

Пловдивски университет “Паисий Хилендарски”

Факултет по Математика и Информатика

Катедра “Компютърни системи”

Учебно помагало

по

„Бази от данни“

Автори: Гл. ас. д-р Георги Чолаков
Гл. ас. д-р Магдалена Веселинова

Рецензенти: Проф. д-р Станимир Стоянов
Доц. д-р Ася Стоянова-Дойчева

ISBN: 978-619-202-541-0

2019 г.

Съдържание

Съдържание	2
Списък с фигурите	5
Списък с примерите	8
1. Въведение	11
1.1. За кого е подходящо?.....	11
1.2. Цели на курса	11
1.3. Защо бази от данни?	11
1.4. Релационни бази от данни	12
1.5. Означения	13
2. Език SQL	14
2.1. В тази тема.....	14
2.2. Език за дефиниране на данни	14
2.3. Език за манипулиране на данни	17
2.4. Език за контрол на достъпа	19
2.5. Език за контрол на транзакциите	20
2.6. Типове данни	20
2.7. Оператори, идентификатори, коментари	23
2.8. Стандартни функции	24
2.9. Задачи.....	28
3. Създаване на база данни за търговска фирма.....	30
3.1. В тази тема.....	30
3.2. Теоретични понятия	30
3.3. Проектиране на базата данни	31
3.4. Нормализиране на модела	35
3.5. Създаване на таблиците	41

3.6.	Манипулиране на данните	57
3.7.	Задачи	58
4.	Извличане на данни	59
4.1.	В тази тема.....	59
4.2.	SELECT	59
4.3.	UNION	62
4.4.	INTERSECT	65
4.5.	MINUS	68
4.6.	JOIN	70
4.7.	Други.....	82
5.	Изгледи	87
5.1.	В тази тема.....	87
5.2.	Теоретични понятия	87
5.3.	Създаване на изгледи	88
5.4.	Манипулиране на данни чрез изглед.....	90
5.5.	Задачи	92
6.	Транзакции	93
6.1.	В тази тема.....	93
6.2.	Теоретични понятия	93
6.3.	Оператори за контрол на транзакциите	94
6.4.	Примери	95
6.5.	Задачи	96
6.6.	Вижте още... ..	97
7.	Съхранени процедури и функции	102
7.1.	В тази тема.....	102
7.2.	Процедури.....	102
7.3.	Функции.....	105

7.4. Задачи	108
8. Курсори	109
8.1. В тази тема.....	109
8.2. Теоретични понятия	109
8.3. Създаване и използване на курсори	110
8.4. Задачи	112
9. Тригери	113
9.1. В тази тема.....	113
9.2. Теоретични понятия	113
9.3. Изтриване, промяна и забраняване	114
9.4. Създаване на тригери	115
9.5. Задачи	119
10. Приложение 1 – решения на задачите	120
Език SQL	120
Създаване на база данни за търговска фирма	121
Извличане на данни	122
Изгледи	125
Транзакции	125
Съхранени процедури и функции	126
Курсори	128
Тригери	129
11. Приложение 2 – ресурси	131
Използвани източници	132

Списък с фигурите

Фигура 3-1. Концептуален (Entity/Relationship) модел на данните.....	33
Фигура 3-2. Концептуален (Entity/Relationship) модел с идентификатори в обектите (без ORDERS).....	35
Фигура 3-3. Концептуален модел след нормализацията.....	38
Фигура 3-4. Физически модел на базата данни.	40
Фигура 3-5. Схема на първата създавана таблица – REGIONS.....	41
Фигура 3-6. Схемата след създаване на таблицата COUNTRIES.....	42
Фигура 3-7. Схемата след създаване на таблицата CUSTOMERS.	43
Фигура 3-8. Схемата след създаване на таблицата JOBS.....	45
Фигура 3-9. Схемата след създаване на таблицата EMPLOYEES.	46
Фигура 3-10. Схемата след създаване на таблицата DEPARTMENTS.....	49
Фигура 3-11. Схемата след създаване на таблицата ORDERS.	52
Фигура 3-12. Схемата след създаване на таблицата PRODUCTS.....	53
Фигура 3-13. Схемата след създаване на таблицата ORDER_ITEMS.	56
Фигура 4-1. Резултати от SELECT – данни на служителите.....	60
Фигура 4-2. Резултати от SELECT – продукти с цена над 2000.....	61
Фигура 4-3. Резултати от SELECT – брой на всички служители.....	61
Фигура 4-4. Резултати от SELECT – брой служители по отдели.....	62
Фигура 4-5. Резултатно множество от записи при оператор UNION.....	63
Фигура 4-6. Резултати от UNION – държави с клиенти или отдели.....	64
Фигура 4-7. Резултати от UNION ALL.....	65
Фигура 4-8. Резултатно множество от записи при оператор INTERSECT (сечение).....	66
Фигура 4-9. Резултати от INTERSECT.....	66
Фигура 4-10. Резултатно множество от записи при оператор MINUS (разлика).....	68
Фигура 4-11. Резултат от оператор EXCEPT (разлика).....	69

Фигура 4-12. Диаграма на оператор NATURAL JOIN.....	70
Фигура 4-13. Резултат от оператор NATURAL JOIN.	71
Фигура 4-14. Резултат от оператор INNER JOIN.....	72
Фигура 4-15. Резултат от оператор INNER JOIN.....	73
Фигура 4-16. Диаграма на оператор LEFT OUTER JOIN.....	73
Фигура 4-17. Резултат от оператор LEFT OUTER JOIN.	74
Фигура 4-18. Диаграма на оператор RIGHT OUTER JOIN.	75
Фигура 4-19. Резултат от оператор RIGHT JOIN.....	75
Фигура 4-20. Диаграма на оператор FULL OUTER JOIN.	76
Фигура 4-21. Резултат от оператор FULL JOIN.	77
Фигура 4-22. Резултат от оператор EQUI-JOIN.	78
Фигура 4-23. Резултат от оператор SEMI-JOIN.	79
Фигура 4-24. Резултат от оператор ANTI JOIN.	80
Фигура 4-25. Резултат от оператор CROSS JOIN.	81
Фигура 4-26. Резултат от оператора TOP.....	83
Фигура 4-27. Резултат: 7-те продукта с най-ниска цена в Oracle.	83
Фигура 4-28. Резултат от оператора TOP WITH TIES.	84
Фигура 4-29. Резултат от OFFSET и FETCH.....	85
Фигура 4-30. Резултат от OFFSET и FETCH за порция от 5 реда.	86
Фигура 6-1. Стартиране на транзакция T1 без потвърждаване/отхвърляне на промените. 98	
Фигура 6-2. Стартиране на транзакция T2, прочита непотвърдените данни.	99
Фигура 6-3. Стартиране на транзакция T2 с ниво на изолация, непозволяващо прочитане на непотвърдени данни.	99
Фигура 6-4. Демонстрация на deadlock – резултати от първата транзакция.	101
Фигура 6-5. Демонстрация на deadlock – стартиране на втора транзакция.	101
Фигура 7-1. Резултат от изпълнение на процедурата CUSTOMER_ORDERS.	103
Фигура 7-2. Резултат от изпълнение на процедурата EMPLOYEES_ORDERS.	104

Фигура 7-3. Резултат от изпълнение на процедурата GET_CUSTOMERS_EMPLOYEES_COUNT.	105
Фигура 7-4. Резултат от изпълнение на функцията DEPARTMENTS_SALARY.	106
Фигура 7-5. Резултат от изпълнение на функцията EMPLOYEES_JOBS.	107
Фигура 9-1. Копие на новия ред в inserted таблицата и празна таблица deleted.	116
Фигура 9-2. Актуализирани редове в inserted таблицата.	116
Фигура 9-3. Редове в deleted таблицата.	117
Фигура 9-4. Генериране на грешка в тригер.	119
Фигура 10-1. Резултат от изпълнението на тригера TRG_CUSTOMERS_HIST.	130

Списък с примерите

Пример 4-1. Да се изведат имената, датите на назначаване и заплатите на всички служители.	60
Пример 4-2. Да се изведат всички данни за продуктите, с цена по-голяма от 2000. Резултатът нека бъде подреден по цена на продукт възходящо.	60
Пример 4-3. Да се изведе броя на всички служители.	61
Пример 4-4. Да се изведе броя служители, групирани по отдела, в който работят.	61
Пример 4-5. Да се изведат идентификаторите на държавите, в които има клиенти или отдели на фирмата.	63
Пример 4-6. Да се изведат идентификаторите на държавите, в които има клиенти или отдели на фирмата. Нека в резултатния набор участват и дублиращите се записи.	64
Пример 4-7. Да се изведат идентификаторите на държавите, в които има клиенти и отдели на фирмата едновременно.	66
Пример 4-8. Да се изведат идентификаторите на държавите, в които има клиенти и в същото време няма отдели на фирмата.	68
Пример 4-9. Да се изведат държавите и регионите, в които се намират.	71
Пример 4-10. Да се изведат държавите и регионите, в които се намират.	71
Пример 4-11. Да се изведат имената на клиентите, имената на държавите, от които са, и имената на регионите на държавите.	72
Пример 4-12. Да се изведат регионите и държавите, които се намират в тях. Резултатният набор да включва и регионите, в които няма въведени държави.	74
Пример 4-13. Да се изведат държавите и регионите, в които се намират. Резултатният набор да включва държавите, за които няма въведен регион.	75
Пример 4-14. Да се изведат държавите и регионите, в които се намират. Резултатният набор да включва държавите, за които няма въведен регион и регионите, за които няма въведени държави.	76
Пример 4-15. Да се изведат държавите и регионите, в които се намират.	77
Пример 4-16. Да се изведат отделите, в които има назначени служители.	78

Пример 4-17. Да се изведат имената на клиентите, които все още не са правили поръчки.	79
Пример 4-18. Да се изведат комбинациите между всички региони и държави, сортирани по име на държава.....	80
Пример 4-19. Да се изведат 7-те продукта с най-ниска цена.	83
Пример 4-20. Да се изведат първите 5 служители, след което вторите 5, сортирани по идентификатор (с цел лесно проследяване на порциите редове, които се извличат).	85
Пример 4-21. Да се изведат петимата служители, започвайки от 10-ти ред, подредени по дата на постъпване в компанията. Първата заявка ще покаже всички с цел демонстрация, втората ще извърши подбора.	85
Пример 5-1. Да се създаде изглед, който съдържа име и фамилия на клиентите, както и номер и дата на поръчките, които те са направили.	88
Пример 5-2. Да се модифицира горният изглед така, че да съдържа и колона с името на съответния служител, обработил поръчката.....	89
Пример 5-3. Да се модифицира горния изглед така, че да съдържа само поръчките, обработени от служител с идентификатор = 167.....	89
Пример 5-4. Да се създаде изглед, съдържащ име и фамилия на служител и общата сума на поръчките, които той е обработил.	89
Пример 5-5. Да се създаде изглед, който съдържа имена, отдел и заплата на 5-мата служители с най-висока заплата. За да бъдат извлечени служителите, подредени по заплата, очевидно ще трябва да бъдат сортирани по този критерий.	90
Пример 5-6. Да се добави нов запис в таблицата CUSTOMERS през изгледа CUSTOMERS_COUNTRIES.	91
Пример 5-7. Да се промени фамилията на клиент с идентификатор 10.	91
Пример 5-8. Да се изтрие клиент с идентификатор 10.....	92
Пример 6-1. Да се направи транзакция, която добавя нов клиент и създава поръчка за него, включваща два продукта.....	95
Пример 6-2. Да се направи транзакция, която променя фамилията на клиент с идентификатор = 1001, след което отхвърля направените промени.	95

Пример 6-3. Да се направи транзакция, която въвежда нов клиент, поставя точка на запис, въвежда поръчка, след което отхвърля промените до точката на запис, т.е. отхвърля се само поръчката.	96
Пример 7-1. Да се създаде съхранена процедура, която да извежда списък с поръчките на определен клиент, както и информация за самия клиент.	103
Пример 7-2. Да се създаде процедура, която за подадена като входен параметър поръчка (идентификатор) извежда имена на служител, който я е обработил, както и общата ѝ стойност.	104
Пример 7-3. Да се създаде процедура с изходни параметри, която ще връща като параметри броя на клиентите и броя на служителите.	104
Пример 7-4. Да се създаде функция, връщаща като скаларна стойност текст, съдържащ името на отдел (подаден като параметър) и обща стойност на заплатите в него.	106
Пример 7-5. Да се създаде функция, връщаща като резултат резултатен набор служителите с техните длъжности.	107
Пример 8-1. Да се създаде курсор, който демонстрира прочитане на данни ред по ред от курсор. Резултатният набор за целта ще съдържа всички клиенти от Германия. Стъпките от жизнения цикъл са обозначени в коментари.	110
Пример 8-2. Да се създаде курсор, който демонстрира обновяване и изтриване на ред от курсор. За целта резултатният набор ще съдържа всички записи от таблицата с професиите.	111
Пример 9-1. Да се създаде тригер, който ще се изпълнява след добавяне, промяна и изтриване на записи в таблицата PRODUCTS и ще извежда всички записи от inserted и deleted, за да се види какво реално те ще съдържат след всяка команда.	115
Пример 9-2. Да се създаде тригер, изпълняващ се при добавяне на запис или промяна на данни на служител (таблица EMPLOYEES).	117

1. Въведение

1.1. За кого е подходящо?

Това учебно помагало е разработено в съответствие с изискванията и учебната програма на следните специалности от Факултета по математика и информатика в Пловдивски университет „Паисий Хилендарски“:

- Софтуерно инженерство
- Софтуерни технологии и дизайн
- Информатика
- Бизнес информационни технологии

1.2. Цели на курса

Този курс представя работата с релационни бази от данни чрез стандарта за релационни езици SQL. Основна цел е обучаващият се да научи основните концепции на релационните бази от данни, използването на конкретната реализация на езика SQL за създаване на обекти, манипулиране и извличане на данни от тях. Важно е да се отбележи, че не всички аспекти от езика ще бъдат дискутирани в детайли, така че този документ не представлява изчерпателна документация на целия език (стандарт) с всичките му възможности и за всички реализации.

По време на курса ще бъдат използвани оператори от стандарта SQL, като в определени случаи ще бъдат споменавани някои различия между реализациите му във водещите релационни бази от данни като Oracle, Microsoft SQL Server и MySQL.

1.3. Защо бази от данни?

В днешния свят информацията е повсеместна, произтича от всичко заобикалящо ни, прониквайки в съзнанието на хората с и без тяхно съгласие. През времето на целия си живот ние генерираме и използваме информация. Това е най-конвертируемият продукт, генериращ доходи и работа (както и проблеми) за огромно и нарастващо количество специалисти по целия свят. Той се търгува и краде по всевъзможни начини, използва се за всякакви цели и практически е невъзможно влиянието му да бъде игнорирано от никого. А базите от данни се доказват като най-добрият подход за съхранение и управление на данните, обработката им и извличането на информация и знания.

В днешни дни няма бизнес, който да не съхранява данни, жизненоважни за оцеляването му. Данните и извлечената от тях информация са основните аргументи, на база на които се вземат решения, определящи посоката на развитие и правещи разликата между успех и провал.

Исторически еволюцията на системите за управление на бази от данни (СУБД) може да се раздели на три етапа: навигационни, релационни/SQL и пост-релационни бази от данни.

Навигационните включват йерархични и мрежови модели. Те са известни още от 60-те години.

Релационният модел е предложен през 1970 г. от д-р Едгар Код, наричан по-късно баща на релационния модел. Базиран на теория на множествата и релационната алгебра, той е сред малкото концепции в софтуерния свят, който е само допълван и основата му остава непроменена от тогава. През 80-те години релационните системи се развиват в широк мащаб, а през 90-те те стават доминиращи в глобален аспект – вероятно заради свойства като ефективност, простота и интуитивност, а тази доминация е актуална и днес.

В ерата на пост-релационните бази са разработени обектно-ориентираният модел (80-те години), както и хибридният обектно-релационен. През последните няколко години известност добиват т.нар. NoSQL и Big Data бази от данни. Тяхната основна концепция е да съхраняват големи обеми от неструктурирани данни – от Интернет пространството като история на плащания, клиентски предпочитания, тенденция на разглеждане и поведение, от социални мрежи като Facebook, Twitter, LinkedIn, данни от мобилни устройства, сензори (GPS, RFID, метеорологични и др.), генериращи огромни обеми от данни в различни формати – текст, изображения, звук и видео.

Все пак, този курс ще обърне внимание на релационните бази от данни като най-универсални и представляващи доминантна част от пазара.

1.4. Релационни бази от данни

Най-просто казано базите данни са компютърни системи, които служат за съхранение на данни. Релационните такива съхраняват данните в таблици с редове и колони. Колоните представляват атрибутите, а редовете съдържат данните. Накратко, таблиците имат следните характеристики:

- Таблицата се възприема като двумерна структура, съставена от редове и колони;
- Всеки ред съдържа една инстанция на обекта, за който таблицата съхранява данни;
- Всяка колона представя атрибут на обекта и има уникално име;

- Сечението на ред с колона трябва да съдържа атомарна (скаларна, единична) стойност;
- Всички стойности в една колона трябва да са от един и същ тип;
- Подредбата на редовете и колоните в таблицата се смята за несъществена (те са множества, където наредба няма);
- Всяка таблица трябва да има атрибут или комбинация от атрибути, идентифициращи уникално всеки ред.


Релационният модел има три аспекта: релационни обекти; правила за интегритет (цялостност) на данните; релационна алгебра – множество от оператори за опериране с данните. В този документ ще бъдат засегнати всички тези аспекти с конкретни примери и обяснения.


Темата за проектиране и дизайн на базата данни също ще бъде засегната, като ще бъдат представени аргументите за създаване на правилен дизайн, в който да не се проявяват аномалии. Ще бъдат представени основните нормални форми и проблемите, които те решават, подобрявайки дизайна.

Ще бъдат демонстрирани и някои предизвикателни аспекти от работата с SQL – управление на транзакции, работа със съхранени процедури/функции, тригери, курсори.

1.5. Означения

Използваните означения в този документ са илюстрирани, както следва:

 Дефиниция

 Бележка/важна информация.

Команда на T-SQL (Microsoft SQL Server) .

Команда на PL/SQL (Oracle) .

Команда на SQL/PSM (MySQL) .

2. Език SQL

SQL (Structured Query Language) представлява стандарт за релационни езици. Той включва множество от оператори за управление на обектите и манипулиране на данните, разделени формално в няколко подмножества. Стандартът се актуализира периодично, но всеки производител на релационни бази данни сам решава до каква степен да следва препоръките му и кои функционалности да реализира. Често производителите отиват далеч отвъд изискванията на стандарта в определени аспекти, водени от нуждите в чисто практическата приложимост на дадени оператори, и от друга страна не разработват други аспекти от стандарта, считайки ги за слабо приложими в практиката.

По време на курса ще бъдат използвани примери от реализациите на този стандарт в следните сървъри за бази от данни със съответните им наименования:

- Oracle – PL/SQL;
- Microsoft SQL Server – Transact-SQL (T-SQL);
- MySQL – SQL/PSM.

☞ С оглед на вариращите в различните реализации команди за пълния им синтаксис е добре да се използва документацията за съответната реализация.

Също така командите в стандарта са нечувствителни относно използването на малки/главни букви, така че може да бъде използвана предпочитана конвенция, стига кодът да бъде добре форматиран и четлив – с този аргумент тук ще бъдат използвани предимно команди с главни букви за по-добра четливост.

2.1. В тази тема...

Ще бъдат разгледани:

- Кой са основните оператори на езика;
- Какво е предназначението на всеки от тях;
- Как да се използват с цел ефективно управление на базата данни.

2.2. Език за дефиниране на данни

Първото подмножество от езика включва операторите за манипулиране на структурите, съхраняващи данните, както и други обекти, които ще бъдат разгледани по-късно. Известно

е със своето съкращение **DDL** (Data Definition Language) и включва следните оператори: **CREATE, ALTER, DROP**.

2.2.1. CREATE

Служи както за създаване на обекти, така и за самата база от данни. В различните сървъри за релационни бази от данни създаването на база от данни варира. Ето демонстрация на създаването на база от данни с име MYDB:

```
--MSSQL:  
CREATE DATABASE MYDB
```

```
--Oracle:  
CREATE USER MYDB IDENTIFIED BY MY_PASSWORD;  
GRANT CONNECT TO MYDB;  
GRANT RESOURCE TO MYDB;
```

👉 В Oracle идеологията за база данни и потребител е различна, от което идва и разликата в синтаксиса, използващ ключовата дума USER и задължителното задаване на парола за достъп, my_password в случая. Също така трябва да бъдат зададени и права, за да може потребителят да управлява обектите в схемата.

```
--MySQL - коя да е от двете команди  
CREATE DATABASE MYDB  
CREATE SCHEMA MYDB
```

След като е създадена базата от данни (или т.нар. схема/потребител в някои реализации) и направена текуща (в Oracle е такава след идентификация с потребител и парола) с командата `USE MYDB`, следва демонстрация на създаването на таблица в нея:

```
CREATE TABLE STUDENTS  
(  
    FAC_NUM INT NOT NULL,  
    NAME VARCHAR(50) NOT NULL,  
    PHONE VARCHAR(40)  
)
```

2.2.2. ALTER

Служи за модификация на вече създадени обекти. Използва се за добавяне, изтриване и промяна на колони в създадени таблици, добавяне и изтриване на ограничения в съществуващи таблици, и др. Следващият пример демонстрира добавяне на колона в създадената преди това таблица STUDENTS:

```
ALTER TABLE STUDENTS  
ADD ADDRESS VARCHAR(70)
```

За промяна на типа от данни на колона в таблица ще бъдат демонстрирани следните разлики в синтаксиса на различните реализации:

```
--MSSQL:  
ALTER TABLE STUDENTS  
ALTER COLUMN ADDRESS TEXT
```

```
--MySQL и Oracle (преди версия 10g)  
ALTER TABLE STUDENTS  
MODIFY COLUMN ADDRESS CHAR(200);
```

```
--Oracle (версия 10g и по-нови)  
ALTER TABLE STUDENTS  
MODIFY ADDRESS VARCHAR(500);
```

2.2.3. DROP

Служи за изтриване на създадени обекти, както и на вече създадена база от данни.

Този оператор може да се използва в комбинация с ALTER за изтриване на отделни елементи от даден обект – напр. колони или ограничения от дадена таблица. Следващата команда демонстрира изтриване на колоната phone от таблицата students:

```
ALTER TABLE STUDENTS DROP COLUMN PHONE;
```

Следният код изтрива вече създадена таблица:

```
DROP TABLE STUDENTS
```


Следният код демонстрира изтриване на база данни с име MYDB в различните реализации:

```
--MSSQL
DROP DATABASE MYDB

--MySQL
DROP DATABASE MYDB

--Oracle
DROP USER MYDB
```

👉 Командата за изтриване на база данни или схема/потребител (в Oracle) може да не бъде успешна ако тази база данни е текуща или потребителят е включен (Oracle). За целта трябва да бъде изпълнена от друга текуща база данни или потребителят да бъде изключен.

2.3. Език за манипулиране на данни

Следващото подмножество от езика включва операторите за манипулиране на данни. Известно е със своето съкращение **DML** (Data Manipulation Language) и включва следните оператори: **INSERT, UPDATE, DELETE, MERGE, SELECT**.

👉 За целите на демонстрацията, ако таблицата или базата от данни е била изтрита с горните команди, трябва пак да бъдат създадени, за да могат да бъдат изпълнявани следващите команди.

```
CREATE DATABASE MYDB
GO
USE MYDB

CREATE TABLE STUDENTS
(
    FAC_NUM INT NOT NULL,
    NAME VARCHAR(50) NOT NULL,
    PHONE VARCHAR(40)
)
```

2.3.1. INSERT

Служи за добавяне на нов ред в таблица. Възможно е конструкцията да бъде написана по два начина. Първият начин специфицира имената на колоните и стойностите, които ще бъдат въведени. Следващите команди демонстрират въвеждане на нов студент в таблицата students и по двата начина.

```
INSERT INTO STUDENTS (FAC_NUM, NAME, PHONE)
VALUES (190126, 'Иван Иванов', '0889574829')
```

Ако бъдат добавени стойности за всички колони на таблицата не е необходимо да бъдат специфицирани имената на колоните в заявката, но трябва да се знае какъв е редът на създаване на колоните в таблицата - редът на стойностите трябва да е същия като колоните на таблицата. Синтаксисът на конструкцията ще бъде следният:

```
INSERT INTO STUDENTS
VALUES (190127, 'Петър Петров', '0876123456')
```

2.3.2. UPDATE

Служи за промяна на съществуващи данни в таблица. Следващият пример демонстрира команда за промяна на телефонния номер на въведения по-горе студент:

```
UPDATE STUDENTS
SET PHONE = '0877454647'
WHERE FAC_NUM = 190126
```

👉 Клаузата WHERE определя кой запис трябва да бъде променен. Ако бъде пропусната, всички записи в таблицата ще бъдат променени.

2.3.3. DELETE

Служи за изтриване на редове от таблица. Следва пример за изтриване на ред от таблицата students:

```
DELETE FROM STUDENTS
WHERE FAC_NUM = 190126
```

👉 Клаузата WHERE определя кой запис трябва да бъде изтрит. Ако бъде пропусната, всички записи в таблицата ще бъдат изтрити.

2.3.4. MERGE

Използва се за синхронизиране на данните в целевата таблица чрез добавяне/промяна/изтриване на редове на база на разлики, намерени след съединение (join) с таблицата-източник на данните [1].

👉 Без да се омаловажава полезността на този оператор, демонстрацията му ще бъде пропусната поради неговата рядка употреба и специфики в отделните реализации. При необходимост от употребата му не е зле да се направи справка с актуалната за реализацията му документация.

2.3.5. SELECT

Служи за извличане на данни от базата от данни. Връщаните данни са в табличен вид и често се наричат резултатен набор.

Ако дотук командите са изпълнявани последователно таблицата може да няма редове, затова трябва да бъдат изпълнени няколко INSERT команди с произволни данни, за да има резултати при извличането на данни.

Следва демонстрация на заявка, с която се извеждат имената и телефонните номера на всички студенти:

```
SELECT NAME, PHONE  
FROM STUDENTS
```

Изброяването на всички колони от таблицата може да бъде заместено със символа *:

```
SELECT *  
FROM STUDENTS
```

Извличането на данни е най-често използваната команда, затова тя ще бъде разгледана подробно по-нататък.

2.4. Език за контрол на достъпа

Това подмножество от езика включва операторите за контролиране на достъпа до данните. Известно е със своето съкращение **DCL** (Data Control Language) и включва следните оператори: **GRANT**, **REVOKE (DENY)**.

2.4.1. GRANT

Служи за предоставяне на достъп или привилегии до обектите на базата от данни на потребителите. Следващата демонстрация е на конструкция, която разрешава на ролята Public да извлича данни от таблицата students:

```
GRANT SELECT ON STUDENTS TO PUBLIC
```

2.4.2. REVOKE

Служи за премахване на потребителските права или привилегии за достъп до обектите на базата от данни. Следващата демонстрация е на конструкция, която отменя разрешението за извличане на данни от таблицата students на ролята Public:

```
REVOKE SELECT ON STUDENTS TO PUBLIC
```

👉 В T-SQL има и оператор DENY, който служи за отнемане на права, не позволявайки това правило да бъде заобикаляно чрез участие в групи или роли.

2.5. Език за контрол на транзакциите

Това подмножество от езика включва операторите за управление на транзакциите. Известно е със своето съкращение **TCL** (Transaction Control Language) и включва следните оператори: **COMMIT**, **SAVEPOINT**, **ROLLBACK**, **SET TRANSACTION**. Тези оператори ще бъдат разгледани по-късно.

2.6. Типове данни

В тази част са разгледани някои от предоставяните от MS SQL Server [2] системни типове данни. Освен това има възможност за дефиниране на потребителски типове, базирани на системните типове данни.

👉 Различните версии на една и съща база от данни може да се различават в типовете данни, затова консултирането с документацията не вреди. Освен това различните доставчици на бази данни често реализират типове данни с имена, различни от тези по стандарт. Също така някои от типовете данни може да имат т.нар. синоними.

2.6.1. Целочислени

- **bigint** – интервал на допустимите стойности от -2^{63} (-9,223,372,036,854,775,808) до 2^{63-1} (9,223,372,036,854,775,807). Размерът на заеманата памет е 8 байта;
- **int** – интервал на допустимите стойности от -2^{31} (-2,147,483,648) до 2^{31-1} (2,147,483,647). Размерът на заеманата памет е 4 байта;
- **smallint** – интервал на допустимите стойности от -2^{15} (-32,768) до 2^{15-1} (32,767). Размерът на заеманата памет е 2 байта;
- **tinyint** – интервал на допустимите стойности от 0 до 255. Размерът на заеманата памет е 1 байт.

2.6.2. Логически

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

2.6.3. Числа с плаваща запетая с фиксирана точност

- **decimal/numeric** - 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.

☞ Типовете данни decimal и numeric са синоними и могат да бъдат използвани взаимозаменяемо.

2.6.4. Числа с приблизителна точност

- **float [(n)]** - интервал на допустимите стойности от $-1.79E+308$ до $-2.23E-308$, 0 и от $2.23E-308$ до $1.79E+308$ в зависимост от стойността на параметъра n. 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 $-1.18E-38$, 0 и от $1.18E-38$ до $3.40E+38$. Размерът памет е 4 байта, еквивалентно число на float(24).

☞ SQL Server третира стойностите на n както следва: ако $1 \leq n \leq 24$, n се приема за 24; ако $25 \leq n \leq 53$, n се приема за 53. Синоним за real е float(24).

2.6.5. 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.

2.6.6. Дата и час

- **date** - дата от 1 януари 0001 до 31 декември 9999, с точност до 1 ден. Размерът на заеманата памет е 3 байта. Формат YYYY-MM-DD;
- **time** - час от 00:00:00.0000000 до 23:59:59.9999999, с точност до 100 наносекунди. Размерът на заеманата памет е между 3 и 5 байта. Формат hh:mm:ss[.nnnnnnnn];
- **smalldatetime** - дата и час от 1 януари 1900 до 6 юни 2079, с точност до 1 минута. Размерът на заеманата памет е 4 байта. Формат YYYY-MM-DD hh:mm:ss;

- **datetime** - дата и час от 1 януари 1753 до 31 декември 9999 с точност до 3 милисекунди. Размерът на заеманата памет е 8 байта. Формат YYYY-MM-DD hh:mm:ss[.nnn];
- **datetime2** - дата и час от 1 януари 0001 до 31 декември 9999 с точност до 100 наносекунди. Размерът на заеманата памет между 6 и 8 байта. Формат YYYY-MM-DD hh:mm:ss[.nnnnnnnn];
- **datetimeoffset** – дата, час и часова зона от 1 януари 0001 до 31 декември 9999 с точност до 100 наносекунди. Размерът на заеманата памет между 8 и 10 байта. Формат YYYY-MM-DD hh:mm:ss[.nnnnnnnn] [+|-]hh:mm.

2.6.7. Низове

- **char** [(*n*)] - низ с фиксирана дължина. Максимална дължина 8000 символа;
- **varchar** [(*n* | **max**)] - низ с променлива дължина. Максимална дължина 8000 символа. Ако за параметър се използва **max** означава, че размерът за съхранение е максимум до 2 GB;
- **text** - низ с променлива дължина. Максимален размер до 2 GB.

2.6.8. Низове във формат Unicode

- **nchar** [(*n*)] - низ с фиксирана дължина. Максимална дължина 4000 символа;
- **nvarchar** [(*n* | **max**)] - низ с променлива дължина. Максимална дължина 4000 символа. Ако за параметър се използва **max** означава, че размерът за съхранение е максимум до 2 GB;
- **ntext** - низ с променлива дължина. Максимален размер до 1 GB.

2.6.9. Двоични данни

- **binary** [(*n*)] - двоични данни с фиксирана дължина. Размер до 8000 байта;
- **varbinary** [(*n* | **max**)] - двоични данни с променлива дължина. Размер до 8000 байта;
- **image** - двоични данни с променлива дължина. Размер до 2GB.

☞ Типовете данни **text**, **ntext** и **image** ще бъдат премахнати в следващите версии на MSSQL, затова се препоръчва избягване на тяхната употреба, заменяйки ги с някой от **nvarchar(max)**, **varchar(max)** или **varbinary(max)**.

2.7. Оператори, идентификатори, коментари

2.7.1. Аритметични

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

2.7.2. Оператори за сравнение

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

2.7.3. Логически оператори

- И – AND;
- Или – OR;
- Отрицание – NOT.

2.7.4. 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 трябва да 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;
- Идентификатор, започващ с цифра, означава временна таблица или процедура.

☞ Изброените правила важат в T-SQL [3], в други реализации правилата за идентификатори варират, затова консултацията с документацията е препоръчителна.

2.7.5. Коментари

Многоредовите коментари започват с /* и завършват с */. Текстът между /* и */ ще бъде игнориран.

Коментар на ред започва с --. Всички команди или текст между -- и края на реда ще бъдат игнорирани.

2.8. Стандартни функции

Ще бъдат описани накратко някои от по-често използваните вградени в MS SQL функции, в други реализации тези функции могат да варират.

2.8.1. Агрегатни функции

Агрегатните функции извършват изчисление на една или повече стойности и връщат като резултат единична стойност. Агрегатните функции често се използват в комбинация с GROUP BY и HAVING клаузите на оператора SELECT [4].

- SUM ([ALL | DISTINCT] expression) – връща сумата от всички стойности или само от DISTINCT стойностите в израза;
- AVG ([ALL | DISTINCT] expression) – връща средно аритметична стойност на всички стойности или само на различните (DISTINCT) стойности в израза (колоната);
- COUNT ([ALL | DISTINCT] expression) - връща броя ненулеви стойности. Когато се използва DISTINCT връща броя на уникалните стойности;
- COUNT(*) - връща броя редове, които отговарят на критериите на SELECT израза. В този 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.

📖 Агрегатните функции игнорират редове със стойности NULL, с изключение на COUNT(*). 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.

2.8.2. 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.

2.8.2.1. Функции за преобразуване на типове

Служат за явно преобразуване на един тип в друг.

- CAST (expression AS data_type [(length)]) ;
- CONVERT (data_type [(length)] , expression [, style])
- PARSE (string_value AS data_type [USING culture])
- TRY_CAST (expression AS data_type [(length)])
- TRY_CONVERT (data_type [(length)] , expression [, style])
- TRY_PARSE (string_value AS data_type [USING culture])

Където:

- ✓ expression – валиден израз, резултатът от който да бъде преобразуван;
- ✓ string_value - nvarchar(4000) стойност, представляваща форматираната стойност, която да бъде преобразувана;
- ✓ data_type – тип, към който се прави преобразуване;
- ✓ length – опционен параметър за символните типове данни (низове);
- ✓ style – стил, използван при преобразуване на дати в низове;
- ✓ culture – опционен низ, който идентифицира как стойността на string_value да бъде форматирана.

2.8.2.2. Функции за дата

- CURRENT_TIMESTAMP – връща системната дата на сървъра в тип DATETIME;
- GETUTCDATE() – връща текущото UTC време, изчислява се от операционната система, на която работи сървъра, в тип DATETIME;
- GETDATE() – връща системната дата на сървъра в тип DATETIME;
- SYSDATETIME() – връща системната дата на сървъра в тип DATETIME2;
- SYSUTCDATETIME() – връща текущото UTC време на сървъра в тип DATETIME2;
- SYSDATETIMEOFFSET() – връща системната дата и часовия пояс на сървъра в тип DATETIMEOFFSET;
- DATEADD (datepart, number, date) – връща дата, образувана чрез добавяне към датата date, number на брой части datepart;

- DATEDIFF (*datepart* , *startdate* , *enddate*) – връща число, разликата в части от вид *datepart*, между две дати;
- DATEDIFF_BIG (*datepart*, *startdate*, *enddate*) – връща число, разликата в части от вид *datepart*, между две дати, като върнатото число е от тип BIGINT;
- 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;
- DAY (*date*) – връща като число ден от определена дата;
- MONTH (*date*) – връща като число месец от определена дата;
- YEAR (*date*) – връща като число година от определена дата.
- И др.

Където стойностите на *datepart* са от следната таблица:

datepart	Съкращение
year	yy, yyyy
quarter	qq, q
month	mm
dayofyear	dy, y
day	dd, d
week	wk, ww
weekday	dw
hour	hh
minute	mi, n
second	ss, s

millisecond	ms
microsecond	mcs
nanosecond	ns
TZoffset	tz
ISO_WEEK	ISOWK, ISOWW

2.8.2.3. Математически функции

- ABS (numeric_expression) – връща абсолютна стойност, променя отрицателните стойности в положителни стойности;
- DEGREES (numeric_expression) – връща съответния ъгъл в градуси, по зададени радиани;
- RAND ([seed]) – генератор на случайни числа между 0 и 1 или цяло ако е указано цялото число seed;
- ROUND (numeric_expression, length [, function]) – връща числова стойност, закръглена до определена дължина или точност;
- SQRT (float_expression) – връща корен квадратен на зададеното число;
- PI() – връща стойността на числото Пи.
- SQUARE (float_expression) – връща квадрат на посочената стойност;
- POWER (float_expression , y) – връща стойността на посоченото число на посочената степен;
- И др.

2.8.2.4. Функции за работа с низове

- CONCAT (string_value1, string_value2 [, string_valueN]) – връща низ, който е резултат от конкатенацията или сливането на два или повече низа;
- LEN (string_expression) – връща броя символи в низа;
- LTRIM (character_expression) – премахва интервалите в началото на низа;
- RTRIM (character_expression) – премахва интервалите в края на низа;
- TRIM(character_expression) – премахва интервалите в началото и края на низа;
- REPLACE (string_expression, string_pattern , string_replacement) – заменя всички съвпадения на определен низ с друг низ;

- REVERSE (character_expression) – обръща низа;
- SUBSTRING (expression, start, length) – връща част от символен или бинарен израз;
- LOWER (character_expression) – връща низа, като преобразува главните букви в малки;
- UPPER (character_expression) – връща низа, като преобразува малките букви в главни;
- И др.

2.8.2.5. Системни функции

Ще бъдат разгледани някои от по-често използваните системни функции:

- ISNULL (expression, replacement) – ако параметърът expression бъде изчислен до NULL, той се заменя от израза replacement;
- ISNUMERIC (expression) – връща 1, ако изразът се изчислява до числов тип, и 0 - в противен случай;
- IIF (boolean_expression, true_value, false_value) – изчислява първия аргумент и връща втория аргумент, ако първият аргумент е true, в противен случай връща втория аргумент;
- HOST_NAME() – връща името на работната станция;
- HOST_ID() – връща идентификатора на работната станция. Идентификационният номер на работната станция е идентификационният номер на процеса (PID) на приложението на клиентския компютър, който се свързва към SQL Server.

2.8.2.6. Функции за работа с типове TEXT и IMAGE

- TEXTPTR (column) – връща стойността на текстовия указател, която съответства на колона с text, ntext или image във varbinary формат;
- TEXTVALID ('table.column', text_ptr) – проверява дали конкретен текстов указател е валиден.

2.9. Задачи

Задача 2-1. Да се създаде база от данни с име TESTDB.

Задача 2-2. Да се създаде таблица в тази база данни с име COUNTRIES и следните колони:

- country_code – с тип char(3);
- name – тип varchar(40);
- population – тип int.

Задача 2-3. Да се добави нов атрибут в таблицата COUNTRIES с име phone_code и тип цяло число до 3 цифри.

Задача 2-4. Да се добави ред в таблицата COUNTRIES със следните данни:

- country_code: BGR;
- name: България;
- population: 7500000;
- phone_code: 359.

Задача 2-5. Да се промени населението на България на 6 милиона.

Задача 2-6. Да се изтрият всички редове в таблицата COUNTRIES.

Задача 2-7. Да се изтрие таблицата COUNTRIES от базата данни.

Задача 2-8. Да се изтрие базата данни TESTDB.

3. Създаване на база данни за търговска фирма

3.1. В тази тема...


- Ще бъде направена кратка демонстрация на процеса на проектиране и нормализация;
- Ще бъде създаден прост Entity-Relationship модел на данните;
- Ще бъдат създадени таблиците в базата данни чрез съответните команди и ще бъдат използвани от тук нататък в примерите;
- Ще бъдат създадени първичните и външните ключове на всяка от таблиците;
- Ще бъдат попълнени таблиците с данни;
- Ще бъдат демонстрирани прости примери за промяна на данните.

3.2. Теоретични понятия


При проектиране на една база от данни се вземат предвид взаимоотношенията между атрибутите на обектите, които ще бъдат съхранявани в нея. Обикновено се започва от Entity-Relationship диаграма, демонстрираща тези взаимоотношения. Често това се нарича концептуален модел. От него се съставя физическият модел на базата данни, който е обвързан с типа и версията на конкретно използваната система за управление на бази от данни.

В процеса на проектиране се вземат предвид функционалните зависимости между атрибутите и така се цели правилното подреждане на атрибутите в таблици с цел достигане разумна степен на нормализация – обикновено трета нормална форма се счита за най-разумна от практическа гледна точка. Добрата нормализация ще гарантира липсата на т.нар. аномалии на промените.

Просто практическо правило гласи, че всяка таблица трябва да представя само един обект – ако представя повече означава, че съдържа излишество, което от своя страна ще даде шанс за проява на аномалиите на промените.

 Първичен ключ – комбинация от атрибути (колони) в таблицата, чиито стойности са уникални за всеки ред, като в същото време не може да се изключи някоя от тях без това да наруши уникалността. Използват се за идентификация на ред от таблицата.

Най-предпочитани в практиката са ключовете, съставени от една колона (прости), защото най-лесно се използват.

 **Външен ключ** - комбинация от атрибути (колони) в таблицата, чиито стойности са измежду стойностите на референциран първичен ключ. Те дават възможност ред от една таблица да бъде асоцииран с ред/редове от друга таблица.

Тук става ясно, че ако първичният ключ е от няколко колони, външният също ще трябва е такъв, защото ще трябва да съдържа стойност от първичния. Простите ключове не носят излишество на данните и затова биват предпочитани.

Един от най-предизвикателните аспекти на традиционния дизайн е нормализацията. Въпреки че е смятан за труден процес, той всъщност е относително прост след като са разбрани основните концепции за:

- Аномалиите на промените;
- Зависимостите между данните.

Процесът на нормализация представлява декомпозиция на големи, неефективно структурирани таблици, на по-малки, по-ефективно структурирани, като при този процес не трябва да има загуби на данни, т.е. той трябва да е обратим. Нормализацията поддържа твърдението, че добре дефинираната база данни не съдържа дублираща се информация и излишните данни са сведени до минимум. Това гарантира интегритета (съгласуваността) на данните и тяхната коректност при извличането им.

Декомпозирането на некоректно структурираните таблици не е произволен процес. Това е методичен и последователен процес на привеждане на определена таблица в целевата нормална форма. Основната причина за нормализиране на таблиците е осигуряването на смислената им и ефективна структура. Некоректно проектираните таблици обикновено проявяват лош интегритет на данните и са субект на аномалии на промените и проблеми, породени от зависимости между данните. Ако не бъдат решени тези проблеми, на по-късен етап може да бъде открито, че данните, които биват извличани, са неконсистентни, неточни и дори напълно неверни.

3.3. Проектиране на базата данни

Процесите на проектиране и нормализация впоследствие ще бъдат демонстрирани накратко, започвайки с кратко описание на данните, които трябва да се съхраняват. Те включват:

1. Отделите на фирмата и тяхното географско местоположение:

- име на отдел;
 - ръководител на отдела;
 - град, област, адрес, пощенски код;
 - държава и географски регион на отдела.
2. Служителите на фирмата и заеманите от тях длъжности:
- име и фамилия на служител;
 - email, телефон;
 - дата на постъпване на работа;
 - заплата;
 - заемана длъжност;
 - минимална и максимална заплата за съответната длъжност;
 - отдел, в който работи служителят;
 - прекият му ръководител.
3. Клиенти на фирмата:
- име, фамилия;
 - адрес, държава;
 - email;
 - пол.
4. Продуктите, с които фирмата търгува:
- име на продукт;
 - кратко описание;
 - цена.
5. Направените поръчки от клиентите:
- номер на поръчката;
 - дата на поръчката;
 - клиент;
 - служител, обработил поръчката;
 - адрес за доставка на поръчката;
 - съдържание на поръчката – продукти, цени и количество.

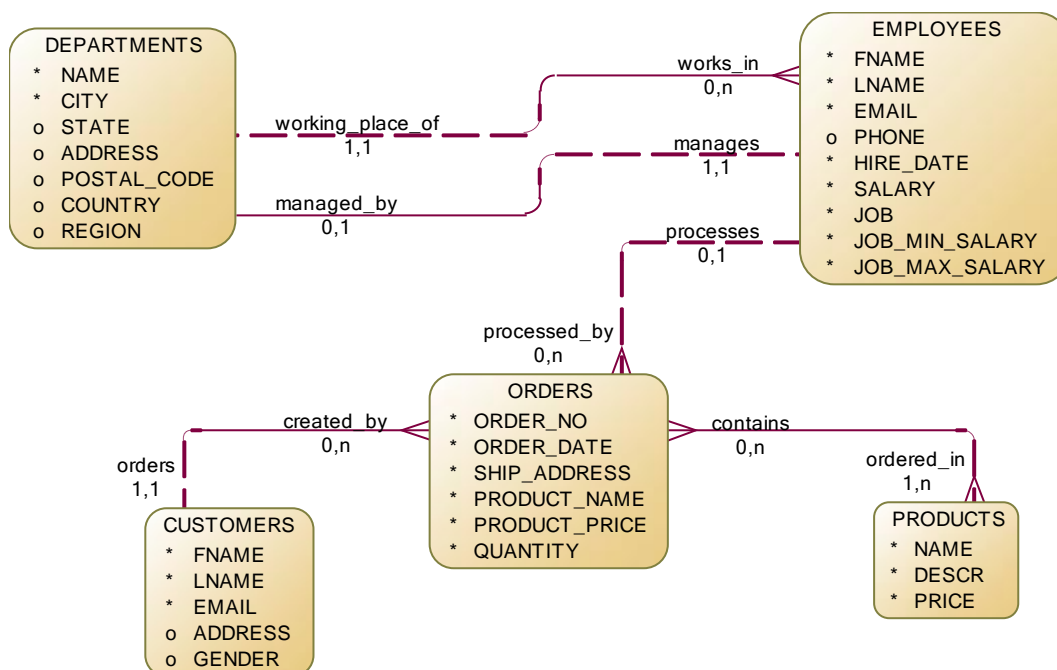
Някои зависимости между данните:

- ✓ Един клиент може да има нула или повече поръчки, но една поръчка е направена от точно един клиент;
- ✓ Една поръчка може да включва много продукти и един продукт може да бъде поръчван в много поръчки;

- ✓ В един отдел могат да работят нула и повече служители, но точно един служител е управител на отдела, т.е. не е задължително всеки служител да е управител на отдел;
- ✓ Един служител може да обработва нула или повече поръчки.

При първата итерация може да се направи концептуалния модел да представя данните така, както са изброени в изискванията, за да се започне от някъде все пак. Всяка следваща итерация при проектирането трябва да прецизира все по-точно модела, повишавайки степента на нормализация с цел избягване на аномалии на промените.

Имайки предвид това множество от данни може да бъде скицирана следната схема, която представя обектите, които ще съхраняват тези данни. Концептуалният ѝ вид е показан на следващата фигура:



Фигура 3-1. Концептуален (Entity/Relationship) модел на данните.

Връзките между отделните обекти (entities) представят взаимоотношенията между тях (relationships), ролите им и кардиналностите, с които участват в тези взаимоотношения. Тези обекти (а и някои взаимоотношения) по-късно ще се превърнат в таблици във физическия модел.

Кардиналност – числово изражение на това ред от една таблица с колко от друга таблица е асоцииран. В термините на обекти и взаимоотношения показва инстанция на един обект колко инстанции на другия обект притежава. Напр. един клиент може да има 0 или n на брой поръчки.

Видове:

1:1 – едно-към-едно: когато ред от едната таблица се асоциира с точно един ред от другата таблица и обратно;

1:N – едно-към-много: когато ред от едната таблица може да се асоциира с много редове от втората таблица, но ред от втората се асоциира само с един от първата;

M:N – много-към-много: когато ред от първата може да се асоциира с много от втората и ред от втората – с много от първата. Такова взаимоотношение се реализира с допълнителна таблица, разлагаща го на две от вид 1:N.

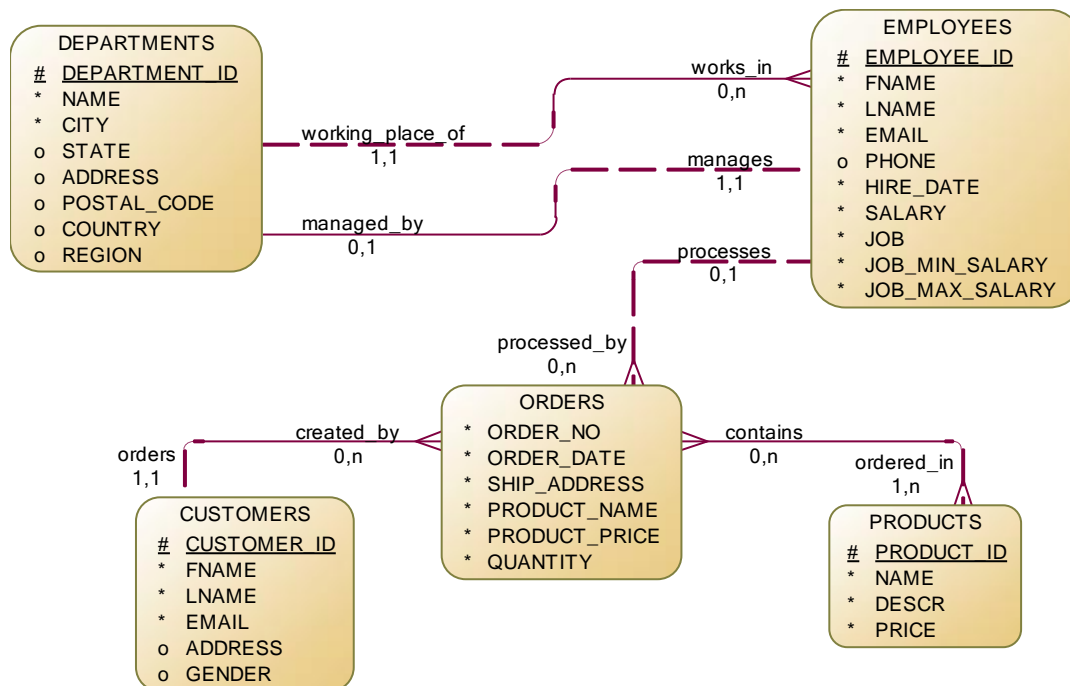
Разяснения:

- customer може да поръчва (orders) 0 или n на брой поръчки; order е създаден (created_by) от точно един customer. Това взаимоотношение е едно-към-много (1:N);
- product участва (ordered_in) в 0 или n orders; order съдържа (contains) 1 или n products. Това взаимоотношение е M:N;
- order е обработен (processed_by) от 0 или 1 employee; employee обработва (processes) 0 или n поръчки;
- department е работно място на (working_place_of) 0 или n служители; employee работи (works_in) в точно един department;
- employee е управител на (manages) 0 или 1 отдел; department се управлява от (managed_by) точно един employee.

Не е нужно да се прави опит за създаване на концептуалния модел перфектно още на първата итерация от процеса на проектиране. Неговата идея е да може да бъде коментиран с колеги, сътрудници и консултанти, вследствие на което пропуските да бъдат отстранени.

Желателно е на този етап да се определят атрибутите, които ще идентифицират всяка инстанция на обектите (т.е. идентификаторите на редовете от бъдещите таблици). Ако липсват атрибути, които да носят уникалност, се добавя такъв, който да изпълнява единствено тази роля. Такъв е случаят с EMPLOYEES. Освен това често в практиката дори при наличие на комбинация от атрибути, които да носят уникалност, се предпочита добавянето на един атрибут, който да бъде първичен ключ, с цел простота и удобство при използването (понякога бива наричан сурогатен ключ). Следващата итерация на модела ще бъде прецизирана чрез добавяне на атрибути за първични ключове в обектите.

Изключение ще прави ORDERS, където на по-късен етап ще има вече прецизиран ключ – order_no.



Фигура 3-2. Концептуален (Entity/Relationship) модел с идентификатори в обектите (без ORDERS).

3.4. Нормализиране на модела

Време е да бъде огледан моделът от друг аспект – доколко правилно са разположени данните в обектите, които ще се превърнат в таблици. Ако те съдържат данни за повече от един обект ще се проявяват аномалии на промените.

3.4.1. Аномалии на промените

Пръв ще бъде разгледан обектът EMPLOYEES – освен атрибутите на служителите там има и атрибути, свързани със заеманата длъжност (JOB...), т.е. този обект представя всъщност два такива – служител и заемана длъжност. В случай на необходимост от въвеждане на нова длъжност със съответните минимална и максимална заплата може да възникнат неудобства:

employee_id	fname	lname	...	job	job_min_salary	job_max_salary
?	?	?	?	Счетоводител	1000	2500

Какво трябва да се въведе в атрибутите за служител в новия ред? Т.е., за да бъде въведена новата длъжност ще е нужен поне един служител, който я заема. Но как да бъде нает счетоводител, след като още няма такава длъжност в компанията? Изглежда, че са нужни повече данни, отколкото изисква длъжността, която трябва да бъде въведена. Това е **аномалия на добавянето (insert anomaly)**.

Ако трябва да бъде изтрит служител, който е единствен с тази длъжност, може отново да се стигне до неудобства:

employee_id	fname	lname	...	job	job_min_salary	job_max_salary
108	Иван	Петров	...	Програмист	2000	4000


Изтриването на реда ще изтрие и данните за длъжността, този път изтривайки повече, отколкото е нужно (и правилно). Това е **аномалия на изтриването (delete anomaly)**.

При наличие на няколко служителя с една и съща длъжност, данните за тази длъжност ще се повтарят за всеки от тях:


employee_id	fname	lname	...	job	job_min_salary	job_max_salary
108	Иван	Петров	...	Програмист	2000	5000
109	Никола	Иванов	...	Програмист	2000	5000
110	Петър	Стоев	...	Програмист	2000	4000

Ако трябва да бъде увеличена максималната заплата на 5000 за тази длъжност, ще трябва да се направи за всички редове, където тя се среща. Ако бъде пропуснат дори един ред ще са налице неконсистентни данни, защото няма да е ясно еднозначно дали максималната заплата за тази длъжност е 4000 или 5000. Това е **аномалия на актуализацията (update anomaly)**.


Тези аномалии са породени от факта, че този обект всъщност представя два такива. Целта на нормализацията е да премахне появата на тези аномалии, като преобразува структурата на таблиците в по-ефективна такава и в крайна сметка всяка от тях да представя (съдържа данни за) само един обект.

 Първа нормална форма – всички атрибути трябва да съдържат скаларни (атомарни, единични) стойности.

В разглеждания модел може да се смята, че няма атрибути с комплексни стойности, така че таблиците в него са в първа нормална форма.

 Втора нормална форма – да е в първа нормална форма и всички атрибути в таблицата трябва да зависят от целия първичен ключ.

Тъй като ключове са съставени от само един атрибут, може да се приеме, че е достигната и втора нормална форма.

 Трета нормална форма – да е във втора нормална форма и всички атрибути в таблицата да бъдат функционално зависими само от първичния ключ. Целта е да се гарантира, че няма т.нар. транзитивни зависимости – А определя В, В определя С, следователно А определя С транзитивно. Ако такива са налице, значи таблицата представя повече от един обект.

В модела има такива примери и един от тях е EMPLOYEES: знаейки кой е служителът еднозначно може да бъде намерена коя е неговата длъжност; знаейки вече длъжността, еднозначно се намира минималната и максималната заплати, следователно employee_id транзитивно определя job_min_salary и job_max_salary. За да не се появяват аномалии на промените ще трябва да бъдат премахнати тези транзитивни зависимости.

Нормализационната процедура представлява декомпозиция на таблиците така, че те да представят по един обект.

Обектът ORDERS също представя не само поръчката, но и списъка на всички продукти, поръчани в нея, заедно с единичната им цена и поръчано количество. Това са по-скоро атрибути на взаимоотношението между PRODUCTS и ORDERS, така че ще отидат в таблицата, в която то ще се превърне заради това, че е от вид M:N.

DEPARTMENTS също представя повече от един обект, защото в себе си има данни за държави и региони, на които те принадлежат – също ще трябва да бъде декомпозиран.

След тези промени моделът има следния вид:

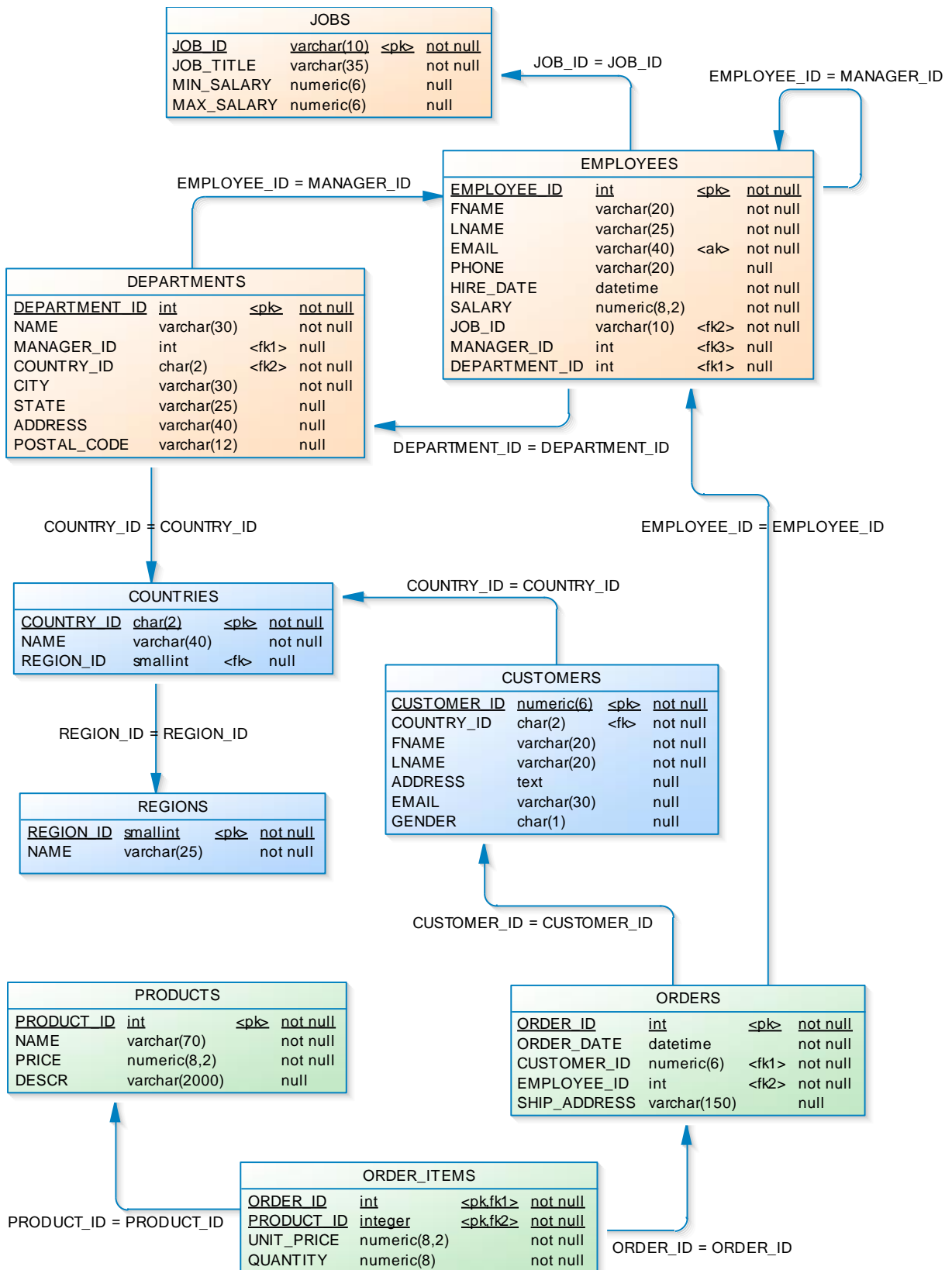
която може бъде направена с цел уеднаквяване на именуването на ключовете, е `order_no` да стане `order_id`.

Простият алгоритъм е:

- Всеки обект се превръща в таблица;
- Взаимоотношенията от вид 1:1 поставят атрибутите в една таблица;
- Взаимоотношенията 1:N стават външни ключове в таблиците от страната на кардиналността N;
- Взаимоотношенията от вид M:N се реализират с допълнителна таблица, декомпозираща ги на две от тип 1:N. В тази таблица идентификатор може да е комбинацията от идентификаторите на двата обекта от взаимоотношението (или поне трябва да е алтернативен ключ, т.е. да гарантира уникалност). Ако взаимоотношението има свои характеристики (атрибути), то те стават колони в тази таблица – такъв е случаят с това между `ORDERS` и `PRODUCTS`. То става допълнителната таблица, която разлага взаимоотношението от вид M:N на две 1:N (понякога бива наричана с жаргона „междинна таблица“).



Различното оцветяване на таблиците в модела се прави, за да се разделят тематично. Това улеснява ориентирането в модела, когато той съдържа много таблици.



Фигура 3-4. Физически модел на базата данни.

След като физическият модел вече е готов, може да се премине към създаването на таблиците в MS SQL Server.

3.5. Създаване на таблиците

При започване на стъпката за създаване на обектите в базата данни обикновено се подбират първо относително независимите обекти – това са тези таблици, които нямат външни ключове. Ако се започне от таблица, която има външен ключ към друга, то той ще трябва да бъде създаден чак след като тази друга таблица е вече създадена.

Разбира се, първо ще бъде създадена базата данни:

```
CREATE DATABASE TRADECOMPANY
```

3.5.1. REGIONS

Сега от схемата по-горе може да бъде избрана първата таблица. Удобен кандидат за първа изглежда тази с регионите:

REGIONS			
<u>REGION_ID</u>	<u>smallint</u>	<u><pk></u>	<u>not null</u>
NAME	varchar(25)		not null

Фигура 3-5. Схемата на първата създавана таблица – REGIONS.

Описание на атрибутите:

- REGION_ID – идентификатор на регион, използван за първичен ключ. Тип – цяло число;
- NAME – име на регион. Задължително е наличието на стойност в атрибута. Тип – низ до 25 символа.

Ограничения, наложени върху атрибутите:

- ✓ PK_REGIONS – първичен ключ, включващ атрибута REGION_ID;
- ✓ добре е стойностите в колоната NAME да са уникални, за да не се допусне въвеждането на два региона с едно и също име, затова ще бъде демонстриран най-краткият синтаксис за създаване на т.нар. уникален индекс, съдържащ само тази колона.

Описаната таблица с ограниченията ще бъде създадена със следния код:

```
-- Новата база данни трябва да е текуща  
USE TRADECOMPANY
```

```
CREATE TABLE REGIONS
(
    REGION_ID SMALLINT NOT NULL PRIMARY KEY IDENTITY(1, 1),
    NAME VARCHAR(25) NOT NULL UNIQUE
)
```

Това е възможно най-краткия код, с който се създава:

- първичен ключ, задавайки освен това свойство на полето REGION_ID да стартира стойностите си от 1 и да се увеличава автоматично с 1 за всеки нов ред (IDENTITY);
- уникален индекс, включващ полето NAME и гарантиращ, че в него няма да има дублиращи се стойности.

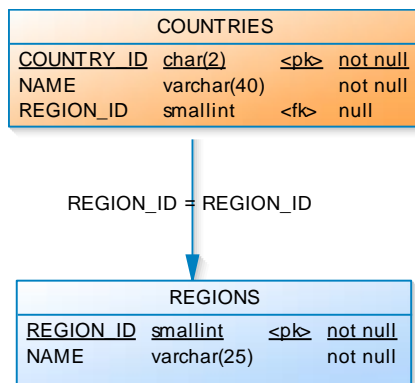
☞ При създаване на обектите синтаксисът може да има много различни варианти, които да направят едно и също. Често първо се създава таблицата и след това ограниченията, свързани с нейните колони. При създаване на следващите обекти (таблицы, ограничения) ще бъдат демонстрирани някои от тях.

3.5.2. COUNTRIES

Отново, водейки се по схемата, се избира следващата таблица – тази с държавите изглежда зависима (има външен ключ) само от REGIONS, която вече е създадена, затова е удобно да е следващата.

☞ Последната новосъздадена таблица ще бъде оцветена, за да се отличава в модела.

След създаването ѝ схемата ще изглежда така:



Фигура 3-6. Схемата след създаване на таблицата COUNTRIES.

Описание на атрибутите:

- COUNTRY_ID – идентификатор на държава (ISO код [5]), използван за първичен ключ.
Тип – низ от 2 символа;

- NAME – име на държава. Тип – низ до 40 символа;
- REGION_ID – идентификатор на регион, в който се намира държавата. Тип – цяло число. Тъй като този атрибут ще представлява външен ключ към регионите, типът му трябва да е като на съответния първичен ключ, който ще референцира. Този атрибут ще бъде оставен да приема нулева стойност, което означава, че ще е възможно дадена държава да не е асоциирана с регион.

Ограничения, наложени върху атрибутите:

- ✓ PK_COUNTRY – ограничение за първичен ключ на атрибута country_id;
- ✓ FK_COUNTRIES_REGIONS – ограничение за външен ключ върху атрибута REGION_ID към таблицата REGIONS и съответния ѝ атрибут REGION_ID.

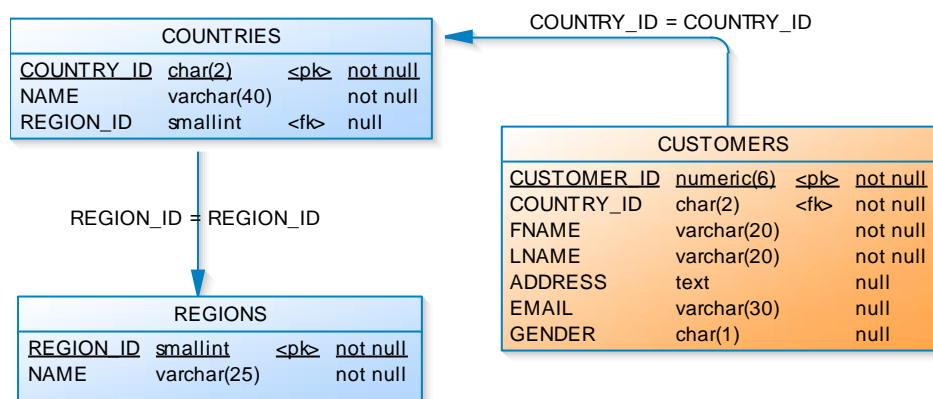
Ще бъде използван следният код:

```
CREATE TABLE COUNTRIES
(
  COUNTRY_ID CHAR(2)      NOT NULL,
  NAME       VARCHAR(40) NOT NULL,
  REGION_ID  SMALLINT     NULL,
  CONSTRAINT PK_COUNTRY PRIMARY KEY (COUNTRY_ID),
  CONSTRAINT FK_COUNTRIES_REGIONS FOREIGN KEY (REGION_ID)
    REFERENCES REGIONS (REGION_ID)
)
```

От кода се вижда как бяха създадени този път ограниченията след дефиницията на колоните, но отново в дефиницията на таблицата.

3.5.3. CUSTOMERS

Следващата удобна за създаване таблица е CUSTOMERS, която ще съхранява клиентите:



Фигура 3-7. Схемата след създаване на таблицата CUSTOMERS.

Описание на атрибутите:

- CUSTOMER_ID - идентификатор на клиент, използван за първичен ключ. Тип – цяло число с дължина до 6 цифри. Задължително е наличието на стойност в атрибута;
- COUNTRY_ID – идентификатор на държава. Тип – низ от 2 символа. Задължително е наличието на стойност в атрибута;
- FNAME – име на клиента. Тип – низ до 20 символа;
- LNAME – фамилия на клиента. Тип – низ до 20 символа;
- ADDRESS – адрес на клиента. Тип – текст;
- EMAIL – e-mail адрес. Тип – низ до 30 символа;
- GENDER – пол на клиента. Тип – 1 символ.

Ограничения, наложени върху атрибутите:

- ✓ Стойност по подразбиране за полето GENDER - ако не е указано друго, по подразбиране в полето се въвежда стойност 'M';
- ✓ Ограничение CUST_GENDER за проверка на въвежданите стойности в полето GENDER. Допустимите стойности са символи 'M' и 'F';
- ✓ Ограничение PK_CUSTOMERS за първичен ключ върху поле CUSTOMER_ID;
- ✓ Ограничение FK_CUSTOMER_COUNTRIES за външен ключ върху поле COUNTRY_ID към обекта COUNTRIES и съответния му атрибут COUNTRY_ID.

Ще бъде използван следният код:

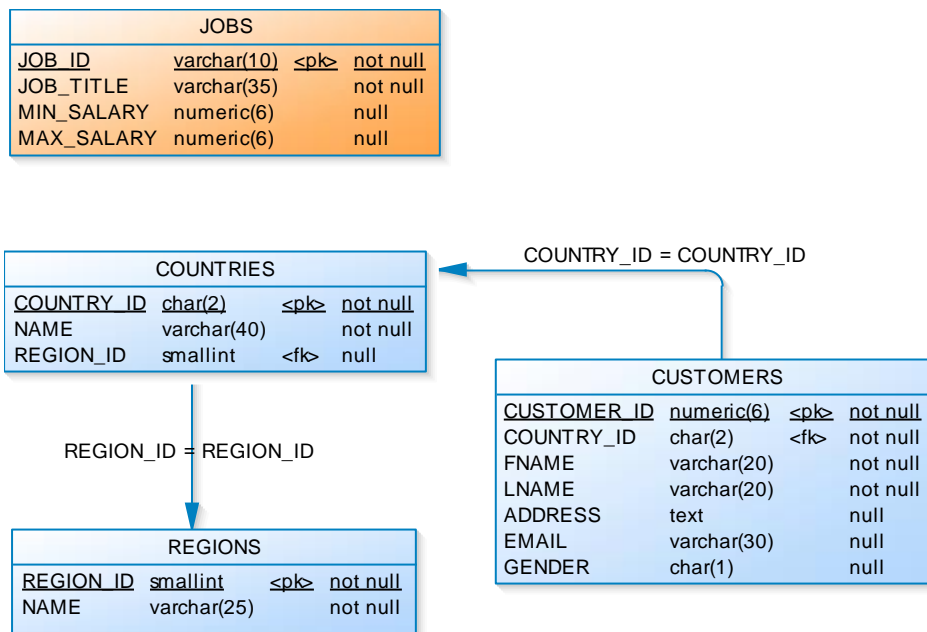
```
CREATE TABLE CUSTOMERS
(
    CUSTOMER_ID NUMERIC(6) NOT NULL,
    COUNTRY_ID CHAR(2) NOT NULL,
    FNAME VARCHAR(20) NOT NULL,
    LNAME VARCHAR(20) NOT NULL,
    ADDRESS TEXT NULL,
    EMAIL VARCHAR(30) NULL,
    GENDER CHAR(1) NULL DEFAULT 'M'
    CONSTRAINT CUST_GENDER CHECK (GENDER IS NULL OR (GENDER IN ('M', 'F'))),
    CONSTRAINT PK_CUSTOMERS PRIMARY KEY (CUSTOMER_ID)
)

ALTER TABLE CUSTOMERS
    ADD CONSTRAINT FK_CUSTOMER_COUNTRIES FOREIGN KEY (COUNTRY_ID)
        REFERENCES COUNTRIES (COUNTRY_ID)
```

👉 Тук външният ключ беше създаден след създаването на таблицата, за да се демонстрира още един възможен подход.

3.5.4. JOBS

Следвайки схемата, следващата таблица, с която е лесно да се продължи, е таблицата JOBS. Тя ще съхранява данни за длъжностите на фирмата. Схемата ще придобие следния вид:



Фигура 3-8. Схемата след създаване на таблицата JOBS.

Описание на атрибутите:

- JOB_ID - идентификатор на длъжност, използван за първичен ключ. Тип – низ с дължина до 10 символа. Задължително е наличието на стойност в атрибута;
- JOB_TITLE – наименование на длъжност. Тип – низ до 35 символа. Задължително е наличието на стойност в атрибута;
- MIN_SALARY – минимална годишна заплата за длъжност. Тип – цяло число до 6 цифри;
- MAX_SALARY – максимална годишна заплата за длъжност. Тип – цяло число до 6 цифри.

Ограничения, наложени върху атрибутите:

- ✓ Ограничение PK_JOBS за първичен ключ върху поле JOB_ID.

Ще бъде използван следният код:

```
CREATE TABLE JOBS
(
    JOB_ID      VARCHAR(10) NOT NULL PRIMARY KEY,
```

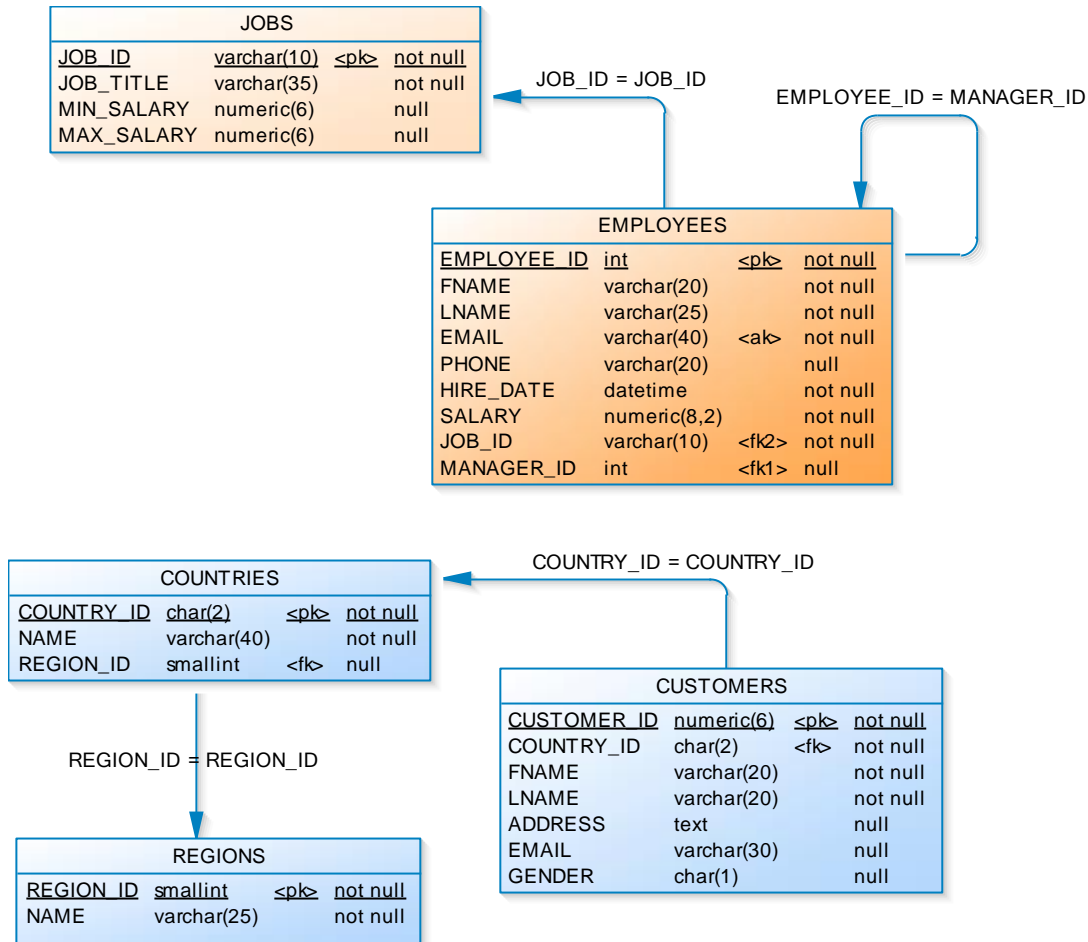
```

JOB_TITLE  VARCHAR(35) NOT NULL,
MIN_SALARY NUMERIC(6)   NULL,
MAX_SALARY NUMERIC(6)   NULL
)

```

3.5.5. EMPLOYEES

Следваща е таблицата, в която ще бъдат съхранявани данни за служителите на фирмата – EMPLOYEES. Схемата ще изглежда по следния начин след създаването ѝ:



Фигура 3-9. Схемата след създаване на таблицата EMPLOYEES.

Интересното и различно в тази таблица е, че тя ще съдържа дървовидна структура, описваща йерархията на служителите в компанията. Това се реализира чрез полето MANAGER_ID, което ще има ограничение за външен ключ, референциращ същата таблица, полето EMPLOYEE_ID. Стойността в MANAGER_ID за всеки служител ще съдържа идентификатора на прекия му ръководител.

Описание на атрибутите:

- EMPLOYEE_ID – идентификатор на служителя, използван за първичен ключ. Тип – цяло число. Задължително е наличието на стойност в атрибута;
- FNAME – малко име на служителя. Тип – низ до 20 символа. Задължително е наличието на стойност в атрибута;
- LNAME – фамилия на служителя. Тип – низ до 25 символа. Задължително е наличието на стойност в атрибута;
- EMAIL – електронна поща на служителя, използвана за алтернативен ключ (т.е. ще бъде наложена уникалност на това поле). Тип – низ до 40 символа. Задължително е наличието на стойност в атрибута;
- PHONE – телефон на служител. Тип – низ до 20 символа;
- HIRE_DATE – дата на назначаване на служителя. Тип – дата и час. Задължително е наличието на стойност в атрибута;
- SALARY – годишна заплата на служителя. Тип – число с плаваща запетая с дължина до 6 цифри за цялата част и 2 за дробната. Задължително е наличието на стойност в атрибута;
- JOB_ID – идентификатор на длъжността, която заема служителят. Тип – низ до 10 символа. Задължително е наличието на стойност в атрибута;
- MANAGER_ID – идентификатор на прекия началник на служителя. Тип – цяло число;
- DEPARTMENT_ID – идентификатор на отдела, в който работи служителят. Тип – цяло число.

Ограничения, наложени върху атрибутите:

- ✓ Ограничение PK_EMPLOYEES за първичен ключ върху полето EMPLOYEE_ID;
- ✓ Ограничение UK_EMAIL за уникалност върху полето EMAIL;
- ✓ Ограничение SALARY_CHECK за проверка на въвежданите стойности в поле SALARY. Допустимите стойности са числа по-големи от 0;
- ✓ Ограничение FK_EMPLOYEE_MANAGERS за външен ключ върху полето MANAGER_ID към същата таблица и съответния ѝ атрибут EMPLOYEE_ID;
- ✓ Ограничение FK_EMPLOYEE_JOBS за външен ключ върху полето JOB_ID към таблицата JOBS и съответния ѝ атрибут JOB_ID;
- ✓ Ограничение FK_EMPLOYEE_DEPARTMENTS за външен ключ върху полето DEPARTMENT_ID към таблицата DEPARTMENTS и съответния ѝ атрибут DEPARTMENT_ID.

Ще бъде използван следният код:

```
CREATE TABLE EMPLOYEES
(
    EMPLOYEE_ID INT NOT NULL,
    FNAME VARCHAR(20) NOT NULL,
    LNAME VARCHAR(25) NOT NULL,
    EMAIL VARCHAR(40) NOT NULL,
    PHONE VARCHAR(20) NULL,
    HIRE_DATE DATETIME NOT NULL,
    SALARY NUMERIC(8,2) NOT NULL
    CONSTRAINT SALARY_CHECK CHECK (SALARY > 0),
    JOB_ID VARCHAR(10) NOT NULL,
    MANAGER_ID INT NULL,
    DEPARTMENT_ID INT NULL,
    CONSTRAINT PK_EMPLOYEES PRIMARY KEY (EMPLOYEE_ID),
    CONSTRAINT UK_EMAIL UNIQUE (EMAIL),
    CONSTRAINT FK_EMPLOYEE_JOBS FOREIGN KEY (JOB_ID) REFERENCES JOBS (JOB_ID),
    CONSTRAINT FK_EMPLOYEE_MANAGERS FOREIGN KEY (MANAGER_ID)
        REFERENCES EMPLOYEES (EMPLOYEE_ID)
)
```

☞ Изискването за ограничение за външен ключ върху полето DEPARTMENT_ID към таблицата DEPARTMENTS и съответното поле DEPARTMENT_ID ще бъде реализирано по-късно след създаването на тази таблица, защото към момента тя все още не съществува.

3.5.6. DEPARTMENTS

Следващата таблица, която ще бъде създадена, е DEPARTMENTS. Тя ще съхранява данни за отделите на фирмата. След създаването ѝ схемата ще има вида, показан на Фигура 3-10.

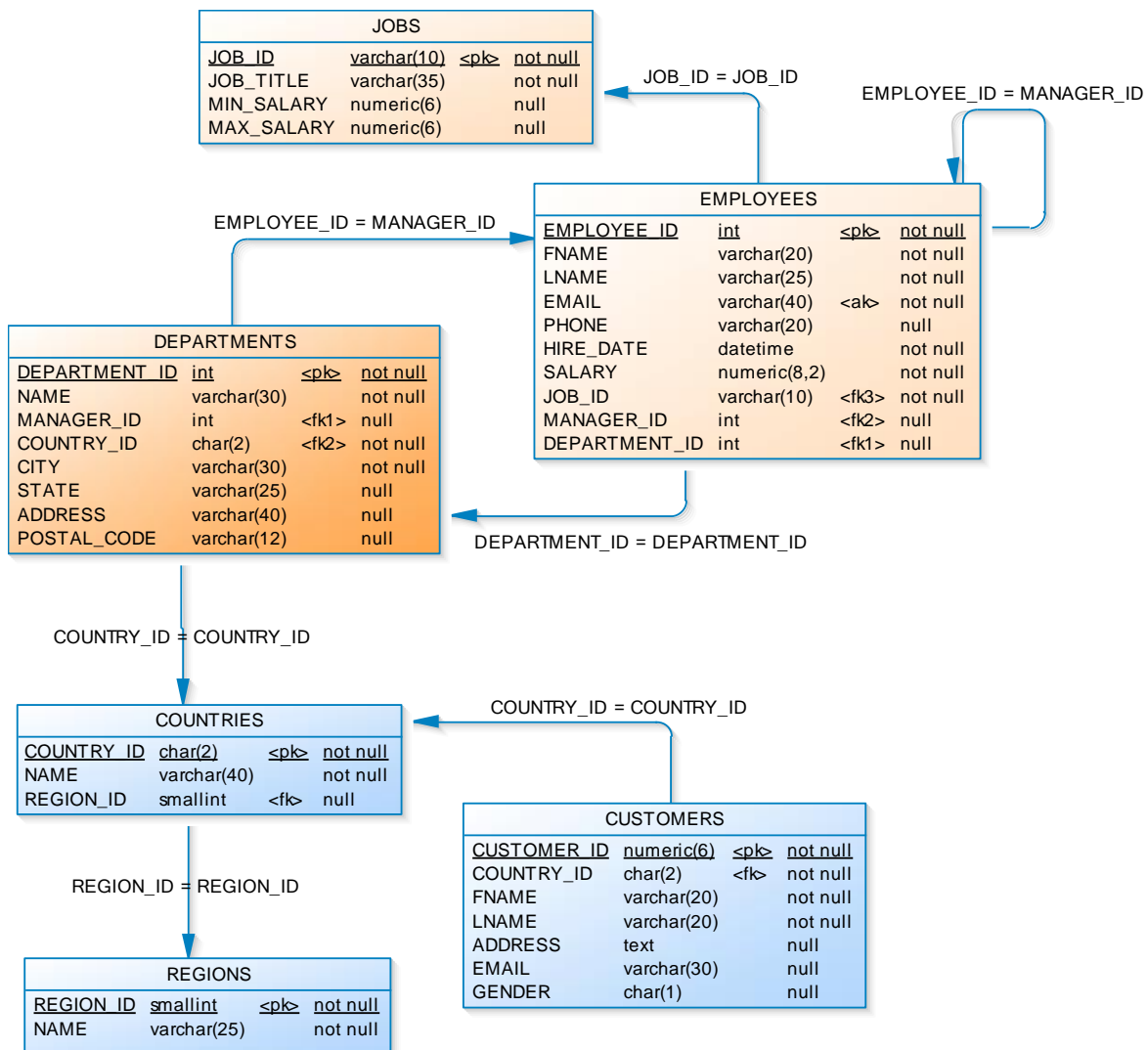
Описание на атрибутите:

- DEPARTMENT_ID – идентификатор на отдел, използван за първичен ключ. Тип – цяло число. Задължително е наличието на стойност в атрибута;
- NAME – име на отдел. Тип – низ до 30 символа. Задължително е наличието на стойност в атрибута;
- MANAGER_ID – идентификатор на ръководител на отдела. Тип – цяло число;
- COUNTRY_ID – идентификатор на държава. Тип – низ от 2 символа. Задължително е наличието на стойност в атрибута;
- CITY – град, в който се намира отдела. Тип – низ до 30 символа. Задължително е наличието на стойност в атрибута;

- STATE – област, в която се намира отдела. Тип – низ до 25 символа;
- ADDRESS – адрес на отдела. Тип – низ до 40 символа;
- POSTAL_CODE – пощенски код на града, в който се намира отдела. Тип – низ до 12 символа.

Ограничения, наложени върху атрибутите:

- ✓ Ограничение PK_DEPT за първичен ключ върху полето DEPARTMENT_ID;
- ✓ Ограничение FK_DEPT_COUNTRIES за външен ключ върху полето COUNTRY_ID към таблицата COUNTRIES и съответния ѝ атрибут COUNTRY_ID;
- ✓ Ограничение FK_DEPT_MGR за външен ключ върху полето MANAGER_ID към таблицата EMPLOYEES и съответния ѝ атрибут EMPLOYEE_ID.



Фигура 3-10. Схемата след създаване на таблицата DEPARTMENTS.

Ще бъде използван следният код:

```
CREATE TABLE DEPARTMENTS
(
    DEPARTMENT_ID INT NOT NULL,
    NAME VARCHAR(30) NOT NULL,
    MANAGER_ID INT NULL,
    COUNTRY_ID CHAR(2) NOT NULL,
    CITY VARCHAR(30) NOT NULL,
    STATE VARCHAR(25) NULL,
    ADDRESS VARCHAR(40) NULL,
    POSTAL_CODE VARCHAR(12) NULL,
    CONSTRAINT PK_DEPT PRIMARY KEY (DEPARTMENT_ID),
    CONSTRAINT FK_DEPT_MGR FOREIGN KEY (MANAGER_ID)
        REFERENCES EMPLOYEES (EMPLOYEE_ID),
    CONSTRAINT FK_DEPT_COUNTRIES FOREIGN KEY (COUNTRY_ID)
        REFERENCES COUNTRIES (COUNTRY_ID)
)

ALTER TABLE EMPLOYEES
ADD CONSTRAINT FK_EMPLOYEE_DEPARTMENTS FOREIGN KEY (DEPARTMENT_ID)
REFERENCES DEPARTMENTS
```

👉 След създаването на таблицата DEPARTMENTS беше добавен към EMPLOYEES външният ключ, асоцииращ служителя с отдела, в който работи.

3.5.7. ORDERS

Създаването на схемата ще продължи с таблицата ORDERS, която ще съхранява данни за направените поръчки. След създаването ѝ схемата ще изглежда по начин, показан на Фигура 3-11.

Описание на атрибутите:

- ORDER_ID - идентификатор на поръчка, използван за първичен ключ. Тип – цяло число. Задължително е наличието на стойност в атрибута;
- ORDER_DATE – дата и час, когато е направена поръчката. Тип – дата и час. Задължително е наличието на стойност в атрибута;
- CUSTOMER_ID – идентификатор на клиента, направил поръчката. Тип – цяло число с дължина до 6 цифри;
- EMPLOYEE_ID – идентификатор на служителя, обработил поръчката. Тип – цяло число. Задължително е наличието на стойност в атрибута;

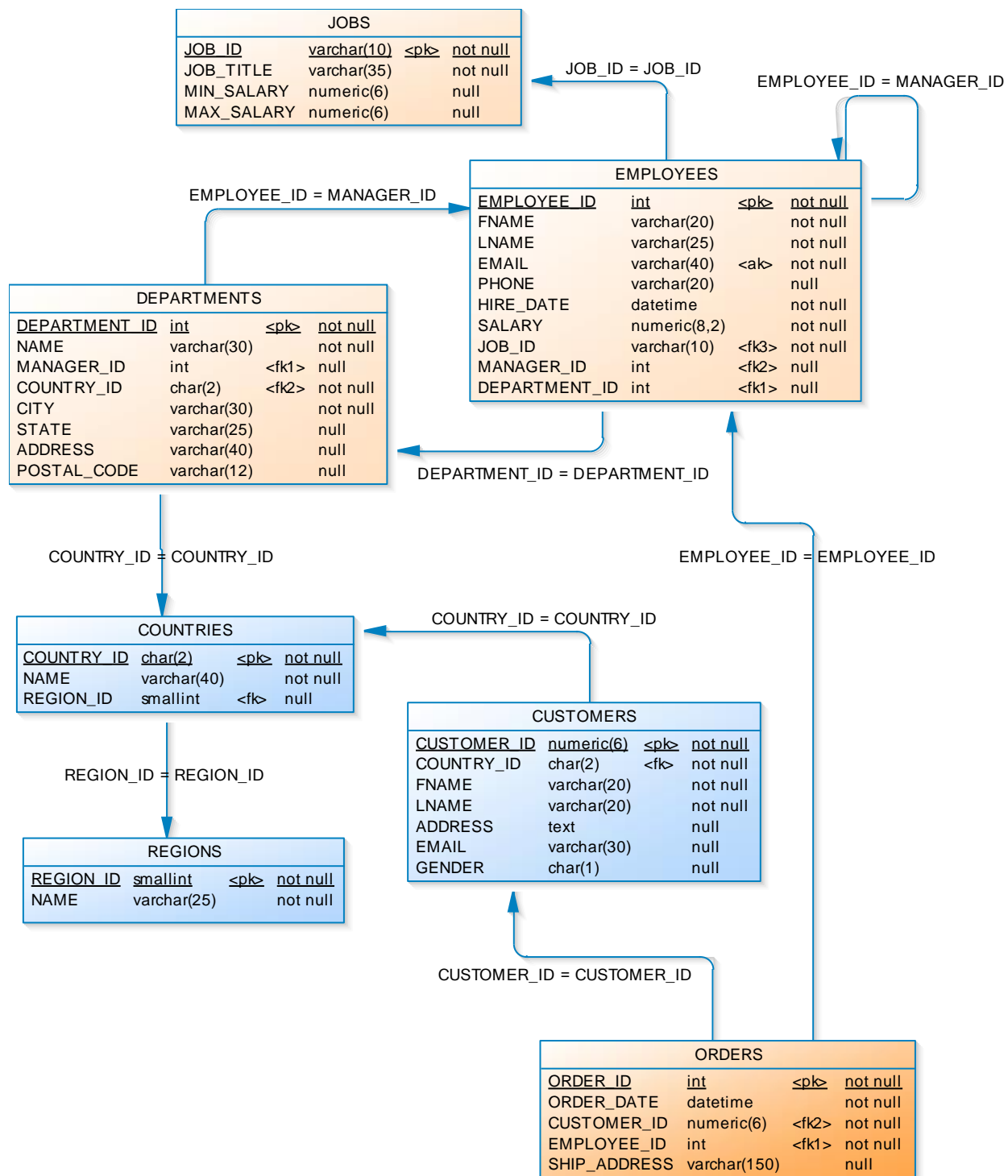
- SHIP_ADDRESS – адрес за доставка. Тип – низ до 150 символа.

Ограничения, наложени върху атрибутите:

- ✓ Ограничение PK_ORDERS за първичен ключ върху полето ORDER_ID;
- ✓ Ограничение външен ключ върху полето CUSTOMER_ID към таблицата CUSTOMERS и съответния й атрибут CUSTOMER_ID;
- ✓ Ограничение външен ключ върху полето EMPLOYEE_ID към таблицата EMPLOYEES и съответния й атрибут EMPLOYEE_ID.

Ще бъде използван следният код:

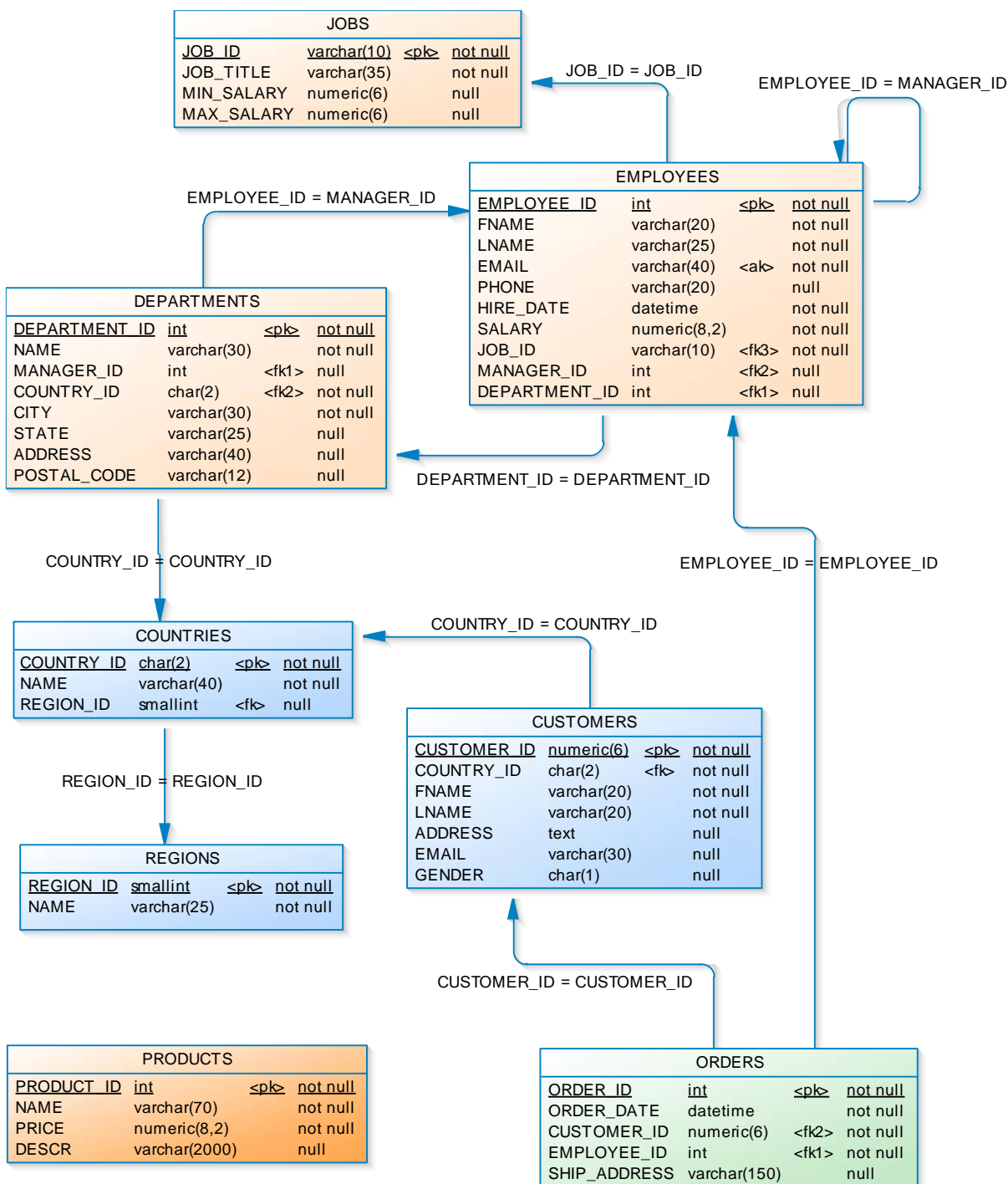
```
CREATE TABLE ORDERS
(
  ORDER_ID      INT           NOT NULL PRIMARY KEY,
  ORDER_DATE    DATETIME      NOT NULL,
  CUSTOMER_ID   NUMERIC(6)     NOT NULL FOREIGN KEY REFERENCES CUSTOMERS,
  EMPLOYEE_ID   INT           NOT NULL FOREIGN KEY REFERENCES EMPLOYEES,
  SHIP_ADDRESS  VARCHAR(150)  NULL
)
```



Фигура 3-11. Схемата след създаване на таблицата ORDERS.

3.5.8. PRODUCTS

Следвайки схемата идва ред за създаването на таблицата, в която ще бъдат съхранявани данни за продуктите, с които търгува фирмата. След създаването ѝ схемата ще изглежда по следния начин:



Фигура 3-12. Схемата след създаване на таблицата **PRODUCTS**.

Описание на атрибутите:

- PRODUCT_ID - идентификатор на продукт, използван за първичен ключ. Тип – цяло число. Задължително е наличието на стойност в атрибута;
- NAME – име на продукт. Тип – низ до 70 символа. Задължително е наличието на стойност в атрибута;
- PRICE – цена на продукт. Тип – число с плаваща запетая с дължина до 8 цифри. Задължително е наличието на стойност в атрибута;
- DESCR – кратко описание на продукт. Тип – низ до 2000 символа.

Ограничения, наложени върху атрибутите:

- ✓ Ограничение първичен ключ върху поле PRODUCT_ID.

Ще бъде използван следният код:

```
CREATE TABLE PRODUCTS
(
    PRODUCT_ID INT NOT NULL PRIMARY KEY,
    NAME VARCHAR(70) NOT NULL,
    PRICE NUMERIC(8,2) NOT NULL,
    DESCR VARCHAR(2000) NULL,
)
```

3.5.9. ORDER_ITEMS

Последната таблица, която остана да бъде създадена, е таблицата ORDER_ITEMS. Тя ще съхранява данни за съдържанието на поръчките. Крайната схема ще има вида, показан на Фигура 3-13.

Описание на атрибутите:

- ORDER_ID - идентификатор на поръчка, използван за част от първичния ключ. Този атрибут асоциира даден продукт с поръчката, в която участва, т.е. ще бъде външен ключ към поръчките (таблицата ORDERS). Тип – цяло число, задължително е наличието на стойност в атрибута;
- PRODUCT_ID – идентификатор на продукт, използван за част от първичния ключ. Този атрибут ще бъде външен ключ към продуктите (таблицата PRODUCTS). Тип – цяло число, задължително е наличието на стойност в атрибута;
- UNIT_PRICE – единична цена на продукт. Тип – число с плаваща запетая с дължина до 6 цифри за цялата част и до 2 за дробната. Задължително е наличието на стойност в атрибута;

- QUANTITY – брой поръчани от този продукт. Тип – цяло число с дължина до 8 цифри. Задължително е наличието на стойност в атрибута.

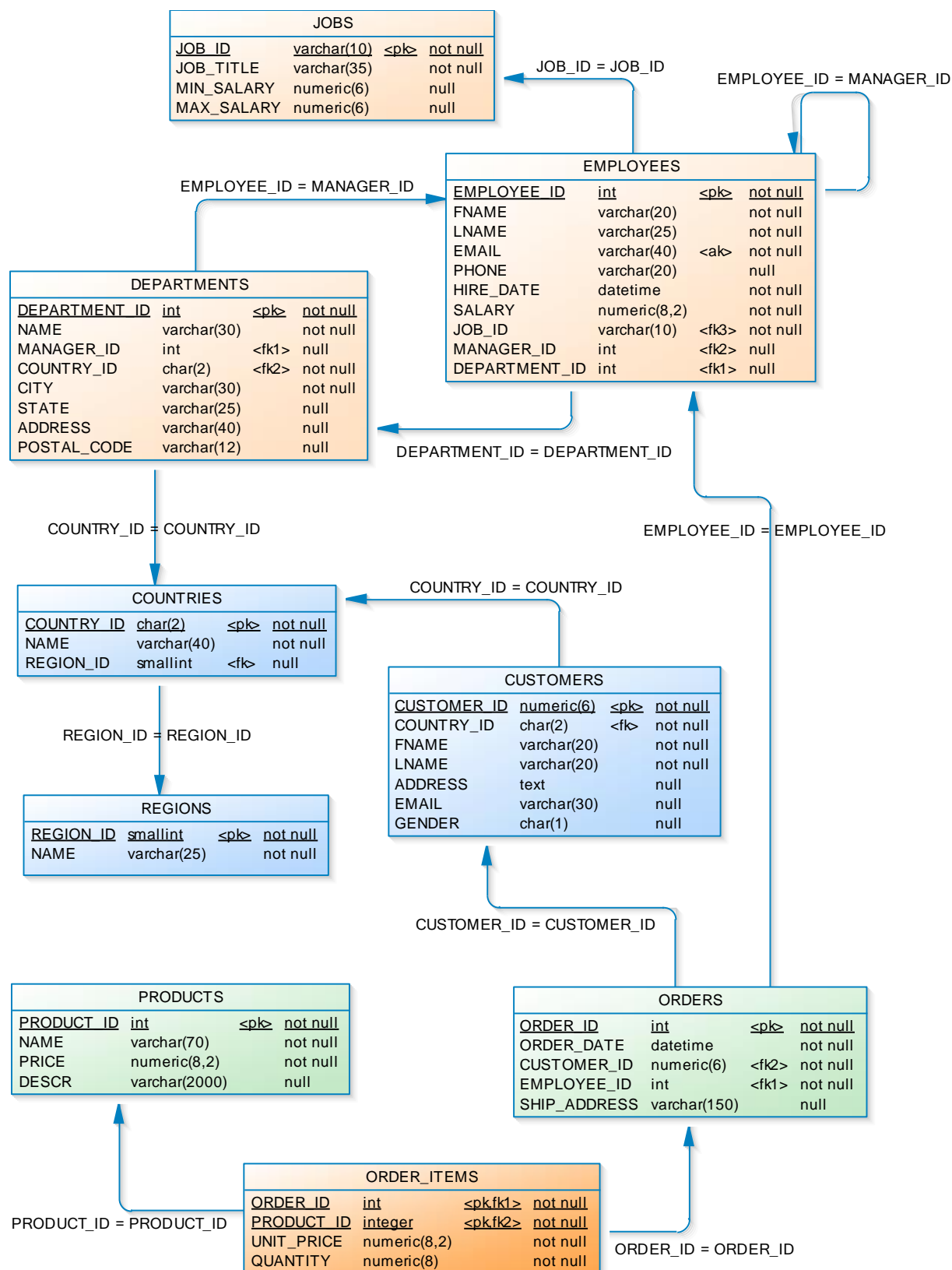
Ограничения, наложени върху атрибутите:

- ✓ Ограничение PK_ORDER_ITEMS за първичен ключ върху полетата ORDER_ID и PRODUCT_ID;
- ✓ Ограничение FK_ORD_ITEM_ORDERS за външен ключ върху полето ORDER_ID към таблицата ORDERS и съответния ѝ атрибут ORDER_ID;
- ✓ Ограничение FK_ORD_ITEM_PRODUCTS за външен ключ върху полето PRODUCT_ID към таблицата PRODUCTS и съответния ѝ атрибут PRODUCT_ID.

Ще бъде използван следният код:

```
CREATE TABLE ORDER_ITEMS
(
  ORDER_ID      INT           NOT NULL,
  PRODUCT_ID    INTEGER       NOT NULL,
  UNIT_PRICE    NUMERIC(8,2)  NOT NULL,
  QUANTITY      NUMERIC(8)    NOT NULL,
  CONSTRAINT PK_ORDER_ITEMS PRIMARY KEY (ORDER_ID, PRODUCT_ID),
  CONSTRAINT FK_ORD_ITEM_ORDERS FOREIGN KEY (ORDER_ID)
    REFERENCES ORDERS (ORDER_ID)
    ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT FK_ORD_ITEM_PRODUCTS FOREIGN KEY (PRODUCT_ID)
    REFERENCES PRODUCTS (PRODUCT_ID)
)
```

👉 Външният ключ към таблицата ORDERS има зададена опция за каскадно изтриване и актуализиране, което означава, че при изтриване на поръчка каскадно ще се изтриват всички редове, добавени към поръчката. При актуализация на идентификатор на поръчка в ORDERS, каскадно ще се актуализира идентификатора на същата поръчка в съответните редове в ORDER_ITEMS.



Фигура 3-13. Схемата след създаване на таблицата ORDER_ITEMS.

3.6. Манипулиране на данните

3.6.1. Добавяне

Добавянето на данни в таблиците започва с добавяне по един запис във всяка таблица. Трябва да се следи реда, в който данните се добавят в таблиците, където има външни ключове. Добавянето на данни обикновено започва от таблиците, които нямат външни ключове, и продължава с тези, които референцират данни от първите. Ще бъде използван редът на създаване на таблиците от схемата за въвеждане и на данните:

Добавяне на регион:

```
INSERT INTO REGIONS (NAME)
VALUES ('Източна Европа');
```

Добавяне на държава към този регион:

```
INSERT INTO COUNTRIES (COUNTRY_ID, NAME, REGION_ID)
VALUES ('BG', 'България', 1);
```

Добавяне на клиент от тази държава:

```
INSERT CUSTOMERS (CUSTOMER_ID, COUNTRY_ID, FNAME, LNAME, ADDRESS, EMAIL, GENDER)
VALUES (1001, 'BG', 'Георги', 'Генев', 'бул. България 100, Пловдив',
'ggenev@gmail.com', 'М');
```

Добавяне на длъжност:

```
INSERT INTO JOBS (JOB_ID, JOB_TITLE, MIN_SALARY, MAX_SALARY)
VALUES ('SA_REP', 'Търговски представител', 9000, 17000);
```

Добавяне на отдел:

```
INSERT INTO DEPARTMENTS (DEPARTMENT_ID, NAME, COUNTRY_ID, CITY, STATE,
ADDRESS, POSTAL_CODE)
VALUES (80, 'Продажби', 'BG', 'Пловдив', 'Пловдив', 'бул. Марица 10', '4000')
```

Добавяне на служител в този отдел:

```
INSERT INTO EMPLOYEES (EMPLOYEE_ID, FNAME, LNAME, EMAIL, PHONE, HIRE_DATE,
JOB_ID, SALARY, DEPARTMENT_ID)
VALUES (1501, 'Петър', 'Тодоров', 'petert@trade_company.com', '0899332013',
CONVERT(DATE, '30-01-2017', 105), 'SA_REP', 19000, 80);
```

Добавяне на продукт:

```
INSERT INTO PRODUCTS (PRODUCT_ID, NAME, DESCR, PRICE)
VALUES (20001, 'Смартфон Samsung Galaxy S10e', 'Смартфон Samsung Galaxy S10e,
Dual SIM, 128GB, 6GB RAM, 4G, Black.', 1400);
```

Добавяне на поръчка, направена от въведения клиент и обработена от въведения служител:

```
INSERT ORDERS (ORDER_ID, CUSTOMER_ID, EMPLOYEE_ID, SHIP_ADDRESS, ORDER_DATE)
VALUES (1, 1001, 1501, 'бул. България 140, Пловдив', CONVERT(DATETIME, '13-07-2010 10:30', 105));
```

Добавяне на продукт към поръчката:

```
INSERT INTO ORDER_ITEMS (ORDER_ID, PRODUCT_ID, UNIT_PRICE, QUANTITY)
VALUES (1, 20001, 1400, 2);
```

3.6.2. Промяна

Следният код е за промяна на заплатата на добавения служител с идентификатор 1501:

```
UPDATE EMPLOYEES
SET SALARY = 20200
WHERE EMPLOYEE_ID = 1501
```

3.6.3. Изтриване

С цел демонстрация ако се приеме, че поръчка с идентификатор 1 е анулирана, в този случай трябва да се изтрие съответният ред от таблицата ORDERS, който обаче е референциран от ред(ове) в таблицата ORDER_ITEMS. Референциалното ограничение между двете таблици има зададена опция за каскадно изтриване и актуализиране, което означава, че редовете в ORDER_ITEMS ще се изтрият автоматично.

```
DELETE FROM ORDERS
WHERE ORDER_ID = 1
```

3.7. Задачи

Задача 3-1. Да се изтрият всички редове от всички таблици в базата от данни, след което да се въведат данните чрез командите от [inserts_mssql.sql](#) файла.

Задача 3-2. Да се увеличи количеството с 2 броя и да се намали единичната цена с 5% на продукт с идентификатор 2254 в поръчка с идентификатор 2354.

Задача 3-3. Да се изтрие служител с идентификатор 183.

4. Извличане на данни

4.1. В тази тема...

Ще бъде демонстрирано:

- Какви релационни оператори има на разположение от стандарта SQL;
- Как могат да бъдат използвани, за да се комбинират данните и да бъдат извлечени в удобен вид.

Преди да се започне с тези команди е възможно структурата на базата от данни или данните в нея да не са съвсем удобни, защото са тествани доста операции върху тях или са прескочени някои стъпки от описаните до момента, затова може да се започне „на чисто“ със създаването на нова база от данни и добавяне на тестови данни от файловете, описани в Приложение 2 – ресурси.

4.2. SELECT



Операторът SELECT се използва за извличане на данни от базата данни. Върнатите данни са в табличен вид, който често се нарича резултатен набор.

Опростен синтаксис на оператора SELECT:

```
SELECT column_name_1, ..., column_name_n  
[ INTO new_table ]  
FROM table_source  
[ WHERE search_condition ]  
[ GROUP BY group_by_expression ]  
[ HAVING search_condition ]  
[ ORDER BY order_expression [ ASC | DESC ] ]
```

Описание на клаузите:

- SELECT – след тази ключова дума се указват колоните, които ще присъстват в резултатния набор, т.е. от които ще бъдат извлечени данни;
- INTO – използва се, когато целта е от върнатия резултатен набор да бъде създадена нова таблица с неговата структура и данни;
- FROM – указва откъде ще се извличат данните - таблици, изгледи (функции, синоними в някои езици като PL/SQL);

- WHERE – указва условията за търсене, на които трябва да отговарят редовете, за да участват в резултатния набор;
- GROUP BY – служи за формиране на групи от записи по зададени критерии, върху всеки от които да бъде изпълнена агрегираща функция, например;
- HAVING – специфицира условие за групите от записи или резултат, получен от агрегатна функция, т.е. тук може да се прави сравнение с резултати от агрегиращи функции. Тази клауза обикновено се използва с GROUP BY, но може и без нея в някои специфични случаи;
- ORDER BY – указва начина на сортиране на редовете в резултатния набор.

Пример 4-1. Да се изведат имената, датите на назначаване и заплатите на всички служители.

```
SELECT FNAME, LNAME, HIRE_DATE, SALARY
FROM EMPLOYEES
```

Получава се резултатът, показан на Фигура 4-1.

👉 Тук отново символът * може да замени изброяването на всички колони от таблиците, от които се извличат данни.

	FNAME	LNAME	HIRE_DATE	SALARY
1	Иван	Иванов	2011-06-17	27000.00
2	Петя	Василева	2015-09-21	18000.00
3	Димитър	Господинов	2012-01-13	19000.00
4	Александър	Петров	2010-01-03	12000.00
5	Боряна	Иванова	2010-05-21	12000.00
6	Димо	Тодоров	2005-06-25	7800.00
7	Васил	Петров	2008-02-05	8800.00
8	Диана	Лазарова	2009-02-07	7200.00
9	Нина	Георгиева	2015-08-17	12000.00

Фигура 4-1. Резултати от SELECT – данни на служителите.

Пример 4-2. Да се изведат всички данни за продуктите, с цена по-голяма от 2000. Резултатът нека бъде подреден по цена на продукт възходящо.

```
SELECT *
FROM PRODUCTS
WHERE PRICE > 2000
ORDER BY PRICE
```

Резултатният набор изглежда по начина, показан на Фигура 4-2.

☞ Клаузата ORDER BY подрежда резултатния набор по азбучен ред по посоченото поле или полета. Сортирането може да бъде във възходящ ред и низходящ ред, съответно чрез добавяне на ключовите думи ASC и DESC след името на полето. Ако ключова дума липсва – приема се по подразбиране ASC.

	PRODUCT_ID	NAME	PRICE	DESCR
1	2255	Лаптоп Gaming ASUS TUF FX504GM-EN048	2075.00	Лаптоп Gaming ASUS TUF FX504GM-EN048 с процесор ...
2	2459	Лаптоп ASUS S406UA-BM034T	2399.00	Лаптоп ASUS S406UA-BM034T с процесор Intel® Core™ i...
3	2414	UltraBook ASUS UX480FD-BE012R	2454.00	UltraBook ASUS UX480FD-BE012R with processor Intel® ...
4	1772	Лаптоп Gaming ASUS GL703GE-EE023	2456.00	Лаптоп Gaming ASUS GL703GE-EE023 with processor Int...
5	3354	Лаптоп Lenovo IdeaPad 530S-14IKB	2543.00	Лаптоп Lenovo IdeaPad 530S-14IKB с процесор Intel® Co...
6	3334	Лаптоп 2 in 1 Lenovo YOGA 730-15IWL	2612.00	Лаптоп 2 in 1 Lenovo YOGA 730-15IWL with processor Inte...
7	3399	Лаптоп Gaming ASUS TUF FX705GD-EW141	2815.00	1Лаптоп Gaming ASUS TUF FX705GD-EW141 с процесо...
8	2411	Смартфон Samsung Galaxy S10+	3200.00	Смартфон Samsung Galaxy S10+, Dual SIM, 1TB, 12GB R...
9	3071	Лаптоп Gaming ASUS ROG Strix Scar II GL504GM-ES040	3233.00	Лаптоп Gaming ASUS ROG Strix Scar II GL504GM-ES040 ...
10	2254	Ultrabook ASUS Zenbook UX430UN-GV075T	3453.00	1Ultrabook ASUS Zenbook UX430UN-GV075T с процесор...
11	3073	Лаптоп Gaming MSI GE73 Raider RGB 8RF	4224.00	Лаптоп Gaming MSI GE73 Raider RGB 8RF with processo...
12	2253	Лаптоп 2 in 1 Lenovo YOGA C930-13IKB	4399.00	Лаптоп 2 in 1 Lenovo YOGA C930-13IKB with processor Int...
13	1743	Лаптоп 2 in 1 Lenovo YOGA Glass C930-13IKB	4800.00	Лаптоп 2 in 1 Lenovo YOGA Glass C930-13IKB with proce...
14	1792	Лаптоп Lenovo ThinkPad X1 Carbon 6th	5225.00	Лаптоп Lenovo ThinkPad X1 Carbon 6th with processor Int...

Фигура 4-2. Резултати от SELECT – продукти с цена над 2000.

Пример 4-3. Да се изведе броя на всички служители.

```
SELECT COUNT (EMPLOYEE_ID)
FROM EMPLOYEES
```

Получава се следният резултат:

	(No column name)
1	101

Фигура 4-3. Резултати от SELECT – брой на всички служители.

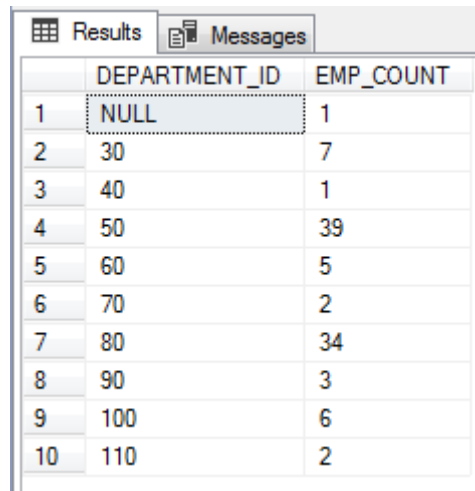
Пример 4-4. Да се изведе броя служители, групирани по отдела, в който работят.

```
SELECT DEPARTMENT_ID, COUNT (EMPLOYEE_ID) AS EMP_COUNT
FROM EMPLOYEES
GROUP BY DEPARTMENT_ID
```

Резултатният набор изглежда по начина, показан на Фигура 4-4.

☞ Изразът “COUNT(EMPLOYEE_ID) AS EMP_COUNT” означава, че последният атрибут в резултатния набор ще има име EMP_COUNT. По този начин могат да се преименуват (зададат псевдоними) както колони, така и таблици, участващи в заявките. При тази

заявка GROUP BY формира групи от редове за всяка уникална стойност в DEPARTMENT_ID, а функцията COUNT преброява ненулевите стойности колоната EMPLOYEE_ID за всяка група.



	DEPARTMENT_ID	EMP_COUNT
1	NULL	1
2	30	7
3	40	1
4	50	39
5	60	5
6	70	2
7	80	34
8	90	3
9	100	6
10	110	2

Фигура 4-4. Резултати от SELECT – брой служители по отдели.


4.2.1. Задачи

Задача 4-1. Да се изведат имената, заплатите и идентификаторите на длъжностите на служителите, работещи в отдели 50 и 80. Резултатът да е подреден по фамилия на служител във възходящ ред.

Задача 4-2. Да се изведат общата сума на заплатите и броя служители в отдел 60.

Задача 4-3. За всички поръчки да се изведат идентификатор на поръчка и обща стойност на поръчката. Резултатът да е подреден по стойност на поръчката в низходящ ред.

4.3. UNION

 Операторът UNION комбинира резултатните набори, получени от изпълнението на две или повече заявки в общ резултатен набор, съдържащ всички редове, върнати от заявките, като дублиращите се редове (ако има такива) се елиминират [6, 7].

Резултатните набори, които се обединяват, трябва да отговарят на следните условия:

- Всяка заявка в оператора UNION трябва да има еднакъв брой колони;
- Колоните трябва да имат съвместими типове от данни;

- В целия израз може да присъства само една клауза ORDER BY накрая, сортираща крайния резултат от обединението.



Фигура 4-5. Резултатно множество от записи при оператор UNION.

Синтаксис на оператора UNION:

```
SELECT column_name_1, ..., column_name_n
FROM table1
UNION [ALL]
SELECT column_name_1, ..., column_name_n
FROM table2
[ UNION [ALL]
SELECT column_name_1, ..., column_name_n
FROM table3
...];
```


Пример 4-5. Да се изведат идентификаторите на държавите, в които има клиенти или отдели на фирмата.

```
SELECT COUNTRY_ID
FROM CUSTOMERS
UNION
SELECT COUNTRY_ID
FROM DEPARTMENTS
ORDER BY COUNTRY_ID
```

Резултатният набор от изпълнението на заявката има вида, показан на Фигура 4-6.

👉 При обединение ORDER BY клаузата може да бъде приложена само на обединения резултат, не и на отделните заявки. В нея колоните могат да се цитират по име или по индекс, стартирайки от 1. Имената на колоните в крайния резултат идват от първата

заявка, така че ако са нужни конкретни имена на колоните там е мястото да бъдат именувани.

 По подразбиране операторът UNION премахва дублиращите се редове. Но все пак ако трябва да бъдат запазени в резултатния набор може да се използва клаузата UNION ALL.


	COUNTRY_ID
1	BE
2	BG
3	CA
4	DE
5	FR
6	IT
7	UK
8	US

Фигура 4-6. Резултати от UNION – държави с клиенти или отдели.

Пример 4-6. Да се изведат идентификаторите на държавите, в които има клиенти или отдели на фирмата. Нека в резултатния набор участват и дублиращите се записи.

```
SELECT COUNTRY_ID
FROM CUSTOMERS
UNION ALL
SELECT COUNTRY_ID
FROM DEPARTMENTS
ORDER BY 1
```

Резултатния набор от изпълнението на заявката изглежда по начина, показан на Фигура 4-7.

 Този пример демонстрира сортиране по индекс на колона, но за предпочитане е колоните да бъдат цитирани по име.

	COUNTRY_ID
1	BE
2	BG
3	BG
4	BG
5	BG
6	BG
7	BG
8	BG
9	BG
10	BG
11	BG

✓ Query executed successfully.


Фигура 4-7. Резултати от UNION ALL.

4.3.1. Задачи

Задача 4-4. Да се изведат всички малки имена на клиенти и служители с евентуалните повторения, сортирани в низходящ ред по име.

Задача 4-5. Да се изведат име и фамилия на клиенти и служители без повторения, а като трета колона за клиентите да се използва израз, генериращ низа „Клиент (<идентификатор>)“, за служителите – „Служител (<идентификатор>)“.

4.4. INTERSECT

 Операторът INTERSECT (сечение) комбинира резултатните набори, получени от изпълнението на две заявки в общ резултатен набор, съдържащ сечението на редовете от двете множества, т.е. съдържа общите за двата резултатни набора редове, ако има дубликати те се елиминират.

При използването на оператора INTERSECT трябва да бъдат спазвани следните правила:

- Броят на колоните във всички заявки трябва да бъде еднакъв;
- Колоните трябва да бъдат от съвместими типове от данни.



Фигура 4-8. Резултатно множество от записи при оператор INTERSECT (сечение).

Синтаксис на оператора INTERSECT:

```
SELECT column_name_1, ..., column_name_n
FROM table1
INTERSECT
SELECT column_name_1, ..., column_name_n
FROM table2;
```

Пример 4-7. Да се изведат идентификаторите на държавите, в които има клиенти и отдели на фирмата едновременно.

```
SELECT COUNTRY_ID
FROM CUSTOMERS
INTERSECT
SELECT COUNTRY_ID
FROM DEPARTMENTS
```

Ето го и резултатния набор от изпълнението на заявката:

Results		Messages	
		COUNTRY_ID	
1		BE	
2		BG	
3		IT	
4		US	

Фигура 4-9. Резултати от INTERSECT.

☞ Не всички реляционни езици поддържат оператора INTERSECT, но за да се изведе същия резултатен набор може да се използва алтернативен код. Например MySQL не

поддържа оператора INTERSECT, но може да бъде симулиран, използвайки предикатите IN, EXISTS или ANY.

📖 Операторът IN е логически оператор и позволява да се провери дали дадена стойност съвпада с някоя от множество стойности – ако да, връща true, иначе false.

📖 ANY също е логически оператор, който сравнява скаларна стойност с множество от стойности, върнати от подзаявка. ANY се използва в комбинация с оператор за сравнение. Ако се използва ANY с оператор равно (=), както е демонстриран в следващите заявки, той е еквивалент на IN.

📖 Операторът EXISTS е логически оператор и ще върне стойност TRUE, ако вложената заявка върне поне един ред, иначе false (при празен резултатен набор).

```
--MySQL
SELECT DISTINCT COUNTRY_ID
FROM CUSTOMERS
WHERE COUNTRY_ID IN (SELECT COUNTRY_ID
                     FROM DEPARTMENTS)

--или
SELECT DISTINCT COUNTRY_ID
FROM CUSTOMERS
WHERE EXISTS (SELECT *
              FROM DEPARTMENTS
              WHERE DEPARTMENTS.COUNTRY_ID = CUSTOMERS.COUNTRY_ID)

--или
SELECT DISTINCT COUNTRY_ID
FROM CUSTOMERS
WHERE COUNTRY_ID = ANY (SELECT COUNTRY_ID
                       FROM DEPARTMENTS)
```

Този код ще работи и на други платформи, поддържащи оператора IN. За да се премахнат евентуалните дубликати се използва ключовата дума DISTINCT.

4.4.1. Задачи

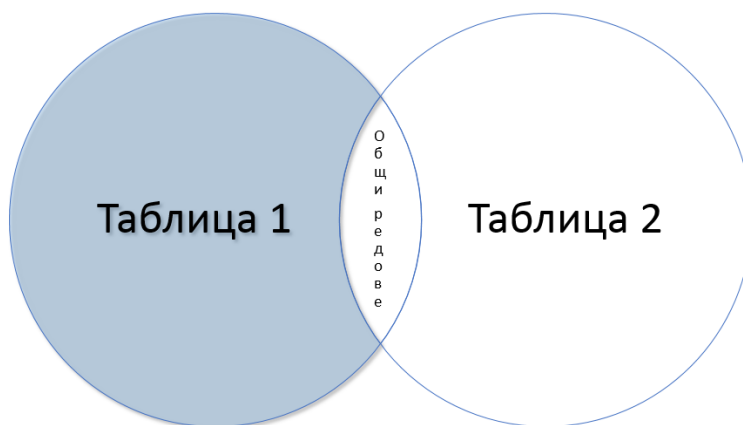
Задача 4-6. Да се изведат общите собствени имена на клиенти и служители.

4.5. MINUS

📖 Операторът MINUS (разлика) връща редовете, върнати от първата заявка, които не се срещат измежду редовете, върнати от втората заявка.

При използването на оператора MINUS трябва да бъдат спазвани следните правила:

- Броят на колоните във двете заявки трябва да бъде еднакъв;
- Колоните трябва да бъдат от съвместими типове от данни.



Фигура 4-10. Резултатно множество от записи при оператор MINUS (разлика).

Синтаксис на оператора MINUS:

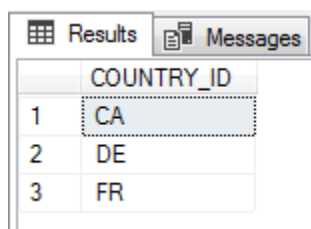
```
SELECT column_name1, ..., column_name_n
FROM table1
MINUS
SELECT column_name1, ..., column_name_n
FROM table2
```

👉 MySQL не поддържа този оператор. В MSSQL операторът е EXCEPT. Добрата новина е, че може да се реализира с алтернативен код.

Пример 4-8. Да се изведат идентификаторите на държавите, в които има клиенти и в същото време няма отдели на фирмата.

```
-- MSSQL
SELECT COUNTRY_ID FROM CUSTOMERS
EXCEPT
SELECT COUNTRY_ID FROM DEPARTMENTS
```

Резултатният набор от изпълнението на заявката е следният:



	COUNTRY_ID
1	CA
2	DE
3	FR

Фигура 4-11. Резултат от оператор EXCEPT (разлика).


```
-- Oracle
SELECT COUNTRY_ID
FROM CUSTOMERS
MINUS
SELECT COUNTRY_ID
FROM DEPARTMENTS
```

Операторът може да бъде симулиран, използвайки алтернативен код, който би върнал същите резултати, с предикатите NOT IN, NOT EXISTS или ALL:

```
-- MySQL (също Oracle и MSSQL)
SELECT DISTINCT COUNTRY_ID
FROM CUSTOMERS
WHERE NOT EXISTS (SELECT * FROM DEPARTMENTS
                  WHERE DEPARTMENTS.COUNTRY_ID = CUSTOMERS.COUNTRY_ID)

-- или
SELECT DISTINCT COUNTRY_ID
FROM CUSTOMERS
WHERE COUNTRY_ID NOT IN (SELECT COUNTRY_ID FROM DEPARTMENTS)

-- или
SELECT DISTINCT COUNTRY_ID
FROM CUSTOMERS
WHERE COUNTRY_ID <> ALL (SELECT COUNTRY_ID FROM DEPARTMENTS)
```

 ALL е логически оператор и се използва в комбинация с оператор за сравнение. Позволява да се сравни скаларна стойност с множество от стойности, върнати от вложена заявка. Връща true ако операторът за сравнение върне true за всички сравнения, иначе false. Операторът за сравнение <>, използван в комбинация с ALL, е аналог на оператора NOT IN.

4.5.1. Задачи

Задача 4-7. Да се изведат собствените имена на клиенти, които не се срещат сред тези на служителите.

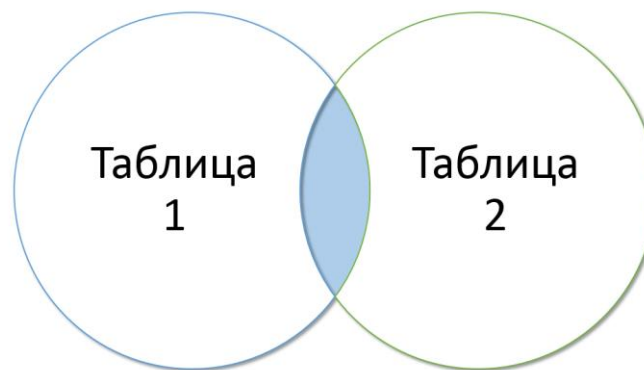
4.6. JOIN

📖 Операторът JOIN [8] се използва за извличане на данни от две или повече таблици, като редовете им се комбинират на база на логическа връзка между таблиците. Обикновено тази връзка е първичен/външен ключ, но не е задължително.

Логическата връзка често се нарича условие на съединението, което може да бъде реализирано във FROM или WHERE клаузата. Ще бъде демонстриран и в двата варианта.

4.6.1. NATURAL JOIN

📖 NATURAL JOIN извежда всички възможни свързани двойки от записи, така че всяка двойка да има равенство на стойностите във всички атрибути с еднакви имена от двете релации. Операторът създава неявна join клауза, базирана на общите колони на двете таблици, чиито редове се съединяват.



Фигура 4-12. Диаграма на оператор NATURAL JOIN.

NATURAL JOIN синтаксис:

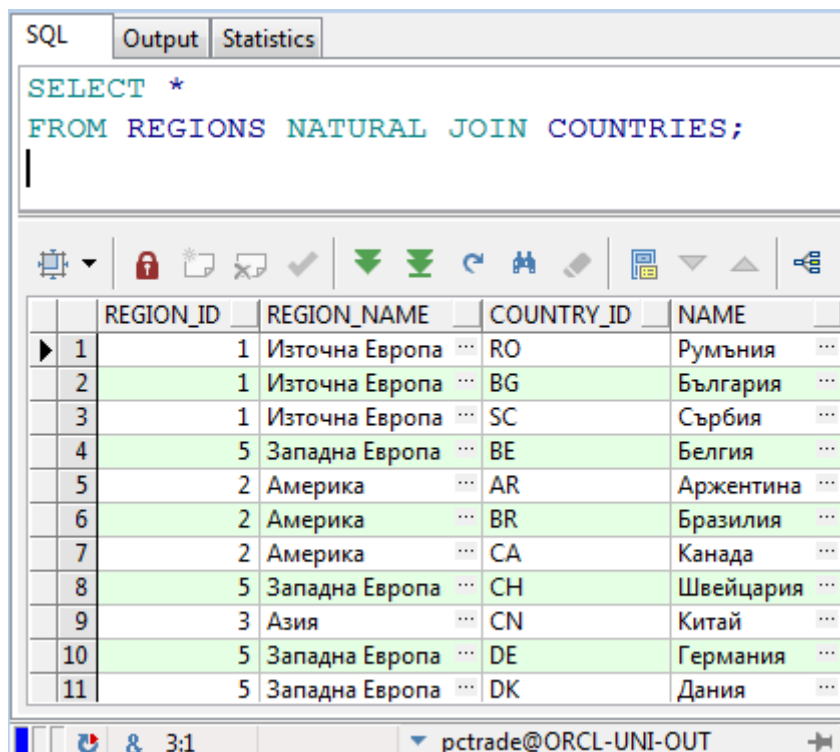
```
SELECT *  
FROM table1 NATURAL JOIN table2;
```

👉 Тъй като T-SQL не поддържа NATURAL JOIN, този оператор ще бъде демонстриран с PL/SQL в Oracle. В останалите езици може да се използва алтернативен код.

Пример 4-9. Да се изведат държавите и регионите, в които се намират.

```
SELECT *  
FROM REGIONS NATURAL JOIN COUNTRIES;
```

Резултатният набор от изпълнението на заявката е следният:



The screenshot shows the SQL Developer interface with the 'Output' tab selected. The query 'SELECT * FROM REGIONS NATURAL JOIN COUNTRIES;' is entered in the SQL window. Below the query, a table displays the results of the join. The table has five columns: an index, REGION_ID, REGION_NAME, COUNTRY_ID, and NAME. There are 11 rows of data, each representing a country and its corresponding region. The regions are 'Източна Европа', 'Западна Европа', 'Америка', 'Азия', and 'Дания'.

	REGION_ID	REGION_NAME	COUNTRY_ID	NAME
1	1	Източна Европа	RO	Румъния
2	1	Източна Европа	BG	България
3	1	Източна Европа	SC	Сърбия
4	5	Западна Европа	BE	Белгия
5	2	Америка	AR	Аржентина
6	2	Америка	BR	Бразилия
7	2	Америка	CA	Канада
8	5	Западна Европа	CH	Швейцария
9	3	Азия	CN	Китай
10	5	Западна Европа	DE	Германия
11	5	Западна Европа	DK	Дания

Фигура 4-13. Резултат от оператор NATURAL JOIN.

4.6.2. INNER JOIN

Вътрешните съединения (INNER JOIN) извеждат редовете от две или повече таблици, които имат съвпадащи стойности в колоните, посочени в условието за сравнение.

Може да се каже, че този оператор разширява синтаксиса на NATURAL JOIN, като дава възможност да бъдат сравнявани колони с различни имена.

INNER JOIN синтаксис:

```
SELECT column_name_1,..., column_name_n  
FROM table1 INNER JOIN table2 ON table1.column_name = table2.column_name;
```

Пример 4-10. Да се изведат държавите и регионите, в които се намират.

```
SELECT *  
FROM COUNTRIES INNER JOIN REGIONS ON REGIONS.REGION_ID = COUNTRIES.REGION_ID
```

Ето резултатния набор от изпълнението на заявката:

Results		Messages			
	COUNTRY_ID	NAME	REGION_ID	REGION_ID	NAME
1	AR	Аржентина	2	2	Америка
2	BE	Белгия	5	5	Западна Европа
3	BG	България	1	1	Източна Европа
4	BR	Бразилия	2	2	Америка
5	CA	Канада	2	2	Америка
6	CH	Швейцария	5	5	Западна Европа
7	CN	Китай	3	3	Азия
8	DE	Германия	5	5	Западна Европа
9	DK	Дания	5	5	Западна Европа
10	EG	Египет	4	4	Среден изток и Африка
11	FR	Франция	5	5	Западна Европа
12	HK	Хонгконг	3	3	Азия
13	IL	Израел	4	4	Среден изток и Африка
14	IN	Индия	3	3	Азия
15	IT	Италия	5	5	Западна Европа

Query executed successfully.

Фигура 4-14. Резултат от оператор INNER JOIN.

Ако бъде пропусната ключовата дума INNER резултатът ще бъде същият, защото това е типът съединение по подразбиране. Разликата между INNER JOIN и NATURAL JOIN е броят на колоните, върнати като резултат – вторият не връща колони с дублиращи се имена.

Следващият пример демонстрира заявка, в която се прави съединение на три таблици.

Пример 4-11. Да се изведат имената на клиентите, имената на държавите, от които са, и имената на регионите на държавите.

```
SELECT FNAME, LNAME, COUNTRIES.NAME, REGIONS.NAME
FROM CUSTOMERS JOIN COUNTRIES ON CUSTOMERS.COUNTRY_ID = COUNTRIES.COUNTRY_ID
JOIN REGIONS ON REGIONS.REGION_ID = COUNTRIES.REGION_ID
```

Част от резултатния набор от изпълнението на заявката е показан на Фигура 4-15.

Сред записите, които участват в резултатния набор на горния пример, няма да се появят регионите, които нямат държави, както и държавите, за които няма въведени клиенти – това прави вътрешното съединение, връщащо само редовете, които имат съответни в таблиците, които участващи в операцията.

	FNAME	LNAME	NAME	NAME
82	Хана	Кант	Германия	Западна Европа
83	Хана	Филд	Германия	Западна Европа
84	Маргет	Полини	Италия	Западна Европа
85	Хари	Тайлър	Канада	Америка
86	Маргит	Гарнър	Италия	Западна Европа
87	Мария	Вардън	Италия	Западна Европа
88	Марло	Ландис	Италия	Западна Европа
89	Кеир	Гарсия	Италия	Западна Европа
90	Марло	Лотън	Италия	Западна Европа
91	Мег	Сен	Белгия	Западна Европа
92	Марга...	Паул	Италия	Западна Европа
93	Хари	Тейлър	Канада	Америка
94	Маргит	Гарнър	Италия	Западна Европа
95	Мария	Варс	Италия	Западна Европа
96	Марло	Ландис	Италия	Западна Европа

Query executed successfully.

Фигура 4-15. Резултат от оператор INNER JOIN.

4.6.3. LEFT OUTER JOIN

LEFT OUTER JOIN връща като резултатен набор всички записи от лявата таблица (Таблица 1), комбинирани със съответните им записи (спрямо условието на съпоставка) от дясната таблица (Таблица 2), като тук вече ще участват и тези от лявата, които нямат съответни в дясната, като техните колони, идващи от дясната таблица, ще съдържат нулеви стойности.



Фигура 4-16. Диаграма на оператор LEFT OUTER JOIN.

LEFT OUTER JOIN синтаксис:

```
SELECT column_name_1,..., column_name_n  
FROM t1 LEFT [OUTER] JOIN t2 ON t1.column_name = t2.column_name;
```

Пример 4-12. Да се изведат регионите и държавите, които се намират в тях. Резултатният набор да включва и регионите, в които няма въведени държави.

```
SELECT *  
FROM REGIONS R LEFT OUTER JOIN COUNTRIES C ON R.REGION_ID = C.REGION_ID
```

👉 В тази заявка се използват псевдоними за имената на таблиците (преименувани са за обхвата на заявката) с цел съкращаване на кода. При преименуването може да се използва ключовата дума AS, но и без нея също работи в повечето езици.

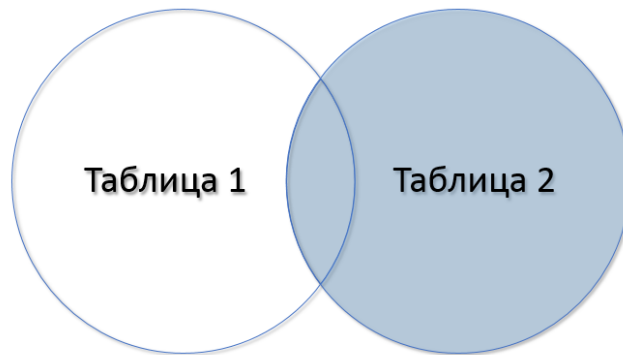
Ето част от резултатния набор от изпълнението на заявката:

	REGION_ID	NAME	COUNTRY_ID	NAME	REGION_ID
16	4	Среден изток и Африка	KW	Кувейт	4
17	4	Среден изток и Африка	NG	Нигерия	4
18	4	Среден изток и Африка	ZM	Замбия	4
19	4	Среден изток и Африка	ZW	Зимбабве	4
20	5	Западна Европа	BE	Белгия	5
21	5	Западна Европа	CH	Швейцария	5
22	5	Западна Европа	DE	Германия	5
23	5	Западна Европа	DK	Дания	5
24	5	Западна Европа	FR	Франция	5
25	5	Западна Европа	IT	Италия	5
26	5	Западна Европа	NL	Холандия	5
27	5	Западна Европа	UK	Великобритания	5
28	6	Северен полюс	NULL	NULL	NULL

Фигура 4-17. Резултат от оператор LEFT OUTER JOIN.

4.6.4. RIGHT OUTER JOIN

📖 RIGHT OUTER JOIN връща като резултатен набор всички записи от дясната таблица (Таблица 2), комбинирани със съответните им записи (спрямо условието на съпоставка) от лявата таблица (Таблица 1), като тук вече ще участват и тези от дясната, които нямат съответни в лявата, като за тях колоните, идващи от лявата таблица, ще съдържат нулеви стойности.



Фигура 4-18. Диаграма на оператор RIGHT OUTER JOIN.

RIGHT OUTER JOIN синтаксис:

```
SELECT column_name_1, ..., column_name_n
FROM t1 RIGHT [OUTER] JOIN t2 ON t1.column_name = t2.column_name;
```

Пример 4-13. Да се изведат държавите и регионите, в които се намират. Резултатния набор да включва държавите, за които няма въведен регион.


```
SELECT *
FROM REGIONS RIGHT JOIN COUNTRIES ON REGIONS.REGION_ID = COUNTRIES.REGION_ID
```

Част от резултатния набор от изпълнението на заявката е показан на Фигура 4-19.

	REGION_ID	NAME	COUNTRY_ID	NAME	REGION_ID
1	2	Америка	AR	Аржентина	2
2	NULL	NULL	AU	Австралия	NULL
3	5	Западна Европа	BE	Белгия	5
4	1	Източна Европа	BG	България	1
5	2	Америка	BR	Бразилия	2
6	2	Америка	CA	Канада	2
7	5	Западна Европа	CH	Швейцария	5
8	3	Азия	CN	Китай	3
9	5	Западна Европа	DE	Германия	5
10	5	Западна Европа	DK	Дания	5
11	4	Среден изток и Африка	EG	Египет	4
12	5	Западна Европа	FR	Франция	5
13	NULL	NULL	GR	Гърция	NULL
14	3	Азия	HK	Хонгконг	3

Фигура 4-19. Резултат от оператор RIGHT JOIN.

4.6.5. FULL OUTER JOIN

 FULL OUTER JOIN връща като резултатен набор всички записи от лявата таблица, комбинирани със съответните им записи от дясната таблица, като тук вече ще участват всички редове и от двете, като тези от лявата, които нямат съответни в дясната, ще имат нулеви стойности за колоните, идващи от дясната таблица и обратно.



Фигура 4-20. Диаграма на оператор FULL OUTER JOIN.

FULL OUTER JOIN синтаксис:

```
SELECT column_name_1, ..., column_name_n  
FROM t1 FULL OUTER JOIN t2 ON t1.column_name = t2.column_name;
```

Пример 4-14. Да се изведат държавите и регионите, в които се намират. Резултатния набор да включва държавите, за които няма въведен регион и регионите, за които няма въведени държави.

```
SELECT *  
FROM REGIONS FULL JOIN COUNTRIES ON REGIONS.REGION_ID = COUNTRIES.REGION_ID
```

Част от резултатния набор от изпълнението на заявката е показан на Фигура 4-21.


	REGION_ID	NAME	COUNTRY_ID	NAME	REGION_ID
18	4	Среден изток ...	ZM	Замбия	4
19	4	Среден изток ...	ZW	Зимбабве	4
20	5	Западна Европа	BE	Белгия	5
21	5	Западна Европа	CH	Швейцария	5
22	5	Западна Европа	DE	Германия	5
23	5	Западна Европа	DK	Дания	5
24	5	Западна Европа	FR	Франция	5
25	5	Западна Европа	IT	Италия	5
26	5	Западна Европа	NL	Холандия	5
27	5	Западна Европа	UK	Великобритания	5
28	6	Северен полюс	NULL	NULL	NULL
29	NULL	NULL	AU	Австралия	NULL
30	NULL	NULL	GR	Гърция	NULL

Фигура 4-21. Резултат от оператор FULL JOIN.

4.6.6. Други JOIN вариации

За изчерпателност тук ще бъдат споменати някои JOIN вариации, които всъщност представляват частни случаи на някои от описаните до момента оператори, но са известни с други имена.

4.6.6.1. EQUI-JOIN

 По същество това е NATURAL JOIN или INNER JOIN, чието условие за съпоставка използва оператор „=“, т.е. горните два в най-честата си употреба са EQUI-JOIN оператори.

Синтаксис на EQUI-JOIN:

```
SELECT column_name_1, ..., column_name_n
FROM table1, table2
WHERE table1.column_name = table2.column_name;
```

Пример 4-15. Да се изведат държавите и регионите, в които се намират.

```
SELECT *
FROM REGIONS, COUNTRIES
WHERE REGIONS.REGION_ID = COUNTRIES.REGION_ID
```

Получава се резултатният набор, показан на Фигура 4-22.

Results		Messages			
	REGION_ID	NAME	COUNTRY_ID	NAME	REGION_ID
1	2	Америка	AR	Аржентина	2
2	5	Западна Европа	BE	Белгия	5
3	1	Източна Европа	BG	България	1
4	2	Америка	BR	Бразилия	2
5	2	Америка	CA	Канада	2
6	5	Западна Европа	CH	Швейцария	5
7	3	Азия	CN	Китай	3
8	5	Западна Европа	DE	Германия	5
9	5	Западна Европа	DK	Дания	5
10	4	Среден изток и Африка	EG	Египет	4
11	5	Западна Европа	FR	Франция	5
12	3	Азия	HK	Хонгконг	3
13	4	Среден изток и Африка	IL	Израел	4

Query executed successfully.

Фигура 4-22. Резултат от оператор EQUI-JOIN.

4.6.6.2. NON-EQUI JOIN

NON-EQUI JOIN е JOIN с WHERE клауза, която съдържа условие, което използва някой от операторите за сравнение, различен от „=" (>, <, >=, <=, <>, !=). Извеждат се всички редове, за които е изпълнено поставеното условие.

4.6.6.3. SEMI-JOIN

Извежда редове от първата таблица, за които има поне едно съвпадение с редове от втората релация. Всеки ред от първата релация участва най-много веднъж в резултатния набор, дори да има повече от един съвпадащи реда във втората релация. Реализира се с предикатите IN и EXISTS.

Пример 4-16. Да се изведат отделите, в които има назначени служители.

```
SELECT NAME
FROM DEPARTMENTS
WHERE DEPARTMENT_ID IN (SELECT DEPARTMENT_ID
                        FROM EMPLOYEES)
```

Получава се резултатният набор, показан на Фигура 4-23.

Results		Messages
	NAME	
1	Покупки	
2	Човешки ресурси	
3	Доставки	
4	ИТ	
5	Връзки с обществеността	
6	Продажби	
7	Мениджмънт	
8	Финанси	
9	Счетоводство	

Фигура 4-23. Резултат от оператор SEMI-JOIN.

Същият резултат може да бъде постигнат и с предиката EXISTS:

```
SELECT NAME
FROM DEPARTMENTS D
WHERE EXISTS (SELECT 1
              FROM EMPLOYEES E
              WHERE E.DEPARTMENT_ID = D.DEPARTMENT_ID)
```

4.6.6.4. ANTI JOIN

Извежда редовете от първата релация, които нямат съвпадение с редове от втората релация. Реализира се с предикатите NOT IN и NOT EXISTS. По логика той прилича на оператора MINUS.

Пример 4-17. Да се изведат имената на клиентите, които все още не са правили поръчки.

```
SELECT FNAME, LNAME
FROM CUSTOMERS
WHERE CUSTOMER_ID NOT IN (SELECT CUSTOMER_ID
                          FROM ORDERS)
```

Резултатният набор е показан на Фигура 4-24.

Results		Messages
	FNAME	LNAME
1	Ивайла	Петрова
2	Георги	Георгиев
3	Даниел	Костов
4	Георги	Димитров
5	Иван	Иванов
6	Катя	Георгиева
7	Elizabeth	Brown
8	Илия	Бояджиев
9	Иван	Луков
10	Ради	Колев
11	Райна	Драганова
12	Симеон	Петров
13	София	Атанасова

✓ Query executed successfully.

Фигура 4-24. Резултат от оператор ANTI JOIN.

Същият резултат може да бъде постигнат и с предиката NOT EXISTS:

```
SELECT FNAME, LNAME
FROM CUSTOMERS
WHERE NOT EXISTS (SELECT *
                  FROM ORDERS
                  WHERE CUSTOMERS.CUSTOMER_ID = ORDERS.CUSTOMER_ID)
```

4.6.6.5. CROSS JOIN

CROSS JOIN генерира резултатен набор, в който всеки ред от едната таблица е комбиниран с всеки ред от другата таблица – т.нар. Декартово произведение. Броят на редовете в резултатния набор е равен на произведението от броя редове в първата и броя редове във втората таблица. Той е полезен ако са нужни всички комбинации на редове – напр. цветове, размери, дати и т.н. Синтаксис:

```
SELECT *
FROM table1 CROSS JOIN table2;
```

Пример 4-18. Да се изведат комбинациите между всички региони и държави, сортирани по име на държава.

```
SELECT *
FROM REGIONS CROSS JOIN COUNTRIES
ORDER BY COUNTRIES.NAME
```


Ето част от резултатния набор, демонстриращ, че държавите са комбинирани с всеки регион (и обратно - регионите с всяка държава):

Results Messages					
	REGION_ID	NAME	COUNTRY_ID	NAME	REGION_ID
1	3	Азия	AU	Австралия	NULL
2	2	Америка	AU	Австралия	NULL
3	5	Западна Европа	AU	Австралия	NULL
4	1	Източна Европа	AU	Австралия	NULL
5	6	Северен полюс	AU	Австралия	NULL
6	4	Среден изток и Африка	AU	Австралия	NULL
7	4	Среден изток и Африка	AR	Аржентина	2
8	6	Северен полюс	AR	Аржентина	2
9	1	Източна Европа	AR	Аржентина	2
10	5	Западна Европа	AR	Аржентина	2
11	2	Америка	AR	Аржентина	2
12	3	Азия	AR	Аржентина	2
13	3	Азия	BE	Белгия	5
14	2	Америка	BE	Белгия	5
15	5	Западна Европа	BE	Белгия	5
16	1	Източна Европа	BE	Белгия	5
17	6	Северен полюс	BE	Белгия	5
18	4	Среден изток и Африка	BE	Белгия	5

Query executed successfully.

Фигура 4-25. Резултат от оператор CROSS JOIN.

Лесно могат да се филтрират географски верните комбинации с добавяне на условие за равенство в WHERE клаузата по REGION_ID атрибутите от двете таблици, достигайки на практика до INNER JOIN.

Същият резултат (произведение) може да бъде постигнат и чрез следната заявка (пропусκαе сортирането):

```
SELECT *
FROM REGIONS, COUNTRIES
```

4.6.7. Задачи

Задача 4-8. Да се изведат идентификаторите и датите на поръчките, както и имената на служителите, които са ги обработили.


Задача 4-9. Да се изведат имената на всички клиенти и идентификаторите на поръчките им. В резултатния набор да участват и клиентите, които все още не са правили поръчки.

Задача 4-10. Да се изведат имената на продуктите, които не са поръчвани до момента.

Задача 4-11. Да се изведат имената на всички клиенти, които са от държави в регион „Западна Европа“.


4.7. Други

4.7.1. TOP

 SELECT TOP клаузата позволява да се ограничи броят на редовете, върнати в резултата от заявката, до определен брой редове или процент. TOP връща първите N реда в неопределен ред, което налага използването на клаузата ORDER BY, за да може да бъде ограничен резултатният набор до първия N брой реда в желан ред.

Синтаксис:

```
SELECT TOP number|percent [ WITH TIES ] column_name(s)
FROM table_name
WHERE condition;
```

 В различните бази от данни този оператор често е с различни имена. MySQL поддържа клаузата LIMIT за извеждане на определен брой редове, Oracle поддържа функцията ROWNUM (от версия 12c вече има клауза OFFSET-FETCH).

MySQL:

```
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number;
```

Oracle:

```
SELECT column_name(s)
FROM table_name
WHERE ROWNUM <= number;
```

Пример 4-19. Да се изведат 7-те продукта с най-ниска цена.

```
SELECT TOP 7 *
FROM PRODUCTS
ORDER BY PRICE
```

Резултатът е:

	PRODUCT_ID	NAME	PRICE	DESCR
1	2266	Micro usb кабел ALIEN	1.50	NULL
2	3057	Писалка стилус за таблет PEN	4.00	NULL
3	2243	Кабел Zik OTG Galaxy TAB	5.00	NULL
4	1738	Скрийн протектор LENOVO YOGA TABLET 2 10.1	6.00	NULL
5	2299	Кабел OTG A+, USB-MicroUSB	6.00	NULL
6	3133	Tempered Glass за Huawei Y5	7.00	Съклен скрийн протектор, Tempered Glass за Huawei...
7	1739	USB памет Kingston DataTraveler 100 G3	9.00	6GB, USB 3.0

Фигура 4-26. Резултат от оператора TOP.

Само за демонстрация нека бъдат извлечени 7-те продукта с най-ниска цена в Oracle:

```
SELECT *
FROM (SELECT * FROM products ORDER BY PRICE)
WHERE ROWNUM <= 7
```

Резултатът е:

	PRODUCT_ID	NAME	PRICE	DESCR
1	2266	Micro usb кабел ALIEN	1,50	...
2	3057	Писалка стилус за таблет PEN	4,00	...
3	2243	Кабел Zik OTG Galaxy TAB	5,00	...
4	1738	Скрийн протектор LENOVO YOGA TABLET 2 10.1	6,00	...
5	2299	Кабел OTG A+, USB-MicroUSB	6,00	...
6	3133	Tempered Glass за Huawei Y5	7,00	Съклен скрийн протектор, Tempered Glass за Huawei Y5 2018.
7	2752	Anti Shock гръб Carbon за iPhone 5/5S/SE, Черен	9,00	...

Фигура 4-27. Резултат: 7-те продукта с най-ниска цена в Oracle.

Ако се обърне внимание на 7-я ред в двата резултатни набора става ясно, че е различен в MSSQL и Oracle. Всъщност проблемът не е в разлика между сървърите, а в това, че има поне още един продукт с цена, равна на тази на последния (7-я) извлечен. В този случай няма как да се знае кой ще е 7-ят ред, защото явно има повече от един, отговарящ на условието да бъде последен в нашия резултатен набор. Ако трябва да бъдат изведени и продуктите с цена, равна на последната изведена, трябва да се използва TOP с WITH TIES в MSSQL:

```
SELECT TOP 7 WITH TIES *
FROM PRODUCTS
ORDER BY PRICE
```

Сега в резултата се вижда, че има още 3 продукта на същата цена като 7-я:

Results		Messages		
	PRODUCT_ID	NAME	PRICE	DESCR
1	2266	Micro usb кабел ALIEN	1.50	NULL
2	3057	Писалка стилус за таблет PEN	4.00	NULL
3	2243	Кабел Zik OTG Galaxy TAB	5.00	NULL
4	1738	Скрийн протектор LENOVO YOGA TABLET 2 10.1	6.00	NULL
5	2299	Кабел OTG A+, USB-MicroUSB	6.00	NULL
6	3133	Tempered Glass за Huawei Y5	7.00	Съклен скриин протектор, Tempered Glass за Huaw...
7	2496	Зарядно устройство за автомобил	9.00	Зарядно устройство за автомобил dual USB A+ Char...
8	2752	Anti Shock гръб	9.00	Anti Shock гръб Carbon за iPhone 5/5S/SE, Черен
9	2264	Кабел A+ lightning, MFI за iPhone	9.00	NULL
10	1739	USB памет Kingston DataTraveler 100 G3	9.00	6GB, USB 3.0

Фигура 4-28. Резултат от оператора TOP WITH TIES.

4.7.2. OFFSET и FETCH

Клаузите OFFSET и FETCH [9] се използват заедно със SELECT и ORDER BY в случаите, когато е нужно да се вземат определени подмножества от редовете, върнати от SELECT-а – например при нужда от показване на първите N реда от резултатния набор, след това вторите N на брой и т.н. (известно в web приложенията като paging).

OFFSET указва броя на редовете, които да бъдат пропуснати преди редовете, които ще бъдат включени в резултатния набор.

Клаузата FETCH определя броя на редовете, които да бъдат върнати след OFFSET.

Синтаксис:

```
SELECT column_name(s)
FROM table_name
ORDER BY column_list [ASC | DESC]
OFFSET offset_row_count {ROW | ROWS}
FETCH {FIRST | NEXT} fetch_row_count {ROW | ROWS} ONLY
```

- ☞ OFFSET може да бъде използвана единствено с ORDER BY клауза, не може да бъде използвана самостоятелно. Стойността на OFFSET може да бъде константа/променлива, трябва да бъде по-голяма или равна на 0, не може да бъде отрицателна, в противен случай ще върне грешка.
- ☞ FETCH не може да се използва самостоятелно, използва се в комбинация с OFFSET. Стойността на FETCH може да бъде константа/променлива, трябва да бъде по-голяма или равна на 1.

Пример 4-20. Да се изведат първите 5 служители, след което вторите 5, сортирани по идентификатор (с цел лесно проследяване на порциите редове, които се извличат).

```
-- Първите 5 реда
SELECT EMPLOYEE_ID, FNAME, LNAME, SALARY
FROM EMPLOYEES
ORDER BY EMPLOYEE_ID
OFFSET 0 ROWS
FETCH NEXT 5 ROWS ONLY

-- Вторите 5 реда (от 6-ти до 10-ти)
SELECT EMPLOYEE_ID, FNAME, LNAME, SALARY
FROM EMPLOYEES
ORDER BY EMPLOYEE_ID
OFFSET 5 ROWS
FETCH NEXT 5 ROWS ONLY
```

Резултатът е:

Results		Messages		
	EMPLOYEE_ID	FNAME	LNAME	SALARY
1	100	Иван	Иванов	27000.00
2	101	Петя	Василева	18000.00
3	102	Димитър	Господинов	19000.00
4	103	Александър	Петров	12000.00
5	104	Боряна	Иванова	12000.00

	EMPLOYEE_ID	FNAME	LNAME	SALARY
1	105	Димо	Тодоров	7800.00
2	106	Васил	Петров	8800.00
3	107	Диана	Лазарова	7200.00
4	108	Нина	Георгиева	12000.00
5	109	Даниел	Филипов	10000.00

Фигура 4-29. Резултат от OFFSET и FETCH.

Пример 4-21. Да се изведат петимата служители, започвайки от 10-ти ред, подредени по дата на постъпване в компанията. Първата заявка ще покаже всички с цел демонстрация, втората ще извърши подбора.

```
-- ВСИЧКИ
SELECT EMPLOYEE_ID, FNAME, LNAME, SALARY
FROM EMPLOYEES
ORDER BY HIRE_DATE DESC
```

```
-- от 10 до 14 ред включително
SELECT EMPLOYEE_ID, FNAME, LNAME, SALARY
FROM EMPLOYEES
ORDER BY HIRE_DATE DESC
OFFSET 9 ROWS -- прескачат се първите 9
FETCH NEXT 5 ROWS ONLY -- вземат се следващите 5 от 10-я включително
```

Резултатът е:

	EMPLOYEE_ID	FNAME	LNAME	SALARY
1	140	Делян	Петров	3500.00
2	153	Кристиан	Огнянов	8300.00
3	166	Стефан	Александров	6900.00
4	172	Елена	Гатева	7600.00
5	195	Виолета	Донева	3800.00
6	171	Станислав	Стефанов	7600.00
7	165	Добри	Лесов	8800.00
8	185	Александра	Бонева	5100.00
9	133	Димитър	Матев	5400.00
10	149	Елена	Златкова	9500.00
11	123	Стоил	Вълков	8900.00
12	191	Тихомир	Маринов	2300.00
13	161	Сара	Иванова	7800.00
14	138	Стефан	Стилев	4100.00
15	158	Александър	Михов	9100.00
16	190	Михаил	Гатев	3900.00
17	194	Димитър	Маринов	4200.00
18	178	Карина	Георгиева	7100.00
19	115	Александър	Кръстев	6100.00
20	122	Петър	Костов	9900.00

	EMPLOYEE_ID	FNAME	LNAME	SALARY
1	149	Елена	Златкова	9500.00
2	123	Стоил	Вълков	8900.00
3	191	Тихомир	Маринов	2300.00
4	161	Сара	Иванова	7800.00
5	138	Стефан	Стилев	4100.00

Фигура 4-30. Резултат от OFFSET и FETCH за порция от 5 реда.

4.7.3. Задачи

Задача 4-12. Да се изведат вторите 10 най-добре платени служители (подредени по заплата низходящо).

Задача 4-13. Да се изведат име, фамилия и пол на клиентите, направили последните 5 поръчки.

5. Изгледи

5.1. В тази тема...

Ще бъдат разгледани:

- Изгледите - какво представляват и как се създават;
- За какво могат да бъдат използвани;
- Какви ограничения трябва да бъдат съобразени при работа с тях.

5.2. Теоретични понятия

Изгледът представлява виртуална таблица, базирана на SELECT израз, от чийто резултатен набор идва и съдържанието ѝ. Т.е. изгледът няма собствени данни, той се представя в базата данни само със своята дефиниция.

👉 Забележка: описанието дотук не важи за материализираните изгледи (snapshots, indexed views), които съхраняват резултатния набор като собствени данни с различни опции за актуализация в зависимост от конкретната платформа.

Редовете и колоните идват от таблиците (могат да се използват също и други изгледи), използвани в SELECT израза, на който се базира изгледът.

Погледнато от друга страна изгледът може да играе роля на филтър за достъп до данните на таблиците, от които идват данните му. Изгледите могат да бъдат използвани за:

- ✓ ограничаване на достъпа на потребителите до данни само от определени редове и колони от таблица;
- ✓ комбиниране на данни (използване на съединения, обединения) от множество таблици така, че да изглеждат като от една;
- ✓ и др.

Изгледите дават възможност потребителите да виждат данните по различни начини, дори когато използват едни и същи данни по едно и също време. Това е изключително предимство, даващо възможност на потребители с различни нужди и умения да използват едни и същи данни по различен начин.

Изгледите могат също да бъдат **обновяеми**, т.е. чрез тях могат да се променят данни в таблиците, върху които те са дефинирани. Но за да е налице тази възможност има

множество правила, които трябва да се спазват. Те могат да бъдат специфични за конкретната база от данни (производител/версия), но тук ще бъдат представени няколко ограничения, които са характерни за повечето популярни бази от данни:

- тригерите, създавани за изглед, могат да бъдат дефинирани само като INSTEAD OF (а не BEFORE/AFTER за тригериращото събитие);
- не могат да се използват агрегиращи функции, сортиране, групиране, оператори за обединение/сечение/разлика;
- INSERT/UPDATE командата трябва да засяга само една от таблиците, цитирани в заявката на изгледа;
- И др.

Също така всички колони на изгледа трябва да са именувани – това е съществено за тези, които се изчисляват от аритметични изрази, функции или константи.

5.3. Създаване на изгледи

Ще бъде използван следният опростен синтаксис за създаване на изгледи:

```
CREATE VIEW VIEW_NAME [(column [ ,...n ])]  
AS  
select_statement  
[WITH CHECK OPTION]
```

Списъкът от колони след името на изгледа се използва, когато трябва да се даде определено име на колона, получена от израз, например.

Ако бъде използвана опцията WITH CHECK OPTION при опит за промяна на данните на изгледа така, че ред(ове) биха изчезнали от изгледа, операцията ще бъде отхвърлена.

Пример 5-1. Да се създаде изглед, който съдържа име и фамилия на клиентите, както и номер и дата на поръчките, които те са направили.

```
CREATE VIEW CUSTOMERORDERS  
AS  
SELECT C.FNAME + ' ' + C.LNAME AS CUSTOMER, O.ORDER_ID, O.ORDER_DATE  
FROM ORDERS O, CUSTOMERS C  
WHERE O.CUSTOMER_ID = C.CUSTOMER_ID
```

Да се изведат данните от изгледа:

```
SELECT * FROM CUSTOMERORDERS
```


Пример 5-2. Да се модифицира горният изглед така, че да съдържа и колона с името на съответния служител, обработил поръчката.

```
ALTER VIEW CUSTOMERORDERS AS
SELECT C.FNAME + ' ' + C.LNAME AS CUSTOMER,
       O.ORDER_ID, O.ORDER_DATE,
       E.FNAME + ' ' + E.LNAME AS EMPLOYEE
FROM EMPLOYEES E, ORDERS O, CUSTOMERS C
WHERE E.EMPLOYEE_ID = O.EMPLOYEE_ID
AND O.CUSTOMER_ID = C.CUSTOMER_ID
```

Пример 5-3. Да се модифицира горния изглед така, че да съдържа само поръчките, обработени от служител с идентификатор = 167.

```
ALTER VIEW CUSTOMERORDERS AS
SELECT C.FNAME + ' ' + C.LNAME AS CUSTOMER,
       O.ORDER_ID, O.ORDER_DATE,
       E.FNAME + ' ' + E.LNAME AS EMPLOYEE
FROM EMPLOYEES E, ORDERS O, CUSTOMERS C
WHERE E.EMPLOYEE_ID = O.EMPLOYEE_ID
AND O.CUSTOMER_ID = C.CUSTOMER_ID
AND E.EMPLOYEE_ID = 167
```

Пример 5-4. Да се създаде изглед, съдържащ име и фамилия на служител и общата сума на поръчките, които той е обработил.

```
CREATE VIEW EMPLOYEEORDERS AS
SELECT E.FNAME, E.LNAME, SUM(OI.UNIT_PRICE*OI.QUANTITY) AS ORDERS_TOTAL
FROM EMPLOYEES E, ORDERS O, ORDER_ITEMS OI
WHERE E.EMPLOYEE_ID = O.EMPLOYEE_ID
AND O.ORDER_ID = OI.ORDER_ID
GROUP BY E.EMPLOYEE_ID, E.FNAME, E.LNAME
```

👉 **Защо се налага групиране по E.EMPLOYEE_ID?** Защото групирането по име и фамилия ще обобщи всички служители със съвпадащи такива, а честа ситуация е да има хора, чиито име и фамилия съвпадат, което ще доведе до неточни резултати. Затова в групирането се използва и колона, която уникално да идентифицира всеки служител, разделяйки хипотетичните Иван Иванов с идентификатор 100 и Иван Иванов с идентификатор 101 в отделни групи, което е и коректното.

Да се изведат данните от изгледа, като се сортират по обща сума на поръчките низходящо.

```
SELECT * FROM EMPLOYEEORDERS
ORDER BY ORDERS_TOTAL DESC
```

Пример 5-5. Да се създаде изглед, който съдържа имена, отдел и заплата на 5-мата служители с най-висока заплата. За да бъдат извлечени служителите, подредени по заплата, очевидно ще трябва да бъдат сортирани по този критерий.

```
CREATE VIEW EMP_TOP_SALARIES
AS
SELECT TOP 5 FNAME, LNAME, NAME AS DEPARTMENT, SALARY
FROM EMPLOYEES E, DEPARTMENTS D
WHERE D.DEPARTMENT_ID = E.DEPARTMENT_ID
ORDER BY SALARY DESC
```

👉 Клаузата ORDER BY в изглед може да се използва само за определяне на редовете, които се връщат от операторите TOP или OFFSET в дефиницията на изгледа. Клаузата ORDER BY не гарантира подредени резултати при извеждане на съдържанието на изгледа, освен ако ORDER BY също не е посочен в самата заявка, използваща изгледа. Ако ORDER BY бъде използвана без гореспоменатите оператори, би се получило следното съобщение:



Msg 1033, Level 15, State 1, Procedure EMP_TOP_SALARIES, Line 6
The ORDER BY clause is invalid in views, inline functions, derived tables, subqueries, and common table expressions, unless TOP, OFFSET or FOR XML is also specified.

5.4. Манипулиране на данни чрез изглед

Тук ще бъде разгледано съвсем просто манипулиране на данни чрез изглед. Възможностите за манипулиране на данни са много по-големи от тези, които ще бъдат демонстрирани.

Има някои ограничения за модифициране на данни чрез изгледи, за които вече споменахме в началото, които могат да варират в различните версии и платформи:

- Ако е използвана опцията [WITH CHECK OPTION] редовете не могат да бъдат модифицирани така, че те да изчезнат от изгледа. Модификациите, които биха предизвикали това, биват отхвърлени със съобщение за грешка;
- Сървърът за бази данни трябва да може еднозначно да реши операцията за модифициране на базовите таблици към коя точно от тях е насочена, защото промяната може да се случи само в една от тях. Затова колоните, изброени в

конструкцията INSERT или UPDATE, трябва да принадлежат само на една от таблиците в дефиницията на изгледа;

- При модифициране на данни в колоните на базовата таблица трябва да се спазват ограниченията (ако има такива), наложени върху стойностите на тази колона;
- За да може да се изтрие ред от изглед в клаузата FROM на дефиницията му трябва да има само една таблица;
- И др.

5.4.1. Добавяне на данни през изглед

Следващият изглед, базиран на JOIN между таблиците COUNTRIES и CUSTOMERS, ще демонстрира манипулирането на данни:

```
CREATE VIEW CUSTOMERS_COUNTRIES
AS
SELECT C.COUNTRY_ID, C.NAME, C.REGION_ID,
       M.CUSTOMER_ID, M.FNAME, M.GENDER, M.LNAME,
       M.COUNTRY_ID AS CUST_COUNTRY_ID, M.EMAIL
FROM COUNTRIES C JOIN CUSTOMERS M ON C.COUNTRY_ID = M.COUNTRY_ID
```

Пример 5-6. Да се добави нов запис в таблицата CUSTOMERS през изгледа CUSTOMERS_COUNTRIES.

```
INSERT INTO CUSTOMERS_COUNTRIES (CUSTOMER_ID, FNAME, LNAME, CUST_COUNTRY_ID)
VALUES (10, 'Иван', 'Петров', 'BG')
```

Изпълнението на тази команда добавя нов запис в таблицата CUSTOMERS, попълвайки стойности само в посочените колони. Това е възможно, защото останалите колони на таблицата допускат NULL стойности, т.е. това е още едно условие, което трябва да бъде съобразено, когато се създават изгледи с идея да се добавят данни чрез тях. Това също може да се разглежда и като вариант да не се дава възможност да бъдат променяни стойностите на определени колони, пропускайки ги в изгледа.

5.4.2. Променяне на данни през изглед

Пример 5-7. Да се промени фамилията на клиент с идентификатор 10.

```
UPDATE CUSTOMERS_COUNTRIES
SET LNAME = 'Колев'
WHERE CUSTOMER_ID = 10
```

5.4.3. Изтриване на данни през изглед

Пример 5-8. Да се изтрие клиент с идентификатор 10.

```
DELETE FROM CUSTOMERS_COUNTRIES  
WHERE CUSTOMER_ID = 10
```

👉 Изпълнението на тази команда в MSSQL ще предизвика следната грешка:

Msg 4405, Level 16, State 1, Line 10

View or function 'CustomersCountries' is not updatable because the modification affects multiple base tables.

Това е защото изгледът е базиран на две таблици, а за да може да се изтрива от него трябва да е базиран само на една, както по-горе беше споменато.

5.5. Задачи

Задача 5-1. Да се създаде изглед, който съдържа имената на продуктите и общо поръчано количество от продукт.

Задача 5-2. Да се създаде изглед, който съдържа десетимата клиенти с най-голям брой поръчки. Ако последният клиент има равен брой поръчки с други клиенти, те също да участват в изгледа.


6. Транзакции

6.1. В тази тема...


Ще бъдат разгледани:

- Какво е транзакция;
- Кога е удачно да се използва;
- Как се стартира транзакция;
- Как се потвърждават/отхвърлят промените, направени в транзакция.

6.2. Теоретични понятия

 Транзакцията представлява последователност от една или повече SQL команди, които базата данни третира като единична операция – или всички операции се изпълняват, или никоя от тях. Затова транзакцията често е наричана логическа единица.

Предназначението на транзакциите е да комбинират множество команди (INSERT/UPDATE/DELETE/SELECT), чиято цел е да бъдат изпълнени всички, като в случай на неуспех на някоя от тях да могат да бъдат игнорирани промените, направени от останалите, оставяйки данните в базата в състоянието им преди стартирането на транзакцията. Ако командите не са логически свързани и промените им не зависят една от друга, то транзакция не е необходима. Не е необходима и когато се изпълнява една команда, защото по дефиниция всяка команда е атомарна, т.е. неделима.

 Транзакциите притежават следните свойства:

Атомарност (Atomicity) – всички операции в транзакцията се изпълняват успешно или в случай на пропадане всички промени, направени до момента, се игнорират.

Консистентност (Consistency) – трансформират данните от едно консистентно (вярно) състояние към друго консистентно състояние.

Изолираност (Isolation) – отделните транзакции са изолирани една от друга – никоя от тях не знае за съществуването на останалите.

Трайност (Durability) – след като транзакцията е приключила и промените са потвърдени, същите остават налични дори и при последващ срив в системата.

Промените, направени от операциите в транзакция, не са видими за другите транзакции, преди те да са потвърдени с оператора COMMIT.

☞ За да не си пречат взаимно, транзакциите използват заключване на използваните ресурси. Ако дадена транзакция изисква заключване на ресурс, използван от друга, тя обикновено се налага да изчака другата да го освободи. Освобождаването на заключванията става с приключването на транзакцията с COMMIT или ROLLBACK операторите.

6.3. Оператори за контрол на транзакциите

6.3.1. BEGIN TRANSACTION

Този оператор сигнализира за начало на транзакцията. Всички команди след него ще бъдат част от тази транзакция:

```
BEGIN TRAN[SACTION] [tran_name]
```

Опционално може да бъде зададено име на транзакцията.

☞ Някои езици нямат такъв оператор – напр. PL/SQL. В Oracle транзакцията започва от някое от следните събития – изпълнението на първата команда, която променя данни; използване на SET TRANSACTION; използване на DBMS_TRANSACTION пакета.

6.3.2. COMMIT

Този оператор сигнализира за успешен край на транзакцията и промените от нея биват направени постоянни и видими за другите транзакции.

```
COMMIT [TRAN[SACTION]] [tran_name]
```

След изпълнението му всички заключвания, направени от транзакцията, биват освободени.

6.3.3. SAVEPOINT

Точките на запис дават възможност промените до момента да бъдат записани, с възможност тези след точката на запис да бъдат отхвърлени. Това е възможност за стъпка встрани от идеологията за логическата единица, но в някои случаи може да се окаже полезна възможност.

```
SAVE TRAN[SACTION] savepoint_name
```

6.3.4. ROLLBACK

Отхвърля направените промени от транзакцията или до точката на запис, ако такава има и е указана.

```
ROLLBACK [TRAN[SACTION]] [tran_name|savepoint_name]
```

След изпълнението му всички заключения, направени от транзакцията, биват освободени.

6.3.5. SET TRANSACTION

Този оператор служи за промяна на параметрите на транзакцията, напр. промяна на нивото на изолация.

6.4. Примери

Пример 6-1. Да се направи транзакция, която добавя нов клиент и създава поръчка за него, включваща два продукта.

```
BEGIN TRANSACTION

INSERT CUSTOMERS (CUSTOMER_ID, COUNTRY_ID, FNAME, LNAME, ADDRESS, EMAIL, GENDER)
VALUES (1001, 'BG', 'Иван', 'Николов', 'бул. България 236, Пловдив',
'ivannikolov@mail.com', 'M');

INSERT ORDERS (ORDER_ID, CUSTOMER_ID, EMPLOYEE_ID, SHIP_ADDRESS, ORDER_DATE)
VALUES (1, 1001, 107, 'бул. Македония 12, Пловдив', GETDATE());

INSERT ORDER_ITEMS (ORDER_ID, PRODUCT_ID, UNIT_PRICE, QUANTITY)
VALUES (1, 1726, 99, 1);

INSERT ORDER_ITEMS (ORDER_ID, PRODUCT_ID, UNIT_PRICE, QUANTITY)
VALUES (1, 1782, 615, 1);

COMMIT TRANSACTION
```

Пример 6-2. Да се направи транзакция, която променя фамилията на клиент с идентификатор = 1001, след което отхвърля направените промени.

```
BEGIN TRANSACTION

PRINT 'Фамилия преди промяната: '

SELECT LNAME
FROM CUSTOMERS
```

```

WHERE CUSTOMER_ID = 1001

UPDATE CUSTOMERS SET LNAME = 'Иванов'
WHERE CUSTOMER_ID = 1001

PRINT 'Фамилия след промяната: '

SELECT LNAME
FROM CUSTOMERS
WHERE CUSTOMER_ID = 1001

ROLLBACK TRANSACTION

PRINT 'Фамилия след отхвърлянето на промяната: '

SELECT LNAME
FROM CUSTOMERS
WHERE CUSTOMER_ID = 1001

```

Пример 6-3. Да се направи транзакция, която въвежда нов клиент, поставя точка на запис, въвежда поръчка, след което отхвърля промените до точката на запис, т.е. отхвърля се само поръчката.

```

BEGIN TRAN

INSERT CUSTOMERS (CUSTOMER_ID, COUNTRY_ID, FNAME, LNAME, GENDER)
VALUES (1002, 'BG', 'Петър', 'Василев', 'M');

SAVE TRAN point1

INSERT ORDERS (ORDER_ID, CUSTOMER_ID, EMPLOYEE_ID, ORDER_DATE)
VALUES (2, 1002, 110, GETDATE());

ROLLBACK TRAN point1

COMMIT TRAN

```

6.5. Задачи

Задача 6-1. Да се направи транзакция транзакция, която има за цел да изтрие отдел „Мениджмънт“, като преди това прехвърли всички служители от него в отдел „Администрация“.

Задача 6-2. Да се направи транзакция транзакция, която изтрива продукт с идентификатор 1726, като преди това го изтрива от всички поръчки, в които участва, като накрая отхвърля направените промени.

6.6. Вижте още...

6.6.1. Нива на изолация на транзакциите



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 коректност на данните, но може да се отрази отрицателно на едновременната работа (конкурентността).

Нивата на изолация имат за цел да решават следните проблеми:

- **Четене на непотвърдени данни (dirty read)** – ситуация, в която транзакцията чете данни, които биват променяни от друга транзакция, която все още не ги е потвърдила/отхвърлила с commit/rollback. Ако втората транзакция отхвърли направените промени, то първата се оказва, че е прочела данни, които не съществуват;
- **Неконсистентен анализ (non repeatable read)** – случва се, когато транзакция прочете един и същ ред два пъти и получи различни резултати – очевидно променени между четенията от друга транзакция;
- **Четене на редове-фантоми (phantom read)** – случва се, когато една транзакция извлича резултатен набор по даден критерий два или повече пъти, и при различните извличания броят редове е различен, т.е. появяват се/изчезват редове при всяко следващо извличане – очевидно данните биват променяни от друга транзакция и някои редове излизат извън критерия на извличане, а нови други попадат в него.

Стандартът SQL представя 4 нива на изолация със следното поведение:

- READ UNCOMMITTED – допуска прочитане и на непотвърдени данни;
- READ COMMITTED – четене само на потвърдени данни. Това обикновено е нивото по подразбиране;
- REPEATABLE READ – повторяемо четене;

- **SERIALIZABLE** – сериализируемо.

Транзакциите имат различно поведение в зависимост от това какво ниво на изолация им е зададено. Следната таблица илюстрира поведението при различните нива на изолация:

Ниво на изолация	Четене на непотвърдени данни	Неконсистентен анализ	Четене на фантоми
READ UNCOMMITTED	Да	Да	Да
READ COMMITTED	Не	Да	Да
REPEATABLE READ	Не	Не	Да
SERIALIZABLE	Не	Не	Не

👉 Нивата на изолация варират в различните реализации на стандарта, напр. T-SQL има допълнително ниво SNAPSHOT, а PL/SQL няма READ UNCOMMITTED.

6.6.2. Демонстрация на четене на непотвърдени данни

1. Стартира се една транзакция T1, която добавя един ред, но не потвърждава/отхвърля промените все още:

```

inconsistent_tran1...deCompany (sa (51))*
BEGIN TRAN T1
INSERT CUSTOMERS(CUSTOMER_ID,COUNTRY_ID,FNAME,LNAME)
VALUES (1003, 'BG', 'Стоян', 'Петров');
  
```

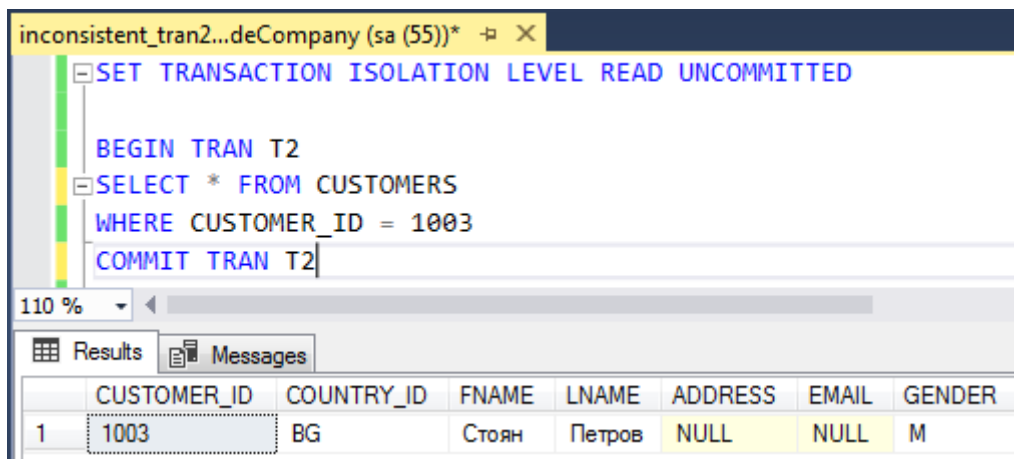
110 %

Messages

(1 row affected)

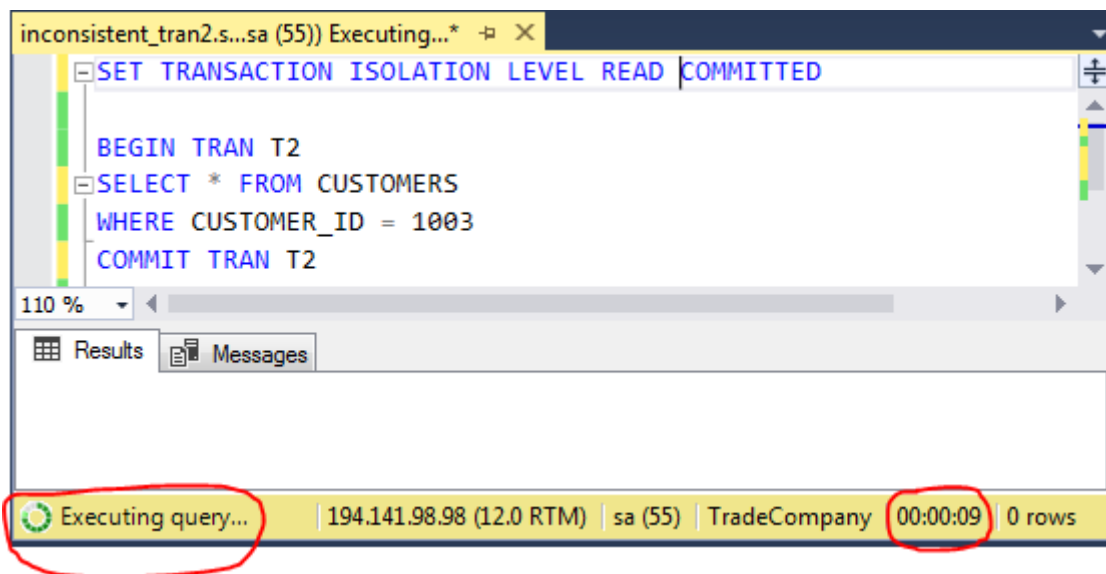
Фигура 6-1. Стартиране на транзакция T1 без потвърждаване/отхвърляне на промените.

2. Стартира се друга транзакция T2, която е с ниво на изолация READ UNCOMMITTED, и тя трябва да види новодобавения ред от T1:



Фигура 6-2. Стартиране на транзакция T2, прочитаща непотвърдените данни.


3. Променя се нивото на изолация на T2 на READ COMMITTED и тогава тя ще премине в изчакване на commit/rollback от T1 преди да прочете данните:



Фигура 6-3. Стартиране на транзакция T2 с ниво на изолация, непозволяващо прочитане на непотвърдени данни.

Транзакция T2 ще продължи да чака, докато транзакция T1 потвърди или отхвърли промените.

6.6.3. Deadlock ситуации

 Това са ситуации, 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 превъртjana 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 проблема.

Как да бъдат избягвани deadlock ситуацияите

Достъпът до таблиците да е 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!

6.6.4. Deadlock демонстрация

За целта ще бъдат стартирани две транзакции, които ще обновяват едни и същи редове в две таблици, като само редът на обновяване на таблиците ще е различен, за да се постигне заключване.



The screenshot shows a SQL query window titled "SQLQuery1.sql - 19...eCompany (sa (51))*". The query contains the following statements: `BEGIN TRANSACTION`, `UPDATE CUSTOMERS SET FNAME = 'PETER' WHERE CUSTOMER_ID = 101`, `WAITFOR DELAY '00:00:05'`, `UPDATE COUNTRIES SET NAME = 'ARGENTINA 2' WHERE COUNTRY_ID = 'AR'`, and `COMMIT`. Below the query, a scroll bar is set to 110%. The Messages pane at the bottom shows two messages: "(1 row affected)" and "(1 row affected)".

```
BEGIN TRANSACTION

UPDATE CUSTOMERS SET FNAME = 'PETER' WHERE CUSTOMER_ID = 101

WAITFOR DELAY '00:00:05'

UPDATE COUNTRIES SET NAME = 'ARGENTINA 2' WHERE COUNTRY_ID = 'AR'

COMMIT
```

110 %

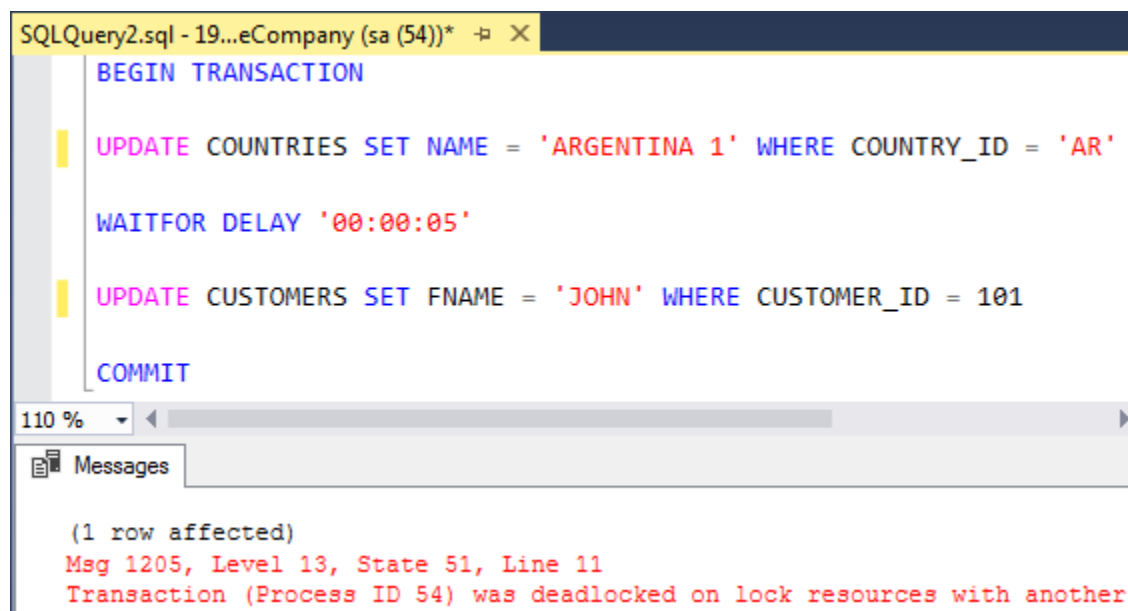
Messages

(1 row affected)

(1 row affected)

Фигура 6-4. Демонстрация на deadlock – резултати от първата транзакция.

Тази транзакция очевидно успя да приключи успешно. Но втората транзакция показва съобщение за грешка:



The screenshot shows a SQL query window titled "SQLQuery2.sql - 19...eCompany (sa (54))*". The query contains the following statements: `BEGIN TRANSACTION`, `UPDATE COUNTRIES SET NAME = 'ARGENTINA 1' WHERE COUNTRY_ID = 'AR'`, `WAITFOR DELAY '00:00:05'`, `UPDATE CUSTOMERS SET FNAME = 'JOHN' WHERE CUSTOMER_ID = 101`, and `COMMIT`. Below the query, a scroll bar is set to 110%. The Messages pane at the bottom shows three messages: "(1 row affected)", "Msg 1205, Level 13, State 51, Line 11", and "Transaction (Process ID 54) was deadlocked on lock resources with another".

```
BEGIN TRANSACTION

UPDATE COUNTRIES SET NAME = 'ARGENTINA 1' WHERE COUNTRY_ID = 'AR'

WAITFOR DELAY '00:00:05'

UPDATE CUSTOMERS SET FNAME = 'JOHN' WHERE CUSTOMER_ID = 101

COMMIT
```

110 %

Messages

(1 row affected)

Msg 1205, Level 13, State 51, Line 11

Transaction (Process ID 54) was deadlocked on lock resources with another

Фигура 6-5. Демонстрация на deadlock – стартиране на втора транзакция.

Тази транзакция беше избрана за жертва и не успя, нейните промени са отхвърлени. Така Transaction Manager компонентът разреши ситуацията.

7. Съхранени процедури и функции


7.1. В тази тема...

Ще бъдат разгледани:

- Какво е съхранена процедура;
- Как се създават и изпълняват съхранени процедури;
- Какво е функция;
- Как се създават и използват функции.

7.2. Процедури

7.2.1. Теоретични понятия

 Съхранените процедури са именувани програмни единици, съдържащи команди на T-SQL, които са компилирани и съхранени на сървъра за бази данни. Това им дава възможност за максимално бързо изпълнение и обработка на данните (същото важи и за съхранените функции).

Освен това използването им може да намали драматично мрежовия трафик между клиент и сървър, защото по мрежата се предава само командата за извикване на процедурата/функцията, евентуално връщайки резултат след изпълнението към клиента.

Опростен синтаксис за деклариране на съхранена процедура:

```
CREATE PROCEDURE procedure_name
AS
    sql_statement
GO
```

След името на процедурата може да следва списък както от входни, така и от изходни параметри, които представляват върнат от процедурата резултат.

Създадена процедура може да бъде изпълнена по следния начин:

```
EXECUTE procedure_name
или
EXEC procedure_name
```

7.2.2. Промяна и изтриване на съхранени процедури

За да бъде променена вече създадена съхранена процедура трябва да се изпълни следната команда:

```
ALTER PROCEDURE procedure_name
AS
    sql_statement
GO
```

За да бъде изтрита съхранена процедура трябва да се изпълни следната команда:

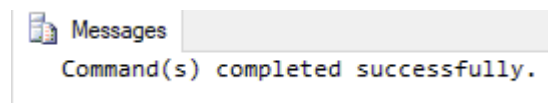
```
DROP PROCEDURE procedure_name
```

7.2.3. Примери

Пример 7-1. Да се създаде съхранена процедура, която да извежда списък с поръчките на определен клиент, както и информация за самия клиент.

```
CREATE PROCEDURE CUSTOMER_ORDERS
AS
    SELECT FNAME, LNAME, EMAIL, ADDRESS, ORDER_ID, ORDER_DATE
    FROM CUSTOMERS JOIN ORDERS ON CUSTOMERS.CUSTOMER_ID = ORDERS.CUSTOMER_ID
    WHERE CUSTOMERS.CUSTOMER_ID = 102
```

Ако командата се компилира успешно ще се получи следното съобщение:



След като процедурата е била създадена вече може да бъде изпълнена.

```
EXECUTE CUSTOMER_ORDERS
```

Получава се следният резултат:

	FNAME	LNAME	EMAIL	ADDRESS	ORDER_ID	ORDER_DATE
1	Христо	Петров	hristopetrov@gmail.com	бул. Черни връх 10, София	2357	2008-01-08 17:42:15.983
2	Христо	Петров	hristopetrov@gmail.com	бул. Черни връх 10, София	2358	2010-01-09 00:00:00.000
3	Христо	Петров	hristopetrov@gmail.com	бул. Черни връх 10, София	2397	2016-01-15 17:42:15.983
4	Христо	Петров	hristopetrov@gmail.com	бул. Черни връх 10, София	2431	2015-09-15 00:00:00.000
5	Христо	Петров	hristopetrov@gmail.com	бул. Черни връх 10, София	2432	2009-11-14 10:30:00.000

Фигура 7-1. Резултат от изпълнение на процедурата CUSTOMER_ORDERS.

Пример 7-2. Да се създаде процедура, която за подадена като входен параметър поръчка (идентификатор) извежда имена на служител, който я е обработил, както и общата ѝ стойност.

```
CREATE PROCEDURE EMPLOYEES_ORDERS @ORDER int
AS
SELECT FNAME, LNAME, O.ORDER_ID, SUM(UNIT_PRICE*QUANTITY) AS TOTAL
FROM EMPLOYEES E JOIN ORDERS O ON E.EMPLOYEE_ID = O.EMPLOYEE_ID
JOIN ORDER_ITEMS OI ON OI.ORDER_ID = O.ORDER_ID
WHERE O.ORDER_ID = 2354
GROUP BY FNAME, LNAME, O.ORDER_ID
```

Тя може да бъде изпълнена по някой от следните начини:

```
EXEC EMPLOYEES_ORDERS @ORDER = 2354
-- или
EXEC EMPLOYEES_ORDERS 2354
```

👉 Параметрите могат да бъдат предавани анонимно/позиционно като се зададат стойностите подред или могат да бъдат указани явно по име, при което редът на изброяване не е от значение.

Получава се следният резултат:

	FNAME	LNAME	ORDER_ID	TOTAL
1	Жана	Лилова	2354	1097547.00

Фигура 7-2. Резултат от изпълнение на процедурата EMPLOYEES_ORDERS.

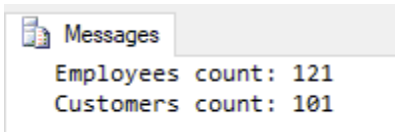
Пример 7-3. Да се създаде процедура с изходни параметри, която ще връща като параметри броя на клиентите и броя на служителите.

```
CREATE PROCEDURE GET_CUSTOMERS_EMPLOYEES_COUNT @CUST_COUNT INT OUTPUT,
@EMPS_COUNT INT OUTPUT
AS
SELECT @CUST_COUNT = COUNT(*) FROM CUSTOMERS;
SELECT @EMPS_COUNT = COUNT(*) FROM EMPLOYEES;
```

Следва изпълнение на процедурата и показване на резултата:

```
DECLARE @EMPS INT, @CUSTS INT
EXEC GET_CUSTOMERS_EMPLOYEES_COUNT @EMPS OUTPUT, @CUSTS OUTPUT
PRINT 'Employees count: ' + CAST(@EMPS AS VARCHAR)
PRINT 'Customers count: ' + CONVERT(VARCHAR, @CUSTS)
```


☞ Декларирани са две променливи, които ще бъдат предадени (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 се отпечатват брой служители и брой клиенти. Явното преобразуване се налага, защото типовете не са съвместими. Демонстрирани са две функции за явно преобразуване на типове.



```
Messages
Employees count: 121
Customers count: 101
```

Фигура 7-3. Резултат от изпълнение на процедурата GET_CUSTOMERS_EMPLOYEES_COUNT.

7.3. Функции

📖 Съхранените функции са именувани програмни единици, съдържащи команди на T-SQL, които са компилирани и съхранени на сървъра за бази данни. При извикването си те връщат стойност на мястото на извикването им, което дава възможност някои от тях (скаларните) да бъдат използвани в SELECT израз.

Поддържат се следните видове дефинирани от потребителя функции в зависимост от типа на върнатата от тях стойност – скаларна или резултатен набор.

7.3.1. Скаларни функции

Скаларните функции връщат като резултат скаларна стойност. Скаларните функции помагат да бъде опростен кода. Например, ако е нужно сложно изчисление, което се появява в много заявки, вместо да се включи формулата във всяка заявка може да бъде създадена скаларна функция, която капсулира формулата и я използва в заявките.

Опростен синтаксис за дефиниране на скаларна функция:

```
CREATE FUNCTION [schema.]function_name (parameter_list)
RETURN data_type AS
BEGIN
    statements
    RETURN value
END
```

Пример 7-4. Да се създаде функция, връщаща като скалярна стойност текст, съдържащ името на отдел (подаден като параметър) и обща стойност на заплатите в него.

```
CREATE FUNCTION DEPARTMENTS_SALARY (@DEPT_ID INT) RETURNS VARCHAR(200)
AS
BEGIN
    DECLARE @SUM_SALARY NUMERIC(10,2), @NAME VARCHAR(50)

    SELECT @NAME = D.NAME, @SUM_SALARY = SUM(E.SALARY)
    FROM DEPARTMENTS D JOIN EMPLOYEES E ON D.DEPARTMENT_ID = E.DEPARTMENT_ID
    WHERE D.DEPARTMENT_ID = @DEPT_ID
    GROUP BY D.DEPARTMENT_ID, D.NAME

    RETURN 'Общата сума на заплати в ' + @NAME + ' (ID = ' +
        CAST(@DEPT_ID AS VARCHAR) + ') е ' +
        CAST(@SUM_SALARY AS VARCHAR) + ' .'
END
```

Тази функция ще бъде демонстрирана в заявка, извличаща идентификаторите на отделите, които се подават като параметър на функцията. Забелязва се, че функцията се изпълнява по веднъж за всеки ред от резултатния набор, върнат от SELECT-а, в който се използва.

```
SELECT DBO.DEPARTMENTS_SALARY(DEPARTMENT_ID)
FROM DEPARTMENTS
```

👉 Схемата е обектът в базата данни, който притежава функцията. DBO идва от database owner – това е специфична черта за MSSQL.

Получава се следният резултат:

Results		Messages
(No column name)		
1	Общата сума на заплати в Администрация (ID = 10) е 64000.00.	
2	NULL	
3	Общата сума на заплати в Покупки (ID = 30) е 51050.00.	
4	Общата сума на заплати в Човешки ресурси (ID = 40) е 9800.00.	
5	Общата сума на заплати в Доставки (ID = 50) е 164400.00.	
6	Общата сума на заплати в ИТ (ID = 60) е 47800.00.	
7	Общата сума на заплати в Връзки с обществеността (ID = 70) е 15300.00.	
8	Общата сума на заплати в Продажби (ID = 80) е 327265.00.	
9	Общата сума на заплати в Финанси (ID = 100) е 62690.00.	
10	Общата сума на заплати в Счетоводство (ID = 110) е 24300.00.	

Фигура 7-4. Резултат от изпълнение на функцията DEPARTMENTS_SALARY.

7.3.2. Функции, връщащи резултатен набор

Тези функции връщат резултат във вид на таблица. Като следствие, те могат да бъдат използвани във FROM клаузата по същия начин, както таблиците.

Пример 7-5. Да се създаде функция, връщаща като резултат резултатен набор служителите с техните длъжности.

```
CREATE FUNCTION EMPLOYEES_JOBS () RETURNS TABLE
AS
RETURN
SELECT FNAME, LNAME, JOB_TITLE
FROM EMPLOYEES E JOIN JOBS J ON E.JOB_ID = J.JOB_ID
```

Функцията се изпълнява със следната команда:

```
SELECT * FROM DBO.EMPLOYEES_JOBS ()
ORDER BY FNAME, LNAME
```

Получава се следният резултат:

	FNAME	LNAME	JOB_TITLE
1	Александра	Бонева	Доставчик
2	Александра	Вишева	Доставчик
3	Александър	Емилов	Мениджър продажби
4	Александър	Кръстев	Търговски представител
5	Александър	Михов	Търговски представител
6	Александър	Петров	Програмист
7	Алина	Христова	Търговски представител
8	Антон	Канев	Доставчик
9	Антон	Петров	Търговски представител
10	Антон	Филипов	Стоков мениджър
11	Боряна	Димитрова	Доставчик
12	Боряна	Иванова	Програмист

Фигура 7-5. Резултат от изпълнение на функцията EMPLOYEES_JOBS.

7.3.3. Промяна и изтриване на съхранени функции

Промяна и изтриване става по вече познатия дотук начин:

```
ALTER FUNCTION function_name ...

DROP FUNCTION function_name
```

7.4. Задачи

Задача 7-1. Да се създаде процедура, която извежда всички отдели и броя служители в тях.

Задача 7-2. Да се създаде процедура с входен параметър номер на поръчка, която извежда списък на продуктите, единичните цени и количествата, участващи в тази поръчка.

Задача 7-3. Да се създаде процедура, която връща като параметри (OUTPUT) броя поръчки и общата стойност на поръчките на клиент с идентификатор, подаден като входен параметър.

Задача 7-4. Да се създаде функция, която за подадена като параметър (идентификатор) длъжност връща като скаларна стойност наименование на длъжност и средната заплата за тази длъжност.

8. Курсори

8.1. В тази тема...

Ще бъдат разгледани:

- Какво е курсор;
- Кога е удачно да се използва;
- Какви са основните етапи от жизнения цикъл на курсорите;
- Промяна на данни чрез курсор.

8.2. Теоретични понятия



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 примерната компания трябва да обработи поръчка, нормално е той да използва някое приложение, с което да позиционира в списъка с поръчки и да обработи избраната, 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.

Курсорът може да бъде представен като именуван резултатен набор, който може да се обхожда ред по ред, като във всеки момент има текущ ред, на който курсорът е позициониран. На този текущ ред може да бъдат променени определени данни или просто да бъдат извлечени, или текущо позиционирания ред да бъде изтрит. Курсорите успешно могат да бъдат използвани в тялото на тригер или процедура/функция.



Има различни типове курсори [10] в MSSQL, но ще бъдат демонстрирани само сървърните курсори на Transact-SQL, и то не във всичките им вариации.

Жизненият цикъл на курсора се състои от следните стъпки:

1. Деклариране на курсора чрез конструкцията DECLARE [11].
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 ѝ 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 курсорът.
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.

8.3. Създаване и използване на курсори

Ще бъдат използвани само някои от възможностите на курсорите с демонстративна цел, затова синтаксисът, който ще бъде представен, е възможно най-простият такъв:

```
DECLARE cursor_name CURSOR
FOR select_statement
```

Пример 8-1. Да се създаде курсор, който демонстрира прочитане на данни ред по ред от курсор. Резултатният набор за целта ще съдържа всички клиенти от Германия. Стъпките от жизнения цикъл са обозначени в коментари.

```
DECLARE @CUST_ID VARCHAR(10), @F_NAME VARCHAR(20), @L_NAME VARCHAR(20)

-- 1
DECLARE CUSTOMERS_CURSOR CURSOR FOR
SELECT CUSTOMER_ID, FNAME, LNAME
FROM CUSTOMERS
WHERE COUNTRY_ID = 'DE'

-- 2
OPEN CUSTOMERS_CURSOR
```

```

-- 3
FETCH NEXT FROM CUSTOMERS_CURSOR INTO @CUST_ID, @F_NAME, @L_NAME

PRINT 'Клиенти от Германия: '
PRINT '-----'

WHILE @@FETCH_STATUS = 0
BEGIN
-- 4
    PRINT @F_NAME + ' ' + @L_NAME + ' (ID = ' + @CUST_ID + ') '

    FETCH NEXT FROM CUSTOMERS_CURSOR INTO @CUST_ID, @F_NAME, @L_NAME
END

-- 5
CLOSE CUSTOMERS_CURSOR

-- 6
DEALLOCATE CUSTOMERS_CURSOR

```

Пример 8-2. Да се създаде курсор, който демонстрира обновяване и изтриване на ред от курсор. За целта резултатният набор ще съдържа всички записи от таблицата с професиите.

```

DECLARE @JOB_ID VARCHAR(15), @JOB_TITLE VARCHAR(25), @MIN_SAL DECIMAL

DECLARE JOBS_CURSOR CURSOR FOR
SELECT JOB_ID, JOB_TITLE, MIN_SALARY FROM JOBS

OPEN JOBS_CURSOR
FETCH NEXT FROM JOBS_CURSOR INTO @JOB_ID, @JOB_TITLE, @MIN_SAL

SET NOCOUNT ON -- спираме автоматичното изписване на броя намерени редове

WHILE @@FETCH_STATUS = 0
BEGIN
    IF (@JOB_ID = 'IT_PROG')
    BEGIN
        PRINT 'Минималната заплата за ' + @JOB_TITLE +
            ' ще бъде променена.'
        UPDATE JOBS SET MIN_SALARY = 5000
        WHERE CURRENT OF JOBS_CURSOR
    END

    IF (@JOB_ID = 'MK_REP')
    BEGIN

```

```

        PRINT @JOB_TITLE + ' ще бъде изтрита!'
        DELETE FROM JOBS WHERE CURRENT OF JOBS_CURSOR
    END

    FETCH NEXT FROM JOBS_CURSOR INTO @JOB_ID, @JOB_TITLE, @MIN_SAL
END

CLOSE JOBS_CURSOR
DEALLOCATE JOBS_CURSOR

SET NOCOUNT OFF

```

От кода се вижда, че при промяна или изтриване на данни чрез курсор единствената разлика от познатите досега команди за тези операции е в WHERE клаузата. При работата с курсори изразът CURRENT OF cursor_name определя това текущият ред на курсора да бъде засегнат от съответната команда за обновяване или изтриване. Използваната опция SET NOCOUNT ON (OFF) забранява (позволява) извеждането на броя засегнати редове автоматично – тук се използва само за прегледност на резултатите.

8.4. Задачи

Задача 8-1. Да се създаде курсор, който съдържа име, фамилия, телефон и заплата на всички служители със заплата от 10000 до 13000. Да се обходи курсорът, като се извеждат имената, телефона и заплата на текущия служител, след което ако заплата му е под 12000 да се увеличи с 5%.


9. Тригери

9.1. В тази тема...

Ще бъдат разгледани:

- Какво е тригер;
- Кога е удачно да се използва;
- Какви видове тригери съществуват;
- Кой са събитията, при които те се стартират.


9.2. Теоретични понятия

 Тригерът е именувана програмна единица, която се стартира при настъпване на определено събитие в базата данни и за разлика от процедурите и функциите не може да бъде извиквана по име.

Събитията, които могат да задействат тригер, са различни видове:

- Включване/изключване на потребител в базата;
- Промяна на данни с DDL команди;
- Промяна на обекти с DML команди, изпълнение на операции GRANT И REVOKE, и др.

Един и същ тригер може да бъде „прикачен“ към няколко събития, като обикновено има механизми, с които в неговия код (тялото му) може да се провери кое е събитието, което го е задействало, и така има възможност бизнес-логиката да бъде разделена.

 Различните бази от данни могат да поддържат различни видове тригери и тригериращи събития, както и моменти на изпълнение – преди, след или вместо тригериращото събитие. Тук отново консултация с документацията на използваната конкретна реализация няма да навреди [12, 13, 14].

Ще бъдат демонстрирани тригери, които се изпълняват при DML команди. За целта ще бъде използван следният опростен синтаксис:

```
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 моментът за изпълнение на тригера (FOR и AFTER са синоними). След това се изброяват събитията, при настъпване на които да се задейства тригера – валидни типове от DML командите са INSERT, UPDATE, DELETE.

👉 В Oracle и MySQL могат да бъдат създадени тригери, които да се изпълняват както преди, така и след тригериращото събитие, докато в MSSQL са налични само вторите.

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

9.3. Изтриване, промяна и забраняване

Следващите команди служат за управление на тригери в MSSQL.

👉 Разбира се, в други реализации тези команди варират.

9.3.1. Изтриване

За да бъде изтрит тригер трябва да се изпълни следната команда:

```
DROP TRIGGER trigger_name
```

👉 Ако се изтрие таблица или изглед, то всички свързани тригери също се изтриват.

9.3.2. Промяна

За промяна на тригер може просто да бъде изтрит и създаден отново. Но ако трябва да бъде прескочена операцията с изтриването, командата е следната:

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

9.3.3. Забраняване

За да бъде забранено изпълнението на тригер за конкретна таблица, командата е следната:

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

За да бъде позволено отново изпълнението на тригер, трябва да се изпълни следната команда:

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

☞ За да бъдат забранени или разрешени всички тригери на таблицата вместо името на тригера трябва да се използва ключовата дума ALL.

9.4. Създаване на тригери

📖 В MSSQL DML тригерите имат на разположение две специални таблици: **inserted** и **deleted**.

📖 Таблицата **inserted** съдържа копия на редовете, които са били добавени/променени в целевата таблица от изпълнения INSERT/UPDATE, който е задействал тригера.

📖 Таблицата **deleted** съдържа копие на редовете, които са били изтрети/променени в целевата таблица от изпълнения DELETE/UPDATE, който е задействал тригера.

📖 При команда UPDATE **inserted** ще съдържа копия на променените редове след промяната, а **deleted** ще съдържа копия на променените редове преди промяната!

Тези таблици често се използват за:

- Разширяване на възможностите за валидиране на данните при добавяне/промяна/изтриване;
- Добавяне/промяна на данни в таблици, когато тригерът е създаден върху изглед;
- И др.

Пример 9-1. Да се създаде тригер, който ще се изпълнява след добавяне, промяна и изтриване на записи в таблицата PRODUCTS и ще извежда всички записи от inserted и deleted, за да се види какво реално те ще съдържат след всяка команда.

```
CREATE TRIGGER TRG_INSERT_PRODUCTS
ON PRODUCTS
AFTER INSERT, UPDATE, DELETE
AS

PRINT 'inserted records are:'
SELECT * FROM INSERTED

PRINT 'deleted records are:'
SELECT * FROM DELETED
```

След създаването тригерът може да бъде тестван с добавяне на един нов запис (за прегледност на резултата може да се превключи опцията на резултатите на „Results To Text“ (Ctrl+T), както и да се премахне автоматичното показване на броя засегнати редове с командата "SET NOCOUNT ON"):

```
SET NOCOUNT ON
INSERT INTO PRODUCTS (PRODUCT_ID, NAME, DESCR, PRICE)
VALUES (1, 'Мобилен телефон Samsung A40', 'Ергономичен корпус...', 380);
```

Резултатите ще покажат наличие на копие на новия ред в inserted таблицата и празна таблица deleted:

Results		
inserted records are:		
PRODUCT_ID	NAME	PRICE
1	Мобилен телефон Samsung A40	380.00
deleted records are:		
PRODUCT_ID	NAME	PRICE

Фигура 9-1. Копие на новия ред в inserted таблицата и празна таблица deleted.

Сега ще бъдат добавени няколко реда с една команда, като за целта ще бъдат използвани данни пак от същата таблица PRODUCTS, но ще леко модифицирани, за да не се дублират със съществуващите:

```
INSERT INTO PRODUCTS (PRODUCT_ID, NAME, PRICE)
SELECT PRODUCT_ID + 10000, 'NEW ' + NAME, PRICE*1.2
FROM PRODUCTS
WHERE product_id BETWEEN 2400 AND 2405
```

Резултатите изглеждат така:

Results		
inserted records are:		
PRODUCT_ID	NAME	PRICE
12404	NEW Часовник Smartwatch Garmin Fenix 5	985.20
12403	NEW Безжични слушалки AirPods	56.40
12402	NEW Слушалка Bluetooth Plantronics Explorer 500	128.40
12400	NEW Монитор Gaming LED IPS Lenovo 27	417.60
deleted records are:		
PRODUCT_ID	NAME	PRICE

Фигура 9-2. Актуализирани редове в inserted таблицата.

Вижда се, че SELECT операторът е върнал 4 реда, които бяха добавени в PRODUCTS и съответно техни копия тригерът намери в inserted.

Сега ще бъдат изтрети новодобавените редове, за да се види, че техни копия ще бъдат налични за тригера в deleted:

```
DELETE FROM PRODUCTS
WHERE PRODUCT_ID > 10000
```

И резултатът е:

Results		
inserted records are:		
PRODUCT_ID	NAME	PRICE

deleted records are:		
PRODUCT_ID	NAME	PRICE

12404	NEW Часовник Smartwatch Garmin Fenix 5	985.20
12403	NEW Безжични слушалки AirPods	56.40
12402	NEW Слушалка Bluetooth Plantronics Explorer 500	128.40
12400	NEW Монитор Gaming LED IPS Lenovo 27	417.60

Фигура 9-3. Редове в deleted таблицата.

Сега ще бъде демонстрирана смислена бизнес логика в тригер. В следващия пример тригер ще проверява при добавяне на запис или промяна на данни на служител (таблица EMPLOYEES) дали не е променена заплатата му и ако е така, ще проверява дали новата заплата е в границите за заемана длъжност, които се съхраняват в таблицата JOBS. За целта тригерът трябва да сработва при събитията INSERT и UPDATE в таблицата EMPLOYEE, правейки описаните проверки.

Пример 9-2. Да се създаде тригер, изпълняващ се при добавяне на запис или промяна на данни на служител (таблица EMPLOYEES).

```
CREATE TRIGGER TRG_IU_EMP
ON EMPLOYEES
AFTER INSERT, UPDATE
AS
-- Само за информация показваме кои са редовете, които биват променени
PRINT 'Променени редове:'
SELECT I.FNAME + ' ' + I.LNAME AS NAME, I.SALARY, J.MIN_SALARY,
       J.MAX_SALARY
FROM INSERTED I, JOBS J
WHERE J.JOB_ID = I.JOB_ID;
```

```

-- Проверка дали стойността в колоната SALARY е била променена
IF UPDATE(SALARY)
BEGIN
    -- Декларираме курсор, чрез който ще обхождаме всички променени редове
    -- от inserted и за всеки от тях ще проверяваме границите на заплатата
    DECLARE UPD_CURSOR CURSOR LOCAL FOR
        SELECT I.FNAME + ' ' + I.LNAME, I.SALARY, J.MIN_SALARY, J.MAX_SALARY
        FROM INSERTED I, JOBS J
        WHERE J.JOB_ID = I.JOB_ID;
    -- тук направихме вътрешно съединение, за да имаме налични в курсора
    -- освен новата стойност на заплатата и нейните граници за съответната
    -- длъжност от JOBS

    -- Декларираме променливи, в които ще съхраняваме стойностите на полетата
    -- за всеки ред от курсора при обхождането му
    DECLARE @NAME VARCHAR(50), @NEW_SAL DECIMAL, @MIN_SAL DECIMAL,
            @MAX_SAL DECIMAL;

    -- Следва обхождане на курсора и проверка на всяка заплата дали е в
    -- определения диапазон в JOBS
    OPEN UPD_CURSOR;
    FETCH NEXT FROM UPD_CURSOR INTO @NAME, @NEW_SAL, @MIN_SAL, @MAX_SAL;

    WHILE @@FETCH_STATUS = 0
    BEGIN
        -- Проверка дали новата заплата е между границите
        IF (@NEW_SAL NOT BETWEEN @MIN_SAL AND @MAX_SAL)
        BEGIN
            -- При първия намерен ред със заплата извън диапазона генерираме
            -- грешка с подходящо съобщение, затваряме курсора и отхвърляме
            -- направените промени и приключваме обработката
            DECLARE @MSG VARCHAR(300);
            SET @MSG = 'Невалидна нова заплата за ' + @NAME + ': ' +
                CAST(@NEW_SAL AS VARCHAR) + '! Трябва да е между ' +
                CAST(@MIN_SAL AS VARCHAR) + ' и ' +
                CAST(@MAX_SAL AS VARCHAR);

            CLOSE UPD_CURSOR;
            DEALLOCATE UPD_CURSOR;
            RAISERROR(@MSG, 16, 1);
            ROLLBACK;
            RETURN;
        END;

        FETCH NEXT FROM UPD_CURSOR INTO @NAME, @NEW_SAL, @MIN_SAL, @MAX_SAL;
    END;

```

```

CLOSE UPD_CURSOR;
DEALLOCATE UPD_CURSOR;

END;

```

За да се тества действието на тригера ще бъде направен опит за промяна на заплатите на трима служителя със стойност, която е извън границите за техните длъжности (отново резултатите ще бъдат показани в текстов вид с Ctrl+T):

```

UPDATE EMPLOYEES
SET SALARY = 15000
WHERE EMPLOYEE_ID IN (100, 110, 120)

```

Резултатът е следният:

Results			
Променени редове:			
name	salary	min_salary	max_salary
Момчил Димитров	15000.00	7500	13500
Георги Георгиев	15000.00	7200	12000
Иван Иванов	15000.00	25000	50500

Msg 50000, Level 16, State 1, Procedure TRG_IU_EMP, Line 42 (Batch Start Line 0)
 Невалидна нова заплата за Момчил Димитров: 15000! Трябва да е между 7500 и 13500
 Msg 3609, Level 16, State 1, Line 2
 The transaction ended in the trigger. The batch has been aborted.

Фигура 9-4. Генериране на грешка в тригер.

Вижда се, че тригерът спира обработка още при първия намерен невалиден запис.

9.5. Задачи

Задача 9-1. Да се създаде тригер, който при всяка промяна на фамилия на клиент записва ред в нова таблица CUSTOMERS_HIST с атрибути:

- идентификатор на клиент;
- стара фамилия;
- нова фамилия.

10. Приложение 1 – решения на задачите

Език SQL

Задача 2-1. Да се създаде база от данни с име TESTDB.

```
CREATE DATABASE TESTDB
```

Задача 2-2. Да се създаде таблица в тази база данни с име COUNTRIES и следните колони:

- country_code – с тип char(3);
- name – тип varchar(40);
- population – тип int.

```
CREATE TABLE COUNTRIES  
(  
    COUNTRY_CODE CHAR(3),  
    NAME VARCHAR(40),  
    POPULATION INT  
)
```

Задача 2-3. Да се добави нов атрибут в таблицата COUNTRIES с име phone_code и тип цяло число до 3 цифри.

```
ALTER TABLE COUNTRIES  
ADD PHONE_CODE NUMERIC(3,0)
```

Задача 2-4. Да се добави ред в таблицата COUNTRIES със следните данни:

- country_code: BGR;
- name: България;
- population: 7500000;
- phone_code: 359.

```
INSERT INTO COUNTRIES (COUNTRY_CODE, NAME, POPULATION, PHONE_CODE)  
VALUES ('BG', 'България', 7500000, 359)
```


Задача 2-5. Да се промени населението на България на 6 милиона.

```
UPDATE COUNTRIES  
SET POPULATION = 6000000  
WHERE COUNTRY_CODE = 'BG'
```

Задача 2-6. Да се изтрият всички редове в таблицата COUNTRIES.

```
DELETE FROM COUNTRIES
```

Задача 2-7. Да се изтрие таблицата COUNTRIES от базата данни.

```
DROP TABLE COUNTRIES
```

Задача 2-8. Да се изтрие базата данни TESTDB.

```
DROP DATABASE TESTDB
```

Създаване на база данни за търговска фирма

Задача 3-1. Да се изтрият всички редове от всички таблици в базата от данни, след което да се въведат данните чрез командите от inserts_mssql.sql файла.

👉 Най-лесно е да се започне по ред, обратен на този на въвеждането – от последната таблица към първата. По този начин няма да се получават грешки, че даден ред е референциран от външен ключ.

```
DELETE FROM ORDER_ITEMS;  
DELETE FROM ORDERS;  
DELETE FROM CUSTOMERS;  
DELETE FROM PRODUCTS;  
DELETE FROM EMPLOYEES;  
DELETE FROM JOBS;  
DELETE FROM DEPARTMENTS;  
DELETE FROM COUNTRIES;  
DELETE FROM REGIONS;
```

Въвеждането на данните от [файла](#) може да стане с copy/paste и изпълнението им в базата от данни.

Задача 3-2. Да се увеличи количеството с 2 броя и да се намали единичната цена с 5% на продукт с идентификатор 2254 в поръчка с идентификатор 2354.

```
UPDATE ORDER_ITEMS
SET UNIT_PRICE *= 0.95, QUANTITY += 2
WHERE ORDER_ID = 2354
AND PRODUCT_ID = 2254
```

Задача 3-3. Да се изтрие служител с идентификатор 183.

```
DELETE EMPLOYEES
WHERE EMPLOYEE_ID = 183
```

👉 Както се вижда от кода, T-SQL позволява пропускане на ключовата дума FROM, което обаче не е така в Oracle и MySQL, където тя е задължителна.

Извличане на данни

Задача 4-1. Да се изведат имената, заплатите и идентификаторите на длъжностите на служителите, работещи в отдели 50 и 80. Резултатът да е подреден по фамилия на служител във възходящ ред.

```
SELECT FNAME, LNAME, SALARY, JOB_ID
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 50 OR DEPARTMENT_ID = 80
ORDER BY LNAME ASC
```

Задача 4-2. Да се изведат общата сума на заплатите и броя служители в отдел 60.

```
SELECT SUM(SALARY), COUNT(EMPLOYEE_ID)
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 60
```

Задача 4-3. За всички поръчки да се изведат идентификатор на поръчка и обща стойност на поръчката. Резултатът да е подреден по стойност на поръчката в низходящ ред.

```
SELECT ORDER_ID, SUM(UNIT_PRICE*QUANTITY) AS TOTAL
FROM ORDER_ITEMS
GROUP BY ORDER_ID
ORDER BY TOTAL DESC
```

Задача 4-4. Да се изведат всички малки имена на клиенти и служители с евентуалните повторения, сортирани в низходящ ред по име.

```
SELECT FNAME FROM CUSTOMERS
UNION ALL
SELECT FNAME FROM EMPLOYEES
ORDER BY FNAME DESC
```

Задача 4-5. Да се изведат име и фамилия на клиенти и служители без повторения, а като трета колона за клиентите да се използва израз, генериращ низа „Клиент (<идентификатор>)“, за служителите – „Служител (<идентификатор>)“.

```
SELECT FNAME, LNAME, 'Клиент (' + COUNTRY_ID + ') '
FROM CUSTOMERS
UNION
SELECT FNAME, LNAME, 'Служител (' + CAST(DEPARTMENT_ID AS VARCHAR) + ') '
FROM EMPLOYEES
```

Задача 4-6. Да се изведат общите собствени имена на клиенти и служители.

```
SELECT FNAME
FROM CUSTOMERS
INTERSECT
SELECT FNAME
FROM EMPLOYEES
```

Задача 4-7. Да се изведат собствените имена на клиенти, които не се срещат сред тези на служителите.

```
SELECT FNAME
FROM CUSTOMERS
EXCEPT
SELECT FNAME
FROM EMPLOYEES
```

Задача 4-8. Да се изведат идентификаторите и датите на поръчките, както и имената на служителите, които са ги обработили.

```
SELECT ORDER_ID, ORDER_DATE, FNAME, LNAME
FROM ORDERS O JOIN EMPLOYEES E ON E.EMPLOYEE_ID = O.EMPLOYEE_ID
```

Задача 4-9. Да се изведат имената на всички клиенти и идентификаторите на поръчките им. В резултатния набор да участват и клиентите, които все още не са правили поръчки.

```
SELECT FNAME, LNAME, ORDER_ID
FROM ORDERS O RIGHT JOIN CUSTOMERS C
ON C.CUSTOMER_ID = O.CUSTOMER_ID
```

Задача 4-10. Да се изведат имената на продуктите, които не са поръчвани до момента.

```
SELECT NAME
FROM PRODUCTS
WHERE NOT EXISTS (SELECT 1
                  FROM ORDER_ITEMS
                  WHERE ORDER_ITEMS.PRODUCT_ID = PRODUCTS.PRODUCT_ID)
```

Задача 4-11. Да се изведат имената на всички клиенти, които са от държави в регион „Западна Европа“.

```
SELECT C.FNAME, C.LNAME, S.NAME, R.NAME
FROM CUSTOMERS C, COUNTRIES S, REGIONS R
WHERE C.COUNTRY_ID = S.COUNTRY_ID
AND S.REGION_ID = R.REGION_ID
AND R.NAME = 'Западна Европа'
```

Задача 4-12. Да се изведат вторите 10 най-добре платени служители (подредени по заплата низходящо).

```
SELECT EMPLOYEE_ID, FNAME, LNAME, SALARY
FROM EMPLOYEES
ORDER BY SALARY DESC
OFFSET 9 ROWS -- прескачаме първите 9
FETCH NEXT 10 ROWS ONLY -- вземаме следващите 10 от 10-я включително
```

Задача 4-13. Да се изведат име, фамилия и пол на клиентите, направили последните 5 поръчки.

```
SELECT C.FNAME, C.LNAME, C.GENDER, O.ORDER_DATE
FROM ORDERS O, CUSTOMERS C
WHERE O.CUSTOMER_ID = C.CUSTOMER_ID
ORDER BY ORDER_DATE DESC
```

```
OFFSET 0 ROWS  
FETCH NEXT 5 ROWS ONLY
```

Изгледи

Задача 5-1. Да се създаде изглед, който съдържа имената на продуктите и общо поръчано количество от продукт.

```
CREATE VIEW PRODUCT_QUANTITY  
AS  
SELECT NAME, SUM(QUANTITY) AS SUM_QUANTITY  
FROM PRODUCTS P JOIN ORDER_ITEMS OI  
ON P.PRODUCT_ID = OI.PRODUCT_ID  
GROUP BY P.PRODUCT_ID, NAME
```

Задача 5-2. Да се създаде изглед, който съдържа десетимата клиенти с най-голям брой поръчки. Ако последният клиент има равен брой поръчки с други клиенти, те също да участват в изгледа.

```
CREATE VIEW CUST_ORDER_NUM  
AS  
SELECT TOP 10 WITH TIES FNAME + ' ' + LNAME AS CUSTOMER, COUNT(ORDER_ID) AS  
ORDER_NUM  
FROM CUSTOMERS C JOIN ORDERS O  
ON C.CUSTOMER_ID = O.CUSTOMER_ID  
GROUP BY C.CUSTOMER_ID, FNAME + ' ' + LNAME  
ORDER BY ORDER_NUM DESC
```

Транзакции

Задача 6-1. Да се направи транзакция транзакция, която има за цел да изтрие отдел „Мениджмънт“, като преди това прехвърли всички служители от него в отдел „Администрация“.

```
BEGIN TRAN  
  
UPDATE EMPLOYEES  
SET DEPARTMENT_ID = 10  
WHERE DEPARTMENT_ID = 90;
```

```
DELETE DEPARTMENTS
WHERE DEPARTMENT_ID = 90;

COMMIT TRAN
```

Задача 6-2. Да се направи транзакция транзакция, която изтрива продукт с идентификатор 1726, като преди това го изтрива от всички поръчки, в които участва, като накрая отхвърля направените промени.

```
BEGIN TRAN

DELETE ORDER_ITEMS
WHERE PRODUCT_ID = 1726;

DELETE PRODUCTS
WHERE PRODUCT_ID = 1726;

ROLLBACK TRAN
```

Съхранени процедури и функции

Задача 7-1. Да се създаде процедура, която извежда всички отдели и броя служители в тях.

```
CREATE PROCEDURE DEPT_EMPLOYEES
AS
    SELECT NAME, COUNT(EMPLOYEE_ID) EMP_COUNT
    FROM DEPARTMENTS D LEFT JOIN EMPLOYEES E
        ON E.DEPARTMENT_ID = D.DEPARTMENT_ID
    GROUP BY D.DEPARTMENT_ID, NAME
```

Изпълнение:

```
EXEC DEPT_EMPLOYEES
```

Задача 7-2. Да се създаде процедура с входен параметър номер на поръчка, която извежда списък на продуктите, единичните цени и количествата, участващи в тази поръчка.

```
CREATE PROCEDURE ORD_PRODUCTS @ORD_ID INT
AS
    SELECT P.NAME, OI.UNIT_PRICE, OI.QUANTITY
    FROM ORDERS O JOIN ORDER_ITEMS OI ON O.ORDER_ID = OI.ORDER_ID
```

```

        JOIN PRODUCTS P ON P.PRODUCT_ID = OI.PRODUCT_ID
WHERE O.ORDER_ID = @ORD_ID

```

Изпълнение:

```
EXEC ORD_PRODUCTS @ORD_ID = 2354
```

Задача 7-3. Да се създаде процедура, която връща като параметри (OUTPUT) броя поръчки и общата стойност на поръчките на клиент с идентификатор, подаден като входен параметър.

```

CREATE PROCEDURE GET_CUST_TOTAL
    @CUSTID INT, @ORD_COUNT INT OUTPUT, @TOTAL NUMERIC(12,2) OUTPUT
AS
    SELECT @ORD_COUNT = COUNT(ORDER_ID) FROM ORDERS WHERE CUSTOMER_ID = @CUSTID
    SELECT @TOTAL = SUM(UNIT_PRICE*QUANTITY) FROM ORDERS O JOIN ORDER_ITEMS OI
ON OI.ORDER_ID = O.ORDER_ID WHERE CUSTOMER_ID = @CUSTID

```

Изпълнение:

```

DECLARE @ORD_COUNT INT, @TOTAL NUMERIC(12,2)
EXEC GET_CUST_TOTAL 101, @ORD_COUNT OUTPUT, @TOTAL OUTPUT
SELECT @ORD_COUNT AS PRICE, @TOTAL

```

Задача 7-4. Да се създаде функция, която за подадена като параметър (идентификатор) длъжност връща като скаларна стойност наименование на длъжност и средната заплата за тази длъжност.

```

CREATE FUNCTION JOBS_AVG_SALARY(@JOB_ID VARCHAR(10)) RETURNS VARCHAR(200)
AS
BEGIN
    DECLARE @AVG_SALARY NUMERIC(10,2), @JOB_TITLE VARCHAR(50)

    SELECT @JOB_TITLE = ISNULL(JOB_TITLE, ''),
           @AVG_SALARY = AVG(SALARY)
    FROM JOBS J LEFT JOIN EMPLOYEES E ON J.JOB_ID = E.JOB_ID
    WHERE J.JOB_ID = @JOB_ID
    GROUP BY J.JOB_ID, JOB_TITLE

    RETURN 'Средната заплата за длъжност ' + @JOB_TITLE + ' (ID = ' +
           CAST(@JOB_ID AS VARCHAR) + ') е ' +
           CAST(@AVG_SALARY AS VARCHAR) + '.'
END

```

Изпълнение:

```
SELECT DBO.JOBS_AVG_SALARY(JOB_ID)
FROM JOBS
```

Курсори

Задача 8-1. Да се създаде курсор, който съдържа име, фамилия, телефон и заплата на всички служители със заплата от 10000 до 13000. Да се обходи курсорът, като се извеждат имената, телефона и заплата на текущия служител, след което ако заплата му е под 12000 да се увеличи с 5%.

```
DECLARE @FNAME VARCHAR(20), @LNAME VARCHAR(25), @PHONE VARCHAR(20), @SALARY
DECIMAL

DECLARE EMP_CURSOR CURSOR FOR
SELECT FNAME, LNAME, PHONE, SALARY FROM EMPLOYEES
WHERE SALARY BETWEEN 10000 AND 13000

OPEN EMP_CURSOR
FETCH NEXT FROM EMP_CURSOR INTO @FNAME, @LNAME, @PHONE, @SALARY

SET NOCOUNT ON -- спираме автоматичното изписване на броя намерени редове

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT @FNAME + ' ' + @LNAME + ' (' + @PHONE + ') - ' +
        CAST(@SALARY AS VARCHAR)

    IF (@SALARY < 12000)
    BEGIN
        PRINT 'Заплата на ' + @FNAME + ' ' + @LNAME + ' ще стане от ' +
            CAST(@SALARY AS VARCHAR) + ' на ' + CAST(@SALARY*1.05 AS VARCHAR)
        UPDATE EMPLOYEES
        SET SALARY = SALARY*1.05
        WHERE CURRENT OF EMP_CURSOR
    END

    FETCH NEXT FROM EMP_CURSOR INTO @FNAME, @LNAME, @PHONE, @SALARY
END

CLOSE EMP_CURSOR
DEALLOCATE EMP_CURSOR
SET NOCOUNT OFF
```


Тригери

Задача 9-1. Да се създаде тригер, който при всяка промяна на фамилия на клиент записва ред в нова таблица CUSTOMERS_HIST с атрибути:

- идентификатор на клиент;
- стара фамилия;
- нова фамилия.

Нека първо бъде създадена новата таблица с посочените атрибути:

```
CREATE TABLE CUSTOMERS_HISTORY
(
    CUSTOMER_ID INT,
    OLD_LNAME VARCHAR(20),
    NEW_LNAME VARCHAR(20)
)
```

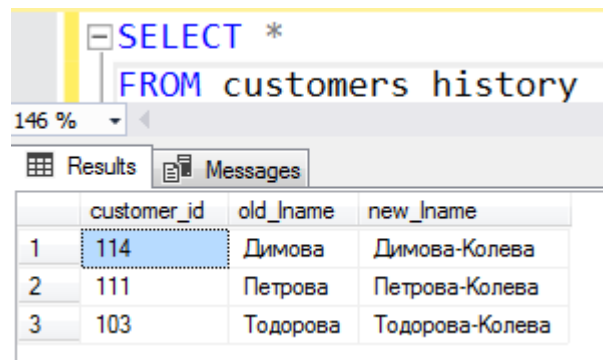
Сега може да бъде създаден тригерът:

```
CREATE TRIGGER TRG_CUSTOMERS_HIST
ON CUSTOMERS
FOR UPDATE
AS
    IF UPDATE(LNAME)
    BEGIN
        INSERT INTO CUSTOMERS_HISTORY(CUSTOMER_ID, OLD_LNAME, NEW_LNAME)
        SELECT I.CUSTOMER_ID, D.LNAME, I.LNAME
        FROM INSERTED I, DELETED D
        WHERE I.CUSTOMER_ID = D.CUSTOMER_ID
    END
```

За да се провери неговото действие ще бъде направена промяна на фамилиите на трима клиенти:

```
UPDATE CUSTOMERS
SET LNAME = LNAME + '-' + 'Колева'
WHERE CUSTOMER_ID IN (103, 111, 114)
```

Тригерът е записал резултата, показан на Фигура 10-1.



SQL Query: `SELECT * FROM customers history`

Zoom: 146 %

Results | Messages

	customer_id	old_name	new_name
1	114	Димова	Димова-Колева
2	111	Петрова	Петрова-Колева
3	103	Тодорова	Тодорова-Колева

Фигура 10-1. Резултат от изпълнението на тригера TRG_CUSTOMERS_HIST.

11. Приложение 2 – ресурси

За създаване на описаните таблици и попълването им с данни могат да бъдат използвани следните файлове:

- За създаване на таблиците:
 - За MS SQL: [create db mssql.sql](#)
 - За Oracle: [create db oracle.sql](#)
 - За MySQL: [create db mysql.sql](#)
- За попълване на таблиците с данни:
 - За MS SQL: [inserts mssql.sql](#)
 - За Oracle: [inserts oracle.sql](#)
 - За MySQL: [inserts mysql.sql](#)

ИЗПОЛЗВАНИ ИЗТОЧНИЦИ

- [1] Microsoft, „MERGE (Transact-SQL),“ [Онлайн]. Available: <https://docs.microsoft.com/en-us/sql/t-sql/statements/merge-transact-sql?view=sql-server-2017>. [Отваряно на 2019].
- [2] Microsoft, „Transact-SQL Reference (Database Engine),“ [Онлайн]. Available: <https://docs.microsoft.com/en-us/sql/t-sql/language-reference?view=sql-server-2017>. [Отваряно на 2019].
- [3] Microsoft, „Database Identifiers,“ [Онлайн]. Available: <https://docs.microsoft.com/en-us/sql/relational-databases/databases/database-identifiers?view=sql-server-2017>. [Отваряно на 2019].
- [4] Microsoft, „Aggregate Functions (Transact-SQL),“ [Онлайн]. Available: <https://docs.microsoft.com/en-us/sql/t-sql/functions/aggregate-functions-transact-sql?view=sql-server-2017>.
- [5] Wikipedia, „ISO 3166-1 alpha-2,“ [Онлайн]. Available: https://en.wikipedia.org/wiki/ISO_3166-1_alpha-2. [Отваряно на 2019].
- [6] Oracle, „The UNION [ALL], INTERSECT, MINUS Operators,“ [Онлайн]. Available: https://docs.oracle.com/cd/B19306_01/server.102/b14200/queries004.htm. [Отваряно на 2019].
- [7] Microsoft, „Set Operators - UNION (Transact-SQL),“ [Онлайн]. Available: <https://docs.microsoft.com/en-us/sql/t-sql/language-elements/set-operators-union-transact-sql?view=sql-server-2017>. [Отваряно на 2019].
- [8] Microsoft, „Joins (SQL Server),“ [Онлайн]. Available: <https://docs.microsoft.com/en-us/sql/relational-databases/performance/joins?view=sql-server-2017>. [Отваряно на 2019].
- [9] SQLServerTutorial.net, „SQL Server OFFSET FETCH,“ [Онлайн]. Available: <http://www.sqlservertutorial.net/sql-server-basics/sql-server-offset-fetch/>. [Отваряно на 2019].

- [10] Microsoft, „Cursors,“ [Онлайн]. Available: <https://docs.microsoft.com/en-us/sql/relational-databases/cursors?view=sql-server-2017>. [Отваряно на 2019].
- [11] Microsoft, „DECLARE CURSOR (Transact-SQL),“ [Онлайн]. Available: <https://docs.microsoft.com/en-us/sql/t-sql/language-elements/declare-cursor-transact-sql?view=sql-server-2017>. [Отваряно на 2019].
- [12] Microsoft, „CREATE TRIGGER (Transact-SQL),“ [Онлайн]. Available: <https://docs.microsoft.com/en-us/sql/t-sql/statements/create-trigger-transact-sql?view=sql-server-2017>. [Отваряно на 2019].
- [13] Oracle, „MySQL 8.0 Reference Manual /.../ 13.1.22 CREATE TRIGGER Syntax,“ [Онлайн]. Available: <https://dev.mysql.com/doc/refman/8.0/en/triggers.html>. [Отваряно на 2019].
- [14] Oracle, „Using Triggers,“ [Онлайн]. Available: https://docs.oracle.com/database/121/TDDDG/tdddg_triggers.htm#TDDDG50000. [Отваряно на 2019].
- [15] C. C. a. S. Morris, Database Systems: Design, Implementation, and Management, 12th Edition, Cengage Learning, 2017.
- [16] S. Ardeleanu, Relational Database Programming - A Set-Oriented Approach, apress, 2016.
- [17] S. G. Elvis C. Foster, Database Systems - A Pragmatic Approach, Second Edition, apress, 2016.
- [18] R. E. Sikha Bagui, Database Design Using Entity-Relationship Diagrams - Second Edition, CRC Press, 2012.
- [19] S. V. Louis Davidson, Exam Ref 70-762 Developing SQL Databases, Pearson Education, Inc., 2017.
- [20] C. J. Date, SQL and Relational Theory - How to Write Accurate SQL Code, THIRD EDITION, O'Reilly Media, Inc., 2015.