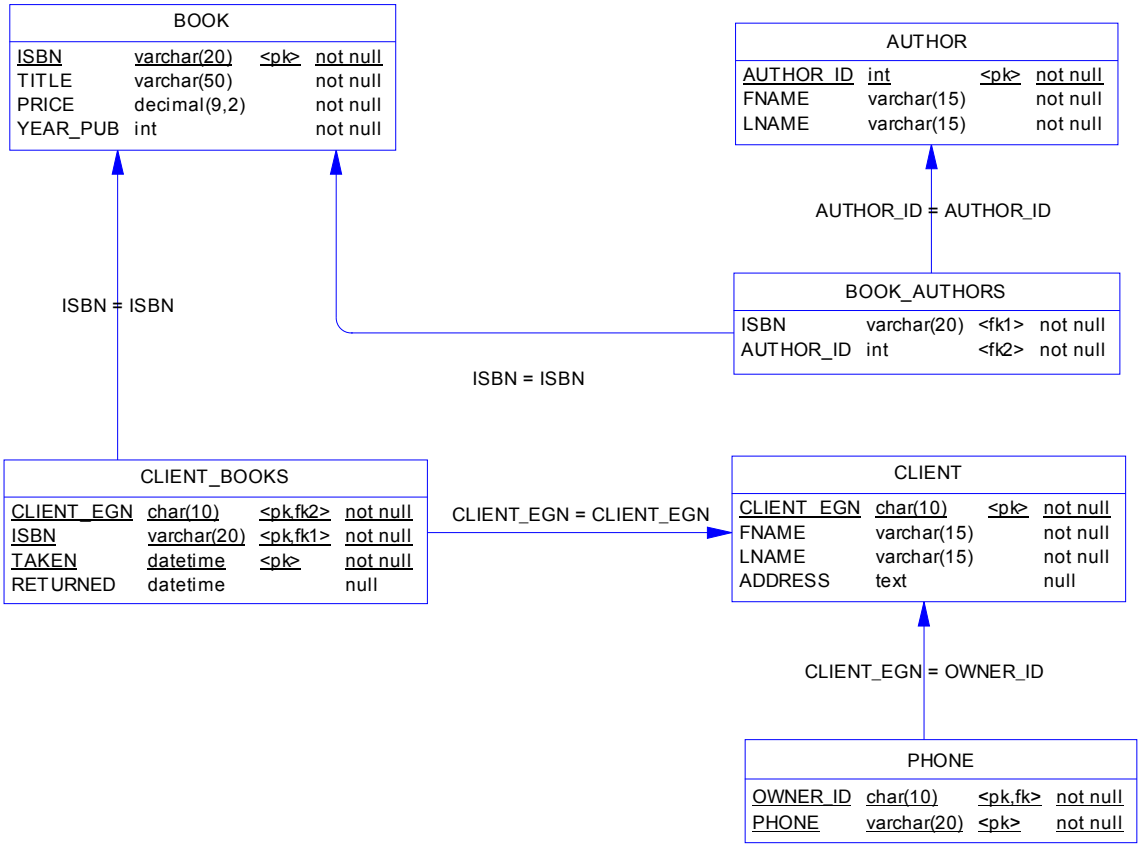
- ✓ `PK_CLIENT_BOOKS` – ограничение за първичен ключ върху атрибутите client_egn, isbn и taken. Включването на атрибута taken се налага от факта, че ако първичният ключ се съставя само от другите два атрибута, то един клиент ще може да вземе определена книга само веднъж;
- ✓ `FK_CLIENT_CLIENT` – ограничение за външен ключ към обекта CLIENT и съответния му атрибут client_egn;
- ✓ `FK_CLIENT_BOOK` - ограничение за външен ключ към обекта BOOK и съответния му атрибут isbn.

```
create table CLIENT_BOOKS
(
   CLIENT_EGN           int                  not null,
   ISBN                 varchar(20)          not null,
   TAKEN                datetime             not null,
   RETURNED             datetime             null,
   constraint PK_CLIENT_BOOKS primary key (CLIENT_EGN, ISBN, TAKEN)
)
go

alter table CLIENT_BOOKS
   add constraint FK_CLIENT_B_REFERENCE_BOOK foreign key (ISBN)
      references BOOK (ISBN)
go

alter table CLIENT_BOOKS
   add constraint FK_CLIENT_B_REFERENCE_CLIENT foreign key (CLIENT_EGN)
      references CLIENT (CLIENT_EGN)
go
```

Окончателният вид на схемата е следният:


BOOK
ISBN varchar(20) <pk> not null
TITLE varchar(50) not null
PRICE decimal(9,2) not null
YEAR_PUB int not null
AUTHOR
AUTHOR_ID int <pk> not null
FNAME varchar(15) not null
LNAME varchar(15) not null
AUTHOR_ID = AUTHOR_ID
BOOK_AUTHORS
ISBN varchar(20) <fk1> not null
AUTHOR_ID int <fk2> not null
ISBN = ISBN
ISBN = ISBN
CLIENT_BOOKS
CLIENT_EGN char(10) <pk,fk2> not null
ISBN varchar(20) <pk,fk1> not null
TAKEN datetime <pk> not null
RETURNED datetime null
CLIENT_EGN = CLIENT_EGN
CLIENT
CLIENT_EGN char(10) <pk> not null
FNAME varchar(15) not null
LNAME varchar(15) not null
ADDRESS text null
CLIENT_EGN = OWNER_ID
PHONE
OWNER_ID char(10) <pk,fk> not null
PHONE varchar(20) <pk> not null

4. Манипулиране на данните

4.1. Добавяне

Преди да покажем с примери извличането на данни ще демонстрираме тяхното добавяне, така също ще имаме тестови данни, които да извличаме. При добавянето трябва да следим за реда на добавяне на данни – в таблиците, където има външни ключове, данните трябва да се добавят след добавяне на тези данни, от които зависят (към които сочат външните ключове).

Добавяне на данни в таблицата BOOK, т.е. въвеждане на книги.

```
insert into BOOK (ISBN, TITLE, PRICE, YEAR_PUB)
values ('954-9656-03-5', 'Въведение в SQL', 10.30, 1997)

insert into BOOK (ISBN, TITLE, PRICE, YEAR_PUB)
values ('954-9656-03-1', 'Transact-SQL', 28.10, 1997)

insert into BOOK (ISBN, TITLE, PRICE, YEAR_PUB)
values ('954-9656-03-9', 'Научете сами SQL (част 1)', 6.50, 1998)

insert into BOOK (ISBN, TITLE, PRICE, YEAR_PUB)
values ('954-9656-04-7', 'Научете сами SQL (част 2)', 6.50, 1998)

insert into BOOK (ISBN, TITLE, PRICE, YEAR_PUB)
values ('1-55860-576-2', 'SQL for smarties: advanced SQL
programming', 95.00, 1999)

insert into BOOK (ISBN, TITLE, PRICE, YEAR_PUB)
values ('5-7789-45-675', 'SQL Server 7.0', 17.80, 1999)

insert into BOOK (ISBN, TITLE, PRICE, YEAR_PUB)
values ('3-213-77-959', 'Алхимикът', 14.60, 2006)
```

Добавяне на автори.

```
insert into AUTHOR (AUTHOR_ID, FNAME, LNAME)
values (64, 'Евлоги', 'Георгиев')

insert into AUTHOR (AUTHOR_ID, FNAME, LNAME)
values (56, 'Joe', 'Celko')

insert into AUTHOR (AUTHOR_ID, FNAME, LNAME)
values (72, 'Светла', 'Колева')

insert into AUTHOR (AUTHOR_ID, FNAME, LNAME)
values (54, 'Петър', 'Димитров')

insert into AUTHOR (AUTHOR_ID, FNAME, LNAME)
values (98, 'Паулу', 'Куелю')
```

Добавяне на данни за това кой автор кои книги е писал или от гледна точка на книгите коя книга от кои автори е писана.

```
insert into BOOK_AUTHORS (ISBN, AUTHOR_ID)
values ('954-9656-03-9', 64)

insert into BOOK_AUTHORS (ISBN, AUTHOR_ID)
values ('954-9656-04-7', 64)

insert into BOOK_AUTHORS (ISBN, AUTHOR_ID)
values ('1-55860-576-2', 56)

insert into BOOK_AUTHORS (ISBN, AUTHOR_ID)
values ('954-9656-03-5', 64)

insert into BOOK_AUTHORS (ISBN, AUTHOR_ID)
values ('954-9656-03-5', 56)

insert into BOOK_AUTHORS (ISBN, AUTHOR_ID)
values ('954-9656-03-5', 54)

insert into BOOK_AUTHORS (ISBN, AUTHOR_ID)
values ('3-213-77-959', 98)
```

Добавяне на клиенти на библиотеката.

```
insert into CLIENT (CLIENT_EGN, FNAME, LNAME, ADDRESS)
values ('6812138761', 'Стефан', 'Георгиев', NULL)

insert into CLIENT (CLIENT_EGN, FNAME, LNAME, ADDRESS)
values ('8809112342', 'Иван', 'Пеев', 'Пловдив, бул."Скопие" - 23')

insert into CLIENT (CLIENT_EGN, FNAME, LNAME, ADDRESS)
values ('7607053453', 'Мария', 'Петрова', 'Пловдив,бул."Източен"-17')

insert into CLIENT (CLIENT_EGN, FNAME, LNAME, ADDRESS)
values ('9112314654','Елена','Симеонова','София,бул."В.Левски"-106')

insert into CLIENT (CLIENT_EGN, FNAME, LNAME, ADDRESS)
values ('7509157765', 'Димитър', 'Георгиев', NULL)

insert into CLIENT (CLIENT_EGN, FNAME, LNAME, ADDRESS)
values ('8510231100', 'Христо', 'Иванов', NULL)
```

Добавяне на телефонните номера на клиентите.

```
insert into PHONE(OWNER_ID, PHONE)
values('6812138761', '032/651219')

insert into PHONE(OWNER_ID, PHONE)
values('6812138761', '+359889654387')

insert into PHONE(OWNER_ID, PHONE)
values('6812138761', '679088')

insert into PHONE(OWNER_ID, PHONE)
values('7607053453', '+359897825365')

insert into PHONE(OWNER_ID, PHONE)
values('7607053453', '046/56-12-76')

insert into PHONE(OWNER_ID, PHONE)
values('9112314654', '0886547324')

insert into PHONE(OWNER_ID, PHONE)
values('7509157765', '0893546221')

insert into PHONE(OWNER_ID, PHONE)
values('7509157765', '231098')

insert into PHONE(OWNER_ID, PHONE)
values('8510231100', '0887645324')

insert into PHONE(OWNER_ID, PHONE)
values('8510231100', '02/9436547')

insert into PHONE(OWNER_ID, PHONE)
values('8510231100', '046/762376')
```

Добавяне на данни за това кой клиент кои книги е вземал и кога ги е върнал, ако го е направил.

```
insert into CLIENT_BOOKS (CLIENT_EGN, ISBN, TAKEN, RETURNED)
values ('6812138761', '954-9656-03-9', '1999.10.23 18:30',
'1999.11.03 8:15:9')

insert into CLIENT_BOOKS (CLIENT_EGN, ISBN, TAKEN, RETURNED)
values ('6812138761', '954-9656-04-7', '2001.2.23 7:28:44',
'2001.3.25 19:20')

insert into CLIENT_BOOKS (CLIENT_EGN, ISBN, TAKEN, RETURNED)
values ('6812138761', '1-55860-576-2', '2003.10.23 13:02:20', null)

insert into CLIENT_BOOKS (CLIENT_EGN, ISBN, TAKEN, RETURNED)
values ('6812138761', '954-9656-03-5', '2002.2.27 10:28:21', null)

insert into CLIENT_BOOKS (CLIENT_EGN, ISBN, TAKEN, RETURNED)
values ('8809112342', '1-55860-576-2', '2000.3.25 10:1:22',
'2000.4.12 9:21:55')

insert into CLIENT_BOOKS (CLIENT_EGN, ISBN, TAKEN, RETURNED)
values ('8809112342', '954-9656-04-7', '2002.3.25 10:1:22',
'2002.4.15')

insert into CLIENT_BOOKS (CLIENT_EGN, ISBN, TAKEN, RETURNED)
values ('9112314654', '5-7789-45-675', '2004.6.21', null)

insert into CLIENT_BOOKS (CLIENT_EGN, ISBN, TAKEN, RETURNED)
values ('9112314654', '954-9656-04-7', '2004.6.21', null)

insert into CLIENT_BOOKS (CLIENT_EGN, ISBN, TAKEN, RETURNED)
values ('9112314654', '1-55860-576-2', '2004.5.11', '2004.5.29')

insert into CLIENT_BOOKS (CLIENT_EGN, ISBN, TAKEN, RETURNED)
values ('7509157765', '1-55860-576-2', '2005.5.11', null)

insert into CLIENT_BOOKS (CLIENT_EGN, ISBN, TAKEN, RETURNED)
values ('8809112342', '954-9656-04-7', '2007.4.5', null)
```

4.2. Променяне

За да демонстрираме променяне и изтриване на данни ще използваме клиента с идентификатор ‘8510231100’. Следващата команда ще промени името и фамилията му.

```
update client
set fname = 'Михаил', lname = 'Йорданов'
where client_egn = '8510231100'
```

Сега ще променим идентификатора на клиента, за да демонстрираме `ON UPDATE CASCADE` опцията на референциалното ограничение, наложено върху атрибута `OWNER_ID` по-рано. След изпълнението на тази команда стойностите

на външните ключове на записите за телефоните на този клиент ще бъдат актуализирани.

```
update client
set client_egn = '8510231101'
where client_egn = '8510231100'
```

Когато клиент връща книга, този факт се отразява в таблицата CLIENT_BOOKS по следния начин – в полето returned на записа, който представлява вземането на конкретната книга от този клиент на определена дата се въвежда датата на връщане. Да предположим, че на 08.10.2004 клиент с client_egn = ’9112314654’ връща книгата с ISBN = '5-7789-45-675', която е взел на 21.06.2004 г. Командата е следната:

```
update client_books
set returned = '2004-10-08'
where client_egn = '9112314654'
and isbn = '5-7789-45-675'
and taken = '2004-06-21'
```

Пояснение: тъй като за уникалното идентифициране на определен запис е необходима стойността на първичния ключ, тук използвахме и трите стойности в WHERE клаузата, защото цитираните там три полета формират съставния първичен ключ на таблицата. Ако не бяхме прецизни до край и пропуснехме полето isbn например, тогава щяхме да въведем факта, че този клиент е върнал всички книги, които е взел на датата 21.06.2004, а не само книгата с номер ‘5-7789-45-675’, защото щяха да бъдат обновени данните във всички записи от таблицата, които отговарят на тези две условия.

Накратко, командата действа по следния начин: променят се стойностите на изброените полета за всички записи, които отговарят на условията, посочени в WHERE клаузата. В първия пример има само един клиент с идентификатор 5, следователно от промените ще бъде засегнат само този запис, което и беше наша цел. Тъй като командата за обновяване е доста проста тя няма да бъде разглеждана подробно. Важно е да се отбележи, че в WHERE клаузата условията могат да бъдат с висока степен на сложност, но тази клауза ще бъде демонстрирана подробно при извличането на данни.

4.3. Изтриване

Да предположим, че този клиент вече не използва мобилния си номер. Тогава той може да бъде изтрит по следния начин:

```
delete from phone
where owner_id = '8510231101'
and phone = '0887645324'
```

Пояснение: подобно на командата за промяна на данните, тук също има WHERE клауза, в която се задават условията, на които трябва да отговарят записите, за да бъдат изтрити. Тъй като в таблицата PHONES първичния ключ е съставен, то използвахме стойности за всички полета, които го съставят, за да идентифицираме уникално търсения за изтриване запис. Ако WHERE клаузата бъде пропусната, то това означава, че командата ще се опита да изтрие всички записи в таблицата.

Да предположим, че въведеният клиент вече не е такъв на нашата библиотека. Можем да го изтрием със следващата команда:

```
delete from client
where client_egn = '8510231101'
```

Но към този запис на клиент има свързани телефонни номера. При задаване на ограничението за външен ключ в таблицата PHONE указахме, че при изтриване от главната таблица CLIENT операцията да бъде разширена каскадно, т.е. да бъдат изтрити и записите, в които стойността на полето owner_id = ‘8510231101’. Това са записите с телефонните номера на изтривания клиент. От казаното дотук следва, че при успешно изпълнение на изтриването на този клиент неговите телефони автоматично трябва да бъдат изтрити.

5. Конструкции за извличане на данни

5.1. SELECT

Извличането на данни от базата данни става с помощта на оператора **SELECT**. Неговият основен формат е:

```
SELECT column_list
[ INTO new_table ]
FROM table_source
[ WHERE search_condition ]
[ GROUP BY group_by_expression ]
[ HAVING search_condition ]
[ ORDER BY order_expression [ ASC | DESC ] ]
```

- SELECT – тази клауза указва колоните, които ще присъстват в резултатния набор;
- INTO – създава нова таблица и добавя в нея редовете от резултатния набор;
- FROM – указва таблицата или таблиците, от които се извличат данни;
- WHERE – указва условията за търсене, на които трябва да отговарят редовете в резултатния набор;
- GROUP BY – указва групи от записи, които да бъдат комбинирани в резултатния набор;
- HAVING – специфицира условие за групите от записи или резултат, получен от агрегатна функция. Тази клауза обикновено се използва с GROUP BY клаузата, но когато е без нея HAVING служи като WHERE клаузата;
- ORDER BY – указва начина на сортиране на редовете в резултатния набор.

В следващите примери ще бъдат демонстрирани и обяснени основните клаузи от този формат.

Резултатът от изпълнението на заявката в общия случай представлява резултатен набор (множество от редове).

1. Да се изведе информация за всички книги в библиотеката:
    a. Подредени по заглавие по азбучен ред – възходящ и низходящ;

```
SELECT * FROM BOOK ORDER BY TITLE ASC
SELECT * FROM BOOK ORDER BY TITLE DESC
```

*Пояснение: символът * заменя изброяването на всички полета от таблицата. Клаузата ORDER BY <field_name> подрежда резултатния набор по азбучен ред по посоченото поле. Сортирането може да бъде по възходящ ред – чрез добавяне на ключовата дума ASC след името на полето, низходящ – чрез добавяне на ключовата дума DESC. Ако ключова дума липсва – приема се по подразбиране ASC.*

b. Подредени по година на публикуване – възходящо и низходящо;

```
SELECT * FROM BOOK ORDER BY YEAR_PUB ASC
SELECT * FROM BOOK ORDER BY YEAR_PUB DESC
```

c. Подредени по заглавие и година на публикуване – възходящо и низходящо;

```
SELECT * FROM BOOK ORDER BY TITLE ASC, YEAR_PUB ASC
SELECT * FROM BOOK ORDER BY TITLE DESC, YEAR_PUB DESC
```

d. Подредени по цена – възходящо и низходящо;

```
SELECT * FROM BOOK ORDER BY PRICE ASC
SELECT * FROM BOOK ORDER BY PRICE DESC
```

2. Да се изведе информация за авторите, от които има книги в библиотеката:
   a. Подредени по фамилно име – възходящо и низходящо;

```
SELECT * FROM AUTHOR ORDER BY LNAME ASC
SELECT * FROM AUTHOR ORDER BY LNAME DESC
```

3. Да се изведе информация за клиентите на библиотеката, включваща име и фамилия:
   a. Подредени по малко име по азбучен ред – възходящ и низходящ;

```
SELECT FNAME, LNAME FROM CLIENT
ORDER BY FNAME
```

4. Да се изведе следната информация:
   a. Общата сума на всички книги в библиотеката:

```
SELECT SUM(PRICE) FROM BOOK
```

Пояснение: функцията SUM(field_name) извършва сумиране по колоната, зададена като аргумент и връща сумата от стойностите на всички полета с ненулева стойност от записите, участващи в заявката (в случая всички записи от таблицата със стойности, различни от NULL в това поле участват в заявката поради липсата на допълнителни условия в WHERE клауза.

b. Общата сума от цените на всички книги в библиотеката, публикувани след определена година;

```
SELECT SUM(PRICE) FROM BOOK
WHERE YEAR_PUB > 1998
```

Пояснение: в този случай се извършва сумиране по колоната PRICE. Заявката връща сумата от стойностите на всички полета с ненулева стойност от записите, участващи в нея - в случая всички записи от таблицата със стойности на полето YEAR_PUB над 1998.

c. Общите цени на книгите, групирани по година на публикуване.

```
SELECT SUM(PRICE) FROM BOOK
GROUP BY YEAR_PUB
```

Пояснение: в този случай се извършва сумиране по колоната PRICE като записите, чиито стойности се сумират, са групирани по стойността им в полето YEAR_PUB, т.е. сумирането се извършва по записите с еднаква стойност в това поле и резултатът са сумите на всички групи записи с еднаква година на публикуване.

d. Общите цени на книгите, средната цена на книга за всяка група, минимална и максимална цена на книга в група, както и брой на книги в група, групирани по година на публикуване.

```
SELECT SUM(PRICE), AVG(PRICE), MIN(PRICE), MAX(PRICE), COUNT(*) AS CNT
FROM BOOK
GROUP BY YEAR_PUB
```

*Пояснение: изразът "*COUNT(*) AS CNT*" означава, че последният атрибут в резултатната релация ще има име CNT. По този начин можем да преименуваме както атрибути, така и таблици, участващи в заявките.*

5. Да се изведе следната информация за авторите:
   a. Всички, чието малко име е "Joe";

```
SELECT * FROM AUTHOR
WHERE FNAME = 'Joe'
```

b. Всички, чиято фамилия завършва на “ев”.

```
SELECT * FROM AUTHOR
WHERE LNAME LIKE '%ев'
```

Пояснение: операторът LIKE е разширение на сравняването на низове (=) като са добавени възможности за задаване на маски в низа, който се търси. Символът ‘%’ означава, че на негово място може да бъде произволен низ. В случая се намират всички низове, които завършват на низа “ев”.

5.1.1. Вложени заявки

Вложените заявки (подзаявки) позволяват резултатът от една заявка да бъде използван като операнд в друга такава. Това се дължи на факта, че резултатът от прилагането на релационен оператор е отново релация. По този начин заявки могат да бъдат влагани, образувайки сложни изрази.

1. Да се изведе име и фамилия на всички клиенти, които имат въведен поне един телефонен номер.

```
SELECT FNAME, LNAME
FROM CLIENT
WHERE CLIENT_EGN IN (SELECT OWNER_ID FROM PHONE)
```

Вложената заявка връща резултатен набор от идентификаторите на всички клиенти, които имат въведени телефонни номера в таблицата PHONE. Външната заявка използва тоя резултатен набор, за да ограничи крайният резултатен набор само до тези клиенти, чиито идентификатори се срещат сред върнатите от вложената заявка.

Забележка: за да стане по-ясно изпълнението на заявките маркирайте вложената заявка и изпълнете само нея – така се вижда сред какви стойности сравнява външната заявка.

5.2. Оператор UNION

Можем да комбинираме (обединим) резултатните набори, получени от изпълнението на две или повече заявки, в един резултатен набор, съдържащ всички редове, върнати от заявките. Това става чрез оператора **UNION**. Неговият формат е:

```
{ < query specification > | ( < query expression > ) }
UNION [ ALL ]
< query specification | ( < query expression > )
[ UNION [ ALL ] < query specification | ( < query expression > )
[ ...n ] ]
```

Има две основни правила, на които трябва да отговарят резултатните набори, които обединяваме:

- Броят и редът на колоните на заявките, чиито резултати обединяваме, трябва да бъде еднакъв за всички заявки;
- Типовете данни на колоните, чиито данни обединяваме, трябва да бъдат съвместими, например, ако първата колона от заявките е число, то първите колони във всички заявки трябва да бъдат от числов тип, и така нататък до последната колона от заявките;
- В целия израз може да присъства само една клауза ORDER BY накрая, сортираща всички обединени резултатни набори.

По подразбиране операторът UNION премахва дублиращите се редове, но ако желаем те да бъдат запазени в резултатния набор можем да използваме ключовата дума ALL, показана в синтаксиса, и резултатният набор ще съдържа и дублиращите се редове, ако има такива. Ако тази ключова дума не е указана, дубликатите се игнорират.

2. Да се изведе информация за:
    a. Най-скъпата книга;

```
SELECT * FROM BOOK WHERE PRICE = (SELECT MAX(PRICE) FROM BOOK)
```

Пояснение: функцията MAX(field_name) връща най-голямата стойност от стойностите на полето, зададено като аргумент, от записите от таблицата, които отговарят на условията в WHERE клаузата. В случая такава няма, т.е. всички записи от таблицата със стойности, различни от NULL в това поле, участват в заявката. Функцията MIN(field_name) връща най-малката стойност при описаните по-горе условия.

    b. Най-евтината книга;

```
SELECT * FROM BOOK WHERE PRICE = (SELECT MIN(PRICE) FROM BOOK)
```

    c. Най-скъпата и най-евтина книга, обединени в един резултатен набор;

```
SELECT isbn, title, price FROM BOOK
WHERE PRICE = (SELECT MAX(PRICE) FROM BOOK)
UNION
SELECT isbn, title, price FROM BOOK
WHERE PRICE = (SELECT MIN(PRICE) FROM BOOK)
ORDER BY 2
```

Забележка: при обединения ORDER BY може да бъде приложен само на обединения резултатен набор, не и на отделните заявки. И тъй като колоните в отделните заявки могат да имат различни имена, то е позволено да укажем сортиране в ORDER BY клаузата не само по име на колона, а и по индекс – както в горния пример сортираме по втората колона от резултатния набор.

d. Най-скъпата и най-евтина книга, обединени в един резултатен набор – втори вариант. Тук добавяме колона в резултатните набори, чиято стойност ще се изчислява от израз;

```
SELECT isbn, title, price, 'Най-скъпата е публикувана ' +
CAST(year_pub as varchar)
FROM BOOK
WHERE PRICE = (SELECT MAX(PRICE) FROM BOOK)
UNION
SELECT isbn, title, price, 'Най-евтината е публикувана ' +
CAST(year_pub as varchar)
FROM BOOK
WHERE PRICE = (SELECT MIN(PRICE) FROM BOOK)
ORDER BY 2
```

Забележка: в изразната колона се налага да преобразуваме явно стойността на колонота YEAR_PUB до низ, за да няма двусмислие при прилагане на оператора +, т.е. да бъде интерпретиран като оператор за конкатенация,

e. Най-скъпата и най-евтина книга – друг вариант;

```
SELECT * FROM BOOK
WHERE PRICE = (SELECT MAX(PRICE) FROM BOOK)
   OR PRICE = (SELECT MIN(PRICE) FROM BOOK)
```

3. Да се изведе информация за клиентите на библиотеката, включваща име, фамилия и телефони на клиентите:

Пояснение: данните, които се изискват в тази задача, се намират в различни таблици, т.е. ще трябва да комбинираме в резултатния набор колони от различни таблици. Нека пробваме следващата заявка и да разгледаме получения резултатен набор.

```
SELECT CLIENT_EGN, FNAME, LNAME, OWNER_ID, PHONE
FROM CLIENT, PHONE
```

Коментар: полученият резултатен набор се състои от комбинациите на всеки запис от първата таблица с всеки от втората – това е Декартово произведение на двете таблици или т.нар. CROSS JOIN. Естествено в този резултатен набор има неверни данни. Искаме всеки клиент да бъде

комбиниран с телефоните, които са негови, т.е. искаме да останат само тези записи, които имат съвпадащи стойности в полетата CLIENT_EGN и OWNER_ID – това са верните комбинации от записи. Нека добавим това условие в WHERE клаузата на заявката.

```
SELECT CLIENT_EGN, FNAME, LNAME, OWNER_ID, PHONE
FROM CLIENT, PHONE
WHERE CLIENT_EGN = OWNER_ID
```

Коментар: това, което реализирахме с добавяне на просто условие за съпоставка в WHERE клаузата, е вътрешно съединение между използваните две таблици. В следващата тема съединенията ще бъдат разгледани подробно.

5.3. Оператор JOIN

Можем да съединим две или повече таблици чрез оператора **JOIN**. За разлика от UNION, чрез който обединявахме редовете от резултатните набори, с JOIN можем да обединим колоните на две или повече таблици. Условието за съединение може да бъде указано в FROM или в WHERE клаузата, но е препоръчително да е в FROM клаузата.

Съединенията могат да бъдат разделени на следните категории:

- Вътрешни (INNER JOIN) – това е категорията по подразбиране. Тези съединения използват оператор за сравнение на редове от две таблици, базирано на стойностите в общи колони за всяка таблица;
- Външни (OUTER JOIN) – такова съединение може да бъде ляво (LEFT OUTER JOIN или LEFT JOIN), дясно (RIGHT OUTER JOIN или RIGHT JOIN) или пълно (FULL OUTER JOIN или FULL JOIN) външно съединение – когато е указано във FROM клаузата, което е препоръчително (Transact-SQL поддържа външни съединения и в WHERE клаузата (заложени от по-старите версии), но ако ги използваме трябва много внимателно да бъдат обмислени всички условия в WHERE клаузата, за да избегнем неочаквани резултати).
- CROSS JOINS – връща Декартово произведение на двете таблици.

Резултатния набор при прилагане на LEFT OUTER JOIN включва **всички** редове от таблицата, стояща в лявата част на съединението (отляво на оператора), не само съвпадащите с дясната таблица. Ако определен ред от лявата таблица няма съвпадение в дясната таблица, то стойностите на колоните, идващи от дясната таблица ще бъдат NULL.

RIGHT OUTER JOIN е обратен на LEFT JOIN – в резултатния набор ще участват **всички** редове от дясната таблица, а ако някой от тях няма съвпадение в лявата таблица, то стойностите на колоните, идващи от лявата таблица ще бъдат NULL.

Пълното външно съединение FULL OUTER JOIN връща всички редове от лявата и дясната таблици. Ако някой от записите от едната таблица няма съвпадение в другата таблица, колоните, които идват от другата таблица ще съдържат стойности NULL.

4. Да се изведе информация за клиентите на библиотеката, включваща име, фамилия и телефони на клиентите. Да се използва вътрешно съединение:

Пояснение: ще преработим предната заявка така, че да използва оператор JOIN за коректно извличане на данните от двете таблици.

```
SELECT CLIENT_EGN, FNAME, LNAME, OWNER_ID, PHONE
FROM CLIENT INNER JOIN PHONE ON CLIENT_EGN = OWNER_ID
```

Забележка: ако пропуснем ключовата дума INNER заявката пак ще работи по същия начин, защото това е типът съединение по подразбиране. Освен това вътрешното съединение включва само записите от таблиците, които имат съответствия. В резултатния набор не присъстват клиентите, които нямат въведен поне един телефонен номер.

5. Да се изведе информация за клиентите на библиотеката, включваща име, фамилия и телефони на клиентите. В резултатният набор да участват и клиентите, които нямат въведени телефони:

Пояснение: тук ще трябва да използваме външно съединение, за да бъдат запазени записите за клиентите, които нямат съответни в таблицата с телефоните.

```
SELECT CLIENT_EGN, FNAME, LNAME, OWNER_ID, PHONE
FROM CLIENT LEFT OUTER JOIN PHONE ON CLIENT_EGN = OWNER_ID
```

Забележка: използваме ляво външно съединение, защото искаме всички записи от таблицата от лявата част на оператора JOIN, комбинирани с евентуалните им съответни от таблицата от дясната част на оператора. Ако разменим местата на двете таблици, за да постигнем същия резултат трябва да използваме дясно външно съединение.

Вижда се, че се появи клиент, който няма въведен телефонен номер. За него стойностите от таблицата PHONE са NULL. За да направим резултатния набор по-прегледен може да използваме функцията COALESCE, връщаща първия ненулев аргумент.

```
SELECT CLIENT_EGN, FNAME, LNAME, OWNER_ID, COALESCE(PHONE, '<Няма>')
PHONENUM
FROM CLIENT LEFT OUTER JOIN PHONE ON CLIENT_EGN = OWNER_ID
```

6. Да се изведе информация за:
    a. Авторите и ISBN номерата на книгите, които те са написали – да се използва съединение на таблиците с оператор JOIN във FROM клаузата;

```
SELECT AUTHOR.FNAME, AUTHOR.LNAME, BOOK_AUTHORS.ISBN
FROM AUTHOR JOIN BOOK_AUTHORS
ON AUTHOR.AUTHOR_ID = BOOK_AUTHORS.AUTHOR_ID
```

Пояснение: в тази заявка се извършва сливане на таблици във FROM клаузата.

    b. Авторите и ISBN номерата на книгите, които те са написали – да се използва съединение на таблиците в WHERE клаузата;

```
SELECT A.FNAME, A.LNAME, BA.ISBN
FROM AUTHOR A, BOOK_AUTHORS BA
WHERE A.AUTHOR_ID = BA.AUTHOR_ID
```

Пояснение: в тази заявка се извършва сливане на таблици в WHERE клаузата. Ако този пример клаузата WHERE бъде пропусната би се получило Декартово произведение между таблиците AUTHOR и BOOK_AUTHORS. Използваме също преименуване на таблиците, които използваме в заявката – AUTHOR на A, BOOK_AUTHORS на BA. По този начин можем да съединим таблица със самата себе си като използваме две различни имена в заявката за една и съща таблица – така няма да възникне двусмислие в условието за съединение в WHERE клаузата.

В горните две заявки извеждахме информация за авторите и съответните книги, които те са написали. Но съединенията, които използвахме по подразбиране са вътрешни (INNER), т.е. извеждат само записите, които имат съвпадения за посочените полета в условието на съединението. Но знаем, че има въведен автор, който все още няма публикувана книга и той не се появява в резултатния набор. Ако искаме да изведем всички автори, без значение дали имат издадени книги или не, и съответните им книги, ако са имат издадени такива, трябва да използваме външно съединение.

c. Всички автори и ISBN номерата на книгите, които те са написали, ако имат такива – да се използва външно съединение на таблиците в FROM клаузата;

```
SELECT A.*, BA.*
FROM AUTHOR A LEFT OUTER JOIN BOOK_AUTHORS BA
ON A.AUTHOR_ID = BA.AUTHOR_ID
```

Пояснение: в тази заявка се извършва ляво външно съединение на таблици в FROM клаузата. Това означава, че ще участват всички записи от лявата таблица AUTHOR, като тези от тях, които нямат съответни в дясната таблица BOOK_AUTHORS ще имат стойности NULL в колоните, идващи от дясната таблица. Същият резултат бихме постигнали и ако разменим местата на двете таблици в FROM клаузата и използваме дясно (RIGHT) вместо ляво (LEFT) външно съединение. Това е демонстрирано в следващия пример.

d. Вземаните книги и съответно клиентите, които са ги вземали. Но искаме да покажем и евентуалните нови клиенти, които все още не са вземали книги - да се използва дясно външно съединение на таблиците в FROM клаузата;

```
SELECT CB.*, C.*
FROM CLIENT_BOOKS CB RIGHT JOIN CLIENT C
ON CB.CLIENT_EGN = C.CLIENT_EGN
```

e. Книга с конкретен идентификатор от кои автори е писана;

```
SELECT AUTHOR.* FROM AUTHOR, BOOK_AUTHORS
WHERE AUTHOR.AUTHOR_ID = BOOK_AUTHORS.AUTHOR_ID
AND BOOK_AUTHORS.ISBN = '954-9656-03-5'
```

Пояснение: както в горната заявка, но тук сливане то само м/у AUTHOR и BOOK_AUTHORS като е указана конкретна стойност на идентификатор на книга.

f. Най-евтината и най-скъпата книга(и) на определен автор в отделни заявки;

```
SELECT TOP 1 WITH TIES BOOK.* FROM BOOK
WHERE BOOK.ISBN IN
(SELECT BOOK_AUTHORS.ISBN FROM BOOK_AUTHORS
  WHERE BOOK_AUTHORS.AUTHOR_ID = 64)

ORDER BY PRICE ASC
```

```
SELECT TOP 1 WITH TIES BOOK.* FROM BOOK
WHERE BOOK.ISBN IN
(SELECT BOOK_AUTHORS.ISBN FROM BOOK_AUTHORS
  WHERE BOOK_AUTHORS.AUTHOR_ID = 64)

ORDER BY PRICE DESC
```

Пояснение: тези заявки намират най-евтината и съответно най-скъпата книга, в чието писане е участвал авторът с идентификатор 64. *Самата заявка би върнала всички книги от автора, отговарящи на условията в WHERE клаузата, но изразът TOP 1 ограничава заявката да върне само записът, който е пръв в резултатния набор, а WITH TIES гарантира, че ако има още записи с параметър цена, равен на първия запис, то те също ще присъстват в резултатния набор.*

g. Авторите на най-скъпата и най-евтината книга в библиотеката;

```
SELECT author.*, book.price
FROM book, author, book_authors
WHERE book.isbn = book_authors.isbn
AND author.author_id = book_authors.author_id
AND (book.price = (SELECT min(price)FROM book)
     OR
     book.price = (SELECT max(price)FROM book)
    )
```

Пояснение: Освен стандартните свързвания в where клаузата се поставя условие за избор на най-евтина или най-скъпа книга (две еднотипни условия обединени с логическо или-OR). Във вложените заявки се извлича минималната (първата заявка) или максималната (втората) цена. Извежда се информация само за тези книги, чиито цени са еднакви с цените, извлечени от вложените заявки.

h. Авторите на най-новата и най-старата книга в библиотеката.

```
SELECT author.*, book.*
FROM book, author, book_authors
WHERE book.isbn = book_authors.isbn
AND author.author_id = book_authors.author_id
AND (book.year_pub = (SELECT min(year_pub) FROM book)
     OR
     book.year_pub = (SELECT max(year_pub) FROM book)
    )
```

Пояснение: Освен стандартните свързвания в where клаузата се поставя условие за избор на най-нова или най-стара книга (две еднотипни условия, обединени с логическо или - OR). Във вложените заявки се извлича

минималната (първата заявка) или максималната (втората заявка) година на публикуване. Извежда се информация само за тези книги, чиито години на публикуване са еднакви с годините, извлечени от вложените заявки.

7. Да се изведе информация за:

    a. Определена книга от кои клиенти е вземана;

```
SELECT distinct client.fname, client.lname
FROM client, book, client_books
WHERE client.client_egn = client_books.client_egn
AND book.isbn = client_books.isbn
AND book.title = 'Научете сами SQL (част 2)'
```

Пояснение: В from клаузата участват таблиците client (от нея извеждаме информация), book (в нея търсим заглавието на книгата) и client_book (в нея са описани вземанията "клиент взима книга"). В where клаузата първо се правят (стандартните) свързвания на трите таблици (стойност на първичен ключ е равна на стойност на външен) и освен това се проверява дали е взета конкретната книга (която търсим). За всеки запис (ред) от Декартовото произведение на трите таблици (то се извършва във from клаузата) се изчислява стойността на израза във where клаузата – ако тя е true, то информация от този ред ще се изведе в крайния резултат. DISTINCT премахва повтарящите се редове (за да не излизат еднакви записи за клиенти, които са взимали по няколко пъти книгата).

Забележка: Ако има книги с еднакви имена, но различни първични ключове ще бъде изведена информация за клиенти взели поне една от тези книги, което може да се окаже логическа грешка. Тогава по-добре ще е търсенето да става по първичен ключ, например book.isbn = '1234-567-89-0'.

    b. Книгата, вземана най-много и най-малко пъти;

```
SELECT book.title, count(*) count_books
FROM book, client_books
WHERE book.isbn = client_books.isbn
GROUP BY book.title
HAVING count(*) <= ALL(SELECT count(book.title)
                       FROM book, client_books
                       WHERE book.isbn = client_books.isbn
                       group by book.title)
    OR count(*) >= ALL(SELECT count(book.title)
                       FROM book, client_books
                       WHERE book.isbn = client_books.isbn
                       group by book.title)
```

Пояснение: Тук нещата са толкова лесни, че са излишни каквито и да са обяснения ☺. Основната заявка връща заглавията и броя вземания на всяка книга. Вложените заявки връщат броя вземания на всички книги. В клаузата HAVING се ограничава броя вземания на всяка книга да е <= или >= на всички (ALL) стойности, върнати от вложените заявки – това означава, че в резултатния набор ще участват само редовете с екстремни стойности в броя вземания – най-много и най-малко.

c. Книгата, вземана най-много и най-малко пъти – прецизиран вариант;

Горната заявка има един недостатък. Тя използва само вътрешни съединения, което значи, че ако има книга, която не е вземана, то тя не участва в сравненията и в крайния резултатен набор. Сега ще я прецизираме така, че да участват и книгите, които досега не са вземани.

```
SELECT book.title, count(client_books.isbn) count_books
FROM book left join client_books ON book.isbn = client_books.isbn
GROUP BY book.title
HAVING count(client_books.isbn) <= ALL(SELECT count(cb.isbn)
                        FROM book b left join client_books cb
                        ON b.isbn = cb.isbn
                        group by b.title)
    OR count(*) >= ALL(SELECT count(book.title)
                        FROM book, client_books
                        WHERE book.isbn = client_books.isbn
                        group by book.title)
```

Забележка: налага се да използваме външно съединение в основната заявка и във вложената, с чиито резултати се сравнява за минимална стойност на вземанията; също се налага функцията COUNT вече да има параметър колона от таблица, която би имала стойност NULL за книгите, които не са вземани нито веднъж, тъй като тази агрегатна функция брои ненулевите стойности в колоните.

d. Определен клиент кои книги е вземал;

```
SELECT DISTINCT book.*
FROM client, book, client_books
WHERE client.client_egn = client_books.client_egn
AND book.isbn = client_books.isbn
AND client.fname = 'Иван'
AND client.lname = 'Пеев'
```

Пояснение: Обяснението е подобно на задача 5.а. Всъщност вместо последните две сравнения по-логично е да използваме търсене по първичен ключ (например client.client_egn = ‘6812138761’), защото е възможно да

съществуват много хора с еднакви имена (тогава ще бъде изведена информация за книги взети от различни хора, но случайно с еднакви имена).

e. Кой клиент кои книги е взел преди определена дата;

```
SELECT client.fname + ' ' + client.lname , client_books.taken, book.*
FROM client, book, client_books
WHERE client.client_egn = client_books.client_egn
AND book.isbn = client_books.isbn
AND client_books.taken < '2002-10-23 18:30:10.000'
```

Пояснение: '2002-10-23 18:30:10.000' е един начин за записване на дата. Дата може да бъде записана в по-съкратен вариант '1999-10-23' както и с други разделители. Но такъв директен начин на сравняване на дата е непрепоръчителен, защото тази заявка става зависима от настройката на формата на датата на конкретния сървър. Долният вариант вече е прецизен, защото чрез функцията CONVERT явно преобразуваме низа до тип datetime по указан формат (120) – вж. Books Online.

```
SELECT client.fname + ' ' + client.lname , client_books.taken, book.*
FROM client, book, client_books
WHERE client.client_egn = client_books.client_egn
AND book.isbn = client_books.isbn
AND client books.taken < CONVERT(datetime, '2002-10-23 18:30', 120)
```

f. Кой клиент кои книги не е върнал;

```
SELECT book.*, client.*
FROM client, book, client_books
WHERE client.client_egn = client_books.client_egn
AND book.isbn = client_books.isbn
AND client_books.returned IS NULL
```

Пояснение: Проверката дали дадена стойност е null става с оператора IS NULL.

6. Изгледи

7. Транзакции

Транзакцията представлява логическа единица за операция върху базата данни.

Една транзакция може да съдържа (и само тогава има смисъл) няколко SQL команди. Промените, направени от командите в транзакцията не са постоянните, преди изпълнение на оператор `COMMIT`.

Оператори:

1. `BEGIN TRAN[SACTION] [tran_name]` – указва стартиране на транзакция с определено име, задаването на което е незадължително. Може да бъде използвано съкращението `BEGIN TRAN`;
2. `COMMIT TRAN[SACTION] [tran_name]` – указва потвърждаване на направените промени от командите в транзакцията, след което транзакцията приключва. Може да бъде използвано съкращението `COMMIT TRAN` или само `COMMIT`;
3. `SAVE TRAN[SACTION] savepoint_name` – указва потвърждаване на направените до срещане на този оператор промени. При отхвърляне на промени, направени от следващи операции тези то точката на запис се запазват;
4. `ROLLBACK TRAN[SACTION] [transaction_name | savepoint_name]` – отхвърля направените промени от операции в транзакцията до точката на запис, а ако такава липсва – от цялата транзакция, като с това последната приключва.

1. Транзакция, която променя адреса на клиент на библиотеката и след това добавя телефонен номер за същия клиент, след което потвърждава направените промени.

```
BEGIN TRANSACTION

UPDATE client SET address = 'Смолян, ул. Бряст - 11'
WHERE client_egn = '8809112342'

INSERT PHONE(owner_id, phone)
VALUES('8809112342', '2-45-76')

COMMIT TRANSACTION
```

2. Примерна транзакция, която променя фамилията на автор с author_id = 54, след което отхвърля направените промени.

```
BEGIN TRANSACTION

PRINT 'Фамилия преди промяната: '

SELECT lname
FROM author
WHERE author_id = 54

UPDATE author SET lname = 'Колев'
WHERE author_id = 54

PRINT 'Фамилия след промяната: '

SELECT lname
FROM author
WHERE author_id = 54

ROLLBACK TRANSACTION

PRINT 'Фамилия след отхвърлянето на промяната: '

SELECT lname
FROM author
WHERE author_id = 54
```

3. Транзакция, която променя адреса на клиент на библиотеката и след това прави опит за добавяне на телефонен номер за същия клиент, който вече е добавен от първата команда. Целта на този пример е да демонстрира проста проверка за грешка след изпълнение на всяка команда.

```
BEGIN TRANSACTION

UPDATE client SET address = 'Пловдив, бул. Цар Асен - 11'
WHERE client_egn = '8809112342'

IF (@@ERROR <> 0)
  PRINT 'Грешка при промяна на адрес!'
ELSE
  PRINT 'Адресът е променен успешно.'

INSERT PHONE(owner_id, phone)
VALUES('8809112342', '2-45-76')

IF (@@ERROR <> 0)
  PRINT 'Грешка при добавяне на телефонен номер!'
ELSE
  PRINT 'Телефонният номер е добавен успешно.'

COMMIT TRANSACTION
```

Забележка: ако изпълним отново тази транзакция втората команда за добавяне на телефон ще пропадне, защото този номер вече е въведен за този клиент и ще нарушим уникалността на ключа в таблицата. Но въпреки възникналата грешка транзакцията ще бъде потвърдена и промяната от първата команда ще е постоянна, защото в края на транзакцията се изпълнява COMMIT независимо от възникналите грешки.

Освен това от примера става ясно, че въпреки възникналите грешки (с изключение на фатални грешки от рода на липсващи обекти в базата данни, синтактични грешки, които не могат да бъдат уловени в процес на компилация и др.) промените, направени от успешните операции са потвърдени в края на транзакцията. Изводът е, че ако искаме при възникване на каквато и да е грешка в транзакцията ни същата да бъде превъртяна назад, то трябва да реализираме логика за проверка, както е показано в примера. Глобалната променлива @@ERROR съдържа стойност 0, ако не е възникнала грешка при изпълнение на последната SQL команда, и стойност различна от 0, когато е налице такава.

4. Транзакция, която въвежда нов клиент на библиотеката, поставя точка на запис, след което добавя телефонен номер на клиент, като в края отхвърля направените промени след точката на запис. Този пример демонстрира как промените, направени до точката на запис, могат да останат валидни при отхвърляне на тези след тях.

```
BEGIN TRANSACTION

INSERT client(client_egn, fname, lname)
VALUES('6612123456', 'Иван', 'Маринов')

SAVE TRANSACTION point1

INSERT PHONE(owner_id, phone)
VALUES(6, '0899756435')

ROLLBACK TRANSACTION point1

COMMIT TRANSACTION
```

Пояснения: ако искаме да отхвърлим промените до определена точка на запис, то трябва да я укажем в ROLLBACK конструкцията, в противен случай цялата транзакция бива отхвърлена.

7.1. Нива на изолация на транзакциите в T-SQL

Нивото, на което една транзакция е готова да приема непотвърдени данни, се нарича ниво на изолация (isolation level).

Нивото на изолация представлява степента, до която една транзакция трябва да бъде изолирана от другите транзакции. По-ниското ниво увеличава възможностите за едновременна работа, но за сметка на коректността на данните. И обратно, по-високо ниво на изолация гарантира за коректност на данните, но може да се отрази отрицателно на едновременната работа.

В езикът Transact-SQL са реализирани 4 нива на изолация със следното поведение:

- READ UNCOMMITTED – прочитане на непотвърдени данни;
- READ COMMITTED – четене само на потвърдени данни;
- REPEATABLE READ – повторяемо четене;
- SERIALIZABLE – сериализируемо.

Транзакциите имат различно поведение в зависимост от това какво ниво на изолация им е зададено.

Ако транзакциите се изпълняват на сериализируемо ниво на изолация всички едновременни припокриващи се транзакции са гарантирано сериализируеми.

Следната таблица илюстрира поведението при различните нива на изолация.

Ниво на изолация	Незавършена зависимост	Неконсистентен анализ	Четене на фантоми
Непотвърдено четене	Да	Да	Да
Потвърдено четене	Не	Да	Да
Повторяемо четене	Не	Не	Да
Сериализируемо	Не	Не	Не

7.2. Deadlock ситуации

Тези ситуации се случват, когато две или повече транзакции се конкурират за едни и същи ресурси, като в същото време всяка от тях може да продължи само ако другата освободи блокираните от нея ресурси. В такъв момент те биха стигнали до момент на безкрайно изчакване.

В MSSQL Server системата автоматично открива такива ситуации и ги разрешава, прекъсвайки транзакцията, която би „струвала по-малко” – т.е. тази, която по-лесно би била превъртяна назад, т.е. тази, която е направила най-малко промени до момента. Ако ситуацията не бъде разрешена се прекъсва следващата транзакция и така до решаване на ситуацията.

7.2.1. Пример на deadlock ситуация

Най-прост вариант за демонстрация е чрез осъществяване на две връзки към сървъра и стартиране на приложените скриптове в разстояние на 5 секунди.

```
set transaction isolation level read committed

begin tran

update book
set price = 14.69
where isbn = '3-213-77-959'

waitfor delay '00:00:05'

select lname
from author
where author_egn = '9876274857'

commit
```

```
set transaction isolation level read committed

begin tran

update author
set lname = 'Коелю-2'
where author_egn = '9876274857'

waitfor delay '00:00:05'

select price
from book
where isbn = '3-213-77-959'

commit
```

За избраната за жертва транзакция ще бъде изведено съответното съобщение.

7.2.2. Как да бъдат избягвани deadlock ситуациите

За да бъде намален риска от такива ситуации могат да бъдат следвани някои прости правила:

- Обектите да се използват в един и същ ред – когато има обработка на таблици от повече от една транзакция е добре (ако е възможно) последователността на работа с таблиците в транзакциите да е един и същ;
- Транзакциите да бъдат възможно най-краткотрайни;
- Използване на най-ниското възможно ниво на изолация;

- Неизползване на транзакции, извършващи запитване към потребител и изчакващи негово решение или такива, които биха изчаквали настъпване на определено събитие, за да продължат.

8. Съхранени процедури и функции

8.1. Опростен синтаксис и примери

Съхранените процедури представляват компилирани пакети от команди, които се съхраняват на сървъра и могат да бъдат изпълнявани по всяко време.

Опростен синтаксис за деклариране на процедура е следният:

```
CREATE PROC[EDURE] procedure_name [{@parameter data_type} [OUTPUT]] [,...n]

AS sql_statement [ ...n ]
```

Пояснение: след името на процедурата може да следва списък с параметри. В този списък могат да бъдат указани както входни, така и изходни параметри, които представляват върнат от процедурата резултат. Тяхната стойност се предава по адрес. Броят на параметрите в една процедура може да бъде до 2100.

За да демонстрираме най-простия вариант за създаване на процедура ще изпълним следната команда:

```
create procedure get_authors
as
select * from author
```

След успешното изпълнение на тази команда ще имаме готова за употреба процедура с име get_authors, чието действие е да извлича всички данни от таблицата AUTHOR. Можем да я изпълним по един от следните начини:

```
execute get_authors

или

exec get_authors
```

Ще създадем процедура, която ще извлича име, фамилия и брой книги на автор по зададен като входен параметър ЕГН.

```
create procedure get_author @au_id int
as

select a.fname, a.lname, count(ba.author_id) pub_count
from author a left join book_authors ba
on a.author_id = ba.author_id
where a.author_id = @au_id
group by a.author_id, a.fname, a.lname
```

Така създадената процедура може да бъде извикана по един от следните два начина:

```
exec get_author 64

или

exec get_author @au_id = 64
```

Т.е. параметрите могат да бъдат указани анонимно, като им се зададат стойностите подред, или да бъдат указани явно по име, така че редът на подаване да не е от значение.

Следващият пример ще демонстрира процедура с изходни параметри. Тази процедура ще връща като параметри броя на авторите и броя на книгите в библиотеката. Декларирането на параметър като изходен става с ключовата дума OUTPUT след декларацията на параметъра.

```
create procedure get_author_book_count @au_count int output,
@book_count int output
as

select @au_count = count(*) from author
select @book_count = count(*) from book
```

Така създадената процедура ще извикаме в пакет от команди.

```
declare @ac int, @bc int
exec get_author_book_count @ac output, @bc output
print 'Authors count: ' + cast(@ac as varchar)
print 'Books count: ' + convert(varchar, @bc)
```

Пояснение: декларираме две променливи, които ще предадем (по адрес) като параметри на процедурата. Тук е важно да се укаже, че предаваните параметри са изходни (с ключовата дума OUTPUT), в противен случай след изпълнение на процедурата техните стойности ще са NULL. След изпълнение на процедурата отпечатваме брой автори и брой книги в

библиотеката. Явното преобразуване се налага, защото типовете не са съвместими. Демонстрирани са две функции за явно преобразуване на типове.

8.2. Вложени съхранени процедури

Съхранените процедури могат да бъдат влагани и да извикват други процедури. Процедура, извикана от друга процедура, може от своя страна да извика трета процедура. При такова извикване първата процедура има ниво на влагане 1, втората – 2, а третата – 3. Максималното ниво на вложеност може да е 32, като при надвишаването му ще бъде предизвикана фатална грешка, пакетът ще бъде отхвърлен, а всички стартирани транзакции превъртени назад.

За да се определи нивото на вложеност може да се провери стойността на системната променлива @@NESTLEVEL.

8.3. Рекурсия в съхранените процедури

При съхранените процедури може да бъде използвана рекурсия, но е важно да се съобразяваме с максималното ниво на вложеност 32. За да демонстрираме рекурсия ще създадем процедура, която изчислява факториел. В този пример също ще демонстрираме и връщана стойност от процедура (не в параметър) чрез оператора RETURN.

```
create procedure factorial @n int
as

declare @one_less int, @answer int

if (@n < 0 or @n > 12)
 begin
  print 'Непозволена стойност на параметъра!'
  return -1
 end

if (@n = 0 or @n = 1)
  set @answer = 1
else
begin
  set @one_less = @n - 1
  exec @answer = factorial @one_less

  if (@answer = -1)
    return -1

  set @answer = @answer*@n

  if (@@error <> 0)
    return -1
end

return @answer
```

Във версия 2000 на сървъра при създаване на процедурата интерпретаторът извежда посоченото по-долу съобщение, защото процедурата прави обръщение към процедура, която все още не съществува, т.е. към самата себе си:

```
Cannot add rows to sysdepends for the current stored procedure because it
depends on the missing object 'factorial'. The stored procedure will still
be created.
```

След като вече сме създали процедурата можем да я използваме за изчисляване на факториел. Следният пример изчислява 9!.

```
declare @result int
exec @result = factorial 9
print '9! = ' + cast(@result as varchar)
```

Още повече, ще я използваме за генериране на стандартна таблица на факториелите до 12, като я извикаме в цикъл.

```
declare @answer int, @number int

set @number = 0

while (@number <= 12)
begin
  exec @answer = factorial @number

  if (@answer = -1)
  begin
    print 'Грешка при изпълнение!'
    return
  end

  print cast(@number as varchar) + '! = ' + cast(@answer as varchar)

  set @number = @number + 1
end
```

8.4. Функции

Освен процедури, потребителят може да дефинира и свои собствени функции. Разликата между процедурите и функциите е, че функциите връщат като резултат скаларна стойност и могат да бъдат използвани директно в SQL заявка – в това се изразява основната разлика между функциите и процедурите. Те също могат да бъдат изпълнявани и с командата EXECUTE. Освен скаларна стойност функциите също могат да връщат и резултатен набор.

Ще представим опростен синтаксис за създаване на функция, връщаща скаларна стойност като резултат:

```
CREATE  FUNCTION function_name
( [ { @parameter_name [AS] scalar_parameter_data_type [ = default ] } [
,...n ] ] )
RETURNS scalar_return_data_type
 [ AS ]
BEGIN
    function_body
    RETURN scalar_expression
END
```

Следващият пример демонстрира създаване и употреба на функция, връщаща скаларна стойност.

```
create function client_books_count (@cl_egn char(10))
returns varchar(200)
as
begin
  declare @cnt int, @name varchar(30), @taken varchar, @not_returned
varchar

  select @name = c.fname + ' ' + c.lname,
         @taken = count(cb.taken), @not_returned = count(cb.returned)
  from client_books cb right join client c
  on c.client_egn = cb.client_egn
  where c.client_egn = @cl_egn
  group by c.client_egn, c.fname, c.lname

  return 'Клиентът ' + @name + ' (ЕГН = ' + @cl_egn +
         ') е взел ' + cast(@taken as varchar) +
         ' книги, не е върнал ' + cast(@not_returned as varchar)
end
```

Ще демонстрираме тази функция в проста заявка, извличаща идентификаторите на клиентите, които се подават като параметър на функцията. Ще забележим, че функцията се изпълнява по веднъж за всеки ред от резултатния набор.

```
select dbo.client_books_count(client_egn)
from client
```

Следният опростен синтаксис е за функция, връщаща резултатен набор като резултат:

```
CREATE FUNCTION function_name
( [ { @parameter_name [AS] scalar_parameter_data_type [ = default ] } [
,...n ] ] )
RETURNS TABLE
[ WITH < function_option > [ [,] ...n ] ]
[ AS ]
RETURN [ ( ] select-stmt [ ) ]
```

Следващият пример демонстрира създаване и употреба на функция, връщаща резултатен набор.

```
create function BooksByAuthor()
returns table
as
return (select a.author_id, a.lname, b.title
        from author a, book_authors ba, book b
        where a.author_id = ba.author_id
        and ba.isbn = b.isbn
       )
```

Демонстрираме изпълнение на горната функция чрез следната команда:

```
select * from BooksByAuthor()
```

9. Тригери

Когато се налага към базата данни да се добавят сложни правила за цялостност и бизнес-логика, могат да се използват тригери. Тригерите представляват специален тип съхранени процедури, които се прилагат върху таблици и изгледи. Но те не могат да бъдат извиквани както съхранените процедури. Те се задействат при настъпване на събитие за модификация на данните. Един и същ тригер може да се задейства от повече от едно събитие, а в него може да се дефинира бизнес-логика, която да обработва всеки тип събитие. Не може да се създаде тригер за временна или системна таблица.

9.1. Тригерни събития

Три събития автоматично могат да задействат тригер:

- ✓ INSERT – задейства тригерите, които са декларирани, че ще се задействат при добавяне на данни;
- ✓ UPDATE – задейства тригерите, които са декларирани, че ще се задействат при промяна на данни;
- ✓ DELETE - задейства тригерите, които са декларирани, че ще се задействат при изтриване на данни.

9.2. Изпълнение на тригер

Когато добавяне или обновяване на редове задейства тригер, той съхранява новите или модифицирани данни в псевдотаблица, наречена *inserted*. Когато изтриване задейства тригер, той съхранява изтритите записи в псевдотаблица, наречена *deleted*. Също така при промяна на записи таблицата deleted съдържа тези записи във вида им преди промяната, докато inserted съдържа променените записи.

В SQL Server 2000 съществуват два класа тригери: INSTEAD OF и AFTER. Тригерите INSTEAD OF се стартират вместо тригериращото действие. AFTER тригерите се задействат като допълнение на тригериращото действие и са класът тригери по подразбиране. В таблицата са показани основните различия между класовете тригери.

Характеристика	**INSTEAD OF**	**AFTER**
Прилага се към	Таблица или изглед.	Таблица.
Допустим брой	Само по един на тригериращо действие.	Може да има повече от един на таблица.
Ред на изпълнение	Няма значение, защото е	Изпълняват се в произволен

	само един.	ред, ако не е зададен ред на изпълнение чрез системната процедура sp_setttriggerorder.

9.3. Опростен синтаксис за създаване на тригер

Ще използваме следния опростен синтаксис за дефиниране на тригер:

```
CREATE TRIGGER trigger_name
ON { table | view }
FOR | AFTER | INSTEAD OF {[ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ]}
AS
  sql_statement [ ...n ]
```

Посочената команда създава тригер с име *trigger_name*. В клаузата ON се указва към кой обект се прилага (таблица или изглед).

Чрез клаузите FOR, AFTER и INSTEAD OF се указва класът на тригера (FOR и AFTER са синоними). След това се задава тип събитие, при настъпване на което да се задейства тригера – валидни типове са INSERT, UPDATE, DELETE. Типовете могат да бъдат изброени в произволен ред, ако са повече от един.

Клаузата AS бележи началото на кода, който реализира бизнес-логиката на тригера.

9.4. Изтриване, промяна и забраняване

9.4.1. Изтриване

За да изтрием тригер просто трябва да изпълним следната команда:

```
DROP TRIGGER trigger_name
```

Забележка: ако изтрием таблица или изглед, то всички тригери, свързани към него, също се изтриват.

9.4.2. Промяна

За да променим тригер можем просто да го изтрием и да го създадем наново. Но ако искаме да прескочим операцията с изтриването командата е следната:

```
ALTER TRIGGER trigger_name
ON { table | view }
FOR | AFTER | INSTEAD OF {[ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ]}
AS
  sql_statement [ ...n ]
```

9.4.3. Забраняване

За да забраним изпълнението на тригер за конкретна таблица командата е следната:

```
ALTER TABLE table_name DISABLE TRIGGER trigger_name
```

За да позволим отново изпълнението на тригер трябва да изпълним следната команда:

```
ALTER TABLE table_name ENABLE TRIGGER trigger_name
```

Забележка: ако искаме да забраним или разрешим всички тригери на таблицата вместо името на тригера трябва да използваме ключовата дума All вместо името на тригера.

9.5. Създаване на тригери

Ще създадем прост тригер, който ще се изпълнява след добавяне на запис в таблицата AUTHOR и ще извежда всички записи от псевдотаблицата inserted, за да видим реално добавените записи.

```
create trigger insertauthor
on author
after insert
as

print 'New records in author were inserted!'
print 'The inserted records are:'
select * from inserted
```

Да изпробваме сработването на тригера, като въведем един запис в таблицата с командата:

```
insert into AUTHOR (AUTHOR_ID, FNAME, LNAME)
values (1234, 'John', 'Smith')
```

Сега ще изпробваме въвеждането на няколко записа с една команда insert. Следващата команда взема изброените полета на всички редове от таблицата AUTHOR и ги въвежда в таблицата CLIENT – не особено смислена команда, но достатъчно демонстративна.

```
insert into client(client_egn, fname, lname)
select author_id, fname, lname from author
```

Извод: тригерът се изпълнява след всяка команда, а не след добавяне на всеки нов запис, т.е. ако искаме да има някаква логика, която да обработва всеки нов добавен ред, то трябва да обработваме редовете от таблицата inserted.

В следващия пример ще демонстрираме INSTEAD OF тригер, който ще се изпълнява вместо добавяне или редактиране на запис от таблицата CLIENT. По този начин ще забраним добавяне и променяне на записи в тази таблица, като ще извеждаме кои данни няма да бъдат добавени или променени.

```
create trigger insupdclient
on client
instead of insert, update
as

print 'Следните записи няма да бъдат добавени или променени:'
select * from inserted
```

За да проверим този тригер ще изпълним долните две команди една по една, за да видим резултатите от изпълнението им:

```
update client
set fname = 'Спас'
where client_egn = '6812138761'

insert into client(client_egn, fname, lname)
select substring(client_egn, 1, 5) + 'N', fname + ' 2', lname + ' 2'
from client
```

Ще усложним малко следващия пример, като реализираме проверка за това дали е променена конкретно фамилията на клиент на библиотеката и ако е, то ще запишем старата и новата фамилия в друга таблица. Първо ще създадем другата таблица като временна такава.

Забележка: за да реализираме описаната задача ще трябва да забраним или изтрием тригера от предишния пример, който се изпълнява вместо добавяне или промяна на данните в таблицата CLIENT.

```
create table ##client_history
(
  client_egn char(10),
  old_name varchar(15),
  new_name varchar(15)
)
```

В таблицата CLIENT_HISTORY ще записваме идентификатор на клиент, стара и нова фамилия.

Кодът за създаване на тригера е следният:

```
create trigger clienthist
on client
for update
as

if update(lname)
begin
  insert into ##client_history(client_egn, old_name, new_name)
  select i.client_egn, d.lname, i.lname
  from inserted i, deleted d
  where i.client_egn = d.client_egn
end
```

Реализираната бизнес-логика е следната – ако при промяна на данните в таблицата CLIENT е променена стойност в полето lname, то правим съединение на псевдотаблиците inserted и deleted, за да вземем от deleted старата стойност, а от inserted новата стойност и да ги запишем във временната таблица.

След промяна на фамилията на двама клиенти ще извлечем данните от системната таблица, за да видим какво е записал тригерът в нея.

```
update client
set lname = 'Димитров'
where client_egn in ('7509157765', '9112314654')

select * from ##client_history
```

Да предположим, че в таблицата с авторите трябва да имаме колона, в която да се записват броя книги, които имаме от всеки автор. В този случай можем да реализираме тригер, който да актуализира стойността на това поле при добавяне или изтриване на книга в таблицата BOOK_AUTHORS, която съдържа информация за това кой автор коя книга е писал. За целта първо добавяме колона, която да представя броя книги от автора в библиотеката.

```
alter table author
add book_cnt int
```

Сега можем да създадем тригера, който да се грижи за обновяването на стойностите в тази колона.

```
create trigger updatebookcount
on book_authors
for insert, update, delete
as
update author
set publ = (select count(*)
            from book_authors ba
            where author.author_id = ba.author_id)
where author_id in (select author_id from inserted
                    union
                    select author_id from deleted)
```

За реализиране на описаната логика е достатъчен кода без последната WHERE клауза, но така ще се обновяват всеки път данните за всички автори, без значение дали техни книги са били изтрити или прибавени. Ако искаме да сме по-прецизни и да се обновяват само полетата за авторите, за които сме въвели или изтрили книги, то ще трябва да използваме посочения код Данните от псевдотаблиците inserted и deleted се използват за определяне на това книги за автори с кои идентификатори са били изтрити или добавени. Като използваме обединение на резултатните набори избягваме проверка за това дали операцията е изтриване или добавяне.

За да изпробваме ефекта от тригера можем първо да добавим данни, да видим промяната, след което да изтрием няколко записа и отново да видим промяната.

```
insert into BOOK_AUTHORS (ISBN, AUTHOR_ID)
values ('1-55860-576-2', 54)

delete book_authors
where isbn = '954-9656-03-5'
```

В изискванията към базата данни за библиотеката се споменава, че всяка книга може да бъде вземана от различни клиенти, но в различни моменти от времето. Ще реализираме тази логика чрез тригер, който да проверява дали книгите, които взема клиент, са налични в момента. Вземанията на книгите се регистрират с добавяне на нов запис в таблицата CLIENT_BOOKS. Следователно трябва да се реализира тригер, който при добавяне на нови записи в тази таблица да проверява съответните книги, които клиентът иска да вземе, дали не са вече взети.

Най-просто казано тригерът трябва да провери дали сред записите в псевдотаблицата inserted има книги, които са били взети преди и не са върнати все още, т.е. стойността в полето RETURNED е NULL. Тук внимание трябва да се обърне на това, че тригерът ще се изпълни след добавянето на редовете в таблицата, т.е. тези книги дори и да не са били взети преди, то нововъведените

записи за вземането им вече ще са налични в CLIENT_BOOKS. Поради тази причина при сравняването на записите за ново взетите книги и тези от преди трябва да изключим тези на ново взетите. Един от вариантите това да бъде направено е да сравняваме за различие стойностите на полетата CLIENT_ID и TAKEN и ако има разлика поне в едно от тях, можем да приемем, че този запис е за предишно вземане на книга, а не от новите, защото трите полета ISBN, CLIENT_ID и TAKEN образуват първичен ключ, т.е. комбинацията от стойностите им не може да се повтори.

```
create trigger checkbookexist
on client_books
for insert
as

if exists (select *
           from client_books cb, inserted
           where cb.isbn = inserted.isbn
           and cb.returned is null
           and (cb.client_egn <> inserted.client_egn or
                cb.taken <> inserted.taken))
begin
  rollback
  raiserror('Някои от книгите не са налични!', 16, 1)
end
```

Използваме EXIST клаузата, която проверява за наличие на записи в заявката след нея. Ако такива има, то резултатът е true. При такъв резултат отхвърляме добавените записи чрез превъртане назад на неявната транзакция, стартирана от добавянето. Функцията RAISERROR извежда съобщение с код на грешката.

Можем да изпробваме действието на този тригер като се опитаме да регистрираме вземане на вече взета книга със следния код:

```
insert into CLIENT_BOOKS (CLIENT_EGN, ISBN, TAKEN)
values ('7509157765', '954-9656-04-7', '2001.2.23 11:28:44')
```

10. Курсори

10.1. Въведение

Релационните бази данни по принцип са ориентирани за работа с набори от данни, т.е. извличане, модифициране, изтриване на множество от редове, а не ред по ред. От друга страна, езиците за програмиране и приложенията клонят към работа с отделни записи. Напр., когато служител в нашата библиотека трябва да промени данните за някои клиенти нормално е той да използва някое приложение, с което да позиционира в списъка с клиенти, да актуализира данните на избран клиент, след което да продължи с останалите от списъка. Несъответствието между подхода, базиран на записи и този, ориентиран към набори, понякога трябва да бъде елиминирано чрез някакъв механизъм: курсорите.

Ще си представяме курсора като именуван резултатен набор, в който можем да се придвижваме ред по ред, като във всеки момент имаме текущ ред, на който сме позиционирали. На този текущ ред можем да променим определени данни или просто да ги извлечем, или да изтрием текущо позиционирания ред. Курсорите успешно могат да бъдат използвани в тялото на тригер или процедура.

Има различни типове курсори, но само сървърните курсори на Transact-SQL ще представляват интерес за нас, останалите няма да бъдат разглеждани.

За да работим с курсор трябва да предприемем следните стъпки:

1. Деклариране на курсора чрез конструкцията DECLARE.
2. Отваряне на курсора.
3. Извличане на ред от курсора чрез конструкцията FETCH. Това обикновено става в цикъл, в който на всяка итерация се проверява стойността на системната променлива @@FETCH_STATUS след всяко извличане. Ако стойността й е:
   - ✓ 0 – извличането е било успешно;
   - ✓ -1 – няма повече редове в курсора;
   - ✓ -2 – редът вече не съществува в курсора, т.е. той е бил изтрит след отваряне на курсора или променен така, че вече не отговаря на условията, на които отговарят редовете, участващи в резултатния набор на курсора.
4. Прочитане, актуализиране или изтриване на реда, на който е позициониран курсора.

5. Затваряне на курсора. Това прекратява активното действие на курсора. Той все още може да бъде отворен без да се налага да се декларира отново.
6. Освобождаване на курсора, за да бъдат освободени структурите от данни, които съставят курсора.

10.2. Опростен синтаксис

Ще използваме само някои от възможностите на курсорите с демонстративна цел, затова синтаксисът, който ще бъде представен е много опростен.

```
DECLARE cursor_name CURSOR
FOR select_statement
```

10.3. Примери

Първият пример ще демонстрира прочитане на данни ред по ред от курсор. Резултатният набор ще съдържа всички клиенти на библиотеката с известен адрес.

```
declare @cl_egn char(10), @f_name varchar(15), @l_name varchar(15)

-- 1
declare client_cursor cursor for
select client_egn, fname, lname
from client
where address is not null

-- 2
open client_cursor
-- 3
fetch next from client_cursor into @cl_egn, @f_name, @l_name

print 'Клиенти, чиито адреси са известни, са: '
print '------------------------------'

while @@fetch_status = 0
begin
-- 4
   print @f_name + ' ' + @l_name + ' (ЕГН = ' + @cl_egn + ')'

   fetch next from client_cursor into @cl_egn, @f_name, @l_name
end

-- 5
close client_cursor
-- 6
deallocate client_cursor
```

Тук с коментари са отбелязани обяснените по-горе стъпки.

В следващия пример ще бъде демонстрирано обновяване и изтриване на ред от курсор. За целта резултатният набор ще съдържа всички записи от таблицата с клиентите. За целта ще се наложи да изтрием тригера clienthist от примерите за тригерите, защото той използва (може би) несъществуваща в момента временна таблица CLIENT_HISTORY.

```
declare @cl_egn int, @f_name varchar(15), @l_name varchar(15)

declare client_cursor cursor for
select client_egn, fname, lname from client

open client_cursor
fetch next from client_cursor into @cl_egn, @f_name, @l_name

set nocount on

while @@fetch_status = 0
begin
   if (@cl_egn = '7509157765')
   begin
     print 'Фамилията на ' + @f_name + ' ' + @l_name +
           ' ще бъде променена.'
     update client set lname = 'Вълчев'
     where current of client_cursor
   end

   if (@cl_egn = '6612123456')
   begin
     print 'Клиентът ' + @f_name + ' ' + @l_name + ' ще бъде изтрит!'
     delete client where current of client_cursor
   end

   fetch next from client_cursor into @cl_egn, @f_name, @l_name
end

close client_cursor
deallocate client_cursor

print ''
select client_egn, fname, lname from client

set nocount off
```

От кодът се вижда, че при промяна или изтриване на данни единствената разлика от познатите ни досега команди за тези операции е в WHERE клаузата, където се задават критерии, на които записите да отговарят, за да бъдат засегнати от съответната команда. При работата с курсори изразът CURRENT OF cursor_name определя, че текущият ред на курсора ще бъде засегнат от съответната команда за обновяване или изтриване. Използваната опция SET NOCOUNT ON (OFF) забранява (позволява) извеждането на броя засегнати редове.