

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/303920226>

Практикум по бази от данни – I част

Book · January 2012

CITATIONS

0

READS

13,710

1 author:



[Tsvetanka Georgieva-Trifonova](#)

University of Veliko Tarnovo

49 PUBLICATIONS 65 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:



Model for representing text documents based on vector space model and association rules between terms [View project](#)



Customer Feedback Text Analysis for Online Stores Reviews in Bulgarian [View project](#)

Цветанка Георгиева-Трифорова

ПРАКТИКУМ ПО
БАЗИ ОТ ДАННИ

ЧАСТ I

Велико Търново
2012

Настоящият практикум по бази от данни е разработен в съответствие с учебните планове на специалностите „Математика и информатика”, „Информатика” и „Компютърни науки” във Великотърновския университет „Св. св. Кирил и Методий”. Той е предназначен за студентите от редовна форма на обучение, както и за обучаващите се в магистърските програми по информатика. Практикумът може да бъде полезен и на студентите от други специалности и други висши учебни заведения, които изучават учебни дисциплини, свързани с бази от данни.

Основните разгледани теми в настоящата книга са свързани с проектиране на релационни бази от данни; използване на езика SQL за дефиниране, извличане и манипулиране на данни; анализиране и оптимизиране на достъпа до данните. Примерните SQL конструкции са тествани в средата на Microsoft SQL Server 2008. Всички включени в темите примери могат да бъдат изтеглени от адрес <http://practicum.host22.com>.

Предлаганият материал е обсъден и утвърден за печат на заседание на катедра „Информационни технологии” на Великотърновския университет „Св. св. Кирил и Методий”.

Второ преработено и допълнено издание

Рецензенти: доц. д-р Владимир Димитров, доц. д-р Емилия Тодорова

© Цветанка Любомирова Георгиева-Трифенова, автор, 2012

Съдържание

Предговор	5
Глава I Проектиране на бази от данни	7
Тема 1 Проектиране на релационни бази от данни. Нормализация на данните	7
Проектиране на релационни бази от данни	7
Таблицы и уникалност	7
Външни ключове и домейни	8
Релационни връзки	8
Нормализация на данните	10
Първа нормална форма	11
Втора нормална форма	12
Трета нормална форма	12
Допълнителна нормализация	14
Правила за запазване на целостността на данните	17
Денормализация на данните	18
Тема 2 Създаване и управление на бази от данни на SQL Server	23
Системни бази от данни	23
Файлове на базите от данни	24
Създаване на база от данни	24
Файлови групи за база от данни	27
Променяне на базата от данни	28
Изтриване на база от данни	31
Преместване на база от данни на SQL Server	31
Архивиране и възстановяване на база от данни	33
Опции на база от данни на SQL Server	37
Разширяване на база от данни	41
Свиване на база от данни	42
Преименуване на база от данни	45
Тема 3 Използване на SQL за дефиниране на данни	47
Типове данни	47
Създаване на таблица	52
Променяне на вече създадена таблица	54
Изтриване на таблица	55
Тема 4 Ограничаване на стойностите на данните	57
Добавяне и изтриване на ограничения чрез Transact-SQL	57
Разрешаване и забраняване на ограничения	58
Използване на колона IDENTITY	59
Ограничение PRIMARY KEY	61
Ограничения UNIQUE	62
Ограничения FOREIGN KEY	63
Ограничения за валидност на данните	66
Глобално уникални идентификатори	71

Глава II Програмиране на бази от данни	75
Тема 5 Използване на SQL за извличане на данни от таблици.	
Конструкцията SELECT	81
Тема 6 Обобщаване на данните с помощта на агрегатни функции	87
Тема 7 Определяне на броя на редовете за избор	91
Тема 8 Използване на изрази. Изрази за дата/час и интервални изрази.....	93
Тема 9 Функции за работа със символни низове и изрази за конвертиране на типа данни	97
Тема 10 Условни (CASE) изрази	101
Тема 11 Съединения на таблици	104
Тема 12 Създаване на кръстосани заявки	110
Използване на CASE изрази за създаване на кръстосани заявки.....	
Използване на PIVOT за създаване на кръстосани заявки.....	
Тема 13 Обобщаване на данни чрез операторите CUBE и ROLLUP.....	115
Тема 14 Използване на операторите за обединение, сечение и разлика.....	119
Тема 15 Съставяне на подзаявки.....	123
Използване на оператори за подзаявки	125
Взаимосвързани заявки.....	127
Производни таблици	128
Тема 16 Използване на SQL за манипулиране на данни	133
Използване на INSERT за добавяне на данни	133
Използване на конструкция SELECT...INTO за добавяне на данни	134
Използване на UPDATE за променяне на данни	135
Използване на DELETE за изтриване на данни	137
Глава III Администриране на бази от данни	140
Тема 17 Индекси	140
Типове индекси	140
Създаване на индекс	141
Променяне на индекс	142
Изтриване на индекс	142
Тема 18 Анализирание и оптимизиране на достъпа до данните.....	149
Анализирание на заявки	150
Използване на SET конструкции	150
Анализирание на плана за изпълнение.....	151
Оптимизиране на заявки.....	155
Индекси	155
Съветници за настройка на индекси.....	160
Използване на подсказки в заявките	163
Литература	168

Предговор

Голямото значение на компютрите в нашия живот се определя от широкото им внедряване в икономиката. Без този факт те щяха да изглеждат като значително научно постижение на съвременната цивилизация като кацането на Луната – триумф на човешкия разум, но като цяло безполезно начинание за ежедневието ни. Началното развитие на изчислителната техника е свързано предимно с военни приложения (шифроване и разбиване на шифри, изчисления на траектории на снаряди и ракети) и военно-научни приложения (модел на атомната бомба, на водородната бомба и т.н.).

Компютърната техника намери своето масово разпространение в бизнеса с приложението на базите от данни. Неслучайно списание Forbes през 2002 година посочи релационния модел на данни, разработен от Едгар Код, за една от най-важните иновации от последните 85 години.

Почти всяко бизнес приложение поддържа своя база от данни, като се започне от вградените устройства (за управление на битова техника, за автомобилни агрегати и т.н.) и се стигне до мега базите от данни (GoogleEarth, Teradata и др.). В началото достъпни само на големите машини, днес базите от данни са налични в почти всяко едно компютъризирано устройство.

Знанията за базите от данни станаха неразделна част от компютърната грамотност и системите за управление на бази от данни са част от разширените офис пакети. Всичко това предполага изучаване на базите от данни както от информатици, така и от неинформатици, нуждаещи се най-малко от компютърна грамотност. Поради това в курсовете по информатика в повечето бакалавърски и магистърски програми е включено изучаването на бази от данни.

Изучаването на базите от данни се състои в изучаване на проектирането на бази от данни, програмиране на бази от данни и разработката на системи за управление на бази от данни (СУБД). Задължително е запознаването с програмиране на базите от данни, т.е. с начините за търсене в базата от данни и нейното изменение. Проектирането на базите от данни се занимава със създаването на нови бази от данни и изменението на съществуващите такива. Разработката на системи за управление на бази от данни дава основни познания, необходими както за разработката на СУБД, така и за администрирането на базите от данни. В представения материал е следвана тази логика на изложение – разгледани са и трите аспекта на базите от данни върху примера на Microsoft SQL Server 2008: обърнато е внимание на програмирането на базите от данни и проектирането им с прилагането на езика за манипулация на данни SQL, а разработката на СУБД е сведена до знания за администрирането на базите от данни.

Изложението е направено въз основа множество практически примери, без да се навлиза в дебрите на теорията – все пак това е практикум. Използването на една сравнително силна платформа, като Microsoft SQL Server 2008, позволява представянето на основните концепции от тематиката на базите от данни, без да се допуска изкривяване на материята, поради немощ на използваните програмни средства. Трябва да отбележим, че избраният подход ще създаде дългосрочни познания у студентите, тъй като се използват стандартни средства от света на базите от данни – поредицата стандарти SQL. От друга страна, амбициите на Microsoft са в недалечно бъдеще да наложи в своите продукти Microsoft SQL Server като единствена СУБД. Това е особено силната страна на курса в противовес на подхода за изучаване на базите от данни с MS Access, което за съжаление е доста разпространено у нас.

Във връзка с тези съображения, препоръчвам представения материал за учебно пособие на Практикум по бази от данни.

доц. д-р Владимир Димитров
Ръководител
на катедра „Компютърна информатика”
Факултет по математика и информатика
Софийски университет „Св. Климент Охридски”

Глава I Проектиране на бази от данни

Проектиране на релационни бази от данни. Нормализация на данните

Базите от данни съдържат организирана по специален начин информация. Голяма част от съвременните *системи за управление на бази от данни* (СУБД) съхраняват и обработват информацията, използвайки *релационен* модел за управление на бази от данни, доказал своите предимства. В системите за управление на релационни бази от данни (*Relational Database Management Systems – RDBMS*) системата управлява всички данни в *таблицы* (релации). Една таблица съхранява информация относно множество от дадени обекти (например клиенти, продукти) или събития (например прегледи на пациенти, продажби, поръчки) и е съвкупност от редове (записи, кортежи) и колони (полета, атрибути). Всяка колона в таблицата е предназначена да съхранява определен вид информация за обектите или събитията (например имена, дати, цени, количества, адреси и други). Редовете описват всички атрибути на отделен обект или събитие (например данни за конкретен клиент, продукт или продажба).

Една СУБД дава пълен контрол над процеса на дефиниране на данните, работата с тях и споделянето им с други потребители. СУБД предоставя три основни типа възможности:

- дефиниране на данни – дефиниране на таблиците в базата от данни, типа на данните и връзките между тях; задаване на ограничения;
- обработка на данни – избор на определени полета с данни, филтриране, сортиране, извличане на обобщени данни, добавяне, актуализиране, изтриване на определена информация;
- контрол на данни – определяне на кого е позволено да чете, актуализира или да въвежда данни; как да се споделят и актуализират данните от многобройните потребители.

Проектиране на релационни бази от данни

Теорията на релационното проектиране се състои от следните основни понятия: таблици и уникалност, външни ключове и домейни, релационни връзки, нормализация на данните, правила за запазване на целостността на данните.

Таблицы и уникалност

Основно правило в теорията на релационното проектиране на бази от данни е, че всяка таблица трябва да има уникален идентификатор на ред, който представлява колона или група от колони, използвани за разграничаване всеки отделен ред от останалите редове в таблицата. Нарича се *първичен ключ* (*primary key*). Първичният ключ може да бъде прост или съставен. Простият ключ е създаден от една колона, съдържаща уникални стойности за всеки ред от таблицата. Съставният ключ се дефинира от две или повече колони, комбинациите от стойности на които са уникални за всички редове на таблицата. Всяка таблица може да има само един първичен ключ, дори когато няколко колони или комбинации от колони съдържат уникални стойности (наречени *алтернативни ключове*). Изборът на първичен ключ трябва да се основава на принципите за:

- минималност (избират се толкова колони, колкото е необходимо);
- стабилност (избират се колони, които рядко биват променяни);
- простота (от колкото е възможно по-прост тип).

Например, за определяне на първичния ключ за таблицата, съдържаща данните за служителите (фиг. 1), могат да се разгледат следните възможности: EmployeeID, (FirstName, Surname, LastName, Title), PhoneNumber, EGN. Като се следват изискванията за първичен ключ, се достига до отхвърляне на последните три възможности. Имената и телефонните номера не могат да гарантират уникалност за всеки ред от таблицата и биват променяни сравнително често. Колоната EGN нарушава условието за простота – търсенето и сортирането на числени колони е много по-ефективно от това на текстови колони. Следователно най-подходящият избор за първичен ключ в таблицата Employees е EmployeeID.

	EmployeeID	FirstName	Surname	LastName	Title	EGN	HireDate	TerminationDate	PhoneNumber	EmailAddress
+	1	Иван	Иванов	Иванов	продавач	7001291234	05.1.1999 г.		123/123-656 656	
+	2	Петър	Петров	Иванов	продавач	5903221570	05.1.1999 г.			
+	3	Петър	Иванов	Петров	продавач	7603033333	05.1.1999 г.			
+	5	Иван	Петров	Петров	продавач	5505304321	05.1.1999 г.			
▶	(AutoNumber)						26.3.2004 г.			

Record: 5 of 5

Фиг. 1 Примерна таблица за служители

Външни ключове и домейни

Една релационна база от данни съдържа свързани помежду си таблици. Въпреки че първичните ключове са предназначени за отделните таблици, те са особено необходими, когато се създават отношенията между различните таблици в базата от данни. *Външният ключ (foreign key)* е колона или група от колони в дадена таблица, представляваща връзка с друга таблица посредством нейния първичен ключ.

Например, в таблицата Sales, показана на фигура 2, колоната EmployeeID е външен ключ, тъй като се използва, за да определи еднозначно даден служител, т.е. сочи към ред в таблицата Employees.

	SaleID	CustomerID	EmployeeID	SaleDate
+	1	1	1	14.11.2001 г. 22:01:54
+	4	1	2	26.8.2001 г. 01:11:39
▶	(AutoNumber)			26.3.2004 г. 18:23:13

Record: 3 of 3

Фиг. 2 Таблица, съдържаща външен ключ

Ключът, към който сочи един външен ключ, се нарича *родителски ключ*. Родителският ключ трябва да бъде уникален идентификатор, за да може да се определи към кой ред от таблицата на родителския ключ сочи външният ключ. Броят и типът на колоните, съставляващи външния ключ, трябва да съответства на броя и типа на колоните на родителския ключ, но могат да се използват различни имена за тях. Не е задължително стойностите на външния ключ да бъдат уникални в своята-собствена таблица. Те трябва да бъдат в същата област от допустими стойности (домейн), на която принадлежат стойностите на родителския ключ. *Домейните* са съвкупности от стойности, които са допустими за колоните.

Релационни връзки

Дефинирането на външни ключове има за цел да се моделират връзките между обектите в реалния свят. От предишните примери – двете таблици (Employees и Sales) си остават отделни обекти в базата от данни, но когато се извлича информация,

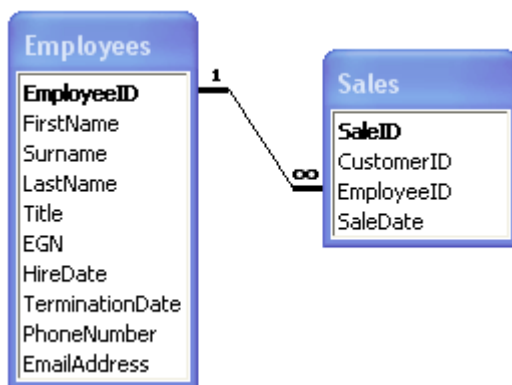
съдържаща се в тях, се свързва стойността на всеки външен ключ с неговия родителски ключ (първичен ключ в съответната таблица) и с останалите колони, които трябва да се изведат от таблиците, следователно когато се дефинират първични и външни ключове, се работи с релационни връзки. *Релационна връзка (relationship)* се нарича връзка между таблици, която е базирана на първичен ключ от едната таблица и външен ключ от другата таблица. Съществуват три типа релационни връзки между таблиците: релационни връзки “едно към едно”; релационни връзки “едно към много”; релационни връзки “много към много”.

Релационни връзки “едно към едно”

Две таблици *A* и *B* са свързани с релационна връзка “едно към едно”, ако за всеки ред от таблицата *A* има най-много един съответстващ ред от таблицата *B*, но всеки ред от таблицата *B* може да има точно един съответстващ ред от таблица *A*. Релационна връзка “едно към едно” се създава, ако и двете свързани колони са първични ключове или имат ограничения за уникалност. Този тип релационна връзка е най-рядко срещан, тъй като информация, свързана по този начин, обикновено се намира в една таблица. От съображения за сигурност може да се раздели информацията в две таблици. Например, част от информацията за пациенти в таблицата *Patients* може да се отдели в таблица *Confidential*, достъпът до която е по-ограничен. Друг подходящ пример е случая, когато е необходимо често да се извършва прехвърляне на данните, съхранявани в част от колоните на дадена таблица, към някое друго приложение. Тогава е удобно таблицата да се раздели на две части (едната включва колоните, които подлежат на трансфер, другата – тези, които не участват в прехвърлянето), свързани с релационна връзка “едно към едно”.

Релационни връзки “едно към много”

Две таблици *A* и *B* са свързани с релационна връзка “едно към много”, ако за всеки ред от таблицата *A* има нула или повече съответстващи редове от таблицата *B*, но всеки ред от таблицата *B* може да има точно един съответстващ ред от таблицата *A*. Този тип релационна връзка е най-често срещан и може да се създаде, ако само една от свързаните колони е първичен ключ или има ограничение за уникалност. Например, таблиците *Publishers* и *Books* са свързани с релационна връзка “едно към много”, тъй като всеки издател издава много книги, но всяка книга се издава точно от един издател. Разгледаните вече таблици *Employees* и *Sales* също са свързани с релационна връзка от този тип (фиг. 3).



Фиг. 3 Таблици, свързани с релационна връзка “едно към много”

Релационни връзки “много към много”

Две таблици *A* и *B* са свързани с релационна връзка “много към много”, ако за всеки ред от таблица *A* има много съответстващи редове от таблицата *B* и обратно. Този тип релационна връзка се създава, като се дефинира трета таблица, наречена *свързваща таблица* (*junction table*), която съдържа първичните ключове от двете таблици като външни ключове; първичният ключ в свързващата таблица е съставен от двата външни ключа. Например, между таблиците *Books* и *Authors* има релационна връзка “много към много”, тъй като една книга може да има повече от един автор и всеки автор може да е написал много книги. Ако имената на авторите се запишат в една колона в таблицата *Books*, ще се затрудни търсенето на книги по автор. Друго възможно решение е да се промени дизайна на таблицата *Books*, като се добави още една колона и т.н.; по този начин се заделя неизползвано пространство в паметта за книги, които имат само по един автор. Затова се създава допълнителна първична таблица за авторите *Authors* и после се създава трета таблица, която съпоставя книгите с авторите им *BookAuthor* (фиг. 4).

Books					
BookID	Title	Authors			
187	T1				
190	T2				

BookAuthor		Authors		
BookID	AuthorID	AuthorID	FirstName	LastName
187	1787	1787	F1	L1
187	1790	1790	F2	L2
190	1787	1800	F3	L3
190	1800			

Фиг. 4 Таблицы за съхраняване на информация за книги и техните автори

Следователно една релационна връзка “много към много” се представя чрез две релационни връзки “едно към много” (фиг. 5).



Фиг. 5 Представяне на релационна връзка „много към много” чрез свързваща таблица

Нормализация на данните

Нормализацията на данните се нарича процеса на разделяне на данните в множество свързани помежду си таблици. Оптимизацията на проекта на базата от данни включва процеса на нормализация. Решенията, взети за структурата на таблиците, могат да повлияят по-късно върху производителността и върху написването на програмите по време на процеса на разработване. Нормализацията на данните осигурява организиране на данните по такъв начин, че:

- актуализирането на някой елемент от данните ще изисква в общия случай действие само на едно място;
- изтриването на определен елемент от данните няма да доведе до нежелана загуба на други данни.

Нормализацията се използва при проектирането на базата от данни, за да бъде сигурно, че крайната база от данни изразява структурата, която минимизира

дублирането на данните и в същото време избягва непълнотите и други аномалии в базата от данни.

Целта на нормализацията е да се декомпозират таблиците в по-малки таблици, дефинирани така, че да се избегне излишно дублиране на данни. След разделянето на данните в отделни по-малки и добре структурирани таблици, се дефинират релационни връзки между тях.

За да се получи нормализация на данните, първо се достига до първа нормална форма (First Normal Form – 1NF), след това до втора нормална форма (Second Normal Form – 2NF) и накрая до трета нормална форма (Third Normal Form – 3NF).

Първа нормална форма

Една таблица е в първа нормална форма, ако всяка колона има точно една стойност за всеки ред, а не списък от стойности. Първата нормална форма изисква всички стойности на данните да бъдат атомарни (т.нар. принцип за неделимост). Пример за таблица, която не е в 1NF, е показан на фигура 6.

SaleID	CustomerID	CustomerName	EmployeeID	SaleDate	ProductID	ProductName	Quantity	Price
1	1	Стоянов	1	20.11.2003 г.	12; 3; 4	име12; име3; име4	1; 8; 587	30,50; 18,87; 53,20
2	2	Христов	1	20.11.2003 г.	18	име18	90	5,50
3	1	Стоянов	2	21.11.2003 г.	3; 87	име3; име87	2,30; 9	18,87; 2
*				03.3.2006 г.				

Фиг. 6 Таблица, която не е в 1NF

Таблицата Sales не е в 1NF, тъй като полетата ProductID, ProductName, Quantity, Price съхраняват списък от стойности за един ред. 1NF забранява също и наличието на повтарящи се групи, дори ако те се съхраняват в различни колони. Например, ако в разглежданата таблица данните за продадените продукти се разположат в отделни колони: PrID1, PrName1, Quant1, Price1, PrID2, PrName2, Quant2, Price2, PrID3, PrName3, Quant3, Price3, както е показано на фигура 7.

SaleID	CustomerID	CustomerName	EmployeeID	SaleDate	PrID1	PrName1	Quant1	Price1	PrID2	PrName2	Quant2	Price2	PrID3	PrName3	Quant3	Price3
1	1	Стоянов	1	20.11.2003 г.	12	име12	1	30,50 лв	3	име3	8,00	18,87 лв	4	име4	5,87	53,20 лв
2	2	Христов	1	20.11.2003 г.	18	име18	90	5,50 лв								
3	1	Стоянов	2	21.11.2003 г.	3	име3	9	18,87 лв	87	име87	2,30	2,00 лв				
*				27.3.2004 г.												

Фиг. 7 Таблица, съдържаща повтаряща се група от колони

Въпреки че този дизайн разделя информацията в отделни колони, има съществени недостатъци. Проблеми могат да възникнат, ако се наложи да се продадат повече от три продукта с една продажба; да се определи количеството, продадено от даден продукт през определен период от време и др.

На фигура 8 е показано как трябва да изглежда таблицата, за да бъде в 1NF.

Sales : Table									
	SaleID	CustomerID	CustomerName	EmployeeID	SaleDate	ProductID	ProductName	Quantity	Price
▶	1	1	Стоянов	1	20.11.2003 г.	12	име12	1,000	30,50 лв
	1	1	Стоянов	1	20.11.2003 г.	3	име3	8,000	18,87 лв
	1	1	Стоянов	1	20.11.2003 г.	4	име4	5,875	53,20 лв
	2	2	Христов	1	20.11.2003 г.	18	име18	90,000	5,50 лв
	3	1	Стоянов	2	21.11.2003 г.	3	име3	2,300	18,87 лв
	3	1	Стоянов	2	21.11.2003 г.	87	име87	9,000	2,00 лв
*					27.3.2004 г.				
Record: 1 of 6									

Фиг. 8 Таблица, която е в 1NF

Втора нормална форма

Една таблица е във втора нормална форма, ако тя е в първа нормална форма и всяка колона, която не е част от ключа (т.е. не е ключова колона), зависи от първичния ключ (който може да е съставен) и не зависи от някое подмножество на първичния ключ. В предишния пример първичният ключ е съставен от колоните SaleID и ProductID. Колоните CustomerID, CustomerName, EmployeeID и SaleDate зависят от SaleID и не зависят от ProductID. Колоните ProductName и Price зависят само от ProductID и не зависят от SaleID. Колоната за продаденото количество от даден продукт с дадена продажба Quantity зависи от двете колони, съставлящи първичния ключ SaleID и ProductID. На фигура 9 са показани получените таблици във 2NF.

Sales : Table					
	SaleID	CustomerID	CustomerName	EmployeeID	SaleDate
▶	1	1	Стоянов	1	20.11.2003 г.
	2	2	Христов	1	20.11.2003 г.
	3	1	Стоянов	2	21.11.2003 г.
*					27.3.2004 г.
Record: 1 of 3					

SaleDetails : Та...			
	SaleID	ProductID	Quantity
	1	3	8,000
	1	4	5,875
	1	12	1,000
	2	18	90,000
	3	3	9,000
	3	87	2,300
▶			
Record: 7			

Products : Table			
	ProductID	ProductName	Price
	3	име3	18,87 лв
	4	име4	53,20 лв
	12	име12	30,50 лв
	87	име87	2,00 лв
▶			
Record: 5			

Фиг. 9 Таблицы, които са във 2NF

Трета нормална форма

Една таблица е в трета нормална форма, ако тя е във втора нормална форма и всяка колона, която не е част от ключ, не зависи от нищо друго, а само от ключа. В предишния пример таблиците SaleDetails и Products съдържат неключови колони, които могат да зависят само от ключа. В таблицата Sales обаче има колона CustomerName, която зависи от CustomerID. Затова се използва нова таблица Customers, която има първичен ключ CustomerID и неключово поле CustomerName (фиг. 10). Връзката между таблиците Customers и Sales е “едно

към много”. По този начин се избягва повторение на данните (имената на клиентите се появяват само веднъж).

The figure shows four database tables in a 3NF structure:

SaleID	CustomerID	EmployeeID	SaleDate
1	1	1	20.11.2003 г.
2	2	1	20.11.2003 г.
3	1	2	21.11.2003 г.
*			27.3.2004 г.

SaleID	ProductID	Quantity
1	3	8,000
1	4	5,875
1	12	1,000
2	18	90,000
3	3	9,000
3	87	2,300

CustomerID	CustomerName
1	Стоянов
2	Христов

ProductID	ProductName	Price
3	име3	18,87 лв
4	име4	53,20 лв
12	име12	30,50 лв
87	име87	2,00 лв

Фиг. 10 Таблицы, които са в 3NF

Друг пример за наличие на зависимост е изчислима колона. Например, ако една таблица съдържа колоните Quantity и Price, добавянето на колона TotalCost (която е равна на Quantity*Price) в същата таблица води до нарушаване на 3NF. Ако такава колона не съществува, изчислението може да се извърши чрез заявка. По този начин се избягва актуализирането на изчислимата колона TotalCost, всеки път когато колоните Quantity и Price се променят.

Едно напълно нормализирано множество от типове обекти и атрибути дава сигурност, че данните за типовете обекти не зависят от съществуването на други типове обекти. Ако се разгледа първоначалната таблица Sales, се забелязва, че ако нямаше продажба със SaleID = 2, нямаше да се появят както данни за клиент с CustomerID = 2, така и данни за продукт с ProductID = 18.

Наличието на списък от стойности в съдържанието на колони прави търсенето в тях невъзможно, редактирането – трудно. С въвеждането на 1NF се дава по-голяма гъвкавост при проследяването на продуктите, техните количества и цени за съответните продажби. Данните, в които е въведена 1NF, обаче трудно се управляват, тъй като информацията се повтаря много пъти, което води до прекалено заемане на дисково пространство; трудна и времепоглъщаща дейност за актуализиране на всички въвеждания, ако се промени името на един клиент или цена на даден продукт. Чрез преместване на данните и получаване на 3NF се постига по-голяма гъвкавост на модела, тъй като може да се управляват клиентите и продуктите в отделни таблици.

Следователно, както се вижда от примерите, основните предимства от нормализацията на данните са:

- данните заемат по-малко място за съхраняване;
- по-бързо се изпълняват актуализациите на данните;
- по-лесно се въвеждат нови данни;
- намалява се наличието на несъгласуваност и противоречивост на данните;
- ясно се дефинират релационните връзки между данните в таблиците;

- получава се гъвкава структура, която позволява лесно разширяване чрез добавяне на нови колони и/или таблици.

Допълнителна нормализация

Допълнителните нормални форми Boyce/Codd и четвърта нормална форма се прилагат в специални и рядко срещани случаи.

Boyce/Codd нормална форма (Boyce/Codd Normal Form – BCNF)

Boyce/Codd нормална форма е по-скоро едно усъвършенстване на 3NF, което е предназначено за ситуациите, които възникват, когато има няколко възможности за ключа със съвпадащи отчасти компоненти. Например, нека таблицата SaleDetails да съдържа колоните SaleID, ProductID, ProductName, Quantity. Ако се приеме, че имената на продуктите са уникални, следователно ключът за таблицата SaleDetails би могъл да бъде съставен от SaleID и ProductID или от SaleID и ProductName. Съществува ненужен излишък от информация, тъй като всеки път, когато се продава продукт трябва да се записва идентификатора и името му. Затова според BCNF трябва да се разделят в отделни таблици прекалено многото ключове. Имената трябва да се появяват само в таблицата, която описва обекта – продукт, т.е. таблицата Products: ProductID, ProductName, Price, др. и таблицата SaleDetails да включва колоните SaleID, ProductID, Quantity, др.

В някои случаи може да се получи таблица, която нарушава BCNF, но тази таблица не може да бъде декомпозирана в таблици, които удовлетворяват BCNF и запазват зависимостите, изпълнени в първоначалната таблица. Следователно, за разлика от първите три нормални форми, BCNF невинаги е постижима. Например, нека е дадена таблицата FavoriteStores, показана на фигура 11.

FavoriteStores		
CustomerName ▾	City ▾	FavoriteStore ▾
Иван Иванов	Горна Оряховица	Пингвините ГО1
Иван Иванов	Велико Търново	Пингвините ВТ1
Петър Иванов	Лясковец	Хеликон
Христо Димитров	Габрово	Сиела
Христо Димитров	Велико Търново	Буквите
Христо Димитров	Горна Оряховица	Пингвините ГО1

Фиг. 11 Таблица, която не е в BCNF

Предназначението на таблицата FavoriteStores е за всяка комбинация от име на клиент (CustomerName) и град (City) да съхранява информация за точно една предпочитана от клиента книжарница (FavoriteStore) в този град. Освен това всяка книжарница се намира в точно един град.

За разглежданата таблица са налице следните възможности за първичния ключ:

- да бъде съставен от CustomerName и City;
- да бъде съставен от CustomerName и FavoriteStore.

Таблицата FavoriteStore не е в BCNF, тъй като колоната City зависи от неключовата колона FavoriteStore. Според BCNF таблицата трябва да се декомпозира в две таблици CustFavoriteStores и Stores (фиг. 12).

CustFavoriteStores		Stores	
CustomerName	Store	Store	City
Иван Иванов	Пингвините ГО1	Пингвините ГО1	Горна Оряховица
Иван Иванов	Пингвините BT1	Пингвините BT1	Велико Търново
Петър Иванов	Хеликон	Хеликон	Лясковец
Христо Димитров	Сиела	Сиела	Габрово
Христо Димитров	Буквите	Буквите	Велико Търново
Христо Димитров	Пингвините ГО1		

Фиг. 12 Таблицы, които са в BCNF

В променения проект таблицата CustFavoriteStores има ключ, съставен от CustomerName и City; ключът на таблицата Stores се състои от колоната Store. Въпреки че получените таблици са в BCNF, те са неприемливи поради следната причина: възможно е да се въведат няколко книжарници от един и същи град за един и същи клиент. Не може да се гарантира, че стойностите на колоните CustomerName и City определят стойността на колоната Store.

Проектът, който отстранява всички разглеждани недостатъци, е възможен. Той включва първоначалната таблица FavoriteStores, допълнена с таблицата Stores (фиг. 13).

FavoriteStores		
CustomerName	City	FavoriteStore
Иван Иванов	Горна Оряховица	Пингвините ГО1
Иван Иванов	Велико Търново	Пингвините BT1
Петър Иванов	Лясковец	Хеликон
Христо Димитров	Габрово	Сиела
Христо Димитров	Велико Търново	Буквите
Христо Димитров	Горна Оряховица	Пингвините ГО1

Stores	
Store	City
Пингвините ГО1	Горна Оряховица
Пингвините BT1	Велико Търново
Хеликон	Лясковец
Сиела	Габрово
Буквите	Велико Търново

Фиг. 13 Таблицы, които отстраняват недостатъците на предишните варианти

Колоната Store в таблицата FavoriteStores е външен ключ, който трябва да сочи ред от таблицата Stores. По този начин описаните недостатъци на предишните варианти са избегнати, но BCNF не е спазена.

Четвърта нормална форма

Една таблица е в четвърта нормална форма, ако са премахнати всички независими групи от колони. Например, нека продуктите могат да бъдат доставяни в различни по размер опаковки от съответните доставчици (фиг. 14).

ProductSupplier : Table				
	ProductID	ProductName	SupplierID	PackSize
▶	3	име3	1	1kg; 2kg; 5kg
	3	име3	2	1kg; 2kg; 5kg
	4	име4	1	0,5kg; 1kg; 2kg
	4	име4	2	0,5kg; 1kg; 2kg
	12	име12	3	0,5l; 1l; 1,5l; 2l
	12	име12	4	0,25l; 0,5l; 1l; 2l
	87	име87	3	0,25l; 0,5l; 1l; 2l
	87	име87	4	0,25l; 0,5l; 1l; 2l
*				
Record: 1 of 8				

Фиг. 14 Таблица за продукти, доставяни в различни по размер опаковки от съответните доставчици

Първата стъпка е създаването на таблици, които са в 1NF, т.е. осигуряването на атомарни стойности за колоната PackSize (фиг. 15).

ProductSupplier : T...			
	ProductID	SupplierID	PackSize
▶	3	1	5kg
	3	1	1kg
	3	1	2kg
	3	2	1kg
	3	2	5kg
	3	2	2kg
	4	1	0,5kg
	4	1	2kg
	4	1	1kg
	4	2	1kg
	4	2	2kg
	4	2	0,5kg
	12	3	1,5l
	12	3	0,5l
	12	3	1l
	12	3	2l
	12	4	0,5l
	12	4	1l
	12	4	1,5l
	12	4	2l
	87	3	0,25l
	87	3	0,5l
	87	3	2l
	87	3	1l
	87	4	2l
	87	4	0,25l
	87	4	0,5l
	87	4	1l
*			
Record: 1			

Products : ...		
	ProductID	ProductName
▶	3	име3
	4	име4
	12	име12
	87	име87
*		
Record: 1		

Фиг. 15 Необходими са две таблици в 1NF

Таблицата ProductSupplier е в Boyce/Codd нормална форма, тъй като всички колони съставят първичния ключ, но съществува излишък от информация, поради наличие на две взаимно независими множества от колони – {ProductID,

PackSize} и {ProductID, SupplierID}. Това означава, че се получава преопределяне на информацията за доставчик и размер на опаковка за даден продукт. Съгласно 4NF е необходимо отделяне в различни таблици на всички взаимно независими групи, в резултат на което се получават таблиците, показани на фигура 16.

The figure shows two database tables side-by-side. The left table is titled 'ProductSupplier' and has two columns: 'ProductID' and 'SupplierID'. It contains 8 rows of data. The right table is titled 'ProductPackSize' and has two columns: 'ProductID' and 'PackSize'. It contains 16 rows of data, showing various product sizes.

ProductID	SupplierID
3	1
3	2
4	1
4	2
12	3
12	4
87	3
87	4

ProductID	PackSize
3	2kg
3	5kg
3	1kg
4	0,5kg
4	1kg
4	2kg
12	0,5l
12	1l
12	1,5l
12	2l
87	0,25l
87	0,5l
87	1l
87	2l

Фиг. 16 Таблицы, които са в 4NF

Необходимостта от прилагане на 4NF произлиза от наличието на множество стойности за атрибутите. Ако в разгледания пример продуктите имат само един доставчик или само един размер на опаковка, 4NF няма нужда да бъде прилагана. Освен това ако двете множества от атрибути не са взаимно независими, таблицата ще нарушава 2NF.

Правила за запазване на целостността на данните

Целостността на данните е важно понятие за проектиране на базите от данни. Има четири вида цялостност на данните:

- цялостност на обект – едно от изискванията на проектирането на релационна база от данни е възможността да се разграничат различните екземпляри на дадено множество от обекти. Това понятие е известно като цялостност на обект и се реализира чрез създаване на първичен ключ. Според това правило за цялостност колоните, съставлящи първичния ключ, не могат да имат стойност NULL. Релационните бази от данни поддържат специална стойност NULL, която указва неизвестните стойности (*unknown*).
- цялостност на област – свързана е с осигуряване на валидност на стойностите на колоните, т.е. да принадлежат на допустима област от стойности. Реализира се с определяне на типа на колоните, допускане или не на стойност NULL, ограничения за валидност, стойност по подразбиране, дефиниране на външен ключ.
- цялостност на връзка – запазва дефинираните отношения (релационни връзки) между таблиците, когато се въвеждат, променят или изтриват редове. Целостността на връзките гарантира, че съществува съгласуваност на стойностите на ключовете между таблиците – първичните и външните в съответните таблици. Реализира се с дефиниране на ограничението външен

ключ. Когато се наложи цялостност на връзка, не се допускат следните действия:

- да се добавят редове (или да се променят стойностите на колоните на външния ключ) в една таблица, която е страната “много” на релационната връзка, ако в първичната таблица, която е страната “едно” на релационната връзка, липсва съответен ред. Например, не е възможно в таблицата `SaleDetails` да се въведе ред с информация за продажба на продукт, който не съществува в таблицата `Products`.
 - да се променят стойностите на колоните на първичния ключ в една таблица, която е страната “едно” на релационната връзка, ако в свързаната таблица (страната “много” на релационната връзка) има поне един съответен ред. Например, недопустимо е да се промени стойността на първичния ключ `ProductID` в таблицата `Products` за продукт, за който съществува поне един съответен ред в `SaleDetails` за осъществена продажба от този продукт.
 - да се изтриват редове от една първична таблица, ако има свързани редове в таблицата с външните ключове, т.е. страната “много” на релационната връзка. Например, не се допуска изтриване на ред от `Customers` за клиент, за който съществува поне един ред в таблицата `Sales` за осъществена продажба на този клиент.
- Дефинирана от потребителя цялостност – дава възможност за определянето на специфични бизнес правила, които не могат да се отнесат към някоя от другите категории цялостност. Реализира се чрез създаване на ограничения, съхранени процедури, функции и тригери.

Денормализация на данните

Понякога е необходимо да се нарушат правилата за нормализация. Обикновено това се прави с цел повишаване на производителността. Когато се вземе решение за денормализация на базата от данни, трябва да се има предвид необходимостта от създаване на приложения (допълнителен код) за избягване на аномалии, които нормализираният проект на базата от данни не допуска. Затова не винаги денормализацията е най-добрия подход. Първоначално се създава напълно нормализирана база от данни (до 3NF или по-висока) и по-късно само ако се установи необходимост от подобряване на производителността, се денормализира. Желателно е да се извършат тестове, чрез които да се направи оценка на подходящия за конкретната база от данни компромис. Някои от случаите, в които би могло да се избере нарушаване на правилата за нормализация, са:

- Съхраняване на изчислима колона с цел повишаване на производителността на отчетите – например добавяне в таблицата `Sales` на колона `TotalForSale`, която съдържа сума от `Quantity*Price` на продуктите, продадени с дадена продажба. Ако често е необходимо да се извличат обобщени резултати, за които се използват сумите на извършените продажби, няма да е нужно свързване на двете таблици всеки път, когато този отчет се генерира. При добавяне, променяне или изтриване на ред в `SaleDetails` трябва да се поддържа актуална колоната `TotalForSale` в таблицата `Sales`. Затова е нужно да се създаде допълнителна процедура, свързана със съответното събитие.
- Освен колоната `SaleID` да се включи и колона `SalesPerson`, съхраняваща името на служителя, осъществил продажбата, в таблицата за издадените фактури `Invoices`. По този начин се нарушава 3NF, тъй като по стойността на колоната

SaleID може да се определи стойността на колоната SalesPerson, но това значително ще подобри производителността на някои обикновено често изпълнявани отчети. Избягва се свързването с таблицата Employees, но се изисква въвеждане на излишна информация, което води до съществуването на риск от аномалии при модифициране на данните (които могат да се предотвратят чрез изпълняването на допълнителен код).

Задачи

Задача 1. Проектира се база от данни, която да обработва информацията, необходима на една *обществена библиотека*. Да се разгледа следната таблица и да се нормализира:

Наемане на книги
Идентификатор на наемане
Дата на наемане на книги
Идентификатор на читател
Име на читател
Презиме на читател
Фамилия на читател
Адрес на читател
Идентификатор на книга
Заглавие
Автор(и)
Издателство
ISBN
Година на издаване
Дата на връщане на книга

Задача 2. Проектира се база от данни, която да съдържа информация за *поръчки на клиенти*. Да се разгледа следната таблица и да се нормализира:

Поръчки на клиенти
Идентификатор на поръчка
Дата на поръчка
Идентификатор на клиент
Наименование на клиент
Адрес на клиент
Идентификатор на продукт
Наименование на продукт
Категория на продукт
Цена на продукт
Поръчано количество от продукт

Задача 3. Проектира се база от данни, която да обработва информацията, необходима на една *книжарница*. Да се разгледа следната таблица и да се нормализира:

Продажби на книги
Идентификатор на продажба
Дата на продажба
Идентификатор на книга
Заглавие на книга
Автор(и)
Издателство
ISBN
Година на издаване
Цена на книга
Продадено количество

Забележка: Колоната “Продадено количество” съдържа броя на продадените екземпляри от дадена книга с дадена продажба.

Задача 4. Проектира се база от данни, която да обработва информацията, необходима на една *видеотека*. Да се разгледа следната таблица и да се нормализира:

Наемане на видеокасети
Идентификатор на наемане
Дата на наемане на касети
Идентификатор на клиент
Име на клиент
Презиме на клиент
Фамилия на клиент
Адрес на клиент
Телефонен номер на клиент
Идентификатор на касета
Заглавие на филм
Категория на филм
Режисьор на филм
Цена на касета
Дата на връщане на касета

Задача 5. Проектира се база от данни, която да обработва информацията, необходима на един *лекарски кабинет*. Да се разгледа следната таблица и да се нормализира:

Прегледи на пациенти
Идентификатор на пациент
Име на пациент
Презиме на пациент
Фамилия на пациент
Адрес на пациент
ЕГН на пациент
Идентификатор на резервация за преглед
Дата на преглед
Час на преглед
Диагноза
Идентификатор на лекарство
Наименование на лекарство
Доза

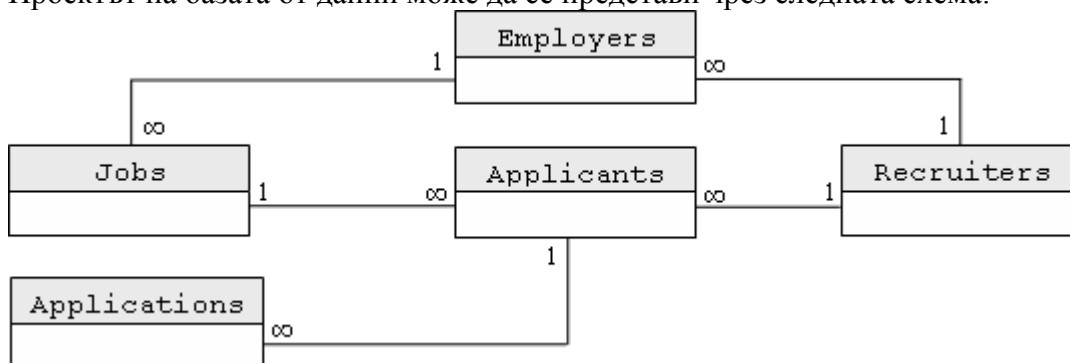
Задача 6. Проектира се база от данни, която да обработва информацията, необходима на едно *висше учебно заведение*. Да се разгледат следните таблици и да се нормализират:

Студенти	Учебна програма
Факултетен номер	Идентификатор на изучаван предмет
Име на студент	Специалност
Презиме на студент	Дисциплина
Фамилия на студент	Семестър
ЕГН на студент	Идентификатор на изпит
Адрес	Номер на изпитен протокол
Телефонен номер	Дата на изпит
Специалност	Сесия
Семестър	
Идентификатор на изпит	
Оценка	

Задача 7. Разработва се проект на база от данни, която ще се използва от *бюро по труда*. В бюрото се приемат молби от хора, търсещи работа и се поддържа информация за свободните длъжности, предложени от съответните работодатели. Всеки кандидат може да декларира повече от едно желания за различни длъжности. Някои от служителите в бюрото по труда търсят подходящи кандидати за свободните работни места, а други се опитват да намерят свободни работни места за регистрираните кандидати. Когато бъде открито съответствие, кандидатът получава информация за контакт със съответния работодател. Ако кандидатът бъде нает, служителят в бюрото по труда, регистрирал работодателя и служителят, регистрирал кандидата за работа, си поделят възнаграждението. Да се определят таблиците, които се изискват за дефиниране на нормализиран проект на базата от данни.

Решение:

Проектът на базата от данни може да се представи чрез следната схема:



В проекта са включени следните основни таблици:

Jobs съдържа информация за предлаганите свободни длъжности – описания на длъжности, заплати, брой свободни работни места. Съдържа колона `EmployerID`, за която е дефинирано ограничение външен ключ, рефериращ таблицата `Employers`, за да се определи работодателя, предлагащ работното място.

Recruiters поддържа информация за всички служители в бюрото по труда.

Employers включва данните за работодателите, предлагащи свободни работни места. За да се определи служителя в бюрото по труда, който е регистрирал съответния работодател, в тази таблица е създадена колона `RecruiterID`, която има наложено ограничение външен ключ, рефериращ таблицата `Recruiters`.

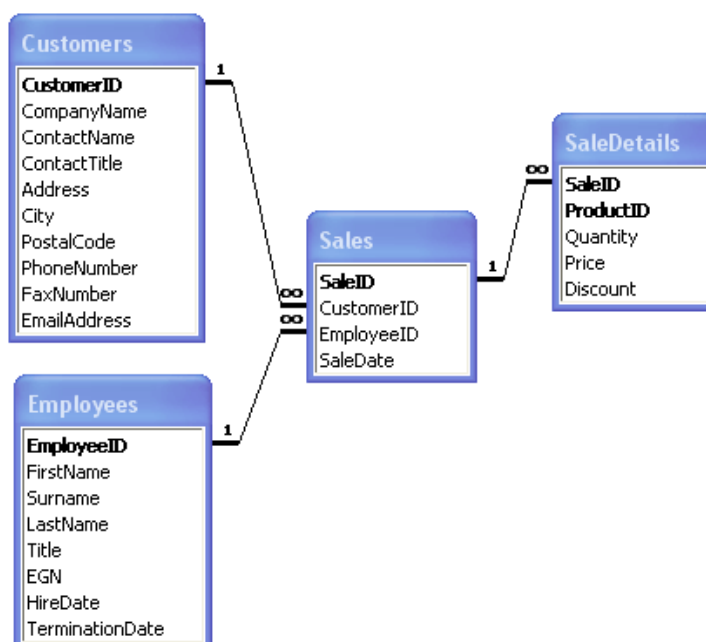
Applicants съдържа данните за кандидатите за работни места – имена, адрес, телефонни номера, тяхната квалификация. Създадена е колона `RecruiterID` с ограничение външен ключ, рефериращ таблицата `Recruiters`. По този начин

се идентифицира служителя, регистрирал кандидата в бюрото по труда. За да може да се отбележи, че даден кандидат е нает, в тази таблица е дефиниран външен ключ, рефериращ таблицата Jobs.

Applications поддържа информация за декларираните от кандидатите желания за различни длъжности, работни заплати. Съдържа външен ключ ApplicationID, рефериращ таблицата Applicants.

Задача 8. Проектира се база от данни, която да обработва информацията, необходима на една система за проследяване на рекламите, включени в програмите на предаванията в телевизионна компания. Необходимо е да се съхранява разписанието на телевизионните излъчвания на реклами, както и да се поддържа информация за рекламодателите. Всеки рекламодател може да има една или повече реклами. Всеки път, когато рекламата се излъчва, в сметката на съответния рекламодател се отбелязва дължимата сума. Всеки месец трябва да бъде изготвен официален отчет, показващ всяка парична сума за излъчена реклама, за да се уведомят рекламодателите за техните счетоводни баланси. Всеки рекламодател може да има няколко сметки. Да се определят таблиците, които се изискват за дефиниране на нормализиран проект на базата от данни.

Задача 9. Разработен е проект на база от данни за фирма, която се занимава с продажба на определени продукти (фиг. 17). Потребителите на базата от данни изказват недоволство от производителността на приложението, когато извличат информация за общата сума на продажбите по име на служител, по име на клиент, по дата. След проучване на проблема е установено, че той се дължи на броя на таблиците, които се свързват, за да бъде пресметната сумата. Да се предложи начин за намаляване на времето за отговор на посочените запитвания.



Фиг. 17 Примерен проект на база от данни за фирма, която се занимава с продажба на определени продукти

Създаване и управление на бази от данни на SQL Server

Една база от данни на SQL Server се състои от набор от обекти (таблици, ограничения, индекси, изгледи, съхранени процедури, дефинирани от потребителя функции, тригери), които представляват логически компоненти, видими за потребителите. Освен това всяка база от данни е реализирана физически като файлове на диска и може да обхваща множество файлове на операционната система.

Системни бази от данни

Всяка нова инсталация на SQL Server включва автоматично няколко бази от данни: *master*, *model*, *tempdb*, *msdb*.

master Базата от данни *master* съдържа системни таблици, съхраняващи информация от инсталацията на сървъра като цяло, както и за всички потребителски създадени бази от данни:

- установените параметри на конфигурацията за цялата система;
- акаунти за логване;
- съществуването на други бази от данни и съществуването на други SQL сървъри (за осъществяване на разпределените операции);
- за дисковото пространство, разположението и използването на файловете на базите от данни.

Базата от данни *master* е от изключително съществено значение за системата, следователно трябва винаги да е налице нейно актуално архивно копие. Операции като създаване, променяне или изтриване на друга база от данни, промяна на стойностите за конфигуриране или промяна на акаунтите за логване предизвикват промяна в *master*, така че след извършване на такива действия трябва да се направи архивно копие на *master*.

model Базата от данни *model* се използва като шаблон за всички бази от данни, които се създават в системата. При всяко създаване на нова база от данни SQL Server прави копие на *model*. Ако трябва всяка нова база от данни да стартира с определени обекти или разрешения за достъп, възможно е те да се поставят в *model* и всички нови бази от данни ще ги наследяват.

tempdb Базата от данни *tempdb* съдържа всички временни таблици и временни съхранени процедури. За разлика от всички други бази от данни тя бива създавана отново, а не възстановявана, при всяко рестартиране на SQL Server и всички предишни обекти, създадени от потребителя, се загубват. Използва се за временни таблици и съхранени процедури, създадени от потребители, както и за работни таблици за съхраняване на междинни резултати, създавани вътрешно от SQL Server в процеса на обработка на заявки и сортиране. Освен това системната база от данни *tempdb* дава възможност да се осъществява обмен на информация между различните едновременни конекции. Всички потребители имат привилегията да създават и използват частни (с префикс #) и глобални (с префикс ##) временни таблици, които се разполагат в *tempdb*. Но по подразбиране потребителите нямат привилегия да използват директно *tempdb* и следователно да създават таблица в нея. На отделни потребители може да се предостави такава привилегия.

msdb Базата от данни *msdb* се използва от услугата SQL Server Agent, която е предназначена за изпълняване на планирани действия като задачи за репликации и архивиране. Информацията, съхранявана в *msdb*, е достъпна от SQL Server Management Studio, така че не е необходимо да се прави директно обръщение към тези таблици.

Файлове на базите от данни

Всяка база от данни на SQL Server обхваща поне два файла на операционната система – един за данни и един за дневника на транзакциите, които се задават при създаване или променяне на базата от данни. SQL Server включва следните три типа файлове на базата от данни:

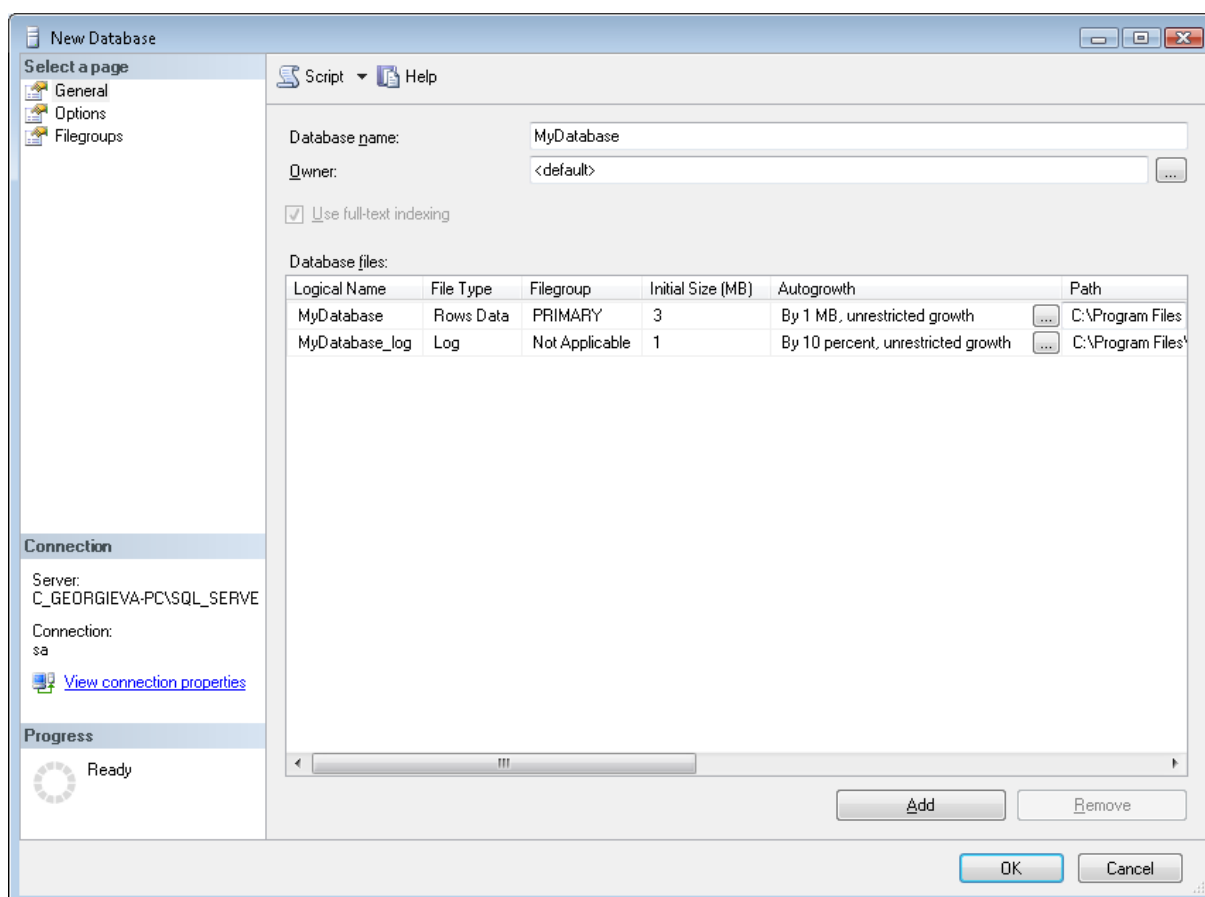
- главни файлове с данни – всяка база от данни има точно един главен файл с данни, който освен данни съхранява и информация за всички останали файлове в базата от данни. Главният файл има разширение *.mdf*;
- второстепенни файлове с данни – една база от данни може да има нула или повече второстепенни файлове с данни. Второстепенните файлове имат разширение *.ndf*;
- файлове-дневници – всяка база от данни има поне един файл-дневник, който съдържа информацията, необходима за възстановяване на всички транзакции в базата от данни. Файловете-дневници имат разширение *.ldf*.

Всички файлове на базата от данни имат няколко свойства, чрез които се задава логическо име на файла, физическо име на файла, начален размер, максимален размер и стъпка на нарастване.

Създаване на база от данни

Всяка система за управление на релационни бази от данни използва SQL (*Structured Query Language* – език за структурирани заявки). Стандартът SQL се дефинира съвместно от ANSI (American National Standards Institute – Американски национален институт по стандартизация) и ISO (International Organization for Standardization – Международна организация по стандартизация). Transact-SQL (T-SQL) представлява надстройка на стандарта SQL.

Най-бързият начин за създаване на база от данни е, като се използва Management Studio на SQL Server, който предоставя графичен интерфейс към команди на SQL и Transact-SQL, изпълняващи различни задачи. Ако се избере командата *New Database...* от контекстното меню на папката *Database* в Microsoft SQL Server Management Studio, се отваря диалоговия прозорец, показан на фигура 1.



Фиг. 1 Създаване на база от данни в Microsoft SQL Server Management Studio

Когато се създава нова потребителска база от данни, SQL Server копира базата от данни *model*, следователно ако има обект, който трябва да бъде създаван във всяка следваща потребителска база от данни, този обект може да се създаде първо в базата от данни *model*. Може да се използва *model* и за да се променят подразбиращите се опции на базата от данни във всички следващи бази от данни. Базата от данни *model* съдържа системни таблици и системни изгледи, които се включват във всяка нова база от данни и се използват за дефиниране и поддържане на базата от данни. Изпълнението на команда на Transact-SQL или системна съхранена процедура, която създава, променя или изтрива обекти на SQL Server, води до промени в редовете на системните таблици.

Общият вид на T-SQL конструкцията за създаване на база от данни е:

```
CREATE DATABASE database_name
[ ON [PRIMARY]
  (
    [ NAME = logical_file_name, ]
    FILENAME = 'os_file_name'
    [, SIZE = size ]
    [, MAXSIZE = {max_size | UNLIMITED} ]
    [, FILEGROWTH = growth_increment ]
  ) [, ... ]
]
[, {FILEGROUP filegroup_name <file_spec> [, ...]} [, ...]
]
[ LOG ON <file_spec> [, ...] ]
```

Параметрите за дефиниране на характеристиките на файловете на базата от данни са:

NAME – логическо име за файла. По подразбиране се използва името на базата от данни.

FILENAME – физическо име на файла, т.е. съответното име на файл от операционната система, включително и пълния път на файла. По подразбиране главният файл с данни има име *database_name.mdf*, файлът дневник – *database_name_log.ldf* и се създават в подразбиращата се папка, както е указано при инсталирането на SQL Server.

SIZE – размер на файла. По подразбиране за главния файл с данни на новата база от данни се използва размерът на главния файл в базата от данни *model*. За всеки второстепенен файл и файл-дневник се използва по подразбиране размер 3MB.

MAXSIZE – максимален размер, до който файлът може да нараства. Ако не се зададе, той ще нараства до запълване на диска, т.е. размерът на файла, ще се смята за неограничен. Неограничено нарастване се задава и чрез ключовата дума UNLIMITED, вместо конкретно число.

FILEGROWTH – стъпка на нарастване на файла на базата от данни. Трябва да е цяло число. Стойност 0 означава отсъствие на нарастване; може да бъде зададена в MB, KB, GB, TB или проценти. Ако стъпката е зададена като процент, тя се определя като процент от размера на файла в момента на разширяването.

Пример 1 Почти всички възможни параметри на командата за създаване на база от данни имат подразбиращи се стойности, така че е възможно да се създаде база от данни чрез използване на проста форма като следната:

```
CREATE DATABASE MyDatabase
```

По този начин се създава база от данни с два файла с логически имена *MyDatabase* и *MyDatabase_log*, физически имена *MyDatabase.mdf* и *MyDatabase_log.ldf* в подразбиращата се папка.

Пример 2

```
CREATE DATABASE MyDatabase
ON
```

```
( NAME = MyDatabase_data,
  FILENAME = 'c:\MySQLdata\MyDatabase_data.mdf',
  SIZE = 3MB,
  MAXSIZE = UNLIMITED,
  FILEGROWTH = 1MB )
```

```
LOG ON
```

```
( NAME = MyDatabase_log,
  FILENAME = 'c:\MySQLdata\MyDatabase_log.ldf',
  SIZE = 1MB,
  MAXSIZE = UNLIMITED,
  FILEGROWTH = 1MB )
```

Пример 3

```
CREATE DATABASE Sales
ON PRIMARY
```

```
( NAME = Sales_data,
  FILENAME = 'd:\dir\subdir\Sales_data.mdf',
  SIZE = 10 MB,
  MAXSIZE = 50 MB,
```

```

        FILEGROWTH = 5 MB ),
    ( NAME = Sales_data2,
      FILENAME = 'd:\dir\subdir\Sales_data2.ndf',
      SIZE = 10 MB,
      MAXSIZE = 50 MB,
      FILEGROWTH = 5 MB )
LOG ON
    ( NAME = Sales_log,
      FILENAME = 'd:\dir\subdir\Sales_log.ldf',
      SIZE = 10 MB,
      MAXSIZE = 25 MB,
      FILEGROWTH = 5 MB )

```

Файлови групи за база от данни

Възможно е групиране на файловете с данни във файлови групи за целите на разполагането и администрирането. Типовете файлови групи са:

- главна файлова група – съдържа главния файл с данни и всички файлове, които не са поставени в друга файлова група. Всички страници от системните таблици се разполагат във файловете от главната файлова група. Един файл в групата става главен файл с данни, който съдържа логическото начало в базата от данни и нейните системни таблици. Ако ключовата дума PRIMARY не е използвана, първият изброен списък от файлове в конструкцията образува главна файлова група.
- дефинирана от потребителя файлова група – може да се създават дефинирани от потребителя файлови групи, като се използва ключовата дума FILEGROUP в командите CREATE DATABASE или ALTER DATABASE.
- подразбираща се файлова група – съдържа страниците на всички таблици и индекси, които не са поставени в някоя конкретна файлова група. При първоначално създаване на базата от данни главната файлова група е подразбираща се. Може да се определи създадена от потребителя файлова група като подразбираща се с помощта конструкцията:

```

ALTER DATABASE database_name
MODIFY FILEGROUP filegroup_name DEFAULT

```

Пример 4 В този пример е показано създаване на база от данни с три файлови групи – главна файлова група с файлове *Sales_data* и *Sales_data2*; файлова група *SalesGrp1* с файлове *SalesGrp1_data1*, *SalesGrp1_data2*, *SalesGrp1_data3*; файлова група *SalesGrp2* с файлове *SalesGrp2_data1* и *SalesGrp2_data2*.

```

CREATE DATABASE Sales
ON PRIMARY
    ( NAME = Sales_data,
      FILENAME = 'c:\MySQLdata\Sales_data.mdf',
      SIZE = 10 MB,
      MAXSIZE = 50 MB,
      FILEGROWTH = 15 % ),
    ( NAME = Sales_data2,
      FILENAME = 'c:\MySQLdata\Sales_data2.ndf',
      SIZE = 5 MB,
      MAXSIZE = 50 MB,
      FILEGROWTH = 10 % ),

```

```

FILEGROUP SalesGrp1
( NAME = SalesGrp1_data1,
  FILENAME = 'c:\MySQLdata\SalesGrp1_data1.ndf',
  SIZE = 2 MB,
  MAXSIZE = 10 MB,
  FILEGROWTH = 1 MB ),
( NAME = SalesGrp1_data2,
  FILENAME = 'c:\MySQLdata\SalesGrp1_data2.ndf',
  SIZE = 1 MB,
  MAXSIZE = 5 MB,
  FILEGROWTH = 1 MB ),
( NAME = SalesGrp1_data3,
  FILENAME = 'c:\MySQLdata\SalesGrp1_data3.ndf',
  SIZE = 1 MB,
  MAXSIZE = 5 MB,
  FILEGROWTH = 1 MB ),
FILEGROUP SalesGrp2
( NAME = SalesGrp2_data1,
  FILENAME = 'c:\MySQLdata\SalesGrp2_data1.ndf',
  SIZE = 2 MB,
  MAXSIZE = 10 MB,
  FILEGROWTH = 1 MB ),
( NAME = SalesGrp2_data2,
  FILENAME = 'c:\MySQLdata\SalesGrp2_data2.ndf',
  SIZE = 5 MB,
  MAXSIZE = 50 MB,
  FILEGROWTH = 10 % )
LOG ON
( NAME = Sales_log,
  FILENAME = 'c:\MySQLdata\Sales_log.ldf',
  SIZE = 5 MB,
  MAXSIZE = 25 MB,
  FILEGROWTH = 5 MB ),
( NAME = Sales_log2,
  FILENAME = 'c:\MySQLdata\Sales_log2.ldf',
  SIZE = 5 MB,
  MAXSIZE = 25 MB,
  FILEGROWTH = 1 MB )

```

Променяне на базата от данни

Конструкцията ALTER DATABASE се използва, за да се промени дефиницията на базата от данни по един от следните начини:

- да се добави един или повече файлове с данни към базата от данни;
- да се добави един или повече файлове-дневници към базата от данни;
- да се добави файлова група към базата от данни;
- да се отстрани файл или файлова група от базата от данни. Файл може да се отстрани само, ако е напълно празен. Отстраняването на файл води до изтриването му като файл на операционната система; отстраняването на файлова група води до отстраняването на всички файлове от нея.
- да се промени съществуващ файл по един от следните начини:
 - да се увеличи стойността на параметъра SIZE;

- да се променят стойностите на параметрите NAME, FILENAME, MAXSIZE и FILEGROWTH.
- да се промени съществуваща файлова група по един от следните начини:
 - да се маркира файлова група като READONLY, така че да не се допуска изменение на обектите в нея. Главната файлова група не може да бъде маркирана като READONLY;
 - да се маркира файлова група като READWRITE, което позволява промени на обектите в нея;
 - да се маркира файлова група като подразбираща се за базата от данни.

Конструкцията ALTER DATABASE може при всяко изпълнение да изпълни само едно от изброените действия.

Общият вид на T-SQL конструкция за променяне на база от данни е:

```
ALTER DATABASE database_name
{  ADD FILE <file_spec> [, ...]
    [TO FILEGROUP filegroup_name]
  | ADD LOG FILE <file_spec> [, ...]
  | ADD FILEGROUP filegroup_name
  | REMOVE FILE logical_file_name
  | REMOVE FILEGROUP filegroup_name
  | MODIFY FILE <file_spec>
  | MODIFY FILEGROUP filegroup_name
    filegroup_property
}
```

Пример 5 Маркира файлова група *SalesGrp1* като подразбираща се:

```
ALTER DATABASE Sales
MODIFY FILEGROUP SalesGrp1 DEFAULT
```

Пример 6 Добавя файл с данни:

```
ALTER DATABASE Sales
ADD FILE
( NAME = New1_data,
  FILENAME = 'c:\MySQLdata\new1_data.ndf',
  SIZE = 5 MB,
  MAXSIZE = 100 MB,
  FILEGROWTH = 5 MB )
```

Пример 7 Добавя два файла с данни във файловата група *SalesGrp1*:

```
ALTER DATABASE Sales
ADD FILE
( NAME = New2_data,
  FILENAME = 'c:\MySQLdata\new2_data.ndf',
  SIZE = 5 MB,
  MAXSIZE = 100 MB,
  FILEGROWTH = 5 MB ),
( NAME = New3_data,
  FILENAME = 'c:\MySQLdata\new3_data.ndf',
  SIZE = 5 MB,
  MAXSIZE = 100 MB,
  FILEGROWTH = 5 MB )
```

```
TO FILEGROUP SalesGrp1
```

Пример 8 Добавя файл-дневник:

```
ALTER DATABASE Sales
ADD LOG FILE
( NAME = New1_log,
  FILENAME = 'c:\MySQLdata\new1_log.ldf',
  SIZE = 5 MB,
  MAXSIZE = 100 MB,
  FILEGROWTH = 5 MB )
```

Пример 9 Премахва файл-дневник:

```
ALTER DATABASE Sales
REMOVE FILE New1_log
```

Пример 10 Променя логическото име на файла с данни *New2_data*:

```
ALTER DATABASE Sales
MODIFY FILE
( NAME = New2_data,
  NEWNAME = SalesNew2_data )
```

Пример 11 Променя физическото име на файла с данни *New3_data*:

```
ALTER DATABASE Sales
MODIFY FILE
( NAME = New3_data,
  FILENAME = 'd:\MySQLdata\SalesNew3_data.ndf' )
```

Освен командата от пример 11, за да се осъществи преместването на файл на базата от данни, е необходимо:

1. базата от данни да се установи в режим *offline*:

```
ALTER DATABASE Sales SET OFFLINE
```
2. да се премести файлът на базата от данни на новото местоположение и/или да се преименува;
3. базата от данни да се установи в режим *online*:

```
ALTER DATABASE Sales SET ONLINE
```

Когато базата от данни е в режим *online*, тя е отворена и готова за използване.

Пример 12 Увеличава размера на файла с данни *New3_data*:

```
ALTER DATABASE Sales
MODIFY FILE
( NAME = New3_data,
  SIZE = 20 MB )
```

Пример 13 Маркира главната файлова група като подразбираща се:

```
ALTER DATABASE Sales
MODIFY FILEGROUP [PRIMARY] DEFAULT
```

Пример 14 Маркира файловата група *SalesGrp1* като READONLY:

```
ALTER DATABASE Sales
MODIFY FILEGROUP SalesGrp1 READONLY
```

За получаване на информация за една база от данни може да се използва както прозореца *Properties* на съответната база от данни в Management Studio, така и системните съхранени процедури `sp_helpdb` и `sp_helpfile`. Системната съхранена процедура `sp_helpdb` позволява да се извлече информация за всички бази от данни или за конкретна, името на която се подава като параметър. Например:

```
EXEC sp_helpdb
или
EXEC sp_helpdb 'MyDatabase'
```

Системната съхранена процедура `sp_helpfile` дава възможност да се получи информация за всички файлове на текущата база от данни или за конкретен файл, логическото име на който се задава като параметър. Например:

```
EXEC sp_helpfile
или
USE MyDatabase
EXEC sp_helpfile 'MyDatabase_data'
```

Извлича се информация за физическото местонахождение на съответния файл, файлова група, на която принадлежи, неговия размер, максималния му размер, стъпка на нарастване, предназначение (за данни или файл-дневник).

Изтриване на база от данни

Възможно е изтриване на една несистемна база от данни, в резултат на което файловете и техните данни се изтриват от диска на сървъра. Базата от данни се изтрива завинаги и не може да бъде възстановена без създадено предварително архивно копие. Използва се конструкцията `DROP DATABASE database_name` или командата *Delete* от контекстното меню на съответната база от данни в Management Studio.

Преместване на база от данни на SQL Server

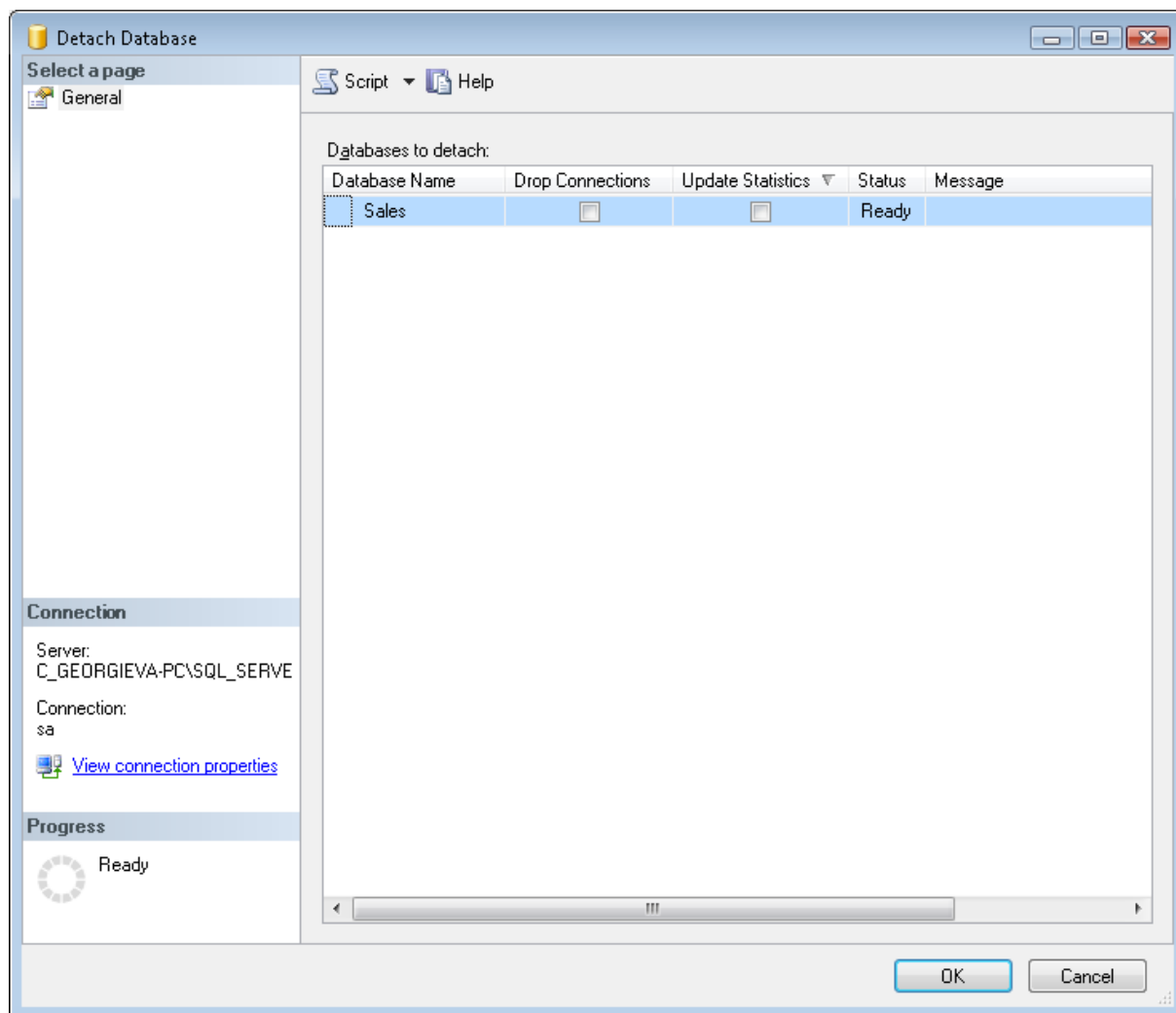
Способността за отделяне и присъединяване отново на базата от данни често е полезен начин, позволяващ създаването на преносима база от данни. За да се премести една база от данни на ново физическо устройство, се прилагат системните съхранени процедури `sp_detach_db` и `sp_attach_db`. Тези съхранени процедури могат да се използват, за да се направи копие на базата от данни за тестване или като алтернатива на командите за архивиране и възстановяване. Изпълнението на съхранената процедура

```
sp_detach_db [@dbname =] 'dbname'
```

отстранява базата от данни от сървъра. Файловете на отстранената база от данни още съществуват, но операционната система ги разглежда като затворени файлове, следователно могат да бъдат копирани, премествани или изтривани като други файлове на операционната система. Например:

```
EXEC sp_detach_db 'MyDatabase'
```

В SQL Server 2008 Management Studio отделянето на база от данни от сървъра се извършва чрез командата *Tasks | Detach ...* от контекстното меню на съответната база от данни (фиг. 2).



Фиг. 2 Отстраняване на база от данни от сървъра

Базата данни може да бъде присъединена отново със `sp_attach_db`:

```
sp_attach_db [@dbname =] 'dbname',
             [@filename1 =] 'filename_1' [, ...]
```

Ако всички файлове още съществуват на първоначалното си местонахождение (например, ако трябва само да се копира базата данни на друг сървър), всичко което е необходимо да се определи, е мястото на главния файл с данни. Той съдържа информация за мястото на всички файлове, принадлежащи на базата данни. Например:

```
EXEC sp_attach_db 'MyDatabase',
                  'c:\MySQLdata\MyDatabase_data.mdf'
```

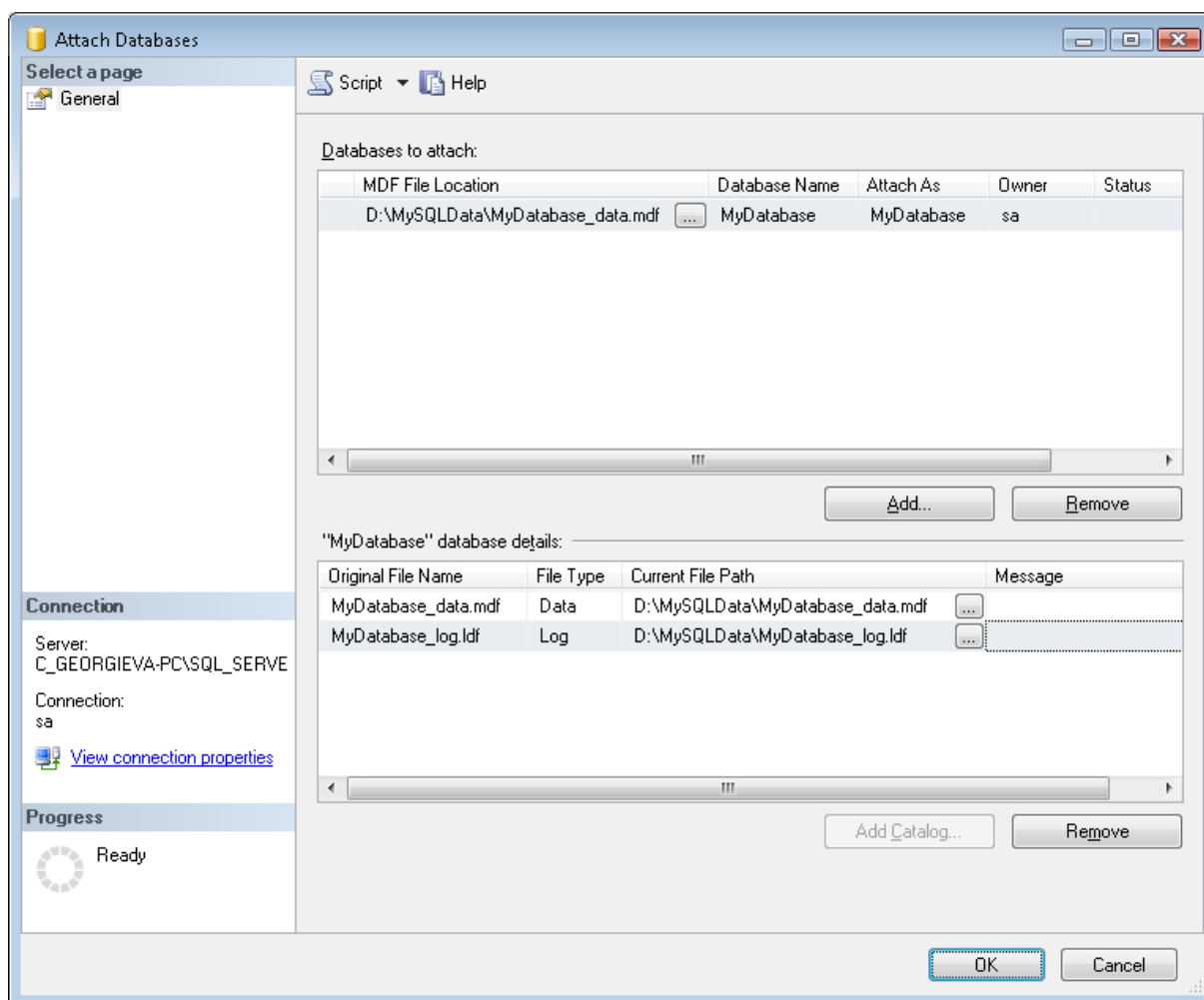
Друга възможност за присъединяване на базата от данни е, като се използва опцията `FOR ATTACH` на командата `CREATE DATABASE`:

```
CREATE DATABASE database_name
ON <file_spec> [ ,... ]
FOR ATTACH
```

Пример 15

```
CREATE DATABASE MyDatabase
ON
( FILENAME = 'd:\MySQLdata\MyDatabase_data.mdf' )
FOR ATTACH
```

В SQL Server 2008 Management Studio присъединяването на база от данни към сървъра се осъществява чрез командата *Attach ...* от контекстното меню на папката *Database* (фиг. 3).

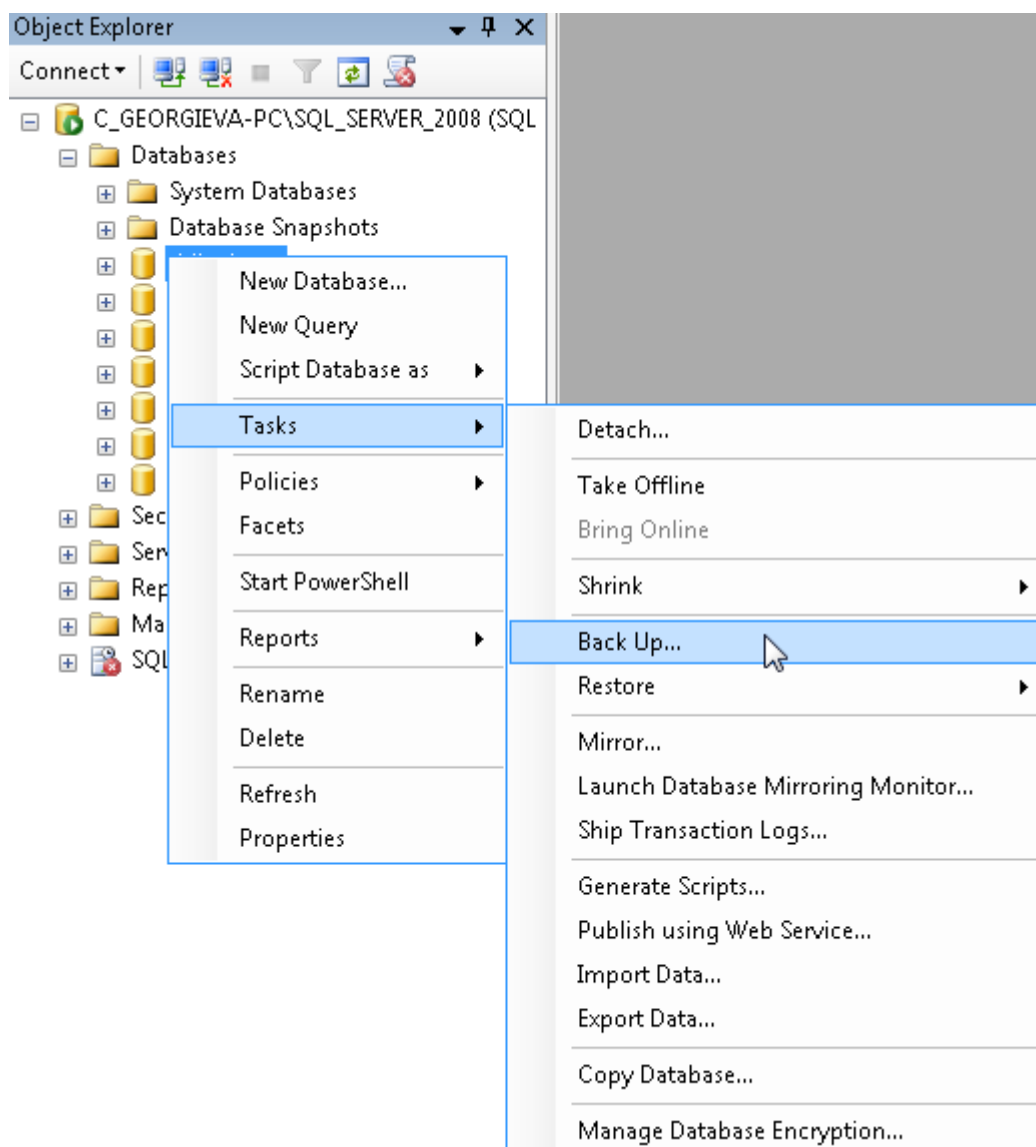


Фиг.3 Присъединяване на база от данни към сървъра

Архивиране и възстановяване на база от данни

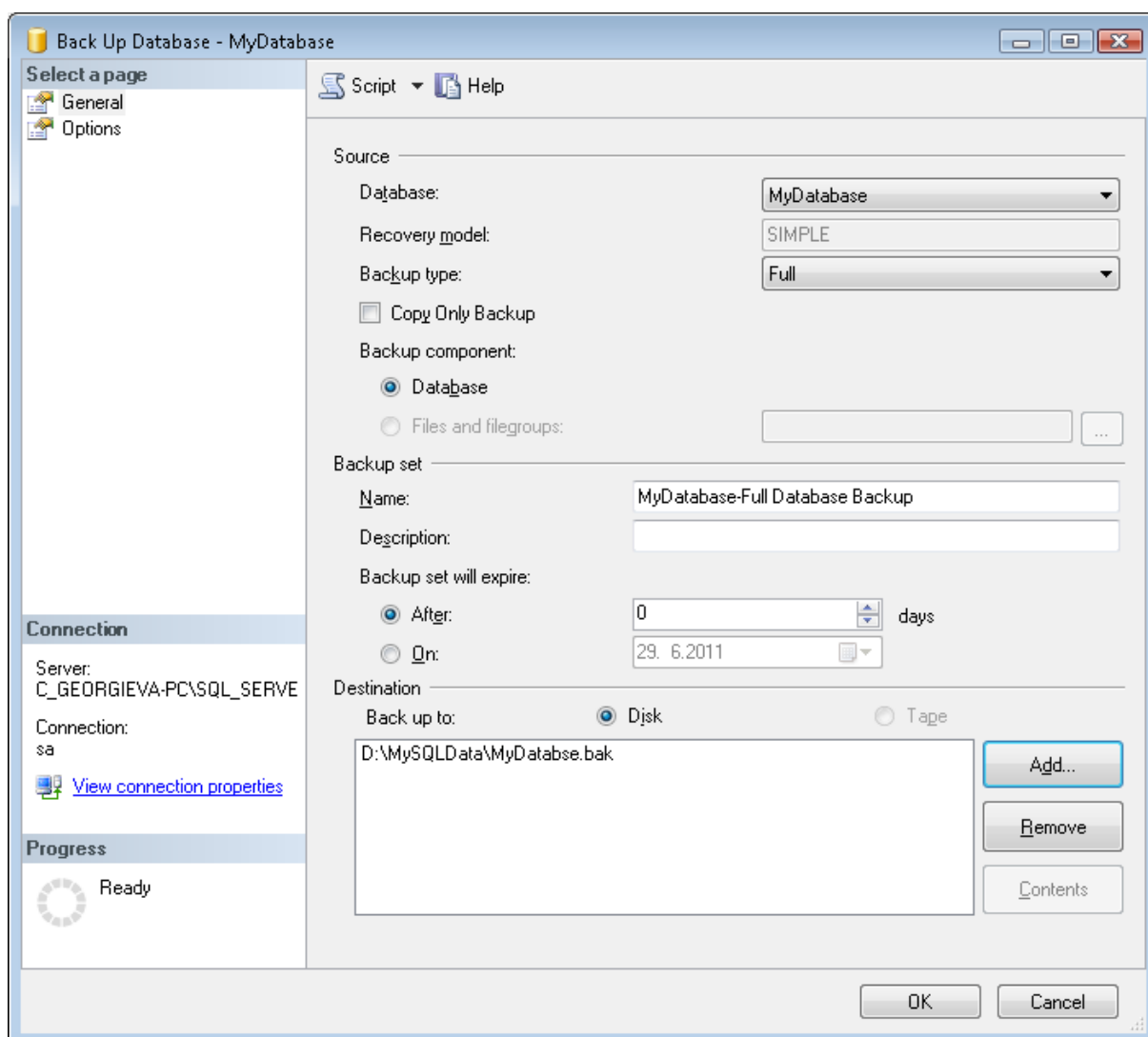
Архивирането и възстановяването на базата от данни от архивно копие позволява пълно възстановяване на данните при възникване на проблеми в системата. Архивирането на базата от данни прави копие на базата от данни, което може да бъде използвано за нейното възстановяване, ако тя бъде загубена. Архивирането копира всичко в базата от данни, включително и необходимите части от файла-дневник, който се използва по време на възстановителните операции на базата от данни за отхвърляне на недовършените транзакции и потвърждаване на завършилите.

В допълнение към възможността за възстановяване при загуба на данни, възстановяването на базата от данни от архивно копие е полезно при несистемни проблеми. Например преместване или копиране на базата от данни от един сървър на друг. Чрез архивиране на базата от данни на единия компютър и възстановяване на базата от данни на друг, копие на базата от данни може да се направи бързо и лесно чрез командите *Tasks / Back Up ...* и *Tasks / Restore* от контекстното меню на съответната база от данни в Management Studio (фиг. 4).



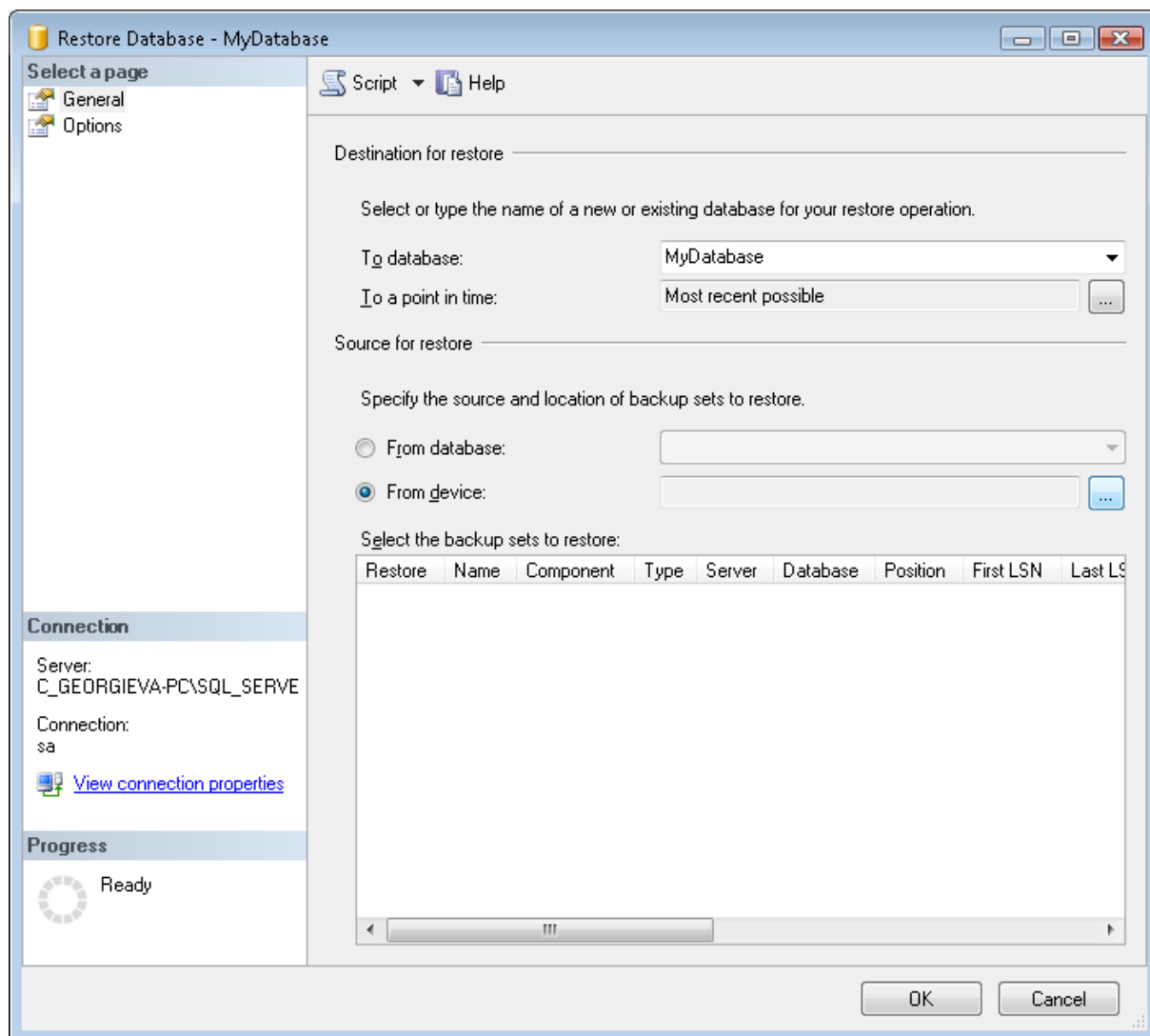
Фиг. 4 Командите за архивиране и възстановяване на база от данни в *Management Studio*

Например, нека е създадено актуално архивно копие на базата от данни MyDatabase, така както е показано на фигура 5.



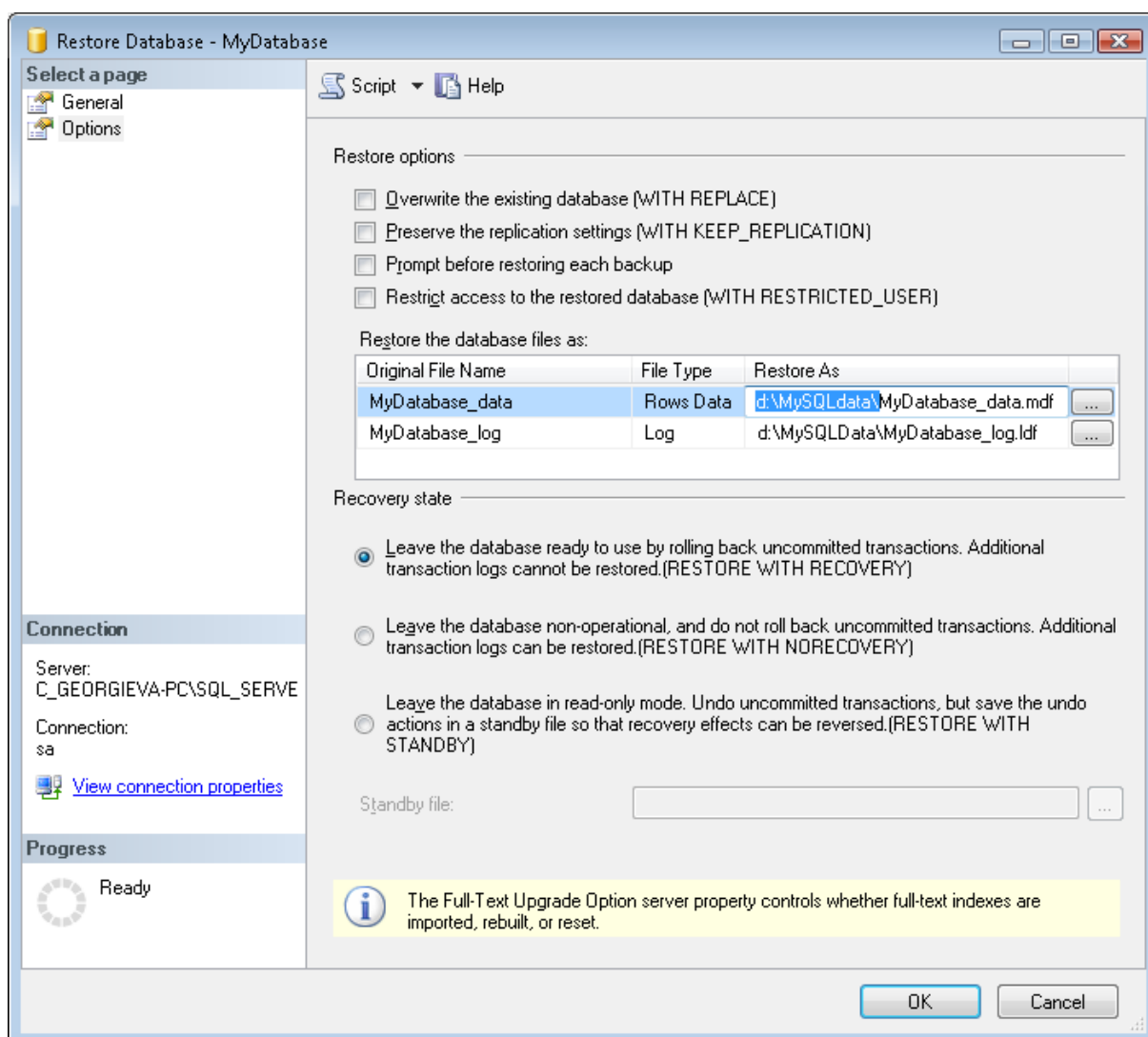
Фиг. 5 Архивиране на базата от данни *MyDatabase*

Ако допуснем, че базата от данни е изтрита и е необходимо да бъде отново възстановена или трябва да бъде преместена на друг сървър чрез създаденото архивно копие, от диалоговата рамка *Restore Database* се избира опцията *From device* (фиг. 6).



Фиг. 6 Избиране на опцията *From device*

В текстово поле *To database* се въвежда името на базата от данни. В страницата *Options* на диалоговата рамка *Restore Database* може изрично да се определи устройството, папката, в която да се създадат файловете на базата от данни, както и техните имена (фиг. 7).

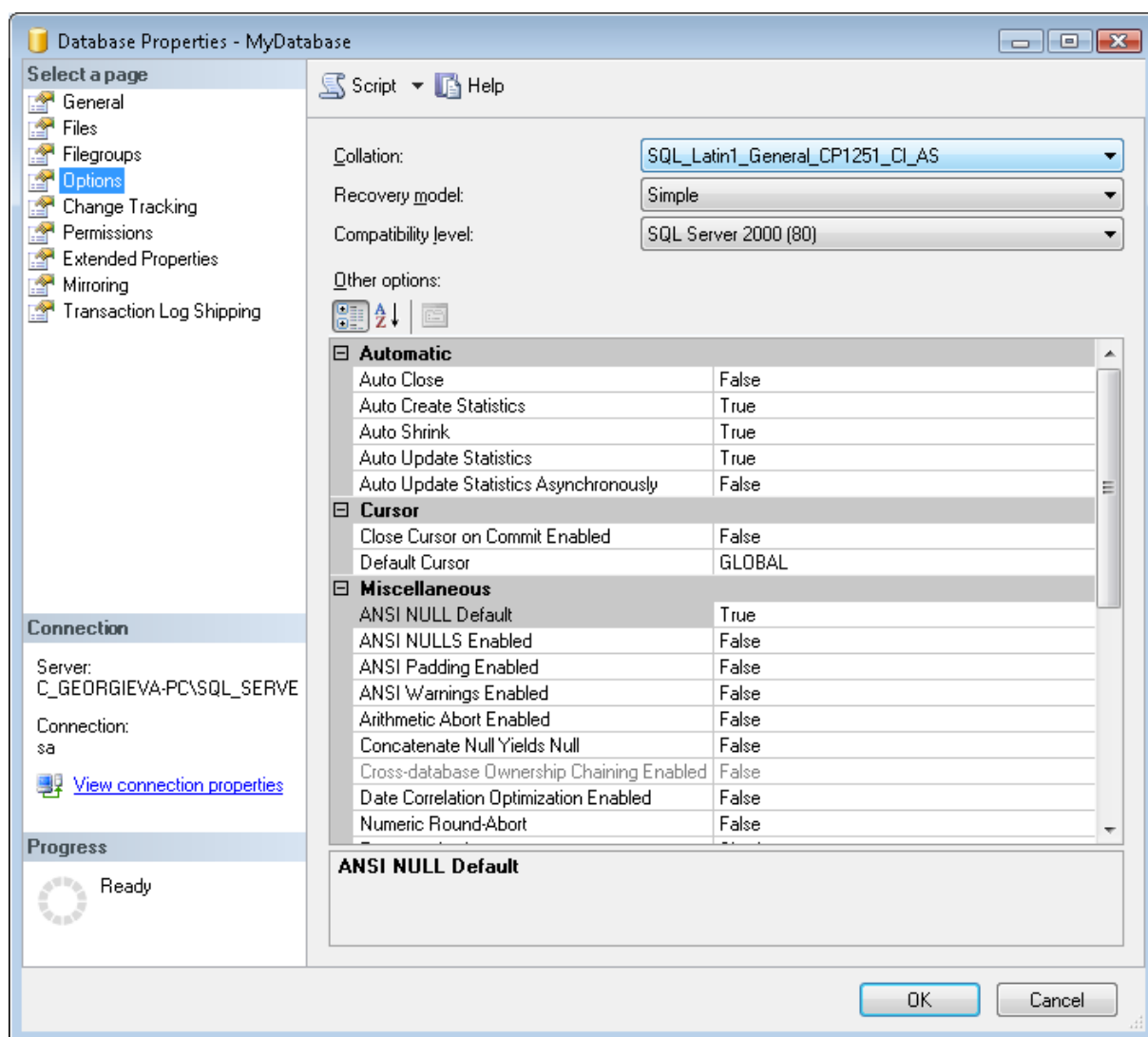


Фиг. 7 Определяне на имената на файловете на базата от данни, устройството и папката, в която да се създадат

Ако база от данни със същото име вече има на сървъра, тя може да бъде припокрита чрез включване на опцията *Overwrite the existing database* (фиг. 7).

Опции на база от данни на SQL Server

Опциите на базата от данни позволяват да се определи нейното поведение в различни ситуации.



Фиг. 8 Опции на базата от данни MyDatabase

Текущите им стойности могат да бъдат проверени чрез Management Studio, като се използва страницата *Options* на диалоговата рамка *Properties* на съответната база от данни (фиг. 8) или чрез системната съхранена процедура `sp_dboption`, включваща и опции, които не са достъпни от графичния интерфейс на Management Studio. Ако се изпълни тази съхранена процедура без параметри (т.е. `EXEC sp_dboption`), се извежда списък с всички опции, които могат да бъдат установени:

Settable database options:

```
-----
ANSI null default
ANSI nulls
ANSI padding
ANSI warnings
arithabort
auto create statistics
auto update statistics
autoclose
autoshrink
concat null yields null
cursor close on commit
```

dbo use only
 default to local cursor
 merge publish
 numeric roundabort
 offline
 published
 quoted identifier
 read only
 recursive triggers
 select into/bulkcopy
 single user
 subscribed
 torn page detection
 trunc. log on chkpt.

Следва списък с описание на значението на опциите:

- ANSI null default Когато е TRUE, подразбиращата се стойност на NULL опцията в конструкциите за създаване и променяне на таблица е NULL, т.е. разрешени са стойности NULL за съответната колона; в противен случай подразбиращата се стойност е NOT NULL.
- ANSI nulls Когато е TRUE, всяко сравнение със стойност NULL, използващо знака за равенство (=), връща резултат UNKNOWN (необходимо е да се използва само IS NULL); в противен случай сравнението на не-Unicode стойности със стойност NULL връща TRUE, ако и двете стойности са NULL.
- ANSI padding Когато е TRUE, винаги се запазват интервалите в края; в противен случай интервалите в края за колоните с променлива дължина (*varchar*, *varbinary*) се отрязват, както и тези с фиксирана дължина (*char*, *binary*), които позволяват NULL, т.е. запазват се само интервалите в края за колони с фиксирана дължина, недопускащи стойност NULL.
- ANSI warnings Когато е TRUE, се извеждат предупреждения в случаи, които според ANSI стандарта се считат за грешки като например наличие на стойност NULL в колоната, за която се прилага обобщаваща функция и други.
- arithabort Когато е TRUE, препълване или грешка при деление на нула предизвикват прекратяване на заявката или пакета. Ако грешката възникне в транзакция, транзакцията се превърта назад. Когато е FALSE, се извежда предупредително съобщение, но заявката, пакетът или транзакцията продължава изпълнението си.
- auto create statistics Когато е TRUE, автоматично се създава статистика за колоните, използвани в условията след WHERE на заявките. Това води до оптимизиране на заявките, тъй като оптимизаторът на заявки на SQL Server използва тази статистика, за да определи най-добрия начин за изпълнение на заявките.
- auto update statistics Когато е TRUE, периодично се обновява съществуващата статистика, за да се поддържа автоматично актуална след промяна на данните.
- autoclose Когато е TRUE, базата от данни се затваря и ресурсите се освобождават, когато се прекрати последната потребителска конекция и всички процеси са приключени. Базата от данни се отваря автоматично, когато някой потребител се опита да я използва отново. Когато е FALSE, базата от данни остава отворена дори и да няма потребители, които да я използват.

- `autoshrink` Когато е TRUE, файловете на базата от данни автоматично периодично се намаляват по размер. Тази опция предизвиква намаляване на файловете, ако над 25% от съответния файл представляват неизползвано пространство. Тогава файлът се намалява, така че само 25% от файловото пространство да е свободно или до началния си размер, като това се установява в зависимост от това кой размер е по-голям.
- `concat null yields null` Когато е TRUE, конкатенирането на низове, поне един от които има стойност NULL, връща като резултат NULL. В противен случай стойността NULL се разглежда като празен низ.
- `cursor close on commit` Когато е TRUE, отворените курсори се затварят автоматично, когато се потвърди транзакцията. В противен случай курсорите се затварят, когато курсорите бъдат затворени изрично или когато се затвори съответната конекция.
- `dbo use only` Когато е TRUE, само собственикът на базата от данни има достъп до нея. Използва се, когато ще се променя базата от данни (например дефиницията на някои обекти в нея) и е необходимо временно да се ограничи достъпът до нея.
- `default to local cursor` Когато е TRUE, по подразбиране курсорите имат локален диапазон; в противен случай – глобален. Диапазонът на курсора може изрично да се зададе в дефиницията му.
- `merge publish` Когато е TRUE, базата от данни може да бъде използвана за публикации при репликации със сливане. Тази опция се променя автоматично при задаване на репликация.
- `numeric roundabort` Когато е TRUE, се генерира грешка, когато възникне загуба на точност в израз. Когато е FALSE, загубата на точност не генерира съобщение за грешка и резултатът се закръгля до точността на колоната или променливата, предназначена да съхранява резултата.
- `offline` Когато е TRUE, базата от данни е затворена и маркирана като офлайн. Използва се при разполагане на базата от данни върху преносим носител като например DVD.
- `published` Когато е TRUE, данни от базата от данни могат да бъдат публикувани за репликация. Тази опция се променя автоматично при задаване на репликация.
- `quoted identifier` Когато е TRUE, идентификаторите могат да се ограждат с кавички или квадратни скоби, символните низове с апострофи; в противен случай идентификаторите могат да се ограждат с квадратни скоби, символните низове с апострофи или с кавички.
- `read only` Когато е TRUE, не могат да се променят данни в базата от данни, но могат да бъдат четени.
- `recursive triggers` Когато е TRUE, тригерите могат да се стартират рекурсивно.
- `select into/bulkcopy` Когато е TRUE, се позволява изпълнението на команди, които не се записват в дневника на транзакциите. Тези команди са SELECT INTO спрямо постоянна таблица, BULK INSERT, *bcp* за импортиране на голямо количество данни, WRITETEXT, UPDATETEXT.
- `single user` Когато е TRUE, само един потребител може да се свърже с базата от данни в даден момент.
- `subscribed` Когато е TRUE, базата от данни може да бъде абонирана за публикация. Тази опция се променя автоматично при задаване на репликация.
- `torn page detection` Когато е TRUE, SQL Server автоматично открива незавършени I/O операции, в резултат на което се извежда съобщение за грешка и се преустановява потребителската конекция.

`trunc. log on chkpt.` Когато е `TRUE`, SQL Server автоматично отрязва дневника на транзакциите, след като транзакциите вече са потвърдени.

За да се промени текущата стойност на някоя от опциите, се използва пълният синтаксис на процедурата:

```
sp_dboption [[@dbname =] 'database_name']
            [, [@optname =] 'option_name']
            [, [@optvalue =] 'value']
```

Пример 16

```
EXEC sp_dboption 'MyDatabase', 'Single user', TRUE
```

Пример 17 Ако се пропусне задаване на нова стойност на параметър, се извежда текущата му стойност. Изпълнението на следната конструкция:

```
EXEC sp_dboption 'MyDatabase', 'Single user'
```

ще изведе следния резултат:

OptionName	CurrentSetting
-----	-----
single user	on

Пример 18 Ако се пропусне задаване на опция, се извежда списък с опциите на съответната база от данни, които са включени. При изпълнението на процедурата по следния начин:

```
EXEC sp_dboption 'MyDatabase'
```

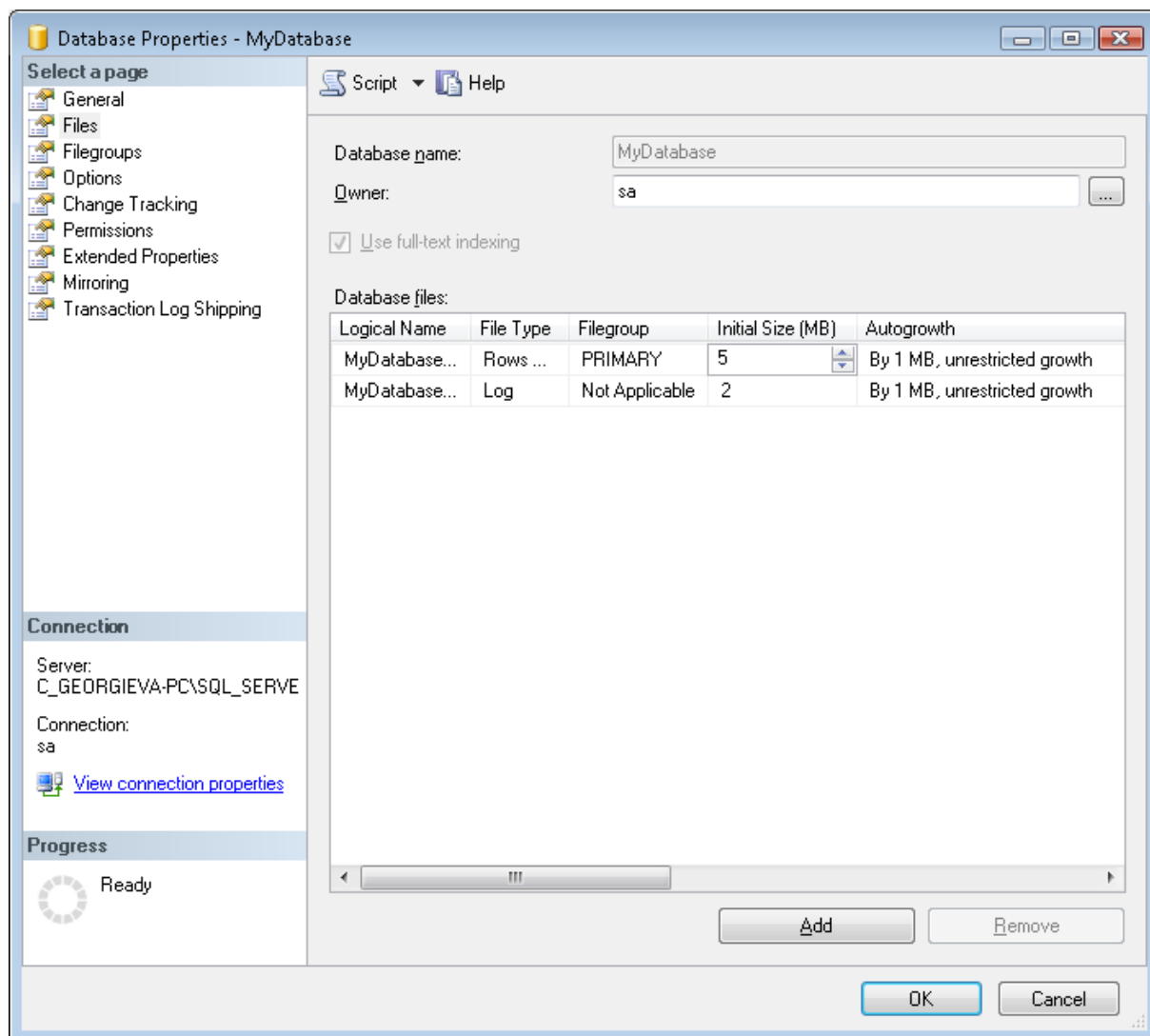
се връща резултат от вида:

```
The following options are set:
```

```
-----
trunc. log on chkpt.
single user
ANSI null default
autoshrink
auto create statistics
auto update statistics
```

Разширяване на база от данни

Разширяването на една база от данни може да се извърши, като се използва Management Studio (фиг. 9).



Фиг. 9 Разширяване на база от данни чрез увеличаване на стойността в текстовото поле *Initial Size (MB)* и/или добавяне на файлове на базата от данни

Конструкцията `ALTER DATABASE` позволява увеличаване на размера на някой от съществуващите файлове на базата от данни и добавяне на допълнителни файлове на базата от данни.

Разширяването на базата от данни може да се извършва автоматично, когато някой от файловете на базата от данни се запълни. Новият размер на файла в този случай се определя от зададената стъпка на нарастване и размера на файла в момента на разширяването.

Свиване на база от данни

Свиването на базата от данни може да се постигне, като се свият отделни файлове или всички файлове на базата от данни. Съществуват три начина за свиване на базата от данни: командите `DBCC SHRINKDATABASE` или `DBCC SHRINKFILE`, Management Studio и опцията `autoshrink`.

- Свиване на база от данни чрез използване на T-SQL

Командата `DBCC SHRINKDATABASE` свива всички файлове в базата от данни (`DBCC` – database consistency checker). Тя има следния синтаксис:

```
DBCC SHRINKDATABASE
```

```
( database_name [, target_percent]
  [, {NOTRUNCATE | TRUNCATEONLY}] )
```

SQL Server сбива данните, като ги премества към началото на файла. По подразбиране освободеното файлово пространство се връща на операционната система. Чрез тази команда не е възможно някой файл на базата от данни да бъде свит до размер по-малък от началния размер на файла (установен при създаването на базата от данни) или размера, до който файлът е бил изрично увеличен (чрез ALTER DATABASE) или намален (чрез DBCC SHRINKFILE). Задават се следните аргументи:

database_name е логическото име на базата от данни, която трябва да се свие.

target_percent е процента на свободното пространство, което трябва да бъде оставено във всеки файл на базата от данни.

Чрез NOTRUNCATE се указва да се остави освободеното файлово пространство във файловете, а не да се връща на операционната система.

Чрез TRUNCATEONLY се указва връщане на всяко неизползвано пространство във файловете с данни на операционната система. Когато се използва тази опция, параметърът *target_percent* се игнорира.

Пример 19

```
DBCC SHRINKDATABASE ( MyDatabase, 10, NOTRUNCATE )
```

Командата DBCC SHRINKFILE свива отделен файл в текущата база от данни.

Тя има следния синтаксис:

```
DBCC SHRINKFILE
( {file_name | file_id}
  { [, target_size]
    | [, {EMPTYFILE | NOTRUNCATE | TRUNCATEONLY}] } )
```

Параметрите *file_name* и *file_id* се използват за определяне на отделен файл на базата от данни. Стойността на *file_id* може да се установи от изпълнението на системната съхранена процедура *sp_helpdb*.

Опционалният параметър *target_size* определя желания размер на файла в MB. Ако не е зададен, файлът се свива толкова, колкото е възможно.

EMPTYFILE указва преместване на всички данни от определения файл в другите файлове от същата файлова група. Тази опция позволява файлът да бъде изтрит впоследствие, като се използва конструкцията ALTER DATABASE.

Опциите NOTRUNCATE и TRUNCATEONLY имат аналогично на предишната конструкция значение.

Пример 20

```
DBCC SHRINKFILE ( MyDatabase_data )
```

Пример 21

```
DBCC SHRINKFILE ( 2 )
```

Пример 22 Нека в базата от данни е добавен файл с данни:

```
USE MyDatabase
GO
ALTER DATABASE MyDatabase
ADD FILE
( NAME = MyDatabase_dat1,
  FILENAME = 'c:\MySQLdata\MyDatabase_dat1.ndf' )
```

След това във файла се добавят данни и ако впоследствие се наложи да бъде изтрит, е необходимо да се изпълни следната конструкция:

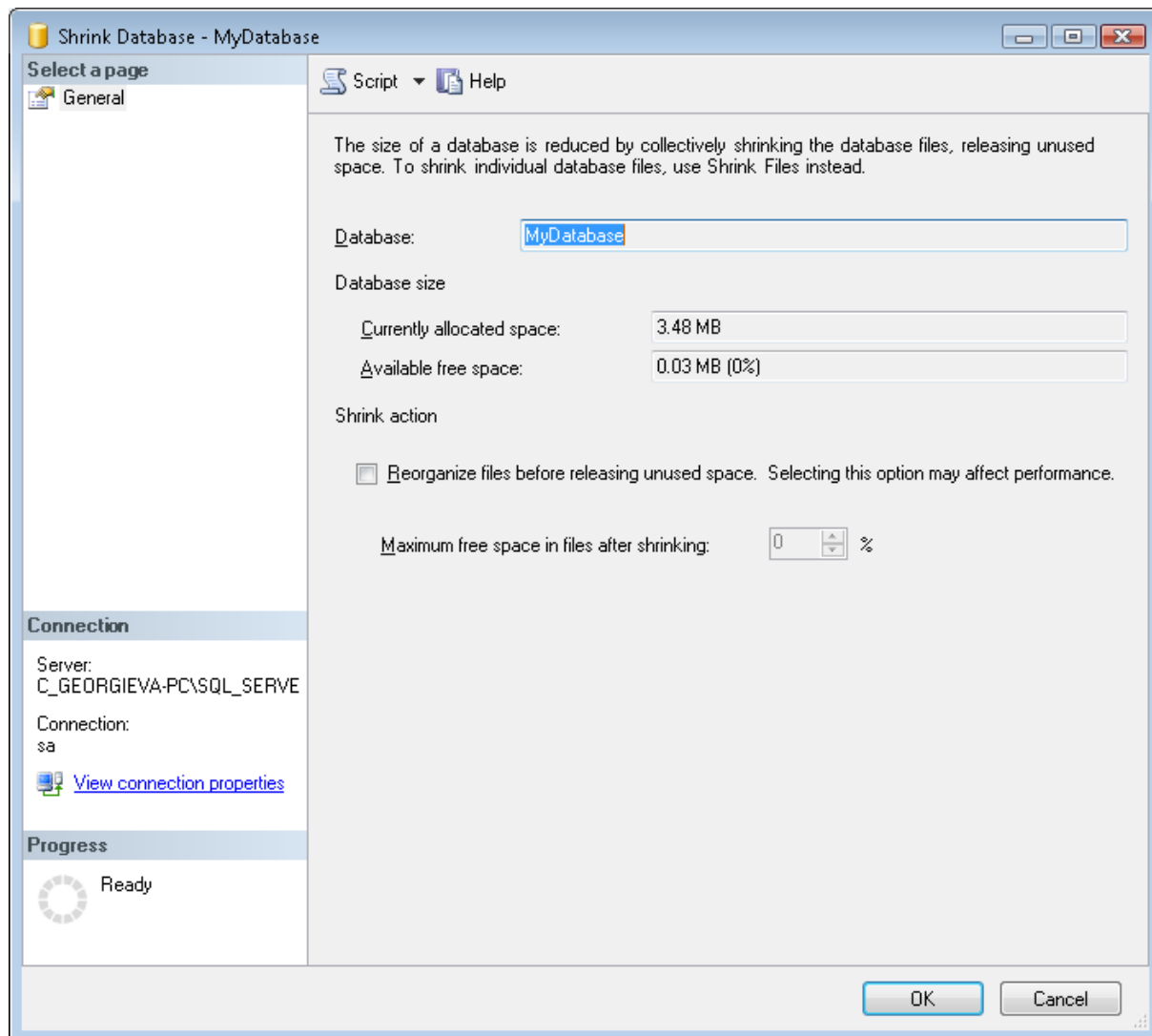
```
DBCC SHRINKFILE ( MyDatabase_data1, EMPTYFILE )
```

Файлът вече е празен и може да бъде отстранен:

```
ALTER DATABASE MyDatabase  
REMOVE FILE MyDatabase_data1
```

- Свиване на база от данни чрез използване на Management Studio

Използва се командата *Tasks / Shrink / Database* от контекстното меню на съответната база от данни (фиг. 10).

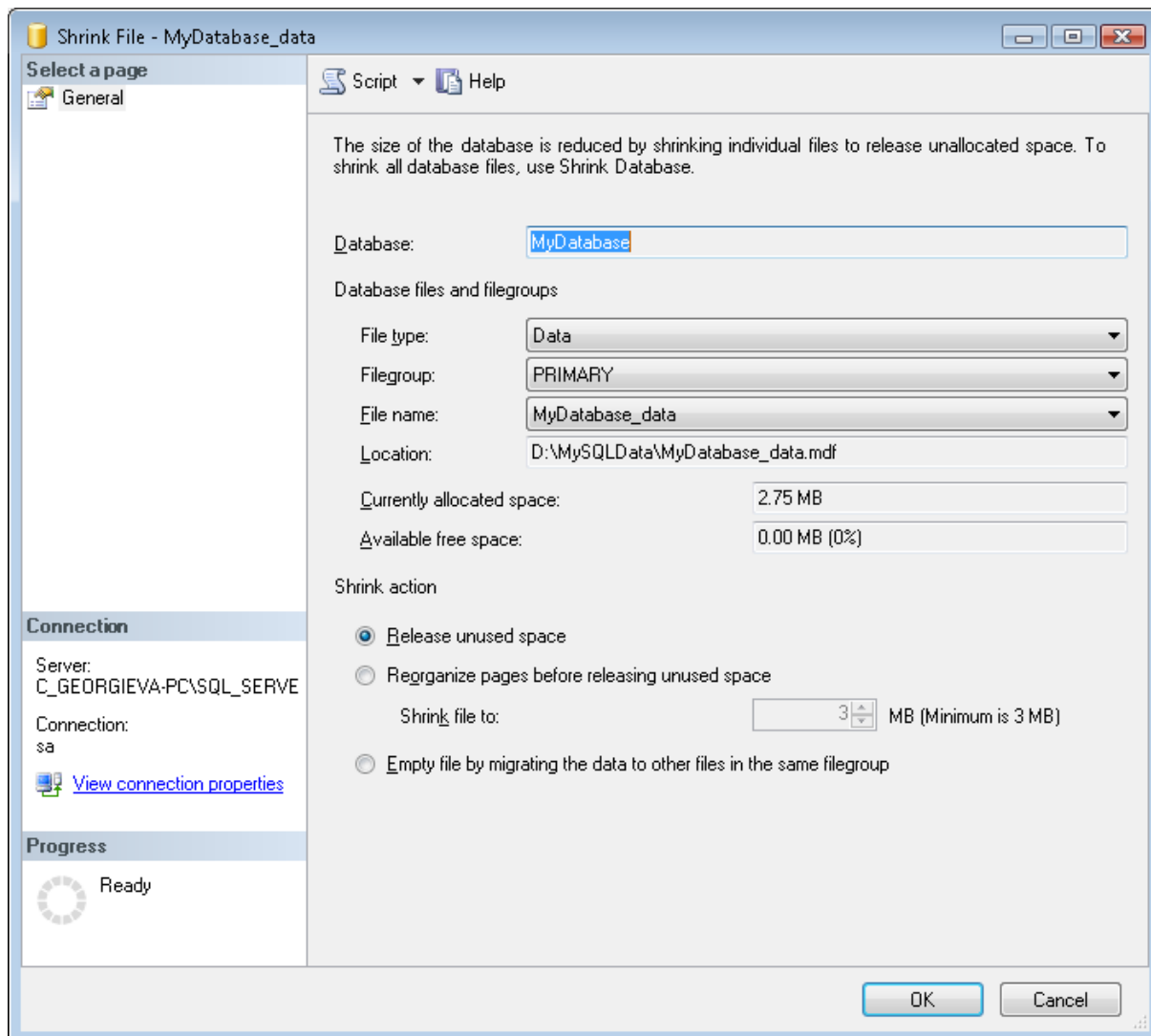


Фиг. 10 Свиване на база от данни

Диалоговата рамка *Shrink File* (фиг. 11), която се отваря при избиране на командата *Tasks / Shrink / Files* от контекстното меню на съответната база от данни, позволява да се зададе свиване на отделен файл на базата от данни. Възможно е да се зададе:

- освобождаване на неизползваното пространство чрез опцията *Release unused space* (фиг. 11);

- реорганизиране на страниците с данни, така че да бъдат преместени в началото на файловете с данни чрез опцията *Reorganize pages before releasing unused space* (фиг. 11);
- график за периодично свиване на файла (файловете) на базата от данни чрез избиране на бутона *Script | Script Action to Job* (фиг. 10 и 11);



Фиг. 11 Свиване на файл на базата от данни

Свиването на една база от данни може да се изпълнява от системния администратор и от собственика на базата от данни.

Преименуване на база от данни

Осъществява се чрез командата *Rename* от контекстното меню на съответната база от данни или чрез `ALTER DATABASE` по следния начин:

```
ALTER DATABASE database_name
MODIFY NAME = new_database_name
```

Пример 23

```
ALTER DATABASE MyDatabase
MODIFY NAME = new_name
```

Задачи

Задача 1. Да се създаде нова база от данни.

Задача 2. Да се преместят файловете, които обхваща създадената в задача 1 база от данни в друга папка.

Задача 3. Да се създаде архивно копие на базата от данни, създадена в задача 1.

Използване на SQL за дефиниране на данни

SQL Server използва стандартен синтаксис ANSI SQL на командите за създаване, променяне и изтриване на таблица. SQL Server Management Studio предоставя визуално средство за проектиране на таблици, използващо попълващи се формуляри, след което командите на SQL биват изпращани на SQL Server, за да се създаде или промени таблица. Освен това този инструмент позволява дефиниране на таблица чрез директно писане и изпълнение на SQL код. Дори когато се използва графичният интерфейс, е важно да може по-късно отново да се създаде таблицата. Затова е добре да се съхраняват всички DDL (*Data Definition Language* – език за дефиниране на данни) команди в скриптове, така че да може лесно да бъдат изпълнени по-късно за пресъздаване на таблицата. Чрез SQL Server Management Studio може да се създаде файл с DDL команди на SQL, които са необходими за повторно създаване на обекта. За целта се използва командата *Script Table as* от контекстното меню на съответната таблица на базата от данни.

Типове данни

Типът на данните е характеристика, която посочва каква информация може да се съхранява в дадена колона, параметър или променлива. SQL Server предоставя набор от системни типове на данните. Освен това може да се създадат и дефинират от потребителя типове на данните, които са базирани на предоставените от системата типове на данните.

Предоставени от системата типове данни

SQL Server използва различни типове данни, които могат да се групират в осем по-обща категории:

- Binary (двоични) – типовете *binary*, *varbinary*, *image* са за съхраняване на двоични низове. Типът *binary(n)* е с фиксирана дължина, докато типовете *varbinary(n)* и *image* са с променлива дължина. Типовете *binary* и *varbinary* позволяват да се съхранява до 8000 байта данни, а типовете *varbinary(max)* и *image* се използват за съхраняване на изключително големи по обем стойности, по-големи от 8000 байта и максимална дължина $2^{31}-1$ байта (до 2GB). Примери за двоични данни са графични и приложни файлове с данни като текстови документи и електронни таблици.
- Character (символни) – типовете *char*, *varchar*, *text* съхраняват буквено-цифрова информация. Типът *char* е с фиксирана дължина, докато *varchar* и *text* са с променлива дължина. Типът *char(n)* се използва, ако се очаква всички стойности да имат приблизително един и същ размер *n*. Типът *varchar(n)* е подходящ, ако полето може да съдържа стойност NULL или ще има низове с различна дължина в широки граници. Типът *varchar(max)* показва, че максималната дължина на съхраняваните низове е $2^{31}-1$ символа. Типът *text* е предназначен за съхраняване на извънредно големи символни низове, по-големи от 8000 символа и максимална дължина $2^{31}-1$ символа.
- Unicode character – типовете *nchar*, *nvarchar*, *ntext* съхраняват Unicode символи. Тъй като Unicode символите заемат два пъти повече памет от стандартните символи, всеки един от тези типове съхранява само половината от техните стандартни съответствия. Типът *ntext* се използва за съхраняване на стрингове по-големи от 4000 Unicode символи и максимална дължина $2^{30}-1$ символа.

Символните типове данни в повечето случаи трябва да съхраняват само данни, които не могат да се съхраняват като числа, но има някои изключения. Телефонни

номера, пощенски код, ЕГН, социалния осигурителен номер са числови стойности, но опитът показва, че е най-удобно, те да бъдат съхранени като символни типове данни. Тъй като тези числа имат определено форматиране, често е по-лесно да се съхраняват като форматиранни символни низове, отколкото да се форматира отново всеки път, когато се извеждат на екрана.

Стандартните символни данни съдържат 255 различни символа и изискват един байт за съхраняване на всички символи. Unicode символите изискват два байта за съхраняване и съдържат достатъчно символи, за да поддържат най-много от световните езици. Стандартните символи изискват по-малко място за съхраняване и предлагат по-голяма съвместимост с други системи. Следователно трябва да се използва Unicode тип данни само, когато е необходимо съхраняване на международен текст.

При формат с фиксирана дължина се определя точно количеството на заделеното за колоната пространство. При формат с променлива дължина се задава максималният размер, но по-къси данни ще заемат по-малко място. Те изискват малко повече преработка от колоните с фиксирана дължина, поради поддържането на тяхната непостоянна дължина. Затова е необходимо да се направи компромис между заемано пространство и допълнителна обработка. Колоните, които изискват по-малко от 20 символа, е по-добре да са с фиксирана дължина, тъй като за да оправдаят големия преразход, свързан с тяхната преработка, колоните трябва да са с по-голяма дължина.

- **Date&Time** (за дата и час) – типовете *datetime*, *smalldatetime* и *datetime2* съхраняват стойности за дата и час. Разликата между тях е точността на времето и областта на датите, които всеки тип може да съхранява. Типът *smalldatetime* е точен до една секунда, типът *datetime* е точен до 3.33 милисекунди, докато *datetime2* е точен до 100 наносекунди. Типът *datetimeoffset* е предназначен за съхраняване на дата и час, като отчита часовите зони; типът *date* съхранява само дата; типът *time* съхранява само часа.
- **Exact Numeric** (точни десетични стойности) – типовете *decimal*, *numeric*, *money*, *smallmoney* съхраняват точно десетичните числа. Типът данни *numeric* е синоним на *decimal*. По следния начин се задава тип с фиксирана точност и размер: `decimal[(p[, s])]`, където *p* е фиксирана точност, т.е. максималния общ брой на десетичните цифри, които могат да се съхраняват от двете страни на десетичната точка; *s* е размер, т.е. максималния брой на цифрите, които могат да бъдат съхранени от дясната страна на десетичната точка. Максималната стойност на *p* е 38, т.е. $1 \leq p \leq 38$ и $0 \leq s \leq p$. Типовете *money* и *smallmoney* са парични типове с точност до десетохилядната за парична единица.

Numeric данните са много по-ефективни за съхраняване от символните, следователно винаги, когато е възможно да бъдат представени данни от колона като число, трябва да се използва числов тип данни (с изключение на вече споменатите случаи). Трябва да се внимава с късите символни кодове за ключови полета. Този тип идентифициращ код е използван в общоприложните среди, където е по-лесно операторът да разчете мнемоничния символен код, отколкото да използва цяло число. В съвременните клиент/сървър среди обаче данните много често се представят на потребителя чрез графичен интерфейс, който може да преведе целочисленото поле на разбираем текст. Затова е по-добре да се използва по-ефективният целочислен код, който е свързан с превеждаща таблица, съдържаща описателно поле, отколкото да се използва символен код за ключовите полета.

- **Approximate Numeric** (за числа с плаваща точка) – типовете *float* и *real* дават възможност за съхраняване на приближение на десетичните числа. Основната разлика между двата типа е областта от стойности (*float* позволява двойна точност). Данните от тип *float* представляват число с плаваща точка от

-1.79E+308 до 1.79E+308. Данните от тип *real* представляват число с плаваща точка от -3.40E+38 до 3.40E+38.

- Integer (цели числа) – типовете *bit*, *bigint*, *int*, *smallint*, *tinyint* съхраняват целочислени стойности в различни области. Типът *bit* е целочислен тип данни с две възможни стойности 0 или 1; типът *bigint* е между -2^{63} и $2^{63}-1$; типът *int* е между -2^{31} и $2^{31}-1$; типът *smallint* е между -2^{15} и $2^{15}-1$; типът *tinyint* е между 0 и $255 = 2^8-1$.
- CLR (*Common Language Runtime*) типове данни:
 - *hierarchyid* – тип данни, който може да съхранява стойности, представляващи позиции в йерархичната структура. Този тип данни използва вградени методи за създаване и манипулация на йерархични върхове. Стойност от тип *hierarchyid* представя позиция в йерархия. Колоната от този тип не представя автоматично структурата дърво. Необходимо е приложението да генерира и асоциира стойности по такъв начин, че да те отразяват йерархичните връзки между редовете.
- Spatial data types (пространствени типове данни):
 - *geography* – тип данни, който се използва за съхраняване на геодезични данни и работа с тях;
 - *geometry* – тип данни, предназначен за съхраняване на равнинни данни и извършване на различни операции върху тях. Представя данните в двумерното Евклидово пространство. За типа данни *geometry* SQL Server 2008 поддържа множество от методи, дефинирани от стандарта Open Geospatial Consortium (OGC), както и допълнителни методи, разширяващи този стандарт.
- Other data types (други типове) – представляват данни, които не могат да се причислят към никоя от останалите категории данни:
 - *cursor* – този тип данни се използва за променливи, чрез които се поддържа референцията към курсора и може да се използва за работа с курсора.
 - *sql_variant* – този тип данни съхранява стойности от различни типове данни, поддържани от SQL Server, с изключение на *text*, *ntext*, *timestamp* и *sql_variant*. Максималната дължина на данните от този тип е 8016 байта.
 - *table* – този тип данни се използва за съхраняване на резултатен набор за по-късна обработка. Основната му употреба е за временно съхраняване на набор от редове, които биват върнати като резултат от дефинирана от потребителя функция.
 - *uniqueidentifier* – данните се състоят от 16-байтово шестнадесетично число, обозначаващо глобално уникален идентификатор (GUID). Езикът Transact-SQL поддържа системната функция *NewID()*, която може да се използва за генериране на стойност от този тип.
 - *timestamp* – (8-байтово число в двоичен формат) този тип данни осигурява уникална стойност за базата от данни всеки път, когато се актуализира или добавя ред. Тази стойност се гарантира да е уникална за базата от данни, но може евентуално да се дублира със стойност от друга база от данни. За разлика от стойност от тип *uniqueidentifier*, която трябва да се получава от функцията *NewID()*, колоните от тип *timestamp* се актуализират автоматично от системата, когато се вмъква или променя ред.
 - *xml* – тип данни за съхраняване на XML данни в колона на таблица или променлива.

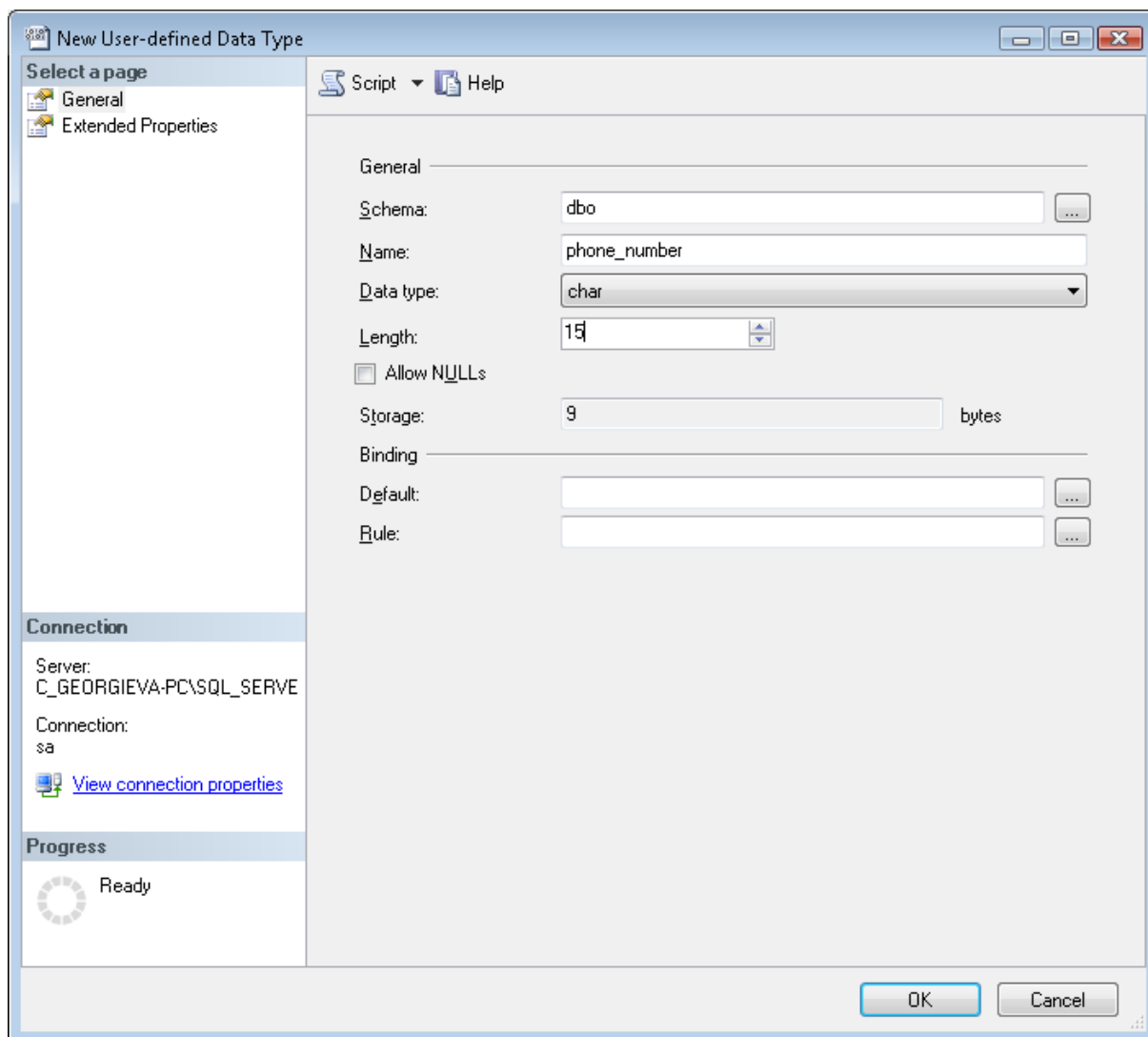
Дефинирани от потребителя типове данни

Дефинираните от потребителя типове данни (UDDTs – User Defined Data Types) се базират на системните типове данни в SQL Server. Те могат да се използват, когато няколко таблици трябва да съхраняват един и същ тип данни в дадени колони и трябва да се гарантира, че тези колони имат абсолютно същия тип данни, дължина и възможност да допускат стойност NULL, стойност по подразбиране, правила. Дефинираните от потребителя типове данни имат следните характеристики:

- основен тип – определен системен тип данни на SQL Server, върху който се базира UDDT;
- NULL опция – определя дали този тип на данните допуска стойност NULL;
- правила (*rules*) – UDDT могат да имат ограничаващи правила, които дефинират област от допустими стойности за колоната;
- стойност по подразбиране (*defaults*).

NULL опцията може да бъде припокрита, когато таблицата се създава или променя.

Дефинираните от потребителя типове данни могат да се добавят и изтриват чрез графичния интерфейс на Management Studio или като се изпълняват командите CREATE TYPE и DROP TYPE. При използване на Management Studio в съответната база от данни се маркира папката *Programmability | Types | User-Defined Data Types* и от контекстното меню се избира *New User Defined Data Type...*, в резултат на което се отваря диалоговия прозорец *New User Defined Data Type* (фиг. 1).



Фиг. 1 Създаване на дефиниран от потребителя тип данни

Синтаксисът на командите за създаване и изтриване на потребителски дефиниран тип данни има вида:

```
CREATE TYPE [schema_name.] type_name
{
    FROM base_type
    [ ( precision [ , scale ] ) ]
    [ NULL | NOT NULL ]
    | AS TABLE
    ( { <column_definition>
      | <computed_column_definition> }
      [ <table_constraint> ] [ ,... n ] ) }
```

```
DROP TYPE [schema_name.] type_name
```

Създаването на потребителски дефиниран тип данни изисква задаването на:

- *schema_name* е наименованието на схемата, към която UDDT принадлежи;
- *type_name* е наименованието на UDDT;
- *base_type* е системен тип данни на SQL Server, на базата на който се създава UDDT;

- *precision* се прилага за типовете *decimal* и *numeric* и представлява неотрицателно цяло число, което показва максималния общ брой на десетичните цифри, които могат да бъдат съхранявани отляво и отдясно на десетичната точка;
- *scale* се прилага за типовете *decimal* и *numeric* и представлява неотрицателно цяло число, което показва максималния брой на десетичните цифри, които могат да бъдат съхранявани отдясно на десетичната точка;
- NULL | NOT NULL определя дали типът допуска стойност NULL;
- <column_definition> дефинира колоните за табличен UDDT, т.е. потребителски дефиниран тип данни, създаден чрез ключовите думи AS TABLE;
- <computed_column_definition> дефинира изчислима колона за табличен UDDT;
- <table_constraint> дефинира ограничения на ниво таблица за табличен UDDT; допускат се ограниченията PRIMARY KEY, UNIQUE и CHECK.

Пример 1 Създаване на дефиниран от потребителя тип данни за колони, съхраняващи телефонни номера:

```
CREATE TYPE phone_number
FROM char(15) NOT NULL
```

Пример 2 Изтриване на съществуващ UDDT:

```
DROP TYPE phone_number
```

Пример 3 Създаване на табличен UDDT:

```
CREATE TYPE LocationTableType AS TABLE
( LocationName varchar(50),
  RegionName varchar(50),
  Country varchar(30) )
```

Създаване на таблица

Общ вид на SQL конструкцията за създаване на таблица:

```
CREATE TABLE
[database_name.[schema_name]. | schema_name.] table_name
( { col_name datatype [null_option]
  | col_name AS computed_col_expression
  } [, ...]
)
[ON {filegroup_name | [DEFAULT]}]
[TEXTIMAGE_ON {filegroup_name | [DEFAULT]}]
```

Параметри за дефиниране на характеристиките на таблицата и колоните в таблицата:

- Таблицата бива създадена в база от данни и принадлежи на една определена схема *schema_name*. Схемите на една база от данни осигуряват възможност за логическо групиране на обектите – таблици, изгледи, съхранени процедури и т.н. Една схема може да се разглежда като контейнер на обекти в базата от данни.

Пълното име на таблицата има вида *database_name.schema_name.table_name*. Подразбиращата се стойност за името на базата от данни е контекста на базата от данни, в която се работи в

момента. Името на таблицата трябва да отговаря на правилата за идентификатор – комбинация от 1 до 128 букви, цифри или символи #, \$, @ и _. Ако се използва заграден в квадратни скоби идентификатор (т.е. []), той може да съдържа ключови думи, интервали и други не цифрово-буквени знаци, които обикновено не се допускат. Идентификаторите могат да се заграждат в квадратни скоби във всяка среда, но за да се заграждат идентификаторите в кавички, е необходимо чрез `SET QUOTED_IDENTIFIER ON` да се включи специална опция (за ограждане на низове или константи за дати в този случай може да се използва само апостроф).

- Имената на колоните трябва да отговарят на правилата за идентификатори и да са уникални за таблицата.
- *datatype* определя един от стандартните SQL Server типове данни или дефиниран от потребителя тип данни.
- *computed_col_expression* представя математически израз, който може да се състави, като се използват имена на други колони в таблицата, вградени функции. За тази колона не се съхраняват никакви данни, а се изчисляват според използваната функция. Изчислимите колони могат да се използват точно както други колони с изключение на това, че не могат да им се задават ограничения и не могат да бъдат въвеждани стойности в тях директно чрез конструкции `INSERT` или `UPDATE`.
- *null_option* – използват се ключовите думи `NULL` или `NOT NULL`, с които се определя дали стойности `NULL` са позволени в колоната. Ако липсва *null_option*, SQL Server приема `NOT NULL`. Опцията `ANSI null default` определя подразбиращото се състояние – ако има зададена стойност `TRUE` (`EXEC sp_dboption database_name, 'ANSI null default', TRUE`), *null_option* по подразбиране ще е `NULL`.
- Опционалните параметри `ON` след дефиницията на колоните позволяват да се наложи създаването на таблицата или колоните от тип *text*, *ntext* и *image* в определена файлова група. Заради изключително големия размер на колоните от тип *text*, *ntext* и *image*, понякога е желателно да се разположат данните от тези колони в отделна файлова група.

Пример 4 Създаване на таблицата `MyTable` в базата от данни `MyDatabase` и подразбираща се схема `dbo`:

```
CREATE TABLE MyDatabase..MyTable
( ID int NOT NULL,
  FirstName varchar(25) NOT NULL,
  MiddleInitial char(1) NOT NULL,
  LastName varchar(25) NOT NULL,
  Age tinyint NOT NULL
)
ON [DEFAULT]
```

Пример 5 Използване на потребителски дефинирания тип данни `phone_number` при създаването на таблица:

```
CREATE TABLE Customers
( CustomerID smallint NOT NULL,
  CustomerName varchar(50) NOT NULL,
  Cust_phone phone_number )
```

Създаването на таблица чрез графичния интерфейс на Management Studio се извършва, като се маркира папката *Tables* на съответната база от данни и се избира *New Tables...* от контекстното меню. Задава се името на новата таблица, след което в прозореца за проектиране на таблици се въвежда необходимата информация. Например, дефиницията на таблицата *MyTable* е показана на фигура 2.

Column Name	Data Type	Allow Nulls
ID	int	<input type="checkbox"/>
FirstName	varchar(25)	<input type="checkbox"/>
MiddleInitial	char(1)	<input type="checkbox"/>
LastName	varchar(25)	<input type="checkbox"/>
Age	tinyint	<input type="checkbox"/>

Фиг. 2 Проектен изглед на примерната таблица *MyTable* в *Management Studio*

Променяне на вече създадена таблица

След създаването на таблица е възможно променяне на нейната структура. Общ вид на SQL конструкцията за променяне на таблица:

```
ALTER TABLE
[database_name.[schema_name]. | schema_name.] table_name
{
    ALTER COLUMN col_name new_datatype [null_option]
    | ADD { col_name datatype [null_option] } [, ...]
    | DROP { COLUMN col_name } [, ...]
}
```

Чрез тази команда може да се добавят, изтриват колони, променя типа или свойството *null_option* на съществуваща колона.

Пример 6 Добавяне на колона в таблицата *MyTable*:

```
ALTER TABLE MyDatabase..MyTable
    ADD EGN char(10) NULL
```

Пример 7 Променяне на съществуващата колона *MiddleInitial*, така че да може да приема стойност *NULL*:

```
ALTER TABLE MyDatabase..MyTable
    ALTER COLUMN MiddleInitial char(1) NULL
```

Пример 8 Изтриване на съществуващата колона *Age*:

```
ALTER TABLE MyDatabase..MyTable
    DROP COLUMN Age
```

За да се промени дефиницията на вече създадена таблица чрез графичния интерфейс на Management Studio, се избира командата *Design* от контекстното меню на съответната таблица. Например, проектният изглед на таблицата *MyTable*, отразяващ промените в нейната дефиниция, е показан на фигура 3.

Column Name	Data Type	Allow Nulls
ID	int	<input type="checkbox"/>
FirstName	varchar(25)	<input type="checkbox"/>
MiddleInitial	char(1)	<input checked="" type="checkbox"/>
LastName	varchar(25)	<input type="checkbox"/>
EGN	char(10)	<input checked="" type="checkbox"/>

Фиг. 3 Промяна на дефиницията на таблицата *MyTable* в *Management Studio*

Изтриване на таблица

Една таблица може да бъде изтрита от базата от данни от нейния собственик или от системния администратор. Когато таблицата бъде изтрита, всички данни, индекси, тригери, ограничения, свързани с нея, също биват изтрети. Изтриване на таблица може да се извърши, като се избере командата *Delete* от контекстното меню на съответната таблица в *Management Studio* или като се използва *Transact-SQL*. Синтаксисът на SQL конструкцията за изтриване е:

```
DROP TABLE
[database_name.[schema_name]. | schema_name.] table_name
```

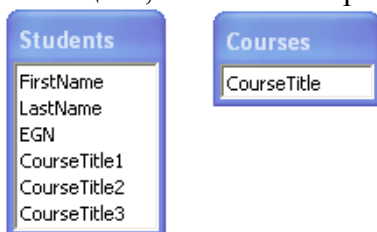
Пример 9

```
DROP TABLE MyDatabase..MyTable
```

Задачи

Задача 1. Да се създадат таблиците, включени в проектите на базите от данни, описани в задачите 1÷9 на тема „Проектиране на релационни бази от данни. Нормализация на данните”.

Задача 2. Разработва се проект на база от данни за система за курсово обучение. Две от таблиците, включени в проекта, са следните:

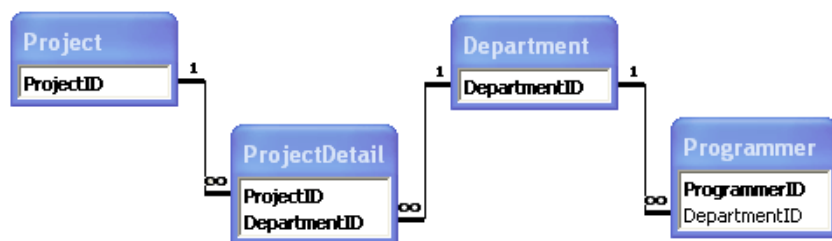


2.1. Да се напишат и изпълнят SQL конструкции за създаването на представените таблици.

2.2. Да се определи дали тези таблици са нормализирани. Ако не са, да се нормализират, след което да се напишат и изпълнят SQL конструкциите, отразяващи промените в проекта на базата от данни.

Задача 3. Проектира се база от данни, съхраняваща информация за проследяване на софтуерни проекти. Един проект може да бъде предприет от един или няколко отдела. Един проект може да бъде възложен на група от програмисти. Един програмист от даден отдел може да бъде включен в разработката на проект, който е поет от друг отдел, както и да изпълнява няколко задачи върху отделен проект.

3.1. Да се разгледа следния проект на базата от данни и да се установят основните му недостатъци:



Да се предложат промени в проекта, които преодоляват посочените недостатъци.

3.2. Да се напишат и изпълнят SQL конструкциите, които създават таблиците, включени в получения проект на базата от данни.

Ограничаване на стойностите на данните

SQL Server поддържа два различни начина за реализиране на целостността на данните – процедурен и декларативен. Процедурният се осъществява чрез изгледи, тригери, съхранени процедури, стойности по подразбиране (*defaults* като обекти), правила (*rules*) за осигуряване на допустима област от стойности. Този начин е по-гъвкав, но предизвиква много изпълнения и са възможни грешки. Декларативният начин за поддържане на цялостност на данните се реализира чрез ограниченията. Те осигуряват сбит и последователен начин да се управляват и четирите типа цялост на данните чрез разширен SQL синтаксис, използван при създаване и променяне на таблици. Ограниченията са по-малко податливи на грешки и предизвикват по-малко изпълнения в сравнение с процедурния метод, но са по-малко гъвкави.

Системните съхранени процедури `sp_help` и `sp_helpconstraint` дават информация за ограниченията в таблиците.

Имената на ограниченията трябва да са уникални в базата от данни, ако името на ограничението се пропусне, системата генерира име.

Използването на конструкциите `CREATE TABLE` и `ALTER TABLE` в Transact-SQL за добавяне и изтриване на ограничения дава повече възможности, но използването на графичния интерфейс на Management Studio за проектиране на таблици е по-лесен начин за дефиниране на ограничения.

Добавяне и изтриване на ограничения чрез Transact-SQL

Ограниченията се декларират на ниво колона или на ниво таблица. Ограниченията, които използват повече от една колона, се декларират на ниво таблица. Дефиниране на ограниченията при създаване на таблица:

```
CREATE TABLE
[database_name.[schema_name]. | schema_name.]table_name
(
    {col_name datatype [null_option] [Identity_option]
                                     [RowGUID_option] [col_constraint[,...]]
    | table_constraint
} [, ...]
)
[ON {filegroup_name | [DEFAULT]}]
[TEXTIMAGE_ON {filegroup_name | [DEFAULT]}]
```

Ограниченията на ниво колона са предпочитани, тъй като правят дефиницията по-лесна за четене. Ограниченията на ниво таблица се използват само когато е необходимо, включват се в края на списъка от колони и се именуват. Например:

```
CREATE TABLE Employees
( EmployeeID int NOT NULL IDENTITY PRIMARY KEY,
  FirstName varchar(25) NOT NULL,
  LastName varchar(25) NOT NULL,
  EGN char(10) NULL CONSTRAINT unique_EGN UNIQUE,
  Salary money NOT NULL CHECK (Salary > 0) DEFAULT 1000 )
```

Дефиниране на ограниченията при променяне на таблица:

```
ALTER TABLE
[database_name.[schema_name]. | schema_name.]table_name
[WITH CHECK | WITH NOCHECK]
{ ALTER COLUMN col_name new_datatype [null_option]
                                     [RowGUID_option]
```

```

|ADD {col_name datatype [null_option] [Identity_option]
      [RowGUID_option] [col_constraint[,...]]
      |table_constraint
      } [, ...]
|DROP {COLUMN col_name|[CONSTRAINT] constraint_name}[,...]
|{CHECK|NOCHECK} CONSTRAINT {ALL | constraint_name [,...]}
|{ENABLE | DISABLE} TRIGGER {ALL | trigger_name [,...]}
}

```

Когато се добавя нова колона с ограничение или се добавя ново ограничение на ниво таблица, синтаксисът, използван за описване на колоните и ограниченията, е точно както при конструкцията CREATE TABLE.

Изтриването на ограничение се осъществява по следния начин:

```

ALTER TABLE
[database_name.[schema_name]. | schema_name.]table_name
DROP [CONSTRAINT] constraint_name

```

Например:

```

ALTER TABLE Employees
DROP CONSTRAINT unique_EGN

```

Разрешаване и забраняване на ограничения

Когато се използва ALTER TABLE, таблиците, които се променят, вероятно вече имат данни в тях. Възможно е някои от тези данни да не удовлетворяват ограничението, което се добавя. Затова SQL Server автоматично проверява съществуващите данни за некоректни данни. Тази проверка може да отнеме време за изпълняване, така че SQL Server позволява да се определи опция, която да я предотврати. С опцията WITH NOCHECK изразът ALTER TABLE забранява проверката на ограничението за нов FOREIGN KEY и CHECK ограничение при тяхното добавяне. Това позволява ограниченията да бъдат добавяни, дори ако съществуват данни, които ги нарушават. По подразбиране е опцията WITH CHECK.

За забраняване на съществуващи ограничения се използва опцията NOCHECK CONSTRAINT в синтаксиса на ALTER TABLE. Всичко, което е необходимо, е името на ограничението, за да може да бъде забранено. Ограничението ще остане забранено, докато не бъде изпълнена командата ALTER TABLE с опцията CHECK CONSTRAINT. Когато забранено ограничение бъде разрешено отново, SQL Server не проверява отново съществуващите данни за нарушаване на ограничението.

Опцията NOCHECK няма ефект при ограниченията PRIMARY KEY, UNIQUE, DEFAULT.

Тригерите могат да бъдат разрешени или забранени, като се използват опциите ENABLE TRIGGER и DISABLE TRIGGER в изрази ALTER TABLE.

Ограниченията могат да се добавят, изтриват, разрешават и забраняват и като се използва прозореца *Design* на Management Studio. За целта се използват командите от меню *Table Designer* (фиг. 1) или бутоните от лентата с инструменти *Table Designer*



Стойностите *seed* и *increment* са опционални и определят съответно първата стойност, която ще се използва и стъпката на нарастване за следваща стойност в колоната. По подразбиране са единици.

Пример 1 Създаване на таблица с няколко колони, първата от които ID е IDENTITY колона с начална стойност и стъпка 1:

```
CREATE TABLE IdentityExample1
( ID int IDENTITY NOT NULL,
  FirstName varchar(25) NOT NULL,
  LastName varchar(25) NOT NULL )
```

Пример 2 Създаване на таблица с IDENTITY колона, на която изрично са дефинирани началната стойност и стъпката:

```
CREATE TABLE IdentityExample2
( ID int NOT NULL IDENTITY(100, 5),
  FirstName varchar(25) NOT NULL,
  LastName varchar(25) NOT NULL )
```

Пример 3 Създаване на таблица с IDENTITY колона със стъпка на нарастване отрицателно число:

```
CREATE TABLE IdentityExample3
( ID int NOT NULL IDENTITY(0, -1),
  FirstName varchar(25) NOT NULL,
  LastName varchar(25) NOT NULL )
```

Когато се вмъкват редове в таблица с колона IDENTITY чрез конструкцията INSERT, колоната IDENTITY трябва да бъде пропусната в списъка от колони. Например:

```
INSERT INTO IdentityExample1 (FirstName, LastName)
VALUES ('AAAAA', 'BBBBB')
```

Опцията IDENTITY_INSERT позволява изрично въвеждане на стойност в колона IDENTITY при използване на конструкцията INSERT. Има следния общ вид:

```
SET IDENTITY_INSERT
[database_name.[schema_name]. | schema_name.]table_name
{ON|OFF}
```

Пример 4

```
SET IDENTITY_INSERT IdentityExample1 ON
```

```
INSERT INTO IdentityExample1 (ID, FirstName, LastName)
VALUES (87, 'AAAAA', 'BBBBB')
```

Ако опцията IDENTITY_INSERT за таблицата IdentityExample1 не е включена, последната конструкция няма да може да се изпълни.

Може да се определи последната IDENTITY стойност, вмъкната в таблица за текущата конекция, чрез системната функция @@IDENTITY. Например:

```
SELECT @@IDENTITY
```

връща последната вмъкната стойност в колона със свойство IDENTITY за текущата конекция (от последната конструкция INSERT). Ако в таблицата, в която се добавя ред, няма колона със свойство IDENTITY, стойността на функцията е NULL.

Ограничение PRIMARY KEY

Ограничението PRIMARY KEY (ограничение първичен ключ) се използва за поддържане на цялостност на обект. Може да бъде дефинирано на ниво колона (за първичен ключ, състоящ се от една колона) или на ниво таблица (за първичен ключ, състоящ се от повече от една колона). При използване на ограничението PRIMARY KEY трябва да се има предвид следното:

- Колоните, съставлящи първичния ключ, трябва да не допускат стойности NULL.
- В една таблица може да има само един първичен ключ.
- В SQL Server върху колоните, съставлящи първичния ключ, се създава автоматично индекс, който по подразбиране е клъстериран. Този индекс не може да бъде изтрит, без да бъде изтрито ограничението първичен ключ, което поддържа.

Създаването на първичен ключ в конструкцията CREATE TABLE или ALTER TABLE има следния синтаксис:

```
[CONSTRAINT pk_name]
PRIMARY KEY [ CLUSTERED | NONCLUSTERED ]
[(col_name1 [, col_name2 [, ..., col_name16]])]
```

Пример 5 Създаване на таблица Employees с ограничение първичен ключ върху колоната EmployeeID:



```
CREATE TABLE Employees
( EmployeeID int          NOT NULL IDENTITY PRIMARY KEY,
  FirstName  varchar(30) NOT NULL,
  LastName   varchar(30) NOT NULL,
  HireDate   datetime     NULL )
```

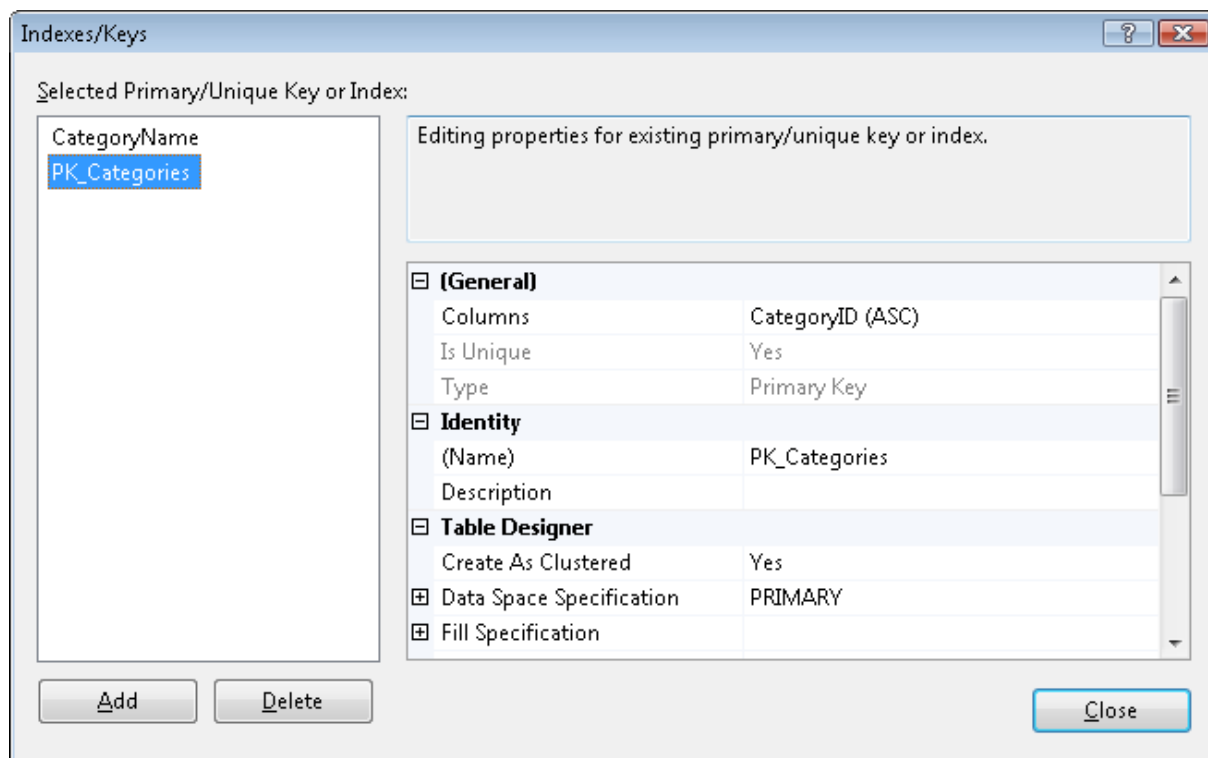
Пример 6 Създаване на таблица Employees и добавяне на ограничение първичен ключ върху колоната EmployeeID:

```
CREATE TABLE Employees
( EmployeeID int          NOT NULL IDENTITY,
  FirstName  varchar(30) NOT NULL,
  LastName   varchar(30) NOT NULL,
  HireDate   datetime     NULL )
ALTER TABLE Employees
ADD CONSTRAINT PK_Employees
PRIMARY KEY (EmployeeID)
```

Пример 7 Създаване на таблица EmployeeReview с първичен ключ PK_Employees, съставен от две колони EmployeeID и ReviewDate:

```
CREATE TABLE EmployeeReview
( EmployeeID int          NOT NULL,
  ReviewDate datetime NOT NULL,
  PerformanceRank tinyint NOT NULL,
  Comments text NULL,
  CONSTRAINT PK_Employees
PRIMARY KEY (EmployeeID, ReviewDate) )
```

Ограничението PRIMARY KEY може да се зададе, като се използва прозорецът *Design* на Management Studio чрез маркиране на колоните, съставлящи първичния ключ и избиране на командата от менюто *Table Designer | Set Primary Key* или бутона *Set Primary Key* . На фигура 2 е показана диалоговата рамка *Indexes/Keys*, която се отваря чрез натискане на бутона *Manage Indexes and Keys*  и позволява задаване на характеристиките на първичния ключ в таблицата *Categories*.



Фиг. 2 Задаване на ограничение първичен ключ

Ограничения UNIQUE

Ограничението UNIQUE (ограничение за уникалност) също се използва за поддържане на цялостност на обект. Ограничението за уникалност е подобно на ограничението първичен ключ, но трябва да се имат предвид следните различия:

- Колоните, съставлящи ограничение за уникалност, могат да допускат стойности NULL.
- В една таблица може да има нула, едно или повече ограничения за уникалност.
- В SQL Server върху колоните, върху които е дефинирано ограничение за уникалност, се създава автоматично уникален индекс, който по подразбиране е неклъстериран.

Създаването на ограничение за уникалност в конструкцията CREATE TABLE или ALTER TABLE има следния синтаксис:

```
[CONSTRAINT unique_name]
UNIQUE [CLUSTERED | NONCLUSTERED]
[(col_name1 [, col_name2 [, ..., col_name16]])]
```

Пример 8 Създаване на таблица Employees с колона EGN, върху която е наложено ограничение за уникалност:

```
CREATE TABLE Employees
( EmployeeID int NOT NULL IDENTITY PRIMARY KEY,
```

```

FirstName varchar(30) NOT NULL,
LastName varchar(30) NOT NULL,
HireDate datetime NULL,
EGN char(10) NULL UNIQUE )

```

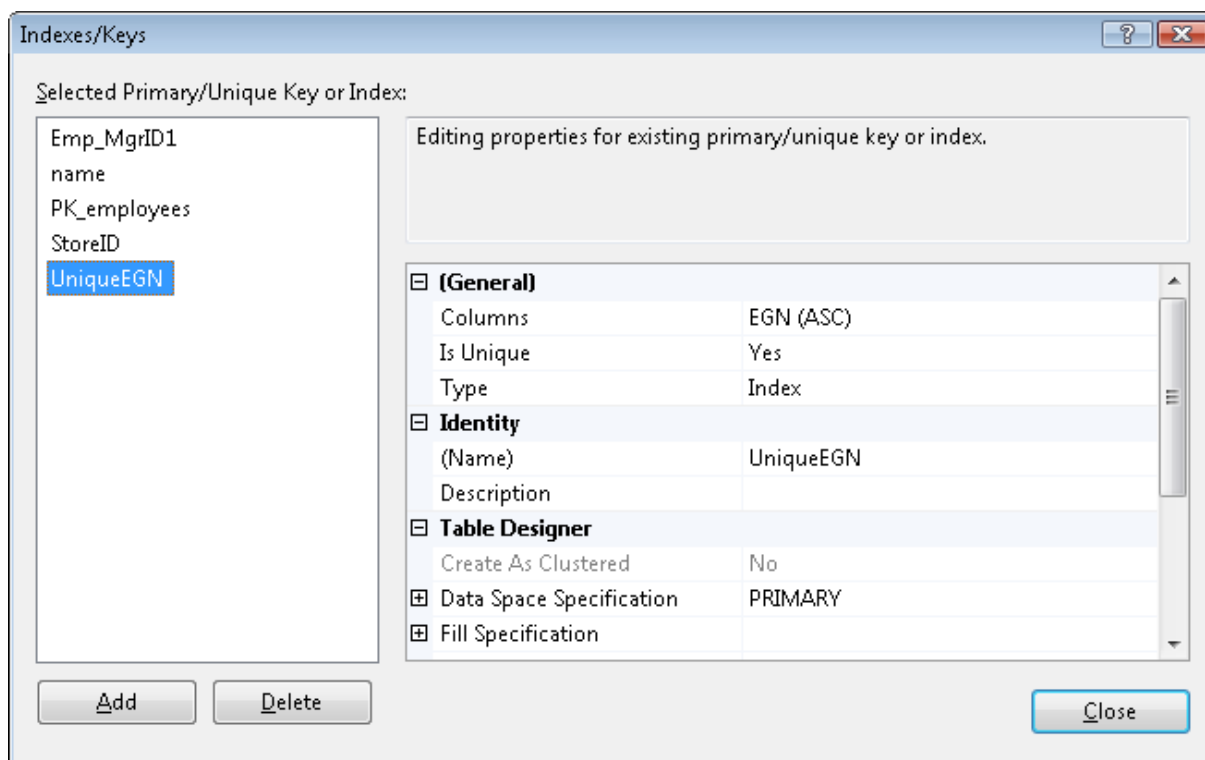
Пример 9 Добавяне на колона EGN, върху която е наложено ограничение за уникалност, във вече създадената таблица Employees:

```

ALTER TABLE Employees
ADD EGN char(10) NULL
CONSTRAINT unique_egn UNIQUE (EGN)

```

Ограничението UNIQUE може да се зададе, като се използва прозореца *Design* на Management Studio. Диалоговият прозорец *Indexes/Key* дава възможност за задаване на ограничение за уникалност на избраната таблица. На фигура 3 е показано задаване на ограничение за уникалност на колона за EGN.



Фиг. 3 Създаване на ограничение за уникалност

Ограничения FOREIGN KEY

Ограничението FOREIGN KEY (ограничение външен ключ) се използва за поддържане на цялостност на връзките между таблиците. Ограничението FOREIGN KEY може да бъде създадено на ниво колона (за външен ключ, състоящ се от една колона) или на ниво таблица (за външен ключ, състоящ се от повече от една колона). При използване на ограничението FOREIGN KEY трябва да се има предвид следното:

- Външният ключ трябва да се отнася към първичен ключ или ограничение за уникалност в реферираната таблица. Външният ключ може да реферира същата таблица (рекурсивна релационна връзка) или друга таблица в същата база от данни. За референция към таблица от друга база от данни се използва тригер.

- Броят на колоните във външния ключ трябва да съответства на броя на колоните в първичния ключ (или ограничението за уникалност) в реферираната таблица. Типовете на колоните също трябва да си съответстват.
- В SQL Server никакви индекси не се създават автоматично върху колоните, участващи във външния ключ. Обикновено трябва да се създаде индекс върху колоните от външния ключ, за да се подобри изпълнението на заявки, използващи външен ключ в съединение.

Предназначението на ограничението външен ключ е да контролира данните, съхранени в рефериращата и реферираната таблица. Родителските записи не могат да бъдат изтрети, ако съществуват съответни записи; родителският ключ не може да се променя, ако има съответни записи в рефериращата таблица; не може да се вмъква запис в рефериращата таблица, ако външният ключ не съответства на никоя първична стойност, т.е. подразбира се опцията NO ACTION. Може да се зададе каскадно изтриване и/или каскадно актуализиране чрез опцията CASCADE. Ако е избрана опцията SET NULL, всички колони на външния ключ се установяват на NULL, при условие че родителските записи са изтрети (при ON DELETE) или актуализирани за колоните, съставлящи родителския ключ (при ON UPDATE). Ако е избрана опцията SET DEFAULT, всички колони на външния ключ се установяват на стойностите си по подразбиране, при условие че родителските записи са изтрети (при ON DELETE) или актуализирани за колоните, съставлящи родителския ключ (при ON UPDATE).

Създаването на външен ключ в конструкцията CREATE TABLE или ALTER TABLE има следния синтаксис:

```
[CONSTRAINT fk_name]
[FOREIGN KEY (col_name1 [, col_name2 [, ..., col_name16]])]
REFERENCES referenced_table_name
(ref_col_name1 [, ref_col_name2 [, ..., ref_col_name16]])
[ON DELETE
    { NO ACTION | CASCADE | SET NULL | SET DEFAULT }]
[ON UPDATE
    { NO ACTION | CASCADE | SET NULL | SET DEFAULT }]
```

Пример 10 Създава се таблица EmployeeReview с външен ключ, рефериращ таблицата Employees. Ограничението FOREIGN KEY има генерирано от системата име.

```
CREATE TABLE EmployeeReview
( EmployeeID int NOT NULL
    REFERENCES Employees(EmployeeID),
  ReviewDate datetime NOT NULL,
  PerformanceRank tinyint NOT NULL,
  Comments text NULL,
  CONSTRAINT PK_EmployeeReview
  PRIMARY KEY (EmployeeID, ReviewDate) )
```

Пример 11 Този пример показва как да се създаде същия външен ключ, но на ниво таблица.

```
CREATE TABLE EmployeeReview
( EmployeeID int NOT NULL,
  ReviewDate datetime NOT NULL,
  PerformanceRank tinyint NOT NULL,
  Comments text NULL,
```

```

CONSTRAINT PK_EmployeeReview
    PRIMARY KEY (EmployeeID, ReviewDate),
FOREIGN KEY (EmployeeID)
    REFERENCES Employees (EmployeeID) )

```

Пример 12 Създава се таблица EmployeeCertification с външен ключ, рефериращ таблицата Employees. Ограничението FOREIGN KEY има генерирано от системата име.

```

CREATE TABLE EmployeeCertification
( EmployeeID int NOT NULL
    REFERENCES Employees (EmployeeID),
  CertificationTestNumber char(5) NOT NULL,
  DateTestPassed datetime NOT NULL,
  PRIMARY KEY (EmployeeID, CertificationTestNumber) )

```

Пример 13 Този пример показва как да се създаде рекурсивна релационна връзка. Таблицата Employees има референция към себе си. На ограничението е дадено име FK_ManagerID.

```

CREATE TABLE Employees
( EmployeeID int NOT NULL IDENTITY PRIMARY KEY,
  FirstName varchar(30) NOT NULL,
  LastName varchar(30) NOT NULL,
  HireDate datetime NULL,
  ManagerID int NULL,
  CONSTRAINT FK_ManagerID
    FOREIGN KEY (ManagerID)
    REFERENCES Employees (EmployeeID) )

```

Пример 14 Добавя се ограничение FOREIGN KEY към съществуваща таблица.

```


CREATE TABLE EmployeeReview
( EmployeeID int NOT NULL,
  ReviewDate datetime NOT NULL,
  PerformanceRank tinyint NOT NULL,
  Comments text NULL,
  CONSTRAINT PK_EmployeeReview
    PRIMARY KEY (EmployeeID, ReviewDate) )

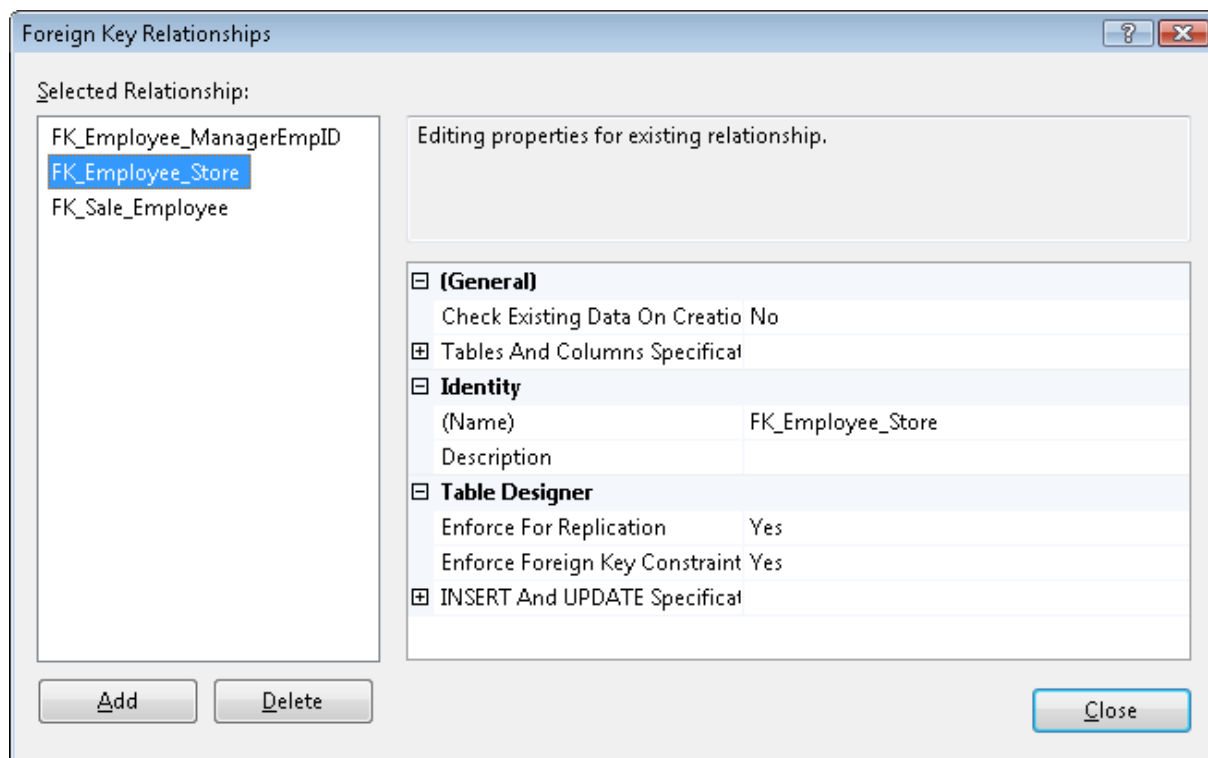
```

```

ALTER TABLE EmployeeReview
ADD CONSTRAINT FK_EmployeeReview
    FOREIGN KEY (EmployeeID)
    REFERENCES Employees (EmployeeID)

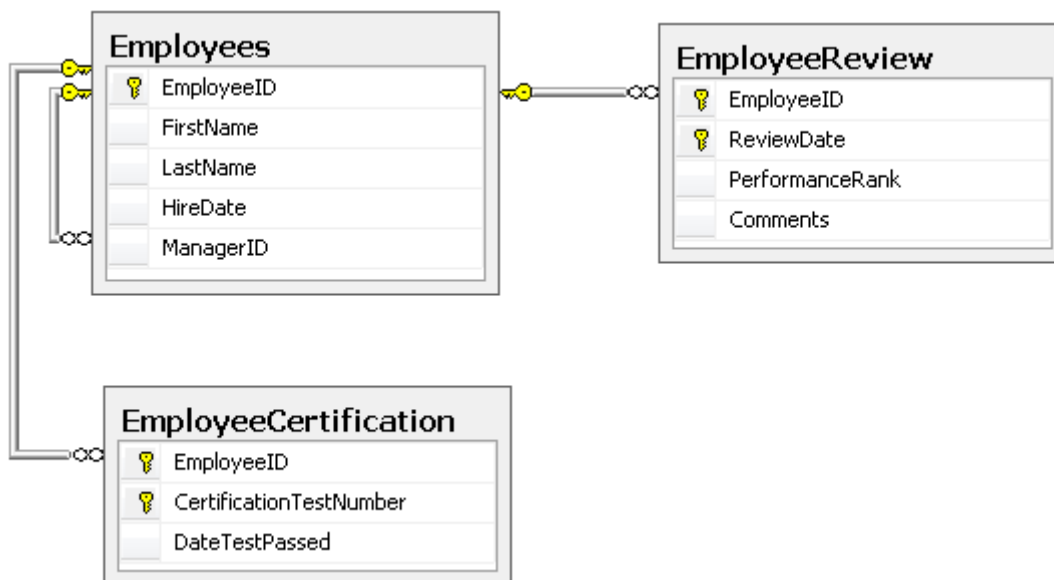
```

Ограниченията FOREIGN KEY могат да се преглеждат, добавят, изтриват, разрешават и забраняват и като се използва прозореца *Design* на Management Studio. Чрез натискане на бутона *Relationships*  се отваря диалоговия прозорец *Foreign Key Relationships*, който дава възможност за преглед на ограниченията външен ключ на избраната таблица, както и за дефиниране на нови (фиг. 4).



Фиг. 4 Задаване на ограничение външен ключ

Другата възможност за създаване на ограничения FOREIGN KEY е чрез *Database Diagrams* в SQL Server Management Studio. За представяне на връзките между разгледаните таблици се получава диаграмата, показана на фигура 5.



Фиг. 5 Задаване на ограничения външен ключ чрез диаграма

Ограничения за валидност на данните

Ограниченията за валидност на данните се използват за поддържане на цялостност на област в SQL Server чрез осигуряване на валидни стойности за колоните. Осъществява се по два начина: чрез проверка на стойностите на колоните според предварително дефинирани правила (при вмъкване и променяне) и чрез стойност по подразбиране за колона, ако стойност не е определена (при вмъкване).

Ограничения CHECK

Ограниченията CHECK осигуряват начин за SQL Server да потвърждава стойностите в колона (или колони), когато се вмъква или променя ред. При използване на ограничението CHECK трябва да се има предвид следното:

- Правилата за валидност на данните в ограничението CHECK трябва да са логически изрази.
- Ограниченията CHECK могат да се отнасят само до една таблица и не могат да използват подзаявки.
- За разлика от правилата (*rules*), ограниченията CHECK могат да се отнасят към други колони в таблицата.
- Ограниченията CHECK могат да се създават на ниво колона или на ниво таблица.

Създаването на ограничение CHECK в конструкцията CREATE TABLE или ALTER TABLE има следния синтаксис:

```
[CONSTRAINT check_constraint_name]  
CHECK (check_expression)
```

Пример 15 Създава се таблица с правило за валидност, което налага въвеждане на телефонен номер, форматиран по следния начин: (###) ###-####. Ограничението CHECK е зададено на ниво таблица и има име CK_ValidPhone.

```
CREATE TABLE Employees  
( EmployeeID int NOT NULL IDENTITY,  
  FirstName varchar(30) NOT NULL,  
  LastName varchar(30) NOT NULL,  
  HireDate datetime NULL,  
  HomePhone char(15) NULL,  
  CONSTRAINT CK_ValidPhone CHECK (HomePhone LIKE  
    '([0-9][0-9][0-9]) [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]'  
) )
```

Пример 16 Добавя се колона EGN с ограничение, осигуряващо въвеждане на точно 10 цифри.

```
ALTER TABLE Employees  
ADD EGN char(10) NULL CONSTRAINT Valid_EGN  
CHECK (EGN LIKE '[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]') )
```

Пример 17 Създава се таблица с ограничение CHECK, декларирано на ниво таблица с име valid_manager, според което никой служител не може да няма началник, освен служител с идентификатор 1, който се предполага, че е директор.

```
CREATE TABLE Employees  
( EmployeeID int NOT NULL IDENTITY PRIMARY KEY,  
  FirstName varchar(30) NOT NULL,  
  LastName varchar(30) NOT NULL,  
  HireDate datetime NULL,  
  ManagerID int NULL  
    CONSTRAINT FK_ManagerID  
      FOREIGN KEY  
        REFERENCES Employees (EmployeeID),  
  CONSTRAINT valid_manager
```

```
CHECK (ManagerID IS NOT NULL OR EmployeeID = 1) )
```

Пример 18 Добавя се ограничението `no_nums`, което гарантира, че във фамилията на служител не могат да бъдат въведени цифри. Не се проверяват съществуващите данни дали удовлетворяват условието на ограничението.

```
ALTER TABLE Employees
WITH NOCHECK
ADD CONSTRAINT no_nums
CHECK (LastName NOT LIKE '%[0-9]%')
```

Пример 19 Добавя се ограничението `end_of_month`, което гарантира, че не могат да се правят промени в таблицата след 28^{мо} число на месеца. Това ограничение никога не се обръща към колона от таблицата.

```
ALTER TABLE Employees
ADD CONSTRAINT end_of_month
CHECK (DATEPART(day, GETDATE()) < 28)
```

Пример 20 Забранява се ограничението `no_nums`.


```
ALTER TABLE Employees
NOCHECK CONSTRAINT no_nums
```

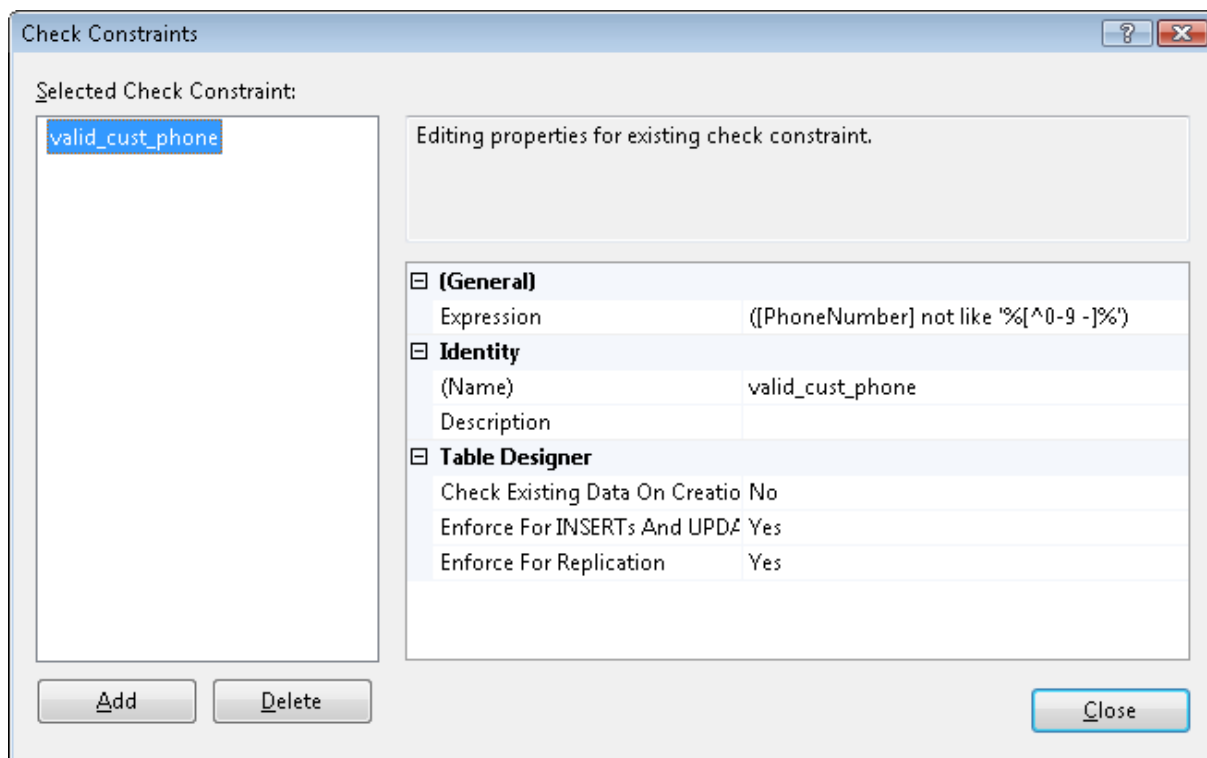
Пример 21 Изтрива се ограничението `no_nums`.

```
ALTER TABLE Employees
DROP CONSTRAINT no_nums
```

Пример 22 Добавя се ограничението `valid_cust_phone`, което гарантира, че в телефонния номер на клиент не могат да бъдат въведени символи, различни от цифри, интервали и тирета. Не се проверяват съществуващите данни дали удовлетворяват условието на ограничението.

```
ALTER TABLE Customers
WITH NOCHECK
ADD CONSTRAINT valid_cust_phone
CHECK (PhoneNumber NOT LIKE '%[^0-9 -]%')
```

Ограничението CHECK може да се добави и като се използва прозореца *Design* в SQL Server Management Studio. Чрез натискане на бутона *Manage Check Constraints*  се отваря диалоговия прозорец *Check Constraints*, който дава възможност за дефиниране на ограничение CHECK на избраната таблица. На фигура 6 е показано как може да се зададе ограничението от последния пример в Management Studio.



Фиг. 6 Ограничение за осигуряване на валидност на стойностите в колоната *PhoneNumber*

Ограничения DEFAULT

Ограниченията DEFAULT (ограничения по подразбиране) се използват, когато се вмъква нов ред в таблица. Ако колоната има ограничение DEFAULT и потребителят не задава изрично стойност за колоната, тогава се въвежда стойността от ограничението. При използване на ограничението DEFAULT трябва да се има предвид следното:

- Само едно ограничение DEFAULT може да се зададе за дадена колона.
- Това ограничение не може да се зададе за колона със свойство IDENTITY или за колона от тип *timestamp*.
- Ограниченията DEFAULT могат да се зададат на ниво колона или на ниво таблица.

Създаването на ограничение DEFAULT в конструкция CREATE TABLE или ALTER TABLE има следния синтаксис:

```
[CONSTRAINT default_constraint_name]
DEFAULT {constant_value | nulladic_function | NULL}
[FOR column_name]
```

Ключовата дума FOR е необходима само, когато ограничението се декларира на ниво таблица. Стойността по подразбиране може да се състои от константа (като число 87 или символен низ 'абв'), NULL или резултат от функция. Използваната функция трябва да връща резултат без да приема параметри, например USER_NAME() и GETDATE(), които връщат съответно името на текущия потребител на базата от данни и текущата дата.

Ограничението DEFAULT е особено важно за колони от тип *UniqueIdentifier*. Може да се въведе стойност, предварително генерирана от друг източник в поле от тип *UniqueIdentifier*, но ако това поле ще се използва за първичен ключ, трябва да се приложи функцията NewID(), която генерира стойност с помощта на уникален

алгоритъм. Ако е указана функцията NewID() в ограничение DEFAULT за такава колона, автоматично се задава глобално уникална стойност за колоната.

При добавяне на колона с ограничение DEFAULT чрез конструкцията ALTER TABLE може да се използват ключовите думи WITH VALUES за попълване на съществуващите редове със съответната стойност:

```
ALTER TABLE table_name
ADD column_name datatype [null_option]
[ [CONSTRAINT default_constraint_name]
  DEFAULT {constant_value | nulladic_function | NULL}
  [WITH VALUES] ]
```

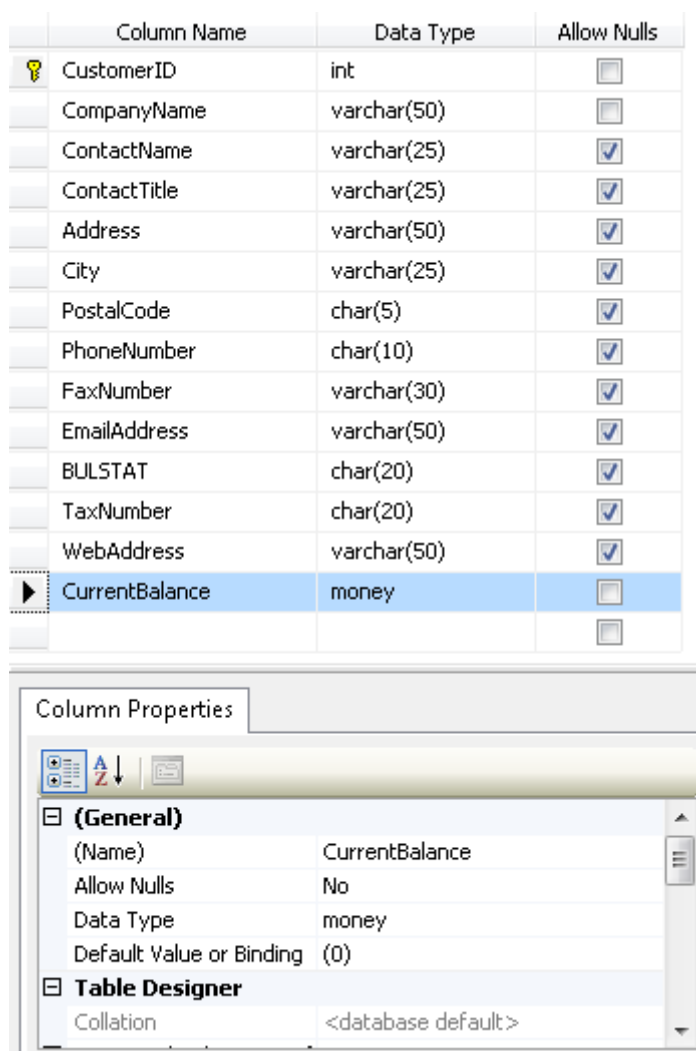
Пример 23 Нека е създадена по следния начин таблицата:

```
CREATE TABLE Customers
( CustomerID int NOT NULL IDENTITY
  CONSTRAINT PK_customers PRIMARY KEY,
  CustomerName varchar(50) NOT NULL,
  Address varchar(50) NULL,
  City varchar(25) NULL,
  PhoneNumber char(15) NULL )
```

В нея са въведени редове, след което е необходимо да се добави колона CurrentBalance и да се попълни със стойност 0:

```
ALTER TABLE Customers
ADD CurrentBalance money NOT NULL
  CONSTRAINT DF_CurrentBalance
  DEFAULT 0 WITH VALUES
```

Ограничението DEFAULT може да се добави и като се използва прозореца *Design* в SQL Server Management Studio. Мрежата, съдържаща колоните на таблицата и техните свойства, има графа *Default Value or Binding* за въвеждане на стойност по подразбиране за всяка колона, за която е възможно (фиг. 7).



Фиг. 7 Задаване на стойност по подразбиране

Глобално уникални идентификатори

Ако едно приложение трябва да генерира идентификаторна колона, която е уникална в рамките на цялата база от данни (или в рамките на всички бази от данни на всички мрежови компютри по света), трябва да се приложат свойството ROWGUIDCOL, тип данни *uniqueidentifier* и функция `NewID()`. Свойството ROWGUIDCOL на колона от тип *uniqueidentifier* се задава, ако тя ще се използва за идентификатор на ред.

Пример 24

```
CREATE TABLE Employees
( EmployeeID uniqueidentifier NOT NULL ROWGUIDCOL
    DEFAULT NewID(),
  FirstName varchar(25) NOT NULL,
  LastName varchar(25) NOT NULL )
```

За да се добави или изтрие свойството ROWGUIDCOL за съществуваща колона от тип *uniqueidentifier*, се използва конструкцията ALTER TABLE по следния начин:

```
ALTER TABLE table_name
ALTER COLUMN column_name {ADD | DROP} ROWGUIDCOL
```

Пример 25

```
CREATE TABLE Employees
```

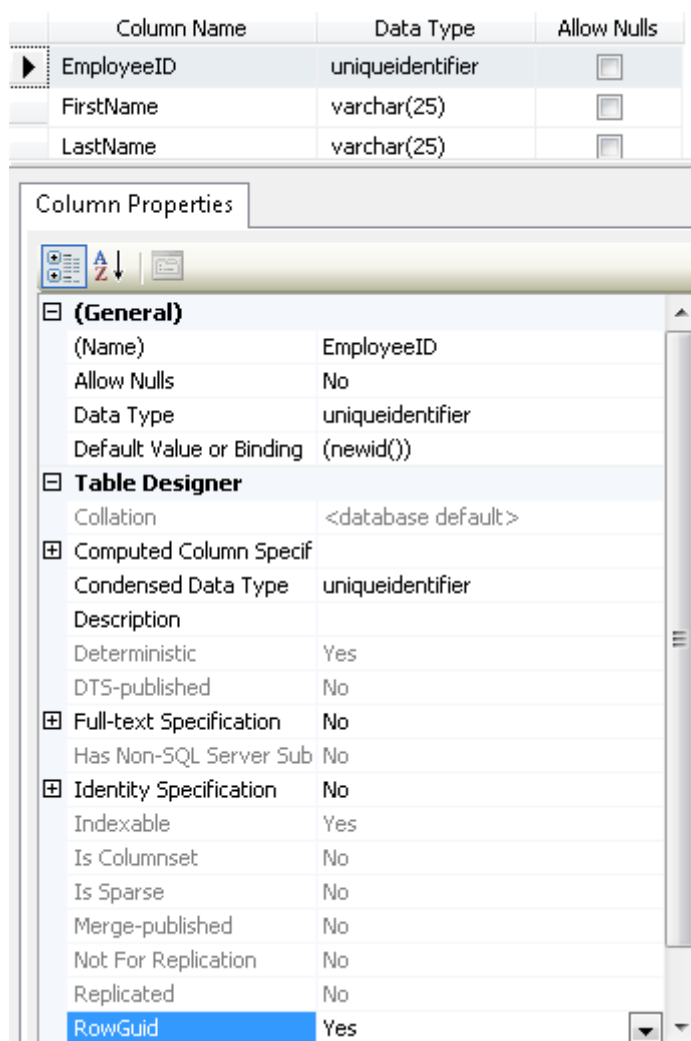


```
( EmployeeID uniqueidentifier NOT NULL DEFAULT NewID(),
  FirstName varchar(25) NOT NULL,
  LastName varchar(25) NOT NULL )
```

```
ALTER TABLE Employees
```

```
ALTER COLUMN EmployeeID ADD ROWGUIDCOL
```

За да се зададе свойството ROWGUIDCOL на колона в прозореца *Design* на Management Studio, се установява стойност Yes на *Is RowGuid* за съответната колона, в резултат на което в *Default Value* се появява (newid()) , както е показано на фигура 8.



Фиг. 8 Задаване на свойство ROWGUIDCOL

Задачи

Задача 1. Да се зададат подходящите ограничения на данните в таблиците, създаването на които е описано в задачите 1÷3 на тема „Използване на SQL за дефиниране на данни”.

Задача 2. Проектира се база от данни, която да съхранява информацията, необходима на една фирма за продажба на определени продукти, за да може по-лесно и по-ефикасно да се управляват наличностите и да се следят продажбите. За всеки *продукт* се съхранява информация за наименованието му; вид (*категория*); доставчик; доставна цена; налично количество; дали продажбата на този продукт е преустановена

или не; количество, което се счита за критично за съответния продукт (т.е. показва необходимост от нова доставка). За *магазините* на фирмата се съхранява информация за име; адрес; град; телефонен номер; факс; e-mail; Web адрес. За *служителите* се поддържа информация за имена; длъжност; ЕГН; магазина, в който работи; дата на постъпване; дата на напускане; телефонен номер; адрес; град; e-mail. За *доставчиците* на стоки се съхранява информация за името на фирмата-доставчик; име за контакт; длъжност на служителя, чието име е съхранено; адрес; телефонен номер; факс; e-mail; Web адрес. За *клиентите* се поддържа информация за име на фирмата-клиент; име за контакт; длъжност; адрес; телефонен номер; факс; e-mail; Web адрес; Bulstat; данъчен номер (за да е възможно издаване на фактури). За *продажбите* се поддържа информация за клиент; служител; дата на продажба; обща сума на продажба; продукти, закупени с дадена продажба; продадено количество от съответния продукт; продажна цена; отстъпка като процент от продажната цена за покупка на голямо количество от даден продукт; отстъпка като процент от общата сума за покупка на стойност над дадена сума или за клиенти с годишна сума на покупките над дадена сума.

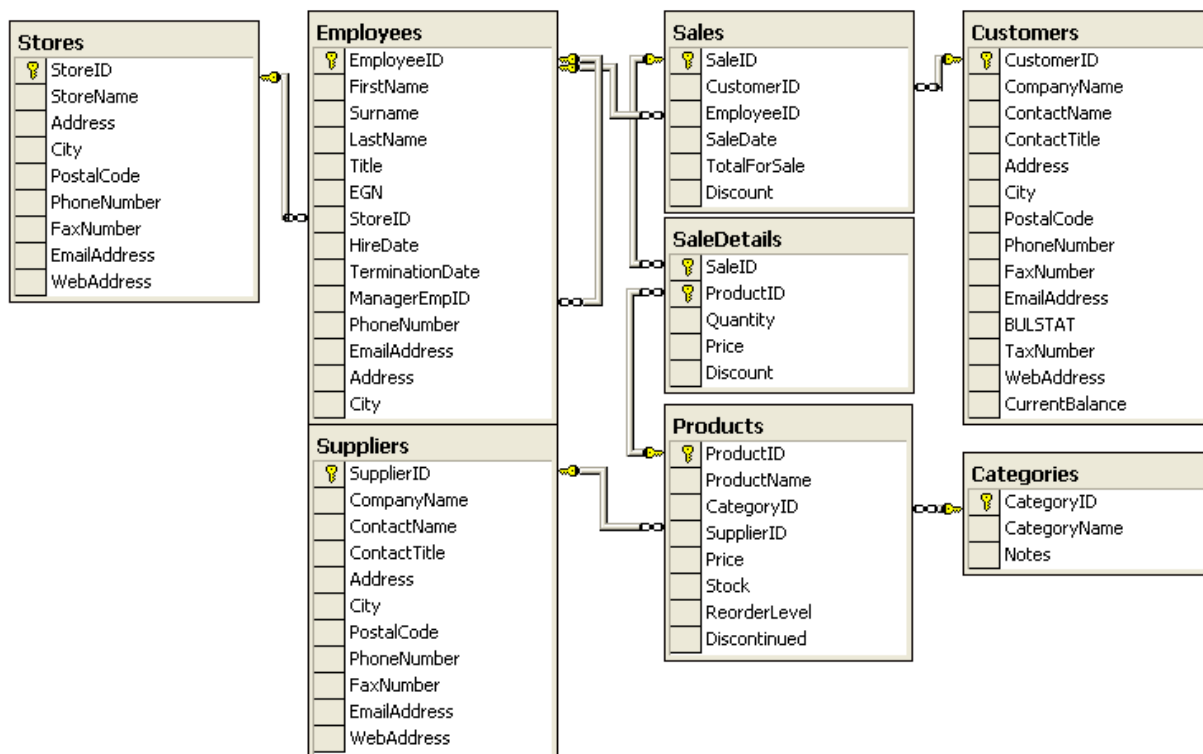
Нека таблиците, които са необходими, са Products, Categories, Suppliers, Employees, Customers, Sales, SaleDetails. Колоните, от които се състоят, са представени в таблица 1. Колоните, съставлящи първичния ключ в съответната таблица на базата от данни, са подчертани.

Таблица	Колони
Products	<u>ProductID</u> , ProductName, CategoryID, SupplierID, Price, Stock, ReorderLevel, Discontinued
Categories	CategoryID, CategoryName, Notes
Suppliers	<u>SupplierID</u> , CompanyName, ContactName, ContactTitle, Address, City, PostalCode, PhoneNumber, FaxNumber, EmailAddress, WebAddress
Employees	<u>EmployeeID</u> , FirstName, Surname, LastName, Title, EGN, StoreID, HireDate, TerminationDate, ManagerEmpID, PhoneNumber, Address, City, EmailAddress
Stores	<u>StoreID</u> , StoreName, Address, City, PostalCode, PhoneNumber, FaxNumber, EmailAddress, WebAddress
Customers	<u>CustomerID</u> , CompanyName, ContactName, ContactTitle, Address, City, PostalCode, PhoneNumber, FaxNumber, EmailAddress, WebAddress, Bulstat, TaxNumber, CurrentBalance
Sales	<u>SaleID</u> , CustomerID, EmployeeID, SaleDate, TotalForSale, Discount
SaleDetails	<u>SaleID</u> , <u>ProductID</u> , Quantity, Price, Discount

Таблица 1 Колоните, съдържащи се в съответните таблици

Забележка Приемаме, че един продукт се доставя от точно един доставчик, т.е. релационната връзка между таблиците Suppliers и Products е “едно към много”. Освен това в базата от данни не е осигурена възможност за подробно проследяване на отделните доставки на различни продукти. Таблиците в разглежданата база от данни ще се използват във всички останали примери, като целта е в тях да се наблегне на основния смисъл, което изисква да се избегне излишното усложняване на модела на базата от данни.

Релационните връзки между таблиците в базата от данни са показани на фигура 9.



Фиг. 9 Диаграма, представяща връзките между таблиците

Задачата е да се реализира проектираната база от данни, като се създадат описаните таблици и се зададат подходящите ограничения на данните.

Глава II Програмиране на бази от данни

Използване на SQL за извличане на данни от таблици. Конструкцията **SELECT**

SQL (*Structured Query Language* – език за структурирани заявки) е стандартния език, използван за дефиниране, манипулиране и извличане на данни от релационните бази от данни. Наименованието на езика произхожда от факта, че заявките са най-често използваната част от SQL. Заявката е конструкция, зададена към системата за управление на бази от данни, която изисква получаването на определена информация. Всички заявки в SQL се състоят от една конструкция и тя е SELECT, т.е. конструкцията SELECT се използва за извличане на данни от една или повече таблици. Резултатът е набор от данни (в табличен вид), състоящ се от колони и редове. Най-простият вид на командата SELECT е:

```
SELECT column_list FROM table_name
```

където *column_list* е списък от имена на колони от таблицата *table_name*, разделени със запетая.

Пример 1

```
SELECT FirstName, Lastname, PhoneNumber
FROM Employees
```

Резултатът от тази заявка (фиг. 1) е таблица от три колони – *FirstName*, *Lastname*, *PhoneNumber* и всички редове в таблицата за служителите *Employees*.

FirstName	Lastname	PhoneNumber
Стоян	Георгиев	778678687
Георги	Христов	78587678687
Ваня	Христова	56346565
Стеля	Миланова	232432545
Атанас	Лазаров	78587678687
Катя	Цветанова	9786445456

Фиг. 1 Примерен резултат от заявката в пример 1

Със символа * се указват всички полета (т.е. колони) на таблицата.

Пример 2

```
SELECT * FROM Customers
```

С опционалната ключова дума AS се задава ново име на предхождания израз.

Пример 3

```
SELECT CompanyName AS [Name of Supplier], Address
FROM Suppliers
```

На фигура 2 е показан примерен резултат от тази заявка.

Name of Supplier	Address
БЕТИ - 2002	бул. България, 34
АЛЕМИТ ООД	бул. Хр. Ботев, 5
АЛПИ ООД	бул. България, 2

Фиг. 2 Примерен резултат от заявката в пример 2

Добавянето на WHERE позволява да се зададе условие, чрез което да се получат не всички редове, а само тези, които удовлетворяват дадено условие. Общият вид на командата в този случай е:

```
SELECT column_list FROM table_name
WHERE search_condition
```

Пример 4 Заявка, която извлича имената и фамилиите на служителите, назначени на длъжност „управител”.

```
SELECT FirstName, LastName
FROM Employees
WHERE Title = 'управител'
```

Примерен резултат от тази заявка е показан на фигура 3.

FirstName	LastName
Георги	Христов
Атанас	Лазаров

Фиг. 3 Примерен резултат от заявката в пример 3

Критериите за избор могат да включват стандартни оператори +, -, *, / , % (остатък при деление), >, <, >=, <=, =, <>, както и някои специални SQL оператори:

- IS NULL проверява дали на полето е зададена стойност. За проверка дали стойността не е NULL се използва `column_name IS NOT NULL`. Когато някоя стойност в колоната е NULL, това означава, че програмата за управление на базата от данни специално е маркирала колоната като не притежаваща никаква стойност за съответния ред. Поради използването на стойност NULL, булевите изрази в SQL освен TRUE или FALSE, могат да бъдат и UNKNOWN. Следователно SQL използва тризначната логика (three-valued logic – 3VL) вместо традиционната двузначна логика. Най-лесният начин за запомняне на правилата за прилагане на операторите AND, OR и NOT при 3VL е, ако TRUE се разглежда като стойност 1, FALSE – като 0 и UNKNOWN – като ½. Тогава резултатът от операцията $x \text{ AND } y$ е $\min(x, y)$; резултатът от $x \text{ OR } y$ е $\max(x, y)$; резултатът от NOT x е $1-x$.

Таблица на истинност за оператора NOT в тризначната логика:

Израз	Логическа стойност на израза
NOT TRUE	FALSE
NOT FALSE	TRUE
NOT UNKNOWN	UNKNOWN

Таблица на истинност за оператора OR в тризначната логика:

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Таблица на истинност за оператора AND в тризначната логика:

AND	TRUE	FALSE	UNKNOWN
-----	------	-------	---------

TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

Пример 5 В резултат на изпълнението на заявката

```
SELECT * FROM Customers
```

```
WHERE City <> 'Велико Търново'
```

редовете със стойност NULL в колоната City няма да се изведат, тъй като WHERE връща само редовете, за които условието има стойност TRUE, а сравнението NULL <> 'Veliko Tarnovo' връща стойността UNKNOWN. За да се върнат редовете със стойност NULL в колоната City, трябва да се използва следната заявка:

```
SELECT * FROM Customers
```

```
WHERE City <> 'Велико Търново' OR City IS NULL
```

- BETWEEN *min_value* AND *max_value* проверява дали стойността е в зададения интервал.

Пример 6 Следната заявка избира всички клиенти, чиито имена са в азбучния обхват А÷Д:

```
SELECT * FROM Customers
```

```
WHERE CompanyName BETWEEN 'A' AND 'Д'
```

- IN (*value1*, *value2*, ..., *valueN*) връща TRUE, ако стойността е в зададения списък от стойности.

Пример 7 Следната заявка избира всички клиенти, чиито идентификатори са 1 или 10, или 87:

```
SELECT * FROM Customers
```

```
WHERE CustomerID IN (1, 10, 87)
```

- LIKE '*шаблон на символен низ*' сравнява символните низове със зададения шаблон и връща TRUE, ако низът отговаря на шаблона. За съставяне на шаблон се използват следните обобщаващи символи:
 - символът долна черта (_) означава един произволен символ;
 - символът процент (%) означава последователност от произволни символи (включително 0 на брой);
 - [] означава един знак, намиращ се в указаната област (например [а-д]) или указаното множество (например [аверХЗ] или [87м-пу-]) означава измежду символите 8, 7, м÷п, у, -);
 - [^] означава един знак, намиращ се извън указаната област (например [^а-д]) или извън указаното множество ([^аверХЗ]);
 - за търсене на низ, в който има % или _, трябва да се дефинира *управляващ символ* (*escape character*). Управляващият символ се използва непосредствено преди знака % или _ и означава, че знакът % или _ трябва да се интерпретира буквално – като обикновен символ, а не като обобщаващ символ.

Пример 8 Извеждане на всички клиенти, съдържащи в email адреса си знака долна черта (_).

```
SELECT * FROM Customers
WHERE EmailAddress LIKE '%/_%' ESCAPE '/'
```

Пример 9 Извеждане на редовете от Categories, съдържащи в колоната Notes низа '87%'.

```
SELECT * FROM Categories
WHERE Notes LIKE '%87!%%' ESCAPE '!'
```

- възможно е да се огради специалния символ в [], за да се използва като обикновен символ.

Пример 10 Извличане на данните за служителите, чиито email адреси започват с 'ivan_':

```
SELECT * FROM Employees
WHERE EmailAddress LIKE 'ivan!_%' ESCAPE '!'
```

или

```
SELECT * FROM Employees
WHERE EmailAddress LIKE 'ivan[_]%'
```

Ако символен низ съдържа символа апостроф и трябва да бъде заграден с апострофи, представянето на апострофа в низа се осъществява чрез два апострофа.

Пример 11 За извеждане на имената и адресите на тези клиенти, за които колоната Address съдържа апостроф:

```
SELECT CompanyName, Address
FROM Customers
WHERE Address LIKE '%''%'
```

Ключовата дума DISTINCT предоставя начин за ограничаване на изходните данни само до уникалните стойности. DISTINCT се поставя в конструкцията SELECT.

Пример 12

```
SELECT DISTINCT City FROM Customers
```

Премахват се повтарящите се редове от крайния резултат и се връщат само различните градове, в които има клиенти.

Друга възможност е сортирането на информацията, получена от SELECT заявката, като се използва изразът ORDER BY, указвайки една или повече колони. Синтаксисът на изразът е:

```
ORDER BY { column_name [sort_order] } [,...]
```

Указването на ред за сортиране не е задължително и се осъществява чрез ключовите думи ASC или DESC за сортиране в нарастващ или намаляващ ред. По подразбиране редовете се сортират в нарастващ ред и ASC може да се пропусне.

Пример 13 Заявка, която избира колони от таблицата за служителите и извежда резултатния набор от редове, сортирани в азбучен ред.

```
SELECT FirstName, LastName, PhoneNumber,
       HireDate, TerminationDate
FROM Employees
ORDER BY FirstName ASC, LastName ASC
```

Пример 14 За разлика от предишния пример тази заявка избира редовете от таблицата за служителите, които не са напуснали (нямат въведена дата на напускане

TerminationDate) и сортира резултатния набор от редове по дата на постъпване в низходящ ред; редовете с една и съща дата на постъпване HireDate се подреждат по азбучен ред.

```
SELECT FirstName, LastName, PhoneNumber,
       HireDate, TerminationDate
FROM Employees
WHERE TerminationDate IS NULL
ORDER BY HireDate DESC, FirstName ASC, LastName ASC
```

Може да се посочат номерата в последователността на избраните колони в списъка с имената на колоните на SELECT вместо техните имена, за да се определят колоните, които да се използват за подреждане на изходните данни.

Пример 15 Заявка, която избира колони от таблицата за клиенти от даден град и извежда резултатния набор от редове, сортирани в азбучен ред.

```
SELECT CompanyName, Address, PhoneNumber
FROM Customers
WHERE City = 'Велико Търново'
ORDER BY 1
```

Задачи

Задача 1. Да се напише заявка, която извежда всички колони от таблицата за продажбите, направени от служителите с идентификатори 2, 5, 12 или 90 в периода между 01/10/2003 и 31/12/2003.

Задача 2. Да се напише заявка, която извежда данните за продуктите, които имат критично налично количество и продажбата им не е преустановена.

Задача 3. Да се напише заявка, която извежда данните за служителите, родени през 1970 година.

Задача 4. Да се напише заявка, която извежда данните за служителите, чиито фамилии започват с някоя от буквите А, У, Д÷И или не започват с буквите Б, Л, М, Р÷Я.

Задача 5. Да се напише заявка, която извежда данните за клиентите, чиито имена (CompanyName) не съдържат цифри.

Задача 6. Да се напише заявка, която извежда данните за продуктите, които имат имена, състоящи се само от буквите а, б, в, д, и, о, п÷я или интервал (т.е. имената на продуктите да са съставени от всички или част от изброените символи).

Задача 7. Да се зададе ограничение за валидност:

7.1. на колоната EGN в таблицата Employees, с което да се гарантира въвеждането на точно 10 цифри;

7.2. на колоната PhoneNumber (в таблиците Employees, Stores, Suppliers, Customers), с което да се гарантира въвеждане на произволна комбинация от цифри, тирета и интервали.

Задача 8. Какви ще бъдат изходните данни от следните заявки:

8.1.

```
SELECT * FROM Employees
WHERE TerminationDate IS NOT NULL
      AND LastName LIKE 'A%'
```

8.2.

```
SELECT * FROM Sales
WHERE (TotalForSale >= 100 OR TotalForsale <=10)
      AND SaleDate BETWEEN '01/01/2003' AND '02/01/2003'
```

8.3.

```
SELECT DISTINCT EmployeeID
FROM Sales
WHERE CustomerID IN (5, 10, 18, 87)
```

8.4.

```
SELECT * FROM Stores
WHERE City IN ('Велико Търново', 'София', 'Варна')
      OR City LIKE 'G%a'
ORDER BY StoreName
```

Използване на SQL за извличане на данни от таблици. Конструкцията **SELECT**

SQL (*Structured Query Language* – език за структурирани заявки) е стандартния език, използван за дефиниране, манипулиране и извличане на данни от релационните бази от данни. Наименованието на езика произхожда от факта, че заявките са най-често използваната част от SQL. Заявката е конструкция, зададена към системата за управление на бази от данни, която изисква получаването на определена информация. Всички заявки в SQL се състоят от една конструкция и тя е SELECT, т.е. конструкцията SELECT се използва за извличане на данни от една или повече таблици. Резултатът е набор от данни (в табличен вид), състоящ се от колони и редове. Най-простият вид на командата SELECT е:

```
SELECT column_list FROM table_name
```

където *column_list* е списък от имена на колони от таблицата *table_name*, разделени със запетая.

Пример 1

```
SELECT FirstName, Lastname, PhoneNumber
FROM Employees
```

Резултатът от тази заявка (фиг. 1) е таблица от три колони – *FirstName*, *Lastname*, *PhoneNumber* и всички редове в таблицата за служителите *Employees*.

FirstName	Lastname	PhoneNumber
Стоян	Георгиев	778678687
Георги	Христов	78587678687
Ваня	Христова	56346565
Стефа	Миланова	232432545
Атанас	Лазаров	78587678687
Катя	Цветанова	9786445456

Фиг. 1 Примерен резултат от заявката в пример 1

Със символа * се указват всички полета (т.е. колони) на таблицата.

Пример 2

```
SELECT * FROM Customers
```

С опционалната ключова дума AS се задава ново име на предхождащия израз.

Пример 3

```
SELECT CompanyName AS [Name of Supplier], Address
FROM Suppliers
```

На фигура 2 е показан примерен резултат от тази заявка.

Name of Supplier	Address
БЕТИ - 2002	бул. България, 34
АЛЕМИТ ООД	бул. Хр. Ботев, 5
АЛПИ ООД	бул. България, 2

Фиг. 2 Примерен резултат от заявката в пример 2

Добавянето на WHERE позволява да се зададе условие, чрез което да се получат не всички редове, а само тези, които удовлетворяват дадено условие. Общият вид на командата в този случай е:

```
SELECT column_list FROM table_name
WHERE search_condition
```

Пример 4 Заявка, която извлича имената и фамилиите на служителите, назначени на длъжност „управител”.

```
SELECT FirstName, LastName
FROM Employees
WHERE Title = 'управител'
```

Примерен резултат от тази заявка е показан на фигура 3.

FirstName	LastName
Георги	Христов
Атанас	Лазаров

Фиг. 3 Примерен резултат от заявката в пример 3

Критериите за избор могат да включват стандартни оператори +, -, *, / , % (остатък при деление), >, <, >=, <=, =, <>, както и някои специални SQL оператори:

- IS NULL проверява дали на полето е зададена стойност. За проверка дали стойността не е NULL се използва *column_name* IS NOT NULL. Когато някоя стойност в колоната е NULL, това означава, че програмата за управление на базата от данни специално е маркирала колоната като не притежаваща никаква стойност за съответния ред. Поради използването на стойност NULL, булевите изрази в SQL освен TRUE или FALSE, могат да бъдат и UNKNOWN. Следователно SQL използва тризначната логика (three-valued logic – 3VL) вместо традиционната двузначна логика. Най-лесният начин за запомняне на правилата за прилагане на операторите AND, OR и NOT при 3VL е, ако TRUE се разглежда като стойност 1, FALSE – като 0 и UNKNOWN – като 1/2. Тогава резултатът от операцията *x* AND *y* е *min(x, y)*; резултатът от *x* OR *y* е *max(x, y)*; резултатът от NOT *x* е 1-*x*.

Таблица на истинност за оператора NOT в тризначната логика:

Израз	Логическа стойност на израза
NOT TRUE	FALSE
NOT FALSE	TRUE
NOT UNKNOWN	UNKNOWN

Таблица на истинност за оператора OR в тризначната логика:

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Таблица на истинност за оператора AND в тризначната логика:

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN

FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

Пример 5 В резултат на изпълнението на заявката

```
SELECT * FROM Customers
```

```
WHERE City <> 'Велико Търново'
```

редовете със стойност NULL в колоната City няма да се изведат, тъй като WHERE връща само редовете, за които условието има стойност TRUE, а сравнението NULL <> 'Veliko Tarnovo' връща стойността UNKNOWN. За да се върнат редовете със стойност NULL в колоната City, трябва да се използва следната заявка:

```
SELECT * FROM Customers
```

```
WHERE City <> 'Велико Търново' OR City IS NULL
```

- BETWEEN *min_value* AND *max_value* проверява дали стойността е в зададения интервал.

Пример 6 Следната заявка избира всички клиенти, чиито имена са в азбучния обхват А÷Д:

```
SELECT * FROM Customers
```

```
WHERE CompanyName BETWEEN 'А' AND 'Д'
```

- IN (*value1*, *value2*, ..., *valueN*) връща TRUE, ако стойността е в зададения списък от стойности.

Пример 7 Следната заявка избира всички клиенти, чиито идентификатори са 1 или 10, или 87:

```
SELECT * FROM Customers
```

```
WHERE CustomerID IN (1, 10, 87)
```

- LIKE '*шаблон на символен низ*' сравнява символните низове със зададения шаблон и връща TRUE, ако низът отговаря на шаблона. За съставяне на шаблон се използват следните обобщаващи символи:
 - символът долна черта () означава един произволен символ;
 - символът процент (%) означава последователност от произволни символи (включително 0 на брой);
 - [] означава един знак, намиращ се в указаната област (например [а-д]) или указаното множество (например [аверХЗ] или [87м-пу-] означава измежду символите 8, 7, м÷п, у, -);
 - [^] означава един знак, намиращ се извън указаната област (например [^а-д]) или извън указаното множество ([^аверХЗ]);
 - за търсене на низ, в който има % или _, трябва да се дефинира *управляващ символ (escape character)*. Управляващият символ се използва непосредствено преди знака % или _ и означава, че знакът % или _ трябва да се интерпретира буквално – като обикновен символ, а не като обобщаващ символ.

Пример 8 Извеждане на всички клиенти, съдържащи в email адреса си знака долна черта (_).

```
SELECT * FROM Customers
WHERE EmailAddress LIKE '%/_%' ESCAPE '/'
```

Пример 9 Извеждане на редовете от Categories, съдържащи в колоната Notes низа '87%'.

```
SELECT * FROM Categories
WHERE Notes LIKE '%87!%%' ESCAPE '!'
```

- възможно е да се огради специалния символ в [], за да се използва като обикновен символ.

Пример 10 Извличане на данните за служителите, чиито email адреси започват с 'ivan_':

```
SELECT * FROM Employees
WHERE EmailAddress LIKE 'ivan!_%' ESCAPE '!'
```

или

```
SELECT * FROM Employees
WHERE EmailAddress LIKE 'ivan[_]%'
```

Ако символен низ съдържа символа апостроф и трябва да бъде заграден с апострофи, представянето на апострофа в низа се осъществява чрез два апострофа.

Пример 11 За извеждане на имената и адресите на тези клиенти, за които колоната Address съдържа апостроф:

```
SELECT CompanyName, Address
FROM Customers
WHERE Address LIKE '%''%'
```

Ключовата дума DISTINCT предоставя начин за ограничаване на изходните данни само до уникалните стойности. DISTINCT се поставя в конструкцията SELECT.

Пример 12

```
SELECT DISTINCT City FROM Customers
```

Премахват се повтарящите се редове от крайния резултат и се връщат само различните градове, в които има клиенти.

Друга възможност е сортирането на информацията, получена от SELECT заявката, като се използва изразът ORDER BY, указвайки една или повече колони. Синтаксисът на изразът е:

```
ORDER BY { column_name [sort_order] } [,...]
```

Указването на ред за сортиране не е задължително и се осъществява чрез ключовите думи ASC или DESC за сортиране в нарастващ или намаляващ ред. По подразбиране редовете се сортират в нарастващ ред и ASC може да се пропусне.

Пример 13 Заявка, която избира колони от таблицата за служителите и извежда резултатния набор от редове, сортирани в азбучен ред.

```
SELECT FirstName, LastName, PhoneNumber,
       HireDate, TerminationDate
FROM Employees
ORDER BY FirstName ASC, LastName ASC
```

Пример 14 За разлика от предишния пример тази заявка избира редовете от таблицата за служителите, които не са напуснали (нямат въведена дата на напускане

TerminationDate) и сортира резултатния набор от редове по дата на постъпване в низходящ ред; редовете с една и съща дата на постъпване HireDate се подреждат по азбучен ред.

```
SELECT FirstName, LastName, PhoneNumber,
       HireDate, TerminationDate
FROM Employees
WHERE TerminationDate IS NULL
ORDER BY HireDate DESC, FirstName ASC, LastName ASC
```

Може да се посочат номерата в последователността на избраните колони в списъка с имената на колоните на SELECT вместо техните имена, за да се определят колоните, които да се използват за подреждане на изходните данни.

Пример 15 Заявка, която избира колони от таблицата за клиенти от даден град и извежда резултатния набор от редове, сортирани в азбучен ред.

```
SELECT CompanyName, Address, PhoneNumber
FROM Customers
WHERE City = 'Велико Търново'
ORDER BY 1
```

Задачи

Задача 1. Да се напише заявка, която извежда всички колони от таблицата за продажбите, направени от служителите с идентификатори 2, 5, 12 или 90 в периода между 01/10/2003 и 31/12/2003.

Задача 2. Да се напише заявка, която извежда данните за продуктите, които имат критично налично количество и продажбата им не е преустановена.

Задача 3. Да се напише заявка, която извежда данните за служителите, родени през 1970 година.

Задача 4. Да се напише заявка, която извежда данните за служителите, чиито фамилии започват с някоя от буквите А, У, Д÷И или не започват с буквите Б, Л, М, Р÷Я.

Задача 5. Да се напише заявка, която извежда данните за клиентите, чиито имена (CompanyName) не съдържат цифри.

Задача 6. Да се напише заявка, която извежда данните за продуктите, които имат имена, състоящи се само от буквите а, б, в, д, и, о, п÷я или интервал (т.е. имената на продуктите да са съставени от всички или част от изброените символи).

Задача 7. Да се зададе ограничение за валидност:

7.1. на колоната EGN в таблицата Employees, с което да се гарантира въвеждането на точно 10 цифри;

7.2. на колоната PhoneNumber (в таблиците Employees, Stores, Suppliers, Customers), с което да се гарантира въвеждане на произволна комбинация от цифри, тирета и интервали.

Задача 8. Какви ще бъдат изходните данни от следните заявки:

8.1.

```
SELECT * FROM Employees
WHERE TerminationDate IS NOT NULL
      AND LastName LIKE 'A%'
```

8.2.

```
SELECT * FROM Sales
WHERE (TotalForSale >= 100 OR TotalForsale <=10)
      AND SaleDate BETWEEN '01/01/2003' AND '02/01/2003'
```

8.3.

```
SELECT DISTINCT EmployeeID
FROM Sales
WHERE CustomerID IN (5, 10, 18, 87)
```

8.4.

```
SELECT * FROM Stores
WHERE City IN ('Велико Търново', 'София', 'Варна')
      OR City LIKE 'G%a'
ORDER BY StoreName
```

Обобщаване на данните с помощта на агрегатни функции

Заявките позволяват създаването на обобщения на данните с помощта на агрегатни (обобщаващи) функции:

- COUNT връща броя на редовете или стойностите в дадена колона, различни от NULL, които са избрани от конкретна заявка.

Пример 1 Следната заявка връща броя на клиентите от даден град:

```
SELECT COUNT(CustomerID) AS [Count of Customers]
FROM Customers
WHERE City = 'Велико Търново'
```

Резултатът от тази заявка (фиг. 1) е таблица от една колона – Count of Customers и един ред, който съдържа брой на клиентите от град Велико Търново, т.е. броя на редовете в таблицата Customers, удовлетворяващи условието на заявката.

Count of Customers
4

Фиг. 1 Примерен резултат от заявката в пример 1

За преброяване на всички редове в дадена таблица се използва функцията COUNT, като се постави знак звездичка (*) вместо име на колона.

Пример 2 Следната заявка извежда броя на всички редове в таблицата за клиентите:

```
SELECT COUNT(*) AS [Count of Customers]
FROM Customers
```

Функцията COUNT може да се използва, за да се получи броя на различните стойности в съответната колона, като се комбинира с DISTINCT.

Пример 3 Заявка за преброяване на служителите, за които има текущо записани продажби в таблицата Sales:

```
SELECT COUNT(DISTINCT EmployeeID) AS [Count of Employees]
FROM Sales
```

Ако не е зададена ключовата дума DISTINCT, се получава броя на всички редове, които имат стойности в колоната идентификатор на служител (EmployeeID), различни от NULL, т.е. извършва се преброяване на всички продажби, тъй като колоната EmployeeID е декларирана като не допускаща стойност NULL.

- SUM връща сумата на всички стойности в дадена колона от числов тип, като игнорира стойностите NULL.

Пример 4 Заявка за намиране на количеството, продадено от даден продукт:

```
SELECT SUM(Quantity) AS SaledQuantity
FROM SaleDetails
WHERE ProductID = 1
```


Пример 5 Заявка за намиране на общата сума от дадена продажба:

```
SELECT SUM(Price*Quantity*(1-Discount)) AS Total
FROM SaleDetails
WHERE SaleID = 5
```

Допуска се използване на DISTINCT за сумирането само на уникалните стойности.

- AVG връща средната аритметична стойност на стойностите в дадена колона от числов тип, като игнорира стойностите NULL.

Пример 6 Заявка за намиране на средната доставна цена на всички продукти, чиято продажба не е преустановена:

```
SELECT AVG(price) AS Average
FROM Products
WHERE Discontinued = 0
```

Допуска се използване на DISTINCT за включване само на различните стойности в пресмятането на средната аритметична стойност.

Включването на GROUP BY позволява редовете от таблицата да се групират по някакъв критерий преди да се приложи обобщаващата функция.

Пример 7 За извеждане на идентификатор на магазин и броя на служителите, работещи в съответния магазин, се използва следната заявка:

```
SELECT StoreID, COUNT(*) AS CountOfEmployees
FROM Employees
GROUP BY StoreID
```

Резултатът от тази заявка (фиг. 2) е таблица от две колони – StoreID и CountOfEmployees и толкова реда, колкото са различните магазини, в които има служители.

StoreID	CountOfEmployees
1	4
2	2

Фиг. 2 Примерен резултат от заявката в пример 2

Пример 8 За извеждане на идентификатор на продажба и обща сума на съответната продажба се използва следната заявка:

```
SELECT SaleID, SUM(price*quantity*(1-discount)) AS Total
FROM SaleDetails
GROUP BY SaleID
```

Пример 9 За извеждане на идентификатор на клиент, идентификатор на служител и броя на продажбите, осъществени от съответния служител на съответния клиент, се използва следната заявка:

```
SELECT CustomerID, EmployeeID, COUNT(*) AS CountOfSales
FROM Sales
GROUP BY CustomerID, EmployeeID
```

- MAX връща най-голямата стойност в дадена колона.

Пример 10 Заявка за намиране на най-голямата продажна цена на даден продукт:

```
SELECT MAX(price) AS [Max Price]
FROM SaleDetails
WHERE ProductID = 5
```

- MIN връща най-малката стойност в дадена колона.

Пример 11 Заявка за намиране на идентификатор на клиент и първата и последната дата, на която е пазарувал съответния клиент:

```
SELECT CustomerID, MIN(SaleDate) AS FirstDate,
                    MAX(SaleDate) AS LastDate
FROM Sales
GROUP BY CustomerID
```

- Функции за статистически анализ StDev (стандартно отклонение) и VAR (дисперсия).

Условният оператор HAVING дава възможност за поставяне на условие за данните, които се получават в резултат на прилагане на GROUP BY.

Пример 12 Заявка за намиране на идентификаторите на клиенти и сумите от общите стойности на покупките на съответните клиенти, по-големи или равни на дадена сума. Резултатният набор от редове е сортиран по сумата на общите стойности на продажбите в низходящ ред.

```
SELECT CustomerID, SUM(TotalForsale) AS Total
FROM Sales
GROUP BY CustomerID
HAVING SUM(TotalForSale) >= 100
ORDER BY Total DESC
```

Задачи

Задача 1. Да се напише заявка, която извежда броя на продуктите, чиято продажба е преустановена.

Задача 2. Да се напише заявка, която извежда идентификатор на продукт и средната продажна цена на съответния продукт.

Задача 3. Да се напише заявка, която извежда идентификатор на служител и броя на продажбите, извършените от съответния служител за периода от време между 01 март 2004 г. и 01 април 2004 г. Резултатният набор от редове да бъде сортиран по броя на продажбите в низходящ ред.

Задача 4. Да се напише заявка, която извежда броя на различните градове, в които има магазини.

Задача 5. Да се напише заявка, която извежда първия клиент по азбучен ред, чието име започва с Л.

Задача 6. Да се напише заявка, която извежда продажбата с най-малка стойност за всеки служител.

Задача 7. Да се напише заявка, която извежда идентификатор на продукт и сума от продаденото количество за съответния продукт.

Задача 8. Да се напише заявка, която извежда идентификатор на продукт, най-ниската и най-високата продажна цена на съответния продукт, както и разликата на двете цени.

Задача 9. Да се напише заявка, която извежда идентификаторите на клиентите и средните суми на общите стойности на покупките на съответните клиенти, по-малки от 50. За получаване на обобщения резултат да се използват само продажбите, направени от служители с идентификатори 2, 5, 56 или 187. Резултатният набор от редове да бъде сортиран по средната сума на покупките в низходящ ред.

Определяне на броя на редовете за избор

Ключовата дума TOP, използвана в списъка за селекция, позволява контрол над количеството на данните, които да бъдат извлечени, т.е. определя броя на редовете за избор, като взема първите *n*. Синтаксисът за използване на TOP е:

```
SELECT [TOP n [PERCENT] [WITH TIES]] column_list
FROM table_name
...
```

Когато заявката включва ORDER BY, TOP прави така, че в резултата да присъстват само първите *n* реда, подредени според израза ORDER BY. Когато заявката не включва израза ORDER BY, не е възможно да се предвиди кои редове ще бъдат върнати.

Пример 1 За извеждане на първите три реда от таблицата за продуктите, сортирани по доставната им цена в намаляващ ред, може да се използва следната заявка:

```
SELECT TOP 3 ProductID, ProductName, Price
FROM Products
ORDER BY Price DESC
```

Ако обаче има продукти с доставна цена, равна на доставната цена на продукта от третия ред, изведен от горната заявка, те няма да се видят. Ако целта е да се изведат всички продукти, които имат същата цена като изброените в списъка, трябва да се използва опцията WITH TIES. Тази опция е допустима само ако конструкцията SELECT включва ORDER BY.

Пример 2

```
SELECT TOP 3 WITH TIES ProductID, ProductName, Price
FROM Products
ORDER BY Price DESC
```

Заявката от пример 2 може да върне повече от три реда, ако има повече продукти с цена, равна на третия по ред с най-висока цена, т.е. при тази операция първо се преброяват редовете, указани от TOP (в този случай три). След като бъдат върнати първите три реда, SQL Server ще провери дали други редове имат стойност, равна на последната върната.

Ако трябва да се изведе определена част от редовете, може да се използва TOP с PERCENT, при което се извършва закръгляне до най-близкото цяло число за брой редове.

Пример 3

```
SELECT TOP 20 PERCENT ProductID, ProductName, Price
FROM Products
ORDER BY Price DESC
```

Броят на редовете, върнати от заявката от пример 3, се определя, като се изчисли 20% от общия брой редове в таблицата за продуктите и резултатът се закръгли до най-близкото цяло число.

Опциите WITH TIES и PERCENT могат да бъдат използвани заедно в заявка, в резултат на което е възможно да се получат допълнителни редове, които имат цена, равна на последната върната.

Пример 4

```
SELECT TOP 20 PERCENT WITH TIES
       ProductID, ProductName, Price
FROM Products
ORDER BY Price DESC
```

Опцията SET ROWCOUNT n се използва за ограничаване на изпращаните от SQL Server до клиента редове до n .

Пример 5

```
SET ROWCOUNT 10
SELECT * FROM Customers
```

Конструкцията SELECT спира да изпраща редове обратно на клиента, след като са били изпратени n реда, т.е. последната заявка ще върне само 10 реда от таблицата за клиентите. Тази опция е полезна при определени ситуации, но има по-ограничена функционалност. Оказва влияние и за конструкции за добавяне, изтриване и актуализиране на редове. За да се възстанови подразбиращото се състояние, при което се връщат или повлияват от модификация всички редове, се използва опцията за $n = 0$, т.е.

```
SET ROWCOUNT 0
```

Задачи

Задача 1. Да се напише заявка, която извежда 5^{те} най-високи продажни цени на даден продукт (например с идентификатор единица).

Задача 2. Да се напише заявка, която извежда 10^{те} най-ниски общи суми на продажби за периода между 01 октомври 2003 г. и 01 февруари 2004 г.

Задача 3. Да се напише заявка, която извежда данните за продажбите с 10^{те} най-високи общи суми на продажби за периода между 01 октомври 2003 г. и 01 февруари 2004 г., в резултат на която да е възможно да се получат допълнителни редове с обща сума, равна на последната върната.

Използване на изрази

Изрази за дата/час и интервални изрази

Transact-SQL предоставя набор от оператори за извършване на аритметични действия, сравняване на стойности, конкатениране на низове, логически оператори. Изразите могат да се използват в SELECT списъка, в условието на WHERE или HAVING.

- аритметични оператори: +, -, *, / , % (остатък при целочислено деление).

Пример 1

```
SELECT SaleID, ProductID, Quantity, Price, Discount,
       Quantity*Price*(1-Discount) AS ExtendedPrice
FROM SaleDetails
```

- математически функции:

ABS(numeric_expr) връща абсолютната стойност на числовия израз *numeric_expr*;

ACOS(float_expr) връща ъгъла в радиани, чийто косинус е зададения числов израз *float_expr*;

ASIN(float_expr) връща ъгъла в радиани, чийто синус е зададения числов израз *float_expr*;

ATAN(float_expr) връща ъгъла в радиани, чийто тангенс е зададения числов израз *float_expr*;

COS(float_expr) връща тригонометричния косинус на даден ъгъл (в радиани), зададен в числовия израз *float_expr*;

SIN(float_expr) връща тригонометричния синус на даден ъгъл (в радиани), зададен в числов израз;

TAN(float_expr) връща тригонометричния тангенс на даден ъгъл (в радиани), зададен в числов израз;

COT(float_expr) връща тригонометричния котангенс на даден ъгъл (в радиани), зададен в числов израз;

DEGREES(numeric_expr) връща градусите, превърнати от радиани от числовия израз;

RADIANS(numeric_expr) връща радианите, превърнати от градуси от числовия израз;

PI() връща константата 3.1415...;

POWER(numeric_expr, y) връща стойността на числовия израз *numeric_expr* на степен *y*;

SQUARE(float_expr) връща квадрата на зададения числов израз;

SQRT(float_expr) връща квадратния корен от зададения числов израз;

EXP(float_expr) връща експоненциалната стойност на зададения числов израз;

LOG(float_expr) връща натуралния логаритъм на зададения числов израз;

LOG10(float_expr) връща логаритъма при основа 10 на зададения числов израз;

SIGN(numeric_expr) връща знака на числовия израз, т.е. -1 – при отрицателна стойност; 0 – при стойност 0; 1 – при положителна стойност;

CEILING(numeric_expr) връща цяло число, най-малко от всички стойности, по-големи или равни на аргумента;

`FLOOR(numeric_expr)` връща цяло число, най-голямо от всички стойности, по-малки или равни на аргумента;

`ROUND(numeric_expr, precision [, truncate])` – ако точността *precision* е положително число, числовият израз се закръгля до толкова позиции след десетичната запетая, колкото са посочени в точността; ако точността е отрицателно число, числовият израз се закръгля до толкова позиции вляво от запетаята, колкото са посочени в точността; числото *truncate* определя дали числовият израз се закръгля или отрязва – ако липсва или е 0, се закръгля; ако е различно от 0, се отрязват толкова знаци, колкото е точността. Например `ROUND(70.46, 0, 1)` връща цялата част на числото, т.е. 70; `ROUND(75.81, -1, 0)` връща 80; `ROUND(75.81, -1, 1)` връща 70;

`RAND([seed])` връща случайно генерирана стойност – число с плаваща запетая между 0 и 1; аргументът *seed* е начална стойност за алгоритмите, генериращи случайни числа (от целочислен тип) и не е задължителен.

Функциите `ABS(numeric_expr)`, `CEILING(numeric_expr)`, `FLOOR(numeric_expr)`, `ROUND(numeric_expr, precision, [truncate])` са най-използвани за търсене на стойност в определен диапазон.

- функции за дата и час:

`DATEADD(interval, number, datetime)` връща дата, добавяйки интервал към зададена дата. Параметърът *interval* може да има една от следните стойности:

година	year, yy, yyyy
тримесечие	quarter, q, qq
месец	month, m, mm
седмица	week, wk, ww
ден	day, d, dd
пореден ден от годината	dayofyear, dy, y
пореден ден от седмицата	weekday, dw, w
час	hour, hh
минута	minute, mi, n
секунда	second, s, ss
милисекунда	millisecond, ms

Параметърът *number* е брой интервали (дни, месеци и т.н.). При отрицателна стойност на този параметър се извършва връщане назад от датата в *datetime*, а при положителна добавяне към нея. Параметърът *datetime* е датата, към която ще се добавя или връща интервал. Ако е константа, трябва да се ограда с апострофи; може да е колона от таблица, оградена при необходимост с [].
Например, `DATEADD(day, 30, '1/1/2004')` връща датата 31 януари 2004 г.

`DATEDIFF(interval, datetime1, datetime2)` връща цяло число, означаващо разликата между две дати. Възможните стойности на параметъра *interval* са същите като при `DATEADD`. Резултатът от функцията е цяло число, което е положително, ако *datetime1* е преди *datetime2*; отрицателно, ако *datetime2* е преди *datetime1*; 0, ако двете дати съвпадат по отношение на зададения интервал.

`DATEPART(interval, datetime)` връща цяло число, представящо зададената чрез *interval* част от дата.

DATENAME(*interval*, *datetime*) връща низ, представящ зададената чрез *interval* част от дата.

GETDATE() връща текущата дата и час в стандартния вътрешен формат на SQL Server за стойности на типа *datetime*.

Пример 2 Заявка, извличаща имената на служителите и броя на годините между датата на назначаване и текущата дата за служителите, които не са напуснали.

```
SELECT FirstName + ' ' + LastName AS Name,
       DATEDIFF(year, HireDate, GetDate()) AS Duration
FROM Employees
WHERE TerminationDate IS NULL
ORDER BY Duration DESC
```

На фигура 1 е показан примерен резултат от тази заявка.

Name	Duration
Ваня Христова	14
Стоян Георгиев	11
Катя Цветанова	6
Стефа Миланова	3

Фиг. 1 Примерен резултат от заявката в пример 2

Пример 3 Заявка, извличаща имената и датите на назначаване на служителите, назначени за последните 10 години.

```
SELECT FirstName + ' ' + LastName AS Name,
       HireDate
FROM Employees
WHERE DateDiff(year, HireDate, GetDate()) <= 10
ORDER BY Name ASC
```

Пример 4 Заявка, извличаща годините, през които са назначавани служители и броя на назначените служители през съответните години.

```
SELECT DATEPART(year, HireDate) AS Year,
       COUNT(*) AS CountOfEmpl
FROM Employees
GROUP BY DATEPART(year, HireDate)
```

Чрез опцията SET DATEFORMAT се променя подразбиращия се формат за датата за текущата конекция. Например чрез:

```
SET DATEFORMAT dmy
```

се установява *dmy* вместо подразбиращата се стойност на опцията *mdy*. За да се избегнат грешки, е добре да се използва формата *ууууmmdd* (или *уумmdd*), който се разпознава независимо от това каква е настройката за DATEFORMAT.

Задачи

Задача 1. Да се напише заявка, която да изведе идентификатор на клиент и сума от общата стойност на покупките на съответния клиент през изминалия месец.

Задача 2. Да се напише заявка, която да изведе данните за всички служители, назначени:

2.1. след датата 01 май 1999 г.;

2.2. през юни 2000 година.

Задача 3. Да се напише заявка, която да изведе идентификатор на служител и брой на осъществените продажби от съответния служител:

3.1. за текущата дата;

3.2. за вчерашна дата.

Задача 4. Да се напише заявка, която да извежда сума от общата стойност на продажбите по месеци за текущата година. Резултатът от заявката да има вида:

Num	Month	SumTotal

1	January	632.9750
2	February	785.7500
3	March	122.7750

Функции за работа със символни низове и изрази за конвертиране на типа данни

Функции за работа със символни низове

В SQL Server функциите за работа със символни низове позволяват да се получат части от низове, да се променят низове, да се получи информация за самите низове. Функциите за низове правят работата със символни данни по-лесна:

`LEN(string)` връща броя на символите в низа.

`LEFT(string, number)` връща *number* на брой символи от низа *string*, започвайки от началото на низа.

`RIGHT(string, number)` връща *number* на брой символи от низа *string*, започвайки от края на низа.

`SUBSTRING(string, start, length)` връща част от низа *string*, като взема *length* символа от *start* позиция.

`UPPER(string)` връща символния низ, като заменя малките букви в него с главни.

`LOWER(string)` връща символния низ, като заменя главните букви в него с малки.

`CHARINDEX(str_to_find, str_to_search [, start_position])` връща началната позиция на *str_to_find* в *str_to_search*; *start_position* определя стартовата позиция, откъдето започва търсенето в *str_to_search* (ако липсва, търсенето започва от първата позиция).

`PATINDEX(pattern, string)` връща позицията в низа *string*, в който за първи път се среща съответствие на шаблона *pattern* или 0, ако шаблонът не е намерен. Шаблонът *pattern* може да бъде произволна последователност от символи, включително и обобщаващи символи, използвани с оператора `LIKE`.

`LTRIM(string)` връща низа, като отстранява евентуалните интервали, с които започва.

`RTRIM(string)` връща низа, като отстранява евентуалните интервали, с които завършва.

`REVERSE(string)` връща низа, като разполага знаците, от които се състои в обратен ред.

`REPLICATE(string, number)` връща низ, в който посоченият първи аргумент е повторен толкова пъти, колкото е *number*.

`SPACE(number)` връща низ, състоящ се от *number* на брой интервала.

`REPLACE(str_to_search, str_to_find, str_to_replace_with)` замества всички срещания на *str_to_find* в *str_to_search* със *str_to_replace_with*.

`STUFF(string_to_modify, start, length, string_to_insert)` трансформира низа *string_to_modify*, като изтрива *length* на брой символи от позиция *start*, вмъквайки *string_to_insert* в *string_to_modify* от позиция *start*.

`ASCII(string)` връща стойността на ASCII кода на най-левия знак в символния низ.

`UNICODE(string)` връща стойността на Unicode кода на най-левия знак в символния низ.

`CHAR(integer_expr)` връща символа, съответстващ на ASCII кода *integer_expr*, който трябва да бъде стойност от 0 до 255. Ако стойността на аргумента е извън тези граници, функцията връща стойност `NULL`.

`NCHAR(integer_expr)` връща Unicode символа, съответстващ на кода `integer_expr`, дефиниран в стандарта Unicode.

`SOUNDEX(string)` връща четирисимволен код на низа, който се използва за намиране на еднакво звучащи думи.

`DIFFERENCE(string1, string2)` връща разликата между стойностите на функцията `SOUNDEX` за двата низа. Резултатът е цяло число между 0 и 4, определящо броя на еднаквите символи в `SOUNDEX` стойностите. Стойност 4 означава еднакво звучащи думи, различаващи се евентуално само в изписването.

`QUOTENAME(string [, 'quote_character'])` връща низ от Unicode символи, ограден с `quote_character`, който може да е ' или ". Ако не е зададен ограничител, се използват квадратни скоби. Тази функция позволява използването на входния низ `string` като идентификатор.

Конкатениране на низове (+), например:

```
SELECT FirstName + ' ' + LastName AS Name
FROM Employees
```

Интервали в края

Без разрешаване на опцията `ANSI_PADDING` (`SET ANSI_PADDING {ON | OFF}`) интервалите в края за колоните с променлива дължина се отрязват, както и тези с фиксирана, които позволяват стойност `NULL`. Запазват се само интервалите в края за колони с фиксирана дължина, не допускащи стойност `NULL`. Ако опцията е включена, винаги се запазват интервалите в края.

ODBC драйверът на SQL Server и OLE DB Provider за SQL Server автоматично установяват `ANSI_PADDING` в `ON` при осъществяване на конекция. Това може да бъде конфигурирано в ODBC източниците на данни, в ODBC атрибутите на конекцията или в свойствата на OLE DB конекцията, зададени в приложението преди осъществяване на конекцията.

Изрази за конвертиране на типа данни

SQL Server предоставя три функции за конвертиране на типове данни:

`CAST(original_expression AS desired_datatype)` конвертира даден израз от определен тип данни в израз от друг тип данни. Позволява променяне на типа на данните, когато се избира поле, което е необходимо при конкатениране на низове или при съединяване на колони, които първоначално не са били замислени като свързани, за изпълнение на математически операции върху колони, които са били дефинирани като символни низове, но в действителност съдържат числа.

Пример 1

```
SELECT LastName + '-' +
       CAST(EmployeeID AS varchar(4)) AS [Name and ID]
FROM Employees
```

`STR(float_expression [, length [, decimal]])` връща низ, представящ число с плаваща точка; `length` е общия брой знаци, които да съставят получения низ, включвайки десетичната точка (.) и отрицателния знак (-), ако съществуват (стойността по подразбиране е 10); `decimal` е брой на разрешените позиции след десетичната точка. Тази функция закръглява до поисканата десетична позиция.

Пример 2

```
SELECT 'Total for sale is:' + STR(TotalForSale, 13, 2) AS Total
FROM Sales
```

`CONVERT(datatype, expression [, style])` преобразува израза *expression* от един тип данни в друг. Опционалният параметър *style* се използва при преобразуване на *datetime* или *smalldatetime* в тип *char*, *nchar*, *varchar* или *nvarchar* и при конвертиране на *float*, *real*, *money* или *smallmoney* в символен тип данни. При преобразуване на *datetime* или *smalldatetime* в низ, ако *style* се пропусне, се използва подразбиращия се формат за дата на SQL Server. Възможните стойности на *style* са:

0 или 100 (подразбиращ се код)	mon dd yyyy hh:mm AM (или PM)
1	mm/dd/yy
2	yy.mm.dd
3	dd/mm/yy
4	dd.mm.yy
5	dd-mm-yy
6	dd mm yy
7	mon dd, yy
8 или 108	hh:mm:ss
9 или 109	mon dd yyyy hh:mm:ss:mmm AM (или PM)
10	mm-dd-yy
11	yy/mm/dd
12	yymmdd
13 или 113	dd mon yyyy hh:mm:ss:mmm (24h)
14 или 114	hh:mm:ss:mmm (24h)
20 или 120	yyyy-mm-dd hh:mm:ss (24h)
21 или 121	yyyy-mm-dd hh:mm:ss:mmm (24h)

При преобразуване на *float* или *real* в символен тип данни стилът определя за реалните числа дали резултатът да бъде показан в научно означение. Възможните стойности на *style* са:

0 (по подразбиране)	Най-много 6 цифри. Използва се научно означение, когато е подходящо.
1	Винаги 8 цифри и научно означение.
2	Винаги 16 цифри и научно означение.

Когато се преобразува от тип данни *money* или *smallmoney* в символен тип, стилът определя наличието на запетая на всеки три цифри отляво на десетичната точка и броя на цифрите след десетичната точка – две или четири. Възможните стойности на *style* са:

0 (по подразбиране)	Без запетая отляво на десетичната точка и две цифри отлясно на десетичната точка.
1	Със запетая на всеки три цифри отляво на десетичната точка и две цифри отлясно на десетичната точка.
2	Със запетая на всеки три цифри отляво на десетичната точка и четири цифри отлясно на десетичната точка.

Задачи

Задача 1. Да се напише заявка, която извежда имената и датата на раждане и възрастта на служителите, като се използва ЕГН.

Задача 2. Да се напише заявка, която извежда дата на продажба, броя на клиентите, направили покупки на тази дата, броя на служителите, осъществили продажби, броя на продажбите, средната стойност на продажбите и общата стойност на продажбите.

Решение:

```
SELECT CAST(CONVERT(char(10), SaleDate, 112)
          AS datetime) AS DateOfSale,
       COUNT(DISTINCT CustomerID) AS CustomersCount,
       COUNT(DISTINCT EmployeeID) AS EmployeesCount,
       COUNT(SaleID) AS SalesCount,
       AVG(TotalForSale) AS AverageTotal,
       SUM(TotalForSale) AS SumTotal
FROM Sales
GROUP BY
       CAST(CONVERT(char(10), SaleDate, 112) AS datetime)
```

Конвертира се стойността на датата и часа на продажба в низ, който не включва времето (т.е. часа, минутите, секундите, милисекундите). Полученият низ се преобразува в *datetime*, при което се използва подразбиращия се час – полунощ. Целта е да се извърши групиране и обобщаване на съответните данни по дата, като се изключи частта за времето.

Задача 3. Да се напише заявка, която извежда датите и сумите от общите стойности на продажбите, извършени на съответните дати, като взема само първите с 5^{те} най-високи суми от общите стойности на продажби. В резултата да е възможно да се получат допълнителни редове с обща сума, равна на последната върната.

Условни (CASE) изрази

Условният израз може да върне една от няколко стойности, в зависимост от това кое условие е изпълнено. Използват се две форми на израза **CASE** – за проверяване за определена стойност на даден израз и за проверяване за стойности на повече изрази. Синтаксисът при първия случай е:

```
CASE value_expression
  { WHEN value_expression
    THEN {value_expression | NULL} }...
  [ELSE {value_expression | NULL}]
END
```

Условията се проверяват подред и първото условие, което се удовлетворява, определя резултата. Ако ELSE се пропусне, се подразбира ELSE NULL.

Пример 1

```
SELECT ProductName,
       CASE discontinued
         WHEN 0 THEN 'Continued'
         WHEN 1 THEN 'Discontinued'
         ELSE 'Unknown'
       END AS DiscontinuedProducts
FROM Products
```

Синтаксисът при втория случай на използване на CASE е:

```
CASE
  {WHEN boolean_expression
    THEN {value_expression | NULL}}...
  [ELSE {value_expression | NULL}]
END
```

Пример 2

```
SELECT SaleID, ProductID, Quantity, Price,
       CASE
         WHEN Quantity >= 20 THEN 0.05
         WHEN Quantity >= 15 THEN 0.03
         WHEN Quantity >= 10 THEN 0.02
         ELSE 0
       END AS ComputedDiscount
FROM SaleDetails
```

SQL Server предоставя три производни на CASE: NULLIF и COALESCE, които са част от ANSI SQL92; ISNULL е допълнителна функция в SQL Server, която не е от ANSI SQL92.

NULLIF приема две стойности за аргументи. Ако двете стойности съвпадат, резултатът е NULL; в противен случай, резултатът е първата от двете стойности, т.е. NULLIF(value_expression1, value_expression2) може да се запише чрез еквивалентен CASE израз по следния начин:

```
CASE value_expression1
  WHEN value_expression2 THEN NULL
```

```
ELSE value_expression1
END
```

COALESCE връща първия израз, който не е NULL в списъка от изрази. Ако всички стойности са NULL, COALESCE извежда NULL. Следователно COALESCE(value_expression1, ..., value_expressionN) може да се запише чрез еквивалентен CASE израз по следния начин:

```
CASE
  WHEN value_expression1 IS NOT NULL
    THEN value_expression1
  WHEN value_expression2 IS NOT NULL
    THEN value_expression2
  ...
  ELSE value_expressionN
END
```

Пример 3 Заявка за извеждане на следващата стойност в колоната идентификатор на продукт чрез намиране на най-високата текуща стойност в колоната идентификатор на продукт (ProductID) от таблицата за продуктите (Products) и добавяне на единица (може да се използва в конструкцията за добавяне на нов ред за задаване на стойност на колоната ProductID):

```
SELECT COALESCE(MAX(ProductID), 0) + 1 AS NextProductID
FROM Products
```

ISNULL приема две стойности за аргументи и връща първата, ако тя не е NULL и втората, в противен случай. Следователно ISNULL(value_expression1, value_expression2) може да се запише чрез еквивалентен CASE израз по следния начин:

```
CASE
  WHEN value_expression1 IS NOT NULL
    THEN value_expression1
  ELSE value_expression2
END
```

Пример 4 Заявка, извеждаща имената и цените на продуктите. Ако продуктът има стойност NULL за колоната price, се извежда 0.

```
SELECT ProductName, ISNULL(price, 0) AS ProductPrice
FROM Products
```

Пример 5 Заявка, извеждаща имената и цените на продуктите. Ако продуктът има стойност NULL за колоната price, се извежда най-ниската цена, която съществува за някой продукт.

```
SELECT ProductName,
  ISNULL(price, (SELECT MIN(price) FROM Products))
  AS ProductPrice
FROM Products
```

Задачи

Задача 1. Да се напише заявка, която извежда данните за продажбите и изчислената отстъпка в зависимост от стойността на TotalForSale: за обща сума на продажбите над 100 – 15%; между 80 и 100 – 10%; между 50 и 80 – 5%; под 50 – 0%.

Задача 2. Да се напише заявка, която извежда имената и доставните цени на продуктите, класифицирани като евтини, средно скъпи и скъпи.

Задача 3. Да се напише заявка, която извежда имената и наличните количества на продуктите. Ако продуктът има стойност NULL за колоната Stock или ако продуктът има стойност за колоната Stock, по-малка или равна на стойността на колоната ReorderLevel, да извежда 'Критично количество'; в противен случай да извежда разликата между наличното количество и количеството, което се счита за критично за съответния продукт.

Задача 4. Да се напише заявка, която извежда следващата стойност в колоната идентификатор на клиент за таблицата Customers.

Съединения на таблици

Чрез командата SELECT може да се укаже дадена заявка да връща определени редове или колони от една таблица. За извличане и обработване на данни от повече от една таблици се извършва съединяване на таблици, което обикновено се изразява в комбиниране на колоните на първичните и външните ключове (на съответните таблици) на съответстващите редове. В SQL синтаксиса ANSI JOIN са налице пет типа операции:

INNER JOIN

TableA **INNER JOIN** *TableB* **ON** *join_condition*

Вътрешното съединение, реализирано с оператора INNER JOIN, връща редовете от коя да е таблица само ако имат съответстващ ред от другата таблица, т.е. извеждат се всички редове, за които специфичното условие за съединение, указано в ON, е удовлетворено.

Пример 1 Заявка, чрез която се извличат имената на служителите, датите и общите стойности на осъществените от тях продажби.

```
SELECT FirstName, LastName,
       SaleDate, TotalForSale
FROM Employees
INNER JOIN Sales
    ON Employees.EmployeeID = Sales.EmployeeID
```

Пример 2 Заявка, извеждаща имената на служителите, дати на продажби, сума от общата стойност на продажбите, осъществени от съответния служител на съответната дата.

```
SELECT e.FirstName, e.LastName,
       CONVERT(datetime, CONVERT(char(10), s.SaleDate, 102))
       AS DateOfSale,
       SUM(TotalForSale) AS Total
FROM Employees AS e
INNER JOIN Sales AS s ON e.EmployeeID = s.EmployeeID
GROUP BY e.FirstName, e.LastName,
         CONVERT(datetime, CONVERT(char(10), s.SaleDate, 102))
```

В този пример е показан начин за задаване на псевдоним на таблица след FROM, като се използва ключовата дума AS. Общият вид на синтаксиса за указване на псевдоним на таблица е:

```
...
FROM table_name [AS] alias_name
INNER JOIN other_table_name [AS] other_alias_name
ON join_condition
...
```

Пример 3 Заявка, извеждаща имената на клиентите, датите на покупките, осъществени от съответните клиенти, имената, продажните цени и продадените количества на продуктите за съответните продажби за текущия ден.

```
SELECT c.CompanyName, s.SaleDate, p.ProductName,
       sd.Price, sd.Quantity
FROM Customers AS c
INNER JOIN Sales s ON c.CustomerID = s.CustomerID
```

```
INNER JOIN SaleDetails sd ON s.SaleID = sd.SaleID
INNER JOIN Products p ON sd.ProductID = p.ProductID
WHERE DATEDIFF(day, s.SaleDate, GetDate()) = 0
```

На фигура 1 е показан примерен резултат от тази заявка.

CompanyName	SaleDate	ProductName	Price	Quantity
Алтметал ЕООД	2011-07-07 10:32:49.327	ябълки	3.00	2.000
Алтметал ЕООД	2011-07-07 10:32:49.327	домати	2.75	1.000
МИЛСТОР 90 ЕООД	2011-07-07 10:32:49.327	портокали	4.00	5.000
МИЛСТОР 90 ЕООД	2011-07-07 10:32:49.327	картофи	3.50	2.000
МИЛСТОР 90 ЕООД	2011-07-07 10:32:49.327	шоколад	2.60	10.000
Деница Стар ЕООД	2011-06-07 10:32:49.327	чипс	1.80	1.000
Деница Стар ЕООД	2011-07-07 10:32:49.327	макарони	2.80	2.000
Деница Стар ЕООД	2011-07-07 10:32:49.327	еклери	1.50	3.000

Фиг. 1 Примерен резултат от заявката в пример 3

LEFT [OUTER] JOIN

TableA **LEFT [OUTER] JOIN** *TableB* **ON** *join_condition*

Лявото външно съединение, реализирано с оператора **LEFT OUTER JOIN**, връща редовете, за които съществува връзка между *TableA* и *TableB*, като освен това връща всички редове от *TableA*, за които няма съответстващ ред от *TableB*. Редовете в една таблица, които в процеса на съединяване не могат да образуват двойки с нито един от редовете на другата таблица, се наричат висящи (*dangling*). Следователно този вид съединение запазва висящите редове от *TableA*. При връщане на висящите редове от *TableA*, всички колони, избрани от *TableB*, се връщат като NULL.

Пример 4 Заявка, извеждаща имената на клиентите, точната дата и час на извършената за текущата дата покупка, имената на закупените продукти, продажната им цена и продаденото от тях количество; за клиенти, които не са пазарували, се извежда име, а всички останали колони се връщат като NULL; не се извежда информация за клиентите, които са пазарували поне веднъж, но не на текущата дата.

```
SELECT c.CompanyName, s.SaleDate, p.ProductName,
       sd.Price, sd.Quantity
FROM Customers AS c
LEFT OUTER JOIN Sales s ON c.CustomerID = s.CustomerID
LEFT OUTER JOIN SaleDetails sd ON s.SaleID = sd.SaleID
LEFT OUTER JOIN Products p ON sd.ProductID = p.ProductID
WHERE DATEDIFF(day, s.SaleDate, GetDate()) = 0
      OR s.SaleDate IS NULL
```

Примерен резултат от тази заявка е показан на фигура 2.

CompanyName	SaleDate	ProductName	Price	Quantity
Алтметал ЕООД	2011-07-07 10:32:49.327	ябълки	3.00	2.000
Алтметал ЕООД	2011-07-07 10:32:49.327	домати	2.75	1.000
Деница Стар ЕООД	2011-06-07 10:32:49.327	чипс	1.80	1.000
Деница Стар ЕООД	2011-07-07 10:32:49.327	макарони	2.80	2.000
Деница Стар ЕООД	2011-07-07 10:32:49.327	еклери	1.50	3.000
Мики-92 ЕООД	NULL	NULL	NULL	NULL
МИЛСТОП 90 ЕООД	2011-07-07 10:32:49.327	портокали	4.00	5.000
МИЛСТОП 90 ЕООД	2011-07-07 10:32:49.327	картофи	3.50	2.000
МИЛСТОП 90 ЕООД	2011-07-07 10:32:49.327	шоколад	2.60	10.000

Фиг. 2 Примерен резултат от заявката в пример 4

Пример 5 Заявка, извеждаща имената на продуктите, точната дата и час на извършената за текущата дата покупка, продажната им цена и продаденото от тях количество; за продукти, които не са купувани, се извежда име, а всички останали колони се връщат като NULL; за продукти, които са купувани поне веднъж, но не на текущата дата, не се извежда информация.

```
SELECT p.ProductName, s.SaleDate, sd.Price, sd.Quantity
FROM Products AS p
LEFT JOIN SaleDetails sd ON sd.ProductID = p.ProductID
LEFT JOIN Sales s ON sd.SaleID = s.SaleID
WHERE DATEDIFF(day, s.SaleDate, GetDate()) = 0
OR s.SaleDate IS NULL
```

RIGHT [OUTER] JOIN

TableA **RIGHT [OUTER] JOIN** *TableB* **ON** *join_condition*

Дясното външно съединение, реализирано с оператора RIGHT OUTER JOIN, връща редовете, за които съществува връзка между *TableA* и *TableB*, като освен това връща всички редове от *TableB*, за които няма съответстващ ред от *TableA*, т.е. този вид съединение запазва висящите редове от *TableB*. При връщане на висящите редове от *TableB*, всички колони, избрани от *TableA*, се връщат като NULL.

Пример 6 Заявката от пример 4 може да се запише чрез дясно външно съединение по следния начин:

```
SELECT c.CompanyName, s.SaleDate, p.ProductName,
       sd.Price, sd.Quantity
FROM Sales AS s
INNER JOIN SaleDetails sd ON s.SaleID = sd.SaleID
INNER JOIN Products p ON sd.ProductID = p.ProductID
RIGHT OUTER JOIN Customers c
ON s.CustomerID = c.CustomerID
WHERE DATEDIFF(day, s.SaleDate, GetDate()) = 0
OR s.SaleDate IS NULL
```

Пример 7 Заявката от пример 5 може да се запише чрез дясно външно съединение по следния начин:

```
SELECT p.ProductName, s.SaleDate, sd.Price, sd.Quantity
FROM Sales s
INNER JOIN SaleDetails sd ON s.SaleID = sd.SaleID
RIGHT JOIN Products p ON sd.ProductID = p.ProductID
```

```
WHERE DATEDIFF(day, s.SaleDate, GetDate()) = 0
      OR s.SaleDate IS NULL
```

FULL [OUTER] JOIN

```
TableA FULL [OUTER] JOIN TableB ON join_condition
```

Пълното външно съединение, реализирано с оператора FULL OUTER JOIN, връща редовете, за които съществува връзка между *TableA* и *TableB*. Освен това връща всички редове от *TableA*, за които няма съответстващ ред от *TableB*, както и всички редове от *TableB*, за които няма съответстващ ред от *TableA*, т.е. този вид съединение запазва висящите редове от *TableA* и *TableB*. При връщане на висящите редове от *TableA*, всички колони, избрани от *TableB*, се връщат като NULL, а при връщане на висящите редове от *TableB*, всички колони, избрани от *TableA*, се връщат като NULL.

Пример 8 Заявка, извеждаща имената на всички клиенти, включително и тези, които не са пазарували и имената на всички продукти, включително и тези, които не са купувани.

```
SELECT c.CompanyName, s.SaleDate, p.ProductName,
       sd.Price, sd.Quantity
FROM Customers AS c
FULL OUTER JOIN Sales s ON c.CustomerID = s.CustomerID
FULL OUTER JOIN SaleDetails sd ON s.SaleID = sd.SaleID
FULL OUTER JOIN Products p ON sd.ProductID = p.ProductID
```

На фигура 3 е показан примерен резултат от тази заявка.

CompanyName	SaleDate	ProductName	Price	Quantity
Алтметал ЕООД	2011-07-07 10:32:49.327	ябълки	3.00	2.000
Алтметал ЕООД	2011-07-07 10:32:49.327	домати	2.75	1.000
Деница Стар ЕООД	2011-06-07 10:32:49.327	чипс	1.80	1.000
Деница Стар ЕООД	2011-07-07 10:32:49.327	макарони	2.80	2.000
Деница Стар ЕООД	2011-07-07 10:32:49.327	еклери	1.50	3.000
МИЛСТОП 90 ЕООД	2011-07-07 10:32:49.327	портокали	4.00	5.000
МИЛСТОП 90 ЕООД	2011-07-07 10:32:49.327	картофи	3.50	2.000
МИЛСТОП 90 ЕООД	2011-07-07 10:32:49.327	шоколад	2.60	10.000
Мики-92 ЕООД	NULL	NULL	NULL	NULL
NULL	NULL	царевичен снакс	NULL	NULL

Фиг. 3 Примерен резултат от заявката в пример 8

CROSS JOIN

```
TableA CROSS JOIN TableB
```

Кръстосаното съединение, реализирано с оператора CROSS JOIN, връща всички редове от *TableA*, комбинирани с всички редове от *TableB*. При този вид съединение липсва ON, т.е. не се задава условие за съединяване чрез свързваща колона между таблиците, следователно CROSS JOIN връща декартовото произведение на двете таблици.

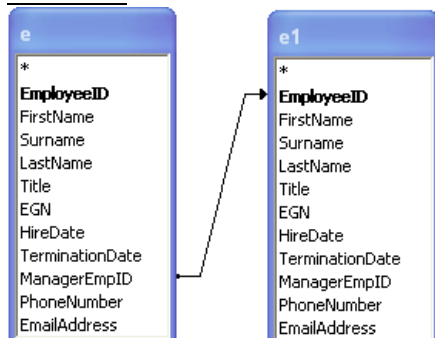
Пример 9

```
SELECT s.City AS Store_City, c.City AS Customer_City
FROM Stores s CROSS JOIN Customers c
```

Задачи

Задача 1. Да се напише заявка, която извежда идентификатор и фамилия на служител, фамилия на мениджъра на съответния служител.

Решение:



```
SELECT e.EmployeeID, e.LastName, e1.LastName AS Manager
FROM Employees AS e
INNER JOIN Employees AS e1
    ON e1.EmployeeID = e.ManagerEmpID
```

Ако колоната ManagerEmpID допуска стойност NULL, за да се изведат и служителите, които нямат мениджъри, се използва външно съединение по следния начин:

```
SELECT e.EmployeeID, e.LastName, e1.LastName AS Manager
FROM Employees AS e
LEFT JOIN Employees AS e1
    ON e1.EmployeeID = e.ManagerEmpID
```

Задача 2. Да се напише заявка, която извежда данните за продажбите и изчислената отстъпка в зависимост от стойността на TotalForSale и CurrentBalance: за обща сума на продажбата над 100 и текущ баланс на клиента над 1000 – 15%; за обща сума на продажбата над 100 и текущ баланс на клиента под 1000 – 10%; за обща сума на продажбата под 100 и текущ баланс на клиента над 1000 – 12%; за обща сума на продажбата под 100 и текущ баланс на клиента под 1000 – 0%.

Задача 3. Да се напише заявка, която извежда име на категория и средната аритметична стойност на доставните цени на продуктите от съответната категория.

Задача 4. Да се напише заявка, която извежда редовете от таблицата Sales, които нямат нито един съответен ред в таблицата SaleDetails.

Задача 5. Да се напише заявка, извеждаща имената на продуктите, които:

5.1. не са продавани и продажбата им не е преустановена;

5.2. са се продавали през изминалия месец, заедно със сума от продадените количества на съответните продукти, сортирани по сума на продаденото количество в низходящ ред.

Задача 6. Да се напише заявка, която извежда имената на клиентите, които не са пазарували.

Задача 7. Да се напише заявка, която извежда името на служителя, името на клиента, дата на продажба, общата стойност на продажба, имената на продуктите, продажните им цени и продадените от тях количества, отстъпки от цената и от общата стойност на продажба с идентификатор `SaleID = 87`.

Създаване на кръстосани заявки

Използване на CASE изрази за създаване на кръстосани заявки

CASE изразите могат да се използват за създаване на т.нар. *кръстосани заявки* (*crosstab report*), които се отличават по това, че конкретни стойности на колони се използват за заглавия на колоните. Подходящи са за таблично представяне на зависимости на стойностите в колоните и редовете.

Пример 1 Кръстосана заявка, която извежда продадените количества на продуктите за всички тримесечия през миналата година във вида:

ProductID	ProductName	I	II	III	IV	Total
1	Продукт А	150000	100000	180000	187000	617000

...

Решение:

```
SELECT p.ProductID, p.ProductName,
       SUM(CASE DATEPART(quarter, SaleDate)
            WHEN 1 THEN quantity
            ELSE 0
            END) AS [I quarter],
       SUM(CASE DATEPART(quarter, SaleDate)
            WHEN 2 THEN quantity
            ELSE 0
            END) AS [II quarter],
       SUM(CASE DATEPART(quarter, SaleDate)
            WHEN 3 THEN quantity
            ELSE 0
            END) AS [III quarter],
       SUM(CASE DATEPART(quarter, SaleDate)
            WHEN 4 THEN quantity
            ELSE 0
            END) AS [IV quarter],
       SUM(quantity) AS Total
FROM Products p
INNER JOIN SaleDetails sd ON p.ProductID = sd.ProductID
INNER JOIN Sales s ON sd.SaleID = s.SaleID
WHERE DATEDIFF(year, SaleDate, GETDATE()) = 1
GROUP BY p.ProductID, p.ProductName
```

Пример 2 Кръстосана заявка, която извежда общата сума от продажбите за последните три месеца по:

2.1. продукти във вида:

Идентификатор	Продукт	Миналият месец	Преди 1 месец	Преди 2 месеца
1	Продукт А	5000	3000	2800

...

2.2. магазини във вида:

Последните три месеца	Магазин А	Магазин В	Магазин С	Общо
януари	6000	5000	3000	14000
февруари	5500	6500	3210	15210
март	6800	3214	5432	15446

Магазин А, Магазин В, Магазин С са имена на магазини, т.е. стойности на колоната StoreName в таблицата Stores.

Решение:

2.1.

```
SELECT p.ProductID AS [Идентификатор],
       p.ProductName AS [Име на продукт],
       SUM(CASE DATEDIFF(month, SaleDate, GETDATE())
            WHEN 1 THEN sd.quantity*sd.price
                        *(1-sd.discount)*(1-s.discount)
            ELSE 0
        END) AS [Миналият месец],
       SUM(CASE DATEDIFF(month, SaleDate, GETDATE())
            WHEN 2 THEN sd.quantity*sd.price
                        *(1-sd.discount)*(1-s.discount)
            ELSE 0
        END) AS [Преди един месец],
       SUM(CASE DATEDIFF(month, SaleDate, GETDATE())
            WHEN 3 THEN sd.quantity*sd.price
                        *(1-sd.discount)*(1-s.discount)
            ELSE 0
        END) AS [Преди два месец]
FROM Products p
INNER JOIN SaleDetails sd ON p.ProductID = sd.ProductID
INNER JOIN Sales s ON sd.SaleID = s.SaleID
GROUP BY p.ProductID, p.ProductName
```

2.2.

```
SELECT DATENAME(month, s.SaleDate)
       AS [Последните три месеца],
       SUM(CASE StoreName
            WHEN 'Магазин А'
                THEN s.TotalForSale*(1-s.discount)
            ELSE 0
        END) AS [Магазин А],
       SUM(CASE StoreName
            WHEN 'Магазин В'
                THEN s.TotalForSale*(1-s.discount)
            ELSE 0
        END) AS [Магазин В],
       SUM(CASE StoreName
            WHEN 'Магазин С'
                THEN s.TotalForSale*(1-s.discount)
            ELSE 0
        END) AS [Магазин С],
       SUM(s.TotalForSale*(1-s.discount)) AS [Общо]
FROM Stores st
INNER JOIN Employees e ON st.StoreID = e.StoreID
```



```

INNER JOIN Sales s ON e.EmployeeID = s.EmployeeID
WHERE DATEDIFF(month, SaleDate, GetDATE())
      BETWEEN 1 AND 3
GROUP BY DATEPART(month, s.SaleDate),
          DATENAME(month, s.SaleDate)
ORDER BY DATEPART(month, s.SaleDate) ASC

```

За заявката от пример 2.2 са необходими CASE изрази от вида:

```

CASE StoreName
  WHEN 'Some store name'
  THEN s.TotalForSale*(1-s.discount)
  ELSE 0
END

```

За всеки израз се използва стойност на колоната StoreName в таблицата Stores. Следната заявка връща като резултат CASE изразите, необходими за съставянето на кръстосаната заявка от пример 2.2:

```

SELECT 'SUM(CASE StoreName
          WHEN '' + StoreName + ''
          THEN s.TotalForSale*(1-s.Discount)
          ELSE 0
        END) AS [' + StoreName + '], '
FROM Stores

```

Използване на PIVOT за създаване на кръстосани заявки

Друг начин за създаване на кръстосани заявки е чрез използване на ключовата дума PIVOT. PIVOT преобразува таблица чрез разполагане на уникалните стойности от една колона в множество колони в резултатната таблица и извършва обобщения върху останалите колони, включени в изрази. UNPIVOT изпълнява обраната операция.

Синтаксисът на PIVOT осигурява по-прост и по-лесен за четене синтаксис в сравнение със SELECT конструкциите, съдържащи CASE изрази.

Синтаксисът на SELECT израз, съдържащ PIVOT, е следния:

```

SELECT non_pivoted_column_list,
       pivoted_column1 AS column_name1,
       pivoted_column2 AS column_name2,
       ...
       pivoted_columnN AS column_nameN
FROM
  (SELECT_query) [AS] alias_source_query
PIVOT
(
  aggregation_function(column_name)
  FOR column_list_with_pivoted_values
  IN
    (pivoted_column1, pivoted_column2, ..., pivoted_columnN)
)
[AS] alias_pivot_table
[ORDER BY column_list_for_sort]

```

Пример 3 Кръстосана заявка за извличане на сумата от общите стойности на продажбите, осъществени от служителите по тримесечия:

```

SELECT EmployeeID,
       [1] AS [I quarter],
       [2] AS [II quarter],
       [3] AS [III quarter],
       [4] AS [IV quarter]
FROM (SELECT EmployeeID, TotalForSale,
       DATEPART(quarter, SaleDate) AS qrt
      FROM Sales
     ) AS piv
PIVOT (SUM(TotalForSale)
      FOR qrt
      IN ([1], [2], [3], [4])) AS res

```

Примерен резултат от заявката е показан на фигура 1.

EmployeeID	I quarter	II quarter	III quarter	IV quarter
1	568.75	1458.20	NULL	NULL
2	NULL	302.075	566.95	13.65
3	903.525	NULL	NULL	860.125
5	NULL	0.00	15.575	NULL

Фиг. 1 Примерен резултат от заявката в пример 1

Пример 4 Решението на пример 1 чрез използване на PIVOT има вида:

```

SELECT ProductID, ProductName,
       [1] AS [I quarter],
       [2] AS [II quarter],
       [3] AS [III quarter],
       [4] AS [IV quarter]
FROM (
  SELECT p.ProductID, p.ProductName, quantity,
         DATEPART(quarter, SaleDate) AS qrt
  FROM Products p
  INNER JOIN SaleDetails sd
    ON p.ProductID = sd.ProductID
  INNER JOIN Sales s ON sd.SaleID = s.SaleID
  WHERE DATEDIFF(year, SaleDate, GETDATE()) = 1
) AS piv
PIVOT (SUM(quantity)
      FOR qrt
      IN ([1], [2], [3], [4])) AS res

```

Пример 5 Решението на пример 2.1 чрез използване на PIVOT има вида:

```

SELECT ProductID AS [Идентификатор],
       ProductName AS [Име на продукт],
       [1] AS [Миналият месец],
       [2] AS [Преди един месец],
       [3] AS [Преди два месеца]
FROM (SELECT p.ProductID, p.ProductName,
       DATEDIFF(month, SaleDate, GETDATE()) AS diff,
       sd.quantity*sd.price*(1-sd.discount)*(1-s.discount)

```

```

        AS t
FROM Products p
INNER JOIN SaleDetails sd
    ON p.ProductID = sd.ProductID
INNER JOIN Sales s ON sd.SaleID = s.SaleID
    ) AS piv
PIVOT (SUM(t)
    FOR diff
    IN ([1], [2], [3])) AS res

```

Пример 6 Решението на пример 2.2 чрез използване на PIVOT има вида:

```

SELECT mn AS [Последните три месеца],
       [Магазин А] AS [Магазин А],
       [Магазин В] AS [Магазин В],
       [Магазин С] AS [Магазин С]
FROM (SELECT StoreName,
             DATEPART(month, s.SaleDate) AS mv,
             DATENAME(month, s.SaleDate) AS mn,
             s.TotalForSale*(1-s.discount) AS t
      FROM Stores st
      INNER JOIN Employees e ON st.StoreID = e.StoreID
      INNER JOIN Sales s ON e.EmployeeID = s.EmployeeID
      WHERE DATEDIFF(month, SaleDate, GetDATE())
             BETWEEN 1 AND 3
      ) AS piv
PIVOT (SUM(t)
    FOR StoreName
    IN ([Магазин А], [Магазин В], [Магазин С])) AS res
ORDER BY mv ASC

```

Задачи

Задача 1. Да се напише кръстосана заявка, която извежда продадените количества на продуктите от всички служители във вида:

Служител	Продукт 1	Продукт 2	Продукт 3	...
Служител А	6000.350	5000.350	3000.420	...
Служител В	5500.750	6500.250	3210.870	...
Служител С	6800.830	3214.900	5432.950	...
...				

Задача 2. Да се напише кръстосана заявка, която извежда общата сума от продажбите за всеки магазин за всяка категория продукти във вида:

Магазин	Категория 1	Категория 2	Категория 3	...
Магазин А	8000.75	8000.85	2000.65	...
Магазин В	3381.00	7500.25	3210.40	...
Магазин С	7800.90	5214.15	4432.50	...
...				

Обобщаване на данни чрез операторите CUBE и ROLLUP

В Transact-SQL има два оператора, които дават възможност да се генерират повече обобщаващи групирания в една заявка. Тези оператори са CUBE и ROLLUP, всеки от които е част от GROUP BY на конструкцията SELECT.

Използване на оператора CUBE за обобщаване на данни

Операторът CUBE генерира набор от резултати, който представлява многомерен куб. Колоните, които трябва да бъдат анализирани, се наричат *измерения (размерности)*. Многомерният куб е набор от резултати, представящ в табличен вид всички възможни комбинации на измеренията.

Пример 1 Заявка, която извежда имена на продукти и стойности за продадените от съответните продукти количества и общото количество на всички продадени продукти.

```
SELECT p.ProductName, SUM(sd.quantity) AS Quantity
FROM Products p
INNER JOIN SaleDetails sd ON p.ProductID = sd.ProductID
GROUP BY p.ProductName WITH CUBE
ORDER BY SUM(sd.quantity)
```

На фигура 1 е показан примерен резултат от тази заявка.

ProductName	Quantity
домати	1.000
чипс	1.000
ябълки	2.000
картофи	2.000
макарони	2.000
еклери	3.000
портокали	5.000
шоколад	10.000
NULL	26.000

Фиг. 1 Примерен резултат от заявката в пример 1

Пример 2 За извеждане на 'Общо' в колоната ProductName за общото количество на всички продадени продукти вместо стойност NULL се използва следната модифицирана заявка (фиг. 2):

```
SELECT ISNULL(p.ProductName, 'Общо') AS ProductName,
       SUM(sd.quantity) AS Quantity
FROM Products p
INNER JOIN SaleDetails sd ON p.ProductID = sd.ProductID
GROUP BY p.ProductName WITH CUBE
ORDER BY SUM(sd.quantity)
```

ProductName	Quantity
домати	1.000
чипс	1.000
ябълки	2.000
картофи	2.000
макарони	2.000
еклери	3.000
портокали	5.000
шоколад	10.000
Общо	26.000

Фиг. 2 Примерен резултат от заявката в пример 2

Пример 3 Заявка, която извежда имена на служители, дати на продажби, осъществени от съответните служители и суми на общите стойности на продажбите; общата сума на всички продажби на съответните служители; общата сума на продажбите на всички служители; общата сума на всички служители по дати.

```
SELECT FirstName + ' ' + LastName AS Name,
       CAST(CONVERT(char(10), SaleDate, 112) AS datetime)
       AS SaleDate,
       SUM(TotalForSale*(1-s.Discount)) AS Total
FROM Employees e
INNER JOIN Sales s ON e.EmployeeID = s.EmployeeID
GROUP BY FirstName + ' ' + LastName,
         CAST(CONVERT(char(10), SaleDate, 112)
         AS datetime)
WITH CUBE
```

Използване на оператора ROLLUP за обобщаване на данни

Операторът ROLLUP се използва за генериране на отчети, които съдържат междинни и крайни обобщения. За разлика от оператора CUBE, операторът ROLLUP генерира набор от резултати, който показва обобщенията за йерархия от стойности в избраните колони. Когато се прилага ROLLUP, е от значение редът, в който са зададени колоните в GROUP BY. Пресмятат се междинни и крайни обобщения върху всички възможни комбинации от колони, изброени след GROUP BY отляво надясно, т.е. първата колона в списъка определя начална точка в йерархията от стойности.

Пример 4 Заявка, която извежда сумата и средната аритметична стойност на количеството, което всеки служител е продал от всеки продукт; количеството, от което всеки продукт е продаден от всички служители; сумарното и средното количество на всички продукти, продадени от всички служители.

```
SELECT ProductName, FirstName + ' ' + LastName AS Name,
       SUM(quantity) AS Quantity,
       AVG(quantity) AS Average
FROM Employees e
INNER JOIN Sales s ON e.EmployeeID = s.EmployeeID
INNER JOIN SaleDetails sd ON s.SaleID = sd.SaleID
INNER JOIN Products p ON sd.ProductID = p.ProductID
GROUP BY ProductName, FirstName + ' ' + LastName
WITH ROLLUP
```

Пример 5 Заявка, която извежда количеството на продадените продукти по имена на продуктите и по продажна цена: количество, продадено от всеки продукт на всяка цена; количество, продадено от всеки продукт на всички цени; сумарното количество, продадено от всички продукти на всички цени.

```
SELECT p.ProductName, sd.Price,
       SUM(sd.quantity) AS SumOfQuantity
FROM Products p
INNER JOIN SaleDetails sd ON sd.ProductID = p.ProductID
GROUP BY p.ProductName, sd.Price WITH ROLLUP
```

Функцията **GROUPING(column_name)** позволява да се разграничи стойност NULL, генерирана от оператор CUBE или ROLLUP от стойност NULL, върната от действителните данни. Функцията GROUPING връща 1, ако стойността на колоната е NULL, генерирана от оператора CUBE или ROLLUP и 0, ако стойността на колоната е от действителните данни. Тази функция може да се използва за обозначаване на обобщаващите стойности по следния начин:

```
CASE GROUPING(column_name)
  WHEN 1 THEN 'ALL'
  ELSE ISNULL(column_name, 'UNKNOWN')
END AS column_name
```

Пример 6 Заявка за извеждане на приходите от продадените продукти по имената на клиентите и по имената на продуктите и на обобщенията на всички комбинации от стойности в колоните CompanyName и ProductName.

```
SELECT CASE GROUPING(CompanyName)
  WHEN 1 THEN 'ALL'
  ELSE ISNULL(CompanyName, 'UNKNOWN')
END AS CompanyName,
CASE GROUPING(ProductName)
  WHEN 1 THEN 'ALL'
  ELSE ISNULL(ProductName, 'UNKNOWN')
END AS ProductName,
SUM(sd.quantity*sd.price*(1-sd.discount)*
    (1-s.discount)) AS Total
FROM Customers c
INNER JOIN Sales s ON c.CustomerID = s.CustomerID
INNER JOIN SaleDetails sd ON s.SaleID = sd.SaleID
INNER JOIN Products p ON sd.ProductID = p.ProductID
GROUP BY CompanyName, ProductName WITH CUBE
```

Пример 7 Заявка за извеждане на средните приходи от продадените продукти по имената на категориите и по имената на продуктите и на обобщенията по стойностите в колоната CategoryName.

```
SELECT CASE GROUPING(CategoryName)
  WHEN 1 THEN 'ALL'
  ELSE ISNULL(CategoryName, 'UNKNOWN')
END AS CategoryName,
CASE GROUPING(ProductName)
  WHEN 1 THEN 'ALL'
  ELSE ISNULL(ProductName, 'UNKNOWN')
```

```

        END AS ProductName,
        AVG(sd.quantity*sd.price*(1-sd.discount)*
              (1-s.discount)) AS Average
FROM Categories c
INNER JOIN Products p ON c.CategoryID = p.CategoryID
INNER JOIN SaleDetails sd ON sd.ProductID = p.ProductID
INNER JOIN Sales s ON s.SaleID = sd.SaleID
GROUP BY CategoryName, ProductName WITH ROLLUP

```

Задачи

Задача 1. Да се създаде заявка, която извежда броя на продажбите по имена на клиентите и по имена на служителите.

Задача 2. Да се създаде заявка, която извежда сума от продажбите по имена на магазините и по имена и идентификатори на служителите.

Решение:

```

SELECT st.StoreName,
       e.FirstName + ' ' + e.LastName + ' - ' +
       LTRIM(STR(e.EmployeeID))
       AS [Name and ID],
       SUM(s.TotalForSale*(1-s.discount)) AS SumOfSales
FROM Stores st
INNER JOIN Employees e ON st.StoreID = e.StoreID
INNER JOIN Sales s ON e.EmployeeID = s.EmployeeID
GROUP BY st.StoreName,
         e.FirstName + ' ' + e.LastName + ' - ' +
         LTRIM(STR(e.EmployeeID))
WITH ROLLUP

```

Задача 3. Да се създаде заявка, която извежда сума от продажбите по имена на продуктите и по дати.

Задача 4. Да се създаде заявка, която извежда броя на продажбите по име на категория и по име на продукт за изминалата година. На обобщаващите редове да се извежда 'ALL' за съответните колони.

Използване на операторите за обединение, сечение и разлика

Множествените оператори в SQL са:

- UNION – обединение;
- INTERSECT – сечение;
- EXCEPT – разлика.

Тези оператори имат следния синтаксис:

```
<SELECT_query>
{{ UNION [ ALL ] | INTERSECT | EXCEPT }
<SELECT_query> } [ ... n ]
```

Операторите UNION, INTERSECT и EXCEPT връщат съответно обединението, сечението и разликата на два или повече резултатни набора в един, ако имат един и същ брой колони и съвместими типове данни.

За сортиране на резултата се използва ORDER BY след последната конструкция SELECT.

При прилагане на оператора UNION по подразбиране дублиращите се редове се елиминират, но е възможно да бъдат включени чрез UNION ALL.

Пример 1 Заявка, чрез която се извеждат имената и градовете на всички клиенти и доставчици съответно с надписи "Customer" и "Supplier".

```
SELECT CompanyName, City, 'Customer' AS Label
FROM Customers
UNION
SELECT CompanyName, City, 'Supplier' AS Label
FROM Suppliers
```

На фигура 1 е показан примерен резултат от тази заявка.

CompanyName	City	Label
Алтметал ЕООД	Велико Търново	Customer
Деница Стар ЕООД	Велико Търново	Customer
МИЛСТОП 90 ЕООД	Велико Търново	Customer
Мики-92 ЕООД	Велико Търново	Customer
АЛЕМИТ ООД	Велико Търново	Supplier
АЛПИ ООД	Лясковец	Supplier
БЕТИ - 2002	Велико Търново	Supplier

Фиг. 1 Примерен резултат от заявката в пример 1

Пример 2 Заявка, която извлича имената, продажните цени и продадените количества на продуктите с продадено количество, по-малко или равно на 5, както и тези с продадено количество, по-голямо или равно на 10 с елиминирание на дублиращите се редове от резултатния набор.

```
SELECT p.ProductName, sd.Price, sd.Quantity
FROM Products p
INNER JOIN SaleDetails sd ON p.ProductID = sd.ProductID
WHERE sd.Quantity <= 5
UNION
SELECT p.ProductName, sd.Price, sd.Quantity
FROM Products p
INNER JOIN SaleDetails sd ON p.ProductID = sd.ProductID
```



```
WHERE sd.Quantity >= 10
```

Тази заявка е еквивалентна на следната:

```
SELECT DISTINCT p.ProductName, sd.Price, sd.Quantity
FROM Products p
INNER JOIN SaleDetails sd ON p.ProductID = sd.ProductID
WHERE sd.Quantity <= 5 OR sd.Quantity >= 10
```

Към дадена колона от обединението може да се прави обръщение, като се използват имената на колоните от първата заявка.

Пример 3 Заявка, чрез която се извеждат градовете на всички магазини, всички клиенти и всички доставчици съответно с надписи "Store", "Customer" и "Supplier". Надписите се съдържат в колона със заглавие Name, по която е сортирано резултатното множество. Дублиращите се редове се елиминират от резултатния набор.

```
SELECT City, 'Store' AS Name
FROM Stores
UNION
SELECT City, 'Customer'
FROM Customers
UNION
SELECT City, 'Supplier'
FROM Suppliers
ORDER BY Name
```

Пример 4 Заявка, извеждаща градовете на всички магазини, всички клиенти и всички доставчици съответно с надписи "Store", "Customer" и "Supplier" – колона със заглавие Name, по която е сортирано резултатното множество; без елиминиране на дублиращите се редове от резултатния набор.

```
SELECT City, 'Store' AS Name
FROM Stores
UNION ALL
SELECT City, 'Customer'
FROM Customers
UNION ALL
SELECT City, 'Supplier'
FROM Suppliers
ORDER BY Name
```

Пример 5 Заявка, извеждаща градовете, в които има магазини, клиенти и доставчици.

```
SELECT City
FROM Stores
INTERSECT
SELECT City
FROM Customers
INTERSECT
SELECT City
FROM Suppliers
ORDER BY City
```

Пример 6 Заявка, извеждаща имената на клиентите, които са пазарували едновременно на две различни дати, например 07.06.2011 и 07.07.2011.

```
SELECT c.CompanyName
FROM Sales s
INNER JOIN Customers c ON s.CustomerID = c.CustomerID
WHERE DATEDIFF(day, s.SaleDate, '2011-06-07') = 0
INTERSECT
SELECT c.CompanyName
FROM Sales s
INNER JOIN Customers c ON s.CustomerID = c.CustomerID
WHERE DATEDIFF(day, s.SaleDate, '2011-07-07') = 0
ORDER BY c.CompanyName
```

Пример 7 Заявка, извеждаща градовете, в които има клиенти, но няма доставчици.

```
SELECT City
FROM Customers
EXCEPT
SELECT City
FROM Suppliers
ORDER BY City
```

Пример 8 Заявка, която извежда редовете от таблицата Sales, които нямат нито един съответен ред в таблицата SaleDetails.

```
SELECT *
FROM Sales
EXCEPT
SELECT s.*
FROM SaleDetails sd
INNER JOIN Sales s ON sd.SaleID = s.SaleID
```

Задачи

Задача 1. Да се създаде обединение на две заявки, което да показва идентификатора и името на всеки клиент и всеки доставчик от даден град. Резултатите да са подредени по азбучен ред. Клиентите да имат надпис "Customer"; доставчиците – "Supplier".

Резултатът от заявката да има вида:

ID	CompanyName	Label
1	Customer 1	Customer
2	Customer 2	Customer
1	Supplier1	Supplier
3	Supplier2	Supplier

Задача 2. Да се създаде обединение на три заявки, което да показва името на категорията и максималната, минималната и средната цена на всички продукти от съответната категория. Редовете с максималната цена за дадена категория да имат надпис "Max", с минимална – "Min", със средна – "Avg". Резултатите да са подредени по азбучен ред на имената на категориите и надписите.

Резултатът от заявката да има вида:

CategoryName	Price	Label
CategoryNameA	21.6666	Avg
CategoryNameA	40.0000	Max
CategoryNameA	10.0000	Min
CategoryNameB	40.0000	Avg
CategoryNameB	40.0000	Max
CategoryNameB	40.0000	Min
CategoryNameC	5.0000	Avg
CategoryNameC	5.0000	Max
CategoryNameC	5.0000	Min

Задача 3. Да се напише заявка, извеждаща идентификаторите на тези продажби, с които са се продали едновременно два различни продукта, например хляб и масло.

Задача 4. Да се напише заявка, която извежда имената на клиентите, които не са пазарували през миналия месец.

Съставяне на подзаявки

SQL предоставя възможност за влягане на заявки една в друга.

Пример 1 Заявка, извеждаща данните за продуктите с минимална доставна цена.

```
SELECT * FROM Products
WHERE price = ( SELECT MIN(price) FROM Products )
```

Пример 2 Заявка, извеждаща данните за продуктите с доставна цена, по-голяма от средната доставна цена на всички продукти.

```
SELECT * FROM Products
WHERE price > ( SELECT AVG(price) FROM Products )
```

В тези примери трябва подзаявката да връща една стойност – един ред и една колона, в противен случай се получава съобщение за грешка. За да се осигури връщане на точно една стойност, се използват обобщаващи функции или условие за стойността на колоната (колони) на първичния ключ или на колона (колони) с ограничение за уникалност. Когато подзаявката не върне резултат, конструкцията няма да върне грешка, но и основната заявка няма да върне резултат.

Transact-SQL позволява извеждане на резултата от подзаявката в списъка с полета на главната заявка.

Пример 3 Заявка, извеждаща име на продукт и брой на продажбите на съответния продукт.

```
SELECT ProductName, ( SELECT COUNT(*)
                        FROM SaleDetails sd
                        WHERE sd.ProductID = p.ProductID )
      AS CountOfProductsSales
FROM Products p
```

Тази заявка е еквивалентна на следната:

```
SELECT ProductName,
      COUNT(sd.ProductID) AS CountOfProductsSales
FROM Products p
LEFT JOIN SaleDetails sd ON sd.ProductID = p.ProductID
GROUP BY ProductName
```

За подзаявки, генериращи произволен брой редове, се използва специалният оператор **IN**.

Пример 4 Заявка, извеждаща идентификатор, име и фамилия на служителите от даден магазин.

```
SELECT EmployeeID, FirstName, LastName
FROM Employees
WHERE StoreID IN ( SELECT StoreID FROM Stores
                  WHERE StoreName = 'Име на магазин' )
```

Тази заявка е еквивалентна на следната:

```
SELECT EmployeeID, FirstName, LastName
FROM Employees e
INNER JOIN Stores s ON e.StoreID = s.StoreID
WHERE StoreName = 'Име на магазин'
```

Едно съединение може да се напише като подзаявка, но обратното не винаги е вярно. Съединението в повечето случаи се изпълнява по-бързо, отколкото еквивалентната подзаявка, но целта на заявката е по-ясно определена, когато даден въпрос бъде решен чрез подзаявка.

Пример 5 Заявка, извеждаща данните за магазините, намиращи се в градове, в които има доставчици.

```
SELECT * FROM Stores
WHERE City IN ( SELECT DISTINCT City FROM Suppliers )
```

Пример 6 Заявка, извеждаща идентификаторите и имената на продуктите, които не са продавани през изминалия месец и продажбата им не е преустановена.

```
SELECT ProductID, ProductName
FROM Products
WHERE ProductID NOT IN
    ( SELECT DISTINCT ProductID
      FROM SaleDetails sd
      INNER JOIN Sales s ON sd.SaleID = s.SaleID
      WHERE SaleDate BETWEEN
          DATEADD(month, -1, GetDate()) AND GetDate() )
AND Discontinued = 0
```

Пример 7 Обединение на две заявки, което показва името и цената на всеки продукт с максимална доставна цена и на всеки продукт с минимална доставна цена. Резултатите са подредени по азбучен ред. Продуктите с максимална цена имат надпис "Highest"; с минимална – "Lowest".

```
SELECT ProductName, Price, 'Highest' AS Label
FROM Products
WHERE price = (SELECT MAX(price) FROM Products)
UNION
SELECT ProductName, Price, 'Lowest' AS Label
FROM Products
WHERE price = (SELECT MIN(price) FROM Products)
ORDER BY ProductName
```

Пример 8 За извеждане на първите три най-високи цени и съответните имена на продукти, ще е необходимо да се използва DISTINCT с TOP. Ако едновременно с това бъде избрано и името на съответния продукт, няма да се получи желаният резултат, защото DISTINCT се прилага върху целия резултатен ред и всяка комбинация ProductName и Price е уникална за таблицата Products. Затова се използва подзаявка, за да се определи кои редове имат една от трите най-високи цени:

```
SELECT ProductID, ProductName, Price
FROM Products
WHERE price IN ( SELECT DISTINCT TOP 3 price
                  FROM Products
                  ORDER BY price DESC )
ORDER BY price DESC
```

В този случай редовете се връщат подредени в низходящ ред по цена, тъй като редът се контролира от външната заявка, към която е добавен ORDER BY.

Използване на оператори за подзаявки – EXISTS, ANY(или SOME), ALL

EXISTS е оператор, който приема подзаявка като аргумент и връща стойност TRUE, ако тази подзаявка връща някакви изходни данни и стойност FALSE – в противен случай. Този оператор не може да върне стойност UNKNOWN. За подзаявките, които използват EXISTS, няма значение коя колона избира EXISTS или дали избира всички колони, защото този оператор просто регистрира дали има някакви изходни данни от подзаявката или не и изобщо не използва генерираните стойности.

Пример 9 Заявка, извеждаща идентификаторите и имената на продуктите, които не са продавани и продажбата им не е преустановена.

```
SELECT p.ProductID, p.ProductName
FROM Products p
WHERE NOT EXISTS (SELECT * FROM SaleDetails sd
                  WHERE sd.ProductID = p.ProductID)
AND p.Discontinued = 0
```

Тази заявка е еквивалентна на следната:

```
SELECT p.ProductID, p.ProductName
FROM Products p
LEFT JOIN SaleDetails sd ON sd.ProductID = p.ProductID
WHERE sd.ProductID IS NULL AND p.Discontinued = 0
```

Пример 10 Заявка, извеждаща имената на продуктите, които са продавани.

```
SELECT p.ProductName
FROM Products p
WHERE EXISTS (SELECT * FROM SaleDetails sd
              WHERE sd.ProductID = p.ProductID)
```

Тази заявка е еквивалентна на следната:

```
SELECT DISTINCT p.ProductName
FROM Products p
INNER JOIN SaleDetails sd ON sd.ProductID = p.ProductID
```

ANY (или **SOME**) връща стойност TRUE, ако някоя стойност, избрана от подзаявката, удовлетворява условието на външната заявка.

Пример 11 Заявка, извеждаща данните за клиентите, намиращи се в градове, в които има магазини.

```
SELECT * FROM Customers
WHERE City = ANY (SELECT DISTINCT City FROM Stores)
Тази заявка е еквивалентна на следната:
SELECT * FROM Customers
WHERE City IN (SELECT DISTINCT City FROM Stores)
```

Пример 12 Заявка, извеждаща продуктите с доставна цена, по-голяма от доставната цена на някой (произволен) продукт от дадена по името си категория.

```
SELECT p.ProductID, p.ProductName, p.Price
FROM Products p
WHERE p.price >
      ANY (SELECT DISTINCT price FROM Products p
          INNER JOIN Categories c
```

```

        ON p.CategoryID = c.CategoryID
    WHERE c.CategoryName = 'name')

```

Тази заявка е еквивалентна на следната:

```

SELECT p.ProductID, p.ProductName, p.Price
FROM Products p
WHERE p.price >
    (SELECT MIN(price) FROM Products p
     INNER JOIN Categories c
       ON p.CategoryID = c.CategoryID
     WHERE c.CategoryName = 'name')

```

Когато подзаявката не генерира изходни данни, ANY връща FALSE.

ALL връща стойност TRUE, ако всяка стойност, избрана от подзаявката, удовлетворява условието на външната заявка.

Пример 13 Заявка, извеждаща продуктите с доставна цена по-голяма от доставната цена на всеки продукт от дадена по името си категория.

```

SELECT p.ProductID, p.ProductName, p.Price
FROM Products p
WHERE p.price >
    ALL (SELECT DISTINCT price FROM Products p
         INNER JOIN Categories c
           ON p.CategoryID = c.CategoryID
         WHERE c.CategoryName = 'name')

```

Тази заявка не е еквивалентна на следната:

```

SELECT p.ProductID, p.ProductName, p.Price
FROM Products p
WHERE p.price >
    (SELECT MAX(price) FROM Products p
     INNER JOIN Categories c
       ON p.CategoryID = c.CategoryID
     WHERE c.CategoryName = 'name')

```

Когато подзаявката не генерира изходни данни, ALL връща TRUE. Следователно за пример 13, когато категория със зададеното име не съществува, първата заявка ще изведе данните за всички продукти от таблицата Products, докато втората (с обобщаващата функция MAX) няма да генерира никакви изходни данни. Освен това ако има продукт от зададената категория, на който не е въведена доставна цена (т.е. price има стойност NULL), първата заявка няма да върне резултат, за разлика от втората, която може и да върне някакви изходни данни.

Пример 14 Заявка, извеждаща идентификаторите и имената на продуктите, които не са продавани и продажбата им не е преустановена.

```

SELECT p.ProductID, p.ProductName
FROM Products p
WHERE p.ProductID <> ALL (SELECT DISTINCT ProductID
                        FROM SaleDetails)
AND Discontinued = 0

```

Взаимосвързани заявки

Взаимосвързаните (съпоставени) заявки използват подзаявки, от които се прави обръщение към таблицата от FROM на външната (основната, главната) заявка. По този начин подзаявката се изпълнява многократно, по един път за всеки ред от таблицата в основната заявка, при което резултатите от всяко изпълнение на подзаявката трябва да бъдат съпоставени със съответния ред от външната заявка. Взаимосвързаните заявки се използват за сравняване на определени редове от една таблица с условие от съответстващата таблица.

Пример 15 Заявка, извеждаща всички редове от SaleDetails, за които продажната цена на съответния продукт е по-голяма или равна на средната аритметична стойност на продажните цени на този продукт с поне 50%.

```
SELECT * FROM SaleDetails sd
WHERE price >= (SELECT 1.5*AVG(price)
                FROM SaleDetails sd1
                WHERE sd.ProductID = sd1.ProductID)
```

Пример 16 Заявка, извеждаща данните за клиентите, пазарували на дадена дата (21.08.2003г.).

```
SELECT * FROM Customers c
WHERE '21.08.2003' IN
      (SELECT DISTINCT CONVERT(char(10), SaleDate, 104)
       FROM Sales s
       WHERE c.CustomerID = s.CustomerID)
```

Тази заявка е еквивалентна на следната:

```
SELECT DISTINCT c.*
FROM Customers c
INNER JOIN Sales s ON c.CustomerID = s.CustomerID
WHERE DATEDIFF(day, s.SaleDate, '20030821') = 0
```

Пример 17 Заявка, която връща продажбите на продуктите, чиито приходи са под 75% от средните приходи на всички продажби на съответния продукт.

```
SELECT sd.SaleID, p.ProductID, p.ProductName,
       sd.Price*sd.Quantity*(1-sd.Discount)
       AS ExtendedPrice
FROM Products p
INNER JOIN SaleDetails sd ON p.ProductID = sd.ProductID
WHERE sd.Price*sd.Quantity*(1-sd.Discount) <
      ( SELECT 0.75*AVG(sd1.Price*sd1.Quantity*
                        (1-sd1.Discount))
        FROM SaleDetails sd1
        WHERE p.ProductID = sd1.ProductID )
```

Взаимосвързаните заявки водят до преизчисляване на подзаявката за всеки ред в оператора SELECT на главната заявка, което може да не е най-ефективния начин за постигане на желанния резултат. Всеки път, когато е възможно, оптимизаторът на заявки на SQL Server обработва заявката, като я преизгражда като съединение между източника, определен във FROM и подзаявката. В много случаи е възможно тази реконструкция да се извърши ръчно, но целта на заявката може да не бъде така ясно определена, както когато проблемът бъде решен чрез подзаявка.

Производни таблици

Производна таблица е условното наименование за резултата от използването на друга конструкция SELECT във FROM на дадена конструкция SELECT.

Пример 18 Заявка, извеждаща броя на продуктите с различни доставни цени и непреустановена продажба.

```
SELECT COUNT(*) AS CountOfProducts
FROM (SELECT DISTINCT Price
      FROM Products
      WHERE Discontinued = 0)
      AS DistinctPrice
```

Пример 19 Заявка, извеждаща датите, на които не е продаван даден по идентификатора си (например 1) продукт. Резултатът е сортиран по дата в низходящ ред.

```
SET DATEFORMAT dmy

SELECT DISTINCT
      CAST(CONVERT(char(10), s.SaleDate, 104) AS datetime)
      AS Dates
FROM Sales s
LEFT JOIN ( SELECT SaleID
            FROM SaleDetails
            WHERE ProductID = 1 ) AS sd
      ON s.SaleID = sd.SaleID
WHERE sd.SaleID IS NULL
ORDER BY Dates DESC
```

Задачи

Задача 1. Да се напише заявка, която да връща имената на клиентите, които не са пазарували през изминалия месец.

Решение:

■ първи начин:

```
SELECT CompanyName
FROM Customers
WHERE CustomerID NOT IN
      ( SELECT DISTINCT CustomerID
        FROM Sales
        WHERE SaleDate BETWEEN DATEADD(month, -1, GetDate())
          AND GetDate() )
```

■ втори начин:

```
SELECT CompanyName
FROM Customers c
WHERE NOT EXISTS
      ( SELECT *
        FROM Sales s
        WHERE s.CustomerID = c.CustomerID
          AND
            SaleDate BETWEEN
              DATEADD(month, -1, GetDate()) AND GetDate() )
```

- *трети начин:*

```
SELECT CompanyName
FROM Customers
WHERE CustomerID <>
  ALL ( SELECT DISTINCT CustomerID
        FROM Sales
        WHERE SaleDate BETWEEN
          DATEADD(month, -1, GetDate()) AND GetDate() )
```

- *четвърти начин:*

```
SELECT CompanyName
FROM Customers
EXCEPT
SELECT CompanyName
FROM Sales s
INNER JOIN Customers c ON s.CustomerID = c.CustomerID
WHERE SaleDate BETWEEN
  DATEADD(month, -1, GetDate()) AND GetDate()
```

Задача 2. Да се напише заявка, която да връща имената на всички продукти, чиято продажба не е преустановена и броя на техните продажби за текущия ден.

Задача 3. Да се напише заявка, която да връща имената на всички служители, които не са напуснали и броя на осъществените от тях продажби за текущия ден.

Задача 4. Да се напише заявка, която да връща датите, на които не е продаван даден по идентификатора си (например 1) продукт. Резултатът да бъде сортиран по дата в низходящ ред.

Решение:

```
SET DATEFORMAT dmy
```

- *първи начин:*

```
SELECT DISTINCT
  CAST(CONVERT(char(10), s.SaleDate, 104) AS datetime)
  AS Dates
FROM Sales s
WHERE 1 NOT IN
  ( SELECT DISTINCT ProductID
    FROM SaleDetails sd
    INNER JOIN Sales s1 ON sd.SaleID = s1.SaleID
    WHERE DATEDIFF(day, s.SaleDate, s1.Saledate) = 0 )
ORDER BY Dates DESC
```

- *втори начин:*

```
SELECT DISTINCT
  CAST(CONVERT(char(10), s.SaleDate, 104) AS datetime)
  AS Dates
FROM Sales s
WHERE NOT EXISTS
  ( SELECT *
    FROM SaleDetails sd
    INNER JOIN Sales s1 ON sd.SaleID = s1.SaleID
```

```

WHERE DATEDIFF(day, s.SaleDate, s1.SaleDate) = 0
AND sd.ProductID = 1 )
ORDER BY Dates DESC

```

Задача 5. Да се напише заявка, която да връща датите, на които са се продавали всички продукти, чиято продажба не е преустановена.

Решение:

```

SELECT DISTINCT
    CONVERT(datetime, CONVERT(char(10), s.SaleDate, 112))
    AS Dates
FROM Sales s
WHERE NOT EXISTS
    (
        SELECT * FROM Products p
        WHERE NOT EXISTS
            (
                SELECT * FROM Sales s1
                INNER JOIN SaleDetails sd
                    ON s1.SaleID = sd.SaleID
                WHERE p.ProductID = sd.ProductID
                AND DATEDIFF(day, s.SaleDate, s1.SaleDate) = 0
            )
        AND Discontinued = 0
    )

```

Най-вътрешната заявка дава всички продажби на съответната дата; по-външната се съпоставя с нея, за да открие продукт, който не е продаден тогава и продажбата му не е преустановена; най-външната заявка връща всички различни дати, които не са от този списък, т.е. няма непродадени продукти, следователно всички са продавани.

Задача 6. Какви ще бъдат изходните данни от следните заявки:

6.1.

```

SELECT *
FROM (SELECT FirstName + ' ' + LastName AS Name,
    CompanyName
    FROM Employees e
    LEFT JOIN Sales s ON e.EmployeeID = s.EmployeeID
    FULL JOIN Customers c
        ON s.CustomerID = c.CustomerID
    WHERE c.City <> 'Велико Търново'
        OR c.City IS NULL) AS T
WHERE T.Name LIKE 'A%'

```

6.2.

```

SELECT DISTINCT c.CompanyName
FROM Customers c
INNER JOIN (SELECT CustomerID
    FROM Sales s
    INNER JOIN SaleDetails sd
        ON s.SaleID = sd.SaleID
    INNER JOIN Products p
        ON p.ProductID = sd.ProductID

```

```

        WHERE ProductName = 'PrName1') AS P1
    ON c.CustomerID = p1.CustomerID
INNER JOIN (SELECT CustomerID
            FROM Sales s
            INNER JOIN SaleDetails sd
                ON s.SaleID = sd.SaleID
            INNER JOIN Products p
                ON p.ProductID = sd.ProductID
            WHERE ProductName = 'PrName2') AS P2
    ON c.CustomerID = p2.CustomerID

```

6.3.

```

SELECT DISTINCT
    CAST(CONVERT(char(10), s.SaleDate, 112) AS datetime)
    AS Dates
FROM Sales s
LEFT JOIN ( SELECT DISTINCT
            CAST(CONVERT(char(10), SaleDate, 112)
                AS datetime) AS SaleDate1
            FROM Sales
            WHERE CustomerID = 1 ) AS c
    ON DATEDIFF(day, s.SaleDate, c.SaleDate1) = 0
WHERE c.SaleDate1 IS NULL

```

6.4.

```

SET DATEFORMAT dmy

SELECT DISTINCT
    CAST(CONVERT(char(10), s.SaleDate, 104) AS datetime)
    AS Dates
FROM Sales s
LEFT JOIN ( SELECT DISTINCT
            CAST(CONVERT(char(10), SaleDate, 104)
                AS datetime) AS SaleDate1
            FROM SaleDetails sd
            INNER JOIN Sales s1 ON sd.SaleID = s1.SaleID
            WHERE sd.ProductID = 1 ) AS sd
    ON DATEDIFF(day, s.SaleDate, sd.SaleDate1) = 0
WHERE sd.SaleDate1 IS NULL

```

6.5.

```

SELECT p.ProductName
FROM Products p
LEFT OUTER JOIN
    ( SELECT DISTINCT ProductID
      FROM SaleDetails sd
      INNER JOIN Sales s ON sd.SaleID = s.SaleID
      WHERE s.SaleDate BETWEEN
          DATEADD(week, -1, GetDate()) AND GetDate() )
    AS s1
    ON p.ProductID = s1.ProductID

```

```
WHERE s1.ProductID IS NULL
```

6.6.

```
SELECT c.CompanyName
FROM Customers c
LEFT OUTER JOIN
    ( SELECT DISTINCT CustomerID
      FROM Sales
      WHERE SaleDate BETWEEN DATEADD(month, -1, GetDate())
                        AND GetDate() )
  AS s
ON c.CustomerID = s.CustomerID
WHERE s.CustomerID IS NULL
```

6.7.

```
SELECT p.ProductName,
      COUNT(s1.ProductID) AS CountOfSales
FROM Products p
LEFT OUTER JOIN
    ( SELECT ProductID
      FROM Sales s
      INNER JOIN SaleDetails sd
        ON s.SaleID = sd.SaleID
      WHERE s.SaleDate BETWEEN DATEADD(day, -1, GetDate())
                        AND GetDate()
    ) AS s1
ON p.ProductID = s1.ProductID
WHERE p.Discontinued = 0
GROUP BY p.ProductName
```

6.8.

```
SELECT e.FirstName, e.LastName,
      COUNT(s.EmployeeID) AS CountOfSales
FROM Employees e
LEFT OUTER JOIN
    ( SELECT EmployeeID
      FROM Sales
      WHERE SaleDate BETWEEN DATEADD(day, -1, GetDate())
                        AND GetDate()
    ) AS s
ON e.EmployeeID = s.EmployeeID
WHERE e.TerminationDate IS NULL
GROUP BY e.FirstName, e.LastName
```

Използване на SQL за манипулиране на данни

SQL разполага с три основни конструкции за манипулиране на данните (*Data Manipulation Language – DML*) – INSERT, UPDATE и DELETE. Конструкциите, които са част от DML, позволяват въвеждане, променяне и изтриване на редове в таблиците, съхранявани в базата от данни.

Използване на INSERT за добавяне на данни

Чрез конструкцията INSERT се добавя един или повече редове в една таблица.

Общ вид на INSERT за добавяне на един ред:

```
INSERT [INTO] {table_name | view_name} [(column_list)]
VALUES (value_list)
```

Пример 1

```
INSERT INTO SaleDetails
      (SaleID, ProductID, Quantity, Price)
VALUES (9, 1, 18, 5)
```

В пример 1 е пропусната колоната Discount от таблицата SaleDetails, в резултат на което се задава стойността по подразбиране за тази колона – 0. Ако колоната няма дефинирана стойност по подразбиране, се задава стойност NULL. Следователно, за да може дадена колона да бъде пропусната в списъка с колони на конструкцията INSERT, трябва да има стойност по подразбиране или да допуска стойност NULL.

Когато се вмъкват редове в таблица с колона IDENTITY чрез конструкция INSERT, колоната IDENTITY трябва да бъде пропусната в списъка от колони.

Пример 2 При добавяне на ред в таблица за продажбите Sales трябва да се пропусне колоната SaleID (освен ако е включена опцията IDENTITY_INSERT чрез SET IDENTITY_INSERT Sales ON).

```
INSERT INTO Sales (CustomerID, EmployeeID)
VALUES (1, 1)
```

За добавяне на съответен ред в таблицата SaleDetails, може да се използва функцията @@IDENTITY.

Пример 3

```
INSERT INTO SaleDetails
      (SaleID, ProductID, Quantity, Price)
VALUES (@@IDENTITY, 1, 18, 5)
```

След изпълнението на конструкцията от пример 3 системната функция @@IDENTITY има стойност NULL, тъй като в таблицата SaleDetails няма колона със свойството IDENTITY.

Общ вид на INSERT за добавяне на множество редове:

```
INSERT [INTO] {table_name | view_name} [(column_list)]
SELECT column_list
FROM other_table_name
...
```

В таблицата се добавя резултатът, върнат от заявката SELECT, ако колоните на таблицата имат една и съща структура с избраните колони.

Пример 4

```
CREATE TABLE Totals
( SaleID int NOT NULL PRIMARY KEY,
  TotalForSale money NOT NULL DEFAULT 0 )

INSERT INTO Totals
SELECT SaleID, SUM(Price*Quantity*(1-Discount))
FROM SaleDetails
GROUP BY SaleID
```

Може да се използва функцията CONVERT, ако типът на данните трябва да бъде преобразуван.

Операцията е атомарна, т.е. пропадането на един ред, например след нарушаване на ограничение, предизвиква отхвърляне на всички редове, избрани от конструкцията SELECT. Съществува изключение от правилото, когато бъде намерен дублиран ключ в уникален индекс, създаден с опцията IGNORE_DUP_KEY.

Използване на конструкцията SELECT...INTO за добавяне на данни

Конструкцията SELECT...INTO дава възможност за създаване на нова таблица и попълване на таблица с набора от резултати от конструкцията SELECT. Общ вид на SELECT...INTO:

```
SELECT column_list
INTO new_table_name
FROM other_table_name
...
```

Тази конструкция е удобна за копиране на таблица или изтриване на колони, които повече не са необходими (като се пропусне ненужната колона от списъка на избора). Ползена е и когато избрани данни от дадена таблица трябва да се модифицират временно и когато е необходимо да се създаде таблица, която е подобна на вече съществуваща таблица и трябва да бъде попълнена със съответните данни.

Пример 5 Друг начин за създаване на таблица Totals от предишния пример е:

```
SELECT SaleID,
       CAST(SUM(Price*Quantity*(1-Discount)) AS money)
       AS TotalForSale
INTO Totals
FROM SaleDetails
GROUP BY SaleID
```

Създадената по този начин таблица няма дефиниран първичен ключ и стойност по подразбиране.

Пример 6 Създаване на таблица с идентификаторите на клиентите и изчисления им текущ баланс.

```
SELECT s.CustomerID,
       CAST(SUM(sd.Price*sd.Quantity*(1-sd.Discount)*
              (1-s.Discount)) AS money)
       AS CurrentBalance
```

```

INTO Totals2
FROM Sales s
INNER JOIN SaleDetails sd ON s.SaleID = sd.SaleID
GROUP BY CustomerID

```

Конструкцията `SELECT...INTO` оперира в специален режим без вписване в дневника на транзакциите, поради което се изпълнява по-бързо. За да се използва `SELECT...INTO` за попълване на постоянна таблица, трябва да бъде включена опцията на базата от данни *select into/bulkcopy* (чрез `EXEC sp_dboption database_name, 'select into/bulkcopy', TRUE`). Опцията не е необходимо да е включена за работа с временни таблици (с префикси # и ##), тъй като те не трябва да бъдат възстановявани. След изпълняване на `SELECT...INTO` с постоянна таблица трябва да се направи архивно копие на цялата база от данни, тъй като дневникът на транзакциите няма да съдържа записи за тези операции.

След всяка една от тези команди може да се използва системната функция `@@ROWCOUNT`, за да се установи броя на засегнатите редове.

Използване на UPDATE за променяне на данни

Конструкцията `UPDATE` може да променя стойностите на данните в отделни редове, в групи редове или всички редове в една таблица. Общ вид на `UPDATE` за променяне на данни:

```

UPDATE {table_name | view_name}
    SET column_name1 = {expression1|NULL|DEFAULT| (SELECT) }
        [, column_name2={expression2|NULL|DEFAULT| (SELECT) }...]
[WHERE {search_condition}]

```

Тази конструкция може да обнови всички редове, които отговарят на дадено условие. Ако се пропусне операторът `WHERE`, всички редове от таблицата се променят. В `SET` могат да се задават множество колони, разделени със запетаи и да се използват текущите стойности на колоните, за да се изчислят новите стойности. Изразът може да е константа, друга колона, аритметичен израз, системна функция, локална променлива, `CASE` израз или стойност, върната от подзаявка. Многоредов `UPDATE` е атомарна операция – ако един ред наруши ограничение, всички изменения, направени от тази конструкция, се превъртат назад.

Пример 7 Установяване на стойност на колоните `quantity` и `price` в таблицата `SaleDetails` за дадена продажба на даден продукт.

```

UPDATE SaleDetails
    SET quantity = 12,
        price = 2
WHERE SaleID = 1 AND ProductID = 5

```

Пример 8 Увеличаване с 50% на продажната цена на продуктите за продажба с даден идентификатор.

```

UPDATE SaleDetails
    SET price = 1.5*price
WHERE SaleID = 1

```


Пример 9 Установяване на стойността на продажната цена на даден продукт за дадена продажба на 50%^{HO} увеличение на максималната доставна цена на всички продукти.

```
UPDATE SaleDetails
    SET price = 1.5*(SELECT MAX(price) FROM Products)
WHERE SaleID = 87 AND ProductID = 1
```

Пример 10 Установяване на датата на продажба с даден идентификатор на вчерашна и идентификатор на клиента на 3.

```
UPDATE Sales
    SET SaleDate = DATEADD(day, -1, GetDate()),
        CustomerID = 3
WHERE SaleID = 87
```

Допълнително в Transact-SQL е налице конструкцията:

```
UPDATE {table_name | view_name}
    SET column_name1 = {expression1|NULL|DEFAULT| (SELECT) }
    [, column_name2={expression2|NULL|DEFAULT| (SELECT) }...]
[FROM {table_source}]
```

Чрез *table_source* се определя таблицата (таблиците), която се използва за критерий за операцията UPDATE.

Пример 11 Увеличаване на продажната цена на продуктите от дадена по идентификатора си продажба с удвоената доставна цена на съответния продукт.

```
UPDATE SaleDetails
    SET price = sd.price + 2*p.price
FROM SaleDetails sd
INNER JOIN Products p ON sd.ProductID = p.ProductID
WHERE SaleID = 87
```

Тази конструкция е еквивалентна на следната:

```
UPDATE SaleDetails
    SET price = sd.price +
        ( SELECT 2*p.price
          FROM Products p
          WHERE sd.ProductID = p.ProductID )
FROM SaleDetails sd
WHERE SaleID = 87
```

Пример 12 Установяване на отстъпката от цената на продукт с идентификатор 3 за продажба с идентификатор 10 в зависимост от продаденото количество: за продадено количество над 20 отстъпка – 10%; между 10 и 20 – 5%; под 10 – 0%.

```
UPDATE SaleDetails
    SET Discount = CASE
                        WHEN Quantity > 20 THEN 0.10
                        WHEN Quantity >= 10 THEN 0.05
                        ELSE 0
                    END
WHERE SaleID = 10 AND ProductID = 3
```

Може да се използва системната функция @@ROWCOUNT за определяне на броя на обработваните редове.

Използване на DELETE за изтриване на данни

Конструкцията DELETE изтрива един или повече редове от една таблица. Общ вид на DELETE за изтриване на данни:

```
DELETE FROM {table_name | view_name}
[WHERE {search_condition}]
```

Тази команда изтрива редовете, удовлетворяващи условието. Ако се пропусне операторът WHERE, всички редове от таблицата се изтриват, но не и самата таблица (за премахване на таблицата се използва DROP TABLE table_name).

Пример 13 Изтриване на всички редове от таблицата SaleDetails за продукти, чиято продажба е преустановена.

```
DELETE FROM SaleDetails
WHERE ProductID IN ( SELECT ProductID FROM Products
                     WHERE Discontinued = 1 )
```

Пример 14 Изтриване на всички покупки на клиента с идентификатор 5, осъществени на 01.01.2003г.

```
DELETE FROM Sales
WHERE CustomerID = 5
      AND DATEDIFF(day, SaleDate, '2003/01/01') = 0
```

В Transact-SQL има допълнителна конструкция:

```
DELETE FROM {table_name | view_name}
[FROM {table_source}]
```

Чрез table_source се определя таблицата (таблиците), която се използва за критерий за операцията DELETE.

Пример 15 Следната команда е еквивалентна на тази от пример 13 за изтриване на редове:

```
DELETE FROM SaleDetails
FROM Products p
INNER JOIN SaleDetails sd ON p.ProductID = sd.ProductID
WHERE Discontinued = 1
```

Може да се използва системната функция @@ROWCOUNT за определяне на броя на изтритите редове.

Задачи

Задача 1. Да се напише команда, която да променя колоната Discount от таблицата SaleDetails за продажбите на даден продукт (ProductID = 10) в зависимост от стойността на quantity: за продадено количество от съответния продукт над 200 отстъпка – 15%; между 100 и 200 – 10%; под 100 – 0%.

Задача 2. Да се напише команда, която да променя колоната Discount от таблицата Sales за дадена продажба (SaleID = 3) в зависимост от стойността на TotalForSale и CurrentBalance: за обща стойност на продажбата над 100 и

текущ баланс над 1000 отстъпка – 15%; за обща стойност на продажбата над 100 и текущ баланс под 1000 – 10%; за обща стойност на продажбата под 100 и текущ баланс над 1000 – 12%; за обща стойност на продажбата под 100 и текущ баланс под 1000 – 0%.

Задача 3. Да се напише команда, която да изтрива всички редове от таблицата Sales, които нямат нито един съответен ред в таблицата SaleDetails.

Задача 4. Да се напише команда, която да изтрива данните за всички служители, които са напуснали.

Задача 5. Да се напише команда (или команди) за създаване на таблица, съдържаща идентификаторите на клиентите и датата на първата им покупка, както и за попълване на тази таблица с данни.

Задача 6. Какъв ще бъде резултатът от изпълнението на всяка една от следните команди:

6.1.

```
CREATE TABLE TotalDaySale
( DateTotal datetime NOT NULL
    CONSTRAINT PK_TotalDaySale PRIMARY KEY,
  DayTotal money NOT NULL
    CONSTRAINT DF_TotalDaySale DEFAULT 0 )
```

```
INSERT INTO TotalDaySale
SELECT CONVERT(char(10), SaleDate, 112),
        SUM(totalForSale)
FROM Sales
GROUP BY CONVERT(char(10), SaleDate, 112)
```

6.2.

```
UPDATE SaleDetails
    SET price = 1.25*price
WHERE ProductID IN ( SELECT ProductID
                     FROM Products p
                     INNER JOIN Categories c
                         ON p.CategoryID = c.CategoryID
                     WHERE c.CategoryName = 'name' )
```

или

```
UPDATE SaleDetails
    SET price = 1.25*sd.price
FROM SaleDetails sd
INNER JOIN Products p ON sd.ProductID = p.ProductID
INNER JOIN Categories c ON p.CategoryID = c.CategoryID
WHERE c.CategoryName = 'name'
```

6.3.

```
UPDATE SaleDetails
    SET price = 1.20*(SELECT MAX(price)
                     FROM Products p
                     INNER JOIN Categories c
```

```
                ON p.CategoryID = c.CategoryID
                WHERE c.CategoryName = 'name')
WHERE SaleID = 12 AND ProductID = 5
```

6.4.

```
UPDATE SaleDetails
    SET price = 1.75*p.price
FROM SaleDetails sd
INNER JOIN Products p ON sd.ProductID = p.ProductID
ИЛИ
UPDATE SaleDetails
    SET price =( SELECT 1.75*price
                  FROM Products p
                  WHERE sd.ProductID = p.ProductID )
FROM SaleDetails sd
```

Глава III Администриране на бази от данни

Индекси

Освен таблиците други важни дискови структури с данни, дефинирани от потребителя, са индексите. Индексът (*index*) на колона или група от колони на дадена таблица е структура от данни, която позволява да се повиши ефективността на процедурата за търсене на конкретни стойности, съхранявани в тази колона или група от колони. Наличието на индекс обикновено води до съществено намаляване на времето, необходимо за обработката на заявките, в които стойностите на дадена колона A се сравняват с константи чрез изрази от вида $A = const$, $A \geq const$ или $A \leq const$.

Въпреки че технологията за създаване на индекси не се регламентира в никой от стандартите на SQL, в повечето комерсиални системи за управление на бази от данни са предвидени средства, позволяващи създаване на индекси за определени колони в конкретни таблици.

Типове индекси

SQL Server поддържа клъстерирани и неклъстерирани индекси. И двата типа използват стандартни *B*-дървета (т.е. балансирани дървета – разликата в дълбочината на произволна двойка листа не надвишава единица), които осигуряват бърз достъп до данните. Броят на нивата в индекса зависи от броя на редовете в таблицата и от размера на индексирания колона (или колони).

Клъстерирани индекси

Клъстерираният индекс определя физическото подреждане на данните в таблицата. Тъй като данните могат да бъдат физически подредени само по един начин, таблицата може да има само един клъстериран индекс. Той позволява данните да бъдат намерени директно на нивото на листовите възли, т.е. листовото ниво на клъстерирания индекс съдържа страници с данни. В SQL Server всички клъстерирани индекси са уникални.

Неклъстерирани индекси

При неклъстерираните индекси нивото на листовите възли съдържа маркер, указващ на SQL Server къде да намери редовете с данни, съответстващи на стойностите на индекса.

Наличието или отсъствието на неклъстериран индекс не влияе на начина, по който са организирани страниците с данни, затова в една таблица може да има повече от един неклъстериран индекс.

Неклъстерираният индекс е полезен само ако е *високо-селективен*, т.е. трябва да може да елиминира от разглеждането голям процент от редовете. Това означава, че трябва да бъде съставен от колона или колони, които предоставят голямо разнообразие на стойностите.

Колоните, съставляващи външните ключове, почти винаги са подходящи за създаване на неклъстерирани индекси, тъй като те често биват използвани за съединяване на таблици. По този начин се подобрява изпълнението на заявки, които извличат редове от поне две таблици.

Създаване на индекс

Създаването на индекс се осъществява с командата `CREATE INDEX`, която в има следния синтаксис:

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED]
    INDEX index_name
    ON {table_name | view_name}
    ( column_name1 [ASC|DESC] [, column_name2 [ASC|DESC]
      [, ..., column_nameN [ASC|DESC]] )
[WITH
    [PAD_INDEX]
    [[,] FILLFACTOR = fillfactor]
    [[,] IGNORE_DUP_KEY]
    [[,] DROP_EXISTING]
    [[,] STATISTICS_NORECOMPUTE]
    [[,] SORT_IN_TEMPDB]
]
[ON filegroup_name]
```

Колоните от тип *ntext*, *text*, *image* и *bit*, както и изчислимите колони не могат да бъдат индексирани. Индексите могат да бъдат създавани в таблици или изгледи. Възможно е определяне на възходящ (ASC) или низходящ (DESC) ред на сортиране за отделните индексни колони. По подразбиране е възходящ ред, т.е. ASC.

Ако е зададено ограничение PRIMARY KEY или UNIQUE, при изпълнението на конструкциите за променяне (UPDATE) или вмъкване (INSERT) на редове, засягащи множество редове, ако дори само един от тях наруши ограничението (т.е. дублира някоя стойност в съответната колона на таблицата), цялата конструкция се прекратява и редовете остават непроменени или не се вмъкват нито един ред. При уникален индекс може да се използва опцията IGNORE_DUP_KEY, така че грешките в уникалността при командата INSERT за множество редове не водят до отхвърляне на цялата конструкция. Всеки ред, който не е уникален, бива отстранен, а останалите редове се въвеждат в съответната таблица.

Опцията FILLFACTOR (коефициент на запълване) се задава в проценти и определя колко свободно място трябва да се запази в страниците на ниво листо на индекса. Ако не е зададен коефициент на запълване, се използва подразбиращата се стойност на ниво сървър, която се задава чрез параметъра *fill factor* на системната съхранена процедура `sp_configure` (`EXEC sp_configure 'fill factor', 0`). Тази стойност по подразбиране е 0, която е оптимизирана настройка на SQL Server, т.е. листовите страници на индексите се запълват, колкото е възможно. За да се зададе действителен процент на запълване, се използва цяло число между 1 и 100. При определяне на подходяща стойност за този параметър, трябва да се има предвид, че задаването на прекалено висок коефициент на запълване води до забавяне на изпълнението на конструкциите за модифициране на данните в таблицата; задаването на прекалено нисък коефициент оказва влияние на производителността при извличане на данни от таблицата. Затова в зависимост от конкретната ситуация е необходимо да се направи компромис между бързо изпълняващи се актуализации и висока производителност при извличане на данни.

Чрез опцията PAD_INDEX се установява запазване на свободно място в страниците на междинните нива на индекса. Опцията е приложима, само ако е зададен коефициент на запълване чрез FILLFACTOR, тъй като се използва тази процентна стойност.

Опцията `DROP_EXISTING` позволява вече създаден индекс да бъде построен отново, т.е. индексът бива изтрит и създаден отново.

Чрез опцията `STATISTICS_NORECOMPUTE` се указва да не се преизчислява автоматично остарялата статистика за индекса. Това ще се отрази на работата на оптимизатора на SQL Server, тъй като поддържането на статистическа информация за съдържанието на таблиците в базата от данни (като например брой редове, брой различни стойности и др.) позволява ефективно да се оптимизира достъпът до данните. За да се възстанови автоматичното актуализиране на статистиката, трябва да се изпълни `UPDATE STATISTICS` без опцията `NORECOMPUTE` или системната съхранена процедура `sp_autostats`.

Чрез опцията `SORT_IN_TEMPDB` се определя междинните резултати при сортиране, използвани при построяване на индекса, да се съхраняват в системната база от данни *tempdb*.

Променяне на индекс

Променянето на индекс се осъществява с командата `ALTER INDEX`, която има следния синтаксис:

```
ALTER INDEX index_name
    ON {table_name | view_name}
{ REBUILD
    [ WITH (
        { PAD_INDEX = { ON | OFF }
        | FILLFACTOR = fillfactor
        | SORT_IN_TEMPDB = { ON | OFF }
        | IGNORE_DUP_KEY = { ON | OFF }
        | STATISTICS_NORECOMPUTE = { ON | OFF } } ) ]
    | DISABLE
    | REORGANIZE
    | SET (
        { | IGNORE_DUP_KEY = { ON | OFF }
        | STATISTICS_NORECOMPUTE = { ON | OFF } } )
}
```

Тази команда позволява пресъздаване, реорганизиране, деактивиране на индекса или задаване на опции на индекса. Чрез ключовата дума:

- `REBUILD` се задава пресъздаване на индекса, като се използват същите колони, тип на индекса, наличие или не на изискване за уникалност и ред на сортиране на индексните колони;
- `DISABLE` се деактивира индекса. Дефиницията на индекса се запазва. За да се активира индексът, е необходимо да се изпълни `ALTER INDEX ... REBUILD` или `CREATE INDEX ... WITH DROP_EXISTING`.
- `REORGANIZE` се указва реорганизиране на нивото на листовите възли на индекса;
- `SET` се задават опции на индекса без неговото пресъздаване или реорганизиране.

Изтриване на индекс

Неизползвани индекси за често променяни таблици би трябвало да бъдат изтривани. В противен случай SQL Server ще заделя ненужни изчислителни ресурси за поддържане на неизползвани индекси.

Изтриването на индекс се осъществява с командата DROP INDEX, която има следния синтаксис:

```
DROP INDEX index_name
      ON {table_name | view_name}
```

Нека приемем, че първичните ключове на таблиците на примерната база от данни, описана в задача 2 на тема „Ограничаване на стойностите на данните”, се поддържат от клъстерирани индекси. Освен това в таблицата Customers са зададени ограничения за уникалност cust_TaxNumber_Unique и cust_Bulstat_Unique съответно върху колоните TaxNumber и Bulstat, в резултат на което автоматично са създадени уникални неклъстерирани индекси. Тогава могат да бъдат създадени някои допълнителни неклъстерирани индекси. Командите за част от тях са представени в пример 1.

Пример 1

- в таблицата Customers


```
CREATE UNIQUE INDEX IX_CompanyName
      ON Customers ( CompanyName )

CREATE INDEX IX_City
      ON Customers ( City )

CREATE INDEX IX_PostalCode
      ON Customers ( PostalCode )
```
- в таблицата Employees


```
CREATE INDEX IX_Name
      ON Employees ( FirstName, LastName )

CREATE UNIQUE INDEX IX_UniqueEGN
      ON Employees ( EGN )
WITH IGNORE_DUP_KEY


CREATE INDEX IX_StoreID
      ON Employees ( StoreID )

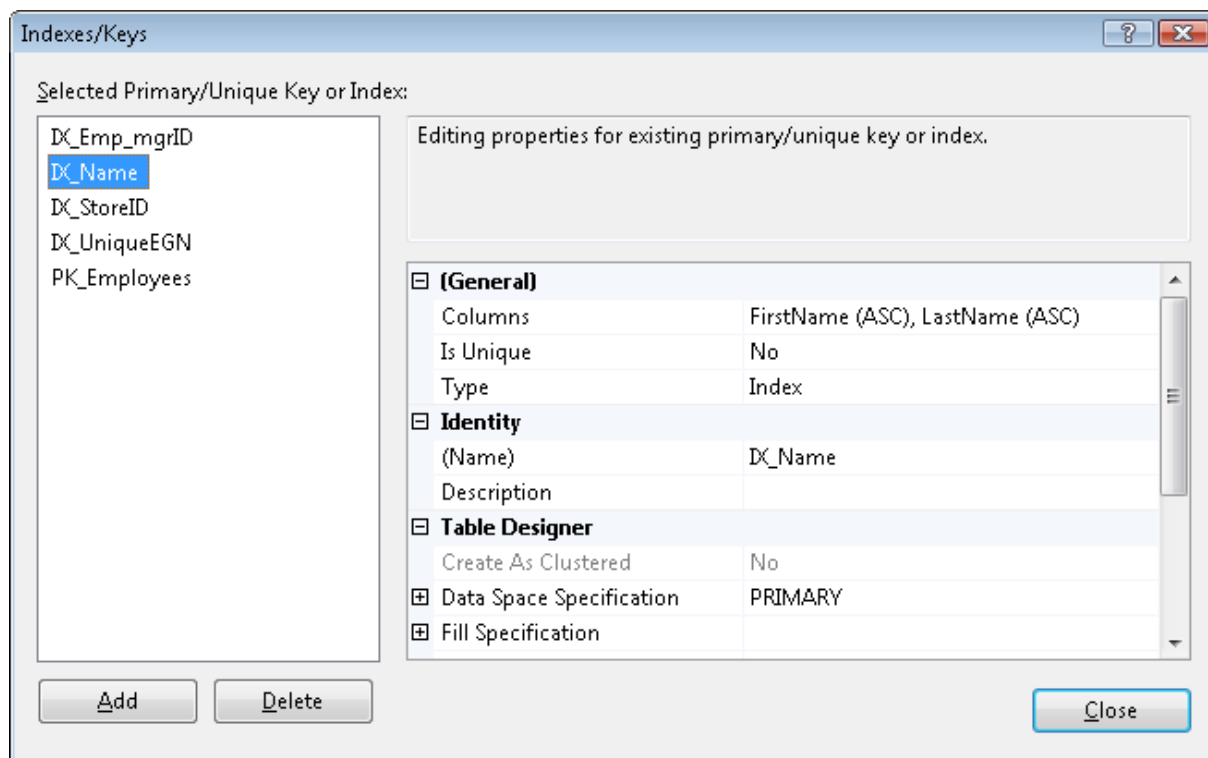
CREATE INDEX IX_Emp_mgrID
      ON Employees ( ManagerEmpID )
```
- в таблицата Products


```
CREATE INDEX IX_CategoryID
      ON Products ( CategoryID )

CREATE UNIQUE INDEX IX_ProductName
      ON Products ( ProductName )

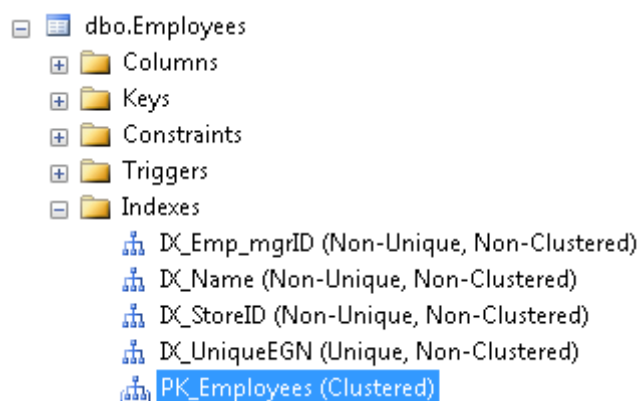
CREATE INDEX IX_SupplierID
      ON Products ( SupplierID )
```


Индекс може да се създаде и изтрие и от прозореца *Design* в Management Studio. Чрез натискане на бутона *Manage Indexes and Keys*  се отваря диалоговият прозорец *Indexes and Keys*, който дава възможност за създаване на нови (*Add*) и изтриване на ненужните индекси (*Delete*) в избраната таблица. На фигура 1 е показано създаване на индекс по този начин.



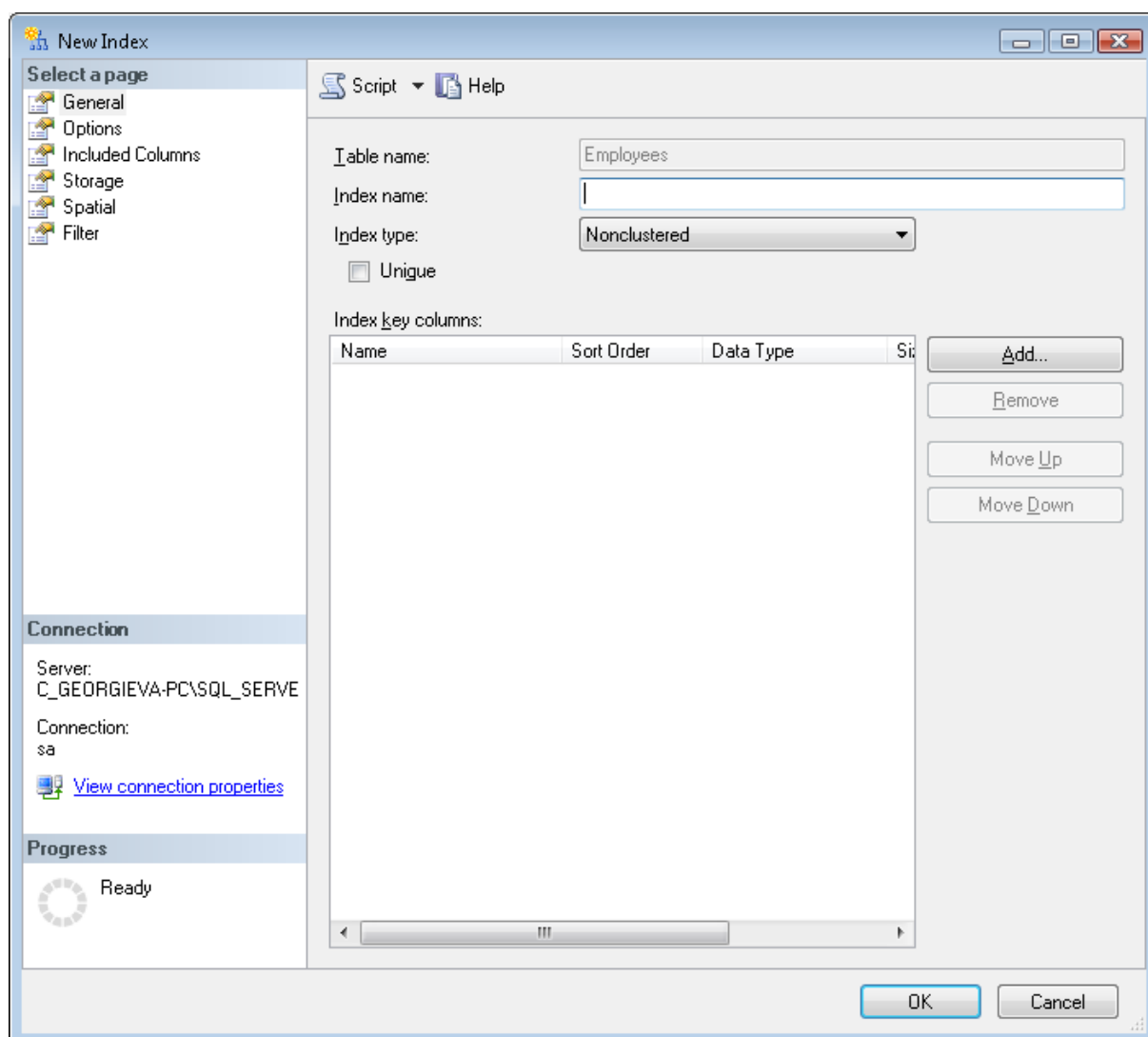
Фиг. 1 Създаване на нов индекс и преглеждане на съществуващите индекси

Management Studio разполага и с друга възможност за управление на индексите в таблиците – прозореца *Index Properties*, който се отваря след двойно щракване върху името на индекса от папката *Indexes* на съответната таблица. Например, индексите в таблицата за служителите са показани на фигура 2.



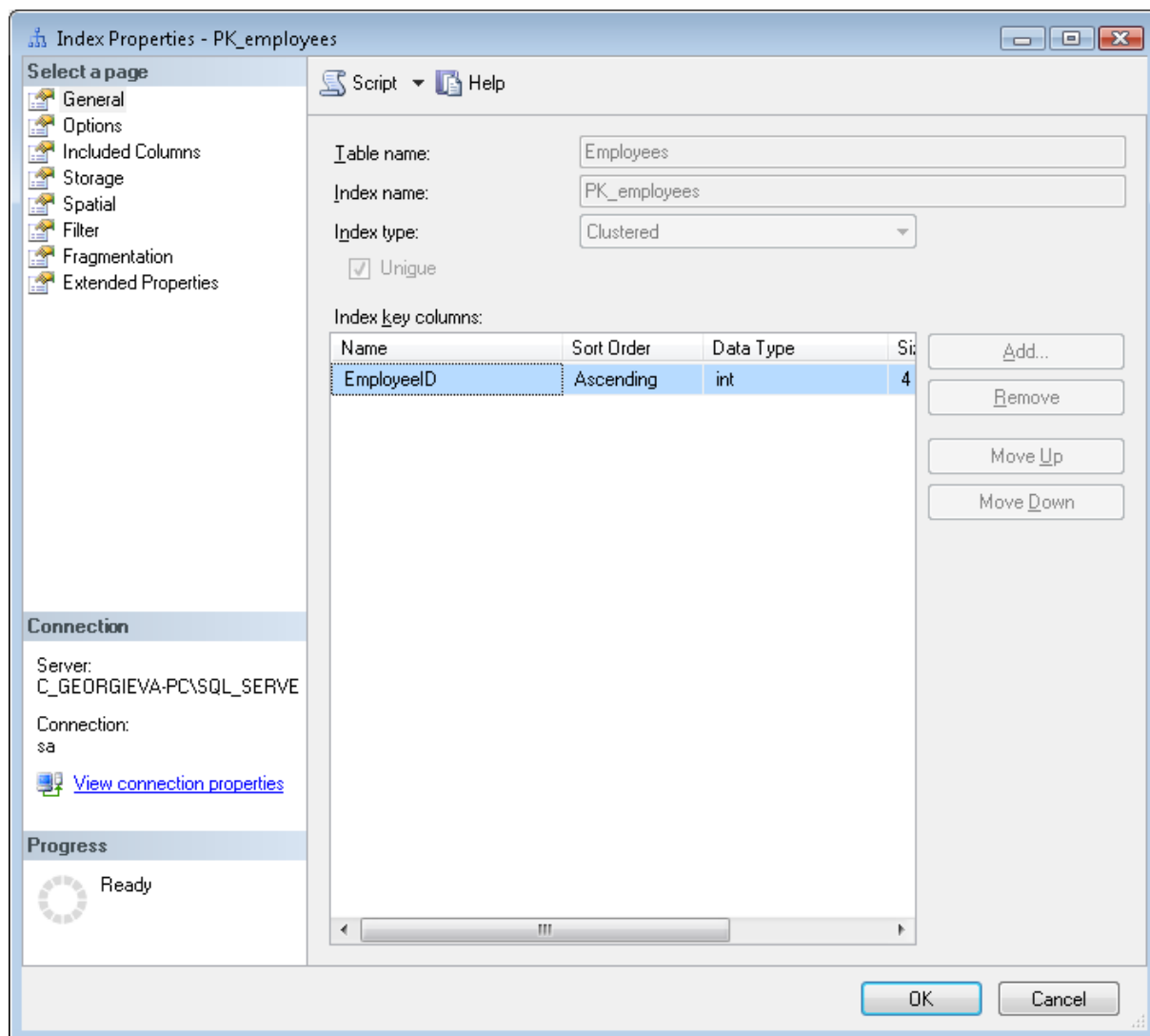
Фиг. 2 Управление на индексите в таблица

За създаване на нов индекс може да се използва прозореца *New Index* (фиг. 3).



Фиг. 3 Създаване на индекс

За преглеждане и промяна на опциите на съществуващ индекс е предназначен прозореца *Index Properties* (фиг. 4).

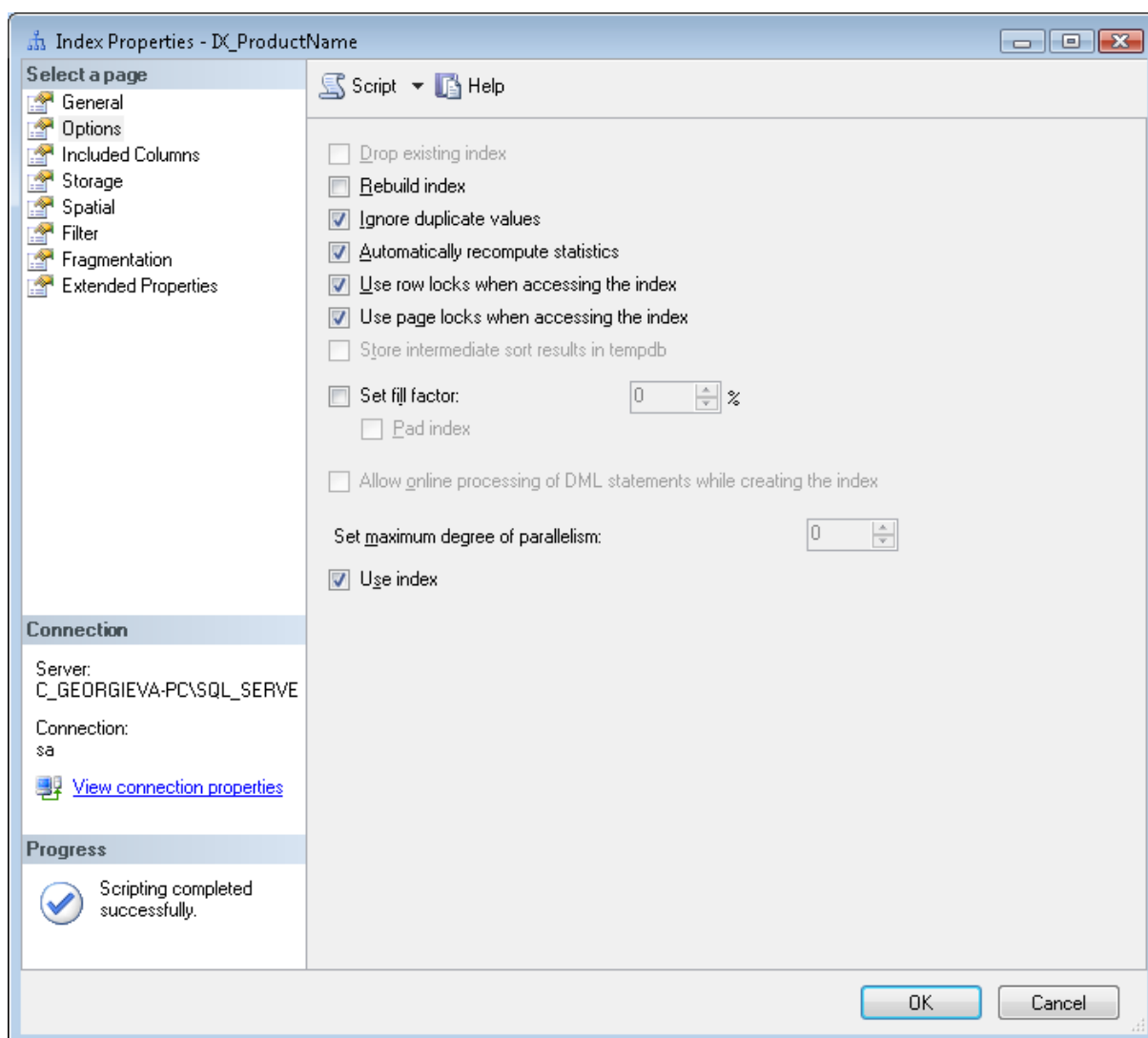


Фиг. 4 Проверяване и редактиране на свойствата на съществуващите индекси

Пример 2 За да се модифицира уникалният индекс IX_ProductName, зададен за колоната ProductName на таблицата Products, така че да игнорира дублиращите се стойности, може да се изпълни следната T-SQL конструкция:

```
ALTER INDEX IX_ProductName
ON Products
SET ( IGNORE_DUP_KEY = ON )
```

Този израз се генерира при включване на опцията *Ignore duplicate values* на диалоговата рамка, показана на фигура 5.



Фиг. 5 Промяна на индекса IX_ProductName

Чрез избиране на бутона *Script* (фиг. 5) може да се провери, редактира и изпълни съответната T-SQL конструкция (фиг. 6).

```
SQLQuery4.sql - C:\...atabase (sa (57))*
USE [MyDatabase]
GO
ALTER INDEX [IX_ProductName]
ON [dbo].[Products]
SET ( IGNORE_DUP_KEY = ON )
GO
```

Фиг. 6 Използване на генерирания SQL израз

По аналогичен начин може да се уточнят характеристиките на индекса, който бива създаден автоматично от SQL Server при задаване на ограничение първичен ключ в дефиницията на таблицата.

За получаване на информацията за индексите в дадена таблица са предназначени и системните съхранени процедури `sp_helpindex` и `sp_statistics`. Например:

```
EXEC sp_helpindex 'Employees'
EXEC sp_statistics 'Employees'
```

Задачи

Задача 1. Да се дефинират подходящите индекси в таблиците, създаването на които е описано в задачите 1÷3 на тема „Използване на SQL за дефиниране на данни”.

Задача 2. Да се обоснове полезността или безполезността на предложените в примерите индекси за таблиците Customers, Employees, Products на базата от данни, описана в задача 2 на тема „Ограничаване на стойностите на данните”. Да се напишат и изпълнят конструкции за:

2.1. изтриване на индексите, които няма да бъдат използвани (ако има такива);

2.2. създаване на подходящи индекси в останалите таблици на базата от данни – Stores, Suppliers, Categories, Sales, SaleDetails.

Анализиране и оптимизиране на достъпа до данните

Всеки път, когато бъде изпратена заявка за изпълнение от SQL Server, релационната машина на SQL Server трябва да вземе решение относно начина, по който да извърши връщането на данните към потребителя. Предназначението на т.нар. *оптимизатор на заявки* е да намира най-ефективния начин за връщане на данни за определено време. Той сравнява различните възможни методи за връщане на данните и избира един. Последователността от стъпките, които релационната машина следва, за да изпълни една заявка, се нарича *план за изпълнение*. Съществуват две характеристики, които трябва да се определят за всеки план за изпълнение:

- Редът, в който се осъществява достъп до таблиците, когато се изпълнява заявката – оказва влияние не върху реда, в който се извеждат редовете в резултатния набор, а върху скоростта на съединенията на таблиците. Ако данните от първата таблица могат да бъдат получени бързо и представляват малко на брой редове, тогава операторът за съединение ще има по-малко данни за обработване и ще може да приключи по-бързо.
- Начинът, по който се извличат данните от таблиците – оптимизаторът на заявки има на разположение следните възможности за избор: сканиране на таблицата за намиране на редовете, съответстващи на условието на заявката; използване на индекс или дори на няколко индекса. Ако таблицата има малко редове, тогава почти винаги по-бързо се изпълнява сканирането на таблицата. Ако броят на редовете, които заявката ще върне е голям, но съществува индекс върху колоните, включени в WHERE, HAVING или ORDER BY, тогава използването на индекса води до по-бързото изпълнение.
- За заявки, включващи съединение на таблици, е налице трета стъпка – определя се типът на съединението, което ще се използва за извличане на информацията от различните таблици.

Съществува голямо разнообразие от типове оптимизатори на заявки, използвани от различните релационни системи за управление на бази от данни. Оптимизаторът на заявки на Microsoft SQL Server се основава на икономичността на тяхното изпълнение. Той определя цена на всеки възможен метод за връщане на данните на базата на ресурсите, необходими за изпълнение на заявката. Процесорното време, I/O операции и т.н. биват оценени въз основа на броя на редовете, които трябва да бъдат прегледани в една отделна операция. След като оптимизаторът определи цените, той ги сумира, за да получи общата цена на съответния план за изпълнение, който проучва. Върху избора на план за изпълнение на заявката, който ще направи оптимизаторът, значително влияние оказва начинът, по който е проектиран моделът на данните в конкретната система.

За някои заявки може да има много възможни планове за изпълнение. Вместо да изследва и да оценява всеки план, SQL Server се опитва да отгатне кой план ще е най-близко до оптималния. Използва различни алгоритми, за да направи това предположение и да намали броя на плановете, които проверява. SQL Server не винаги избира план за изпълнение, който струва най-малко на ресурсите. Намирането на най-ефективния метод изисква ресурси за изследване на всички възможни начини и сравняването им. Вместо това SQL Server е проектиран да избере този план за изпълнение, който има основания да е най-близко до теоретичния минимум и ще върне резултатите на клиента възможно най-бързо. Целта е да се избере един от най-подходящите планове за изпълнение на заявката, като се отдели минимално време за нейното оптимизиране, в резултат на което се минимизира общото време за връщане на данните към потребителя.

Анализиране на заявки

Използване на SET конструкции

SQL Server разполага с множество конструкции, които могат да се изпълнят от клиентското приложение, за да се получи подробна информация, необходима за анализиране и оптимизиране на дадена заявка. Опциите, предназначени за анализиране и оптимизиране на заявки, се включват и изключват и установеното им състояние важи до приключване на текущата конекция (или до изричното указване на другото им състояние).

- `SET FORCEPLAN {ON|OFF}` принуждава оптимизатора на заявки на SQL Server да обработи съединението в заявката в същия ред, в който таблиците са описани след `FROM` на SQL конструкцията.
- `SET NOEXEC {ON|OFF}` предотвратява изпълнението на заявката, като се позволява само проверката на синтаксиса и създаването на план за изпълнение на заявката от оптимизатора.
- `SET SHOWPLAN_ALL {ON|OFF}` позволява конструкцията да не се изпълнява и вместо резултата от нея се извежда подробна информация за начина, по който тя се изпълнява. Резултатният набор съдържа следните колони:
 - `StmtText` е текста на SQL конструкцията за редовете, които не извеждат план за изпълнение и описание на операцията, която свървят трябва да осъществи, за да изпълни заявката.
 - `StmtId` е поредния номер на израза в текущата конекция.
 - `NodeId` е идентификатора на операцията, включена в изпълнението на заявката.
 - `Parent` е идентификатора на предходната операция.
 - `PhysicalOp` е физическия алгоритъм, който се прилага за операцията (сканиране, търсене по индекс и т.н.).
 - `LogicalOp` е оператора от релационната алгебра, който е представен чрез тази операция.
 - `Argument` осигурява допълнителна информация за съответната операция.
 - `DefinedValues` е списък от стойности, разделени със запетаи. Те могат да бъдат изчислими изрази или вътрешни стойности, необходими за изпълнението на заявката.
 - `EstimateRows` е приближение на броя на редовете, които ще бъдат извлечени със съответния оператор.
 - `EstimateIO` е приближение на I/O цена на съответния оператор.
 - `EstimateCPU` е приближение на CPU цена за съответния оператор.
 - `AvgRowSize` е приближение на средния размер в байтове на редовете, които трябва да бъдат обходени от съответния оператор.
 - `TotalSubtreeCost` е приближение на общата цена за съответната операция и всички нейни подчинени операции.
 - `OutputList` е списък от колони, разделени със запетаи. Включва колоните, които ще бъдат извлечени чрез тази операция.
 - `Warnings` е списък с предупредителни съобщения, разделени със запетая. Възможните съобщения са `"NO STATS():"` и `"NO JOIN PREDICATE"`. `"NO STATS():"` със списък от колони може да възникне, ако оптимизаторът за заявки се опитва да прави предположения, но статистиката за колоните не е налична. `"NO JOIN PREDICATE"`

означава, че е включено съединение на таблици без оператор за съединение (например `SELECT ProductName FROM Products p, SaleDetails sd`).

- `Type` е типа на операцията. За първия ред от резултатния набор стойността на тази колона е типът на SQL конструкцията (`SELECT`, `INSERT`, `UPDATE`, `DELETE`). За останалите редове, които представят стъпката от плана за изпълнение, типът е `PLAN_ROW`.
 - `Parallel` има две възможни стойности 0 или 1. Стойност 0 означава, че този оператор не е стартиран паралелно и 1 – наличие на паралелно изпълнение.
 - `EstimateExecutions` е приближение на броя на изпълненията на съответния оператор по време на текущата заявка.
- `SET SHOWPLAN_TEXT {ON|OFF}` връща част от информацията, която се получава чрез `SET SHOWPLAN_ALL`. Резултатът се състои само от една колона `StmtText`, но включва няколко резултатни набора. Стойността на колоната е текста на SQL конструкцията за редовете, които не извеждат плана за изпълнение и описание на съответната операция в противен случай.
 - `SET STATISTICS IO {ON|OFF}` определя дали да се извежда статистическа информация за различните I/O операции във вид на съобщение (за разлика от другите опции, които връщат резултатен набор). Резултатът съдържа следните елементи:
 - `Table` е името на таблицата, за която се отчита статистиката.
 - `Scan count` е броя на сканиранията, които са извършени по време на извличането на данните от таблицата.
 - `logical reads` е броя на прочетените страници с данни от кеш паметта.
 - `physical reads` е броя на прочетените страници от диска.
 - `read-ahead reads` е броя на прочетените страници от кеш паметта при използване на механизма за предварително четене по време на обработка на заявката, който се стреми да прехвърли голяма част от необходимите данни в кеш паметта.
 - `SET STATISTICS PROFILE {ON|OFF}` определя дали да се извежда специфична информация за изпълнението на конструкцията. Резултатният набор включва всички колони, които се извеждат чрез `SET SHOWPLAN_ALL` и следните допълнителни колони:
 - `Rows` е действителния брой на редовете, обработени от съответния оператор.
 - `Executes` е действителния брой на изпълненията на съответния оператор по време на текущата заявка.
 - `SET STATISTICS TIME {ON|OFF}` определя дали да се извеждат съобщения за времето (в милисекунди), необходимо за анализиране, компилиране и изпълняване на всяка конструкция.

Анализиране на плана за изпълнение

Получаването на плана за изпълнение на една заявка е най-лесната част от оптимизацията. Най-продължителната и трудна част е анализирането на този план, при което е необходимо да се определи какви стъпки да се предприемат, за да се осигури оптимално изпълнение на заявката. Оптимизаторът трябва да промени плана за изпълнение в зависимост от различни фактори и да открие най-добрата оптимизация за конкретната ситуация в съответния момент.

Когато се анализира един план за изпълнение, първото нещо, което трябва да се разгледа, е типът на възникналите операции. Съществуват разнообразни операции, които може да се наложи да се изпълнят и всяка от тях изисква различни ресурси. Всяка операция може да се изпълни многократно в една заявка. Не може да се твърди, че една операция е винаги по-ефективна от друга, тъй като това зависи от конкретната заявка.

Списък с операциите, които може да се появят в плана за изпълнение на заявка, е:

- **Bookmark Lookup** Когато чрез индекс се намира ред, който удовлетворява някаква част от заявката, ще е необходимо индексът да използва идентификатора на ред за търсене на действителните редове с данни.
- **Clustered Index Scan** Сканирането на клъстерирания индекс се получава, ако колоните в условието на заявката не са индексирани или оптимизаторът е установил, че индексът няма да може да елиминира достатъчно редове, за да бъде по-ефективен от сканирането.
- **Clustered Index Seek** Прилага се търсене по клъстерирания индекс за намиране на редовете, които удовлетворяват условието на заявката. Това е оптималната операция за повечето заявки.
- **Compute Scalar** Изчисляване на израз, необходим за резултата от заявката или за друга част от заявката (например за филтриране или съединяване).
- **Constant Scan** Заявката изисква константна стойност в някои (или всички) редове.
- **Hash Match** Създава се таблица за всеки ред, който вече е обработен. Следващите редове, които се обработват, се сравняват с редовете от тази таблица за наличие на съответствие. Заявки с `DISTINCT`, обобщаващи функции или `UNION` често имат нужда от такава таблица, за да се премахнат дублиранията.
- **Index Scan** Използва се неклъстерирания индекс за намиране на редовете с данните, но голям брой редове трябва да бъдат върнати или условието в `WHERE` не може да бъде прегледано чрез този индекс. Прочита се част от или целия индекс за удовлетворяване на условието на заявката.
- **Index Seek** Използва се неклъстерирания индекс за намиране на редовете с данните и може да бъде разгледана само част от индекса за удовлетворяване на условието на заявката. Това е една от най-ефективните операции за повечето заявки. Оптимизаторът на заявки я избира, ако създаденият индекс е високо-селективен.
- **Index Spool** Когато редовете биват сканирани, те се разполагат в таблица, която съществува в *tempdb*. Създаден е индекс и когато са необходими допълнителни сканирания на данните, тази таблица може да бъде използвана вместо препрочитане на входните редове. Това е вътрешна оптимизация за сложните заявки и не може да бъде променена.
- **Merge Join** Съединение със сливане възниква, когато двата набора съдържат сортирани данни и е възможно да се извърши сливане. Тази операция е много ефективна, но само когато данните са сортирани. За да се предизвика нейното прилагане може да се добавят индекси върху свързаните колони или колоните, по които се сортира. Може да бъде избрана и когато два или повече индекса се използват за извличане на данни от таблица. Резултатите от всяко търсене по индекс се сливат, за да се получи списък с редовете данни, които трябва да бъдат прочетени.

- **Nested Loops** При това съединение, едната таблица се избира за вътрешна и се сканира за всеки ред от външната. Това не е ефективна операция, освен ако броят на редовете е относително малък.
- **Remote Query** Някаква част от заявката бива изпратена към отдалечен източник. Ако това е бавно изпълняваща се част от заявката, за целите на оптимизацията е необходимо да се изпълни на отдалечен източник. Текстът на заявката се изпраща на отдалечен сървър, за да приложи своя собствена оптимизация.
- **Sort** Сортирането на всички върнати редове е неикономична операция и е еквивалентно на създаването на индекс върху резултатния набор. Предизвиква се от включването на ORDER BY или DISTINCT.
- **Table Scan** Изпълнява се, когато таблицата няма клъстериран индекс. Това е обикновено най-неефективната операция за повечето заявки. Включва прочитане на всеки ред в паметта и извършване на проверки за това дали да бъде върнат от заявката. За да се избегне сканирането на таблицата, трябва да се добавят подходящи индекси.

Пример 1

```
SET SHOWPLAN_TEXT ON
GO
```

Нека разгледаме плана за изпълнение на следната заявка:

```
SELECT c.CompanyName AS Customer,
       MAX(s.TotalForSale) AS MaxTotal
FROM Sales s
INNER JOIN Customers c ON s.CustomerID = c.CustomerID
INNER JOIN Employees e ON s.EmployeeID = e.EmployeeID
WHERE TerminationDate IS NULL
GROUP BY c.CompanyName
```

Получава се следният резултат:

```
StmtText
```

```
-----
SELECT c.CompanyName AS Customer,
       MAX(s.TotalForSale) AS MaxTotal
FROM Sales s
INNER JOIN Customers c ON s.CustomerID = c.CustomerID
INNER JOIN Employees e ON s.EmployeeID = e.EmployeeID
WHERE TerminationDate IS NULL
GROUP BY c.CompanyName
```

```
(1 row(s) affected)
```

```
StmtText
```

```
-----
|--Stream          Aggregate (GROUP          BY: ([c].[CompanyName])
DEFINE: ([Expr1006]=MAX([MyDatabase].[dbo].[Sales].[TotalForSal
e] as [s].[TotalForSale]))
|--Sort (ORDER BY: ([c].[CompanyName] ASC))
|--Nested          Loops (Inner          Join,          OUTER
REFERENCES: ([s].[CustomerID]))
```

```



|--Nested      Loops (Inner      Join,      OUTER
REFERENCES: ([s].[EmployeeID]))
|
|--Clustered   Index
Scan (OBJECT: ([MyDatabase].[dbo].[Sales].[PK_Sales] AS [s]))
|
|--Clustered   Index
Seek (OBJECT: ([MyDatabase].[dbo].[Employees].[PK_Employees] AS
[e]),
SEEK: ([e].[EmployeeID]=[MyDatabase].[dbo].[Sales].[EmployeeID]
as [s].[EmployeeID]),
WHERE: ([MyDatabase].[dbo].[Employees].[TerminationDate] as
[e].[TerminationDate] IS NULL) ORDERED FORWARD)
|--Clustered   Index
Seek (OBJECT: ([MyDatabase].[dbo].[Customers].[PK_Customers] AS
[c]),
SEEK: ([c].[CustomerID]=[MyDatabase].[dbo].[Sales].[CustomerID]
as [s].[CustomerID]) ORDERED FORWARD)

(7 row(s) affected)

```

От втория резултатен набор може да се заключи следното:

- Първият ред показва изпълняването на обобщаващата операция, тъй като заявката съдържа обобщаващата функция MAX.
- Вторият ред е операция за сортиране, която се извършва като вътрешна операция, подготвяща данните за осъществяване на групирането.
- Третият ред включва съединение чрез вложени цикли, т.е. Nested Loops, което се използва за изпълняването на вътрешното съединение между таблиците Sales и Customers.
- Четвъртият ред показва Nested Loops съединение за осъществяване на вътрешното съединение между таблиците Sales и Employees.
- Петият ред е сканиране на клъстериран индекс на таблицата Sales, която е избрана за вътрешна при реализиране на вътрешното съединение с таблицата Employees.
- Шестият ред показва търсене по клъстериран индекс в таблицата Employees, при което се извършва и филтриране, като се проверява стойността на колоната TerminationDate. Ако се добави индекс на колоната TerminationDate, вероятно ще се включи и операцията търсене по този индекс.
- Седмият ред е търсене по клъстериран индекс в таблицата Customers. Тъй като съществува клъстериран индекс в тази таблица, това е изключително ефективна операция.

SQL Server предлага възможност за графично представяне на предполагаемия план за изпълнение на заявката, като се избере от менюто на Management Studio командата *Query / Display Estimated Execution Plan* или бутона  от лентата с инструменти. За визуализиране на действителния план за изпълнение на заявката в графичен режим се използва командата от менюто *Query / Include Actual Execution Plan* или бутона . В резултат на това за всяка изпълнена заявка се появяват две страници – с изходните данни от заявката *Results* и графичен изглед на плана за изпълнение *Execution plan*.

Оптимизиране на заявки

Съществуват няколко различни начина за оптимизиране на заявките. Индексите са най-често използвания метод за подобряване на тяхната производителност. Използването на подсказки в заявките дава възможност да се принуди сървъра да изпълни заявката по определен начин, но трябва да се има предвид, че това може да доведе до изрично избиране на план, който да не е най-оптималния план за изпълнение. Необходими са многобройни проверки за ползата от включването на подсказки в заявките в конкретните условия на съответната система.

В повечето системи често изпълняваните заявки са относително малко на брой. Оптимизирането именно на тези заявки има значителни предимства по отношение на производителността на системата като цяло.

Индекси

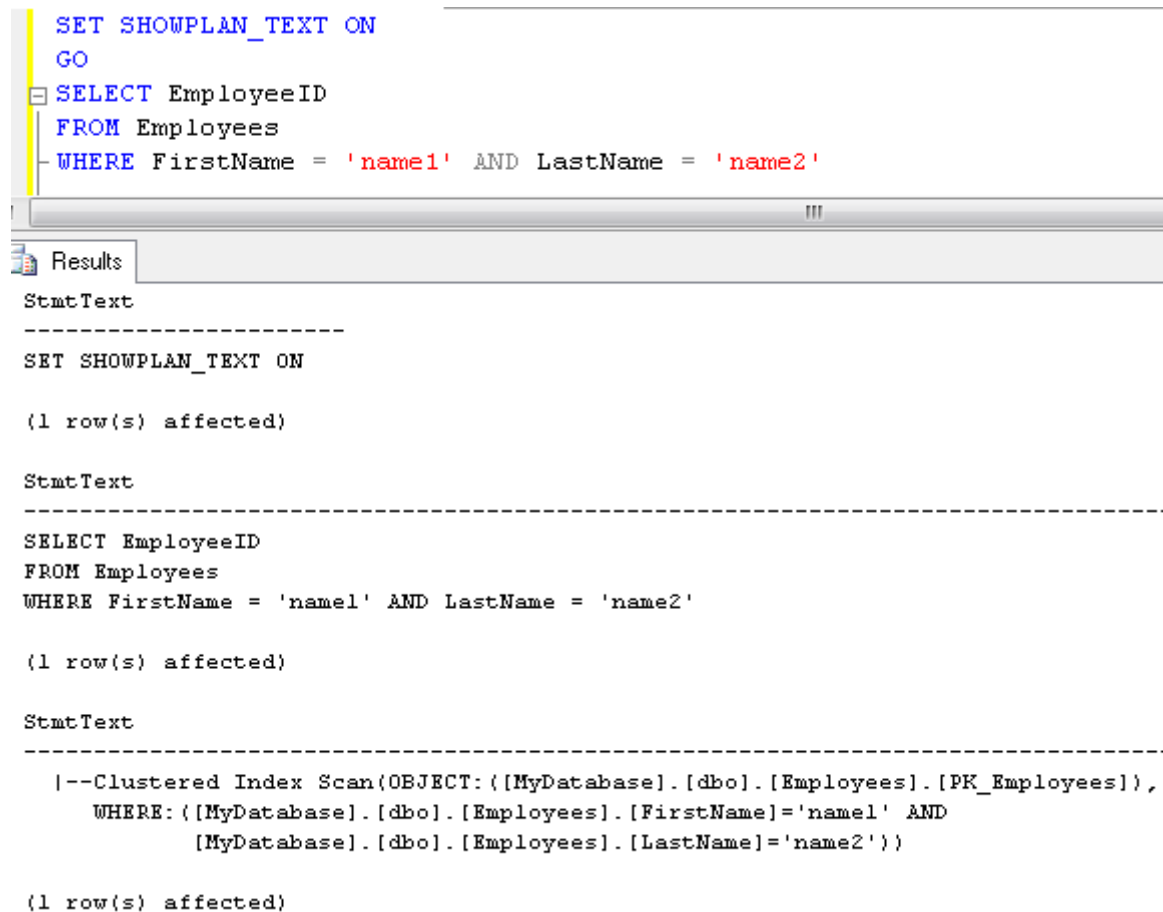
Ако не е възможно да се напише заявката по такъв начин, че да се използват съществуващите индекси, тогава добавянето на друг индекс е обикновено най-добрият начин за подобряване на ефективността на заявката. В повечето случаи, когато може да се използва индекс в заявка или съединение, количеството на използваните ресурси на сървъра (натоварване на процесора, входно-изходни операции) намалява.

За да бъде използван индекс, първата индексирана колона трябва да бъде включена в WHERE като част от условието.

Пример 2 Следната заявка:

```
SELECT EmployeeID
FROM Employees
WHERE FirstName = 'name1' AND LastName = 'name2'
```

може да бъде изпълнена по няколко различни начина от оптимизатора на заявки. Ако не съществува друг индекс освен клъстерирания, тогава оптимизаторът на заявки ще избере да извърши сканиране по клъстерирания индекс за намиране на всички редове, за които FirstName отговаря на 'name1' и LastName съответства на 'name2' (фиг. 1).



```

SET SHOWPLAN_TEXT ON
GO
SELECT EmployeeID
FROM Employees
WHERE FirstName = 'name1' AND LastName = 'name2'

```

Results

```

StmtText
-----
SET SHOWPLAN_TEXT ON

(1 row(s) affected)

StmtText
-----
SELECT EmployeeID
FROM Employees
WHERE FirstName = 'name1' AND LastName = 'name2'

(1 row(s) affected)

StmtText
-----
|--Clustered Index Scan(OBJECT: ([MyDatabase].[dbo].[Employees].[PK_Employees]),
  WHERE: ([MyDatabase].[dbo].[Employees].[FirstName]='name1' AND
    [MyDatabase].[dbo].[Employees].[LastName]='name2'))

(1 row(s) affected)

```

Фиг. 1 План за изпълнение на разглежданата заявка, ако не съществува друг индекс освен клъстерираня

Ако съществува индекс, зададен за колоните FirstName и LastName, тогава оптимизаторът ще изпълни само една операция – търсене по този индекс (фиг. 2).

```

SET SHOWPLAN_TEXT ON
GO
SELECT EmployeeID
FROM Employees
WHERE FirstName = 'name1' AND LastName = 'name2'

```

Results

StmtText

SET SHOWPLAN_TEXT ON

(1 row(s) affected)

StmtText

SELECT EmployeeID
FROM Employees
WHERE FirstName = 'name1' AND LastName = 'name2'

(1 row(s) affected)

StmtText

--Index Seek(OBJECT: ([MyDatabase].[dbo].[Employees].[IX_Name]),
SEEK: ([MyDatabase].[dbo].[Employees].[FirstName]=[@1] AND
[MyDatabase].[dbo].[Employees].[LastName]=[@2]) ORDERED FORWARD)

(1 row(s) affected)

Фиг. 2 План за изпълнение на разглежданата заявка, когато има индекс на колоните *FirstName* и *LastName*

В този случай при изпълнението на заявката:

```

SELECT EmployeeID
FROM Employees
WHERE LastName = 'name'

```

вместо търсене по индекса ще бъде приложено сканиране на индекса, тъй като в условието след WHERE не е използвана първата индексирана колона (фиг. 3).

```

SET SHOWPLAN_TEXT ON
GO
SELECT EmployeeID
FROM Employees
WHERE LastName = 'name'

```

Results

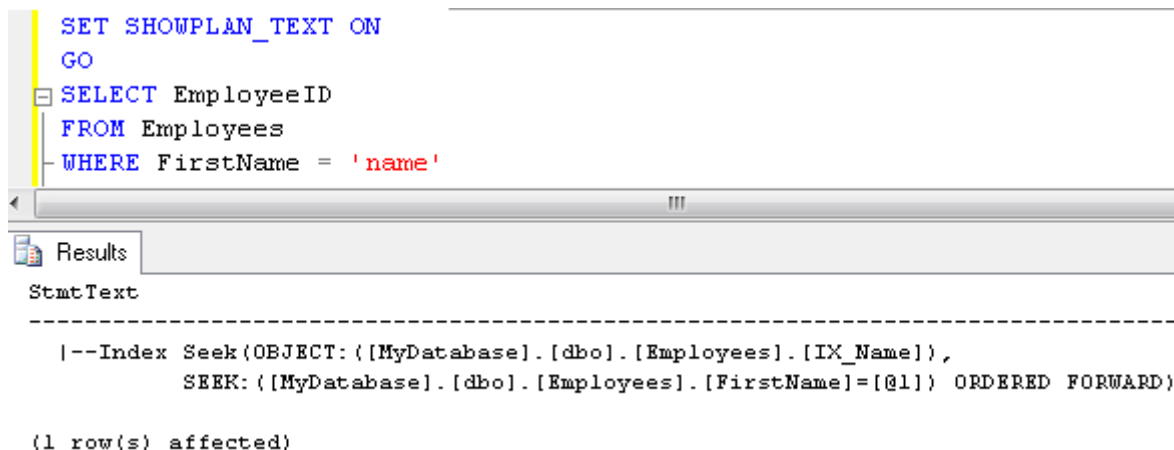
StmtText

--Index Scan(OBJECT: ([MyDatabase].[dbo].[Employees].[IX_Name]),
WHERE: ([MyDatabase].[dbo].[Employees].[LastName]=[@1]))

(1 row(s) affected)

Фиг. 3 План за изпълнение на заявка, в условието на която не е включена първата индексирана колона

Не е необходимо обаче да се включат всички индексирани колони, за да бъде избрано търсене по този индекс (фиг. 4).



```

SET SHOWPLAN_TEXT ON
GO
SELECT EmployeeID
FROM Employees
WHERE FirstName = 'name'

```

Results

StmtText

```

-----
|--Index Seek (OBJECT: ([MyDatabase].[dbo].[Employees].[IX_Name]),
      SEEK: ([MyDatabase].[dbo].[Employees].[FirstName]=[@1]) ORDERED FORWARD)

(1 row(s) affected)

```

Фиг. 4 План за изпълнение на заявка, в условието на която е включена само първата индексирана колона

Използване на клъстериран индекс

Само един индекс може да има в една таблица, който определя физическото подреждане на данните, затова е важно да бъде избран по подходящ начин. Препоръчително е всяка таблица да има клъстериран индекс и по подразбиране първичният ключ се реализира чрез клъстериран индекс. Възможно е обаче уникалността на първичния ключ да бъде наложена с неклъстериран уникален индекс (чрез добавяне на ключовата дума NONCLUSTERED при деклариране на ограничението PRIMARY KEY) и някой от другите индекси да стане клъстериран. Клъстерираните индекси са изключително полезни за диапазонни заявки (например `SELECT * FROM Sales WHERE SaleID BETWEEN 100 AND 200`) и за заявки, при които данните трябва да бъдат подредени в съответствие с клъстерирания ключ.

Избор на неклъстерирани индекси

Този тип индекс е изключително полезен при таблици с много редове, когато се изпълняват заявки, които трябва да върнат малко на брой редове. Ако индексираните колони са включени в условието на заявката, оптимизаторът на заявки избира създадения неклъстериран индекс, когато той е *високо-селективен*, т.е. неклъстерираният индекс трябва да бъде способен да елиминира голям процент от редовете, за да бъде полезен. Обикновено индекс, който не елиминира 95% от редовете, когато бъде изследван от оптимизатора на заявки, той не се използва. Ако индексът не отхвърли голям процент от редовете, по-добре е да се извърши сканиране, при което се чете всяка страница с данни точно един път.

Индексните колони често се използват за съединяване на таблици. Когато се създава ограничение PRIMARY KEY или UNIQUE, автоматично бива създаден и индекс. Но никакви индекси не се създават автоматично за рефериращите колони при ограниченията FOREIGN KEY. Такива колони често биват използвани за съединяване на таблици, така че те са почти винаги между най-подходящите за създаване на неклъстериран индекси.

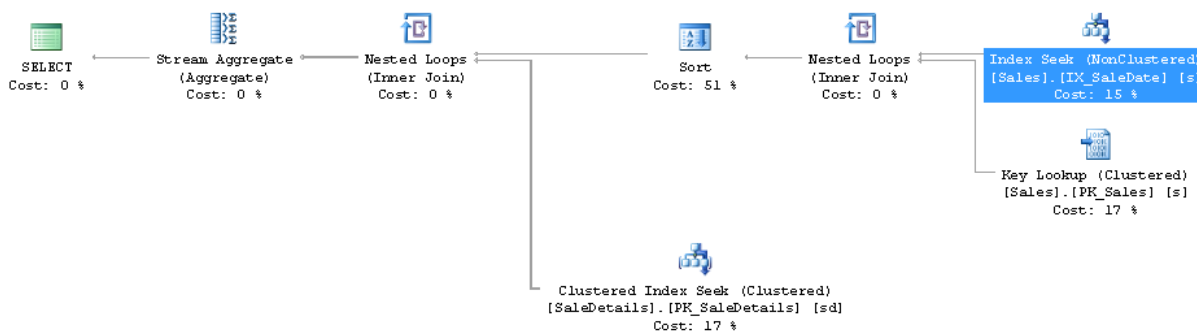
Индексите ускоряват извличането на данните, но от друга страна предизвикват извършването на допълнителна работа при модифицирането им, защото заедно с промените в данните трябва да бъдат поддържани и индексите. За да се избегне

създаването на прекалено много индекси, трябва да се вземе предвид честотата на актуализациите спрямо извличанията. Ако конкретната система е за *подпомагане на вземането на решения (Decision-Support System – DSS)* и рядко се извършват актуализации на съществуващите данни, има смисъл да има индекси, които да бъдат полезни за често изпълняваните заявки. Ако приложението е за *онлайн обработка на транзакции (Online Transaction Processing – OLTP)*, са необходими внимателно подбрани индекси в таблиците.

Пример 3 Действителният план за изпълнението на следната заявка (фиг. 5):

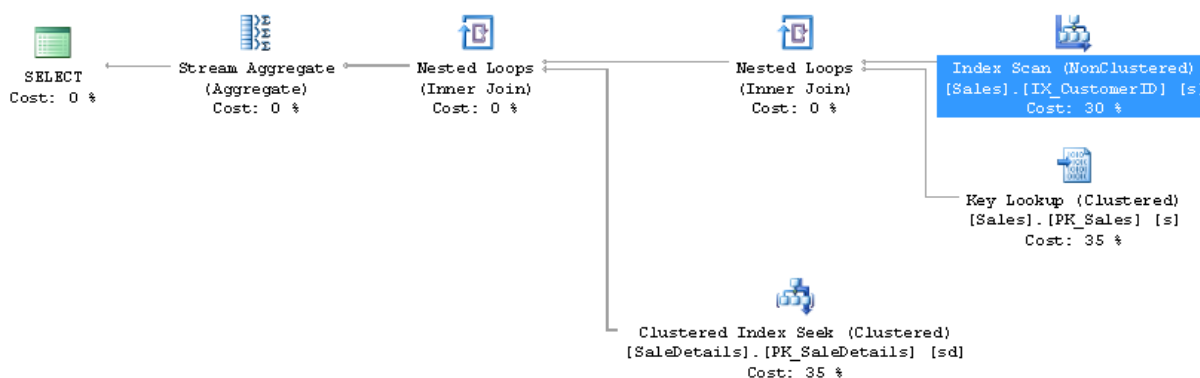
```
SELECT s.CustomerID,
       MAX(sd.quantity*sd.price*(1-sd.discount)*(1-s.discount))
       AS MaxSaleDetail
FROM Sales s
INNER JOIN SaleDetails sd ON s.SaleID = sd.SaleID
WHERE s.SaleDate BETWEEN '01/01/2006' AND '01/01/2011'
GROUP BY s.CustomerID
```

показва, че сървърът изпълнява търсене по неклъстерирания индекс на колоната `SaleDate` в таблицата `Sales`. Сървърът извършва сортиране, търсене по клъстерирания индекс в таблицата `SaleDetails`, накрая използва съединение чрез вложени цикли и прилагане на обобщаващата функция за връщане на резултата.



Фиг. 5 План за изпълнение на примерната заявка

Ако неклъстерираният индекс на колоната `SaleDate` в таблицата `Sales` не съществува, ще се наложи сканиране по неклъстерирания индекс на колоната `CustomerID` в таблицата `Sales`, както и филтриране, за да се ограничи резултатният набор до тези редове, за които `SaleDate` съответства на зададения интервал (фиг. 6).



Фиг. 6 План за изпълнение на примерната заявка, ако липсва индекс на колоната `SaleDate`

Съветници за настройка на индекси

SQL Server 2008 предлага инструмента *Database Engine Tuning Advisor*, който подпомага проектирането на възможно най-добрите индекси. Може да се стартира чрез избиране на *Query | Analyze Query in Database Engine Advisor* или *Tools | Database Engine Tuning Advisor* на Management Studio; както и от *Start | Programs | Microsoft SQL Server 2008 | Performance Tools | Database Engine Tuning Advisor*. С помощта на този инструмент може да се създаде една оптимална съвкупност от индекси за базата от данни. За да се използва *Database Engine Tuning Advisor*, трябва да се определи файл с работно натоварване (*workload*), на който съветникът базира своите препоръки. Всеки файл на SQL Server Profiler Trace може да бъде използван като файл с работно натоварване (.trc или .xml). SQL Server Profiler е инструмент на SQL Server с графичен интерфейс, който позволява да се наблюдават и записват събитията на ядрото на SQL Server във файл с работно натоварване. Един файл с работно натоварване може да съдържа множество от полета: категория на събитието, текст на събитието (например текст на заявка на Transact-SQL), начално време, продължителност на събитието. *Database Engine Tuning Advisor* може да извлича автоматично събития, свързани с ядрото (като конструкция на Transact-SQL) и полета от SQL Server Profiler Trace. Всеки файл, който съдържа множество от конструкции на Transact-SQL, също може да бъде използван като файл с работно натоварване (.sql).

- Опции, които *Database Engine Tuning Advisor* предлага (фиг. 7):

Limit tuning time За големи файлове с работно натоварване и големи бази от данни настройката на индексите може да отнеме значително време и ресурси. За да се намали времето и натоварването на сървъра, се дава възможност потребителят да укаже на съветника да бъде по-малко изчерпателен в търсенето на подходящи препоръки. Един цялостен анализ обаче (т.е. при изключване на тази опция) може да доведе до сумарно по-голямо подобряване на производителността. Опцията е включена по подразбиране.

The screenshot shows the 'Tuning Options' tab of the Database Engine Tuning Advisor. The 'Limit tuning time' checkbox is checked. The 'Stop at' field is set to '08 юли 2011 г.' and '22:30'. Under 'Physical Design Structures (PDS) to use in database', the 'Indexes' radio button is selected. Under 'Partitioning strategy to employ', the 'No partitioning' radio button is selected. Under 'Physical Design Structures (PDS) to keep in database', the 'Keep all existing PDS' radio button is selected.

Фиг. 7 Опции на съветника за индекси

Stop at Задаване на дата и час, когато *Database Engine Tuning Advisor* трябва да прекрати работата си.

Indexes and indexed views Чрез включването на тази опция потребителят указва на съветника да включи препоръки за добавяне на клъстерирани индекси, неклъстерирани индекси и индексирани изгледи.

Indexed views Определя включване на индексирани изгледи в анализа. Препоръките няма да се отнасят за клъстерирани и неклъстерирани индекси.

Include filtered indexes Препоръките трябва да съдържат добавяне на филтрирани индекси.

Indexes Съветникът трябва да включи само препоръки за добавяне на клъстерирани и неклъстерирани индекси.

Nonclustered indexes Изискват се само препоръки за неклъстерираните индекси.

Evaluate utilization of existing PDS only Оценява се ефективността на съществуващите индекси и не се предлагат допълнителни индекси или индексирани изгледи.

No partitioning Резултатът от работата на съветника не съдържа препоръки за дялове.

Full partitioning Чрез включването на тази опция потребителят указва на съветника да включи препоръки за дялове.

Aligned partitioning Препоръчаните нови дялове трябва да бъдат подредени, така че да се улесни поддържането на дяловете.

Do not keep any existing PDS Препоръките трябва да се отнасят и за изтриване на ненужните съществуващи индекси, изгледи и дялове.

Keep indexes only Запазват се всички съществуващи индекси, но се препоръчва изтриване на ненужните индексирани изгледи и дялове.

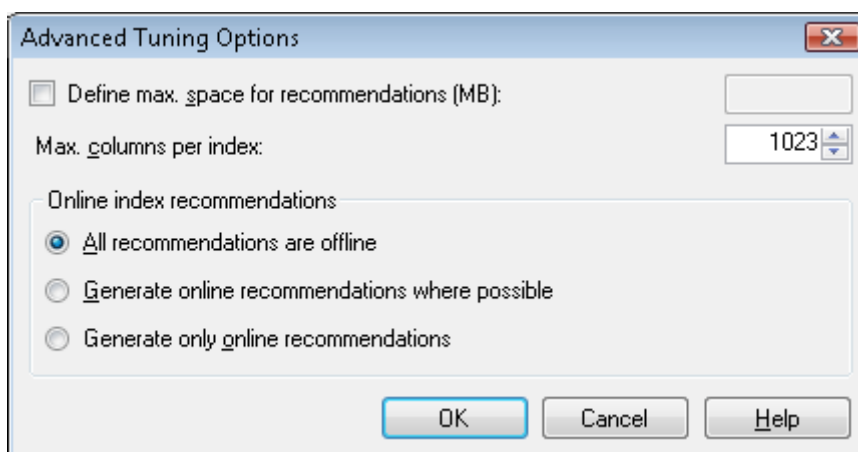
Keep all existing PDS Чрез включването на тази опция потребителят указва на съветника да не изтрива никакви съществуващи индекси, индексирани изгледи и дялове.

Keep clustered indexes only Запазват се всички клъстерирани индекси, но се препоръчва изтриването на ненужните индексирани изгледи, дялове и неклъстерирани индекси.

Keep aligned partitioning Запазват се всички подравнени дялове, но се препоръчва изтриване на ненужните индекси, индексирани изгледи и неподравнени дялове.

- Допълнителни опции в прозореца *Advanced Options* (фиг. 8):

Define max. space for recommendations (MB) Този параметър определя границата на общата сума на цялото пространство за съхраняване на всички индекси, индексирани изгледи и дялове, препоръчани от *Database Engine Tuning Advisor*.



Фиг. 8 Допълнителни опции на съветника за индекси

Max. columns per index Този параметър се установява, за да се зададе максималния брой на колоните, съставлящи индексите.

All recommendations are offline Съветникът трябва да генерира възможно най-подходящите препоръки, но така че да не се налага създаване на някои от структурите онлайн.

Generate online recommendations where possible При генериране на Transact-SQL конструкции, реализиращи направените препоръки, трябва да се избере метод, който да се изпълни, когато сървърът е онлайн, дори ако е налице по-бърз метод, изпълним при режим *offline*.

Generate only online recommendations Съветникът осигурява само тези препоръки, които позволяват на сървъра да остане онлайн.

- Отчети, генерирани от *Database Engine Tuning Advisor*

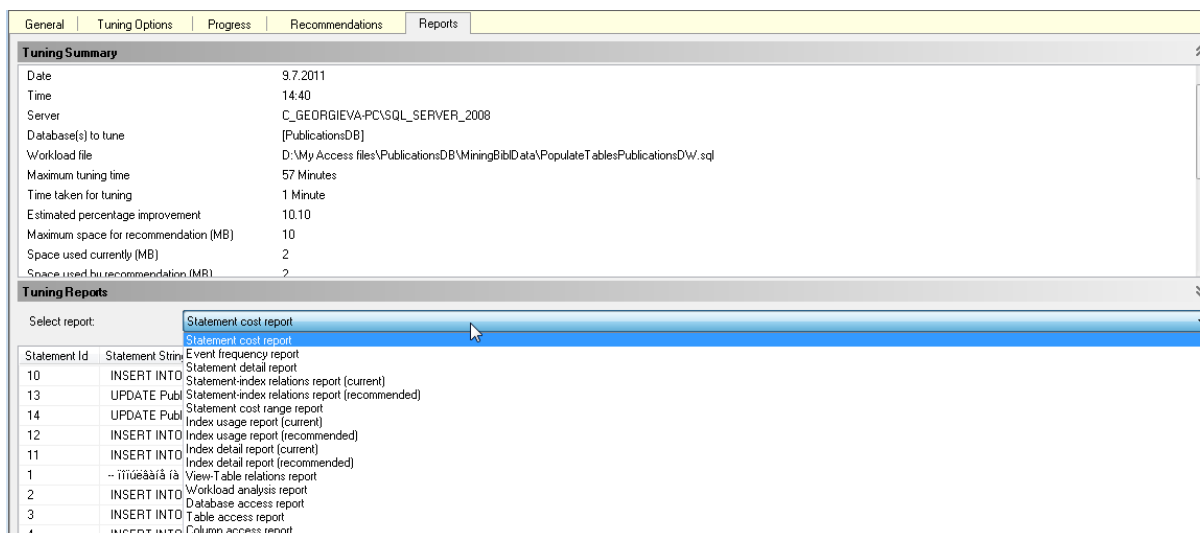
Най-важният резултат от *Database Engine Tuning Advisor* е комбинация от препоръчани индекси. Екранът *Index Recommendation* (фиг. 9) представя списъка с тези индекси, посочвайки определените имена на индекси, реда на колоните в индекса, дали индексът е клъстериран. Съветникът създава и оценка на очакваното подобрение в изпълнението на работно натоварване в сравнение със съществуващата конфигурация, като за целта използва оптимизатора на заявки. Тъй като оптимизаторът се основава на статистическа информация, действителната промяна в изпълнението може да бъде различна от предложената оценка.

General	Tuning Options	Progress	Recommendations	Reports				
Estimated improvement: 10%								
Partition Recommendations								
Index Recommendations								
<input checked="" type="checkbox"/>	Database Name	Object Name	Recommendation	Target of Recommendation	Details	Partition Scheme	Size (KB)	Definition
	PublicationsDB	[dbo].[Papers]	create	_dta_stat_1749581271_1_4				[Paper(D)] [Authors]
	PublicationsDB	[dbo].[Papers]	create	_dta_index_Papers_8_1749...			24	[Year(D)[Publishing] asc] include [Paper(D)] [PaperType(D)] [Journal(D)] [Publis
	PublicationsDB	[dbo].[Papers]	create	_dta_index_Papers_8_1749...			48	[Paper(D) asc] include [Authors]

Фиг. 9 Списък с препоръчани индекси

Database Engine Tuning Advisor генерира и отчети, които осигуряват бъдещ анализ на препоръките. Тези отчети оказват влияние върху решението за това дали препоръките да бъдат приети или отхвърлени. Всички отчети могат да бъдат експортирани в *.xml* файлове за бъдещи анализи.

Подстраницата *Tuning Summary* на страницата *Reports*, показана на фигура 10, осигурява важна обобщаваща информация за изпълнението на *Database Engine Tuning Advisor*.



Фиг. 10 Отчети на съветника за индекси

Някои от отчетите, които могат да бъдат избрани в подстраницата *Tuning Reports* на страницата *Reports*, са (фиг. 10):

Index Usage Report (recommended) представя информация за очакваното относително използване на препоръчаните индекси.

Index Usage Report (current) показва какъв процент от заявките, зададени в работното натоварване, използват всеки от съществуващите индекси. Тази информация е полезна, за да се определи дали даден индекс се използва и да се установи кои индекси трябва да бъдат изтрети, за да се освободи допълнително място и излишната поддръжка за неизползваните индекси.

View – Table Relations Report показва кои таблици участват в създаването на даден изглед.

Statement – Index Relations Report (recommended) предоставя информация за индексите, които биха се използвали при изпълнението на всяка заявка, ако предложените промени бъдат потвърдени.

Statement – Index Relations Report (current) показва кои от съществуващите индекси се използват при изпълнението на всяка заявка.

Statement Cost Report посочва на потребителя намаление или увеличение в цената за изпълнението на 100^{те} най-скъпи конструкции на Transact-SQL във файла с работно натоварване, ако препоръчаната конфигурация бъде приета.

Workload Analysis Report осигурява информация за честотите на командите SELECT, INSERT, UPDATE и DELETE и техните относителни въздействия върху общата цена на работното натоварване.

След като съветникът генерира отчетите си, които могат да се съхранят на диска, е възможно да се реализират препоръките незабавно или да се съхрани файл със скриптовете за създаване (и евентуално изтриване) на индекси.

Използване на подсказки в заявките

SQL Server оптимизира заявките автоматично и в повечето случаи избира най-подходящия план за изпълнение. Все пак съществуват ситуации, в които програмистът или разработчикът на базата от данни може да разбере по-добре данните, отколкото SQL Server. В тези случаи е предоставена възможност да се зададе изрично начинът, по който да се изпълни заявката чрез използване на подсказки.

Подсказки за съединения

Между ключовата дума за типа на съединението и ключовата дума JOIN, програмистът може да вмъкне подсказка за начина на съединяване, който да бъде използван. Възможностите за избор са HASH, LOOP, MERGE или REMOTE. REMOTE се използва, когато е дефиниран свързан сървър, дясната таблица е отдалечена таблица и лявата таблица съдържа малко на брой редове. Тогава съединението се извършва на сървъра, който съдържа дясната таблица.

Пример 4 При следната заявка се предизвиква изпълнението на операцията Hash Match:

```
SELECT c.Companyname, s.SaleID
FROM Customers c
INNER HASH JOIN Sales s ON c.CustomerID = s.CustomerID
```

Подсказки за индекси

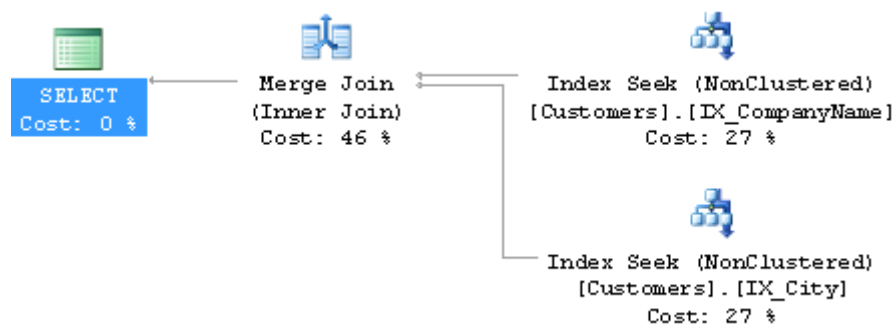
Чрез подсказките за индекси може да се зададе изрично обработката на заявката да се извърши, като се избере определен индекс за някоя таблица. Индексът може да бъде определен по името или идентификатора си. Освен това могат да се зададат няколко индекса за дадена заявка. Определянето на идентификатор на индекс 0, налага да се извърши сканиране на таблицата. Такава подсказка не е препоръчително да се използва, освен ако са направени необходимите проверки и те са потвърдили, че така ще се подобри производителността. Синтаксисът при включване на подсказки за индекси е следния:

```
SELECT column_list
FROM table_name
WITH ( INDEX ( {index_name | index_ID} [, ...] ) )
...
```

Използването на идентификатор на индекс е по-добре да се избягва, тъй като е възможно индексът да бъде изтрит и пресъздаден. Изключение прави клъстерираният индекс, той винаги има идентификатор 1.

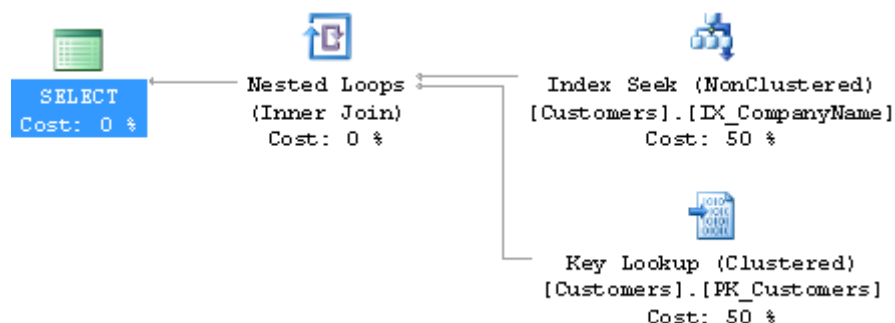
Пример 5 Заявка, при изпълнението на която е наложено изрично използване на неклъстерираните индекси съответно на колоните CompanyName и City:

```
SELECT CustomerID
FROM Customers
WITH (INDEX (IX_CompanyName, IX_City))
WHERE CompanyName = 'name' AND City = 'city'
```



Фиг. 11 План за изпълнението на заявката при включване на подсказка за използване на индексите

Ако не се включи подсказката, при изпълнението на тази заявка се използва само индекса на колоната `CompanyName` (фиг. 12).



Фиг. 12 План за изпълнението на заявката без включване на подсказка за използване на индексите

Подсказки за обработка на заявката

Тези подсказки се поставят в края на `SELECT` израза след ключовата дума `OPTION`. Допустимо е да се включат повече от една ключови думи след `OPTION`, но само една подсказка от всеки тип може да се използва. Синтаксисът при задаване на подсказки за обработка на заявки е следния:

```

SELECT column_list
FROM table_name
...
OPTION ( query_hint [,... n] )

```

Различните типове подсказки за обработка са:

- Подсказки за групиране – за задаване на начина, по който да се изпълняват групиращите операции. Възможностите са `HASH GROUP` или `ORDER GROUP`.
- Подсказки за обединения – `HASH UNION`, `MERGE UNION` или `CONCAT UNION` са възможните начини за указване на това как да се осъществи обединяването на различните резултатни набори. Ако в заявката е използвано обединение `UNION ALL`, не се налага премахване на дублиращите се редове и затова в този случай винаги се прилага конкатенация, т.е. `CONCAT UNION`.
- Подсказки за съединения – `LOOP JOIN`, `HASH JOIN` или `MERGE JOIN` могат да се зададат в `OPTION` както и в `JOIN`, но всяка подсказка в `OPTION` се прилага за всички съединения в заявката и припокрива тези в `JOIN`.
- `FAST n` – изисква `SQL Server` да избере този план за изпълнение, който ще извлече първите n на брой редове възможно най-бързо. Може да се наложи използването на неклъстериран индекс, който съответства на `ORDER BY` и позволява първите n реда да бъдат върнати бързо, след което да се обработят останалите от резултатния набор.
- `FORCE ORDER` – тази подсказка принуждава `SQL Server` да обработи таблиците в същия ред, в който те се появяват след `FROM`. Има смисъл само за вътрешни съединения, в противен случай – се игнорира.
- `MAXDOP n` – припокрива стойността на опцията на сървъра *max degree of parallelism* само за конкретната заявка. Числото n е максималната степен на паралелност.
- `ROBUST PLAN` – изисква да бъде избран този план за изпълнение на заявката, предназначен за възможно най-голям размер на редовете, независимо от вероятността за влошаване на производителността. Тази опция е полезна при

широки колони от символен тип с променлива дължина (*varchar*), тъй като не допуска оптимизаторът да включи в разглеждането определени типове планове, при които е възможно да се получи превишаване на ограничението за дължина на ред на таблица в SQL Server (за междинните таблици, създадени за извършване на обработката) и да се генерира грешка по време на изпълнението на заявката.

- **KEEP PLAN** – чрез тази опция се предотвратява честото прекомпилиране на заявката. След многократни актуализации на индексирани колони в таблицата заявката бива прекомпилирана автоматично. Използването на опцията **KEEP PLAN** гарантира, че заявката няма да бъде прекомпилирана толкова често, когато данните в таблицата бъдат модифицирани.

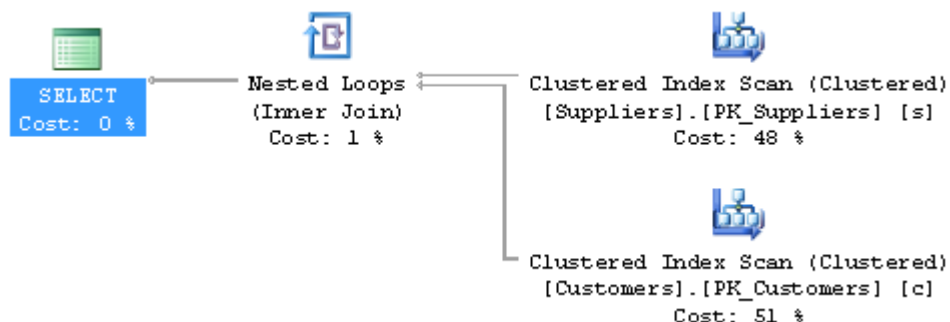
Пример 6 Заявка, в която са включени няколко подсказки за нейната обработка:

```
SELECT ProductName, SUM(quantity) AS TotalQuantity
FROM Products p
INNER JOIN SaleDetails sd ON p.ProductID = sd.ProductID
GROUP BY ProductName
OPTION (ORDER GROUP, MERGE JOIN, FAST 10)
```

Задачи

Задача 1. Да се разгледа графичният вид на плана за изпълнение, показан на фигура 13, на следната заявка:

```
SELECT c.CompanyName AS Customer,
       s.CompanyName AS Supplier
FROM Customers c
CROSS JOIN Suppliers s
WHERE c.City = s.City
```



Фиг. 13 План за изпълнението на дадената заявка

Нека индексите в таблиците *Customers* и *Suppliers* са съответно *PK_Customers* и *PK_Suppliers* – тези, които автоматично се създават, за да се наложи ограничението първичен ключ. Да се предложи начин за подобряване на производителността на тази заявка.

Решение:

Дадено е в предположенията на задачата, че двете колони, използвани в условието на заявката, не са индексирани. Следователно добавянето на индекси на тези колони ще позволи на оптимизатора да извърши търсене по индекс, вместо сканиране на клъстериран индекс. Следният скрипт създава тези индекси в таблиците:

```
CREATE INDEX IX_City ON Customers (City)
CREATE INDEX IX_City ON Suppliers (City)
```

Задача 2. Изследването на плана за изпълнение на дадена заявка показва, че се извършва сканиране по клъстерирания индекс на таблицата. След добавянето на неклъстериран индекс в таблицата на колоната, използвана за съединение, се оказва, че оптимизаторът продължава да изпълнява сканиране по клъстерирания индекс. По какъв начин може да се провери дали прилагането на търсене по неклъстерирания индекс ще доведе до оптимизиране на заявката?

Решение:

Включването на подсказка за индекс с името на съответния неклъстериран индекс ще задължи оптимизатора на заявки да го използва. След това се установява в кой от случаите заявката се изпълнява по-бързо чрез `SET STATISTICS TIME ON`.

Литература

1. Гарсиа-Молина Г., Дж. Ульман, Дж. Уидом, *Системы баз данных*, ИК “Вильямс”, 2002
2. Георгиев Е., *Научете сами SQL, Експрес дизайн*, София, 1998
3. Грубер М., *SQL – Професионално издание*, I и II том, СофтПрес, София, 2001
4. Дебета П., *Въведение в Microsoft SQL Server 2005 за разработчици*, СофтПрес, София, 2005
5. Ернандес М., *Проектиране на бази от данни*, СофтПрес, София, 2004
6. Крѐнке Д., *Теория и практика построения баз данных*, Питер, 2003
7. Пенева Ю., *Базы от данни*, Първа част, Регалия 6, София, 2004
8. Пенева Ю., Г. Тупаров, *Базы от данни*, Втора част, Регалия 6, София, 2004
9. Рачев Б., И. Въллова, С. Арсов, *Базы от данни: Проектиране, създаване и работа в средата на Oracle*, РУ „Ангел Кънчев”, Русе, 2002
10. Станек У., *Microsoft SQL Server 2000 Наръчник на администратора*, СофтПрес, София, 2001
11. Сукъп Р., К. Дилейни, *Microsoft SQL Server 7.0 Поглед отвътре – I и II том*, СофтПрес, София, 2000
12. Хулет Ф., *SQL Ръководство на програмиста*, СофтПрес, София, 2001
13. Шапиро Дж., *SQL Server 2000 Пълно ръководство за администриране*, ИнфоДАР, София, 2001
14. Шапиро Дж., *SQL Server 2000 Пълно ръководство за програмиране*, ИнфоДАР, София, 2001
15. Bieniek D., R. Dyess, M. Hotek, J. Loria, A. Machanic, A. Soto, A. Wiernik, *Microsoft SQL Server 2005 Implementation and Maintenance – Training Kit*, Microsoft Press, 2006
16. Connolly T., C. Begg, *Database Systems: A Practical Approach to Design, Implementation and Management*, Fourth Edition, Addison-Wesley, 2004
17. Microsoft Corporation, *MCSE Training: Microsoft SQL Server 2000 – Проектиране и реализация на бази данни*, I и II том, СофтПрес, София, 2001

Адреси в Интернет

18. <http://free.techno-link.com/database>
19. <http://www.databasejournal.com>
20. <http://www.informit.com>
21. <http://www.microsoft.com/technet>
22. <http://www.microsoft.com/sql>
23. <http://www.searchdatabase.com>
24. <http://www.sqlmag.com>
25. <http://www.sql-server-performance.com>

ПРАКТИКУМ ПО БАЗИ ОТ ДАННИ

ЧАСТ I

Автор:

гл. ас. д-р Цветанка Георгиева-Трифорова

Българска

Второ преработено и допълнено издание

Рецензенти: доц. д-р Владимир Димитров, доц. д-р Емилия Тодорова

Цветанка Георгиева-Трифенова

ПРАКТИКУМ ПО БАЗИ ОТ ДАННИ

ЧАСТ I

Основни теми:

- Проектиране на релационни бази от данни
- Използване на SQL за:
 - ❖ дефиниране на данни
 - ❖ извличане на данни
 - ❖ манипулиране на данни
- Анализиране и оптимизиране на достъпа до данните

Включените в темите примери са достъпни от:

- <http://practicum.host22.com>