Виталий Трунин

ПУТЬ ПРОГРАММИСТА T-SQL

T-SQL ПУТЬ ПРОГРАММИСТА

Виталий Трунин

Самоучитель по языку Transact-SQL. Основы программирования для начинающих разработчиков

Оглавление

Предисловие

Здравствуй уважаемый читатель!

Данная книга предназначена для тех людей, которые не знают языка T-SQL и даже об SQL мало что слышали, но хотят или им нужно освоить данный язык, иными словами, книга для начинающих.

Идея написания данной книги у меня возникла, когда я четко осознал для себя, что все книги по SQL и в частности по T-SQL, которые я читаю, для меня-то может и понятны, но для человека, не имеющего опыта и знаний в языке SQL, они сложные и непонятные! Поэтому если Вы хоть раз начинали читать книгу по SQL или T-SQL и потом благополучно бросали это дело, я Вас понимаю, и специально для Вас я написал данную книгу. Она отличается от всех тех, с которыми я сталкивался. В ней я делаю упор на доступность и усвояемость получаемой информации, а не на глубину. Вы узнаете все то, что Вам потребуется для старта, и даже больше, а если Вы что-то не найдете в данной книге, то Вы легко сможете найти нужную Вам информацию в других источниках, так как Вы уже будете понимать, что Вам нужно и для чего!

В ней я буду рассказывать о личном опыте становления программистом T-SQL, т.е. свой путь от самого начала и до текущего момента, а этот путь был нелегок, так как я, не имея профессионального образования, освоил язык T-SQL и продолжаю осваивать дальше. В этой книге я расскажу про мой путь, а также попытаюсь дать Вам советы, чтобы Ваш путь был немного легче или может быть более правильным, так как я, конечно же, допускал ошибки, бросал изучение, но набирался сил и двигался дальше. Для этого по тексту книги я буду в разных местах писать советы по использованию той или иной инструкции T-SQL или свое отношение к той или иной ситуации, основанные только на личном опыте! В заключении я соберу их все вместе, получилось их 26, так сказать «***26 советов от Виталия Трунина по работе с T-SQL!***», чтобы Вы могли прочитать их все разом и лучше запомнить, а затем применять их на практике. При этом не стоит их воспринимать как стандарт, эти советы - всего лишь опыт одного человека!

Этой книгой я хочу показать Вам, что, даже если у Вас нет возможности пойти учиться по специальности, Вы все равно можете добиться своего, т.е. стать программистом, и речь здесь идёт не только о языке T-SQL – главное иметь цель!

В этой книге я затрону все те аспекты языка T-SQL и Microsoft SQL Server, которые обязательно рано или поздно Вам пригодятся, начиная с пошагового описания процесса установки Microsoft SQL Server и заканчивая сложными конструкциями языка T-SQL. Многое, конечно, я просто не в силах осветить, но это и не цель данной книги. Ее цель - дать Вам возможность освоить язык T-SQL в той мере, в которой это доступно абсолютно каждому. Я начну с основ, с того, с чего нужно начать, все буду объяснять простым доступным языком так, как мне самому бы хотелось, чтобы мне объяснили с самого начала моего пути (*многое, что я читал, узнавал в самом начале, было трудно для понимания и восприятия*), и постепенно буду двигаться к более сложному, в процессе, конечно же, буду приводить много примеров. Методику обучения, которую я применяю в книге, я условно назвал «*Простая последовательная*», суть ее в том, что Вы последовательно переходите от простого и общего к сложному и углублённому. Таким образом, ближе к завершению книги Вы будете наблюдать примеры T-SQL инструкций (*в книге их немало!*), которые, если бы Вы осваивали в самом начале (*как это делается в некоторых других книгах*), Вы бы их просто не поняли, но в этой книге Вы безусловно их будете понимать.

Я не претендую на звание «*самый лучший программист по T-SQL*», но я имею неплохой опыт разработки бизнес логики на данном языке. Весь свой опыт и знания я приобрёл исключительно в рамках саморазвития, я уже говорил, что не имею профессионального образования, но я читал книги, проходил курсы, перенимал опыт других программистов и на текущий момент я имею те знания, которыми Вы будете владеть после прочтения этой книги.

После того, как Вы ее прочтете, Вы без всякого труда сможете писать SQL запросы, разрабатывать процедуры и функции на языке T-SQL, использовать встроенные возможности Microsoft SQL Server и многое другое.

Поэтому, если Вы ничего не знаете о языке T-SQL и о Microsoft SQL Server в целом, но хотите знать, обязательно прочитайте данную книгу.

Введение

Прежде чем начать свой рассказ о языке T-SQL, я расскажу Вам немного о себе, о том, как я пришел в сферу информационных технологий и связал свой жизненный путь с языком T-SQL и с Microsoft SQL Server. Именно здесь Вы узнаете мой путь становления программистом T-SQL.

Об авторе

Меня зовут Виталий Трунин, я родился, вырос и живу в небольшом провинциальном городе. Окончил местный университет по специальности «*Финансы и кредит*».

На текущий момент моя работа связанна с Microsoft SQL Server и языком T-SQL. У меня есть семья, любимая супруга и две замечательные дочурки.

Начало пути программиста

Наверное, всем известно, что после школы нужно идти учиться по той специальности, с которой Вы хотите связать свою жизнь. Я только сейчас понимаю важность образования, раньше я этого не понимал, поэтому и пошел учиться просто на престижный факультет в местный университет, а специальность меня даже не интересовала. Честно скажу, что IT сфера на тот момент меня тоже не интересовала, меня интересовал спорт.

После окончания университета увлечение спортом прошло. У меня стояла задача найти работу, первое, конечно же, что я начал делать - это искать работу по специальности, пока еще в памяти были хоть какие-то знания. Но в небольшом городе без опыта и связей найти работу по моей специальности оказалось очень трудно, поэтому я решил начать с чего-нибудь другого, менее пафосного.

Я устроился в магазин товароведом, спустя примерно 6 месяцев я прекрасно осознавал, что это не для меня, все, что я делал, меня никак не интересовало и больше того, мне не нравилось.

И здесь ключевую роль сыграла моя замечательная супруга, с который мы были женаты с четвертого курса. Она отметила мое увлечение компьютерами. Она думала, что, если я что-то там делаю за компьютером, могу устанавливать программы и все такое, я являюсь программистом, и мне это нравится. Увлечение компьютерами, конечно же, было, но даже тогда я понимал, что я обладал далеко не теми знаниями, которыми должен обладать простой программист или системный администратор.

И она мне посоветовала попробовать поработать в этой сфере. Без профильного образования и без опыта работы я все же нашел работу IT-шника. Это была самая низкооплачиваемая работа в одной государственной структуре.

В итоге, даже на такой должности мне стало настолько интересна эта сфера, что я стал изучать все, с чем я только сталкивался. Я читал книги, проходил обучающие курсы, внимательно слушал и запоминал, когда старшие более опытные сотрудники говорили что-то и делали, я учился у них.

В этот период мне стала интересна тема web-программирования и я создал сайт Info-Comp.ru, который функционирует и по сей день, я на нем публикую различные статьи на тему IT, включая статьи по T-SQL.

И именно тогда я понял, что с направлением своего образования я промахнулся, именно тогда я понял важность этого образования. Я даже немного сожалел, что мне приходиться только сейчас получать такую ценную и важную информацию, которую я мог бы получить ранее.

Как говорит Брайан Трейси, автор многих мотивационных книг и курсов.

> *Никто не лучше вас. Никто не умнее вас. Просто они раньше начали.*

И это на самом деле так, я понимал, что люди, которые занимали высокие позиции в других высокооплачиваемых компаниях, и которым было ненамного больше лет, чем мне тогда, просто раньше начали получать необходимые им знания для достижения высоких результатов.

Поэтому советую всем уделить особое внимание базовому высшему образованию, не просто окончить университет, а попытаться получить максимум знаний и полезной информации. Поверьте, это

того стоит. Многое, конечно, Вам может и не потребоваться, но у Вас будет та база, с помощью которой Вы можете сделать из себя профессионала в своей области, и стать высокоразвитой личностью.

Развитие и становление программистом

Уже через 1,5 года работы IT специалистом в этой организации, я стал начальником отдела, но я все равно понимал, что знания, которыми я обладаю, очень малы, и мне хотелось бы получать все больше и больше знаний и, конечно же, применять их на практике, что в той организации было невозможно.

В связи с этим я устроился работать в компанию, которая уже платила больше, но и требования у них были выше. При этом в момент трудоустройства я думал, что меня не возьмут, так как некоторого опыта, например работы с SQL, у меня не было. Меня взяли, и это был новый этап в моей жизни, а именно знакомство с SQL.

Руководство понимало, что у меня еще знаний недостаточно, но, тем не менее, давало мне различного рода задачи, связанные с SQL. Я, со своей стороны, был только рад, так как это была практика, а теорией я занимался в свободное от работы время (*книги, статьи, обучающие курсы*).

Уже через год я занимался сопровождением базы данных в роли программиста. Еще спустя некоторое время я стал ведущим специалистом в компании по разработке и сопровождению баз данных. Я разрабатывал бизнес-логику на языке T-SQL, участвовал в проектах, в которых был задействован не только язык T-SQL, а также разрабатывал клиентскую часть приложений. Иными словами, на протяжении нескольких лет я продвигался не только по карьерной лестнице, но и постоянно развивался профессионально и личностно (*и развиваюсь до сих пор!*). Точнее даже сказать, что продвижение по службе - это всего лишь следствие моего профессионального развития.

В своей работе я сталкиваюсь с людьми, которые думают, что освоить язык T-SQL - это очень сложно только потому, что они просто не понимают того материала, тех книг и тех инструкций, которые они читают. При этом эти же люди стали отмечать, что тот формат, то изложение, в котором я преподношу необходимую им информацию, удобен и понятен. Поэтому я решил написать книгу, в которой я расскажу о языке T-SQL так, как я умею, а не так как это делают другие авторы. Хотя после прочтения данной книги я Вам рекомендую прочитать и другие книги по T-SQL, так как они Вам уже будут казаться не такими уж и сложными.

Благодарность

Данную книгу я посвящаю своей семье и, в частности, своей любимой супруге Екатерине. Благодаря ей я начал заниматься тем делом, которое мне нравится. Я стал развиваться, как профессионально, так и личностно, с целью стать достойным своей супруги, а она мне в этом безусловно помогает. За это ей большое спасибо! Кать, я люблю тебя!

Также она принимала непосредственное участие в разработке данной книги, она выступала в качестве редактора, и не раз прочитала данную книгу с целью устранить возможные ошибки и опечатки. Честно сказать - это очень сложное дело, писать книгу, поэтому, даже после того как мы много раз прочитали и редактировали данную книгу, в ней могут остаться ошибки или опечатки. И я сразу прошу у Вас прощения за это, не судите строго.

Итак, я думаю, пора приступать к изучению языка T-SQL!

Глава 1 - Базы данных

Теория баз данных

В данной книге мы, конечно же, будем разговаривать о реляционных базах данных, ведь понятие база данных - гораздо более широкое чем то, для чего нам нужен язык T-SQL. Под базой данных можно понимать любой набор информации, которую можно найти в этой базе данных и воспользоваться ей.

Реляционная база данных – это упорядоченная информация, связанная между собой определёнными отношениями. Представлена она в виде таблиц, в которых и лежит вся эта информация. За счет того, что информация упорядочена, разделена на определённые сущности, и представлена в виде таблиц, к ней легко получить доступ, т.е. найти нужную нам информацию. В основе реляционной модели лежит теория множеств, которая подразумевает объединение разных объектов в одно целое, под одним целым в базе данных как раз и имеется в виду таблица. Язык T-SQL работает с множеством, т.е. с таблицами (*или набором данных*) – это Вы должны четко понимать!

Я не зря делаю упор на таблицы, поскольку представление базы данных у Вас в голове, в Вашем воображении, должно быть именно такое. Иными словами, представьте таблицу, затем еще одну и еще, а затем свяжите их, например, атрибутом (*характеристикой*), который есть во всех трех таблицах. Представили? Вот примерно так Вы должны будете представлять себе базу данных, для того чтобы эффективно строить запросы к этой базе данных, т.е. к этой информации, которую Вы только что представили.

Совет 1

Всегда визуализируйте информацию в базе данных в виде таблиц, это поможет Вам лучше выстроить связь между таблицами и написать необходимый SQL запрос.

Без представления структуры реляционной базы данных запрашивать информацию из этой базы или дорабатывать ее Вам будет сложно, практически невозможно. Поэтому для общего понимания базы данных, визуализируйте у себя в голове базу данных в виде таблиц, связанных между собой.

Например, ниже таблица 1 будет содержать сведения о предметах мебели, а Таблица 2 о материалах, из которых они изготовлены, связаны они с помощью идентификатора. Другими словами, в Таблице 1 в столбце «*Идентификатор из таблицы 2 (Id)*» будет храниться ссылка на запись из Таблицы 2, и соотнеся эту ссылку с исходной записью в Таблицы 2 мы будем понимать из какого материала сделан тот или иной предмет.

Таблица 1 – Предметы мебели.

Идентификатор, ключевой столбец (Id)	Наименование предмета	Идентификатор из таблицы 2 (Id)
1	Стул	2
2	Стол	1

Таблица 2 – Материал, из которых изготовлены предметы мебели.

Идентификатор, ключевой столбец (Id)	Материал
1	Дерево
2	Железо

В итоге мы видим, что стул сделан у нас из железа, а стол из дерева.

У Вас, наверное, возник вопрос, почему я перечень материалов вынес в отдельную таблицу, да потому, что это отдельная сущность. В реляционных базах данных есть такое понятие, как «**Нормализация**» - это своего рода процесс удаления избыточных данных. Другими словами, каждая сущность должна храниться отдельно, а в случае необходимости использования этой сущности в другой таблице, на нее делается ссылка, т.е. выстраивается связь.

Существует несколько так называемых *«Нормальных форм»* базы данных. В основном, их пять, и база данных считается нормализованной, если ее таблицы представлены как минимум в третьей нормальной форме. Это примерно означает, что в третьей нормальной форме каждая характеристика сущности зависит от ключа (*ссылки, идентификатора, ведущего на основную запись*).

Например, в нашем случае у сущности «*Предмет мебели*» есть одна характеристика, т.е. материал, из которого изготовлен данный предмет. Мы, вместо того чтобы указать текстом название материала, вынесли перечень материалов в отдельную таблицу и поставили на нее ссылку, таким образом, мы можем сказать, что наша маленькая база находится в третьей нормальной форме!

Для чего же нужна нормализация? А она нужна для того, чтобы увеличить быстродействие, ускорить сортировку данных, уменьшить размер данных, а также увеличить гибкость этой базы данных. Например, в случаях, если у Вас возникла необходимость подкорректировать название материала, вместо «*Дерево*», Вам нужно, чтобы было «*Натуральное дерево*», в нашем примере Вам достаточно один раз изменить его в Таблице 2, но если бы мы не вынесли перечень материалов в отдельную таблицу, а указывали их как текст, то нам бы пришлось изменять название во всех записях таблицы 1! А теперь представьте, что их миллион!

Или, например, пользователь случайно ввел название с ошибкой, например, не «Дерево», а «*Дкрево*», когда создавал запись в таблице 1 (*в случае без использования ссылки и таблицы 2*). Тогда в нашей таблице в скором времени может оказаться и «*Дерево*», и «*Дкрево*», «*Днрево*», и вообще что угодно, ведь это просто текст. С использованием ссылок это исключено! Вы не переживайте, идентификаторы пользователи вводить не будут, в приложении (*которое будет работать с этими данными*) программисты разрабатывают специальный функционал, позволяющий выводить вместо ссылок полное значение.

Как Вы понимаете, в нашей тестовой базе данных всего 2 таблицы с минимумом столбцов. Но если таблиц, то есть сущностей, много, и много характеристик, то разработчики иногда прибегают к умышленной денормализации базы данных, т.е. обратному процессу. Например, при доработке таблицы, в частности, добавлении новой характеристики, они не выносят ее в отдельную таблицу. Это делается для того, чтобы повысить производительность запросов, ведь чрезмерная нормализация базы данных может привести к тому, что при каждой необходимости получить данные, нужно будет обращаться к нескольким таблицам. А если таких связей много, то и таблиц, к которым нужно обращаться, будет много, а это ресурсоемкий и трудоемкий процесс.

Мы с Вами рассмотрели пример базы данных. И именно к такой структуре информации мы будем учиться обращаться, а также дорабатывать эту структуру, и добавлять новую информацию, разрабатывать функционал, позволяющий анализировать эту информацию, и многое другое. Именно это и будет делать программист T-SQL.

Углубляться в теорию баз данных мы не будем. Если Вам интересна эта тема, есть много отдельных книг, которые Вы можете прочитать, но в нашем случае, т.е. для старта программирования на T-SQL, понимание того, что база данных представлена в виде информации, лежащей в таблицах, будет достаточно!

Система управления базами данных (СУБД)

Мы выяснили, что база данных - это некая информация, которая лежит в таблицах. Но для того чтобы хранить эту информацию, управлять и модифицировать ее, нужна специальная программа. Программа, это, конечно, мягко говоря, на самом деле нужна целая информационная система, и такой системой выступают «*Системы управления базами данных*», сокращенно СУБД.

Существует достаточно много различных СУБД, большинство из них стоят немалых денег. Среди всех выделяются по своим возможностям и популярности буквально 3-4 системы, одной из которых и является детище одной из самых крупных IT компаний мира - это Microsoft SQL Server.

Microsoft SQL Server – это не просто система управления базами данных, это целый комплекс приложений, позволяющий хранить и модифицировать данные, анализировать их, осуществлять безопасность этих данных, и многое другое. Лично я ни разу не пожалел, что судьба меня свела именно с этой СУБД и именно с языком T-SQL!

Итак, прежде чем продолжить изучение этой СУБД, Вам нужно подготовить площадку для обучения, тестирования и выполнения всех примеров, которые будут описаны в данной книге.

Успешное обучение возможно только совместно с практикой!

Наполеон Хилл - известный писатель, он является одним из тех, кто основал современный жанр «*самопомощь*», говорил

> *Мастерство приходит только с практикой и не может появиться лишь в ходе чтения инструкций.*

У Microsoft SQL Server есть несколько редакций, одной из которых является **редакция Express**. Она по своей сути и предназначена для обучения. Поэтому под подготовкой площадки я имею в виду установку Microsoft SQL Server в редакции Express. Не переживайте, данная редакция бесплатна, ее для таких целей (*обучения в домашних условиях*) использовать можно.

Если Вы не будете подкреплять полученные знания на практике, то продолжать читать дальше эту книгу бессмысленно. Другими словами, обязательно установите себе на домашний компьютер эту систему, а как это делается, я расскажу и покажу чуть ниже.

Установка Microsoft SQL Server Express

Перед тем как переходить к установке Microsoft SQL Server в редакции Express, Вам необходимо знать, что у данной редакции есть несколько ограничений, например:

- Можно задействовать 1 физический процессор и только 4 ядра;
- Максимальный объем оперативной памяти, который может быть задействован, это 1 ГБ;
- Максимальный размер базы данных 10 ГБ.

Эти ограничения на наше обучение никак не влияют, так как у нас нет необходимости получения максимальной производительности от SQL Server, нам нужно всего лишь, чтобы он функционировал.

Также, для того чтобы установить Microsoft SQL Server Express, Ваш компьютер должен отвечать некоторым системным требованиям, а именно:

- Поддерживаются следующие операционные системы: Windows 8, Windows 8.1, Windows 10, Windows Server 2012, Windows Server 2012 R2 и Windows Server 2016;
- Процессор, совместимый с Intel, с частотой 1 ГГц или выше;
- Оперативной памяти минимум 512 МБ, но рекомендую, конечно же, больше;
- Свободного места на диске как минимум 4,2 гигабайта.

Если Ваш компьютер отвечает всем системным требованиям, можете переходить к процессу установки.

Шаг 1 – Скачивание установщика

Данную редакцию можно официально скачать с сайта Microsoft, я вообще советую всем и всегда скачивать все программное обеспечение только с официальных источников.

Чтобы скачать Microsoft SQL Server Express, перейдите на страницу загрузки SQL Server - https://www.microsoft.com/ru-ru/sql-server/sql-server-downloads

На момент написания книги актуальная версия данной СУБД - это 2017, ее мы и скачиваем. Для того чтобы скачать Microsoft SQL Server в редакции Express, нажимайте скачать в разделе «*Выпуск Express*» (Рис. 1).

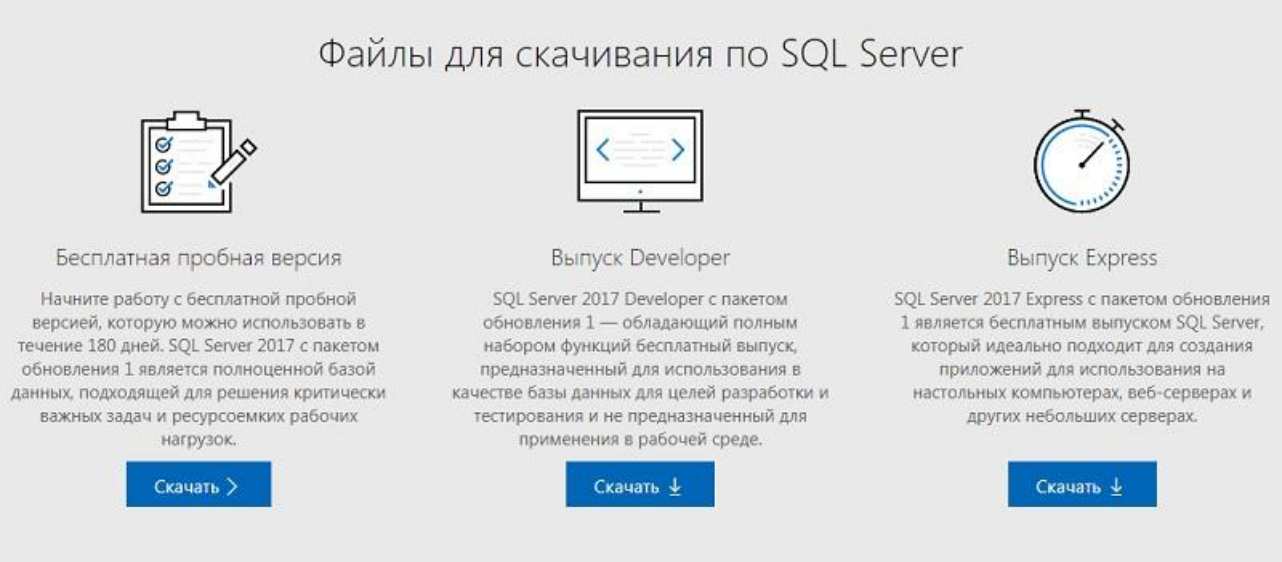


Рис. 1

Также скачать Microsoft SQL Server Express Вы можете и на официальной странице сайта Microsoft, посвященной данному выпуску https://www.microsoft.com/ru-ru/sql-server/sql-server-editions-express.

В итоге у Вас должен загрузиться файл web-установщика **SQLServer2017-SSEI-Expr.exe,** его размер около 5 мегабайт.

Шаг 2 – Запуск установщика и выбор типа установки

Когда Вы скачаете файл, запускайте его, после чего запустится приложение, с помощью которого Вы выберите необходимый тип установки и загрузите все необходимые для Microsoft SQL Server Express установочные файлы.

Для того чтобы посмотреть на расширенный процесс установки с возможностью выбора дополнительных компонентов, выберите «*Пользовательский*» тип. «*Базовый*» тип предполагает установку только ядра СУБД. Чтобы просто скачать дистрибутив Microsoft SQL Server без установки на текущем компьютере, можете выбрать тип «*Скачать носитель*». Я рекомендую Вам выбрать «*Пользовательский*» тип (Рис. 2).

Шаг 3 – Выбор места сохранения установочных файлов

Затем установщик попросит Вас указать каталог, в который необходимо загрузить все установочные файлы. По умолчанию создается каталог «SQLServer2017Media» на диске C. Нажимаем «*Установить*» (Рис. 3).

После чего начнется загрузка файлов установки (Рис. 4).

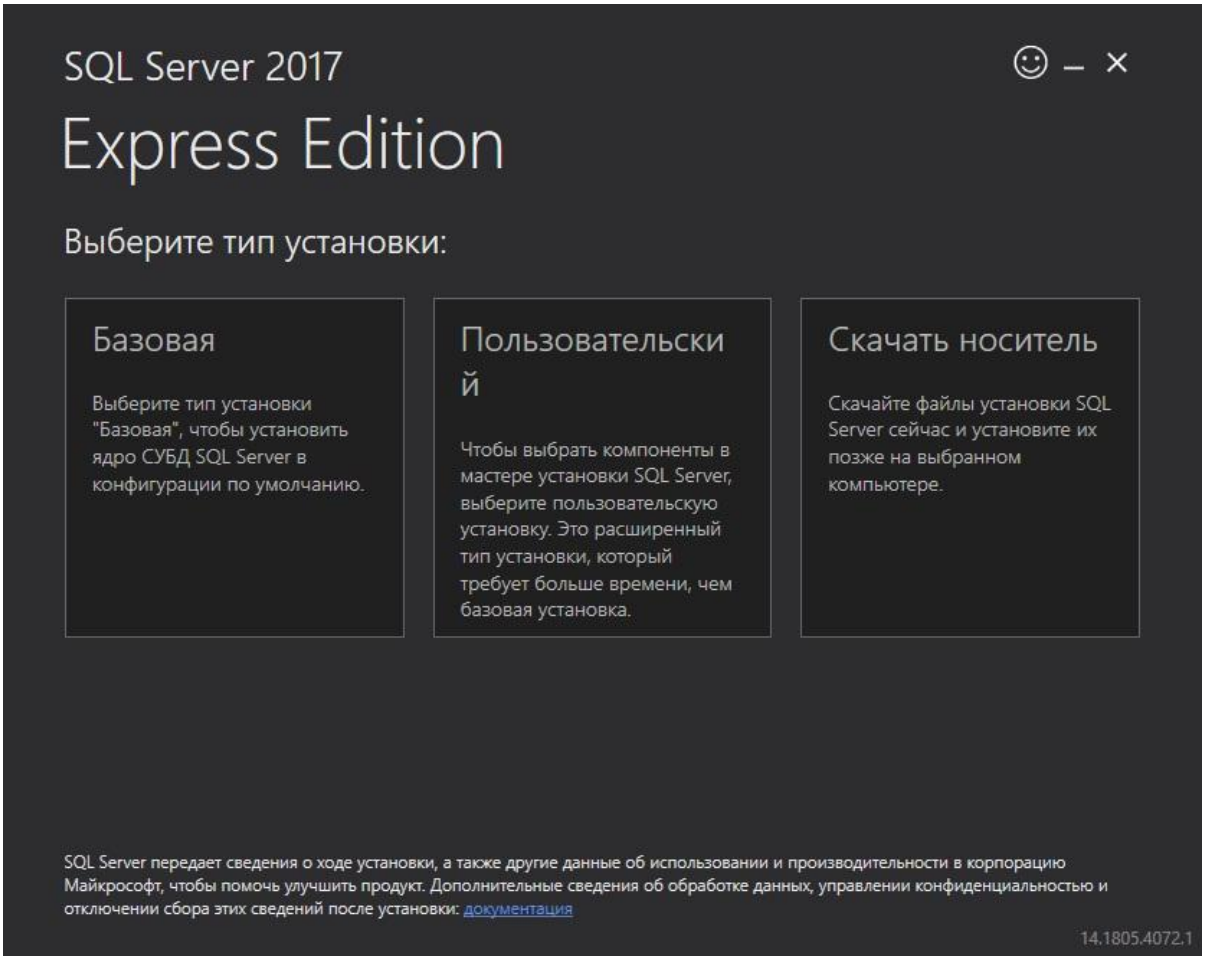

SQL Server 2017
Express Edition
Выберите тип установки:
Базовая
Выберите тип установки "Базовая", чтобы установить ядро СУБД SQL Server в конфигурации по умолчанию.
Пользовательский
Чтобы выбрать компоненты в мастере установки SQL Server, выберите пользовательскую установку. Это расширенный тип установки, который требует больше времени, чем базовая установка.
Скачать носитель
Скачайте файлы установки SQL Server сейчас и установите их позже на выбранном компьютере.
SQL Server передает сведения о ходе установки, а также другие данные об использовании и производительности в корпорацию Майкрософт, чтобы помочь улучшить продукт. Дополнительные сведения об обработке данных, управлении конфиденциальностью и отключении сбора этих сведений после установки: документация
14.1805.4072.1

Рис. 2

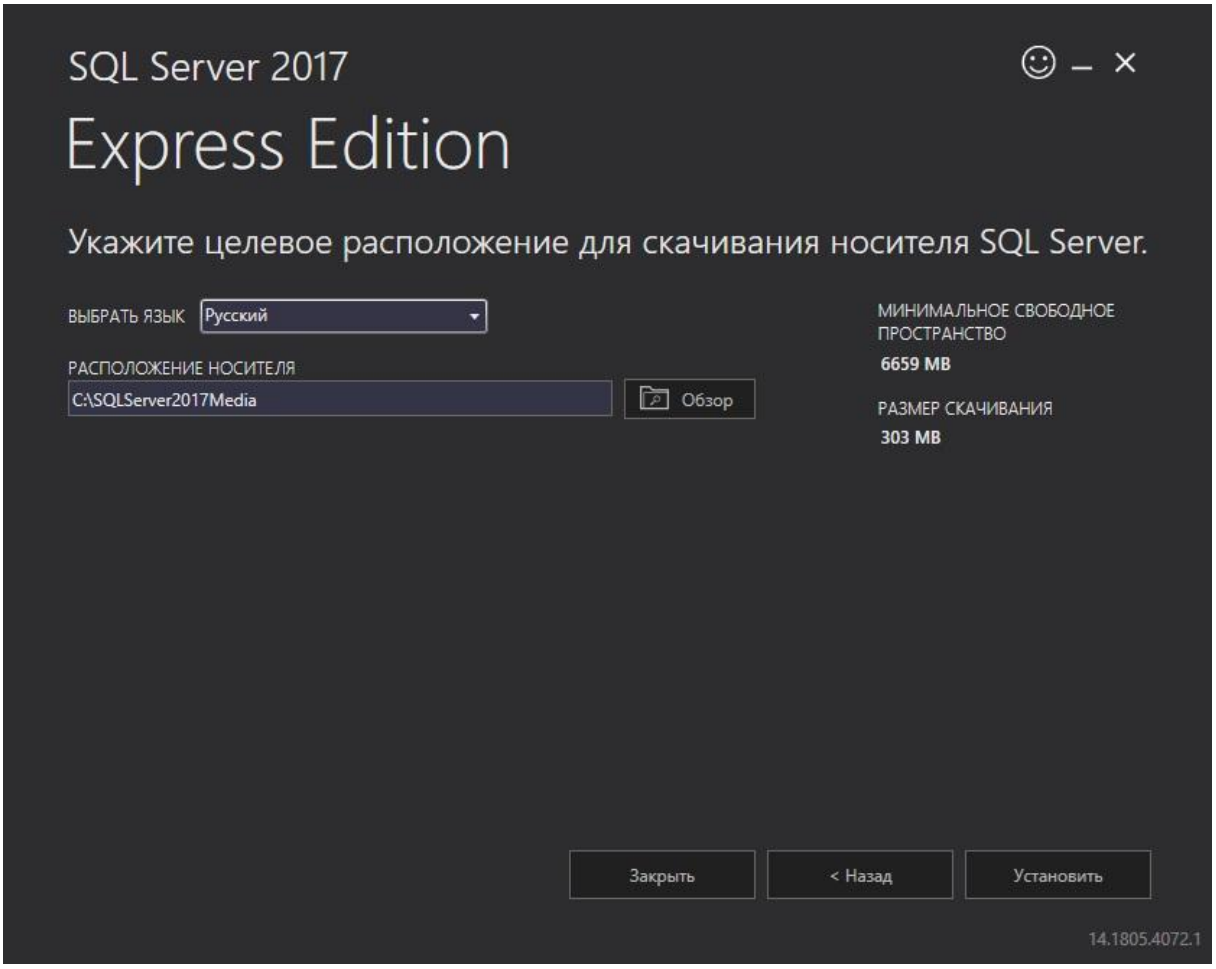

SQL Server 2017
Express Edition
Укажите целевое расположение для скачивания носителя SQL Server.
ВЫБРАТЬ ЯЗЫК
Русский
РАСПОЛОЖЕНИЕ НОСИТЕЛЯ
C:\SQLServer2017Media
Обзор
МИНИМАЛЬНОЕ СВОБОДНОЕ ПРОСТРАНСТВО
6659 MB
РАЗМЕР СКАЧИВАНИЯ
303 MB
Закрыть
< Назад
Установить
14.1805.4072.1

Рис. 3

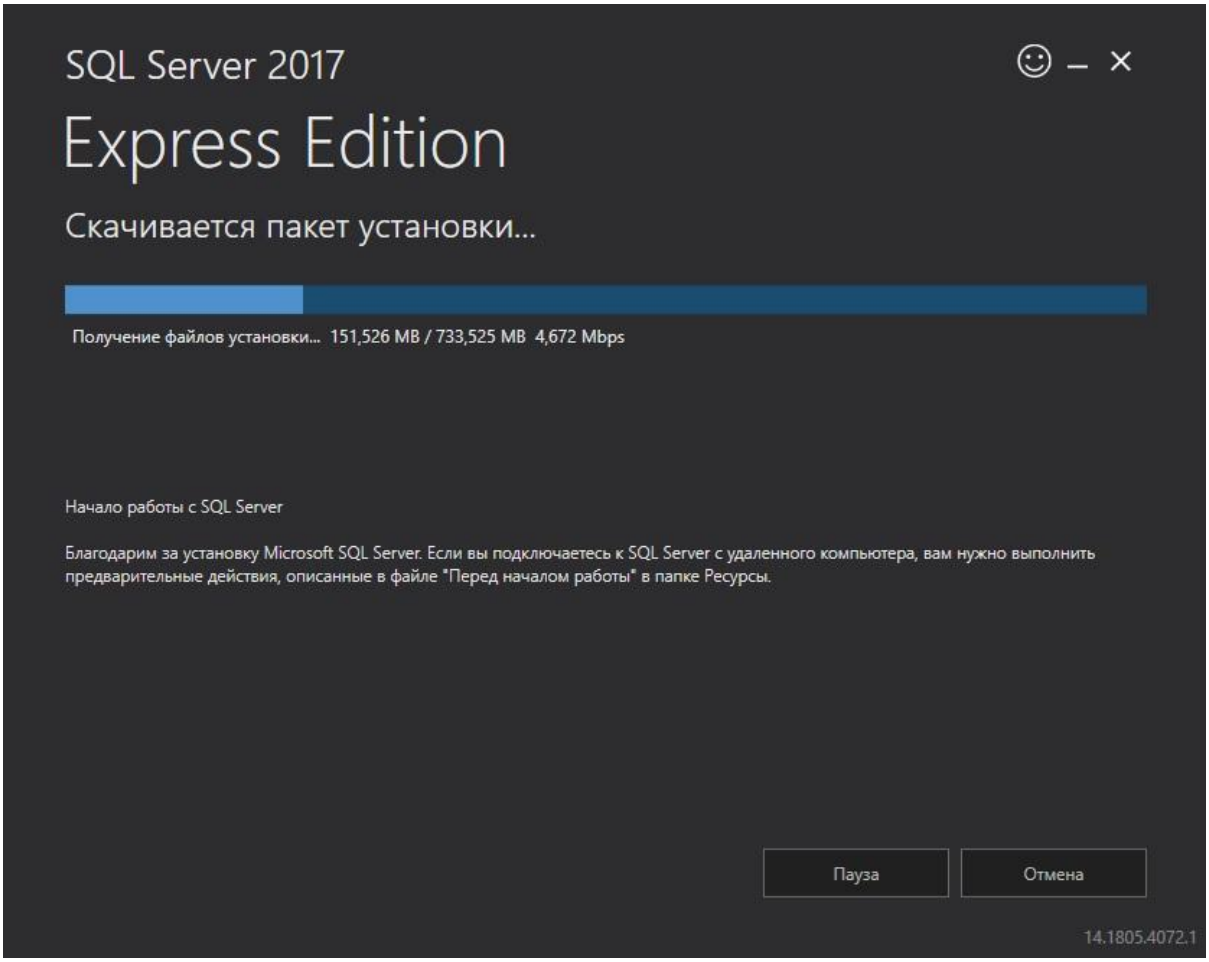


Рис. 4

Шаг 4 – Запуск программ установки SQL Server

Когда загрузка будет завершена, произойдет извлечение файлов установки, и запустится программа установки SQL Server. В нашем случае для новой установки Microsoft SQL Server выбираем первый пункт «*Новая установка изолированного экземпляра SQL Server или добавление компонентов к существующей установке*» (Рис. 5).

Шаг 5 – Принятие условий лицензионного соглашения

Сначала, для того чтобы установить и пользоваться Microsoft SQL Server, нам необходимо принять условия лицензионного соглашения. Для этого прочитайте их, и отметьте галочкой пункт «*Я принимаю условия лицензионного соглашения*» (Рис. 6).

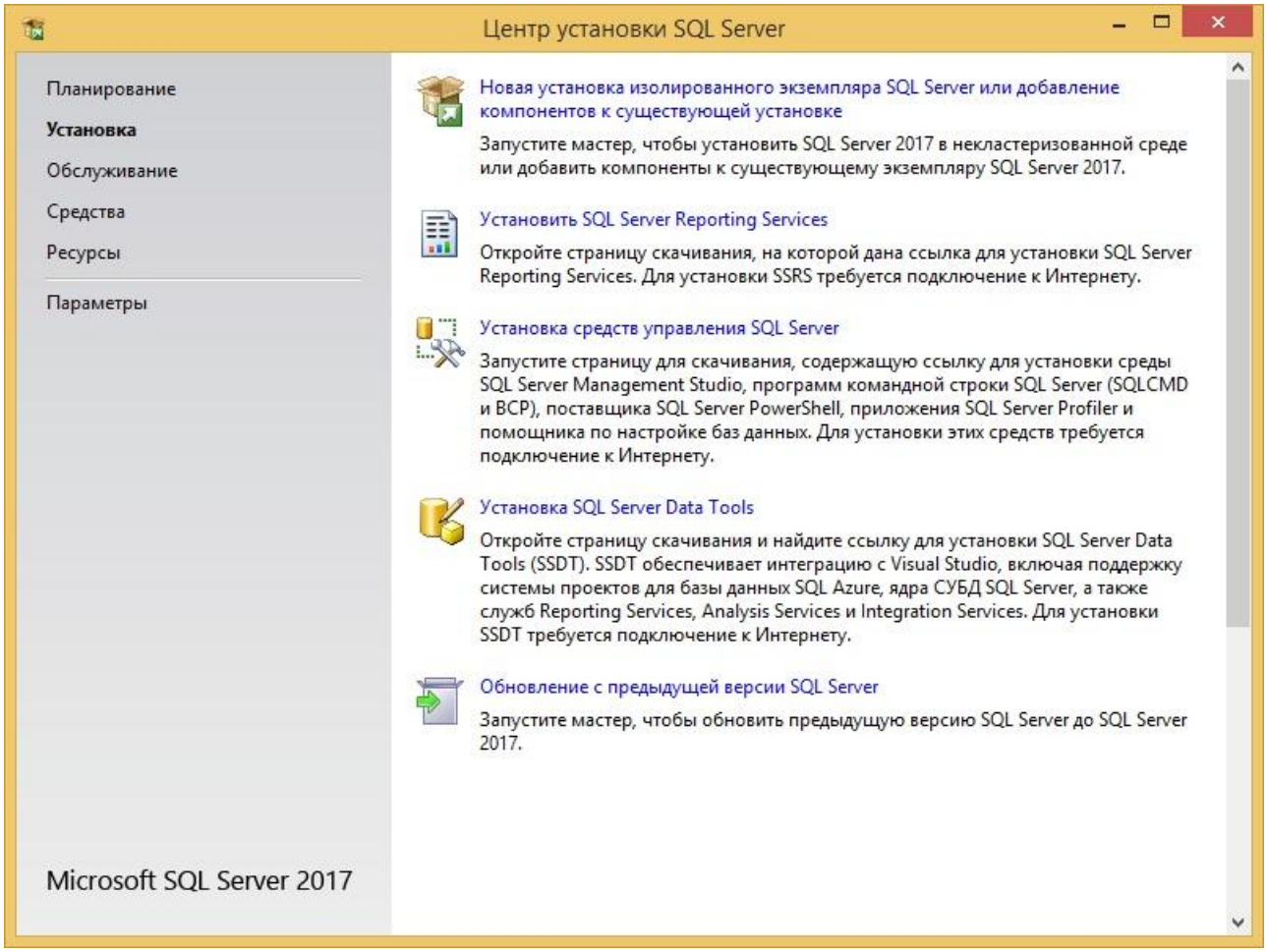

Центр установки SQL Server
Планирование
Установка
Обслуживание
Средства
Ресурсы
Параметры
Новая установка изолированного экземпляра SQL Server или добавление компонентов к существующей установке
Запустите мастер, чтобы установить SQL Server 2017 в некластеризованной среде или добавить компоненты к существующему экземпляру SQL Server 2017.
Установить SQL Server Reporting Services
Откройте страницу скачивания, на которой дана ссылка для установки SQL Server Reporting Services. Для установки SSRS требуется подключение к Интернету.
Установка средств управления SQL Server
Запустите страницу для скачивания, содержащую ссылку для установки среды SQL Server Management Studio, программ командной строки SQL Server (SQLCMD и BCP), поставщика SQL Server PowerShell, приложения SQL Server Profiler и помощника по настройке баз данных. Для установки этих средств требуется подключение к Интернету.
Установка SQL Server Data Tools
Откройте страницу скачивания и найдите ссылку для установки SQL Server Data Tools (SSDT). SSDT обеспечивает интеграцию с Visual Studio, включая поддержку системы проектов для базы данных SQL Azure, ядра СУБД SQL Server, а также служб Reporting Services, Analysis Services и Integration Services. Для установки SSDT требуется подключение к Интернету.
Обновление с предыдущей версии SQL Server
Запустите мастер, чтобы обновить предыдущую версию SQL Server до SQL Server 2017.
Microsoft SQL Server 2017

Рис. 5

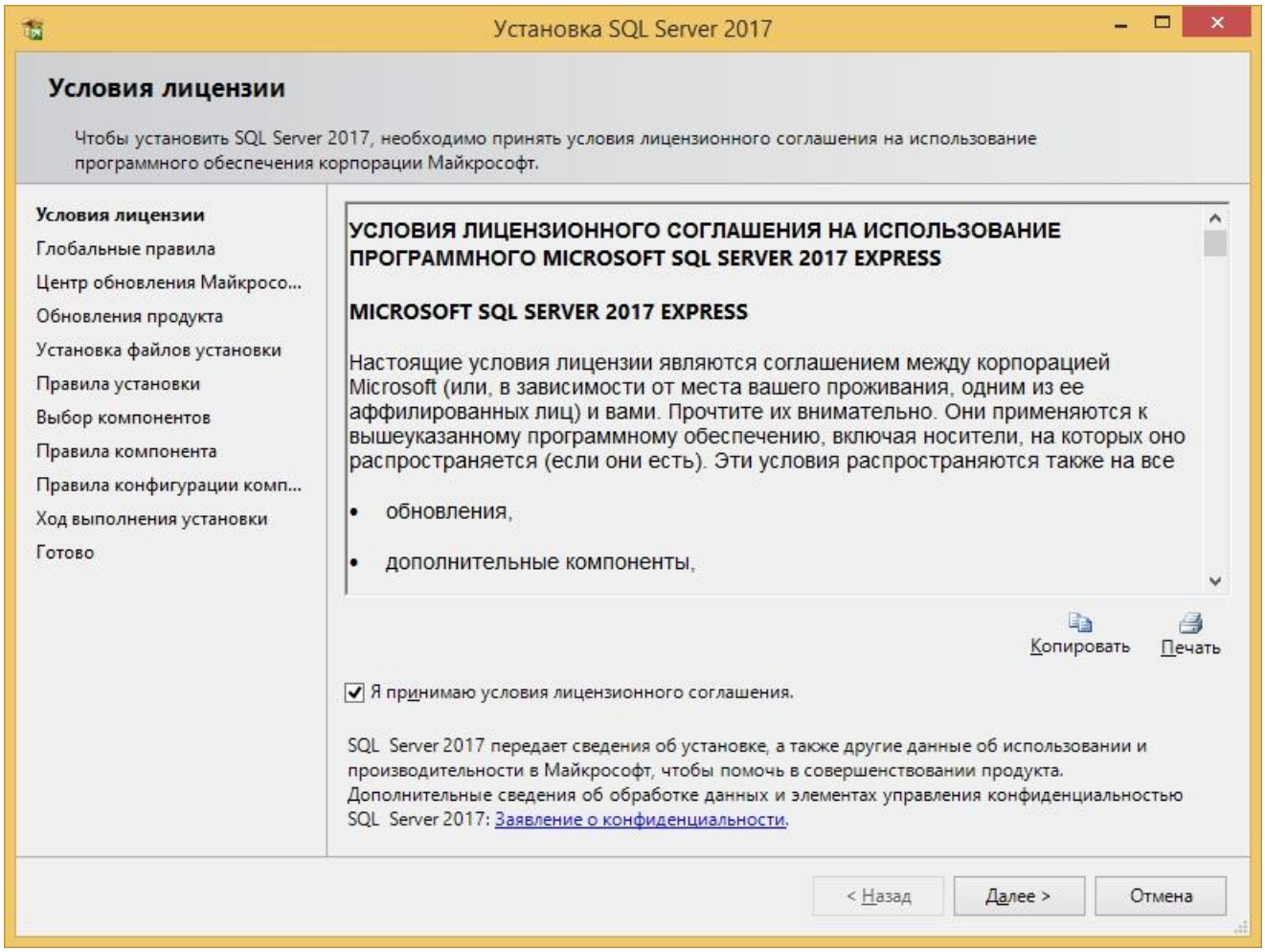

Установка SQL Server 2017
Условия лицензии
Чтобы установить SQL Server 2017, необходимо принять условия лицензионного соглашения на использование программного обеспечения корпорации Майкрософт.
Условия лицензии
Глобальные правила
Центр обновления Майкросо...
Обновления продукта
Установка файлов установки
Правила установки
Выбор компонентов
Правила компонента
Правила конфигурации комп...
Ход выполнения установки
Готово
УСЛОВИЯ ЛИЦЕНЗИОННОГО СОГЛАШЕНИЯ НА ИСПОЛЬЗОВАНИЕ ПРОГРАММНОГО MICROSOFT SQL SERVER 2017 EXPRESS
MICROSOFT SQL SERVER 2017 EXPRESS
Настоящие условия лицензии являются соглашением между корпорацией Microsoft (или, в зависимости от места вашего проживания, одним из ее аффилированных лиц) и вами. Прочтите их внимательно. Они применяются к вышеуказанному программному обеспечению, включая носители, на которых оно распространяется (если они есть). Эти условия распространяются также на все
• обновления,
• дополнительные компоненты,
Копировать
Печать
Я принимаю условия лицензионного соглашения.
SQL Server 2017 передает сведения об установке, а также другие данные об использовании и производительности в Майкрософт, чтобы помочь в совершенствовании продукта. Дополнительные сведения об обработке данных и элементах управления конфиденциальностью SQL Server 2017: Заявление о конфиденциальности.
< Назад
Далее >
Отмена

Рис. 6

Шаг 6 - Глобальные правила

На данном шаге программа установки определит возможные глобальные проблемы, которые могут возникнуть в ходе установки, это так называемые «*Глобальные правила*» (Рис. 7). Например, если Ваша система не соблюдает какое-нибудь из представленных правил, во время установки могут возникнуть проблемы, поэтому рекомендуется все ошибки и предупреждения устранить перед продолжением. Программа автоматически, если все хорошо, перейдет к следующему шагу, в остальных случаях устраняем ошибки и нажимаем «*Далее*».

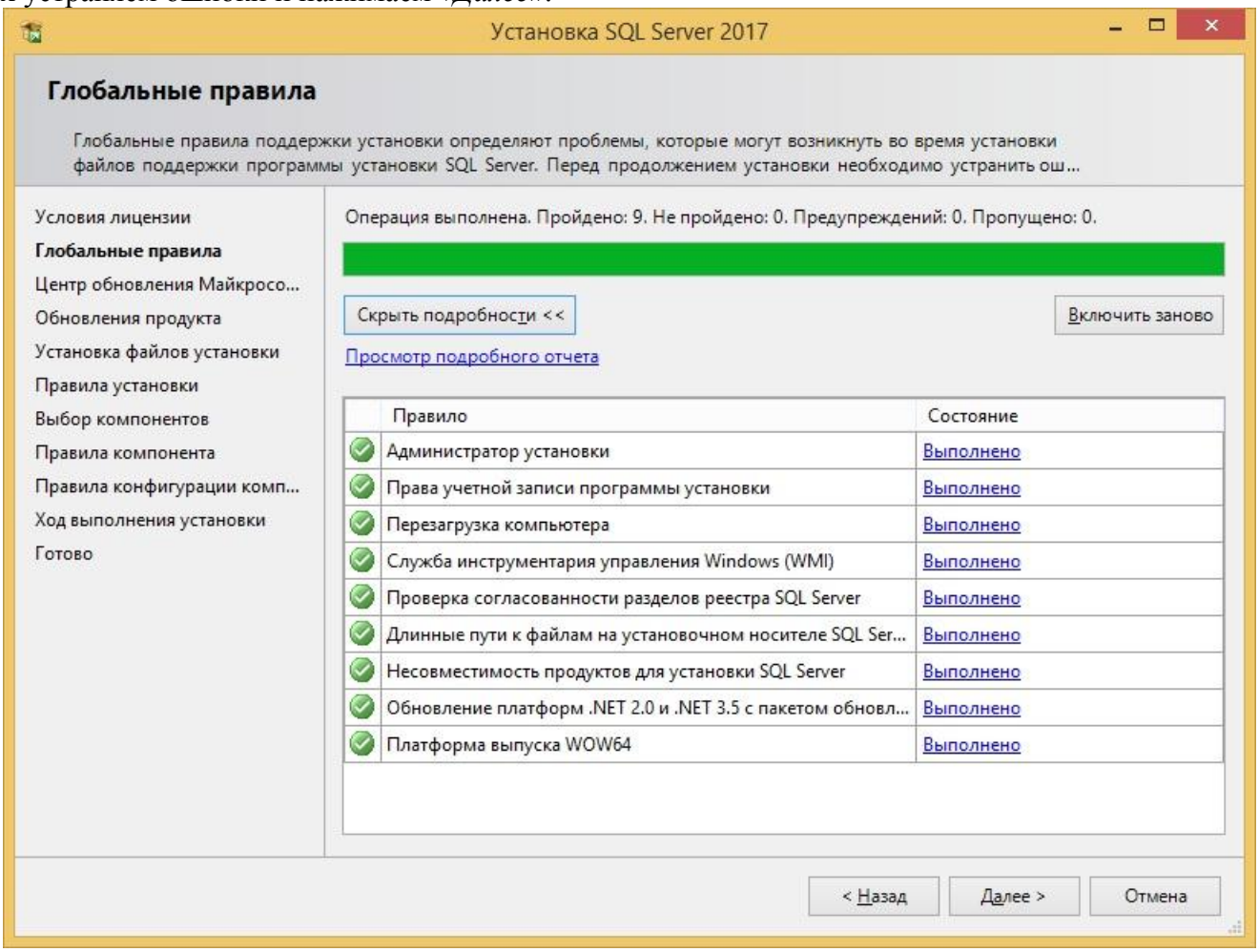


Рис. 7

Шаг 7 – Обновление SQL Server

Здесь Вы можете поставить галочку, если хотите, чтобы Microsoft SQL Server обновлялся, используя «*Центр обновления Microsoft*» операционной системы Windows (Рис. 8). Только следует помнить о том, что «*Центр обновления Windows*» должен быть включен. Нажимаем «*Далее*». А далее программа проверит наличие обновлений в Интернете, после чего мы снова нажимаем «*Далее*» (Рис. 9).

Шаг 8 – Установка файлов установки и правила установки

Программа установки установит необходимые для ее работы файлы, а также определит потенциальные проблемы, которые могут возникнуть при работе программы установки (Рис. 10).

Если никаких проблем нет, то программа установки автоматически перейдет к следующему шагу, в противном случае все ошибки необходимо устранить, а предупреждения проанализировать, так как, например, включенный брандмауэр Windows может затруднить доступ к SQL серверу, поэтому программа установки предупредит Вас о том, что необходимо добавить в брандмауэр соответствующие правила для SQL Server.

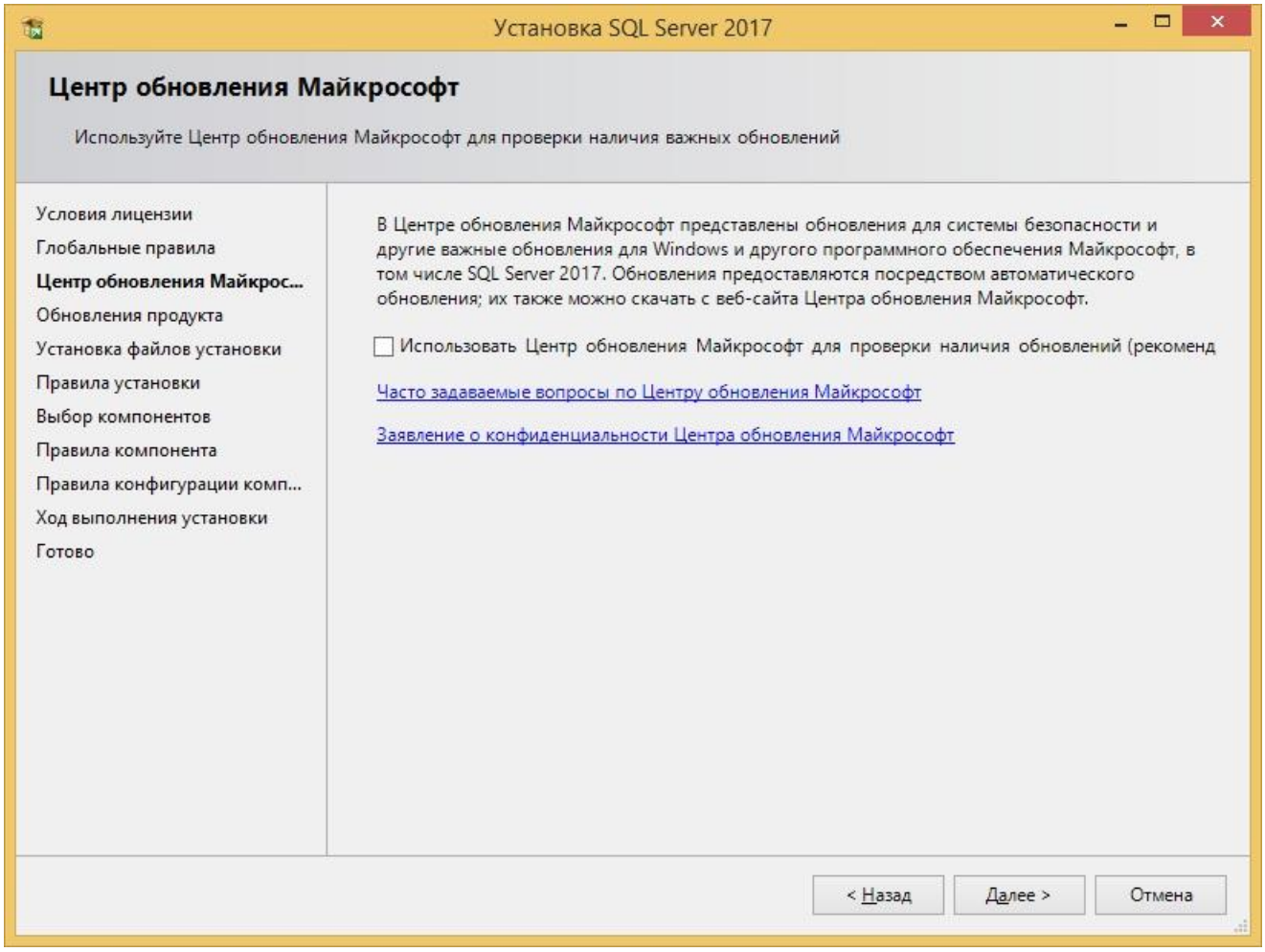

Установка SQL Server 2017
Центр обновления Майкрософт
Используйте Центр обновления Майкрософт для проверки наличия важных обновлений
Условия лицензии
Глобальные правила
Центр обновления Майкрос...
Обновления продукта
Установка файлов установки
Правила установки
Выбор компонентов
Правила компонента
Правила конфигурации комп...
Ход выполнения установки
Готово
В Центре обновления Майкрософт представлены обновления для системы безопасности и другие важные обновления для Windows и другого программного обеспечения Майкрософт, в том числе SQL Server 2017. Обновления предоставляются посредством автоматического обновления; их также можно скачать с веб-сайта Центра обновления Майкрософт.
Использовать Центр обновления Майкрософт для проверки наличия обновлений (рекоменд
Часто задаваемые вопросы по Центру обновления Майкрософт
Заявление о конфиденциальности Центра обновления Майкрософт
< Назад
Далее >
Отмена

Рис. 8

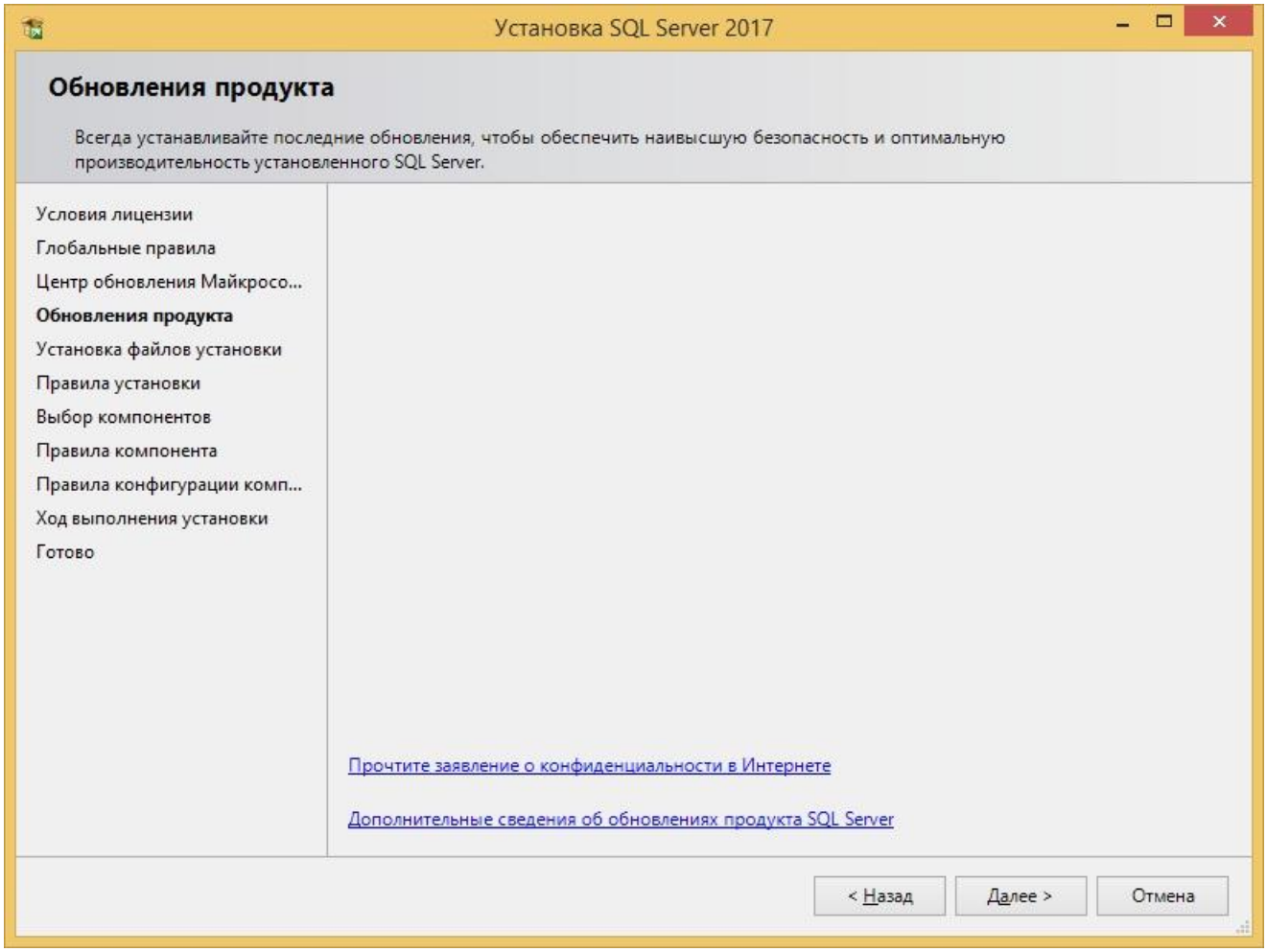

Установка SQL Server 2017
Обновления продукта
Всегда устанавливайте последние обновления, чтобы обеспечить наивысшую безопасность и оптимальную производительность установленного SQL Server.
Условия лицензии
Глобальные правила
Центр обновления Майкросо...
Обновления продукта
Установка файлов установки
Правила установки
Выбор компонентов
Правила компонента
Правила конфигурации комп...
Ход выполнения установки
Готово
Прочтите заявление о конфиденциальности в Интернете
Дополнительные сведения об обновлениях продукта SQL Server
< Назад
Далее >
Отмена

Рис. 9

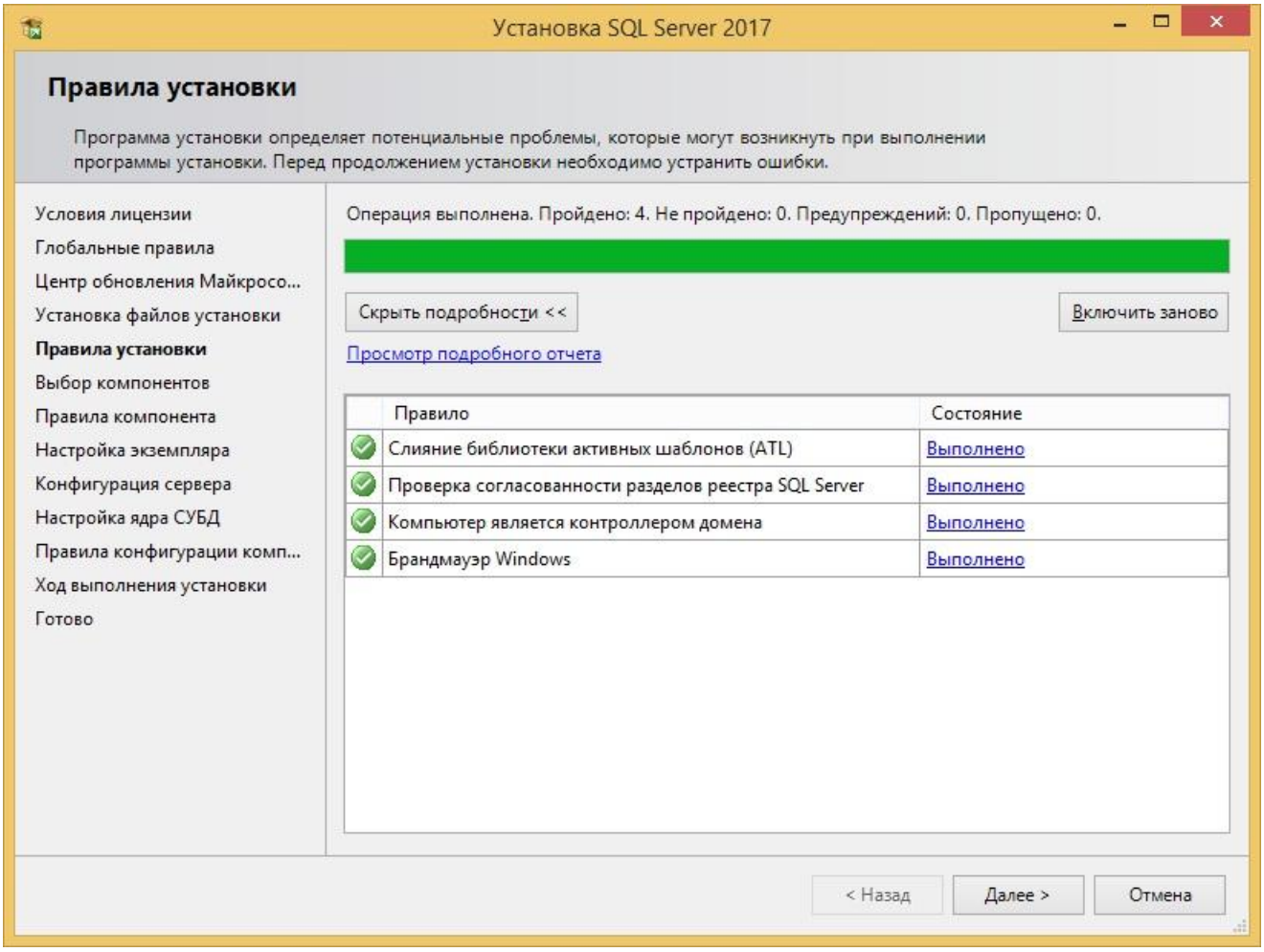


Рис. 10

Шаг 9 – Выбор компонентов SQL Server

Затем нам необходимо выбрать компоненты SQL Server, которые мы хотим установить на компьютер, по умолчанию отмечены практически все (Рис. 11). Для работы с данной книгой многие компоненты Вам не понадобятся, однако я Вам рекомендую оставить все как есть, и установить все компоненты, так как в будущем, может быть, Вам захочется подробней изучить и другие возможности SQL сервера.

Также, в случае необходимости, здесь Вы можете изменить системные каталоги для SQL Server.

Нажимаем «*Далее*».

После того как Вы нажмете далее, программа установки проверит наличие всех необходимых обновлений операционной системы, и если обязательные обновления не установлены в системе, то программа выдаст ошибку, а Вам, соответственно, для продолжения нужно установить соответствующие обновления.

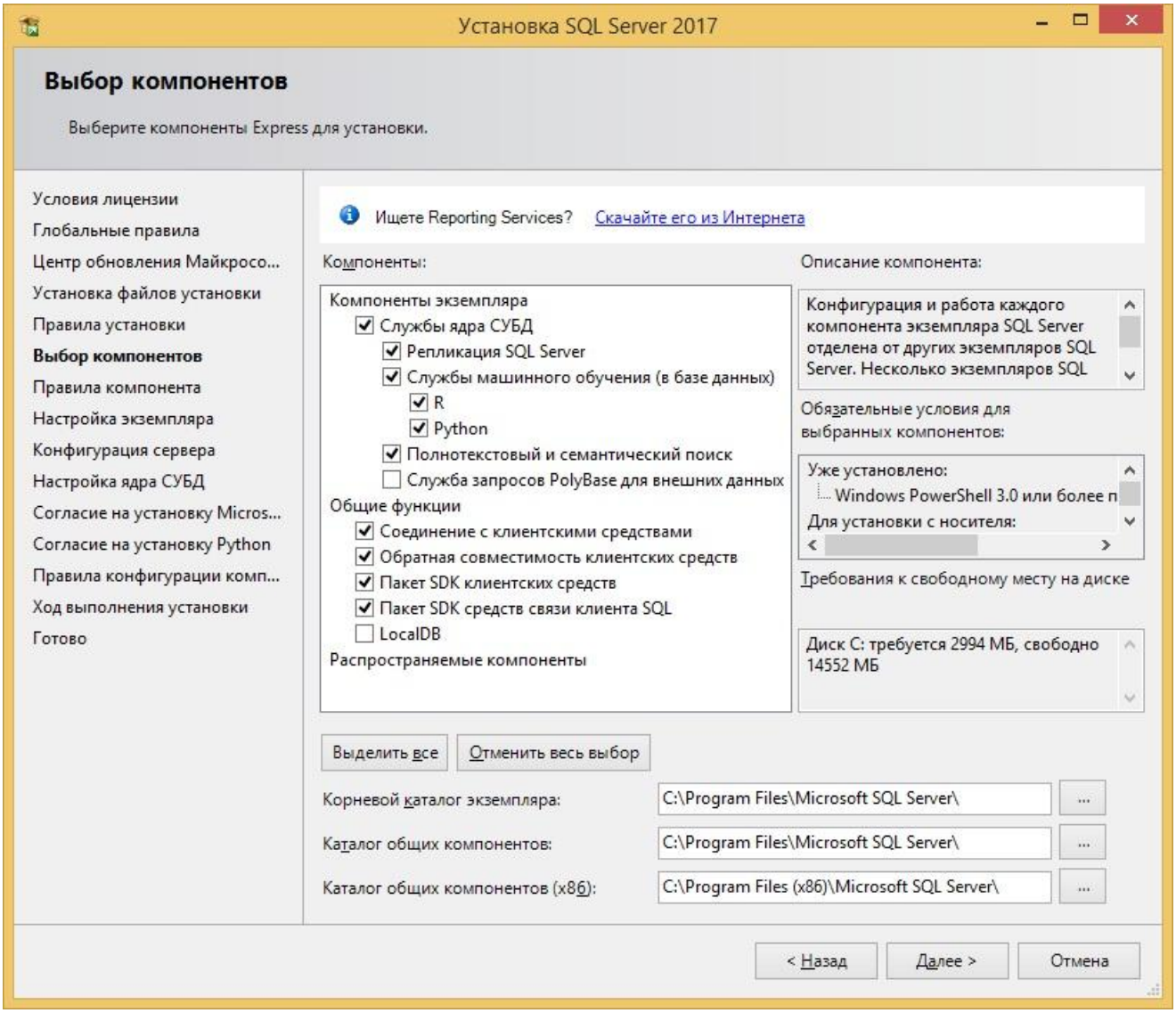


Рис. 11

Шаг 10 – Настройка экземпляра

На этом шаге Вы можете указать конкретное имя экземпляра SQL Server, которое будет использоваться при подключении к SQL серверу, однако если на Вашем компьютере еще нет установленного экземпляра SQL Server, Вы можете выбрать пункт «*Экземпляр по умолчанию*» (Рис. 12). Нажимаем «*Далее*».

Шаг 11 – Конфигурация сервера

Потом Вам необходимо указать от имени какой учетной записи будет работать служба SQL Server (Рис. 13). В нашем случае (*локальная установка для обучения*) мы можем выбрать пользователя, под которым работаем или оставить по умолчанию.

Нажимаем «*Далее*».

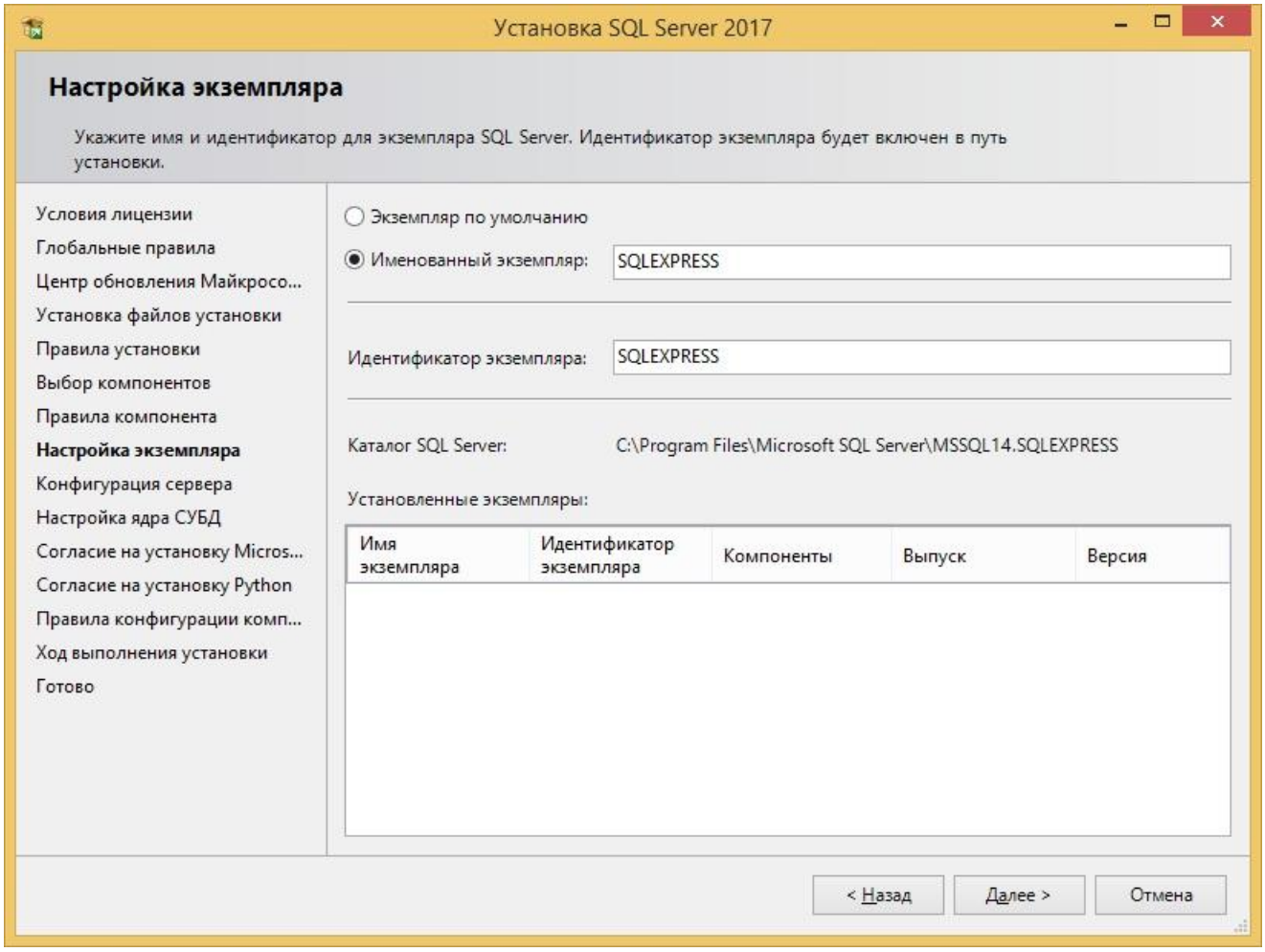

Установка SQL Server 2017
Настройка экземпляра
Укажите имя и идентификатор для экземпляра SQL Server. Идентификатор экземпляра будет включен в путь установки.
Условия лицензии
Глобальные правила
Центр обновления Майкросо...
Установка файлов установки
Правила установки
Выбор компонентов
Правила компонента
Настройка экземпляра
Конфигурация сервера
Настройка ядра СУБД
Согласие на установку Micros...
Согласие на установку Python
Правила конфигурации комп...
Ход выполнения установки
Готово
Экземпляр по умолчанию
Именованный экземпляр:
SQLEXPRESS
Идентификатор экземпляра:
SQLEXPRESS
Каталог SQL Server:
C:\Program Files\Microsoft SQL Server\MSSQL14.SQLEXPRESS
Установленные экземпляры:
Имя экземпляра
Идентификатор экземпляра
Компоненты
Выпуск
Версия
< Назад
Далее >
Отмена

Рис. 12

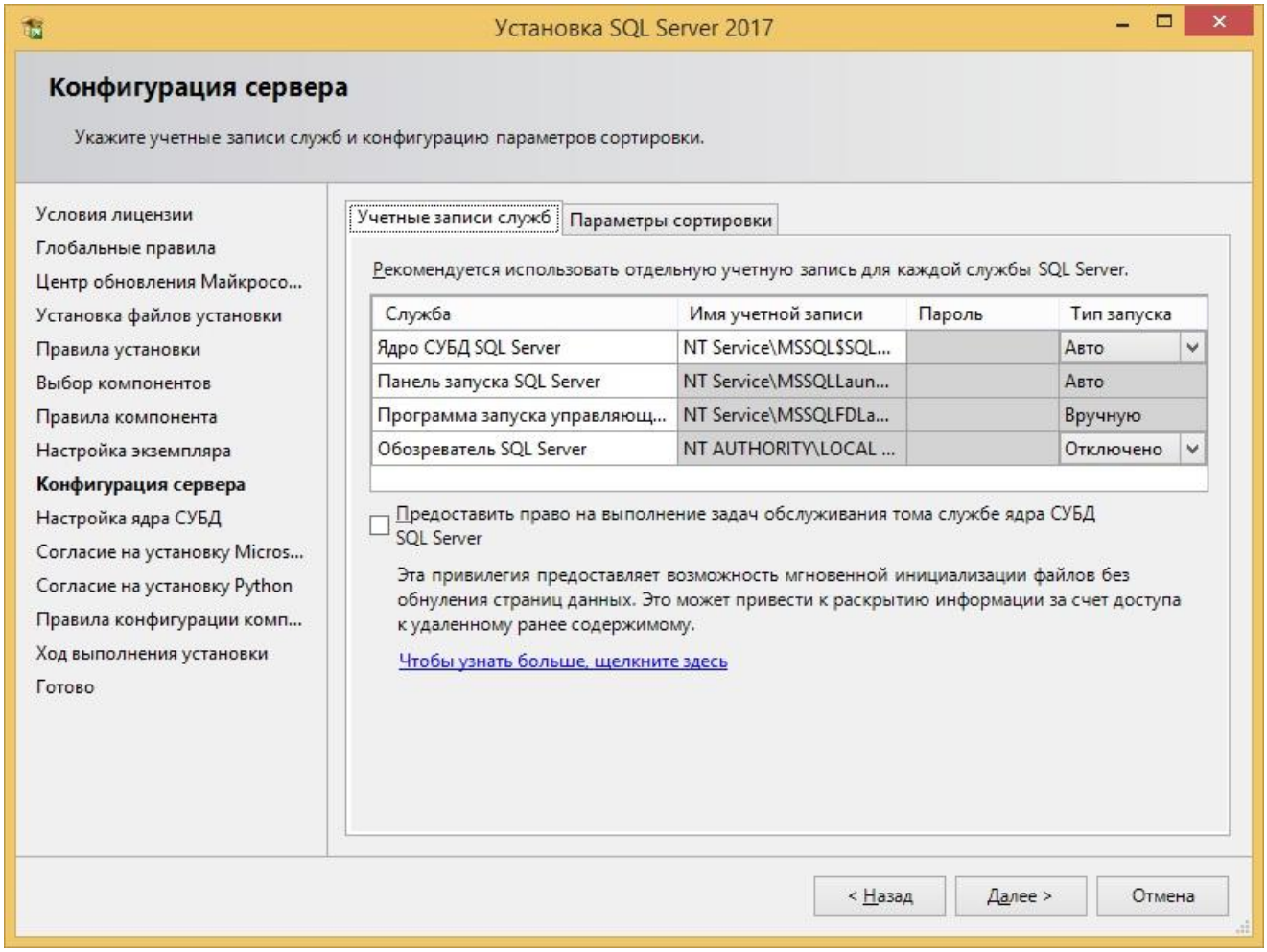

Установка SQL Server 2017
Конфигурация сервера
Укажите учетные записи служб и конфигурацию параметров сортировки.
Условия лицензии
Глобальные правила
Центр обновления Майкросо...
Установка файлов установки
Правила установки
Выбор компонентов
Правила компонента
Настройка экземпляра
Конфигурация сервера
Настройка ядра СУБД
Согласие на установку Micros...
Согласие на установку Python
Правила конфигурации комп...
Ход выполнения установки
Готово
Учетные записи служб
Параметры сортировки
Рекомендуется использовать отдельную учетную запись для каждой службы SQL Server.
Служба
Имя учетной записи
Пароль
Тип запуска
Ядро СУБД SQL Server
NT Service\MSSQL$SQL...
Авто
Панель запуска SQL Server
NT Service\MSSQLLaun...
Авто
Программа запуска управляющ...
NT Service\MSSQLFDLa...
Вручную
Обозреватель SQL Server
NT AUTHORITY\LOCAL ...
Отключено
Предоставить право на выполнение задач обслуживания тома службе ядра СУБД SQL Server
Эта привилегия предоставляет возможность мгновенной инициализации файлов без обнуления страниц данных. Это может привести к раскрытию информации за счет доступа к удаленному ранее содержимому.
Чтобы узнать больше, щелкните здесь
< Назад
Далее >
Отмена

Рис. 13

Шаг 12 – Настройка ядра СУБД

В следующем окне мы можем задать: режим проверки подлинности, администраторов SQL сервера, изменить каталоги данных и внести дополнительные параметры.

Режим проверки подлинности Windows - предполагает интеграцию с учетными записями Windows, т.е. пользователь, который аутентифицировался в Windows, сможет подключиться к SQL серверу.

Смешенный режим – позволяет использовать и проверку подлинности Windows, и встроенную проверку SQL сервера, при которой администратор SQL Server сам создает учетные записи непосредственно в SQL сервере.

В нашем случае мы выбираем «*Режим проверки подлинности Windows*» (Рис. 14).

По умолчанию в администраторы будет добавлен пользователь, от имени которого Вы производите установку SQL Server, но Вы можете добавить и дополнительных администраторов.

Также здесь мы можем изменить каталоги, в которых будут храниться наши пользовательские базы данных, давайте для примера изменим эти каталоги (*в нашем случае для тестовой установки это делать, конечно же, необязательно, но «боевые» базы данных всегда должны храниться в отдельном месте на отдельном хранилище*). Для этого переходим на вкладку «*Каталоги данных*», и указываем пути к новым каталогам для хранения пользовательских баз данных (Рис. 15). Я, для примера, указал для базы данных каталог DataBase на диске D, а для резервных копий каталог BACKUP_DB.

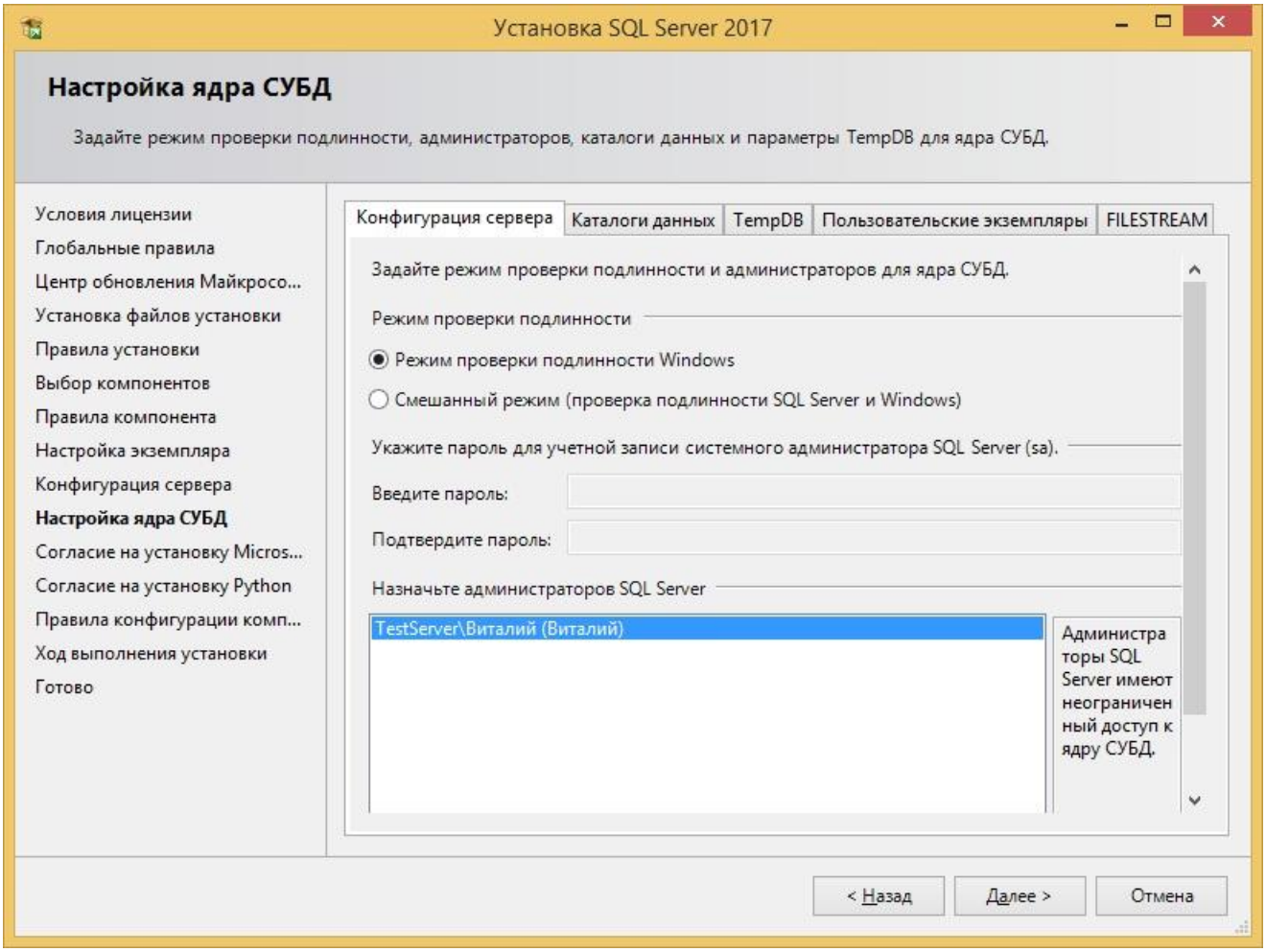


Рис. 14

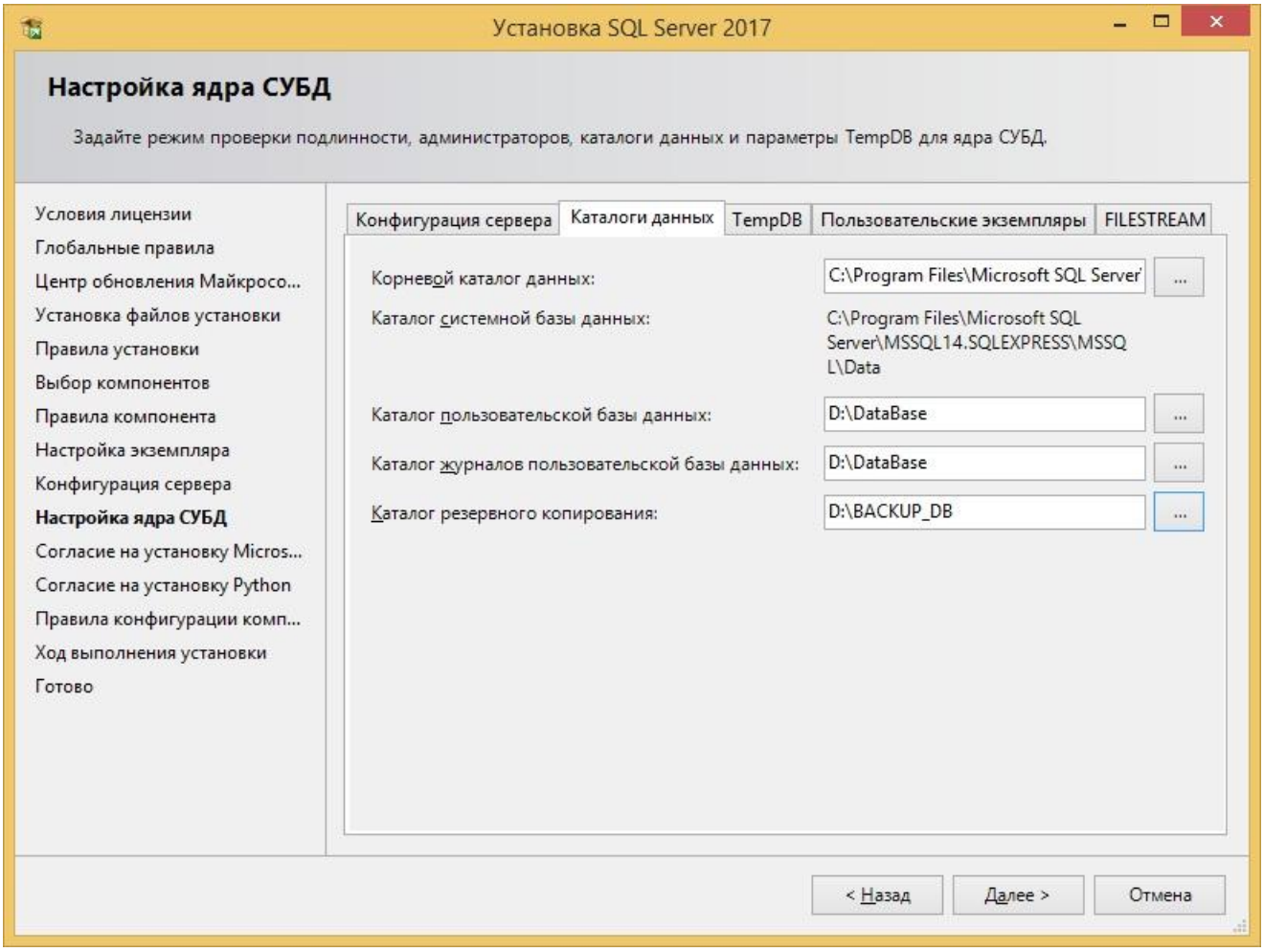


Рис. 15

Шаг 13 – Согласие на установку Microsoft R Open и Python

О том, что это за компоненты, в этой книге я Вам еще расскажу, а пока для продолжения необходимо нажать на кнопку «*Принять*», потом кнопка «*Далее*» станет активной, мы ее и нажимаем (Рис. 16).

Сначала мы согласились на установку Microsoft R Open, после этого нам точно также нужно согласиться на установку Python (Рис. 17). Нажимаем «*Принять*», затем «*Далее*».

Шаг 14 – Начало установки

Если правила конфигурации не определили никаких проблем, то сразу начнется процесс установки SQL Server (Рис. 18).

Шаг 15 – Завершение установки и перезагрузка компьютера

Установка будет завершена, когда программа выдаст соответствующее сообщение (Рис. 19), нажимаем «ОК», закрываем окно программы установки, и перезагружаем компьютер. После перезагрузки процесс установки SQL Server будет окончательно завершен.

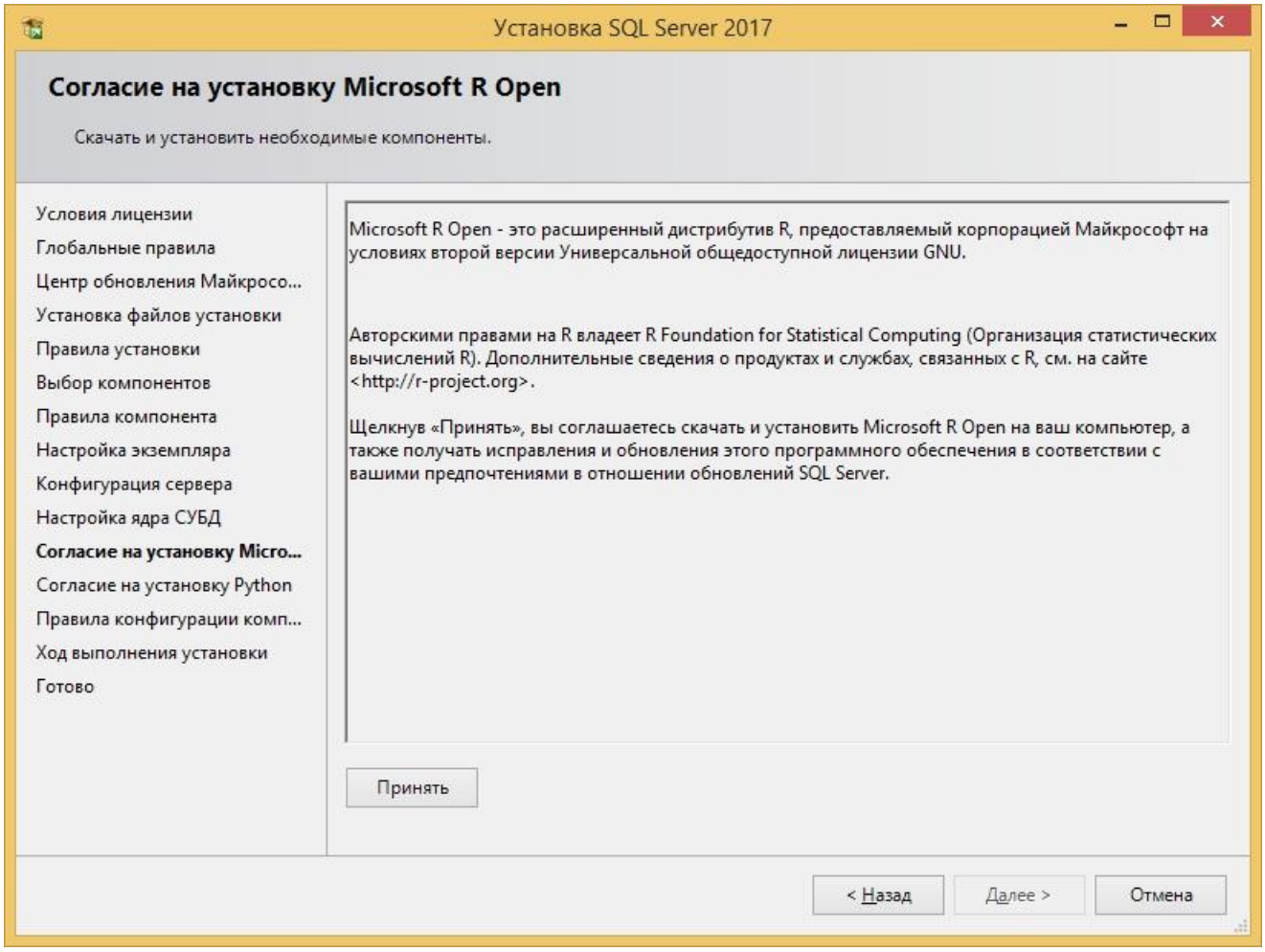

Установка SQL Server 2017
Согласие на установку Microsoft R Open
Скачать и установить необходимые компоненты.
Условия лицензии
Глобальные правила
Центр обновления Майкросо...
Установка файлов установки
Правила установки
Выбор компонентов
Правила компонента
Настройка экземпляра
Конфигурация сервера
Настройка ядра СУБД
Согласие на установку Micro...
Согласие на установку Python
Правила конфигурации комп...
Ход выполнения установки
Готово
Microsoft R Open - это расширенный дистрибутив R, предоставляемый корпорацией Майкрософт на условиях второй версии Универсальной общедоступной лицензии GNU.
Авторскими правами на R владеет R Foundation for Statistical Computing (Организация статистических вычислений R). Дополнительные сведения о продуктах и службах, связанных с R, см. на сайте <http://r-project.org>.
Щелкнув «Принять», вы соглашаетесь скачать и установить Microsoft R Open на ваш компьютер, а также получать исправления и обновления этого программного обеспечения в соответствии с вашими предпочтениями в отношении обновлений SQL Server.
Принять
< Назад
Далее >
Отмена

Рис. 16

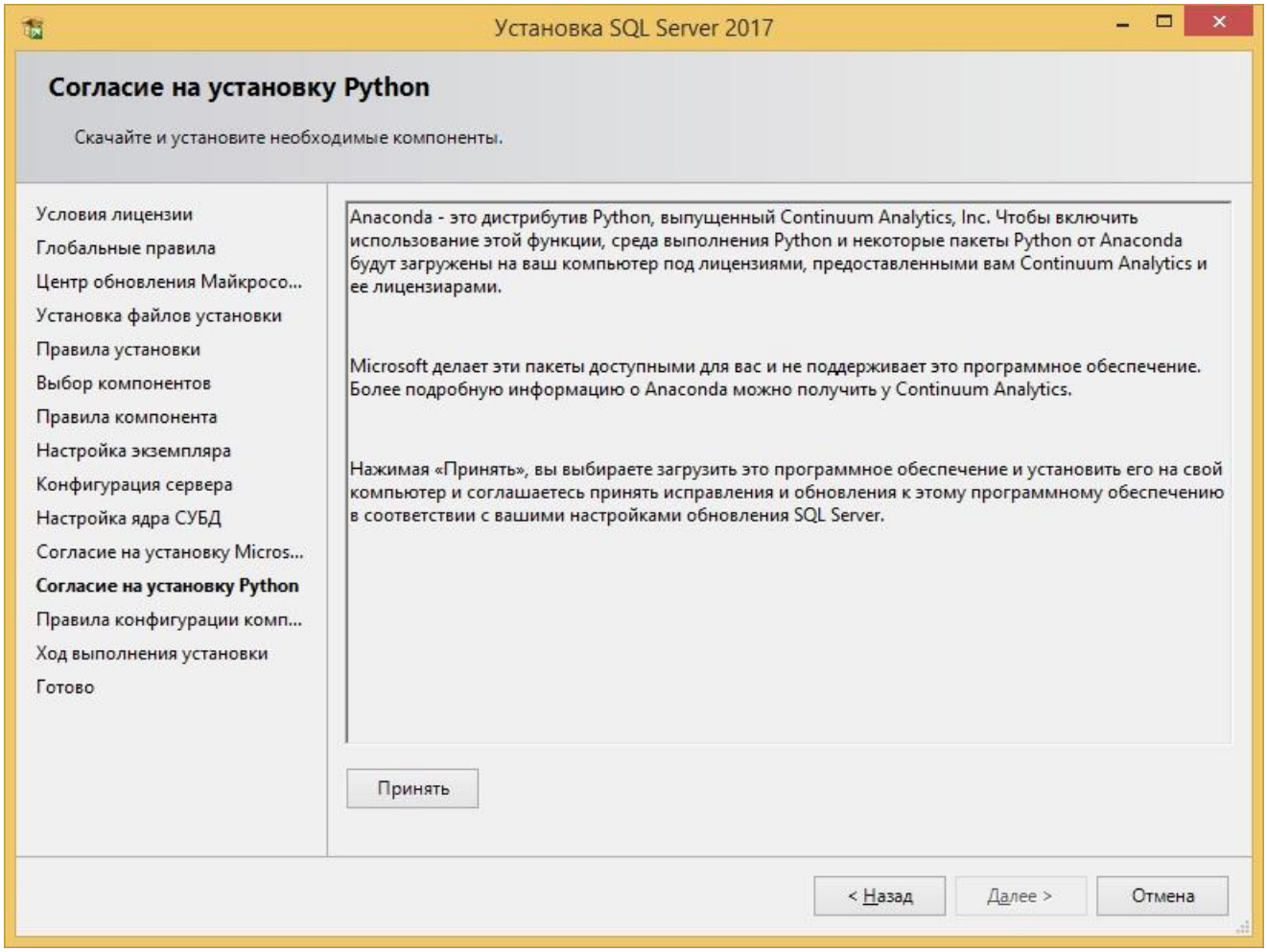

Установка SQL Server 2017
Согласие на установку Python
Скачайте и установите необходимые компоненты.
Условия лицензии
Глобальные правила
Центр обновления Майкросо...
Установка файлов установки
Правила установки
Выбор компонентов
Правила компонента
Настройка экземпляра
Конфигурация сервера
Настройка ядра СУБД
Согласие на установку Micros...
Согласие на установку Python
Правила конфигурации комп...
Ход выполнения установки
Готово
Anaconda - это дистрибутив Python, выпущенный Continuum Analytics, Inc. Чтобы включить использование этой функции, среда выполнения Python и некоторые пакеты Python от Anaconda будут загружены на ваш компьютер под лицензиями, предоставленными вам Continuum Analytics и ее лицензиарами.
Microsoft делает эти пакеты доступными для вас и не поддерживает это программное обеспечение. Более подробную информацию о Anaconda можно получить у Continuum Analytics.
Нажимая «Принять», вы выбираете загрузить это программное обеспечение и установить его на свой компьютер и соглашаетесь принять исправления и обновления к этому программному обеспечению в соответствии с вашими настройками обновления SQL Server.
Принять
< Назад
Далее >
Отмена

Рис. 17

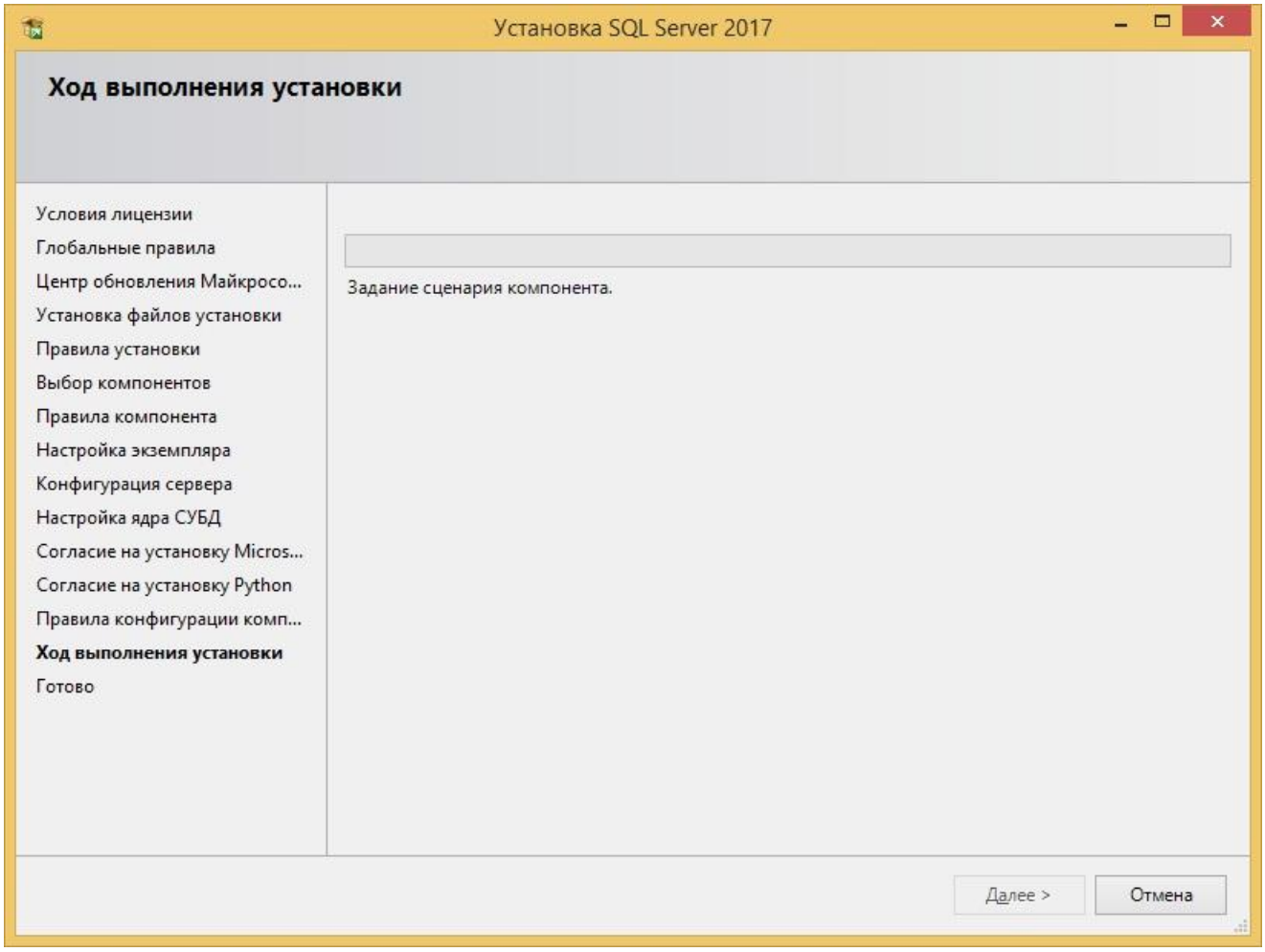

Установка SQL Server 2017
Ход выполнения установки
Условия лицензии
Глобальные правила
Центр обновления Майкросо...
Установка файлов установки
Правила установки
Выбор компонентов
Правила компонента
Настройка экземпляра
Конфигурация сервера
Настройка ядра СУБД
Согласие на установку Micros...
Согласие на установку Python
Правила конфигурации комп...
Ход выполнения установки
Готово
Задание сценария компонента.
Далее >
Отмена

Рис. 18

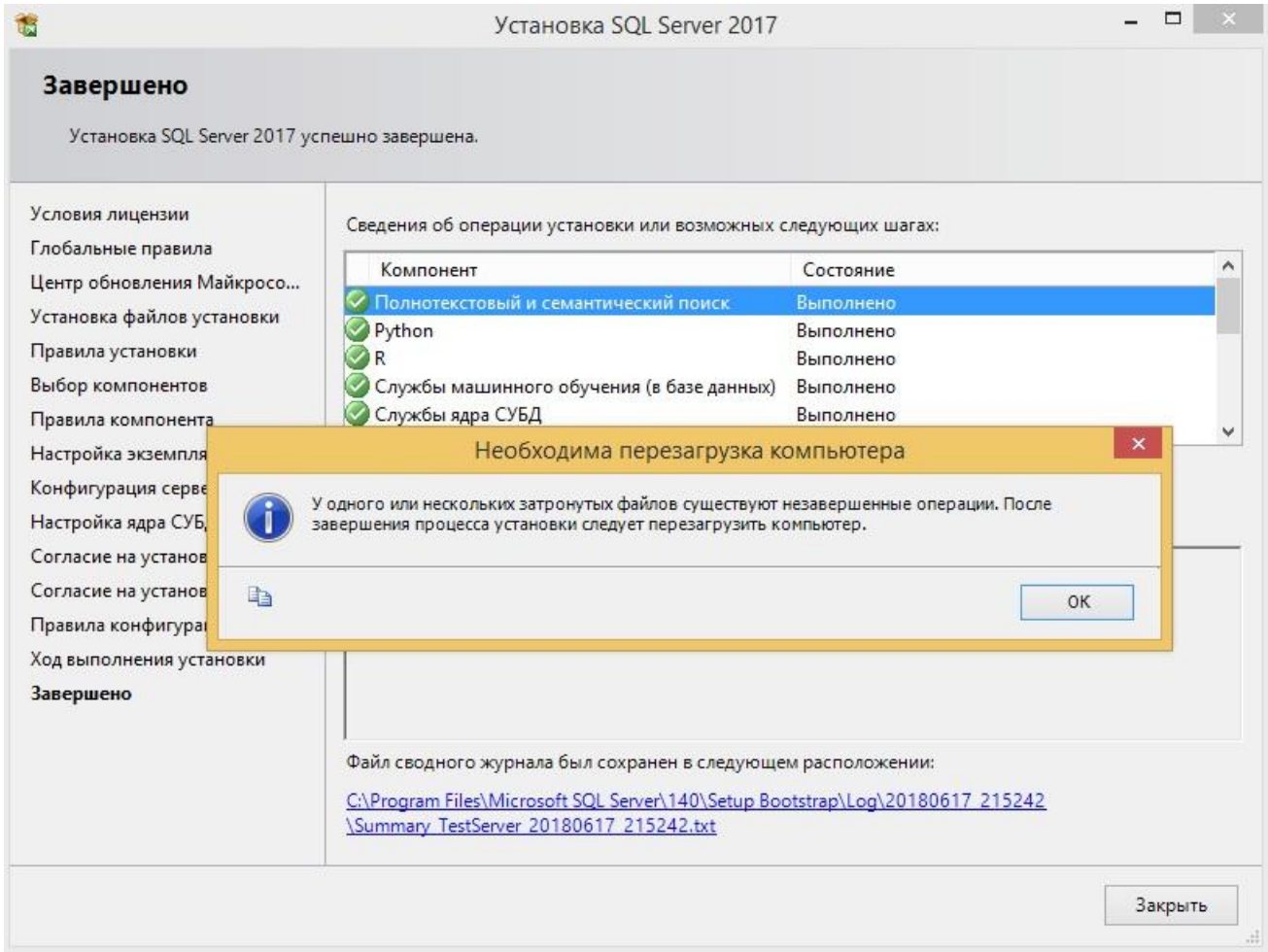

Установка SQL Server 2017
Завершено
Установка SQL Server 2017 успешно завершена.
Условия лицензии
Глобальные правила
Центр обновления Майкросо...
Установка файлов установки
Правила установки
Выбор компонентов
Правила компонента
Настройка ядра СУБД
Ход выполнения установки
Завершено
Сведения об операции установки или возможных следующих шагах:
Компонент
Состояние
Полнотекстовый и семантический поиск
Выполнено
Python
Выполнено
R
Выполнено
Службы машинного обучения (в базе данных)
Выполнено
Службы ядра СУБД
Выполнено
Необходима перезагрузка компьютера
У одного или нескольких затронутых файлов существуют незавершенные операции. После завершения процесса установки следует перезагрузить компьютер.
OK
Файл сводного журнала был сохранен в следующем расположении:
C:\Program Files\Microsoft SQL Server\140\Setup Bootstrap\Log\20180617_215242\Summary_TestServer_20180617_215242.txt
Закрыть

Рис. 19

Основной инструмент для работы с базой данных в Microsoft SQL Server

После того как Вы установили Microsoft SQL Server, нужно установить инструмент, с помощью которого можно будет подключиться к серверу баз данных, и соответственно управлять этими базами данных, например, создавать их, добавлять таблицы, писать SQL запросы, разрабатывать процедуры, функции, управлять пользователями и многое другое. Такой инструмент у Microsoft есть - это среда **SQL Server Management Studio** (SSMS).

Скачать актуальную версию SSMS (*она также бесплатная*) можно со страницы загрузки официального сайта – https://docs.microsoft.com/ru-ru/sql/ssms/download-sql-server-management-studio-ssms.

В принципе Вы также можете открыть приложение «Центр установки SQL Server» (*которое у Вас уже есть*), оно доступно из меню пуск, и выбрать там в разделе «*Установка*» пункт «*Установка средств управления SQL Server*» (Рис. 20), в результате запустится браузер, и Вы попадете на страницу, где Вы можете всегда скачать актуальную версию среды Management Studio, прямую ссылку на данную страницу я указал чуть выше (*на момент написания книги актуальная версия среды Management Studio 17.7*).

В итоге Вы должны загрузить установочный файл **SSMS-Setup-RUS.exe** размером около 838 мегабайт.

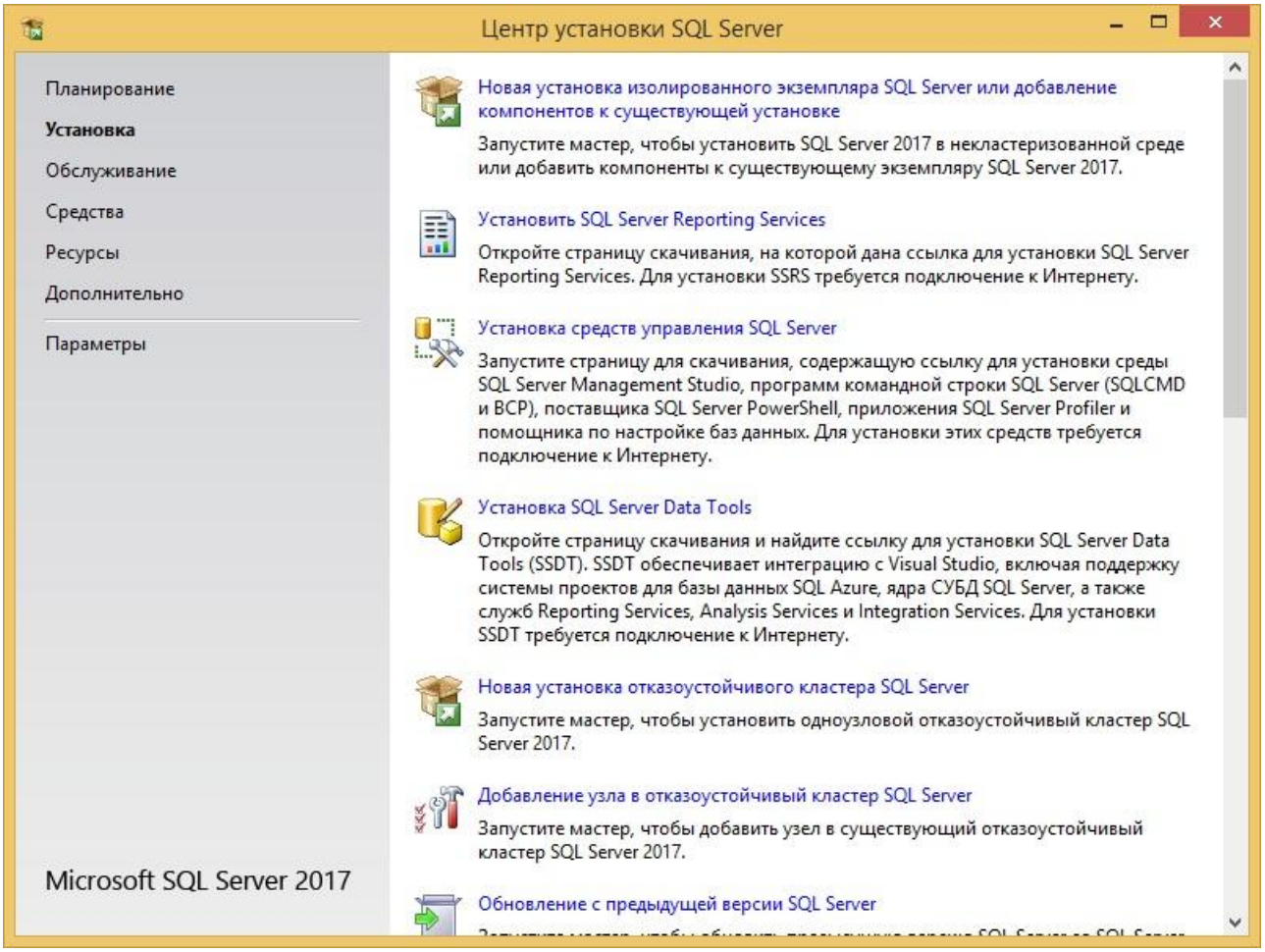


Рис. 20

Установка среды SQL Server Management Studio

Запускаем скаченный файл (*SSMS-Setup-RUS.exe*). Откроется окно программы установки (Рис. 21), в котором Вам необходимо нажать кнопку «*Установить*».

Затем сразу начнется установка среды SQL Server Management Studio и всех необходимых для ее работы компонентов. Для завершения процесса установки необходимо будет перезагрузить компьютер, программа установки выдаст Вам соответствующее сообщение (Рис. 22).

Больше Вам ничего не нужно делать, когда компьютер перезагрузится, уже можно сразу начинать пользоваться средой Management Studio.

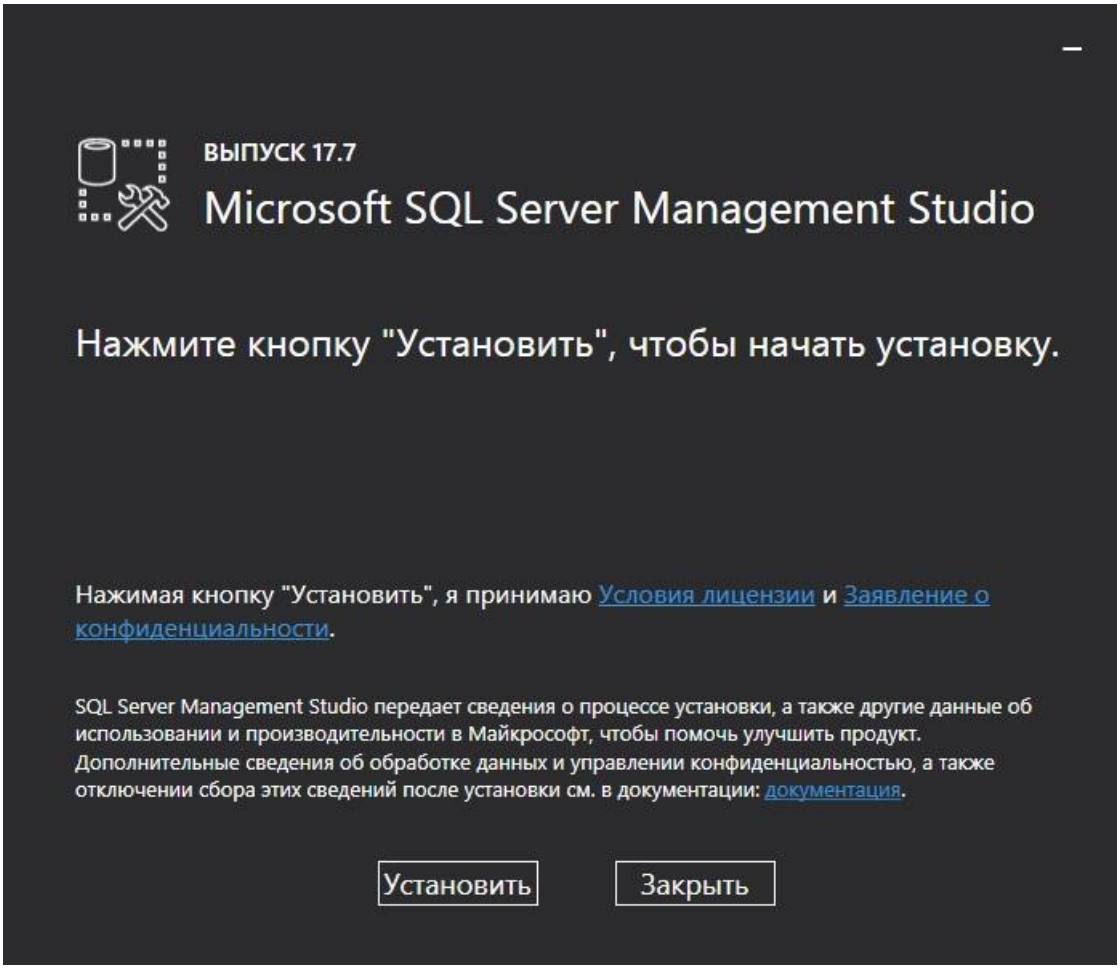


Рис. 21

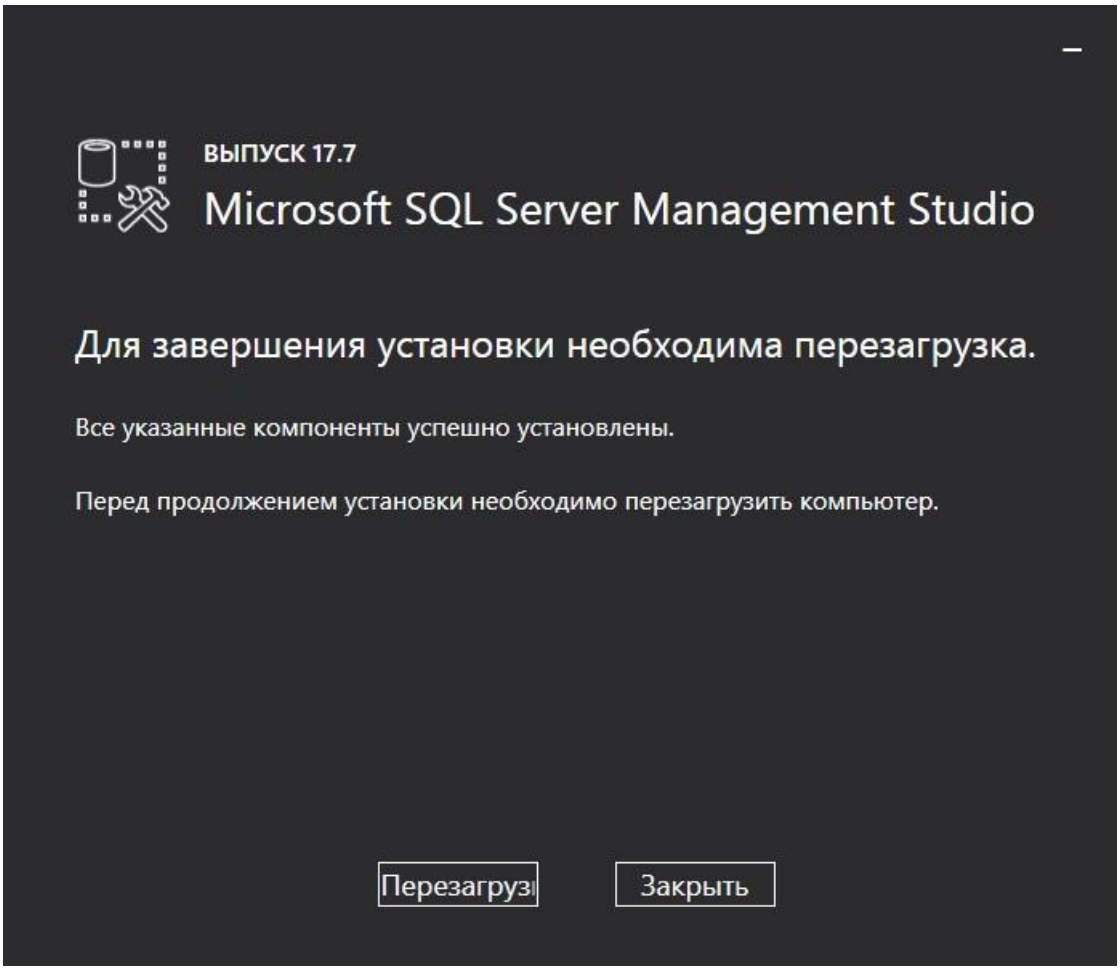


Рис. 22

Ну а мы двигаемся дальше, и переходим уже непосредственно к рассмотрению языка T-SQL.

Язык SQL и T-SQL

Допустим, у Вас уже есть база данных, но как Вам получить данные хранящиеся в ней? Для этого должен быть специальный инструмент, этим инструментом и является язык SQL.

SQL (*Structured Query Language*) - язык структурированных запросов, с помощью данного языка пишутся специальные запросы к базе данных с целью получения этих данных и манипулирования ими.

Язык SQL – это стандарт, он реализован во всех реляционных базах данных, но у каждой СУБД есть расширение этого стандарта, для того чтобы, например, полноценно программировать, получать системную информацию, упрощать SQL запросы. В Microsoft SQL Server таким расширением является язык **Transact-SQL**, сокращенно T-SQL.

Transact-SQL имеет практически все атрибуты полноценного языка программирования - это и переменные, и циклы, и условия, включая, конечно же, все возможности языка SQL с расширенным функционалом.

Если кто-нибудь Вам скажет, что T-SQL - это не язык программирования, можете смело возразить, т.к. с помощью него можно непосредственно в базе данных реализовать своего рода программы, некую логику действий, но об этом мы с Вами поговорим в соответствующих главах этой книги (*процедуры, функции и т.д.*)

Создание базы данных

Итак, у Вас к этому моменту на компьютере уже должен быть установлен Microsoft SQL Server в редакции Express и среда SQL Server Management Studio.

Чтобы создать базу данных в Microsoft SQL Server, запускаем SQL Server Management Studio, например, из меню Пуск.

Подключаемся к компоненту «*Ядро СУБД*», нажимаем кнопку «*Соединить*» (Рис. 23).

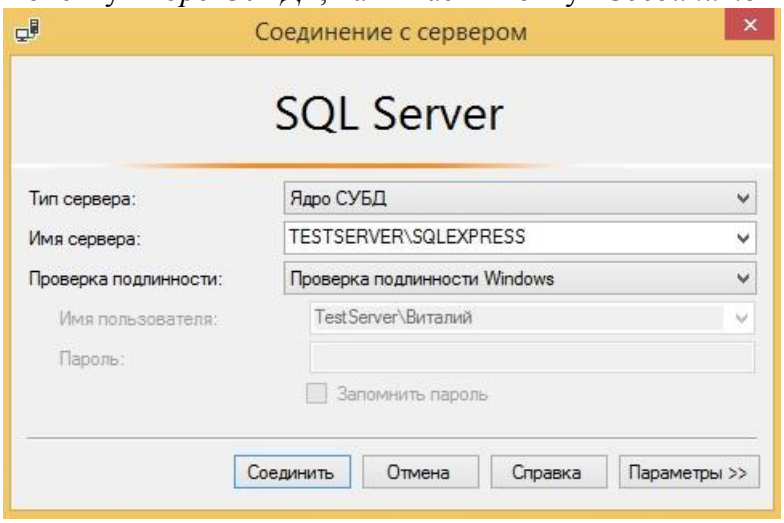


Рис. 23

В обозревателе объектов с левой стороны Вы видите все объекты, которые находятся на текущем сервере. А точнее там пока только системные объекты, ни одной пользовательской базы данных там нет. Поэтому давайте же начнем наше знакомство с Microsoft SQL Server с создания базы данных.

Я покажу Вам, как это можно сделать как с помощью SQL инструкции, так и с помощью графического инструмента SSMS.

Для создания базы данных с использованием SQL инструкции в среде SSMS нажимаем на кнопку «*Создать запрос*» (Рис. 24).

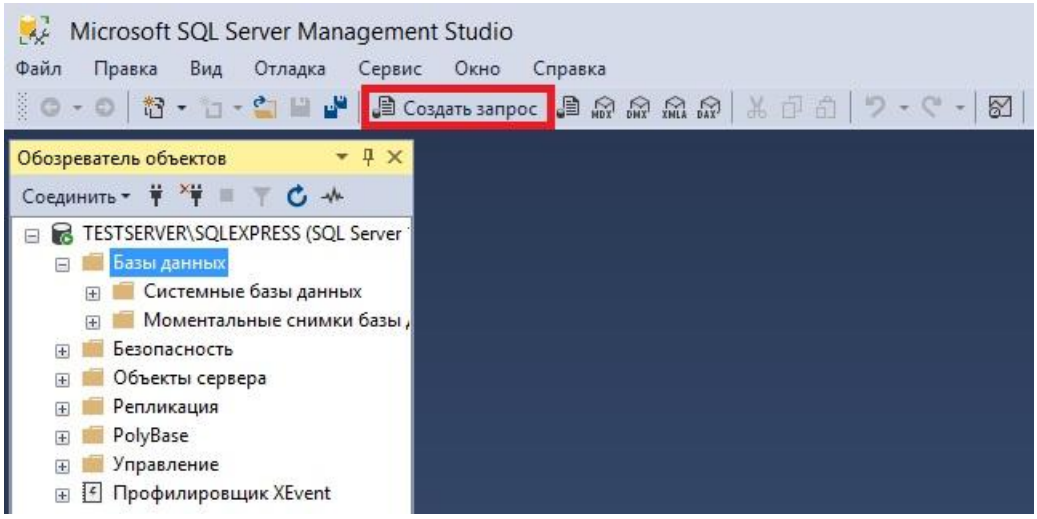


Рис. 24

И вводим следующую инструкцию.

```
CREATE DATABASE TestDB
```

Где, CREATE DATABASE означает, что мы хотим создать базу данных, а TestDB – это имя нашей будущей базы данных.

Нажимаем кнопку «*Выполнить*» (Рис. 25).

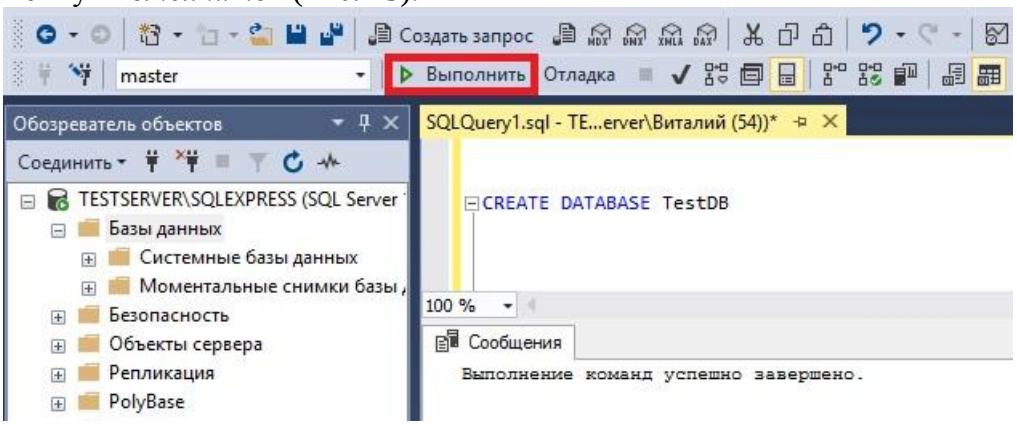


Рис. 25

Все, база данных создана!

Кстати, таким образом, мы будем выполнять все наши SQL запросы и инструкции, т.е. с помощью редактора SQL запросов в Management Studio.

То же самое можно было сделать с помощью графического инструмента, например, щелкнуть правой кнопкой мыши по объекту «*Базы данных*» и выбрать «*Создать базу данных*», затем ввести имя этой базы и нажать «ОК» (Рис. 26).

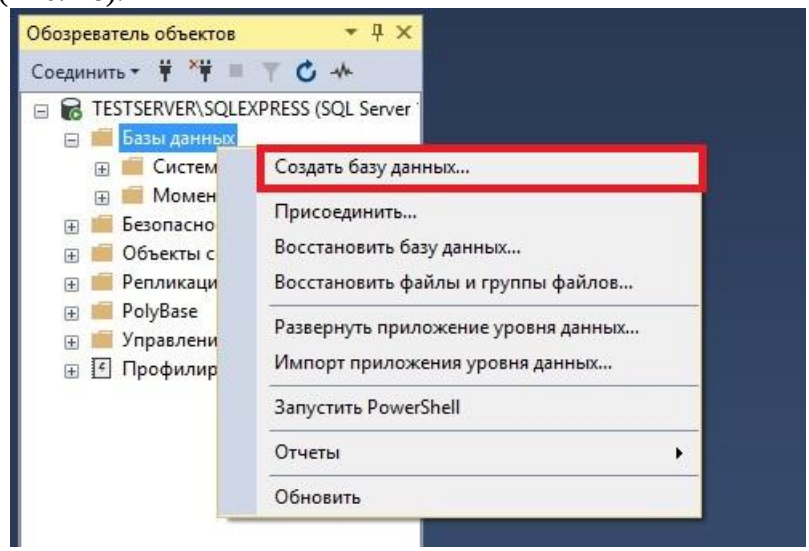


Рис. 26

В данном случае мы просто создали базу данных с параметрами по умолчанию, однако у этой инструкции много различных параметров, например, мы можем указать, где будут располагаться файлы базы данных, и как они будут называться и многое другое. Но сейчас нам пока этого не нужно, главное, мы научились создавать базы данных и пользоваться средой SQL Server Management Studio. Как я уже говорил, мы все будем делать постепенно.

Удаление базы данных

Если по каким-либо причинам Вам больше не нужна база данных, и Вы хотите ее удалить, то это можно сделать с помощью следующей инструкции

```
DROP DATABASE TestDB
```

Где, DROP DATABASE - это команда SQL для удаления базы данных.

Удалить базу можно также в Management Studio, например, выбрать нужную (*т.е. нужный объект в обозревателе объектов*), правой кнопкой нажать на него и выбрать «*Удалить*» (Рис. 27). В открывшемся окне нажать «ОК».

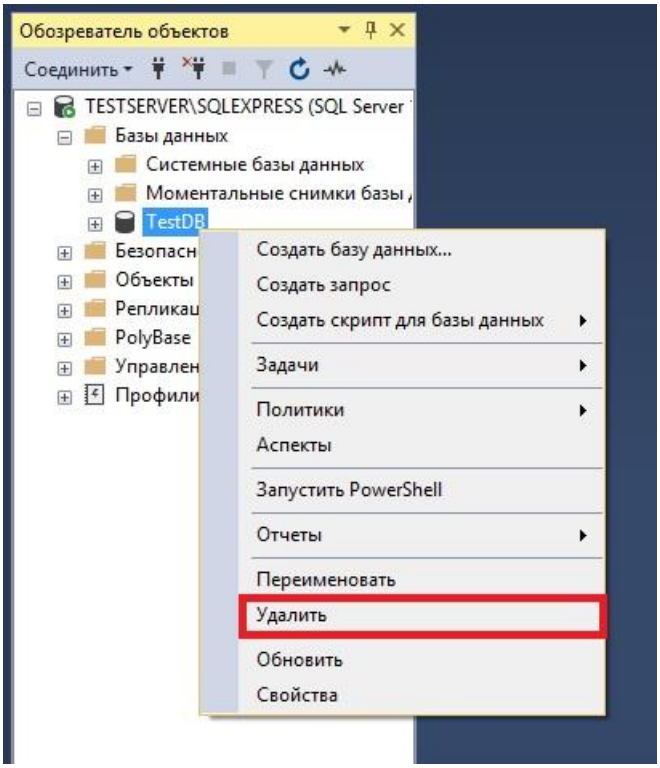


Рис. 27

Следует отметить, что если Вам все-таки необходимо удалить БД, то это можно сделать только когда в данной базе никто не работает, даже Вы, т.е. инструкцию DROP DATABASE нужно выполнять, когда контекст подключения к базе данных в SSMS отличается от базы данных, которую Вы хотите удалить (*например, Вы можете переключить контекст на системную базу данных, используя команду USE. Что это за базы, мы рассмотрим в следующем разделе, а о команде USE мы поговорим чуть позднее*).

Если Вы удалили БД, то создайте ее снова, и после этого можно продолжать читать данную книгу.

Системные базы данных

В Microsoft SQL Server, кроме пользовательских баз данных, есть еще и системные базы данных, это те, которые необходимы для функционирования самой СУБД, их удалить уже нельзя. К системным базам данным относятся следующие базы:

- **master** - база данных для хранения всех данных системного уровня;

- **msdb** - база данных для агента SQL Server;
- **model** - база данных, используемая как шаблон для создания всех пользовательских баз;
- **tempdb** - база данных для временных объектов.

На текущий момент Вы должны просто знать об их наличии.

Физическая структура базы данных в Microsoft SQL Server

Если у себя в голове мы представляем базу данных в виде таблиц, то физически база данных это, конечно же, файлы.

Базы данных, которые мы создаем в Microsoft SQL Server, состоят их нескольких файлов, а именно:

- Файлы данных – файлы, в которых хранятся сами данные;
- Файлы журнала транзакций – файлы, в которые записывается вся информация о выполненных действиях, которая используется для обеспечения целостности и восстановления данных в базе.

Файлы данных делятся на:

- Первичный файл – имеет расширение .mdf (*Master Data File*). Данный файл присутствует в любой БД;
- Вторичные файлы – имеют расширение .ndf (*Not Master Data File*). Данные типы файлов база данных может и не содержать, они создаются дополнительно к первичному файлу.

Файлы журнала транзакций имеют расширение .ldf (*Log Data File*).

Файлы базы данных располагаются в каталоге, который Вы указали в момент установки SQL Server на этапе настройки ядра в поле «*Каталог пользовательской базы данных*» для файлов данных, и в поле «*Каталог журналов пользовательской базы данных*» для журнала транзакций (*см. раздел «Установка Microsoft SQL Server Express»*).

В Microsoft SQL Server есть возможность объединять файлы данных в файловые группы (*файлы журнала транзакций не могут входить в файловые группы*). По умолчанию в SQL сервере создана файловая группа PRIMARY. Один файл данных может входить в состав только одной файловой группы.

Теперь Вы имеете представление о том, где и в каком виде хранятся базы данных в Microsoft SQL Server, а также как они создаются.

Глава 2 -Типы данных в SQL Server

Введение в типы данных SQL Server

Данная глава будет чисто теоретической, так как в ней мы поговорим о таком понятии, как тип данных. Если Вы хоть немного знакомы с программированием, не важно на каком языке, то скорей всего Вы понимаете, о чем сейчас пойдет речь, если же нет, то давайте я кратко объясню, что это такое.

Тип данных – это характеристика, определяющая, какого рода данные будут храниться в объекте, под объектом я здесь имею в виду, например, столбец в таблице или переменная. Данные могут быть, например: целые числа, числовые данные с плавающей запятой, данные денежного типа, дата, время, текст, двоичные данные и так далее. У каждого столбца в таблице, выражения, переменной или параметра есть определенный тип данных.

В T-SQL, как и в любом другом языке программирования, существует набор системных типов данных, который и определяет все доступные по умолчанию типы данных для использования.

Понимание, какой тип данных использовать в том или ином случае, как тот или иной тип данных будет вести себя в различных ситуациях - это важная часть программирования на T-SQL. Поэтому прочитайте внимательно эту главу, узнайте, какие бывают типы данных в T-SQL и для чего использовать тот или иной тип данных. Понимание типов данных поможет Вам правильно спроектировать БД или написать процедуру. Например, выбор оптимального (*корректного*) типа данных для столбца в таблице может в будущем отразиться на размере базы (*если таблица будет интенсивно расти*), а именно, она будет значительно меньше, чем если бы Вы выбрали какой-то некорректный тип данных, причем эта разница может достигать несколько сотен мегабайт, а то и нескольких гигабайт!

Для примера скажу, что одной из самых распространённых ошибок при выборе типа данных является выбор для столбца в таблице, который должен содержать данные 0 или 1, т.е. по своей сути тип данных Boolean, целочисленный тип данных SMALLINT или INT. Забегая вперед, скажу, что такого типа данных как Boolean в T-SQL нет, поэтому для этих целей разработчики и используют похожие (*подходящие*) типы данных, и в большинстве случаев их выбор неправильный. Если Вам нужно хранить только значения 0 или 1 (*т.е. как Boolean, в других языках такой тип есть*), то в T-SQL существует специальный тип данных **BIT**, SQL сервер выделяет для хранения всего 1 байт, но в отличие от типа TINYINT, под который также отводится 1 байт, SQL сервер оптимизирует хранение бит столбцов. Если таблица содержит не больше 8 бит столбцов (*т.е. столбцов с типом данных BIT*), столбцы хранятся как 1 байт, если таких столбцов от 9 до 16, то 2 байта и т.д.

Наверное, я немного Вас запутал, и Вам ничего не понятно, но ничего страшного, после того как Вы ознакомитесь с описанием каждого типа данных в T-SQL, вернитесь к данному разделу и прочитайте его, скорей всего описанная выше ошибка Вам станет более понятной.

В Microsoft SQL Server типы данных делятся на следующие категории:

- Точные числа;
- Приблизительные числа;
- Символьные строки;
- Символьные строки в Юникоде;
- Дата и время;
- Двоичные данные;
- Прочие типы данных.

Давайте же рассмотрим каждую категорию и каждый тип данных.

Точные числа

Наименование типа	Хранилище	Описание
bit	Если в таблице до 8 bit-столбцов - 1 байт, если от 9 до 16, то 2 байта и так далее.	Может принимать значения 1, 0 или NULL. Часто используется как тип данных Boolean. Строковые значения TRUE и FALSE можно преобразовать в значения данного типа: TRUE преобразуется в 1, а FALSE в 0.
tinyint	1 байт	Целые числа от 0 до 255
smallint	2 байта	от -2^15 (-32 768) до 2^15-1 (32 767).
int	4 байта	от –2^31 (–2 147 483 648) до 2^31-1 (2 147 483 647). Это основной целочисленный тип данных в Microsoft SQL Server.
bigint	8 байт	от -2^63 (-9 223 372 036 854 775 808) до 2^63-1 (9 223 372 036 854 775 807).
numeric (p, s) и decimal (p, s)	Точность: от 1 до 9 = 5 байт; от 10 до 19 = 9 байт; от 20 до 28 = 13 байт; от 29 до 38 = 17 байт.	Тип числовых данных с фиксированной точностью и масштабом. numeric и decimal функционально эквивалентны. p (точность) - максимальное количество десятичных разрядов числа, которые будут храниться (как слева, так и справа от десятичной запятой). Точность может быть значением в диапазоне от 1 до 38, по умолчанию 18. s (масштаб) - максимальное количество десятичных разрядов числа справа от десятичной запятой. Максимальное число цифр слева от десятичной запятой определяется как p - s (точность - масштаб). Масштаб может иметь значение от 0 до p, по умолчанию 0. Максимальный размер хранилища зависит от точности. Тип данных numeric и decimal может принимать значение от –10^38+1 до 10^38–1.
smallmoney	4 байта	Тип данных для хранения денежных значений с точностью до одной десятитысячной денежной единицы. Число от -214 748,3648 до 214 748,3647
money	8 байт	Тип данных для хранения денежных значений с точностью до одной десятитысячной денежной единицы. Число от -922 337 203 685 477,5808 до 922 337 203 685 477,5807

Совет 2

Если Вам нужно хранить в столбце таблицы логический тип данных (*только TRUE или FALSE*), т.е. как Boolean в других языках программирования, то используйте тип данных BIT, не нужно использовать SMALLINT или INT.

Приблизительные числа

Наименование типа	Хранилище	Описание
float (n)	Зависит от значения n: От 1 до 24 (7 знаков) = 4 байта; От 25 до 53 (15 знаков) = 8 байт.	Используется для числовых данных с плавающей запятой. n - это количество битов, используемых для хранения мантиссы числа в формате float при экспоненциальном представлении. n определяет точность данных и размер для хранения. Может принимать значение от 1 до 53, по умолчанию 53. Диапазон значений от –1,79E +308 до 1,79E+308.
real	4 байта	Используется для числовых данных с плавающей запятой. real соответствует в ISO типу float(24). Диапазон значений от –3.40E+38 до 3.40E+38.

Совет 3

Не используйте столбцы с типами float и real в условии SQL запросов, так как данные типы не хранят точных значений. Также не используйте float и real в финансовых приложениях, в операциях, связанных с округлением. Для этого лучше использовать decimal, money или smallmoney.

Символьные строки

Наименование типа	Хранилище	Описание
char (n)	n байт	Строка с фиксированной длиной не в Юникоде, где n длина строки (от 1 до 8000). По умолчанию n = 1, если значение n не указано при использовании функций CAST, длина по умолчанию равна 30 (что это за функция мы рассмотрим чуть позже в данной книге).
varchar (n | max)	Размер занимаемой памяти в байтах = количество введенных символов + 2 байта. Если указать MAX, то максимально возможный размер = 2^31-1 байт (2 ГБ).	Строковые данные переменной длины не в Юникоде, где n длина строки (от 1 до 8000). По умолчанию n = 1, если значение n не указано при использовании функций CAST, длина по умолчанию равна 30.
text	Размер занимаемой памяти в байтах = количество введенных символов. Максимальный размер 2^31-1 (2 147 483 647 байт, 2 ГБ).	Строка переменной длины не в Юникоде. Является устаревшим типом данных, рекомендуется использовать varchar(max).

Символьные строки в Юникоде

Наименование типа	Хранилище	Описание
nchar (n)	n * 2 байт	Строка с фиксированной длиной в Юникоде, где n длина строки (от 1 до 4000). По умолчанию n = 1, если значение n не указано при использовании в функции CAST, длина по умолчанию равна 30.
nvarchar (n | max)	Размер занимаемой памяти в байтах = количество введенных символов, умноженное на 2 + 2 байта. Если указать MAX, то максимально возможный размер = 2^31-1 байт (2 ГБ).	Строка переменной длины в Юникоде, где n длина строки (от 1 до 4000). По умолчанию n = 1, если значение n не указано при использовании в функции CAST, длина по умолчанию равна 30.
ntext	Размер занимаемой памяти в байтах = количество введенных символов, умноженное на 2. Максимальный размер 2^30 - 1 (1 073 741 823 байт, 1 ГБ).	Строка переменной длины в Юникоде. Является устаревшим типом данных, рекомендуется использовать nvarchar(max).

Дата и время

Наименование типа	Хранилище	Диапазон	Точность	Описание
date	3 байта	От 01.01.0001 до 31.12.9999	1 день	Используется для хранения даты.
datetime	8 байт	От 01.01.1753 00:00:00 до 31.12.9999 23:59:59,997	0,00333 секунды	Используется для хранения даты, включая время с точностью до одной трехсотой секунды.
datetime2	От 6 до 8 байт (в зависимости от точности: менее 3 цифр = 6 байт, 3-4 цифры = 7 байт, более 4 цифр = 8 байт)	От 01.01.0001 00:00:00.0000000 до 31.12.9999 23:59:59.9999999	100 наносекунд	Расширенный вариант типа данных datetime, имеет более широкий диапазон дат и большую точность в долях секунды (до 7 цифр).
smalldatetime	4 байта	От 01.01.1900 00:00:00 до 06.06.2079 23:59:00	1 минута	Сокращенный вариант типа данных datetime, имеет меньший диапазон дат и не имеет долей секунд.
time [Точность]	От 3 до 5 байт	От 00:00:00.0000000 до 23:59:59.9999999	100 наносекунд	Используется для хранения времени дня. Точность может быть целым числом от 0 до 7, по умолчанию 7 (100 наносекунд, 5 байт). Если указать 0, то точность будет до секунды (3 байта).

datetimeoffset [Точность]	От 8 до 10 байт	От 01.01.0001 00:00:00.0000000 до 9999-12-31 23:59:59.9999999	100 наносекунд	Используется для хранения даты и времени, включая смещение часовой зоны относительно универсального глобального времени. Точность определяет количество знаков в дробной части секунды, данное значение может быть от 0 до 7, по умолчанию 7 (100 наносекунд, 10 байт).

Двоичные данные

Наименование типа	Хранилище	Описание
binary (n)	n байт	Двоичные данные фиксированной длины. n - значение от 1 до 8000. Если не указывать n, то значение по умолчанию 1, если не указать в функции CAST, то 30. Данный тип лучше использовать в случаях, когда размер данных, которые будут храниться в столбце, можно заранее определить.
varbinary (n | max)	Размер занимаемой памяти в байтах = фактический размер данных + 2 байта. Если указать MAX, то максимально возможный размер = 2^31-1 байт (2 ГБ).	Двоичные данные с переменной длиной. n - значение от 1 до 8000. Если не указывать n, то значение по умолчанию 1, если не указать в функции CAST, то 30. Данным типом лучше пользоваться, если размер данных в столбце заранее определить трудно. Если размер данных превышает 8000 байт, необходимо использовать тип varbinary(max).
image	Максимальный размер до 2^31-1 (2 147 483 647 байт, 2 ГБ).	Двоичные данные с переменной длиной. Является устаревшим типом данных, рекомендуется использовать varbinary(max).

Прочие типы данных

Наименование типа	Хранилище	Описание
cursor		Данный тип данных можно использовать в переменных или выходных параметрах хранимых процедур, которые содержат ссылку на курсор. Тип cursor не может быть использован для столбца в таблице.
table		Особый тип данных для переменных, который предназначен для хранения результирующего набора данных. Переменные с данным типом называют – табличные переменные. Подробней о переменных мы поговорим в соответствующей главе.

sql_variant		Универсальный тип данных, который может хранить значения различных типов данных. Однако sql_variant может хранить значения не всех типов, которые есть в SQL сервере, например, следующие типы нельзя сохранить при помощи типа данных sql_variant: varchar(max), varbinary(max), nvarchar(max), xml, text, ntext, image, rowversion, hierarchyid, datetimeoffset, а также пространственные типы данных и определяемые пользователем типы. Тип sql_variant не может также иметь sql_variant в качестве базового типа.
rowversion (timestamp)	8 байт	Тип данных rowversion представляет собой автоматически создаваемые уникальные двоичные числа. В таблице может быть определен только один столбец типа rowversion. После любого обновления строки или вставки новой строки в таблицу, которая содержит столбец типа rowversion, значение увеличенной rowversion вставляется в столбец с данным типом. Поэтому столбец с типом данных rowversion не рекомендуется использовать в ключе, особенно в первичном ключе. timestamp является синонимом типа данных rowversion, но данный синтаксис устарел и его использовать нежелательно.
xml	Не более 2 ГБ.	Используется для хранения XML-данных. Как на T-SQL работать с XML данными в данной книге мы, кончено же, рассмотрим.
uniqueidentifier	16 байт	Глобальный уникальный идентификатор (GUID). Инициализировать столбец или переменную с типом uniqueidentifier можно с помощью функции NEWID или путем преобразования строки xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx, где каждый x – это шестнадцатеричная цифра (0–9 или A–F).
hierarchyid	Максимум 892 байта	Тип данных используется для представления положения в древовидной иерархии.
Пространственные типы		К пространственным типам относятся: **geography** – это географический пространственный тип данных, который используется для представления данных в системе координат круглой земли, **geometry** – это пространственный тип данных для представления данных в евклидовом пространстве (*плоской системе координат*).

Приоритеты типов данных в T-SQL

При написании T-SQL инструкций, когда оператор объединяет два выражения с разными типами данных, происходит неявное преобразование типов, если такое преобразование не поддерживается, SQL сервер будет выдавать ошибку. Чтобы определять, какой тип данных из выражений преобразовывать, SQL Server применяет правила приоритета типов данных. Тип данных, который имеет меньший приоритет, будет преобразован в тип данных с большим приоритетом. Если оба выражения имеют одинаковый тип данных, результат операции будет иметь такой же тип данных.

В Microsoft SQL Server существует следующий приоритет системных типов данных:

1. sql_variant (высший приоритет)
2. xml
3. datetimeoffset
4. datetime2
5. datetime
6. smalldatetime
7. date
8. time
9. float
10. real
11. decimal
12. money
13. smallmoney
14. bigint
15. int
16. smallint
17. tinyint
18. bit
19. ntext
20. text
21. image
22. timestamp
23. uniqueidentifier
24. nvarchar (включая nvarchar(max))
25. nchar
26. varchar (включая varchar(max))
27. char;
28. varbinary (включая varbinary(max))
29. binary (низший приоритет).

Синонимы типов данных в T-SQL

В T-SQL для совместимости со стандартом ISO существуют синонимы системных типов данных. Эти синонимы можно использовать в своих инструкциях точно так же, как и соответствующие системные типы данных. Единственный момент, что после создания объекта в Microsoft SQL Server, синониму назначается базовый тип данных, связанный с этим синонимом, иными словами, каких-либо признаков, что в инструкции использовался именно синоним, нет.

Существуют следующие синонимы:

Системный тип данных	Синоним типа
varbinary	Binary varying
varchar	char varying
char	character
char(1)	character
char(n)	character(n)
varchar(n)	character varying(n)
decimal	Dec
float	Double precision
real	float[(n)]; n = 1-7
float	float[(n)]; n = 8-15
int	Integer
nchar(n)	national character(n)
nchar(n)	national char(n)
nvarchar(n)	national character varying(n)
nvarchar(n)	national char varying(n)
ntext	national text
rowversion	timestamp

Глава 3 - Таблицы

Описание

Как создавать базу данных, мы научились, и мы ее создали, но пока в нашей базе нет никаких данных, а как Вы помните, данные в базе данных представлены в виде таблиц, соответственно, говоря о том, что в БД нет данных, я подразумеваю, что в ней нет еще ни одной таблицы.

Какие типы данных есть в T-SQL, мы также теперь знаем, поэтому мы можем смело переходить к рассмотрению таблиц.

В данной главе мы поговорим о том, как создаются таблицы, какие типы таблиц бывают, как они изменяются и удаляются.

Напомню, что Вы должны обязательно писать и выполнять SQL запросы, которые мы рассматриваем в этой книге! Это ключ обучения! Я могу точно сказать, что если Вы этого не будете делать, то после прочтения книги, даже если Вы сразу перейдете к практике, Вам будет сложнее осваивать T-SQL, чем если бы Вы это делали сразу, ведь практика — это отличный механизм понимания и запоминания того или иного действия. В качестве аналогии приведу пример с вождением автомобиля, Вы можете перечитать кучу книг и знать все, что нужно делать за рулем, но, когда сядете за руль, Вы поймете, что водить Вы не умеете, и скорей всего у Вас даже не получится тронуться с первого раза.

Поэтому применяйте полученные знания сразу на практике, иначе, если не практиковаться, все, что Вы узнали, Вы благополучно забудете спустя некоторое время.

Это было небольшое лирическое отступление, а сейчас давайте продолжать.

Перед тем как переходить к непосредственному созданию таблицы, необходимо сначала ее спроектировать. Для этого Вам нужно:

1. Определить, какие данные будет содержать таблица;
2. Разработать перечень столбцов;
3. Выбрать тип данных для каждого столбца. Как я уже говорил, над типом данных нужно подумать, к этому моменту у Вас уже должно быть представление о том, что за данные будут храниться в каждом из столбцов. Поэтому на основе этих сведений и выбирайте подходящий тип;
4. Определиться, является ли конкретный столбец обязательным к заполнению. Если не обязательный, то значит, он может содержать значения NULL, т.е. неопределённое значение (*подробней об этом значении мы поговорим в Главе 4*);

Также Вам необходимо знать, что в рамках схемы не может быть несколько таблиц с одинаковым названием.

Схема – это пространство имен в базе данных, своего рода контейнер объектов. Их может быть несколько. Схема может принадлежать любому пользователю, и это владение может передаваться.

Если не указывать схему в инструкциях, SQL Server будет считать, что объект находится в схеме по умолчанию, определенной для пользователя. Если пользователю явно схему по умолчанию не определяли, то он привязан к стандартной схеме dbo.

Создание

В качестве примера я предлагаю создать две таблицы:

- TestTable – она будет содержать информацию о товарах;
- TestTable2 – а она будет содержать категории товаров.

В первой таблице, для того чтобы указать, к какой категории товаров относится тот или иной товар, мы поставим ссылку на вторую таблицу.

Общий, упрощённый синтаксис создания таблиц вот такой.

CREATE TABLE Название таблицы (
[Название столбца] [Тип данных] [Возможность принятия значения NULL]
)

Сначала Вы указываете инструкцию **CREATE TABLE**, затем в скобочках, перечисляете список столбцов, которые будет содержать таблица, с указанием типа данных и возможностью принятия значения NULL, т.е. является ли столбец обязательным к заполнению или нет.

Итак, запускаем SSMS, открываем окно создания запроса (*в контексте нашей тестовой базы данных TestDB, проверить и переключить контекст подключения можно на панели редактора SQL запросов Management Studio*), и вставляем следующую SQL инструкцию.

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

GO

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

GO
```

Где,

- ✓ CREATE TABLE – это инструкция создания таблицы;
- ✓ TestTable и TestTable2 – это название создаваемой таблицы;
- ✓ ProductId – имя столбца, который будет содержать идентификатор товара, он имеет тип данных INT, и он обязательный, т.е. значение NULL он не может принимать. Это и понятно, как тогда без него идентифицировать товар. Также для данного столбца мы указали что он является столбцом идентификаторов (IDENTITY), который автоматически генерирует идентификатор, начиная с 1 и шагом 1. В качестве альтернативы можно использовать и специальный объект последовательности, его поддержка появилась в Microsoft SQL Server 2012, но его нужно создавать отдельно;
- ✓ CategoryId – это имя столбца для идентификатора категории товара, тип данных INT, значение NULL принимать не может, т.е. каждый товар должен относиться к какой-нибудь категории. В случае с таблицей 2 для данного поля я также указал свойство IDENTITY;
- ✓ ProductName – имя столбца для хранения названия товара. Тип данных текст длиной не более 100 символов. Также является обязательным;
- ✓ Price – это столбец для цены товара, тип данных денежный, значение NULL, допустим, он может принимать, скажем мы создали товар, но с ценой еще не определились;
- ✓ CategoryName – это столбец для хранения названия категории с текстовым типом длиной не более 100 символов. У каждой категории должно быть название, поэтому он обязательный.

Создавать таблицы можно также и в графическом интерфейсе Management Studio. Для этого откройте среду, в обозревателе объектов найдите базу данных, в которой Вы хотите создать таблицу, затем вызовите меню, щелкнув по контейнеру «*Таблицы*» правой кнопкой мыши и нажмите «*Создать таблицу*». Откроется конструктор, с помощью которого и можно проектировать таблицу (*вводить названия, выбирать тип данных*).

Вычисляемые столбцы

У таблицы в Microsoft SQL Server могут быть так называемые «*Вычисляемые столбцы*». Это такие столбцы, которые вычисляются на основе выражения с участием других столбцов этой же таблицы. Такие столбцы по умолчанию не хранятся физически, все значения обновляются, т.е. рассчитываются каждый раз при обращении к ним. Однако можно указать ключевое слово PERSISTED при создании подобного столбца, которое будет означать, что такой столбец нужно физически хранить, при этом значения данного столбца будет обновляться, когда будут вноситься любые изменения в столбцы, входящие в вычисляемое выражение.

К вычисляемым столбцам можно обращаться так же, как и к обычным столбцам в SQL запросах (*в списке выборки, в условии, при сортировке*).

Например, давайте создадим еще одну таблицу, в которой будет вычисляемый столбец. Допустим, у нас таблица, в которой хранятся весовые товары.

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

В данном примере, вычисляемый столбец Summa, он вычисляется на основе выражения, а именно перемножения веса на цену. При этом мы указали, что эти данные должны храниться физически, т.е. указали ключевое слово PERSISTED.

В случае если у Вас уже создана таблица, и Вам необходимо добавить в нее вычисляемый столбец, то это делается с помощью инструкции **ALTER TABLE**, и команды ADD, подробней об этой инструкции мы поговорим в разделе, посвященном изменению таблиц. Поэтому сейчас я Вам просто покажу, как выглядит инструкция по добавлению вычисляемого столбца. Для примера давайте добавим точно такой же вычисляемый столбец, только, конечно, у него будет другое название, я назвал его SummaDop.

```
ALTER TABLE TestTable3 ADD [SummaDop] AS ([Weight] * [Price])
PERSISTED;
```

Добавление вычисляемых столбцов, как во время, так и после создания таблицы, доступно также и в графической среде SQL Server Management Studio в конструкторе таблиц. Для этого Вам нужно открыть конструктор таблиц, найти в нем раздел «*Спецификация вычисляемого столбца*» и в поле «*Формула*» указать соответствующую формулу.

Удаление

Если у Вас возникла необходимость удалить таблицу, то это можно сделать с помощью следующей инструкции. Для примера давайте удалим таблицу, которую мы создали в предыдущем разделе (*TestTable3*).

```
DROP TABLE TestTable3
```

Где, DROP TABLE – это инструкция удаления таблицы, TestTable3 – это соответственно название той таблицы, которую мы хотим удалить.

В графическом интерфейсе SSMS удалить таблицу можно путем вызова контекстного меню правой кнопкой мыши, предварительно выбрав нужную таблицу. Затем найти и нажать на пункт «*Удалить*».

После этого откроется окно удаления таблицы, Вам необходимо подтвердить свои намерения, нажав «ОК».

При этом стоит понимать, что если в таблице, которую Вы хотите удалить, есть какие-либо связи по ключу, то ее просто так удалить не получится, сначала нужно будет пересмотреть эти связи (*подробней о ключах и ограничениях мы будем с Вами разговаривать в соответствующей главе*).

Изменение

Достаточно часто возникает необходимость внести изменения в существующую таблицу, например, добавить новый столбец или изменить тип данных. В T-SQL для этих целей есть инструкция **ALTER TABLE**. Но так же, как и при удалении таблицы, здесь есть свои нюансы. Например, добавить новый столбец, который не может принимать значения NULL, не получится, если в таблице уже есть данные, нужно сначала добавить этот столбец без этой опции (*т.е. с возможностью принятия значения NULL*), потом заполнить его данными, и уже потом обновить данный параметр, т.е. указать NOT NULL. Или также могут возникнуть проблемы при изменении типа данных. Но мы углубляться в подробности не будем, когда у Вас возникнут такие трудности, Вы уже будете знать, как искать решение той или иной проблемы. Напомню, цель этой книги - дать Вам начальные знания языка T-SQL и Microsoft SQL Server в целом, тем более на текущий момент Вам и так, наверное, поступающей информации достаточно!

Упрощенный синтаксис инструкции ALTER TABLE следующий.

ALTER TABLE [Название таблицы] [Тип изменения] [Название столбца] [Тип данных] [Возможность принятия значения NULL]

В первом примере давайте сделаем в таблице TestTable (*которую мы создали чуть ранее*) столбец Price обязательным для заполнения, т.е. укажем NOT NULL.

```
ALTER TABLE TestTable ALTER COLUMN [Price] [Money] NOT NULL
```

В данном случае инструкция ALTER COLUMN говорит SQL серверу, что необходимо произвести изменения в одном из столбцов.

Сейчас давайте допустим, что нам вдруг вообще больше не нужен этот столбец в этой таблице, и нам его нужно удалить, для этого пишем DROP COLUMN.

```
ALTER TABLE TestTable DROP COLUMN [Price]
```

А теперь в качестве примера давайте вернем назад этот столбец в эту таблицу в том виде, каким он был изначально, другими словами, добавим столбец в таблицу. Это делается с помощью ключевого слова ADD.

```
ALTER TABLE TestTable ADD [Price] [Money] NULL
```

На самом деле синтаксис ALTER TABLE гораздо сложней, возможностей по изменению таблицы намного больше, чем мы сейчас рассмотрели, но если мы в каждой главе будем лезть в дебри, то у Вас скорей всего пропадёт желание изучать данный язык, поэтому учимся основам! Дальше будет еще интересней!

Временные таблицы

В Microsoft SQL Server есть возможность создавать так называемые «*Временные таблицы*» - это таблицы, которые, соответственно, хранятся временно, и они не предназначены для постоянного хранения данных. Для чего нужны такие таблицы? Временные таблицы нужны нам для того, чтобы сохранять некие промежуточные или расчетные данные, которые нужны нам в ходе выполнения SQL инструкции или хранимой процедуры, но после завершения этой инструкции или процедуры они нам больше не нужны.

Другими словами, если Вам необходимо выполнять много расчетов и запоминать их результат для дальнейших расчетов (*например, реализация сложного алгоритма, бизнес логики, отчета и так*

далее), а создавать дополнительные таблицы, которые будут видеть все пользователи, и их, конечно же, нужно потом удалять, неудобно и, главное, неправильно. Так как, если параллельно запустить эти инструкции, возникнет конфликт уникальности таблиц, я говорил об этом ранее, названия таблиц должны быть уникальными в рамках схемы.

В MS SQL Server существует два вида временных таблиц: *локальные и глобальные*. Отличаются они друг от друга зоной видимости и, соответственно, доступностью.

Локальные временные таблицы – видимы только во время текущего сеанса пользователя. Удаляются они автоматически, когда сеанс завершается или завершается выполнение хранимой процедуры, в которой используются эти таблицы. Имя локальной временной таблицы начинается с префикса - символ решетки (#).

Глобальные временные таблицы – видимы во всех сеансах и доступны всем пользователям. Удаляются такие таблицы после завершения сеанса, создавшего таблицу, а также после прекращения связи с этими таблицами других сеансов. Имя глобальной временной таблицы начинается с префикса – 2 символа решетки (##).

Создаются такие таблицы так же, как и обычные, инструкцией CREATE TABLE. Удалить такие таблицы можно с помощью инструкции DROP TABLE.

Рекомендуется контролировать процесс удаления любых временных таблиц, т.е. принудительно удалять их, когда они больше не нужны в процессе выполнения SQL инструкции или хранимой процедуры.

Для примера давайте создадим локальную временную таблицу и затем ее удалим.

Создание временной таблицы.

```
CREATE TABLE #TestTable(
     [Id]           [INT] IDENTITY(1,1) NOT NULL,
     [ProductName]  [VARCHAR](100) NOT NULL,
     [Price]        [Money] NOT NULL
)
```

Удаление временной таблицы.

```
DROP TABLE #TestTable
```

С временными таблицами Вы можете делать все то же самое, что и с обычными таблицами, наполнять их данными, удалять и изменять эти данные и, конечно же, обращаться к ним.

Глава 4 - Выборка данных – оператор SELECT

Введение

База данных у нас есть, таблицы у нас есть, но нет еще данных в этих таблицах, т.е. они пустые, поэтому перед продолжением давайте добавим данные в наши тестовые таблицы.

Ниже я представлю скрипт наполнения таблиц для примеров, описанных в данной главе, в данном случае, Ваша задача просто скопировать его и выполнить в окне запросов среды Management Studio.

Подробно описывать эту инструкцию я пока не буду, так как эту тему мы детально рассмотрим в соответствующей главе этой книги (*Модификация данных*). Единственное скажу, что в этой инструкции мы добавляем данные в наши тестовые таблицы (*TestTable и TestTable2*), которые мы создавали в Главе 3, посвящённой таблицам (*3 строки в таблицу TestTable и 2 строки в таблицу TestTable2*).

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

GO

INSERT INTO TestTable2
     VALUES ('Комплектующие компьютера'),
            ('Мобильные устройства')
GO
```

Описание инструкции SELECT

Чтобы получить данные из базы данных, необходимо выполнить соответствующую выборку из таблиц, иными словами, инструкцию получения нужной информации.

В языке SQL для этих целей существует инструкция SELECT.

SELECT - это команда языка SQL, которая используется для получения информации из базы данных. В результате выполнения инструкции SELECT, т.е. запроса на выборку, SQL сервер вернет Вам соответствующие данные в виде таблицы, обычно это называют результатом выполнения запроса или результирующим набором данных.

SELECT - очень мощная и важная инструкция! Быть может, Вы читаете эту книгу, чтобы научиться писать запросы на выборку данных, так вот, это делается с помощью инструкции SELECT. Каждый программист любого уровня *(или просто системный администратор*), который имеет дело с базами данных, умеет писать запросы на выборку и знаком с оператором SELECT, пусть даже поверхностно, но знаком. Может быть, Вам и не придётся или не потребуется создавать таблицы, но писать запросы SELECT, если Вы работаете с базой данных, Вам потребуется однозначно!

В начале своего пути первое, с чем я столкнулся, это был как раз оператор SELECT, так как мне сначала требовалось просто писать запросы на получения данных. После того как я освоил написание запросов на выборку, и это стало удовлетворять мое руководство, мне стали поступать задачи уже на модификацию данных, доработку структуры базы и так далее.

С первого взгляда, может показаться, что нет ничего сложного в том, чтобы написать запрос на получение данных, может быть это и так, но, если Вам нужно получить данные из сложной структуры, провести глубокий анализ этих данных и представить их в том виде, в котором хочет, например, Ваш начальник, то это сделать «*Просто*» не получится.

Для получения данных инструкция SELECT имеет много секций (*или частей*), каждая из которых также имеет много различных возможностей по их использованию.

Сейчас я Вам представлю упрощённый синтаксис инструкции SELECT, иными словами, основные части (*секции*). В запросах они должны использоваться **строго в определенном порядке** (*как например ниже*).

SELECT *Столбец 1, Столбец 2 …. (Список столбцов для выборки)*
FROM *Источник данных (Таблица)*
JOIN *Объединение с другим источником (Другая таблица)*
WHERE *Условие (Отбор только нужных данных)*
GROUP BY *Группировка данных*
HAVING *Условие по сгруппированным данным*
ORDER BY *Сортировка данных*

Следует отметить, что SQL сервер фактически обрабатывает секции не в таком порядке, как они указаны в синтаксисе выше, это сделано специально, чтобы писать и читать запросы человеку было удобней. Фактически сервер обрабатывает секции в следующем порядке:

1) FROM (*включая JOIN*)
2) WHERE
3) GROUP BY
4) HAVING
5) SELECT
6) ORDER BY

Когда будете писать запросы, помните, что SQL сервер обрабатывает Ваш запрос именно в таком порядке, это поможет Вам в случае появления тех или иных ошибок отследить их и устранить.

Сейчас мы с Вами подробней рассмотрим каждую секцию, а также Вы узнаете много чего интересного еще!

Список выборки команды SELECT

Обязательная и самая главная часть инструкции SELECT – это **список выборки**, другими словами, это указание на те данные, которые Вы хотите получить в результате выполнения запроса SELECT.

В списке выборки Вы должны указывать все столбцы, данные из которых Вам нужно получить. Но также существует возможность указать символ звездочки *, что будет означать, что Вам нужны все данные (*т.е. столбцы*) из источника данных. Обычно * используется только разработчиками для каких-нибудь тестовых запросов, в действующих приложениях или процедурах никаких * не используйте, только перечисление конкретных столбцов, иначе в дальнейшем у Вас могут возникнуть проблемы, к тому же это снижает производительность запросов.

Столбцы в списке выборки указываются через запятую, они возвращаются, т.е. формируют результирующий набор данных, в том порядке, в котором Вы их указали в списке выборки.

Например, давайте сделаем выборку из нашей тестовой таблицы TestTable, которую мы создали чуть ранее в главе 3 и наполнили данными в начале этой главы. Пока секцию FROM я не буду подробно объяснять, единственное скажу, что в ней мы указываем источник данных, в данном случае таблицу, из которой мы хотим получить данные. Результирующий набор данных представлен на рисунке 28 (*наиболее важные и интересные результирующие наборы данных я всегда буду представлять Вам в виде рисунков*).

```
SELECT ProductID, ProductName, Price
FROM TestTable
```

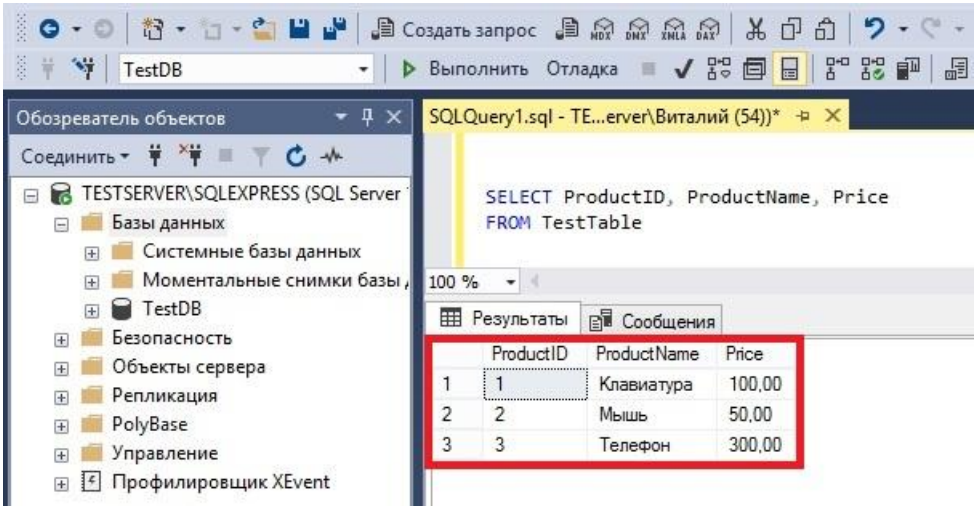


Рис. 28

В данном примере в списке выборки мы указали три столбца, SQL сервер вернул нам результат в виде трех соответствующих столбцов.

Оператор TOP

Инструкция SELECT по умолчанию возвращает все записи из источника с учетом условий. Но бывают случаи, когда Вам необходимо получить только первые несколько строк результирующего набора данных. Для этих целей в T-SQL существует специальный оператор, ключевое слово **TOP**, которое может ограничить количество строк в итоговом наборе данных.

Например, давайте представим, что нам необходимо из таблицы TestTable получить всего 2 первые строки, остальные нам не нужны. Для этого мы напишем следующий запрос на выборку.

```
SELECT TOP 2 ProductID, ProductName, Price
FROM TestTable
```

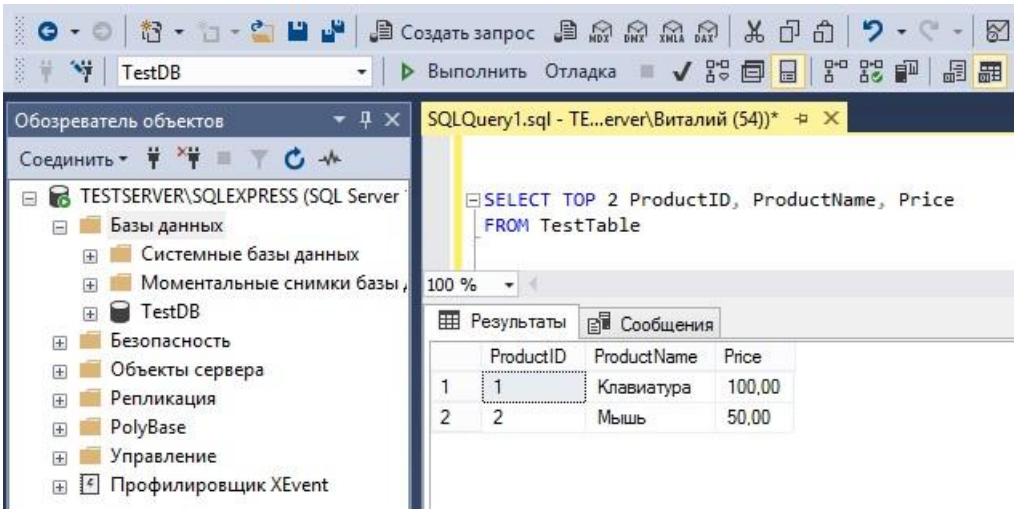


Рис. 29

Как видите, мы после команды SELECT написали ключевое слово TOP, с помощью которого мы и ограничили итоговый набор данных.

Обычно данный оператор используют вместе с сортировкой данных (*ORDER BY мы рассмотрим чуть позже*) чтобы, например, получить первые записи из отсортированных по какому-нибудь столбцу данных. Но в данных случаях у Вас может возникнуть вопрос, как быть, если в столбце, который используется для сортировки, есть повторяющиеся значения, что может дать не совсем тот результат, который хотелось бы. Допустим, Вам нужно определить 5 самых дорогих товаров. Вы, соответственно, отсортируете данные по столбцу с ценой и укажете оператор TOP 5, но, если товаров с одинаковой ценой (*эта цена входит в число самых больших*) несколько, например, 7, Вам все равно вернется 5

строк, что, как Вы понимаете, неправильно, так как самых дорогих товаров на самом деле 7. Чтобы это узнать или, как говорят, определить, есть ли «*хвосты*», т.е. строки с таким же значением, которые не попали в выборку, за счет ограничения TOP, можно использовать параметр **WITH TIES**.

В следующем примере я указал данный параметр, также я здесь использовал сортировку по цене, но об этом чуть позже.

```
SELECT TOP 2 WITH TIES ProductID, ProductName, Price
FROM TestTable
ORDER BY Price DESC
```

Оператору TOP в качестве параметра можно передать значение, характеризующее не только фактическое количество строк, которое необходимо оставить в результирующем наборе данных, но и процент, для этого необходимо указать ключевое слово **PERCENT**. Например, следующий запрос возвращает только 20 процентов итогового результата.

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

Таким образом, если в таблице TestTable будет, например, 100 строк, запрос вернет первые 20 с учетом сортировки.

Оператор DISTINCT

Как Вы уже знаете, SELECT возвращает все записи из источника, но записи в источнике могут повторяться или, может быть, у Вас так сконструирован запрос, что они могут повторяться, при этом в результирующем наборе данных эти повторы Вам не нужны.

Для быстрого решения данной задачи в T-SQL есть ключевое слово **DISTINCT**, которое включает в итоговый набор данных только уникальные записи списка выборки.

Например, нам нужно получить перечень товаров с ценой, но если наименование товара и цена повторяется, то нам эти одинаковые строки не нужны. Мы пишем следующий запрос.

```
SELECT DISTINCT ProductName, Price
FROM TestTable
```

Ключевое слово DISTINCT указывается после слова SELECT. В данном случае в итоговом наборе данных мы получили уникальные сочетания товара и цены.

Секция FROM – источник данных

Вы не могли не заметить, что в предыдущих запросах, которые мы с Вами написали чуть ранее, после списка выборки мы писали слово FROM. И сейчас я расскажу для чего.

Секция FROM указывает на источник данных. В качестве источника обычно выступают: таблицы, представления и табличные функции (*которые мы с Вами еще рассмотрим*).

Другими словами, после слова FROM мы указываем источник данных, к которому необходимо применить список выборки.

SQL сервер расценивает запись, которая идет после FROM, как источник данных, из которого пользователю необходимо получить данные. Таким образом, он определяет, а мы указываем, к какой именно таблице нужно обратиться. Без секции FROM мы просто обращались бы в никуда, поэтому практически всегда секцию FROM нужно указывать, за исключением случаев, когда в списке выборке присутствуют только константы, т.е. определенные значения, указанные вручную (*не столбцы таблицы*).

Помните, я говорил, что можно указывать знак * для того, чтобы получить все столбцы из источника (*например, таблицы*)? Давайте сейчас в качестве примера, а также для закрепления знаний, напишем запрос со звездочкой.

```
SELECT *
FROM TestTable
```

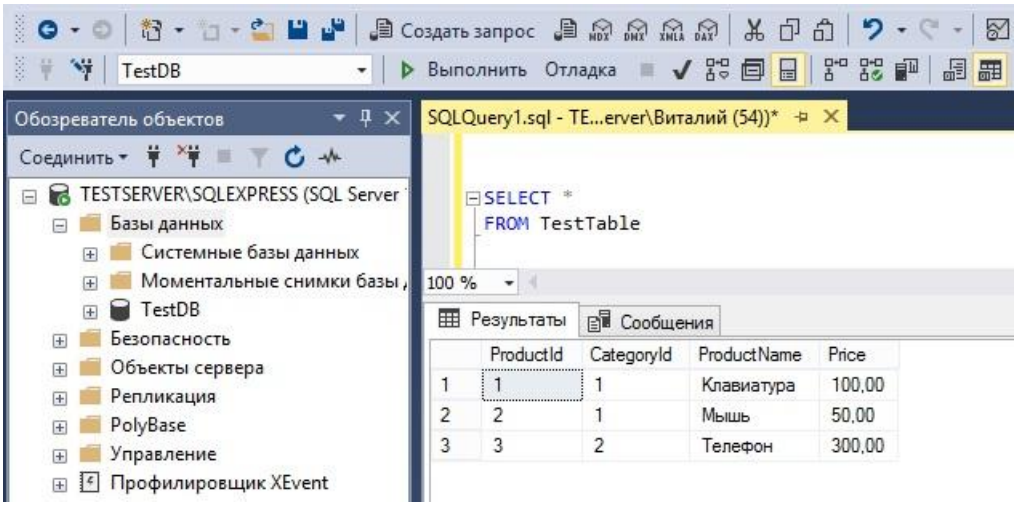


Рис. 30

В данном случае мы получили все записи и все столбцы из таблицы TestTable.

В секции FROM можно указывать источник данных (таблицу или что-то еще) из другой базы данных, и даже из другого сервера. Поэтому в данных случаях необходимо уточнять название объектов, которые выступают в качестве источника данных. Это делается путем указания полного пути к источнику (*четырехсоставное имя объекта*), используя в качестве разделителя объектов точку.

В целом это выглядит следующим образом:

ИмяСервера.ИмяБазы.ИмяСхемы.ИмяТаблицы

Например, если Вы хотите получить данные, которые расположены в другой базе того же сервера, то Вам необходимо указать перед названием таблицы сначала название базы, затем название схемы и только потом название таблицы. Если этого не сделать, как, например, в нашем примере выше, то SQL сервер будет искать данный источник на текущем сервере, в текущей базе и текущей схеме.

Давайте допустим, что наша таблица, из которой нам нужны данные, находится в другой базе данных, т.е. отличной от нашего текущего подключения. Для этого мы напишем по своей функциональности точно такой же SQL запрос, как и Выше, но при этом укажем уточненное название источника, т.е. таблицы.

```
SELECT *
FROM TestDB.dbo.TestTable
```

Где,

- ✓ TestDB – это имя базы данных, в которой расположена таблица;
- ✓ dbo – это имя схемы, к которой относится таблица;
- ✓ TestTable – это, как Вы помните, название нашей таблицы.

Псевдонимы списка выборки и источников данных

Сейчас мы с Вами рассмотрим одну очень полезную и важную вещь, которую Вы обязательно будете использовать на практике. Я говорю о псевдонимах.

Псевдоним (*его называют alias*) – это указание упрощённого (*или сокращенного*) названия столбца в списке выборки или самого источника данных. Другими словами, если имя таблицы или название столбца длинное или непонятное, или просто Вам нужно, чтобы это название отображалось в результирующем наборе с определенным названием, Вы можете задать псевдоним для таких таблиц и столбцов, указав то название, которое Вы хотите.

Псевдонимы задаются один раз после названия исходного объекта с помощью ключевого слова **AS**, т.е. там, где Вы его впервые указали. Иными словами, Вам в любом случае один раз нужно будет написать название исходного объекта, затем Вы уже можете использовать псевдоним для того, чтобы обращаться к этому объекту в текущем запросе.

Псевдоним в списке выборки можно задать для столбца или, например, для выражения (*в случае с расчетными данными*). Псевдонимы в списке выборки (*да и в источнике данных*), как я уже сказал, задаются с помощью ключевого слова AS после исходного названия, однако это необязательно (*AS можно и опустить*), но я рекомендую указывать AS перед псевдонимом, так более наглядно и удобно.

Если Вы указали псевдоним для источника данных, то Вы должны в списке выборки указывать перед названием столбца этот псевдоним, а через точку само название столбца. Можно, конечно, и не указывать псевдоним источника перед столбцом в случаях, если источник один, но если их несколько, то обязательно указывайте в списке выборки ссылку на источник в виде псевдонима, так Вы избежите некоторых проблем. Например, если есть столбцы с одинаковым названием в двух и более таблицах, то, без указания источника в списке выборки для столбцов, запрос завершится с ошибкой, так как SQL сервер не сможет определить, какой именно столбец, т.е. из какой таблицы, Вы хотите получить.

После того как Вы задали псевдоним для источника данных в секции FROM, это означает, что, обращаясь к этому псевдониму, будь то в списке выборки или в других частях (*секциях*) запроса, на самом деле Вы обращаетесь к тому источнику, для которого Вы задали псевдоним. Псевдоним – это некая ссылка на основной объект (*таблица, столбец и т.д.*).

Псевдонимы, как источника, так и столбцов в списке выборки, очень удобны. Например, представьте, что в базе данных есть таблицы с длинным названием – TestTableForProductsName. И если Вам необходимо объединять (*объединение таблиц мы рассмотрим чуть позже*) эту таблицу с другими таблицами, то без указания псевдонима, Вам нужно будет везде, где Вы используете эту таблицу, писать вот такое длинное название. Как понимаете, это не удобно, да и текст запроса вырастет в разы. Но если Вы зададите псевдоним для этой таблицы, например – T1, Вам нужно будет ссылаться не на таблицу (*с длинным названием*), а на этот (*короткий*) псевдоним.

В некоторых случаях указание псевдонима является обязательным, если, например, в качестве источника данных выступает другой запрос (*производная таблицы*), но и это мы рассмотрим чуть позже, так что в этой книге Вас ждет еще много чего интересного!

Для понимания того, как работают псевдонимы на практике, т.е. как задаются псевдонимы, и как на них потом ссылаются, давайте напишем запрос, в котором будут использоваться псевдонимы и источника данных, и псевдонимы в списке выборки, а также Вы увидите, как ссылаться на псевдоним источника данных.

```
SELECT T.ProductID AS ID,
       T.ProductName AS ProductName,
       T.Price AS [Цена]
FROM TestTable AS T
```

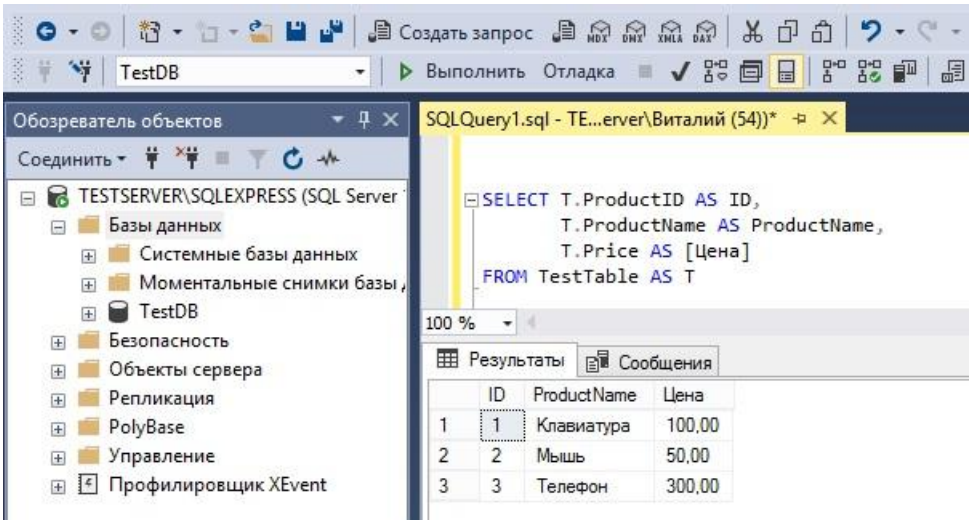


Рис. 31

Здесь я указал псевдоним T для таблицы TestTable, а также задал псевдонимы для всех столбцов. В списке выборки я уже ссылался не на таблицу TestTable, а на псевдоним T. Вы видите, что для первого столбца я задал псевдоним ID, для второго ProductName, т.е. этим я показал, что использовать исходное название для псевдонима можно, для третьего я указал [Цена], т.е. в квадратных скобочках можно написать любое название, даже на русском языке.

Условия – WHERE

Ранее мы с Вами писали запрос на выборку абсолютно всех данных из источника, но на практике это Вам делать придется крайне редко, так как в основном, в тот ли иной момент времени, нам нужны только определенные данные, определенная информация. Поэтому в языке SQL предусмотрена возможность фильтровать данные, т.е. ставить условия – это делается с помощью секции **WHERE.**

Если в запросе определено условие WHERE, то в результирующий набор данных попадут только записи, которые соответствуют условию в секции WHERE.

В WHERE может быть определено как одно выражение в качестве условия, так и несколько, т.е. некая комбинация выражений, которые образуют одно общее условие. Пришло время Вам познакомиться с таким понятием, как Предикат.

Предикат – это любое выражение, результатом которого являются значения TRUE, FALSE или UNKNOWN, иными словами, истина, ложь или неизвестно.

Предикаты используются везде, где требуется логическое значение, например, в условиях WHERE (*и в HAVING и в JOIN, эти секции мы также рассмотрим*).

Для примера давайте напишем запрос и укажем условие WHERE. Допустим, нам нужны все записи из таблицы TestTable с ценой больше 10, т.е. столбец Price > 10, для этого мы пишем следующее.

```
SELECT ProductID,
       ProductName,
       Price
FROM TestTable
WHERE Price > 10
```

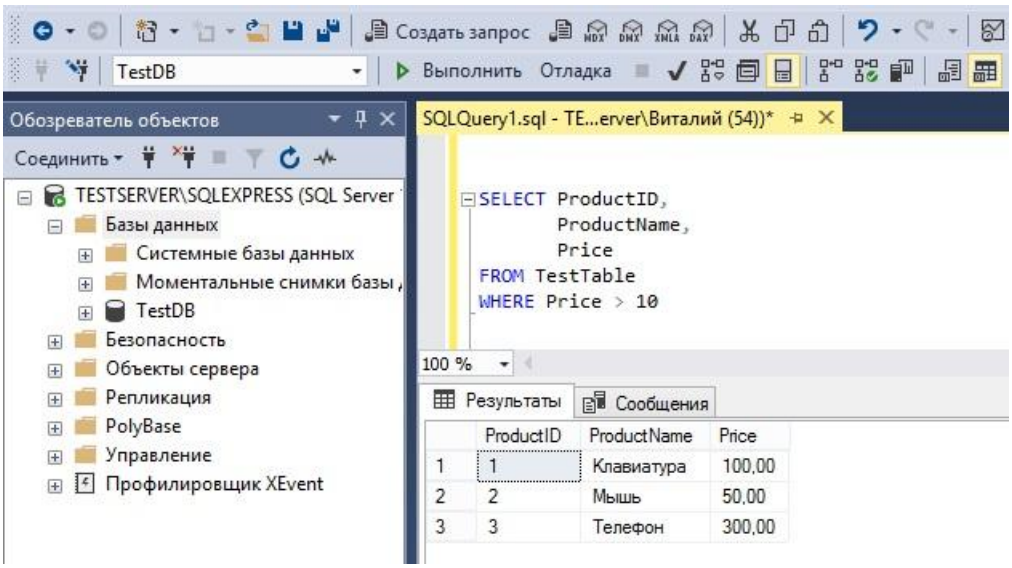


Рис. 32

После слова WHERE мы написали предикат, который возьмёт только те строки таблицы, у которых значения столбца Price больше 10, другими словами, мы сравниваем значения из столбца Price каждой строки со значением 10, и если оно больше 10, то предикат возвращает TRUE (*т.е. истина*) и условие выполняется, строка выводится. Если предикат возвращает FALSE, то, соответственно, условие не выполняется, и строка не выводится.

В предикатах можно использовать много различных операторов, давайте их рассмотрим.

- = (равно) – определяет, равняются ли сравниваемые выражения.
- > (больше) – определяет, превышает ли одно выражение над другим.

- < (меньше) - оператор используется для проверки того, что одно выражение меньше другого.
- <> (не равно) - определяет неравенства двух выражений.
- != (не равно) – оператор T-SQL, который также определяет неравенства двух выражений.
- >= (больше или равно) - оператор используется для проверки превышения либо равенства двух выражений.
- <= (меньше или равно) - оператор используется для проверки того, что одно выражение меньше или равно другому.
- !> (не больше) - оператор T-SQL, который используется для проверки того, что одно выражение не превышает другое выражение.
- !< (не меньше) - оператор T-SQL, который используется для проверки того, что одно выражение не меньше другого.
- LIKE или NOT LIKE – определяет, содержит или не содержит текстовая строка символы, заданные в шаблоне.
- BETWEEN или NOT BETWEEN – определяет, входит ли или не входит одно выражение в заданный диапазон значений.
- IN или NOT IN – определяет, совпадает или не совпадает указанное значение с одним из значений из заданного списка или вложенного запроса (*подзапросы мы, конечно же, рассмотрим!*).
- EXISTS – используется во вложенном запросе для проверки существования строк, возвращенных вложенным запросом.

Пока этого достаточно, но существуют и другие условия поиска, многие из которых мы еще рассмотрим в данной книге.

Как я уже говорил, в секции WHERE мы можем указать не один предикат, а несколько. Например, для того чтобы учитывать все предикаты, нужно их писать через ключевое слово AND, но если Вам нужно построить условие так, чтобы выводились строки, которые выполняют хотя бы одно условие, т.е. один предикат, то предикаты необходимо писать через ключевое слово OR.

А теперь, чтобы все закрепить, давайте напишем несколько примеров, чтобы было наглядно видно, как работают те или иные операторы, и как строятся предикаты.

Чуть ранее мы написали запрос, в котором указали предикат Price > 10, т.е. цена больше 10. Сейчас давайте укажем условие, при котором цена должна быть больше или равна 100, но при этом должна быть меньше или равна 500. Сначала мы используем два предиката, объединив их с помощью ключевого слова AND, а затем используем оператор BETWEEN, который даст точно такой же результат.

Снова повторюсь, обязательно пишите запросы, пробуйте, Вы тем самым привыкаете к языку и лучше усваиваете полученную информацию, также Вам может быть что-то непонятно, а написание и выполнение примеров Вам поможет все понять до конца.

```
SELECT ProductID,
       ProductName,
       Price
FROM TestTable
WHERE Price >= 100 AND Price <= 500


SELECT ProductID,
       ProductName,
       Price
FROM TestTable
WHERE Price BETWEEN 100 AND 500
```

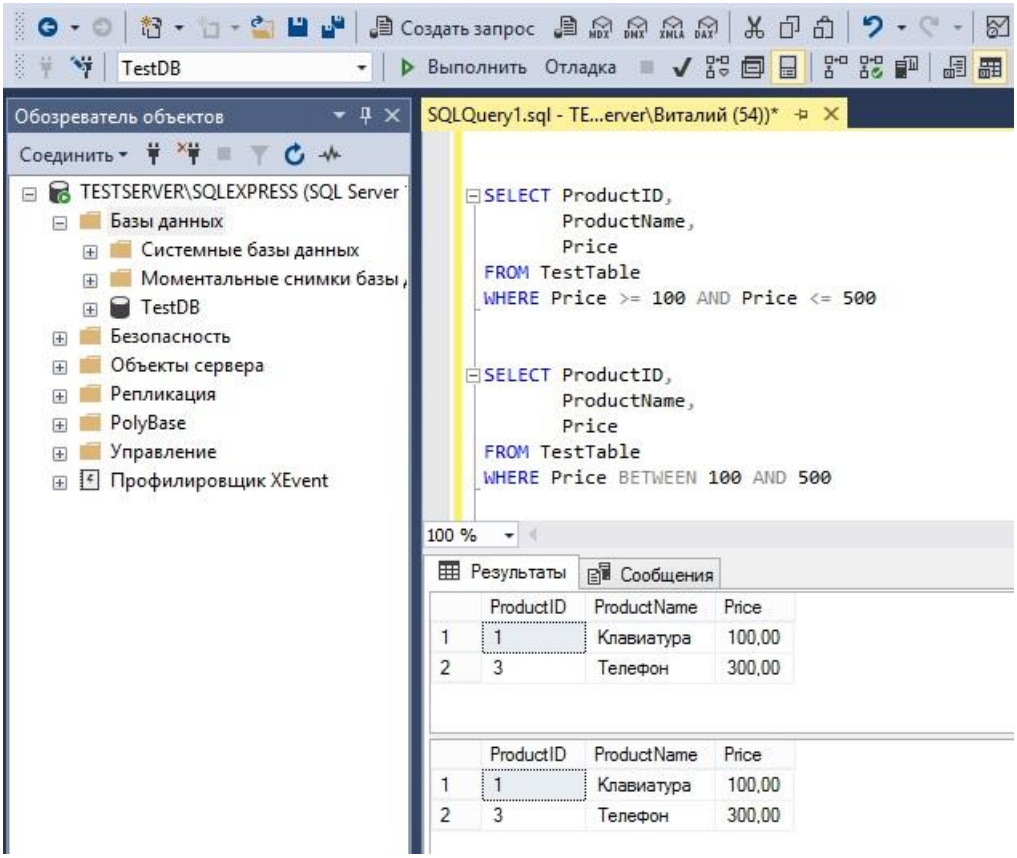


Рис. 33

Теперь давайте представим, что нам нужны все товары, название которых начинаются с буквы «Т» (*такая вот у нас хотелка*☺). Для этого мы можем использовать оператор LIKE, в котором мы укажем некий шаблон поиска данных в строке.

```
SELECT ProductID,
       ProductName,
       Price
FROM TestTable
WHERE ProductName LIKE 'Т%'
```

В секции WHERE, после названия необходимого столбца, мы указали оператор LIKE, затем в апострофах нужные символы, в нашем случае это всего одна буква, после которых мы поставили символ %, который означает, что остальная часть сроки нас не интересует, т.е. только начало. Если бы мы указали % в самом начале нашего шаблона, например, '%Т%', то у нас осуществлялся бы поиск буквы «Т» в любом месте строки, т.е. не важно, что там стоит перед ней и после, главное, что она есть в строке.

Если вдруг Вам необходимо получить только записи с определёнными несколькими значениями, например, Вы хотите знать, какие товары имеют определенную цену, допустим 50 и 100, то Вы для этого можете использовать оператор IN.

```
SELECT ProductID,
       ProductName,
       Price
FROM TestTable
WHERE Price IN (50, 100)
```

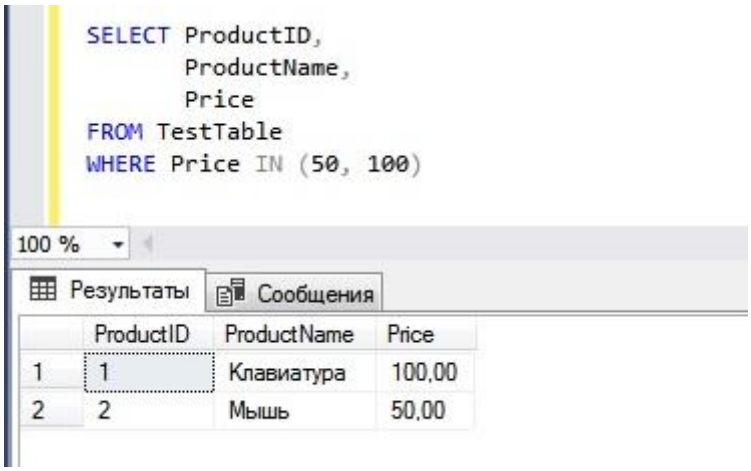


Рис. 34

В данном случае SQL сервер сформирует результирующий набор из строк, столбец Price которых имеет значение 50 или 100. Иными словами, в IN мы перечисляем значения, которые хотим получить.

Пример с OR

Точно такой же результат мы получим, если используем два операнда с использованием равно (=), соединив их оператором OR.

```
SELECT ProductID,
       ProductName,
       Price
FROM TestTable
WHERE Price = 50 OR Price = 100
```

В данном случае мы говорим SQL серверу, что нам нужны записи, цена которых равна 50 или 100. В данном случае логичней конечно использовать оператор IN, и запись короче, и наше условие все-таки подразумевает перечисление. OR используют, когда в операндах участвуют разные столбцы, т.е., к примеру, нам нужны товары с ценой 50 или с определенным наименованием.

NULL значения

В базе данных, в таблицах, существует такое значение как – NULL. Это значение очень важно, точнее важно понимание, что это за значение.

NULL значение – это неопределённое значение, иными словами, значения как такового нет. Это не пусто и не 0, так как пусто и 0 сами по себе являются значениями, а вот если значения нет – это NULL.

NULL это на самом деле плохо, так как такие значения, если они есть в таблицах, нужно учитывать в своих запросах и приложениях. Их не получится сравнить с чем-то, так как значения нет, только также с NULL.

Например, если в нашей тестовой таблице TestTable 100 строк, при этом в столбце Price у 20 строк есть значения NULL, то если мы укажем условие Price > 10, то SQL сервер нам вернет только записи соответствующие условию, т.е. цена должна быть больше 10, строки с NULL обрабатываться вообще не будут. А если у нас более сложная аналитика, то значения NULL могут привести просто к непредсказуемым результатам. Поэтому при проектировании, как таблицы, так и последующих запросов, нужно понимать, может ли в том или ином столбце быть значения NULL, и если могут, то данный факт обязательно нужно учитывать.

Для проверки наличия значений NULL в стандарте языка SQL есть ключевые слова IS NULL и IS NOT NULL, которые можно использовать, например, в условии WHERE.

Если Вам необходимо обработать все записи, в которых есть нормальные значения, т.е. отсутствует NULL, например, вывести все строки из таблицы TestTable, в которых указана цена в столбце Price, можно написать следующий запрос.

```
SELECT ProductID,
       ProductName,
       Price
FROM TestTable
WHERE Price IS NOT NULL
```

Условие Price IS NOT NULL говорит нам о том, что мы хотим обработать только записи, в которых есть значения отличные от NULL.

Еще раз напоминаю, что при написании запросов помните о том, что в источнике данных могут быть значения NULL, которые могут сформировать непредсказуемый результирующий набор данных, это знание, возможно, предотвратит проблемы, которые потенциально могут возникнуть.

Группировка – GROUP BY

И вот мы подошли к уже более серьезным вещам, к таким как: группировка данных и использование агрегирующих функций (*или статистические функции*).

В целом функции мы будем разбирать чуть позже в соответствующем разделе, но, забегая вперед, я скажу, что функция в T-SQL - это действие, операция, вычисление или просто расчет, который зависит от определенных параметров или условий, которые влияют на это действие. Другими словами, функция - это некая мини программа в SQL сервере (*хотя алгоритмы функции могут быть очень сложными!*), которая выполняет какие-то определенные действия.

Агрегирующие функции – это функции, которые выполняют агрегацию данных, некую статистическую операцию на наборе данных (*над строками*) и возвращают одиночное итоговое значение.

Статистические операции - это, например, получение суммы всех строк по определённому столбцу, получение количества строк, а также определение максимального, минимального или среднего значения по столбцу среди всех строк, конечно же, все это можно делать с учетом условий.

По опыту могу сказать, что группировку и выполнение анализа данных с помощью агрегатных функций Вы будете использовать достаточно часто при написании запросов на выборку, так как такие сведения, как количество товаров по определённому признаку, средняя цена, сумма чего-либо и много других обобщенных сведений в работе любого предприятия требуется чуть ли не каждый день!

Как Вы понимаете, обычно сумма или количество, т.е. обобщенные данные, требуются не полностью по источнику, хотя это тоже бывает нужно, а с учетом условий, разбивки на категории, группы. Чтобы стало более понятно, давайте представим, что нам нужно узнать количество товаров в каждой из категорий и вывести все это в итоговом наборе данных. Нам для этого нужно подсчитать количество товаров с учетом группировки по каждой категории. В T-SQL для группировки данных используется отдельная секция инструкции SELECT - **GROUP BY**.

Но прежде чем переходить к секции GROUP BY, давайте сначала познакомимся с некоторыми агрегирующими функциями, которые Вы обязательно будете использовать в своих запросах.

- COUNT() – вычисляет количество значений в столбце *(значения NULL не учитываются*). Если написать COUNT(*), то будут учитываться все записи, т.е. все строки;
- SUM() – суммирует значения в столбце;
- MAX() – определяет максимальное значение в столбце;
- MIN() – определяет минимальное значение в столбце;
- AVG() – определяет среднее значение в столбце.

Для примера давайте напишем запрос и используем сразу все эти функции, и посмотрим на результат (Рис. 35).

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

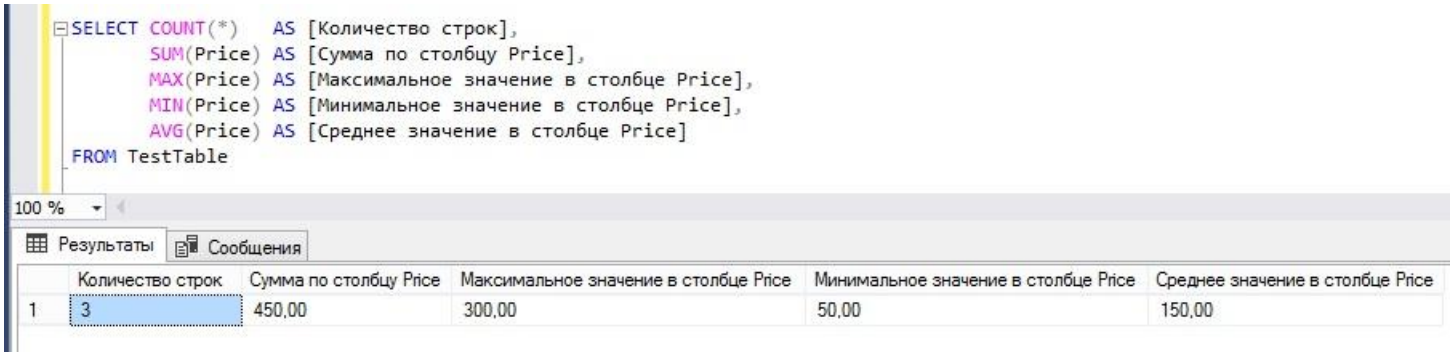


Рис. 35

Вы видите, в результирующем наборе всего одна строка. Она содержит обобщенные статистические данные, при этом эти данные обработаны на основе всей таблицы, т.е. никаких условий мы не ставили. Также Вы видите, что я здесь для наглядности использовал псевдонимы итоговых столбцов, если бы я этого не сделал, то столбцы в результирующем наборе были без названия.

А теперь давайте представим, что нам нужно все то же самое, только с группировкой по категориям. Другими словами, мы хотим знать количество товаров в определенной категории, максимальную, минимальную и среднюю цену товаров в каждой из категорий (*сумму я уберу, а то как-то не логично суммировать цену*). Мы для этого используем предложение GROUP BY, результат представлен на рисунке 36.

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

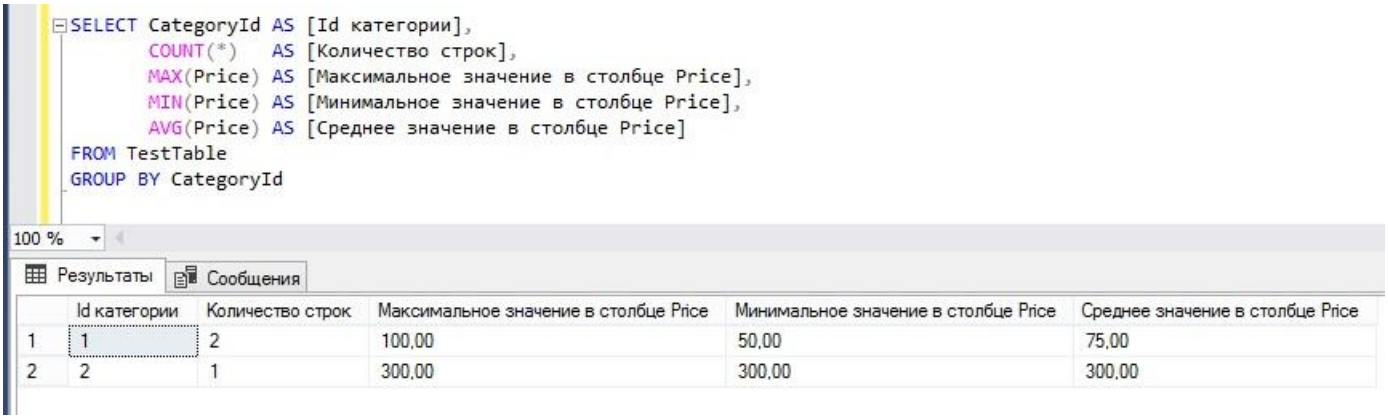


Рис. 36

В списке выборки мы указали столбец CategoryId, для того чтобы включить его в результирующий набор данных, после секции FROM мы указали секцию GROUP BY, в которой написали столбец, по которому группировать данные. Напомню, что синтаксически важно, чтобы секции шли в определённом порядке, данный порядок я указал в разделе «*Описание инструкции SELECT*», поэтому если Вы захотите добавить еще условие в данный запрос, то оно должно быть после секции FROM и перед секцией GROUP BY.

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

В данном примере мы добавили условие, которое говорит SQL серверу, что нам нужны статистические данные всей таблицы TestTable, за исключением товара с идентификатором 1, т.е. ProductId не равен 1.

Условия – HAVING

Группировку данных мы разобрали, ставить условие на выборку данных мы научились, но что делать, если нам нужно поставить условие на агрегированные данные, т.е. на статистическое выражение?

Например, нам нужны категории товаров, в которых присутствуют больше 1 товара. Чтобы это реализовать, мы должны сначала узнать, сколько товаров в той или иной категории, а затем отсечь категории, в которых 1 или вообще нет товаров.

В T-SQL все это можно реализовать с помощью уже известной нам агрегатной функции COUNT, группировки данных GROUP BY и секции HAVING.

HAVING – это некое условие на результат агрегатной функции. Данную секцию мы можем использовать только в сочетании с секцией GROUP BY. Иными словами, чтобы поставить условие на сгруппированные данные, мы должны их сначала сгруппировать.

```
SELECT CategoryId AS [Id категории],
       COUNT(*)   AS [Количество строк]
FROM TestTable
GROUP BY CategoryId
HAVING COUNT(*) > 1
```

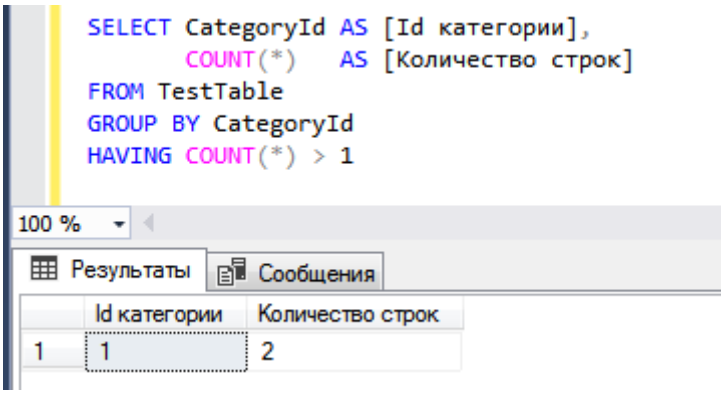


Рис. 37

В данном примере мы как раз и выполнили запрос, который вернул нам список категорий, в которых больше одного товара. Как видите, секция HAVING указывается после секции GROUP BY. После слова HAVING мы указали предикат с использованием статистической функции.

Проводить анализ данных мы научились, идем дальше!

Сортировка - ORDER BY

Если Вы выполняете все запросы, которые указаны в данной книге, то Вы, наверно, заметили, что данные возвращаются в определённом порядке, а именно в том порядке, в котором их хранит SQL сервер. Но на практике чаще требуется получить результирующий набор данных в отсортированном

виде, например, по возрастанию цены, по определённой категории, признаку или характеристике. SQL сервер, конечно же, позволяет пользователю применить свою сортировку, отличную от той, которая используется по умолчанию.

Для этого существует секция **ORDER BY**, в которой можно указать названия столбцов, по которым сортировать данные, или просто указать их порядковый номер в списке выборки, но мне больше нравится указывать название столбца (*и Вам я рекомендую так делать*), так более наглядно, но указание номера столбца иногда более удобно.

```
SELECT ProductID,
       ProductName,
       Price
FROM TestTable
ORDER BY Price
```

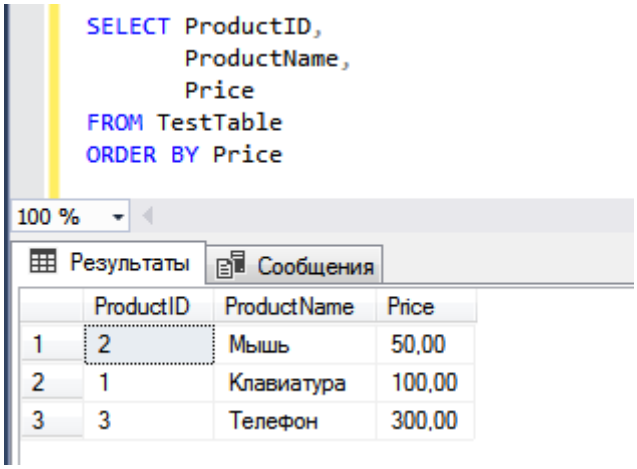


Рис. 38

В данном случае мы получили итоговый набор данных в отсортированном виде, а именно мы отсортировали данные по возрастанию цены. Секцию ORDER BY мы указали в самом конце, так как сортировка данных - это последняя операция над итоговыми данными.

По умолчанию используется сортировка «*По возрастанию*», но если Вам необходимо отсортировать данные «*По убыванию*», то для этого нужно после названия столбца указать ключевое слово DESC

```
SELECT ProductID,
       ProductName,
       Price
FROM TestTable
ORDER BY Price DESC
```

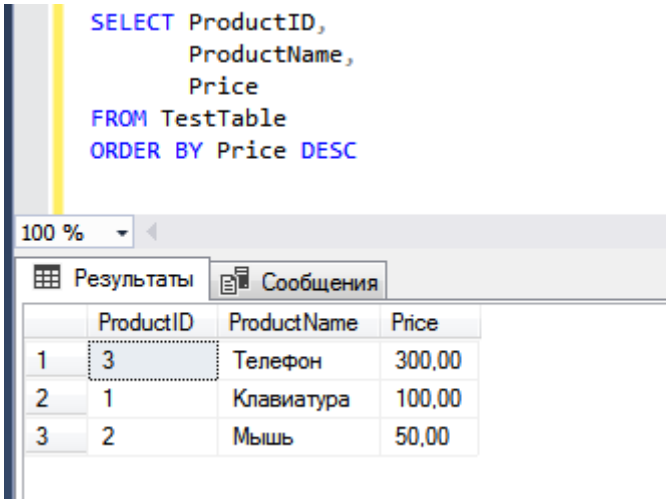


Рис. 39

Если Вам необходимо, чтобы сортировка была сначала по одному столбцу, а потом, в случае одинаковых значений, по другому столбцу, то в ORDER BY можно указать несколько столбцов для сортировки через запятую, SQL сервер будет применять сортировку в порядке указания этих столбцов.

```
SELECT ProductID,
       ProductName,
       Price
FROM TestTable
ORDER BY Price DESC, ProductID
```

Сейчас мы сказали SQL серверу, что нам нужно отсортировать данные по убыванию цены, а потом по возрастанию идентификатора товара.

Ровно то же самое мы получим, если напишем следующий запрос, правда, в нем мы используем порядковые номера столбцов.

```
SELECT ProductID,
       ProductName,
       Price
FROM TestTable
ORDER BY 3 DESC, 1
```

В Microsoft SQL Server 2012 возможности ORDER BY были расширены, а именно добавилась инструкция **OFFSET-FETCH**, которая выбирает и, соответственно, оставляет в результирующем наборе только часть строк, которые нам необходимы в конкретный момент времени. Ее работа похожа на работу оператора TOP, который мы уже с Вами рассмотрели, однако инструкция OFFSET-FETCH имеет больше возможностей, например, мы можем пропускать определённое количество строк.

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

```
SELECT ProductID,
       ProductName,
       Price
FROM TestTable
ORDER BY Price DESC
OFFSET 1 ROWS
```

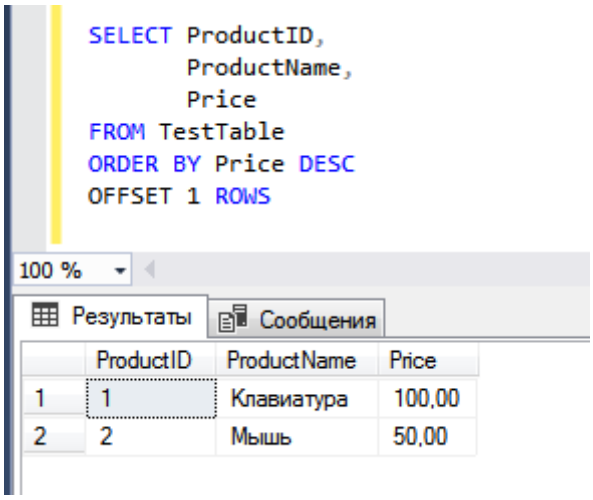


Рис. 40

Мы видим, что строка с ценой 300 была пропущена, так как после сортировки она у нас была первая.

С помощью OFFSET-FETCH можно легко реализовать, например, постраничную выборку, как раз для этого нам и необходимо ключевое слово FETCH.

```
SELECT ProductID,
       ProductName,
       Price
FROM TestTable
ORDER BY Price DESC
OFFSET 1 ROWS FETCH NEXT 1 ROWS ONLY
```

В данном случае первая строка также будет пропущена, но нам вернётся только одна следующая строка, так как мы указали инструкцию **FETCH NEXT 1 ROWS ONLY**. Как Вы понимаете значение 1 можно изменять на нужное Вам.

Такие параметры как PERCENT и WITH TIES, которые есть у TOP, инструкцией OFFSET-FETCH не поддерживаются, так же, как и не поддерживается совместная работа операторов TOP и OFFSET-FETCH в одном запросе SELECT.

Еще раз напоминаю, что OFFSET-FETCH без ORDER BY использовать не получится, что собственно и логично, ведь чтобы вернуть определенную часть результирующего набора, строки должны быть предварительно отсортированы.

Объединение JOIN

Сейчас мы будем с Вами рассматривать еще одну очень важную и полезную секцию инструкции SELECT, хотя все секции, конечно же, важные и нужные, но вот объединение данных из разных источников Вы также будете использовать постоянно. Если сортировку можно, к примеру, и не делать (*например, делать ее уже в графическом отчете или если это выгрузка непосредственно в Excel*), то объединить данные из нескольких таблиц в приложении или в том же Excel уже будет проблематично.

Для чего, Вы спросите, нужно объединять данные? Запросы, т.е. выборки, которые требует руководство, содержат в себе не только данные из одной таблицы, а вполне возможно из многих, ведь руководству абсолютно без разницы, где у нас лежат данные и как хранятся. Да и приложения устроены так, что выводят данные пользователю в удобном формате и с удобной структурой. А если Вы еще вспомните нормализацию базы данных, про то, что каждая сущность хранится в отдельной таблице, то Вы окончательно поймете, что объединять данные, как только это возможно, Вы будете ежедневно!

Как Вы понимаете, для того чтобы объединить данные, нужно знать, по какому ключу их объединять, без точного ключа объединение будет также не точным.

Для объединения данных используется секция **JOIN**, при этом видов объединения несколько и соответственно JOIN-ов тоже несколько.

INNER

Это внутреннее объединение, при котором в результирующий набор данных попадут все записи, которые полностью соответствуют связи, т.е. общим столбцам двух источников.

Чтобы не откладывать, давайте сразу посмотрим на пример, если Вы помните, то мы создали две таблицы, в первой TestTable у нас хранятся товары, а во второй TestTable2 категории с указанием их названия. Давайте их объединим так, чтобы в результирующий набор данных попали записи с названием категории.

```
SELECT T1.ProductName,
       T2.CategoryName,
       T1.Price
FROM TestTable T1
INNER JOIN TestTable2 T2 ON T1.CategoryId = T2.CategoryId
ORDER BY T1.CategoryId
```

Отмечаю, что секция JOIN идет сразу после секции FROM, т.е. мы указываем источник, а затем сразу его объединяем с другим источником.

Перед словом JOIN мы указали слово INNER, которое говорит SQL серверу, что объединение будет внутреннее, т.е. по полному совпадению.

Ключевое слово ON задает условие объединения, т.е. именно здесь Вы указываете, как будет происходить объединение, по каким столбцам или выражениям. Это условие так же, как и в секции WHERE может состоять из нескольких предикатов, объединённых оператором AND или OR.

В данном случае столбец, по которому мы объединяем таблицы - это идентификатор категории (*CategoryId*), так как он есть в обоих таблицах.

Посмотрим на результат (Рис. 41).

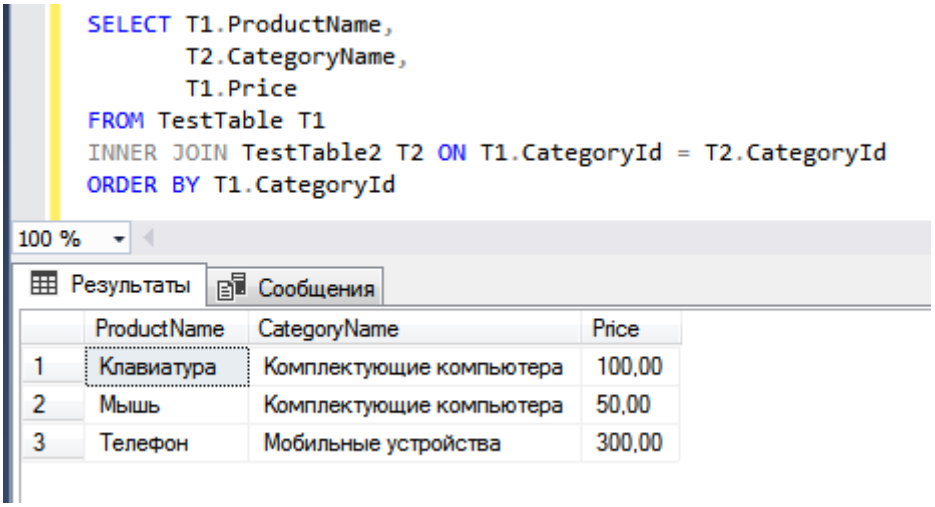


Рис. 41

У нас вывелись все записи из таблицы TestTable и TestTable2, в которых значение столбца CategoryId равняется.

LEFT

Это левое внешнее объединение, при котором возвращаются все записи из таблицы слева, даже если записей в правой таблице не существует. В случаях, когда условие не выполняется, и в правой таблице нет соответствующих записей, в результирующий набор данных от правой таблицы попадают строки со значением NULL. Другими словами, Если Вам нужно получить все записи одной таблицы и, если есть, соответствующие записи в другой, то Вы можете использовать объединение LEFT. Если в таком случае использовать INNER, то Вам вернуться только записи, которые объединены в соответствии с условием, т.е. если нет соответствующих записей в таблице справа, то записи справа тоже не вернутся.

Синтаксически объединение будет выглядеть точно также как в запросе выше, только вместо INNER пишем LEFT.

```
SELECT T1.ProductName,
       T2.CategoryName,
       T1.Price
FROM TestTable T1
LEFT JOIN TestTable2 T2 ON T1.CategoryId = T2.CategoryId
ORDER BY T1.CategoryId
```

В данном случае у нас получился точно такой же результат, так как у нас по факту все записи из левой таблицы по данному условию объединились с записями в правой таблице.

Но для того чтобы показать Вам, что было бы, если условие не выполнялось, давайте напишем два запроса, в которых я покажу отличие в работе INNER от LEFT.

Первый запрос будет с использованием объединения INNER, второй с использованием LEFT. В обоих запросах я укажу несуществующую категорию, например, 3 (*в таблице TestTable2 такой записи нет*). Эти запросы наглядно покажут отличие INNER от LEFT (Рис. 42).

```
SELECT T1.ProductName,
       T2.CategoryName,
       T1.Price
FROM TestTable T1
INNER JOIN TestTable2 T2 ON T1.CategoryId = 3
ORDER BY T1.CategoryId

SELECT T1.ProductName,
       T2.CategoryName,
       T1.Price
FROM TestTable T1
LEFT JOIN TestTable2 T2 ON T1.CategoryId = 3
ORDER BY T1.CategoryId
```

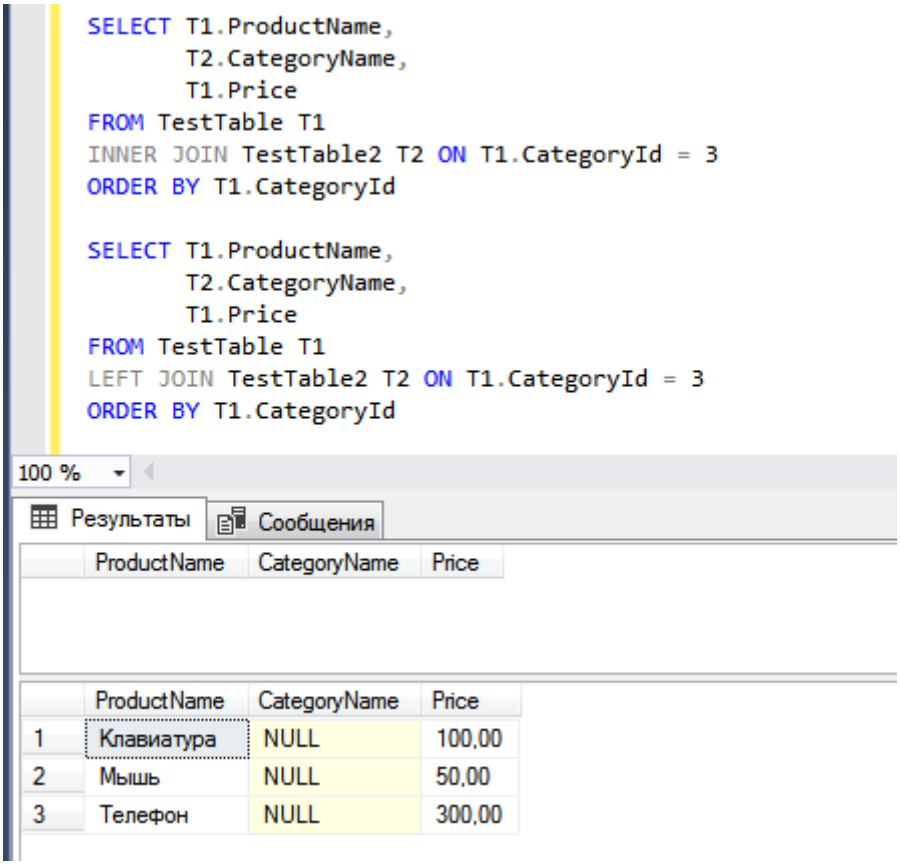


Рис. 42

В первом случае, как видите, не вернулось не одной строки, так как не по одной строке условие не выполнилось. Во втором случае вернулись все записи из таблицы TestTable2 *(т.е. из левой таблицы)*, а данные, которые должны были быть из правой таблицы, заполнились значениями NULL, так как, снова повторюсь, условие не выполнилось.

Теперь я думаю, стало более-менее понятно главное отличие объединения INNER от LEFT. На практике, лично мне, очень часто приходилось использовать именно эти виды объединения, остальные, конечно, я тоже использую, но гораздо меньше. Поэтому я советую в первую очередь понять, как работают именно эти объединения, остальные виды объединений также нужно знать, поэтому давайте продолжать!

RIGHT

Это правое внешнее объединение, работает точно так же, как и LEFT, только для правой таблицы. Иными словами, возвращаются все записи из правой таблицы и соответствующие записи из левой. Если записи в левой таблице отсутствуют, то их значения в результирующем наборе заполняется значениями NULL (*т.е. так же, как и в LEFT*).

```
SELECT T1.ProductName,
       T2.CategoryName,
       T1.Price
FROM TestTable T1
RIGHT JOIN TestTable2 T2 ON T1.CategoryId = T2.CategoryId
ORDER BY T1.CategoryId
```

В этом случае результат будет таким же, так как у нас все записи из таблицы TestTable2 задействованы (*по CategoryId*) в таблице TestTable.

Если мы укажем несуществующую категорию, например, 3 как в примере, который я использовал, для того чтобы показать отличия в работе объединения INNER от LEFT, то получим следующий результат (Рис. 43).

```
SELECT T1.ProductName,
       T2.CategoryName,
       T1.Price
FROM TestTable T1
RIGHT JOIN TestTable2 T2 ON T1.CategoryId = 3
ORDER BY T1.CategoryId
```

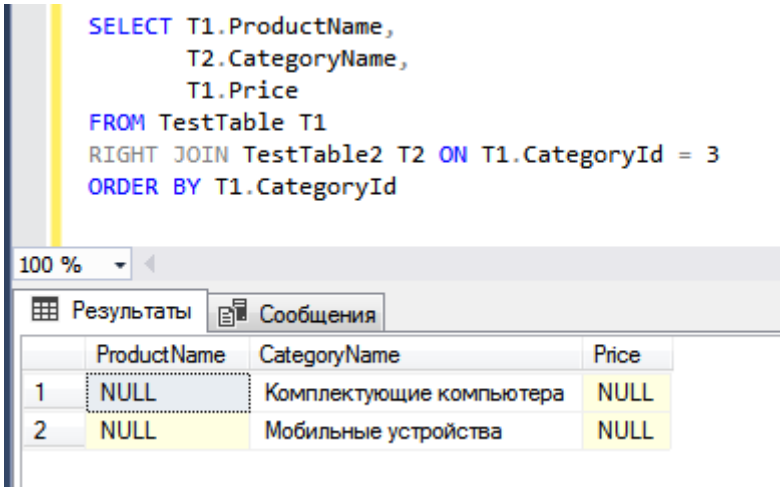


Рис. 43

Записи из таблицы TestTable2 нам вернулись, но так как соответствующих записей в таблице TestTable нет, то и больше нам ничего не вернулось, а столбцы, данные которых мы хотели видеть из таблицы TestTable, заполнились значением NULL.

Самое главное при любом объединении - понимать, какие данные Вы хотите получить в итоговом наборе, и какие данные у Вас хранит тот или иной источник (*таблица*), а также какую они имеют связь (*т.е. общие столбцы*), так как если Вы не понимаете, что Вам нужно и что с чем необходимо для этого объединить, Вы вряд ли сможете получить то, что хотели.

Я всегда вспоминаю одного человека, который со мной работал, он мог использовать запрос, который ему когда-то дали со словами, что он возвращает такие-то данные, допустим, «*Список товаров*», совершенно для других целей, для которых он не предназначен, а потом жаловаться на результат, говоря, что запрос возвращает некорректные данные. При этом, объясняя, «*Что этот запрос возвращает же список товаров, а мне и нужен список товаров с указанием категории, а он по некоторым строкам возвращает товары без указания категории*». Иными словами, он даже не понимал, что именно за запрос он выполняет, хотя он элементарный с использованием простого объединения (*не сложней тех, которые мы используем в наших примерах*), если бы он знал, как именно работает то или иной объединение, то он сам без труда догадался исправить запрос, наверное, даже до первого его запуска. Именно поэтому важно понимать, как работает объединение в SQL.

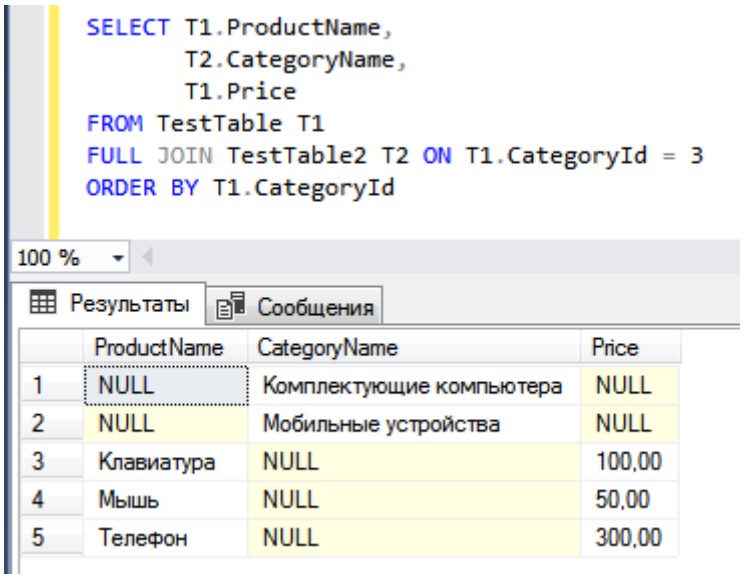
FULL

Полное внешнее объединение, в данном случае возвращаются все записи, как из левой, так и из правой таблицы, даже если условие не выполняется. Все записи, для которых не нашлось совпадения в

соседней таблице, заполняются значениями NULL. Это своего рода объединение и LEFT, и RIGHT, которое применили в одном объединении.

Для наглядности давайте выполним все тот же запрос (Рис. 43) с указанием категории 3 и посмотрим, что получится на этот раз.

```
SELECT T1.ProductName,
       T2.CategoryName,
       T1.Price
FROM TestTable T1
FULL JOIN TestTable2 T2 ON T1.CategoryId = 3
ORDER BY T1.CategoryId
```



	ProductName	CategoryName	Price
1	NULL	Комплектующие компьютера	NULL
2	NULL	Мобильные устройства	NULL
3	Клавиатура	NULL	100,00
4	Мышь	NULL	50,00
5	Телефон	NULL	300,00

Рис. 44

Мы видим, что нам вернулись как будто данные из двух запросов с объединением LEFT и RIGHT. Честно сказать, данный вид объединения мне практически не требовался, а вот следующий требовался часто, поэтому давайте переходить к нему.

CROSS

Это перекрестное объединение, которое выполняет декартово произведение всех строк таблиц, участвующих в объединении. С первого взгляда, если посмотреть на работу данного объединения, т.е. на результирующий набор данных, скорей всего будет непонятно, что это за объединение, как оно работает, Вам покажется, что оно очень сложное.

Но на самом деле это не так, принцип его работы очень прост, каждая строка из одной таблицы объединяется со всеми строками в другой таблице. Таким образом, происходит перемножение количества строк в этих таблицах, в итоге, результирующий набор будет содержать как раз количество записей, которое будет равняться произведению количества строк в первой таблице на количество строк во второй таблице (Рис. 45).

```
SELECT T1.ProductName,
       T2.CategoryName,
       T1.Price
FROM TestTable T1
CROSS JOIN TestTable2 T2
ORDER BY T1.CategoryId
```

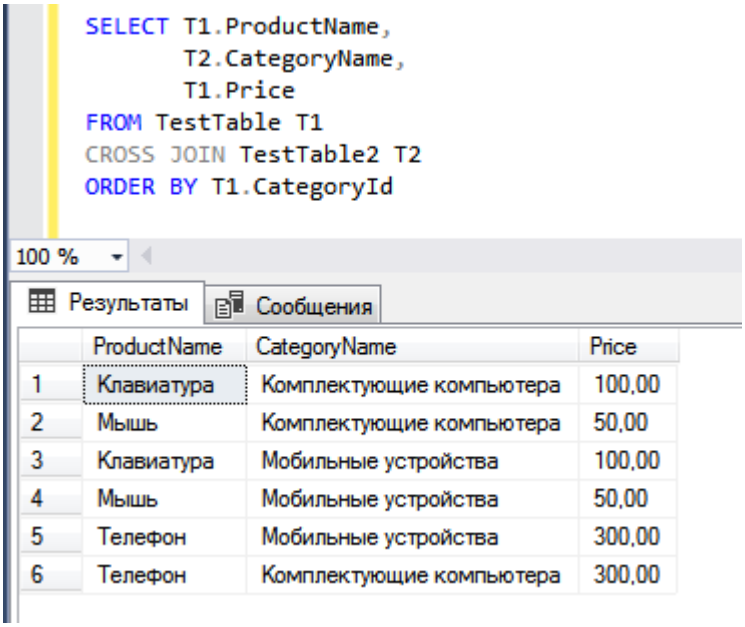


Рис. 45

Однако есть одна особенность, она заключается в том, что, если мы добавим условие WHERE, например, такое же, которое мы использовали в примере с INNER JOIN после ключевого слова ON, кстати, в CROSS JOIN указывать данное ключевое слово, как видите, из предыдущего примера не требуется, мы получим точно такой же результат, как и в случае с INNER JOIN. Другими словами, в таких случаях CROSS JOIN будет вести себя как внутреннее объединение. На это мало кто обращает внимание, так как это действительно редко требуется.

```
SELECT T1.ProductName,
       T2.CategoryName,
       T1.Price
FROM TestTable T1
CROSS JOIN TestTable2 T2
WHERE T1.CategoryId = T2.CategoryId
ORDER BY T1.CategoryId
```

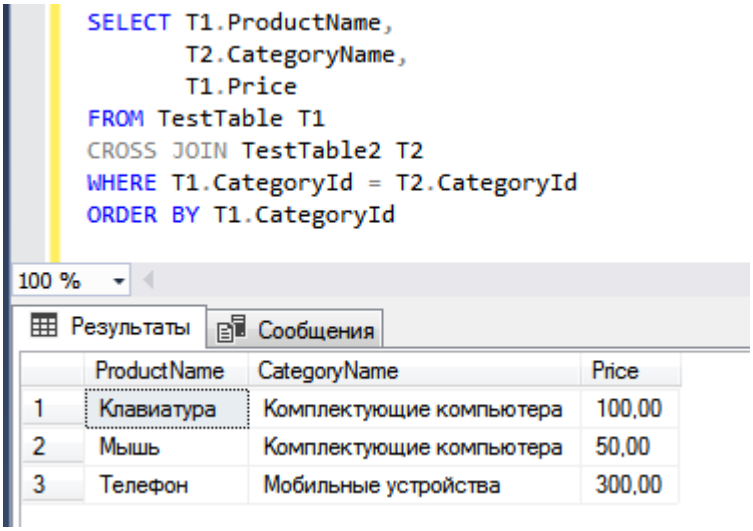


Рис. 46

Объединение UNION

Мы с Вами рассмотрели виды объединений, которые формируют результирующий набор данных за счет объединения таблиц в одной инструкции SELECT, однако существует возможность объединять результаты нескольких запросов, т.е. нескольких SELECT-ов, иными словами, несколько множеств объединять в один итоговый набор данных.

Давайте сразу посмотрим на пример (Рис. 47).

```
SELECT T1.ProductId,
       T1.ProductName,
       T1.Price
FROM TestTable T1
WHERE T1.ProductId = 1

UNION

SELECT T1.ProductId,
       T1.ProductName,
       T1.Price
FROM TestTable T1
WHERE T1.ProductId = 3
```

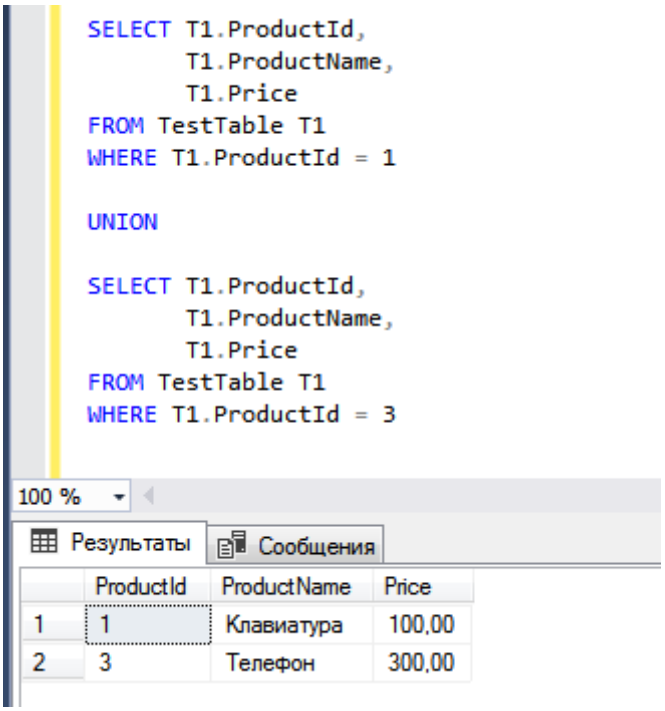


Рис. 47

Мы видим два по сути отдельных запроса, между ними (*т.е. сразу после первого*) указано ключевое слово UNION, которое говорит SQL серверу, что эти запросы нужно объединить в один результирующий набор данных. В итоге мы и получили один общий набор данных, т.е. все данные, которые были после UNION, добавились к данным, которые возвращает первый запрос. Во время этого объединения сами данные никак не модифицируются. Но обязательно следует знать, что UNION возвращает только уникальные объединения строк, т.е. если во втором запросе будет абсолютно одинаковая строка, как и в первом, в результирующий набор попадет только одна строка, а не две. Бывают случаи, сразу скажу достаточно часто, что необходимо включить в результирующий набор все строки, как из первого запроса, так и из второго, для этого в SQL сервер предусмотрено объединение UNION ALL, т.е. дополнительное указание ключевого слова ALL. Оно будет говорить SQL серверу, что в результирующий набор данных нужно включить абсолютно все строки. Давайте посмотрим на примеры, которые показывают различия работы UNION и UNION ALL.

Пример, в котором мы используем UNION.

```
SELECT T1.ProductId,
       T1.ProductName,
       T1.Price
FROM TestTable T1
WHERE T1.ProductId = 1

UNION

SELECT T1.ProductId,
       T1.ProductName,
       T1.Price
FROM TestTable T1
     WHERE T1.ProductId = 1
```

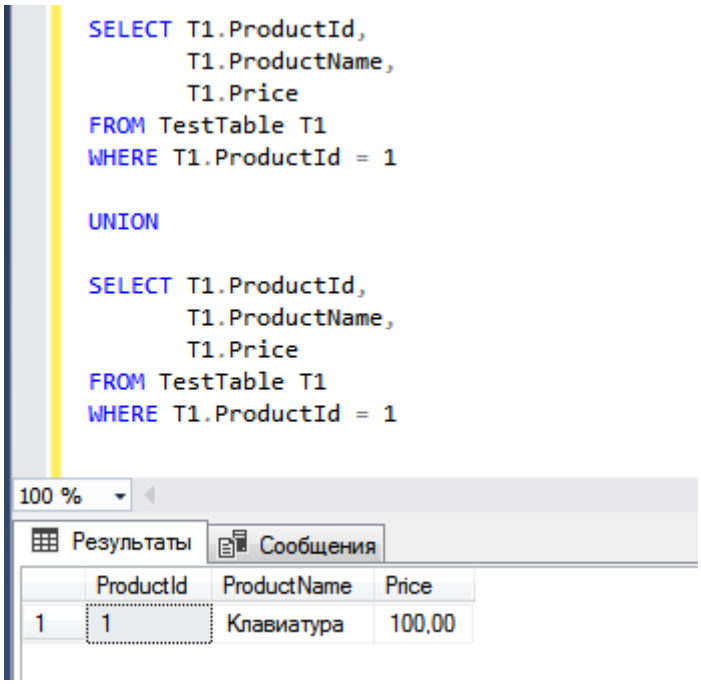


Рис. 48

Пример, в котором мы используем UNION ALL

```
SELECT T1.ProductId,
       T1.ProductName,
       T1.Price
FROM TestTable T1
WHERE T1.ProductId = 1

UNION ALL

SELECT T1.ProductId,
       T1.ProductName,
       T1.Price
FROM TestTable T1
     WHERE T1.ProductId = 1
```

```
SELECT T1.ProductId,
       T1.ProductName,
       T1.Price
FROM TestTable T1
WHERE T1.ProductId = 1

UNION ALL

SELECT T1.ProductId,
       T1.ProductName,
       T1.Price
FROM TestTable T1
WHERE T1.ProductId = 1
```

100 %

Результаты | Сообщения

	ProductId	ProductName	Price
1	1	Клавиатура	100,00
2	1	Клавиатура	100,00

Рис. 49

Мы видим, что в первом случае у нас вернулась всего одна строка, а во втором две одинаковые строки, так как в первом случае мы использовали UNION, т.е. нам нужны были только различающиеся записи, во втором UNION ALL, которое подразумевает добавление в результирующий набор всех строк из двух запросов.

Также хотелось бы отметить, что с помощью объединения UNION и UNION ALL можно объединять не два запроса, а несколько, столько, сколько Вам нужно.

Оба этих объединения требуются достаточно часто, поэтому как они работают, Вам также нужно понять и запомнить. Для того чтобы понимать, как они работают, Вы обязательно должны знать несколько важных правил и условий, без знания которых у Вас не получится использовать эти объединения на практике. Эти правила заключаются в следующем:

- Количество и порядок столбцов во всех запросах должен быть строго одинаковый;
- Тип данных столбцов во всех запросах должен быть совместимым;
- Секцию ORDER BY необходимо указывать после всех запросов, которые объединяются с помощью UNION и UNION ALL.

Объединение INTERSECT и EXCEPT

В T-SQL существуют еще объединения, которые напоминают UNION тем, что работают с несколькими множествами, т.е. запросами - это **INTERSECT и EXCEPT**. Их работа, конечно же, отличается от UNION, и сейчас я Вам расскажу, в чем именно.

Оператор INTERSECT

INTERSECT (*пересечение*) – данный оператор выводит одинаковые строки из первого, второго и последующих наборов данных. Другими словами, он выведет только те строки, которые есть как в первом результирующем наборе, так и во втором (*третьем и так далее*), то есть происходит пересечение этих строк.

Данный оператор будет очень полезен, например, когда необходимо узнать, какие строки есть и в первой таблице и во второй, допустим, нужно определить, есть ли в разных таблицах повторяющиеся записи.

В следующем примере мы используем тот же самый запрос, который мы использовали ранее, только вместо UNION напишем INTERSECT.

```
SELECT T1.ProductId,
       T1.ProductName,
       T1.Price
FROM TestTable T1
WHERE T1.ProductId = 1

INTERSECT

SELECT T1.ProductId,
       T1.ProductName,
       T1.Price
FROM TestTable T1
     WHERE T1.ProductId = 1
```

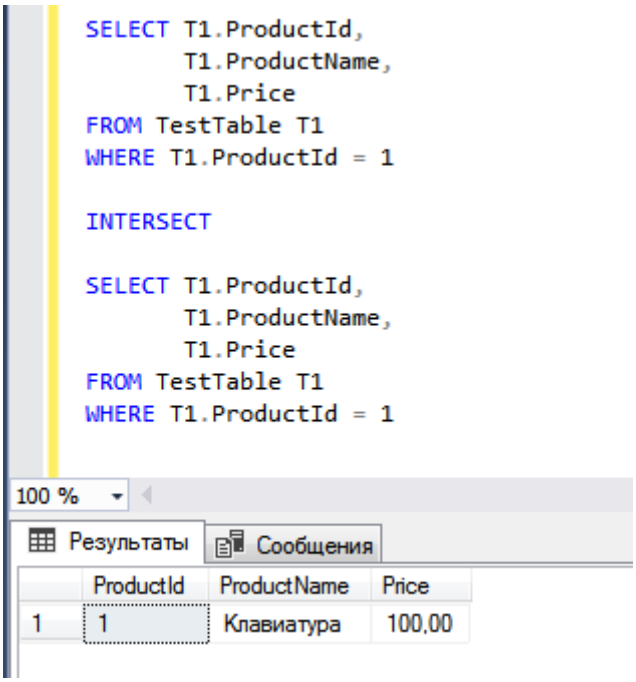


Рис. 50

В итоге в данном случае результат (Рис. 50) у нас такой же, как и с UNION, однако алгоритм работы совершенно другой. В случае с UNION вывелась уникальная строка, в случае с INTERSECT строка, которая есть и в первом запросе и во втором. Если бы во втором запросе точно такой же строки не было, то в результирующем наборе данных у нас ничего не вывелось, а с UNION вывелось бы.

Давайте, для примера, изменим условие во втором запросе, и посмотрим на разницу.

Пример с использованием UNION

```
SELECT T1.ProductId,
       T1.ProductName,
       T1.Price
FROM TestTable T1
WHERE T1.ProductId = 1

UNION

SELECT T1.ProductId,
       T1.ProductName,
       T1.Price
FROM TestTable T1
WHERE T1.ProductId = 2
```

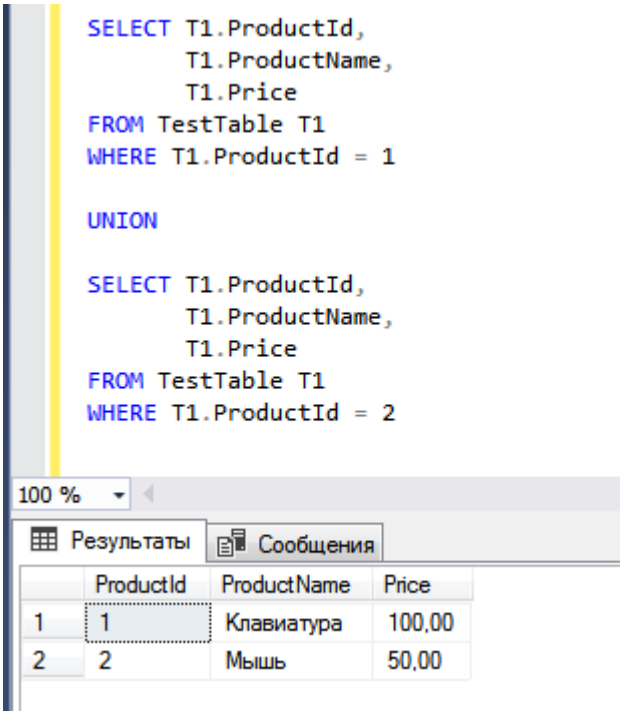


Рис. 51

Пример с использованием INTERSECT

```
SELECT T1.ProductId,
      T1.ProductName,
      T1.Price
FROM TestTable T1
WHERE T1.ProductId = 1

INTERSECT

SELECT T1.ProductId,
       T1.ProductName,
       T1.Price
FROM TestTable T1
WHERE T1.ProductId = 2
```

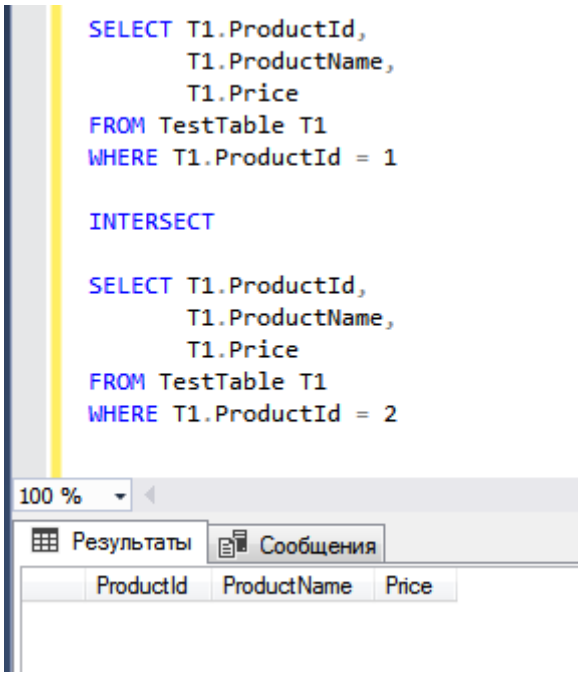


Рис. 52

Как видим, в примере с использованием INTERSECT результат пустой, так как в наборе данных, который возвращает второй запрос, отсутствует запись с ProductId = 1, там только ProductId = 2.

Оператор EXCEPT

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

Пример.

```
SELECT T1.ProductId,
       T1.ProductName,
       T1.Price
FROM TestTable T1
WHERE T1.ProductId = 1

EXCEPT

SELECT T1.ProductId,
       T1.ProductName,
       T1.Price
FROM TestTable T1
     WHERE T1.ProductId = 2
```

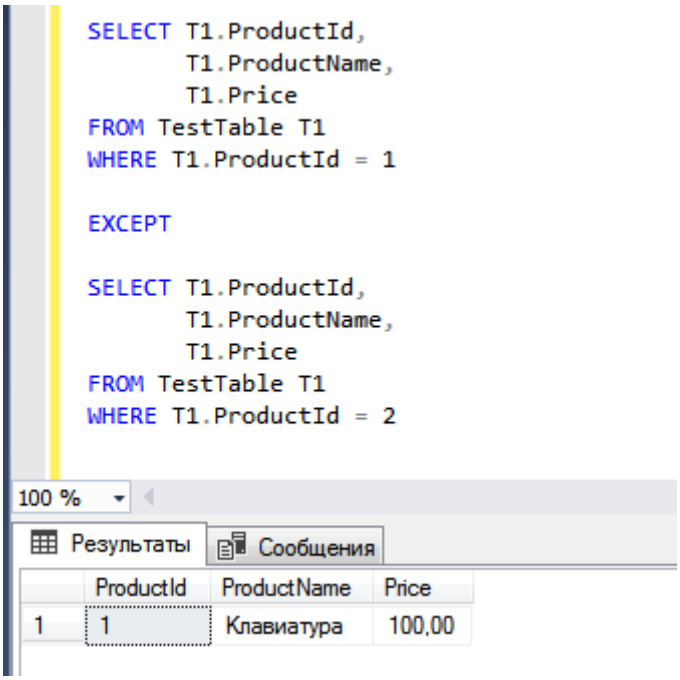


Рис. 53

В этом примере мы заменили INTERSECT на EXCEPT, и теперь у нас выведется одна строка с ProductId = 1 (Рис. 53), так как во втором запросе отсутствует строка с ProductId = 1 из первого запроса.

Все правила и условия, которые необходимо соблюдать при использовании объединения UNION, мы их рассмотрели чуть ранее, относятся также и к операторам INTERSECT и EXCEPT (*количество, порядок и тип столбцов в запросах должен быть одинаковый*).

Про объединения, я думаю, пока достаточно. Если Вам что-то непонятно, советую сразу перечитать соответствующий раздел и продолжать чтение только после того, как у Вас усвоится полученная информация. В идеале Вам нужно пробовать самим писать простенькие запросы в дополнение к примерам, которые мы рассматриваем, тем самым Вы практикуетесь, и информация запоминается и усваивается намного лучше!

Главный фактор успеха на пути к освоению языка T-SQL – это практика!

Подзапросы (вложенные запросы)

Мы подошли к очень интересной возможности языка T-SQL – это подзапросы!

Достаточно часто бывает нужно во время выполнения запроса получить данные из другой таблицы, например, всего один столбец или, что еще чаще, какие-то расчетные данные (*например, статистические, агрегированные данные*). Для этого мы как раз и можем использовать подзапрос, это своего рода отдельный запрос, но находится и выполняется он внутри другого запроса. Его также называют вложенный запрос.

Подзапрос – это SELECT внутри другого SELECT-а!

Допустим, нам нужно получить все категории с указанием количества товаров, относящихся к данной категории, что-то подобное мы уже делали, когда рассматривали группировку данных, но там мы анализировали таблицу с товарами, но если у нас есть категория, в которой нет товаров, то в том запросе она, соответственно, не отобразится в итоговом наборе данных.

Поэтому давайте проведем анализ на основе таблицы с категориями, т.е. TestTable2. В запросе мы используем подзапрос для определения количества товаров, т.е. в подзапросе мы обратимся к другой таблице и подсчитаем количество товаров.

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

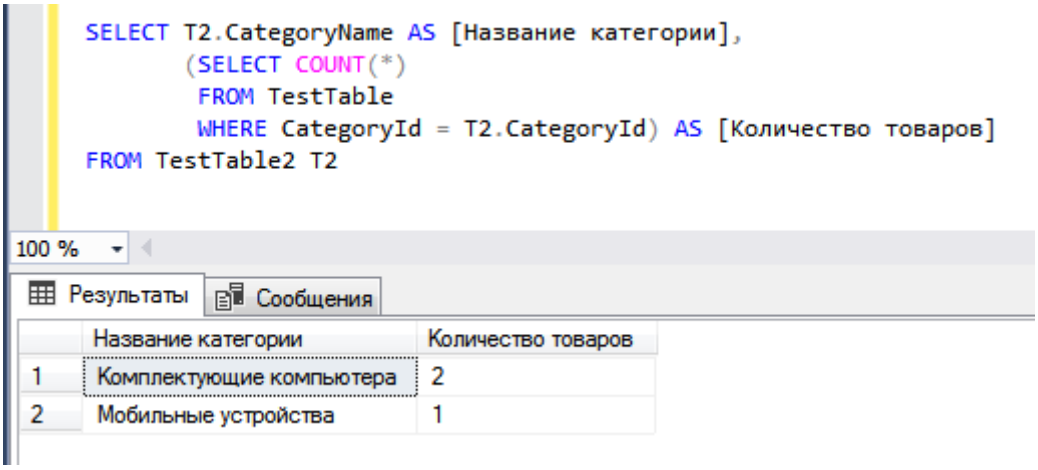


Рис. 54

Мы написали обычный запрос на выборку из таблицы TestTable2, которой задали псевдоним T2. В списке выборки мы указали два столбца, первый CategoryName из таблицы TestTable2, а в качестве второго столбца мы указали подзапрос, которой написали в скобочках () и ему задали псевдоним [Количество товаров]. В данном случае в подзапросе для определения количества товаров в той или иной категории мы использовали связь по столбцу - CategoryId (*идентификатор категории*), таким образом, мы подсчитывали количество товаров только в той категории, которая указана в строке основного запроса. Иными словами, мы передали столбец основного запроса внутрь подзапроса для объединения. Если бы мы этого не сделали (*можете поэкспериментировать, например, убрать условие WHERE CategoryId = T2.CategoryId*), нам вернулось бы одинаковое значение столбца [Количество товаров] во всех строках результирующего набора, так как для всех строк выполнился бы один и тот же подзапрос, т.е. без условий. В данном подзапросе мы использовали агрегатную функцию COUNT, для подсчёта количества товаров, но это Вы уже и сами догадались.

Важным моментом при использовании подзапросов является то, что в них нельзя использовать сортировку (ORDER BY), а также если вложенные запросы указаны в списке выборки (*как в нашем случае*), то они должны возвращать одиночное значение. Другими словами, если Вы подзапрос напишите таким образом, чтобы он возвращал два и более столбцов - будет ошибка!

Подзапросы можно использовать не только в списке выборки, но и в других секциях, например, в условиях, в качестве выражения, можно указать подзапрос, а также очень часто Вам потребуется использовать подзапрос как источник данных, т.е. в секции FROM или JOIN. В данных случаях в

подзапросе можно указывать несколько столбцов и такие подзапросы обычно называют - **Производные таблицы**.

Например, Вам нужно иметь в качестве источника данных в секции FROM не всю таблицу, а только несколько столбцов, для этого Вы можете указать подзапрос в секции FROM вместо указания таблицы.

```
SELECT ProductId, Price
FROM (SELECT ProductId,
             Price
      FROM TestTable) AS Q1
```

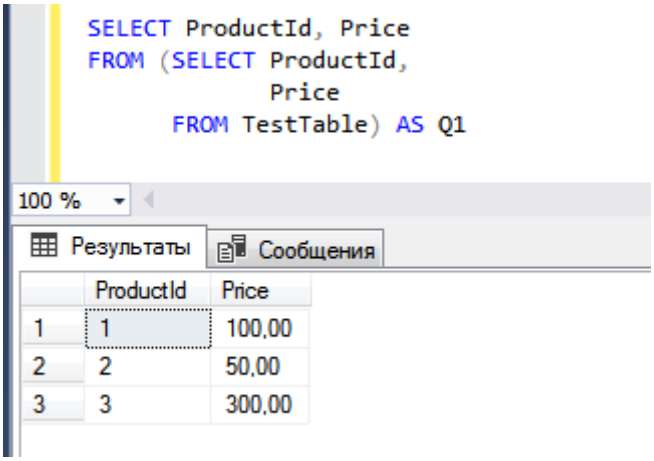


Рис. 55

После FROM в скобочках () мы написали подзапрос к таблице TestTable, в списке выборки которого указали 2 столбца, следует отметить, что в данном случае необходимо обязательно указывать псевдоним для подзапроса (*производной таблицы*).

Надеюсь, понятно, что вместо названия источника данных мы в скобочках пишем нужный нам запрос SELECT, который формирует производную таблицу. На практике, конечно же, в качестве подзапроса будет требоваться указание более сложных запросов.

Также в качестве источника можно указывать производную таблицу и в объединениях JOIN, например.

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

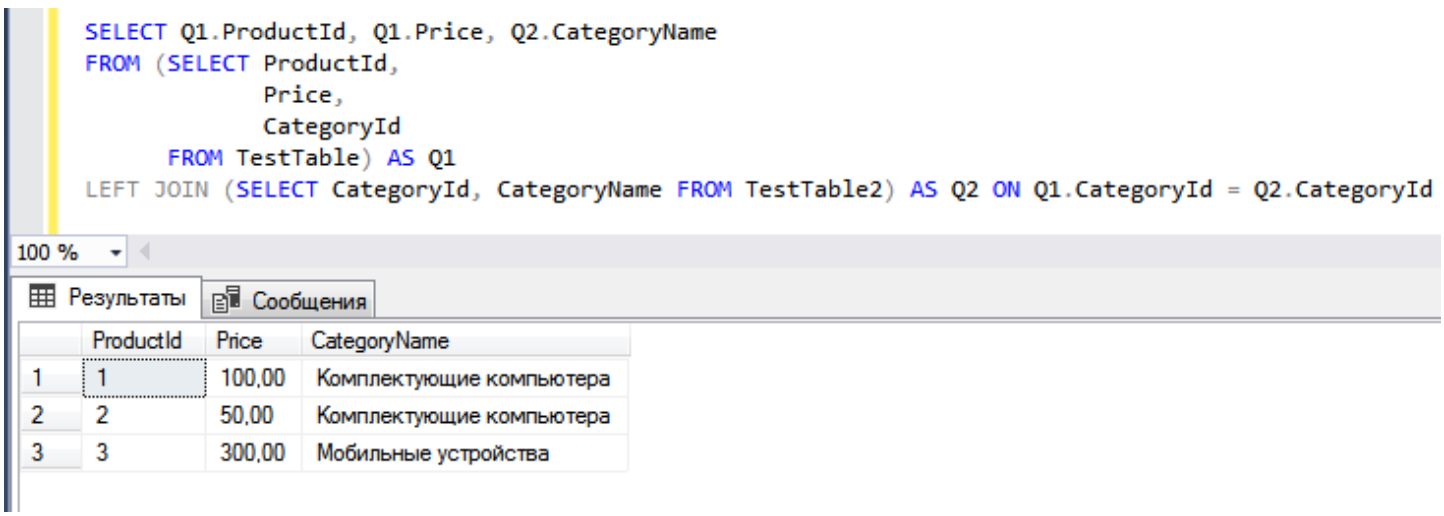


Рис. 56

Здесь мы с Вами написали запрос, в котором у нас и в секции FROM, и в секции JOIN используется запрос, т.е. производная таблица.

Хотя подзапросы и производные таблицы удобны, но увлекаться ими не стоит, так как они уменьшают читабельность кода всего запроса. Вам будет сложно редактировать данный запрос, а если Вы к нему вернетесь, скажем, через полгода, то Вы вообще сначала не поймете, что он делает.

В разовых случаях, когда требуется срочно получить какой-нибудь анализ, то подзапросы могут значительно уменьшить время на подготовку (*написание*) запроса, поэтому подзапросы – это удобно, но в долгосрочной перспективе лучше рассмотреть возможность вынесения подзапроса в отдельную функцию (*в данной книге мы, конечно же, будем рассматривать функции!*) или, в случае с производной таблицей, - в представление. А представление - это еще одна очень полезная и нужная вещь в T-SQL! Поэтому давайте скорей переходить к рассмотрению представлений.

Глава 5 - Представления

Вы уже не раз слышали в данной книге такое необычное слово как – «*Представление*», и Вам, наверное, интересно, что же это такое и как оно работает! Итак…

Описание и типы представлений

Представление (*VIEW*) – это объект базы данных, который хранит в себе запрос SELECT, и в случае обращения к данному объекту будет возвращен результирующий набор данных, который формирует запрос, указанный в определении представления. Иными словами, это виртуальная (*логическая*) таблица, она не содержит в себе данных, но к ней можно обращаться как к обычной таблице, и она будет возвращать Вам данные. Если Вы слышали такое понятие как «*Вьюха*», так вот, это и есть представление!

Помните, мы с Вами говорили про производные таблицы, которые созданы на основе подзапроса, так вот, если такие производные таблицы Вам будут требоваться достаточно часто, Вы можете для удобства и сокращения кода сохранить данный запрос в виде представления. И затем, где Вам требуется использовать именно тот набор данных, который Вы указали в запросе (*а Вы уже, наверное, понимаете, что такие запросы могут очень сложные!*), Вы будете указывать название представления (*даже в объединениях, подзапросах*), точно так же, как мы указывали название таблиц, согласитесь - это очень удобно! Таким образом, Вы упрощаете и сокращаете код запроса, он становится читабельнее и понятнее!

Как Вы уже поняли, представление – это очень хороший объект в базе данных, поэтому существуют не только представления, которые могут создавать пользователи, но и заранее определенные системные представления.

- **Пользовательские представления** – представление, которое создает пользователь.
- **Системные представления** – представление, которое уже создано разработчиками SQL сервера.

Системные представления – содержат в себе очень полезную информацию о метаданных SQL Server. Под метаданными здесь подразумевается абсолютно вся системная информация, которая поможет Вам как разработчику или администратору SQL сервера. Какие они бывают, и какую информацию они возвращают, мы с Вами рассмотрим чуть позже, а пока давайте научимся создавать свои собственные представления!

Процесс создания таблиц, представлений или других объектов базы данных, лично у меня вызывает какое-то чувство гордости и заряжает энергией, ведь я работал, работал и что-то создал, и это круто! Данный факт мотивирует к дальнейшему развитию! Надеюсь, Вы так же, как и я, любите созидать, а не разрушать!

Пользовательские представления

Из названия понятно, что такой тип представления связан с пользователями, т.е. с нами, с разработчиками, а называются они так, потому что именно мы их создаем.

Как мы выяснили ранее, такие представления упрощают нам разработку запросов. Если с такими объектами как, например, триггеры Вы можете и ни разу не столкнуться (*хотя мы их также рассмотрим*), например, Вы простой аналитик, который пишет запросы на выборку, то с представлениями Вы будете работать каждый день!

Поэтому давайте научимся создавать, изменять и удалять данный вид объектов в базе данных.

Создание

Давайте представим, что нам постоянно требуется знать, сколько товаров в той или иной категории, но писать каждый раз запрос, например, тот который мы использовали, когда рассматривали подзапросы, слишком долго, да и не очень удобно. Поэтому мы приняли решение сохранить этот запрос в виде представления, и тогда, когда нам потребуется узнать количество товаров в категории, мы просто будем обращаться к этому представлению.

Создается представление с помощью инструкции CREATE VIEW. Если я этого еще не говорил, то все объекты в SQL сервере создаются с помощью ключевого слова CREATE (*что собственно и логично*).

Для решения нашей задачи мы создадим следующее представление.

```
CREATE VIEW ViewCntProducts
AS
     SELECT T2.CategoryName AS CategoryName,
           (SELECT COUNT(*)
            FROM TestTable
            WHERE CategoryId = T2.CategoryId) AS CntProducts
     FROM TestTable2 T2
```

После инструкции CREATE VIEW мы указали название представления, т.е. Вы его сами должны придумать. Затем мы указали ключевое слово AS, и только после этого мы написали запрос, результирующий набор которого и будет содержать наше представление. Сразу скажу, что в представлении нельзя использовать секцию ORDER BY, т.е. сортировку. В случае необходимости отсортировать данные Вы можете тогда, когда будете обращаться к этому представлению (*использование ORDER BY возможно, только если указан оператор TOP*).

Теперь, для того чтобы получить данные о количестве товаров в категории, мы можем обратиться к представлению ViewCntProducts, например, так.

```
SELECT * FROM ViewCntProducts
```

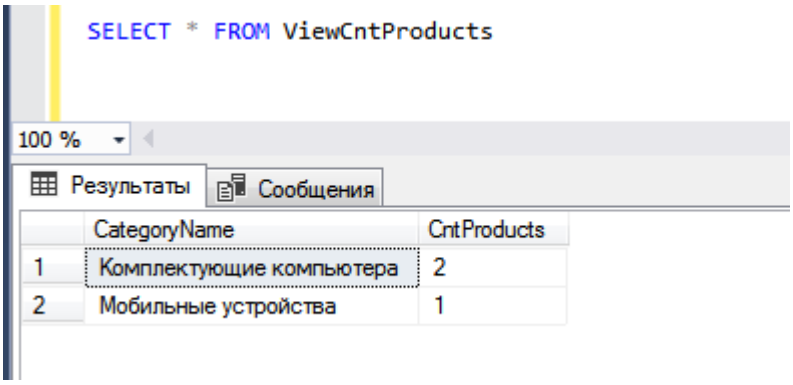


Рис. 57

Как видите, результат тот же (Рис. 57), который возвращал нам исходный запрос. Мы это представление теперь можем использовать в любых своих запросах, включая объединение с таблицами или другими представлениями.

Изменение

А сейчас давайте допустим, что нам нужно, чтобы это представление возвращало еще и идентификатор категории, если Вы обратили внимание, то в предыдущем примере таких данных нет. Для этого используем инструкцию ALTER VIEW, которая подразумевает изменение представления. Если с помощью CREATE объекты создаются, то с помощью ALTER они изменяются.

```
ALTER VIEW ViewCntProducts
AS
      SELECT T2.CategoryId AS CategoryId,
             T2.CategoryName AS CategoryName,
             (SELECT COUNT(*)
             FROM TestTable
             WHERE CategoryId = T2.CategoryId) AS CntProducts
      FROM TestTable2 T2
```

В данном случае мы написали инструкцию ALTER VIEW, что говорит SQL серверу, что мы хотим изменить существующий объект. Затем указали название представления, чтобы он мог определить, какое именно представление мы хотим изменить. После ключевого слова AS мы указали новое определение представления, т.е. измененный запрос SELECT.

Если мы теперь обратимся к представлению, то получим следующие результаты (Рис. 58)

```
SELECT * FROM ViewCntProducts
```

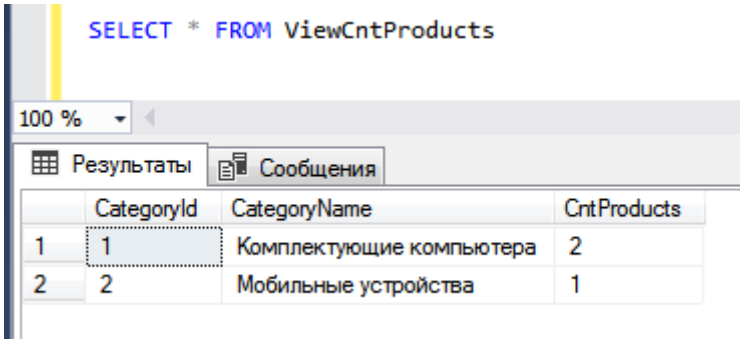


Рис. 58

Удаление

Если Вам уже точно не нужно представление, т.е. Вы им больше не будете пользоваться, и оно не используется в других представлениях, функциях или процедурах (*и функции, и процедуры мы с Вами, конечно же, рассмотрим*), иными словами, на него никто не ссылается, то Вы его можете удалить. Делается это с помощью инструкции DROP VIEW, как Вы уже поняли, с помощью DROP объекты удаляются.

```
DROP VIEW ViewCntProducts
```

Теперь данного представления больше нет, и к нему Вы больше обратиться не сможете. Советую держать свою базу данных в чистоте, т.е. чтобы в ней хранились только нужные объекты. Если не удалять объекты, которые больше никогда не будут использоваться, например, Вы для каких-нибудь разовых целей создали таблицу или представление, то Ваша база превратиться просто в помойку и в ней уже разобраться или найти что-то будет невозможно. Что там говорить, только нужных объектов в базе данных со временем станет очень много, поэтому имеет смысл вести соответствующую документацию, чтобы проще было потом вносить изменения или адаптировать нового сотрудника.

Системные представления

С пользовательскими представлениями, я думаю, понятно, переходим к системным.

Как я уже говорил, системные представления уже созданы в SQL сервере разработчиками, другими словами, системные представления – это встроенные объекты.

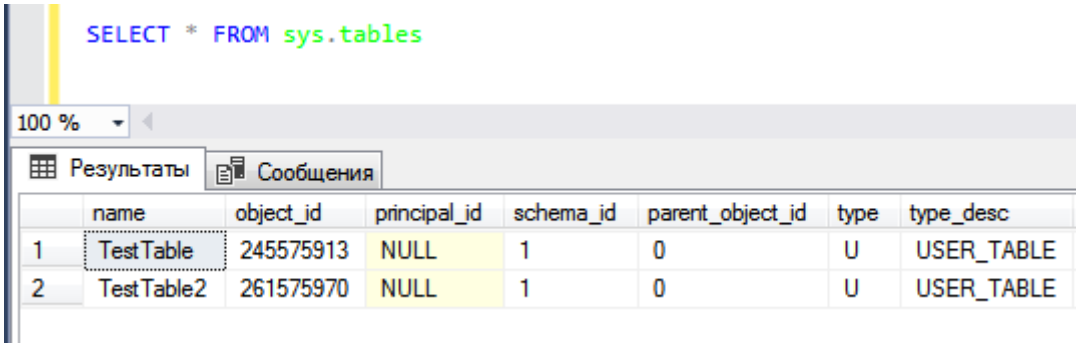
На текущий момент Вы, наверное, даже себе представить не можете, сколько их на самом деле, и какую полезную информацию они могут Вам дать. Я хочу сказать, что системных представлений очень много!

Возвращают они самую разную информацию, например, Вы можете узнать перечень объектов в базе данных, включая их свойства, или посмотреть, какие файлы базы данных включаются в данную базу, в общем, обо всем, что есть и используется в SQL сервере, можно узнать из системных представлений. Рассматривать все мы, конечно же, не будем, поверьте, их действительно много! Однако мы рассмотрим несколько примеров, чтобы Вы разобрались в том, как к ним обращаться, какую и в каком виде они возвращают информацию.

Большинство, да практически все системные объекты, находятся в специальной схеме SYS. Помните, когда мы рассматривали секцию FROM, я говорил, что иногда требуется уточнять название и расположение источников, так вот для того чтобы обратиться к системным представлениям, необходимо перед названием представления указывать имя схемы, т.е. SYS.

Для примера давайте узнаем, какие таблицы в нашей базе данных созданы.

```
SELECT * FROM sys.tables
```



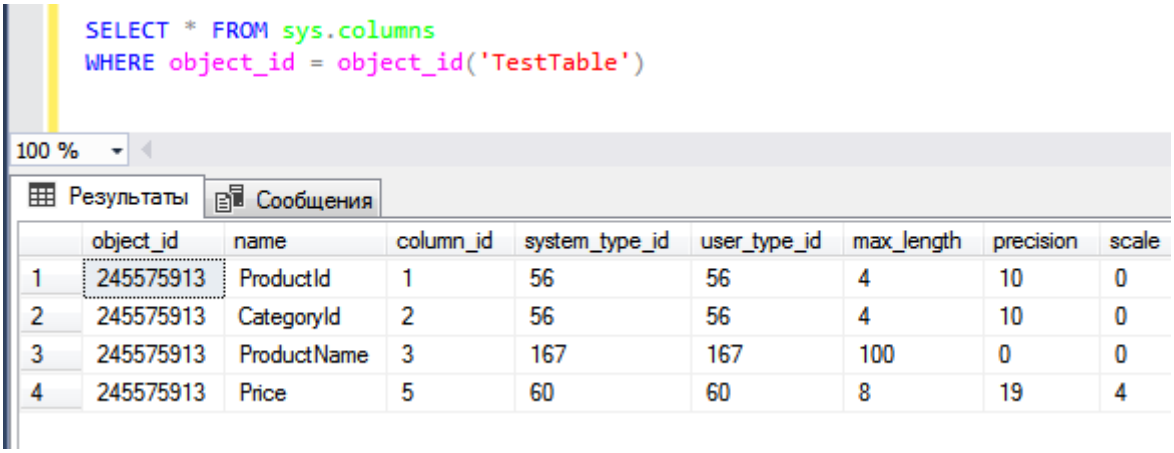
	name	object_id	principal_id	schema_id	parent_object_id	type	type_desc
1	TestTable	245575913	NULL	1	0	U	USER_TABLE
2	TestTable2	261575970	NULL	1	0	U	USER_TABLE

Рис. 59

Надеюсь понятно, что для получения такой информации можно использовать представление **sys.tables**, которое содержит информацию о таблицах.

Вот еще пример, в данном запросе мы с помощью системного преставления **sys.columns**, узнаем, какие колонки есть в нашей таблице TestTable.

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



	object_id	name	column_id	system_type_id	user_type_id	max_length	precision	scale
1	245575913	ProductId	1	56	56	4	10	0
2	245575913	CategoryId	2	56	56	4	10	0
3	245575913	ProductName	3	167	167	100	0	0
4	245575913	Price	5	60	60	8	19	4

Рис. 60

Здесь в условии я указал таблицу, колонки которой я хочу получить. object_id – это идентификатор объекта, а object_id('TestTable') – это встроенная функция, которая возвращает по названию объекта его системный идентификатор (*до системных функций мы еще дойдем!*), то есть условие говорит SQL серверу, что мне нужны колонки только определённого объекта.

В итоге мы и получили характеристики колонок таблицы TestTable.

Точно таким же способом Вы можете узнать информацию обо всех объектах в SQL сервере, для каждого типа объекта существует свое представление.

В среде SQL Server Management Studio Вы можете посмотреть перечень этих системных представлений, для этого найдите в обозревателе объектов раздел «*Представления*», а в нем раздел «*Системные представления*», откройте его плюсиком, и Вы увидите большой перечень объектов.

Для начальных знаний про системные представления, я думаю, достаточно. Главное, Вы знаете, что они есть, как к ним обращаться, и что они возвращают очень полезную информацию! Некоторые «*Админы*» не знают и этого!

Двигаемся дальше! Впереди нас ждет модификация данных, я думаю, не стоит говорить, что добавлять, изменять и удалять данные Вы точно должны уметь, и будете!

Глава 6 - Модификация данных в таблицах

В данной главе мы с Вами будем разговаривать о том, как в SQL сервере происходит модификация данных, то есть добавление, обновление и удаление.

База данные в первую очередь - это информация, а информация должна каким-то образом попадать в базу данных, и если она изменилась, то должна как-то изменяться в базе данных. Информация, как мы знаем, в базе хранится в виде таблиц, поэтому под добавлением, изменением или удалением данных в базе мы подразумеваем добавление, изменение или удаление строк в таблицах. В SQL есть специальные инструкции, с помощью которых мы можем все это делать.

Все инструкции очень важные, поэтому, когда на практике Вы будете использовать эти инструкции, то обязательно четко контролируйте и в случае необходимости проверяйте, что и куда Вы будете добавлять, что и на что Вы будете изменять, что Вы будете удалять, ведь малейшая оплошность и все данные, например, из таблицы, будут удалены или некорректно изменены, что приведет к очень серьезным последствиям, особенно, если у Вас нет архива данных!

Поэтому к запросам на модификацию относитесь очень серьезно, так как если Вы напишите некорректный запрос на выборку - это одно, но, если Вы напишите некорректный запрос на модификацию данных, это другое, иными словами, другие последствия.

Совет 4

Разработку и отладку инструкций на модификацию данных лучше всего осуществлять на тестовых данных, чтобы в случае логической ошибки в запросе ничего страшного не произошло, а на «боевых» данных выполнять только готовые, проверенные, отлаженные инструкции.

Добавление данных – INSERT

Помните, в начале Главы 4. «*Выборка данных – оператор SELECT*» я использовал запросы на добавление данных в наши таблицы, давайте сейчас разберем эти запросы.

Для начала я их напомню. Можете снова их выполнить, таким образом, мы добавим в базу еще точно такие же строки, в следующих разделах мы их обновим и удалим. После запросов на добавление данных, давайте сразу напишем запрос SELECT, для того чтобы посмотреть на итоговый результат.

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

GO

INSERT INTO TestTable2
     VALUES ('Комплектующие компьютера'),
            ('Мобильные устройства')
GO

SELECT * FROM TestTable
SELECT * FROM TestTable2
```

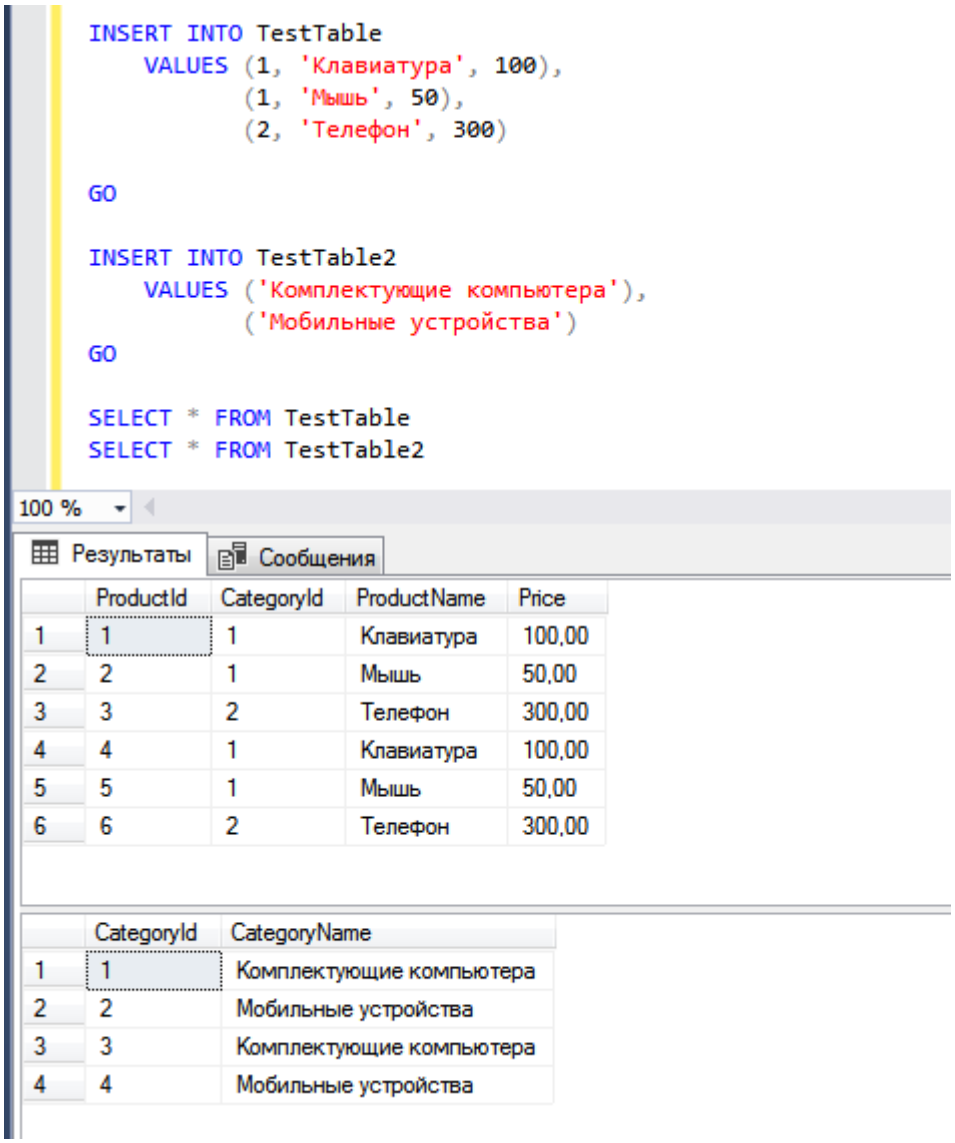


Рис. 61

В результате выполнения данных инструкций в таблицу TestTable добавилось три записи, а в TestTable2 две. Сейчас я поясню каждую команду и каждое ключевое слово в этих инструкциях.

Для добавления данных в таблицу в T-SQL используется инструкция **INSERT INTO**. После этой инструкции идет название таблицы. Затем в нашем случае сразу идет ключевое слово VALUES (*это конструктор табличных значений*), после которого в скобочках идут значения для каждой добавляемой строки. Иными словами, все, что в скобочках - это одна строка, а в ней через запятую мы указываем значение каждого столбца в том порядке, в котором они находятся в таблице.

В нашем случае в таблице у нас есть, если помните, столбец идентификаторов (*ProductId*), но так как он со свойством IDENTITY, он автоматически генерирует идентификаторы, поэтому в такой столбец вставить данные не получится, и мы его не указывали в списке столбцов для вставки.

Если возник вопрос что такое GO, то это просто команда отделения одного пакета инструкций от другого.

Важным является то, что мы здесь использовали сокращенную запись добавления данных в таблицу, так как у нас столбцов мало, и мы точно знаем их порядок. Но на практике так лучше не делать, а если и делать, то, как мы сейчас, либо для теста, либо для маленькой таблицы, порядок и количество столбцов которой нам точно известно, и мы уверены в корректности добавления данных.

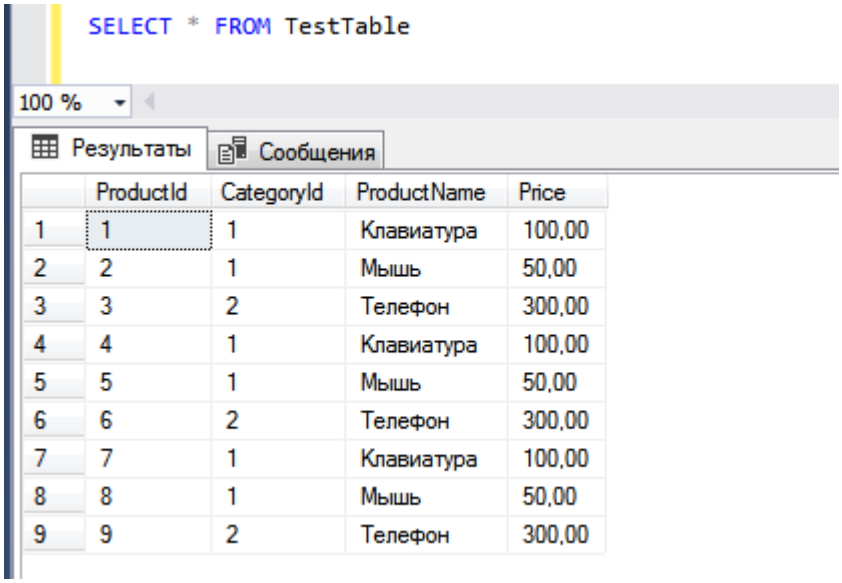
В остальных случаях необходимо перечислять столбцы в скобочках после названия таблицы. Давайте напишем и выполним еще раз инструкцию INSERT, но более правильно с перечислением списка столбцов.

```
INSERT INTO TestTable (CategoryId, ProductName, Price)
     VALUES (1, 'Клавиатура', 100),
            (1, 'Мышь', 50),
            (2, 'Телефон', 300)
```

Вы видите, что в данном случае мы перечислили столбцы (*в этом случае их порядок может отличаться от фактического расположения в таблицы*), в которые будем добавлять данные, главное, нужно понимать, что порядок списка столбцов для вставки должен соответствовать порядку значений для вставки, которые указаны через запятую после ключевого слова VALUES. Другими словами, в нашем случае, в столбец CategoryId первой добавляемой строки запишется значение 1, в столбце ProductName «*Клавиатура*», а в Price 100, во вторую строку соответственно 1 в CategoryId, «*Мышь*» в ProductName, и 50 в Price, в третью, я думаю, смысл понятен.

Можете посмотреть на итоговый результат, выполним соответствующий запрос.

```
SELECT * FROM TestTable
```



	ProductId	CategoryId	ProductName	Price
1	1	1	Клавиатура	100,00
2	2	1	Мышь	50,00
3	3	2	Телефон	300,00
4	4	1	Клавиатура	100,00
5	5	1	Мышь	50,00
6	6	2	Телефон	300,00
7	7	1	Клавиатура	100,00
8	8	1	Мышь	50,00
9	9	2	Телефон	300,00

Рис. 62

Совет 5

При добавлении данных в таблицу всегда указывайте столбцы, в которые необходимо добавить данные. Это избавит Вас от путаницы и возможных проблем с результатом данной операции. Чем четче Вы будете давать инструкции SQL серверу, тем меньше у Вас будет ошибок!

Также обязательно стоит отметить то, что в списке столбцов для вставки Вы должны указать все обязательные столбцы, т.е. те, которые не могут содержать значение NULL. Помните, в Главе 3 при создании таблицы мы указывали параметр NULL или NOT NULL, это как раз и есть признак обязательности столбца. Так как если мы не укажем столбец в списке для добавления данных, в него запишется значение NULL, а если у столбца стоит признак того, что значение NULL запрещены в нем, то будет ошибка, о чем нам, наверное, с удовольствием сообщит SQL сервер.

Данный способ удобен при добавлении отдельных записей или, как в нашем случае, нескольких записей, но на практике достаточно часто требуется осуществить массовую вставку данных в таблицу на основе запроса SELECT (*выборки данных*). Способ с VALUES использовать не получится.

В таких случаях вместо перечисления значений, т.е. команды VALUES, необходимо указывать сам запрос SELECT, результирующий набор данных которого и будет вставлен в таблицу.

Как это делать я сейчас и покажу. На самом деле все просто, Вы пишите инструкцию INSERT INTO, перечисляете столбцы, а потом вместо VALUES пишите запрос, список выборки которого должен соответствовать списку столбцов для вставки, я имею в виду, порядок и, конечно же, сами

данные. Да, еще один очень важный момент, тип данных значений, которые мы будем вставлять, должен соответствовать типу данных столбца, в который будет вставлено это значение, или, хотя бы, поддерживал неявное преобразование в фактический тип данных столбца. И я Вам советую контролировать тип данных (*формат*) значений, которые Вы будете вставлять в таблицу.

В следующем запросе я вставляю данные в таблицу TestTable, на основе выборки (*запроса SELECT*) из той же таблицы, но я не все записи продублирую, а только те, у которых идентификатор больше 3.

```
INSERT INTO TestTable (CategoryId, ProductName, Price)
     SELECT CategoryId, ProductName, Price
     FROM TestTable
     WHERE ProductId > 3
```

Запрос на выборку Вы можете писать практически любой (*например, очень сложные, включающие много расчетов запросы*), главное, чтобы результирующий набор данных соответствовал столбцам, в которые необходимо вставить эти данные.

Таким образом, у нас получилось 2 способа добавления данных, это добавление отдельных строк, т.е. их перечисление, и массовая вставка, при которой мы указываем запрос в качестве источника данных. Однако есть и другие возможности, например, использовать в качестве источника результирующий набор данных процедуры, т.е. вместо запроса SELECT можно указать вызов процедуры, которая возвращает табличные данные, но так как процедуры мы с Вами еще не рассматривали, не забивайте пока этим себе голову! А процедуры мы с Вами подробно рассмотрим чуть поздней.

Давайте подведем итог про инструкцию INSERT, для этого сейчас я Вам покажу упрощённый синтаксис данной инструкции с описанием всех представленных команд и ключевых слов, так сказать, для закрепления полученной информации.

INSERT [INTO] [таблица] (*список столбцов, ...*)
VALUES (*список значений, ...*)
Или
SELECT *запрос на выборку*
Или
EXECUTE *процедура*

Где,

- ✓ INSERT INTO – это команда добавления данных в таблицу;
- ✓ Таблица – это имя целевой таблицы, в которую необходимо вставить новые записи;
- ✓ Список столбцов – это перечень имен столбцов таблицы, в которую будут вставлены данные, разделенные запятыми;
- ✓ VALUES – это конструктор табличных значений, с помощью которого мы указываем значения, которые будем вставлять в таблицу;
- ✓ Список значений – это значения, которые будут вставлены, разделенные запятыми. Они перечисляются в том порядке, в котором указаны столбцы в списке столбцов;
- ✓ SELECT – это запрос на выборку. Результирующий набор данных, который вернет запрос, должен соответствовать списку столбцов;
- ✓ EXECUTE – это вызов процедуры. Результирующий набор данных, который вернет хранимая процедура, должен соответствовать списку столбцов.

Обновление данных – UPDATE

Добавлять данные мы научились, теперь нам нужно научиться их изменять. Изменение данных подразумевает обновление значения в существующих строках таблицы, а именно, обновление значений конкретных ячеек в таблице. Другими словами, если вспомнить нашу тестовую таблицу с товарами, у нас там есть цена товара, так вот, если эта цена изменилась у одного товара, мы соответственно должны найти нужную строку с этим товаром и изменить значение конкретного столбца, в нашем случае столбец, который содержит цену.

В SQL для обновления данных в таблицах используется инструкция **UPDATE**, с помощью которой вносятся любые изменения в данные, которые хранит таблица. С помощью UPDATE можно изменить как одну запись, так и несколько или даже все имеющиеся в таблице.

Итак, давайте представим, что у нашего товара с идентификатором 1 изменилась цена, мы для этого напишем следующий запрос на изменение. Чтобы было наглядней, давайте перед обновлением запустим запрос на выборку, чтобы посмотреть, что у нас было, а после обновления еще раз запустим тот же запрос на выборку, чтобы посмотреть, что получилось.

```
SELECT * FROM TestTable
WHERE ProductId = 1

GO

UPDATE TestTable SET Price = 120
WHERE ProductId = 1

GO

SELECT * FROM TestTable
WHERE ProductId = 1
```

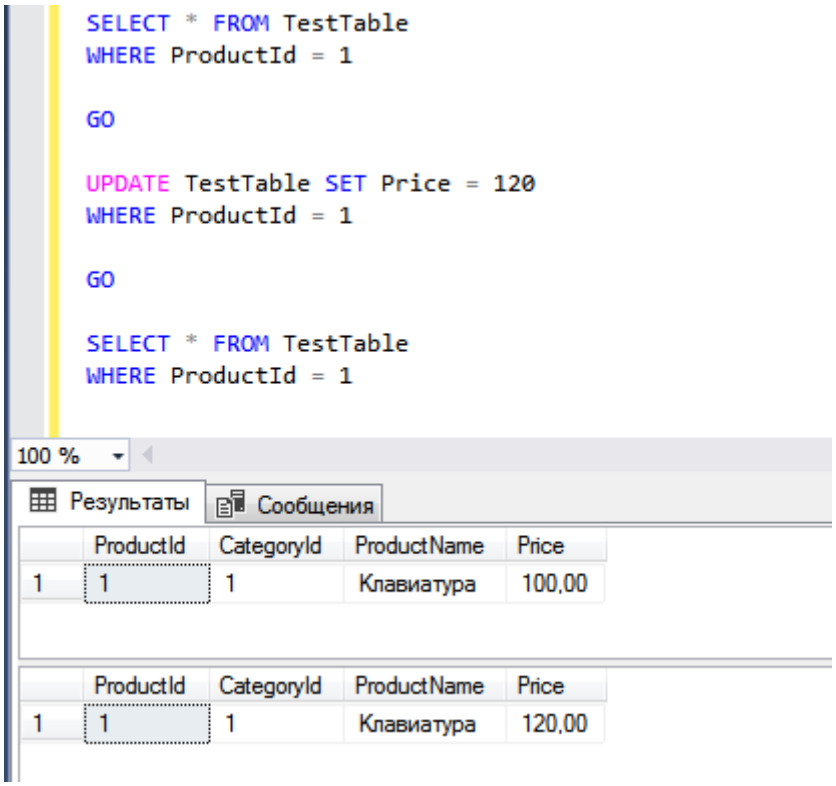


Рис.63

Цена изменилась. Что мы сделали? Мы написали команду UPDATE, затем указали название таблицы, в которой лежат данные, которые нам нужно изменить. Далее написали ключевое слово SET,

после которого мы указали название столбца, который нам нужно изменить, и с помощью оператора равно (=) присвоили новое значение. Также мы указали условие WHERE, чтобы конкретизировать строки, если быть точнее, в нашем случае одну строку. Если бы мы этого не сделали, т.е. WHERE вообще убрали, у нас обновился столбец Price для всех строк таблицы, абсолютно у всех товаров была бы цена 120, что, как Вы понимаете, не очень хорошо. Поэтому необходимо четкое условие, которое Вы предварительно должны проверить на запросе SELECT.

Совет 6

Перед запуском инструкции UPDATE всегда проверяйте наличие и корректность условия, которое предварительно должно быть проверено на SELECT.

Теперь давайте изменим не одну строку, а несколько, и не один столбец, а несколько. Так как на практике в большинстве случаев Вам придётся обновлять набор данных, иными словами, неопределенное количество строк, которое подходит под Ваше логическое условие. Также, достаточно часто, придётся обновлять несколько столбцов, а не один, как в нашем примере. Например, изменилось наименование товара и его цена, соответственно, зачем нам писать два запроса на обновление, если можно все сделать за один.

Для примера давайте обновим все записи, у которых ProductId > 3 (*т.е. те, которые мы создали, когда тренировались вставлять новые строки*), мы у них изменим наименование и цену.

```
SELECT * FROM TestTable
WHERE ProductId > 3

GO

UPDATE TestTable SET ProductName = 'Тестовый товар', Price = 150
WHERE ProductId > 3

GO

SELECT * FROM TestTable
WHERE ProductId > 3
```

```
SELECT * FROM TestTable
WHERE ProductId > 3

GO

UPDATE TestTable SET ProductName = 'Тестовый товар', Price = 150
WHERE ProductId > 3

GO

SELECT * FROM TestTable
WHERE ProductId > 3
```

100 %

Результаты | Сообщения

	ProductId	CategoryId	ProductName	Price
1	4	1	Клавиатура	100,00
2	5	1	Мышь	50,00
3	6	2	Телефон	300,00
4	7	1	Клавиатура	100,00
5	8	1	Мышь	50,00
6	9	2	Телефон	300,00
7	10	1	Клавиатура	100,00
8	11	1	Мышь	50,00
9	12	2	Телефон	300,00
10	13	1	Клавиатура	100,00
11	14	1	Мышь	50,00
12	15	2	Телефон	300,00

	ProductId	CategoryId	ProductName	Price
1	4	1	Тестовый товар	150,00
2	5	1	Тестовый товар	150,00
3	6	2	Тестовый товар	150,00
4	7	1	Тестовый товар	150,00
5	8	1	Тестовый товар	150,00
6	9	2	Тестовый товар	150,00
7	10	1	Тестовый товар	150,00
8	11	1	Тестовый товар	150,00
9	12	2	Тестовый товар	150,00
10	13	1	Тестовый товар	150,00
11	14	1	Тестовый товар	150,00
12	15	2	Тестовый товар	150,00

Рис. 64

В данном случае у всех строк, у которых ProductId > 3, мы изменили значение ProductName на 'Тестовый товар', а Price на 150. Для этого после ключевого слова SET мы через запятую перечислили все столбцы, какие мы хотим обновить, и, соответственно, указали новое значение с помощью оператора =.

Еще достаточно часто требуется перекинуть данные из одной таблицу в другую, т.е. обновить записи одной таблицы на значения, которые расположены в другой. Для этого в процессе инструкции UPDATE можно объединить две и более таблицы по ключу (*записи, которые есть в обеих таблицах*). В нашем случае у нас всего две таблицы, но они у нас имеют связь, и так как большую часть данных мы с Вами уже изменили на непонятные значения, поэтому давайте обновим эти записи еще раз.

```
SELECT * FROM TestTable
WHERE ProductId > 3
GO
UPDATE TestTable SET ProductName = T2.CategoryName, Price = 200
FROM TestTable2 T2
INNER JOIN TestTable T1 ON T1.CategoryId = T2.CategoryId
WHERE T1.ProductId > 3
GO
SELECT * FROM TestTable
WHERE ProductId > 3
```

```
SELECT * FROM TestTable
WHERE ProductId > 3

GO

UPDATE TestTable SET ProductName = T2.CategoryName, Price = 200
FROM TestTable2 T2
INNER JOIN TestTable T1 ON T1.CategoryId = T2.CategoryId
WHERE T1.ProductId > 3

GO

SELECT * FROM TestTable
WHERE ProductId > 3
```

100 %

Результаты | Сообщения

	ProductId	CategoryId	ProductName	Price
1	4	1	Тестовый товар	150,00
2	5	1	Тестовый товар	150,00
3	6	2	Тестовый товар	150,00
4	7	1	Тестовый товар	150,00
5	8	1	Тестовый товар	150,00
6	9	2	Тестовый товар	150,00
7	10	1	Тестовый товар	150,00
8	11	1	Тестовый товар	150,00
9	12	2	Тестовый товар	150,00
10	13	1	Тестовый товар	150,00
11	14	1	Тестовый товар	150,00
12	15	2	Тестовый товар	150,00

	ProductId	CategoryId	ProductName	Price
1	4	1	Комплектующие компьютера	200,00
2	5	1	Комплектующие компьютера	200,00
3	6	2	Мобильные устройства	200,00
4	7	1	Комплектующие компьютера	200,00
5	8	1	Комплектующие компьютера	200,00
6	9	2	Мобильные устройства	200,00
7	10	1	Комплектующие компьютера	200,00
8	11	1	Комплектующие компьютера	200,00
9	12	2	Мобильные устройства	200,00
10	13	1	Комплектующие компьютера	200,00
11	14	1	Комплектующие компьютера	200,00
12	15	2	Мобильные устройства	200,00

Рис. 65

Как видите, у нас в ProductName таблицы TestTable записались данные из столбца CategoryName таблицы TestTable2. Мы для этого после UPDATE указали секцию FROM, в которой написали таблицу - источник, из которого нам нужно получить значения для обновления целевой таблицы TestTable. Затем так же, как и в инструкции SELECT мы объединили таблицы TestTable и TestTable2 с помощью секции INNER JOIN по связи CategoryId, чтобы каждая запись TestTable получила соответствующее значение по данной связи из TestTable2. Также в условии мы оставили предикат ProductId > 3, для того чтобы наши нормальные строки не испортить, ведь, как Вы поняли, по своей сути запрос, который мы выполнили чуть выше, абсолютно нелогичный, наименование категории в наименование товара записали☺.

Совет 7

Запросы на изменение данных лучше всего проектировать так, чтобы они вставляли или изменяли как можно больше строк одной инструкцией, другими словами, если есть возможность заменить несколько инструкций обновления или добавления данных одной инструкцией, заменяйте, так эффективней по производительности.

Удаление данных – DELETE, TRUNCATE

Как говорится: «*Что-что, а удалять то мы умеем!*». Но в T-SQL удалять данные также нужно научиться. Тем более, как Вы понимаете, удаление данных - одна из самых важных операций, которая требует полного контроля над ней. Если Вы удалите что-то не то, т.е. какие-то важные данные для организации и потом не сможете восстановить их, то в лучшем случае Вам объявят выговор и понизят зарплату, в худшем Вас потребуют возместить ущерб, а затем уволят.

Совет 8

При удалении данных Вы так же, как и при обновлении данных, должны четко понимать и протестировать условие на тестовых данных или с помощью инструкции SELECT, иными словами, Вы должны четко видеть, какие данные Вы сейчас удалите.

В T-SQL для удаления данных можно использовать две инструкции **DELETE** и **TRUNCATE**. Их отличие в том, что с помощью DELETE можно удалить часть данных, т.е. применять условие WHERE, а TRUNCATE удаляет все строки в таблице, при этом не записывая удаление отдельных строк в журнал транзакций. Если в таблице есть столбец идентификаторов (*задано свойство IDENTITY*) TRUNCATE сбрасывает счетчик на начальное значение, а DELETE нет. Также инструкция TRUNCATE требует меньше ресурсов, чем DELETE, т.е. она более производительна.

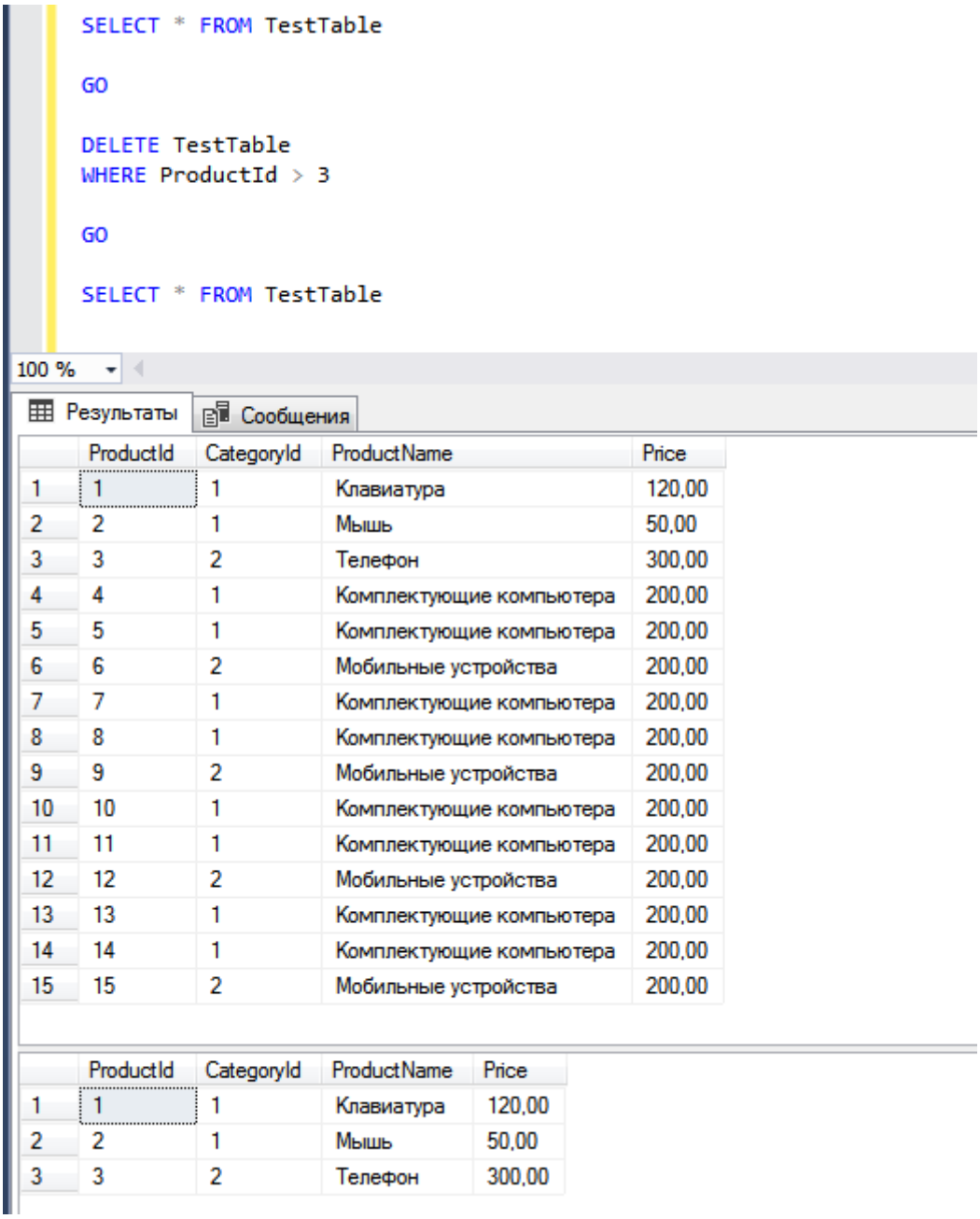
В любом случае TRUNCATE Вы будете использовать редко, по крайней мере, мне она требуется достаточно редко, а вот DELETE – это основная инструкция удаления данных в T-SQL.

Сначала давайте рассмотрим пример использования DELETE, для этого напишем запрос, в котором мы удалим часть данных из нашей таблицы TestTable, те, которые мы уже и так испортили в предыдущем разделе.

```
SELECT * FROM TestTable
GO

DELETE TestTable
WHERE ProductId > 3

GO
SELECT * FROM TestTable
```



	ProductId	CategoryId	ProductName	Price
1	1	1	Клавиатура	120,00
2	2	1	Мышь	50,00
3	3	2	Телефон	300,00
4	4	1	Комплектующие компьютера	200,00
5	5	1	Комплектующие компьютера	200,00
6	6	2	Мобильные устройства	200,00
7	7	1	Комплектующие компьютера	200,00
8	8	1	Комплектующие компьютера	200,00
9	9	2	Мобильные устройства	200,00
10	10	1	Комплектующие компьютера	200,00
11	11	1	Комплектующие компьютера	200,00
12	12	2	Мобильные устройства	200,00
13	13	1	Комплектующие компьютера	200,00
14	14	1	Комплектующие компьютера	200,00
15	15	2	Мобильные устройства	200,00

	ProductId	CategoryId	ProductName	Price
1	1	1	Клавиатура	120,00
2	2	1	Мышь	50,00
3	3	2	Телефон	300,00

Рис. 66

Мы написали команду DELETE, потом написали название таблицы, из которой мы хотим удалить данные, а после написали условие, по которому мы хотим удалить данные. Если бы мы его не указали, у нас удалились все данные из этой таблицы.

Теперь давайте посмотрим, как работает инструкция TRUNCATE.

```
SELECT * FROM TestTable

GO

TRUNCATE TABLE TestTable

GO

SELECT * FROM TestTable
```

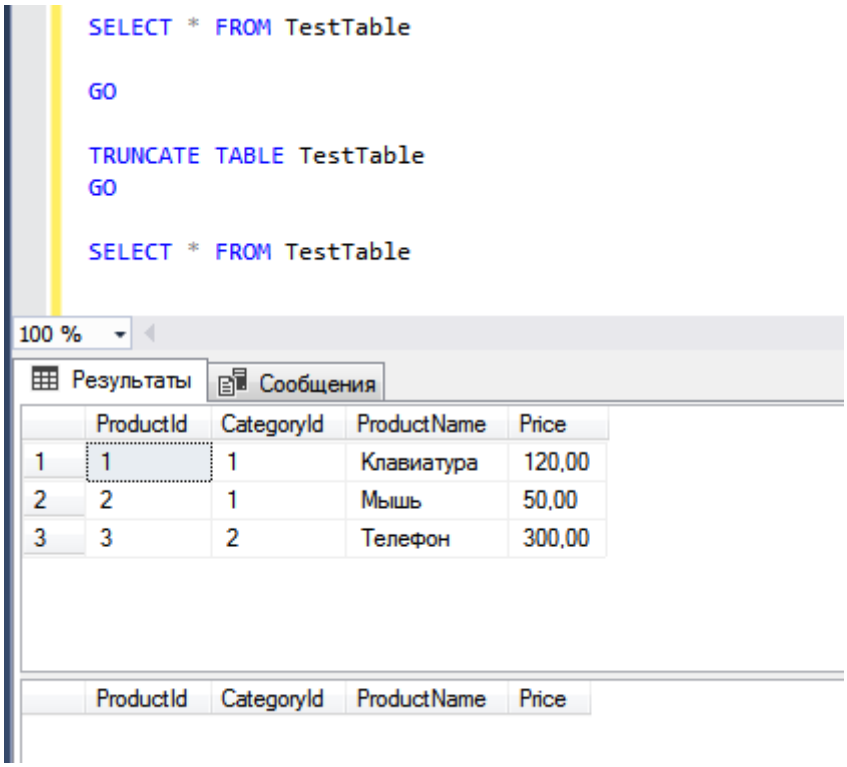


Рис. 67

Мы видим (Рис. 65), что у нас удалились все данные из таблицы TestTable, для этого мы просто написали инструкцию TRUNCATE TABLE и указали таблицу, которую необходимо очистить.

Сейчас давайте выполним INSERT в TestTable, т.е. добавим данные (*те же, которые были раньше*), так как у нас их теперь нет, и заодно посмотрим, что столбец идентификаторов снова начал счетчик с первого значения, в нашем случае с 1. Если бы мы удалили все данные с помощью DELETE, то счетчик не сбросился, и после выполнения следующего запроса он просто продолжился.

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

GO

SELECT * FROM TestTable
```

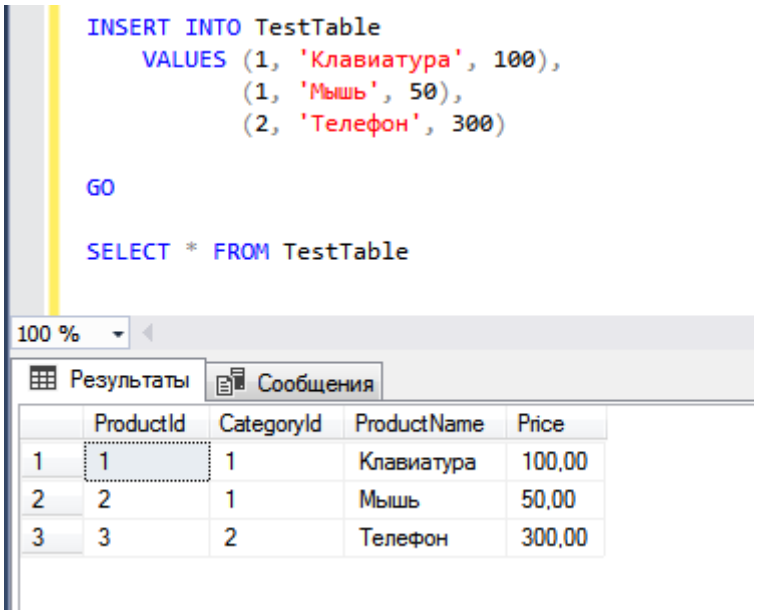


Рис. 68

MERGE

Мы с Вами рассмотрели основные операции модификации данных в Microsoft SQL Server, но также есть еще одна операция модификации данных – это **MERGE**, которая выполняет операции добавления, обновления или удаления данных для таблицы на основе результатов соединения с другой таблицей. Другими словами, с помощью MERGE можно в одной инструкции выполнить INSERT, UPDATE или DELETE на основе выполнения или невыполнения определенного условия. Например, можно синхронизировать две схожие таблицы за счет выполнения соответствующих операций, если в одной таблице есть строка, а в другой ее нет, то выполняем вставку данных, если строка есть, то обновляем ее, а если есть строка, которой не должно быть, то удаляем ее.

Смысл MERGE, я думаю, Вы поняли, это своего рода слияние. Но у данной операции есть один небольшой недостаток, ее синтаксис немного сложен, и начинающим его понять будет, наверное, сложно, но я попробую его объяснить.

Для начала давайте посмотрим на упрощённый синтаксис MERGE.

```
MERGE <Основная таблица>
        USING <Таблица или запрос источника>
        ON <Условия объединения>
        [ WHEN MATCHED [ AND <Доп. условие> ]
                THEN <UPDATE или DELETE>
        [ WHEN NOT MATCHED [ AND Доп. условие> ]
                THEN <INSERT> ]
        [ WHEN NOT MATCHED BY SOURCE [ AND <Доп. условие> ]
                THEN <UPDATE или DELETE> ] [ ...n ]
[ OUTPUT ];
```

Сначала мы пишем MERGE, затем название таблицы, в которую нам необходимо внести изменения. После мы пишем ключевое слово USING и указываем таблицу, с которой нам необходимо выполнить объединение, с помощью ключевого слова ON мы указываем условие объединения.

Если условие, по которому происходит объединение, истина (*WHEN MATCHED*), то мы можем выполнить операции обновления или удаления. Если условие не истина, т.е. отсутствуют данные (*WHEN NOT MATCHED*), то мы можем выполнить операцию вставки (*INSERT добавление данных*). Также если в основной таблице присутствуют данные, которые отсутствуют в таблице (*или результате запроса*) источника (*WHEN NOT MATCHED BY SOURCE*), то мы можем выполнить обновление или удаление таких данных.

Также в дополнение к основным перечисленным выше условиям можно указывать «*Дополнительные условия поиска*», они указываются через ключевое слово AND.

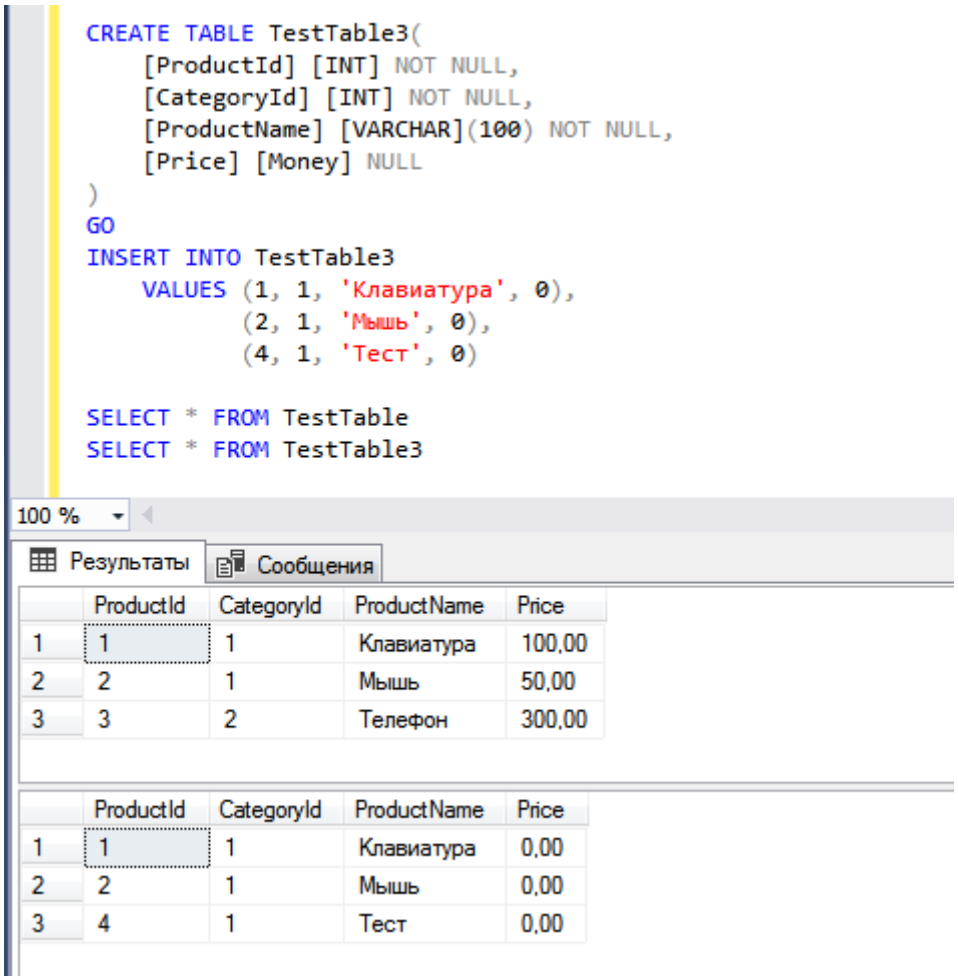
С помощью ключевого слова OUTPUT мы можем посмотреть, какие именно изменения мы внесли.

Давайте перейдем к примерам, чтобы стало более понятно.

Сначала мы создадим еще одну дополнительную таблицу TestTable3 со схожей структурой, как в нашей таблице TestTable. Затем мы добавим в нее данные, которые будут немного отличаться от тех, которые в TestTable.

```
CREATE TABLE TestTable3(
     [ProductId] [INT] NOT NULL,
     [CategoryId] [INT] NOT NULL,
     [ProductName] [VARCHAR](100) NOT NULL,
     [Price] [Money] NULL
)
GO
INSERT INTO TestTable3
     VALUES (1, 1, 'Клавиатура', 0),
            (2, 1, 'Мышь', 0),
            (4, 1, 'Тест', 0)

SELECT * FROM TestTable
SELECT * FROM TestTable3
```



	ProductId	CategoryId	ProductName	Price
1	1	1	Клавиатура	100,00
2	2	1	Мышь	50,00
3	3	2	Телефон	300,00

	ProductId	CategoryId	ProductName	Price
1	1	1	Клавиатура	0,00
2	2	1	Мышь	0,00
3	4	1	Тест	0,00

Рис. 69

Сейчас давайте выполним операцию MERGE на таблице TestTable3 так, чтобы в ней были точно такие же данные, как и в TestTable.

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

SELECT * FROM TestTable
SELECT * FROM TestTable3
```

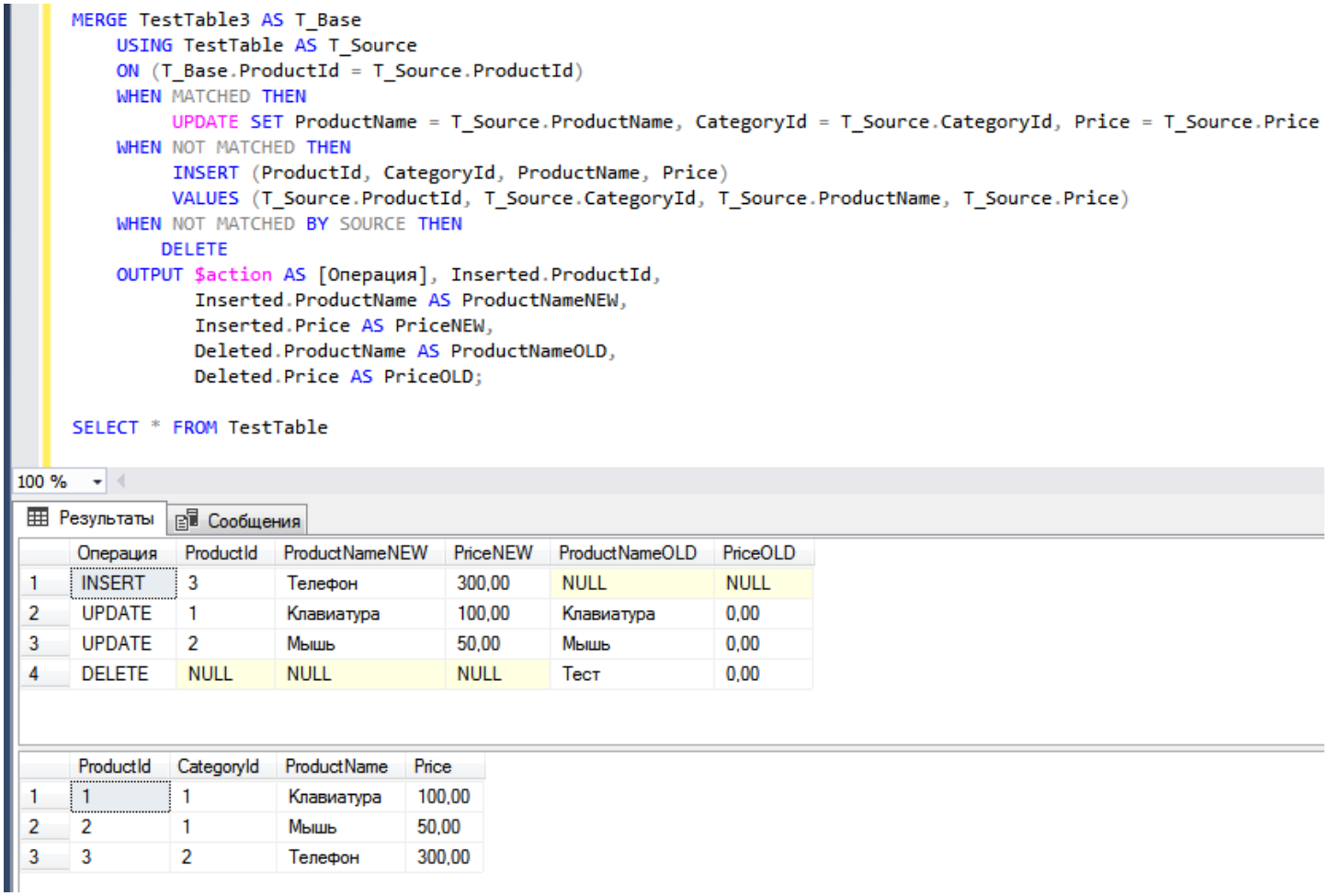


Рис. 70

Если мы посмотрим на результат (Рис. 68), мы увидим, что у нас выполнилась одна инструкция INSERT, две UPDATE и одна DELETE. Так как объединение было выполнено по условию T_Base.ProductId = T_Source.ProductId, т.е. по идентификатору товара, две строки с идентификаторами у нас совпали, по ним и был выполнен UPDATE, инструкция после WHEN MATCHED THEN, одной строки вообще не было, поэтому и был выполнен INSERT, инструкция после WHEN NOT MATCHED

THEN, а одна строка была лишняя (*ProductId = 4*), к ней и применилась инструкция DELETE после WHEN NOT MATCHED BY SOURCE THEN. Таким образом, мы синхронизировали две таблицы. Для того чтобы посмотреть, что именно произошло, мы написали ключевое слово OUTPUT, где переменная $action хранит название операции, которая была выполнена для соответствующей строки, а системные, временные таблицы Inserted и Deleted (*находящиеся в оперативной памяти*) хранят копии строк, над которыми была произведена операция (*более детально инструкцию OUTPUT мы рассмотрим в следующем разделе*). Также хотелось отметить, что в конце инструкции MERGE обязательно должна идти точка с запятой (;) иначе возникнет ошибка.

Я понимаю, что сложно это усвоить, но иногда MERGE бывает очень полезной операцией, поэтому Вы о ней должны знать.

Мы с Вами рассмотрели все операции языка T-SQL, с помощью которых можно модифицировать, т.е. изменять данные. Как я уже говорил, INSERT, UPDATE и DELETE Вы будете использовать часто, а вот MERGE по необходимости.

Если Вам что-то было непонятно, советую перечитать соответствующий раздел и обязательно попрактиковаться, так как успешное обучение языку T-SQL зависит от того, как Вы усваиваете прочитанный материал, и как Вы применяете на практике полученные знания.

Инструкция OUTPUT

В предыдущем разделе, для того чтобы посмотреть, какие именно изменения мы внесли инструкцией MERGE, мы использовали команду OUTPUT, однако данную команду можно использовать не только с MERGE, но и с INSERT, UPDATE и DELETE, не прибегая к дополнительным запросам SELECT, так, как мы это делали, когда рассматривали эти инструкции (*помните, мы писали запрос на выборку перед изменением и после, чтобы посмотреть, что в итоге изменилось*).

OUTPUT – инструкция T-SQL, позволяющая получить изменившиеся строки в результате выполнения инструкций INSERT, UPDATE, DELETE или MERGE.

Данная инструкция очень полезна в тех случаях, когда Вам необходимо проверить или просто знать, какие именно строки были добавлены, удалены или изменены, ведь инструкции по модификации данных такие сведения не возвращают.

Ранее я уже отмечал, что существуют временные таблицы Inserted и Deleted, которые хранят копии изменившихся строк, и имеют такую же структуру, как и целевая таблица. Таким образом, для того чтобы посмотреть изменения, мы должны просто обратиться к этим таблицам в инструкции OUTPUT, указав соответствующий префикс, примерно так же, как мы это делаем в инструкции SELECT, перечисляя названия столбцов.

Inserted указывается для того, узнать добавленные строки, и новые значения в случае с обновлением, а Deleted - для того, чтобы узнать удаленные строки, и старые значения в случае с обновлением данных.

Следующие примеры демонстрируют работу инструкции OUTPUT в сочетании с различными операциями по изменению данных (*пример с MERGE был представлен в предыдущем разделе*).

Пример использования с INSERT.

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

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

100 %

Результаты | Сообщения

	ProductId	CategoryId	ProductName	Price
1	4	1	Тестовый товар 1	300,00
2	5	1	Тестовый товар 2	500,00
3	6	2	Тестовый товар 3	400,00

Рис. 71

В этом примере мы добавляем три строки и сразу возвращаем результат, т.е. добавленные строки, инструкцией OUTPUT, для этого мы обратились к таблице Inserted. Инструкция OUTPUT указывается после инструкции INSERT и определения целевой таблицы.

Пример использования с UPDATE.

```
UPDATE TestTable SET Price = 0
     OUTPUT Inserted.ProductId AS [ProductId],
            Deleted.Price AS [Старое значение Price],
            Inserted.Price AS [Новое значение Price]
WHERE ProductId > 3
```

```
UPDATE TestTable SET Price = 0
       OUTPUT Inserted.ProductId AS [ProductId],
              Deleted.Price AS [Старое значение Price],
              Inserted.Price AS [Новое значение Price]
WHERE ProductId > 3
```

100 %

Результаты | Сообщения

	ProductId	Старое значение Price	Новое значение Price
1	4	300,00	0,00
2	5	500,00	0,00
3	6	400,00	0,00

Рис. 72

В данном случае мы обновили строки, идентификатор которых меньше 3, при этом с помощью OUTPUT мы вывели значения всех изменившихся строк, включая как старые значения, которые были до изменения, так и новые значения.

Пример использования с DELETE.

```
DELETE TestTable
     OUTPUT Deleted.*
WHERE ProductId > 3
```

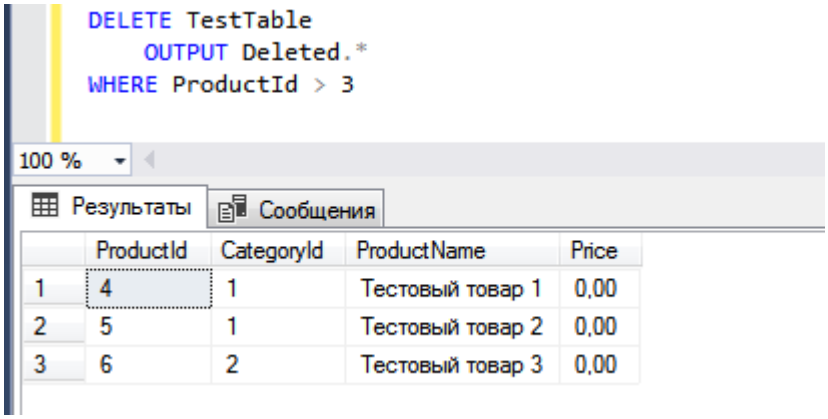


Рис. 73

Если в инструкции OUTPUT обратиться к таблицам Inserted или Deleted указав *, то, как и в случае с SELECT, выведутся все столбцы таблицы. В примере выше для того, чтобы посмотреть значения всех столбцов удаленных строк, указана *.

Глава 7 - Индексы

Как получить данные, а также как их изменить, мы научились, но все это мы делали на очень маленьком объеме данных (*3-5 записей*), за счет этого все наши запросы на выборку выполнялись мгновенно, но если этот объем будет большим, скажем, несколько миллионов строк, что на практике не редкость, запросы с каким-нибудь объединением и условием будут выполняться очень долго. В современном мире ждать получения информации (например, в программе Вы хотите открыть какой-нибудь отчет) по 2-3 минуты – это неприемлемо! Такие запросы должны быть оптимизированы, исключение составляет только запуск каких-нибудь расчетов на большом объеме данных, которые действительно могут выполняться достаточно долго.

Одним из главных методов оптимизации запросов на выборку является добавление индексов к таблице, а что это такое, мы сейчас и рассмотрим.

Индекс - это объект базы данных, который представляет собой структуру данных, состоящую из ключей, построенных на основе одного или нескольких столбцов таблицы, и указателей, которые сопоставляются с местом фактического хранения заданных данных. Иными словами, индексы – это некие ссылки на данные, своего рода предметный указатель, по которым можно быстро найти и получить фактические данные, т.е. строки таблицы. Основное назначение индексов - это как раз быстрый поиск необходимых строк таблицы.

Во время выполнения запроса оптимизатор (*это встроенный механизм в SQL сервере, который отвечает за разбор и превращение текста запроса в конкретные действия*) выбирает оптимальный и самый быстрый план выполнения запроса, т.е. получение данных. Если у таблицы есть индексы, оптимизатор обязательно выберет план, в котором задействуется индекс, а если индексов несколько, то он будет выбирать наиболее эффективный индекс для конкретного запроса.

Например, Вы постоянно делаете запрос на получение определенных столбцов таблицы, при этом запрос выполняется долго, Вы можете создать индекс, который будет включать один или все эти столбцы (*если их не так много*). И при выполнении запроса SQL сервер будет искать записи уже не в таблице, которая имеет много столбцов, которые нам не нужны, а в индексе, в котором он гораздо быстрей найдет необходимые указатели на исходные данные, и по ним обратится к нужным данным в таблице. А если индекс будет содержать абсолютно все данные (*т.е. все столбцы*), которые Вы хотите получить в запросе, то и обращаться к таблице SQL сервер не будет, а просто вернет данные из индекса, это, кстати, называется «*Покрытием запроса*», а такой индекс «*Покрывающим*», такие запросы самые быстрые, но злоупотреблять этим не стоит, так как с увеличением столбцов в индексе, размер индекса также растет.

Типы индексов

В Microsoft SQL Server существуют несколько типов индексов, а именно (*не пугайтесь, когда будете читать описание некоторых из них, после перечисления я расскажу, с какими Вы чаще всего будете сталкиваться*):

- **Кластеризованный** (*Clustered*) – это индекс, который хранит данные таблицы в отсортированном, по значению ключа индекса, виде. У таблицы может быть только один кластеризованный индекс, так как данные могут быть отсортированы только в одном порядке. По возможности каждая таблица должна иметь кластеризованный индекс, если у таблицы нет кластеризованного индекса, такая таблица называется «*кучей*»;
- **Некластеризованный** (*Nonclustered*) – это индекс, который содержит значение ключа и указатель на строку данных, содержащую значение этого ключа. У таблицы может быть несколько некластеризованных индексов. Создаваться некластеризованные индексы могут как на таблицах с кластеризованным индексом, так и без него. Именно этот тип индекса используется для повышения производительности часто используемых запросов, так как

некластеризованные индексы обеспечивают быстрый поиск и доступ к данным по значениям ключа;

- **Фильтруемый** (*Filtered*) – это оптимизированный некластеризованный индекс, который использует предикат фильтра для индексирования части строк в таблице. Если хорошо спроектировать такой тип индекса, то он может повысить производительность запросов, а также снизить затраты на обслуживание и хранение индексов по сравнению с полнотабличными индексами;
- **Уникальный** (*Unique*) – это индекс, который обеспечивает отсутствие повторяющихся (*одинаковых*) значений ключа индекса, гарантируя тем самым уникальность строк по данному ключу. Уникальными могут быть как кластеризованные, так и некластеризованные индексы. Если создавать уникальный индекс по нескольким столбцам, индекс гарантирует уникальность каждой комбинации значений в ключе. Уникальный индекс может быть создан только в том случае, если у таблицы на текущий момент отсутствуют дублирующие значения по ключевым столбцам;
- **Колоночный** (*Columnstore*) – это индекс, основанный на технологии хранения данных в виде столбцов. Данный тип индекса эффективно использовать для больших хранилищ данных, поскольку он может увеличить производительность запросов к хранилищу до 10 раз и также до 10 раз уменьшить размер данных, так как данные в Columnstore индексе сжимаются. Существуют как кластеризованные колоночные индексы, так и некластеризованные;
- **Полнотекстовый** (*Full-text*) – это специальный тип индекса, который обеспечивает эффективную поддержку сложных операций поиска слов в символьных строковых данных;
- **Пространственный** (*Spatial*) – это индекс, который обеспечивает возможность более эффективного использования конкретных операций на пространственных объектах в столбцах с типом данных geometry или geography. Данный тип индекса может быть создан только для пространственного столбца;
- **XML** – это еще один специальный тип индекса, который предназначен для столбцов с типом данных XML. Благодаря XML-индексу повышается эффективность обработки запросов к XML столбцам;
- Также существуют специальные индексы для таблиц, оптимизированных для памяти (*In-Memory OLTP*), такие как: Хэш (Hash) индексы и некластеризованные индексы, оптимизированные для памяти, которые создаются для сканирования диапазона и упорядоченного сканирования.

Индексы, с которыми Вы точно столкнетесь, это кластеризованный, некластеризованный и уникальный, они, наверное, есть во всех базах данных, так как они просто жизненно необходимы, остальные типы индексов уже более специфичны и их наличие зависит от того, какие данные хранит база данных, т.е. для чего она предназначена.

Создание индексов

Переходим к практике, но сразу скажу, что в реальных ситуациях сначала, перед созданием индекса, его необходимо правильно спроектировать, ведь неправильно спроектированный индекс может даже понизить производительность, а не увеличить ее. С рекомендациями по проектированию индексов Вы ознакомитесь чуть позже, в следующих разделах, а начинаю я с создания просто потому, что, для того чтобы проектировать индексы, нужно понимать, как они создаются.

В наших тестовых таблицах отсутствуют какие-либо индексы, давайте это исправим, и создадим соответствующие индексы.

В T-SQL индексы создаются с помощью инструкции CREATE. Сначала давайте создадим кластеризованный индекс для таблицы TestTable.

```
CREATE UNIQUE CLUSTERED INDEX IX_Clustered ON TestTable
  (
       ProductId ASC
  )
```

В данном случае мы написали инструкцию CREATE, затем указали, что за индекс мы хотим создать. Так как мы создаем кластеризованный индекс для ключевого столбца идентификаторов, то он у нас должен быть уникальным, поэтому мы сначала указали UNIQUE – т.е. этот индекс будет уникальным, а потом CLUSTERED, т.е. кластеризованный. Далее мы указали ключевое слово INDEX, для того чтобы SQL сервер понял, какой именно объект мы хотим создать. Затем мы указали имя индекса - IX_Clustered, данное имя должно быть уникальным в пределах таблицы, т.е. других индексов с таким названием у таблицы быть не должно. Инструкция ON TestTable говорит о том, что индекс необходимо создать для таблицы TestTable. Потом в скобочках мы указали название столбца, на основе которого у нас будет создан кластеризованный индекс, а также указали способ сортировки значений ключа, ASC – по возрастанию.

Кластеризованный индекс – это один из основных индексов и, можно сказать, обязательный тип индекса, который должен быть у таблицы. Кластеризованный индекс даже создается автоматически, если мы, например, при создании таблицы указываем конкретный столбец в качестве первичного ключа (*PRIMARY KEY - что это такое, мы рассмотрим чуть позднее*).

Индексы можно создавать также и с помощью SQL Server Management Studio, для этого в обозревателе объектов находим нужную таблицу и щелкаем правой кнопкой мыши по пункту «*Индексы*», выбираем «*Создать индекс*» и тип индекса, в нашем случае «*Кластеризованный*».

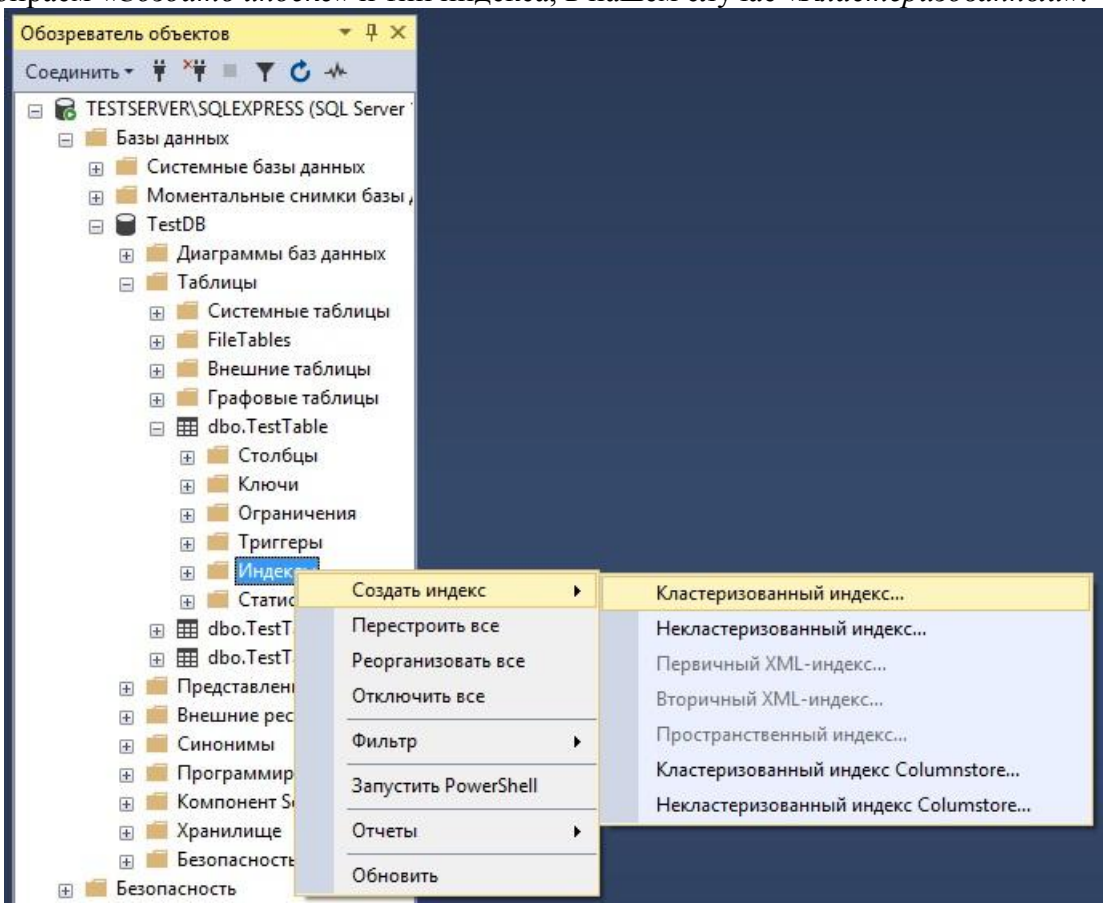


Рис. 74

Откроется форма «*Новый индекс*», где нам необходимо указать имя нового индекса, а также отметить параметр «*Уникальный*», т.е. будет ли этот индекс уникальным (Рис. 75). Потом с помощью кнопки «*Добавить*», выбираем столбец (*ключ индекса*), на основе которого у нас будет создан кластеризованный индекс (Рис. 76). После внесения всех данных нажимаем «ОК».

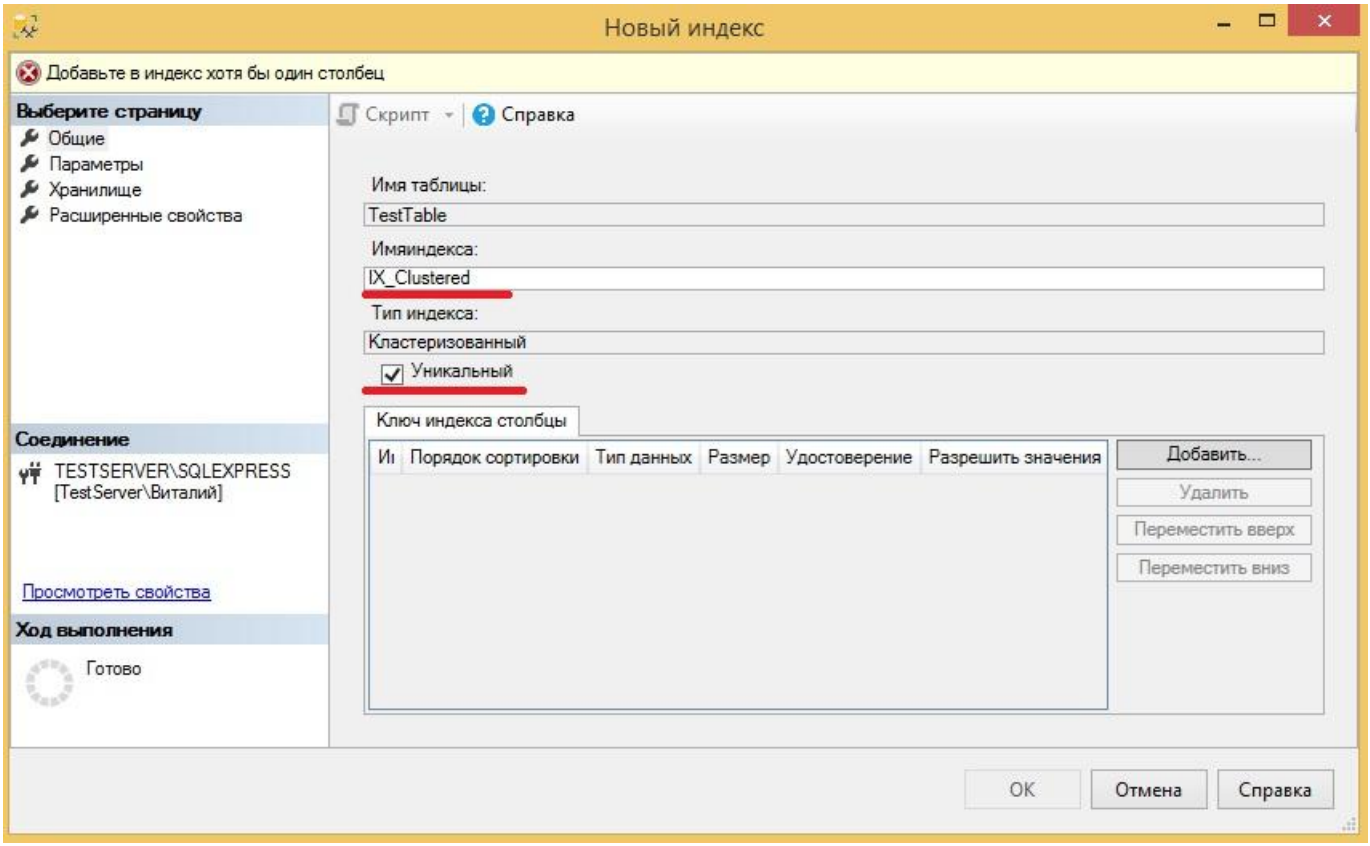


Рис. 75

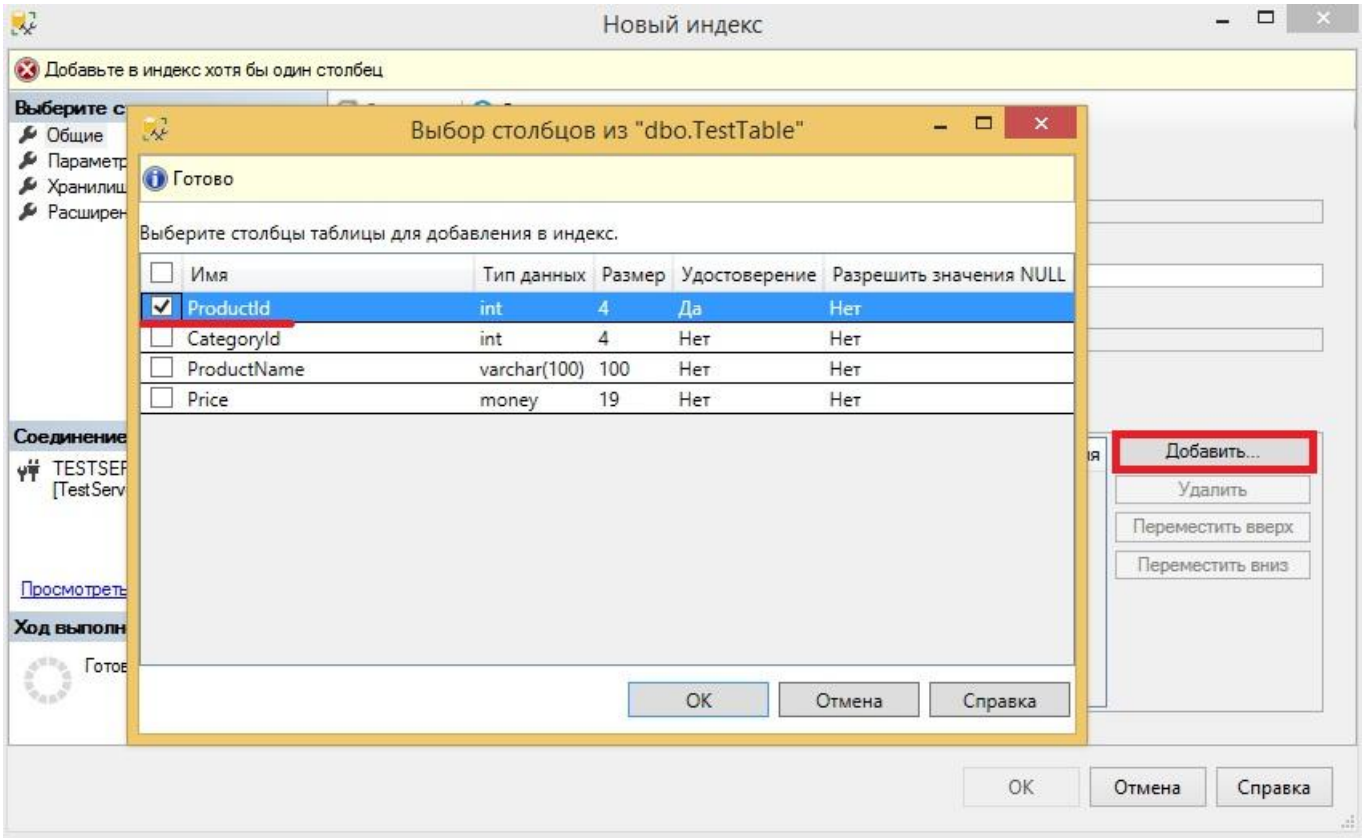


Рис. 76

Кластеризованный индекс научились создавать, но, честно скажу, его принудительно (*т.е. как мы только что*) Вы создавать будете редко, так как, я уже говорил, он автоматически создается, если мы указываем первичный ключ, а первичный ключ необходим практически для большинства таблиц.

Гораздо чаще Вы будете создавать некластеризованные индексы, которые как раз и используют для более быстрого доступа к данным.

Давайте представим, что к нашей таблице мы очень часто обращаемся с условием по категории (CategoryId), и данных у нас много. Для ускорения инструкций SELECT нам нужен некластеризованный индекс по ключу CategoryId. Его мы создадим следующим образом.

```
CREATE NONCLUSTERED INDEX IX_NonClustered ON TestTable
  (
       CategoryId ASC
  )
```

Где, NONCLUSTERED - означает, что мы создаем некластеризованный индекс, вся остальная структура в данном запросе такая же, как и при создании кластеризованного индекса, само собой имя индекса и столбец отличается.

Надеюсь, общий принцип создания индексов понятен. Идем дальше.

Удаление и изменение индексов

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

Допустим, нам нужно в наш некластеризованный индекс IX_NonClustered добавить один ключевой столбец (*ProductName*) и один неключевой столбец (*Price*).

Для этого нам нужно сначала удалить индекс, это делается следующим образом (*все действия также можно сделать и в Management Studio*).

```
DROP INDEX IX_NonClustered ON TestTable;
```

Мы указали инструкцию DROP, затем указали, какой именно объект мы хотим удалить, т.е. INDEX, далее написали название индекса и таблицу, для которой он создан. После этого мы можем создавать новый индекс.

```
CREATE NONCLUSTERED INDEX IX_NonClustered ON TestTable
(
     CategoryId ASC,
     ProductName ASC
)
     INCLUDE (Price);
```

Вы видите, что в скобочках в данном случае после CategoryId через запятую я указал еще один ключевой столбец ProductName, затем после того, как я скобочки закрыл, т.е. перечислил все ключевые столбцы, я написал ключевое слово INCLUDE, которое означает, что мы хотим добавить в индекс не ключевые столбцы, т.е. включенные. Далее в скобочках я также перечислил столбцы для включения в индекс.

Синтаксис T-SQL позволяет в одной инструкции и удалить индекс, и создать его заново, например, для того чтобы изменить его определение.

```
CREATE NONCLUSTERED INDEX IX_NonClustered ON TestTable
(
     CategoryId ASC,
     ProductName ASC
)
     INCLUDE ( Price )
WITH (DROP_EXISTING = ON);
```

Мы написали точно такой же запрос на создание индекса, только в конце добавили ключевое слово WITH, с помощью которого можно задать параметры индекса, в частности, в данном случае мы указали параметр DROP_EXISTING = ON, что означает, что индекс существует и его необходимо удалить и создать заново.

Проектирование индексов

Как я уже говорил, чтобы создать эффективный индекс, необходимо его хорошо спроектировать, а это можно сделать, если придерживаться определенных рекомендаций:

- Одним из самых эффективных индексов является индекс для целочисленных столбцов, которые имеют уникальные значения, поэтому по возможности создавайте индексы для таких столбцов;
- Если таблица очень интенсивно обновляется, то не рекомендуется создавать большое количество индексов, так как это снижает производительность инструкций INSERT, UPDATE, DELETE и MERGE. Потому что после изменений данных в таблице, SQL сервер автоматически вносит соответствующие изменения во все индексы;
- Если таблица с большим объемом данных обновляется редко, при этом она активно используется в инструкциях SELECT, т.е. на выборку данных, то большое количество индексов может улучшить производительность, так как у оптимизатора запросов будет больший выбор индексов при определении наиболее эффективного и быстрого способа доступа к данным;
- Для таблиц с небольшим объемом данных создание некластеризованных индексов с целью повышения производительности может оказаться абсолютно бесполезно, да еще и с затратами на их поддержание. Так как оптимизатору может потребоваться больше времени на поиск данных в индексе, чем просмотр данных в самой таблице. Поэтому не создавайте индексы для таблиц, в которых очень мало данных;
- Кластеризованный индекс необходимо создавать для столбца, который является уникальным и не принимает значения NULL, также длина ключа должна быть небольшой, другими словами, ключ индекса не нужно составлять из нескольких столбцов;
- Некластеризованные индексы лучше всего создавать для всех столбцов, которые часто используются в условиях (WHERE) и в объединениях (JOIN);
- По возможности не стоит создавать индексы, в которых очень много ключевых столбцов, так как это влияет на размер индекса и на ресурсы его поддержания;
- Эффективно использовать покрывающие индексы, т.е. индексы которые включают все столбцы, используемые в запросе. Благодаря этому оптимизатор запросов может найти все значения столбцов в индексе, при этом не обращаясь к данным таблиц, что приводит к меньшему числу дисковых операций ввода-вывода. Это можно достичь с помощью включения в индекс неключевых столбцов (*включенные столбцы*), но также следует принять во внимание, что это влечет за собой увеличение размера индекса;
- Если есть возможность, то рекомендовано заменять неуникальный индекс уникальным для той же комбинации столбцов, это обеспечивает оптимизатору запросов дополнительные сведения, что может сделать индекс более эффективным;
- При создании индекса учитывайте порядок ключевых столбцов, это повышает производительность индекса. Например, столбцы, которые используются в предложении WHERE в условиях поиска равно (=), больше (>), меньше (<) или находящихся в интервале (BETWEEN) или участвуют в соединении (JOIN), должны стоять первыми. Если таких несколько, то упорядочивайте их по уровню различности, т.е. от наиболее четкого к наименее четкому.

Обслуживание индексов

Индексы спроектированы и созданы, и у нас запросы работают быстро, но в результате выполнения операций обновления, добавления или удаления данных в таблицах SQL сервер автоматически вносит соответствующие изменения в индексы, и со временем все эти изменения могут вызвать фрагментацию данных в индексе, т.е. они окажутся разбросанными по базе данных. Фрагментация индексов влечет за собой снижение производительности запросов, поэтому периодически необходимо выполнять операции обслуживания индексов, а именно дефрагментацию. К таким можно отнести операции реорганизации и перестроения индексов.

- **Реорганизация индекса** – это процесс дефрагментации индекса, который дефрагментирует конечный уровень кластеризованных и некластеризованных индексов по таблицам, физически переупорядочивая страницы концевого уровня в соответствии с логическим порядком (*слева направо*) конечных узлов.
- **Перестроение индекса** – это процесс, при котором происходит удаление старого индекса и создание нового, в результате чего фрагментация устраняется.

В каких случаях использовать реорганизацию индекса, а в каких перестроение? Чтобы ответить на этот вопрос сначала необходимо определить степень фрагментации индекса, так как в зависимости от фрагментации индекса тот или иной метод дефрагментации будет предпочтительней и эффективней. Для определения степени фрагментации индекса можно использовать системную табличную функцию **sys.dm_db_index_physical_stats** (*что такое табличные функции, мы рассмотрим чуть позже*), которая возвращает подробные сведения о размере и фрагментации индексов. Например, используя следующий запрос, Вы можете узнать степень фрагментации индексов у всех таблиц в текущей базе данных.

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

В данном случае нас интересует столбец avg_fragmentation_in_percent, т.е. процентная доля логической фрагментации.

Так вот, Microsoft рекомендует:

- Если степень фрагментации менее 5%, то реорганизацию или перестроение индекса вообще не стоит запускать;
- Если степень фрагментации от 5 до 30%, то имеет смысл запустить реорганизацию индекса, так как данная операция использует минимальные системные ресурсы и не требует долговременных блокировок;
- Если степень фрагментации более 30%, то необходимо выполнять перестроение индекса, так как данная операция, при значительной фрагментации, дает больший эффект чем операция реорганизации индекса.

На текущем моменте своего обучения Вы пока можете не задумываться о том, какая степень фрагментации у того или иного индекса, а просто делать перестроение, тем более если у Вас небольшая база данных, и она не требует максимальной отдачи в режиме 24 часа в сутки. В данном случае Вы можете смело периодически выполнять операцию перестроения индексов.

Чтобы контролировать и автоматизировать процесс обслуживания всех индексов в базе данных можно разработать процедуру (*что такое процедуры мы также рассмотрим*), которая и будет все это делать.

Совет 9

Всегда перед запуском массовой операции, которая требует серьезных расчетов на большом объеме данных, необходимо заранее провести обслуживание индексов, т.е. запустить процесс реорганизации и перестроения.

А теперь давайте посмотрим, как проводятся операции реорганизации и перестроения индексов.

Реорганизация индекса.

```
ALTER INDEX IX_NonClustered ON TestTable
  REORGANIZE
```

Перестроение индекса.

```
ALTER INDEX IX_NonClustered ON TestTable
  REBUILD
```

Как видите, для этих операций мы использовали инструкцию **ALTER INDEX**, а команды **REORGANIZE** и **REBUILD,** соответственно, означают реорганизацию и перестроение.

Кстати, в процессе инструкции CREATE INDEX с предложением DROP_EXISTING, если вносятся изменения в его определение, происходит перестроение индекса.

Основные моменты, касающиеся индексов, мы рассмотрели, еще раз напоминаю, что индексы, это, конечно, очень хорошее дело, и они очень полезны, но создавать их нужно с умом, поэтому лучше придерживаться рекомендаций, которые мы рассмотрели в разделе «*Проектирование индексов*».

А сейчас давайте переходить к следующей теме.

Глава 8 - Ограничения

Такие объекты, как индексы, которые обеспечивают нам более быстрый доступ к данным, мы рассмотрели, теперь давайте рассмотрим объекты, которые помогают обеспечивать целостность данных.

Ограничения – это объекты в Microsoft SQL Server, которые задают правила допустимости определенных значений в столбцах. Иными словами, они формируют некое условие на те данные, которые можно ввести в таблицу и хранить в ней.

При планировании структуры таблицы, следует также продумать и допустимые значения, которые могут храниться в столбце, это не менее важно, чем планирование типа данных столбцов.

Ограничения обеспечивают автоматический механизм проверки данных при их внесении, это очень удобно и крайне важно.

Поэтому советую подружиться с ограничениями и использовать их на практике.

Типы ограничений

В SQL сервере существует несколько типов ограничений, с одним, кстати, Вы уже познакомились, помните, когда мы создавали таблицы, для столбцов мы указывали возможность принятия этим столбцом значения NULL, т.е. NOT NULL – данная конструкция является разновидностью ограничения.

Совет 10

При создании таблицы всегда явно указывайте ограничение «NOT NULL» у всех столбцов. По возможности, столбцы не должны принимать значения NULL, иными словами, указывайте NOT NULL, и только в исключительных случаях разрешайте хранения NULL значений.

Следующий тип ограничения, который мы рассмотрим, наверное, самый распространённый и важный – это PRIMARY KEY.

Ограничение PRIMARY KEY

PRIMARY KEY – это ограничение первичного ключа. Первичный ключ представляет собой столбец, значения которого гарантируют уникальность строк. PRIMARY KEY дает нам возможность идентифицировать каждую строку в таблице. Ранее я уже упоминал, что первичный ключ требуется практически для каждой таблицы, и он должен быть у нее один. Обычно первичный ключ, т.е. ограничение PRIMARY KEY, создают для столбца, который играет роль счетчика (*помните IDENTITY*), и он не может содержать значения NULL.

Создав ограничение PRIMARY KEY, Вы можете не беспокоиться о том, что в Вашей таблице товаров вдруг окажется два товара с одинаковым идентификатором, иными словами, два одинаковых товара.

Ограничение PRIMARY KEY может содержать набор столбцов, которые имеют значения, уникально идентифицирующие строку в таблице, т.е. первичный ключ может быть составной, состоящий из нескольких столбцов.

Ограничение FOREIGN KEY

Следующее ограничение **FOREIGN KEY** – это ограничение внешнего ключа. Данное ограничение задает столбец, по которому устанавливается связь с данными в другой таблице. В процессе чтения книги Вы должны были понять, что между таблицами в базе существует некая связь, помните, в наших тестовых таблицах есть общие столбцы, например, CategoryId, так вот, чтобы обеспечить полноценную связь и целостность данных, мы должны были добавить соответствующее

ограничение FOREIGN KEY. Например, у нас, если мы попытаемся добавить в таблицу TestTable новый товар с категорией, которой нет в списке категорий (*TestTable2*), ничего не произойдет. Но как Вы понимаете, это грубейшая ошибка, а самостоятельно контролировать процесс добавления и проверки данных довольно трудоёмкий процесс. Поэтому нам SQL сервер позволяет автоматизировать процесс контроля над данными, а именно: мы можем создать ограничение FOREIGN KEY, и тогда добавить такие некорректные данные будет просто невозможно.

Ограничение UNIQUE

Идем далее, на очереди у нас ограничение **UNIQUE** – это ограничение, которое обеспечивает уникальность значений в столбце или нескольких столбцах. С помощью данного ограничения Вы можете исключить повторяющиеся значения в столбце. В случае, если таблица уже создана, то при добавлении этого ограничения в столбце не должно быть повторяющихся значений.

Ограничение CHECK

CHECK – это проверочное ограничение, которое обеспечивает проверку выполнения определенных условий при добавлении значений в столбец. Иными словами, если Вам требуется чтобы в столбце хранились только определённые значения, например, по бизнес требованию категорически нельзя хранить товары с ценой 666, контролировать процесс добавления и изменения цены самостоятельно очень сложно, поэтому мы должны автоматизировать этот процесс, и лучше всего это сделать с помощью проверочного ограничения, которое просто не даст сохранить такое значение в указанный столбец. Ограничение CHECK проверяет не существование данных, как FOREIGN KEY, не уникальность данных как UNIQUE, а их корректность, с точки зрения заданных правил.

Создать проверочное ограничение CHECK можно с любым логическим выражением, которое возвращает значение TRUE или FALSE, например, в нашем случае с ценой, выражение будет таким Price <> 666, т.е. цена не должна равняться 666. При этом к одному столбцу мы можем применять несколько проверочных ограничений, например, к тому же столбцу с ценой мы можем создать еще одно логическое проверочное ограничение.

Ограничение DEFAULT

Есть еще одна разновидность ограничения в SQL сервер, хотя ограничением я бы не назвал эту возможность, скорей механизм корректного хранения и ввода данных. Это **DEFAULT** – значение по умолчанию. Например, в нашей тестовой таблице разрешены значение NULL для столбца с ценой, но строки с NULL - это не хорошо, поэтому мы можем добавить для данного столбца значение по умолчанию, допустим 0. Иными словами, всякий раз, когда будут добавлены записи с неопределенным значением в столбце Price в таблицу, вместо NULL автоматически будут вставлены значения 0. Если рассматривать DEFAULT как механизм избавления от значений NULL, то, да, наверное, это ограничение.

Подведем итог, мы с Вами рассмотрели 6 типов ограничений:

- Разрешение значений NULL – возможность принятия столбцом значений NULL;
- PRIMARY KEY – ограничение первичного ключа;
- FOREIGN KEY – ограничение внешнего ключа;
- UNIQUE – уникальность значений;
- CHECK – проверочное ограничение;
- DEFAULT – значение по умолчанию.

Совет 11

В процессе планирования таблицы всегда думайте о том, какие именно данные Вам нужны в этой таблице, и на основе проведенного анализа создавайте соответствующие ограничения, это избавит Вас от многочисленных проблем при добавлении данных и тем более, когда Вы будете анализировать эти данные, т.е. делать аналитические выборки.

Создание ограничений

Теорию про ограничения мы рассмотрели, и теперь Вы знаете, что ограничение в SQL сервер - это очень полезные и нужные объекты. Сейчас давайте рассмотрим процесс создания этих ограничений.

Как задается ограничение NOT NULL, Вы уже знаете, просто при создании таблицы или изменении столбца, нужно указать, может ли принимать столбец значения NULL или не может (*NOT NULL*). Для напоминания давайте сделаем столбец Price обязательным к заполнению, т.е. зададим ограничение NOT NULL.

```
ALTER TABLE TestTable ALTER COLUMN [Price] [Money] NOT NULL
```

Иными словами, мы просто выполняем инструкцию ALTER для таблицы и для столбца, указывая при этом новые характеристики этого столбца. Но учтите, что, если в момент выполнения данной инструкции в столбце будут присутствовать записи со значением NULL, возникнет ошибка, соответственно все такие значения нужно будет предварительно устранить, т.е. задать им какое-нибудь значение.

Переходим к следующему ограничению.

Ограничение PRIMARY KEY можно создать как при создании таблицы, так и после, путем ее изменения, т.е. добавления к ней ограничения.

При создании таблицы такие ограничения можно объявить двумя способами, на уровне столбца, и на уровне таблицы. На уровне столбца обычно добавляют ограничения, которые состоят из одного столбца.

Для примера давайте я приведу способ создания таблицы и ограничения PRIMARY KEY двумя вариантами, первый на уровне столбца, второй на уровне таблицы. Вы выберете и выполните ту инструкцию, запись которой Вам больше понравилась.

```
CREATE TABLE TestTable4(
     [CategoryId] [INT] IDENTITY(1,1) NOT NULL CONSTRAINT
PK_CategoryId PRIMARY KEY,
     [CategoryName] [VARCHAR](100) NOT NULL
);

CREATE TABLE TestTable4(
     [CategoryId] [INT] IDENTITY(1,1) NOT NULL,
     [CategoryName] [VARCHAR](100) NOT NULL,
     CONSTRAINT PK_CategoryId PRIMARY KEY (CategoryId)
);
```

Как видите, мы создаем еще одну таблицу TestTable4, и в этой таблице определено ограничение первичного ключа. Ограничение определяется с помощью ключевого стола CONSTRAINT. В первом случае ограничение мы добавляем после всех характеристик столбца, т.е. пишем CONSTRAINT, затем название этого ограничения, потом что это за ограничение. Во втором случае, мы после перечисления всех столбцов, как будто добавляем еще один столбец, но не столбец, а ограничение. Иными словами, пишем CONSTRAINT, название ограничение (*PK_CategoryId*), тип ограничения, и так как это объявление на уровне таблицы, мы указываем еще и столбец (*или столбцы*), который будет выполнять функцию первичного ключа. Хоть мы и не указывали, но у нас автоматически создался кластеризованный индекс, я об этом говорил в главе, посвященной индексам, т.е. когда мы создаем ограничение PRIMARY KEY, SQL сервер сам, автоматически, создает кластеризованный индекс. При этом нам никто не запрещает принудительно указать ключевое слово CLUSTERED после PRIMARY KEY, тем самым мы просто уточняем, что в данной инструкции будет создан еще и кластеризованный индекс.

Если таблица была уже создана без первичного ключа, как, например, мы это делали, когда создавали наши первые тестовые таблицы, то для добавления ограничения в T-SQL используется инструкция ALTER TABLE, т.е. изменение таблицы.

```
ALTER TABLE TestTable ADD CONSTRAINT PK_TestTable PRIMARY KEY
(ProductId)
```

Мы написали инструкцию ALTER TABLE, указали таблицу, в которую хотим внести изменения, затем написали команду ADD CONSTRAINT (*добавление ограничения*), указали имя ограничения (*PK_TestTable*), тип, и в скобочках перечислили столбцы, которые будут играть роль ключевого столбца, т.е. первичного ключа.

Двигаемся дальше, переходим к ограничению FOREIGN KEY.

Данное ограничение также можно определить, как во время создания таблицы, так и после, т.е. добавить это ограничение к таблице.

Сначала давайте создадим новую таблицу, в которой определим два ограничения, PRIMARY KEY и FOREIGN KEY. Ограничение внешнего ключа у нас будет ссылаться на идентификатор в таблице TestTable4, другими словами, мы говорим, что значения определённого столбца, которые будут добавляться в таблицу, обязательно должны уже существовать в TestTable4.

```
CREATE TABLE TestTable5(
     [ProductId] [INT] IDENTITY(1,1) NOT NULL,
     [CategoryId] [INT] NOT NULL,
     [ProductName] [VARCHAR](100) NOT NULL,
     [Price] [MONEY] NULL,
 CONSTRAINT PK_TestTable5 PRIMARY KEY (ProductId),
 CONSTRAINT FK_TestTable5 FOREIGN KEY (CategoryId) REFERENCES
TestTable4 (CategoryId)
     ON DELETE CASCADE
     ON UPDATE NO ACTION
)
```

В данном примере мы после определения первичного ключа (*PK_TestTable5*) через запятую написали ключевое слово CONSTRAINT, говоря этим, что мы хотим добавить еще одно ограничение, затем мы написали имя ограничения FK_TestTable5, потом тип, т.е. FOREIGN KEY, далее в скобочках указали столбец (*CategoryId*), который будет связан с другой таблицей и подлежит проверке. После этого мы указали ключевое слово REFERENCES, название таблицы (*TestTable4*) и столбца (*CategoryId*), с которым необходимо связать столбец из текущей таблицы, т.е. CategoryId. Инструкция ON DELETE CASCADE говорит о том, что если будет удалена запись из связанной (*родительской*) таблицы, т.е. TestTable4, то все записи по этому ключу будут удалены и в таблице TestTable5. Другими словами, если мы удалим категорию, то все товары в этой категории также удалятся. Но существуют и другие действия в таких случаях, например, не удалять, а присвоить значения NULL или значение по умолчанию. Инструкция ON UPDATE задает правила обновления родительской таблицы, т.е. также, если ключ будут обновлен, то необходимо выполнить определённые действия с записями в дочерней таблице.

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

Таким образом, мы имеем две связанные таблицы, и, если Вы попытаетесь добавить прямо сейчас данные в таблицу TestTable5, т.е. товар (при этом, как Вы понимаете, у нас нет ни одной категории), возникнет ошибка, иными словами, нам нужно сначала добавить категории в таблицу TestTable4.

Теперь давайте вернемся к нашим самым первым таблицам, и добавим точно такое же ограничение в таблицу TestTable.

```
ALTER TABLE TestTable2 ADD CONSTRAINT PK_TestTable2 PRIMARY KEY
(CategoryId);

ALTER TABLE TestTable ADD CONSTRAINT FK_TestTable FOREIGN KEY
(CategoryId)
     REFERENCES TestTable2 (CategoryId);
```

Сначала мы добавили ограничение первичного ключа в таблицу с категориями (*первая инструкция ALTER*), а затем добавили ограничение внешнего ключа в таблицу TestTable. Для этого мы сделали практически все то же самое, что мы делали ранее при создании таблицы. А именно, мы указали команду ADD CONSTRAINT, название ограничения, тип и столбец, затем написали REFERENCES, и указали таблицу и столбец, на который ссылаться. В данном случае мы не указывали инструкции ON DELETE и ON UPDATE, т.е. в случае удаления или изменения ключевого столбца будет появляться ошибка.

С помощью именно этого ограничения выстраивается связь между таблицами, оно, наверное, самое сложное, поэтому если Вам непонятно, как оно работает, перечитайте описание данного ограничения и примеры создания.

Совет 12

При планировании структуры базы данных всегда выстраивайте связь между таблицами, которые логически связаны между собой, и создавайте соответствующие ограничения.

Продолжаем. Ограничение **UNIQUE** создается синтаксически так же, как и перечисленные выше. Давайте создадим еще одну таблицу, в которой будет три столбца. Для первого мы создадим ограничение на уровне столбца, для второго на уровне таблицы, для третьего мы создадим ограничение уникальности отдельной инструкцией.

```
CREATE TABLE TestTable6(
     [Column1] [INT] NOT NULL CONSTRAINT PK_TestTable6_C1 UNIQUE,
     [Column2] [INT] NOT NULL,
     [Column3] [INT] NOT NULL,
     CONSTRAINT PK_TestTable6_C2 UNIQUE (Column3)
);

ALTER TABLE TestTable6 ADD CONSTRAINT PK_TestTable6_C3 UNIQUE
(Column3);
```

Как видите, все то же самое, мы пишем ключевое слово CONSTRAINT, имя ограничения и тип, т.е. UNIQUE. В итоге мы имеем таблицу, в которой мы создали три ограничения уникальности, их названия: PK_TestTable6_C1, PK_TestTable6_C2 и PK_TestTable6_C3.

Ограничение **CHECK** можно создать на уровне таблицы, и с помощью инструкции ALTER TABLE уже после создания таблицы. Давайте создадим таблицу с двумя столбцами, для первого мы определим проверочное ограничение в инструкции создания самой таблицы, а для второго уже после ее создания отдельной инструкцией.

```
CREATE TABLE TestTable7(
     [Column1] [INT] NOT NULL,
     [Column2] [INT] NOT NULL,
     CONSTRAINT CK_TestTable7_C1 CHECK (Column1 <> 0)
);

ALTER TABLE TestTable7 ADD CONSTRAINT CK_TestTable7_C2 CHECK (Column2
> Column1);
```

Мы создали два ограничения CK_TestTable7_C1 и CK_TestTable7_C2.

В первом случае, наше проверочное ограничение говорит, что в Column1 не должно быть значений равных 0, а ограничение для второго столбца говорит, что все значения, которые будут добавляться в Column2 должны быть больше чем в Column1. Иными словами, если мы захотим добавить запись со значениями, например, для Column1 = 5, а для Column2 = 3, возникнет ошибка.

Осталось нам только посмотреть, как создается ограничение **DEFAULT** – значение по умолчанию. Для этого давайте создадим еще одну таблицу, в которой будет два столбца. Для первого мы определим значение по умолчанию в процессе создания самой таблицы, а для второго с помощью инструкции ALTER TABLE.

```
CREATE TABLE TestTable8(
     [Column1] [INT] NULL CONSTRAINT DF_C1 DEFAULT (1),
     [Column2] [INT] NULL
);

ALTER TABLE TestTable8 ADD CONSTRAINT DF_C2 DEFAULT (2) FOR Column2;
```

Для первого столбца у нас значение по умолчанию 1, а для второго 2. Во втором случае с помощью ключевого слова FOR мы указали, для какого именно столбца создать ограничение.

Кстати, в первом случае мы могли написать сокращенную запись определения значения по умолчанию, на практике обычно так и делают.

```
CREATE TABLE TestTable8(
     [Column1] [INT] NULL DEFAULT (1),
     [Column2] [INT] NULL
);
```

Только в этом случае имя ограничения автоматически присвоит SQL сервер, поэтому если Вы хотите вести учет всех созданных ограничений, то лучше название придумывать самому (*т.е. как в первом случае DF_C1*), и указывать ключевое слово CONSTRAINT.

Удаление ограничений

Создавать ограничения мы научились, но иногда требуется эти ограничения удалить, например, изменились бизнес правила или что-то в этом роде.

Кстати, в процессе удаления всей таблицы удаляются и все связанные ограничения, и индексы. Поэтому удалять отдельно ограничения, если таблица уже удалена, нет необходимости.

Но если таблица остается, при этом ограничение больше не требуется, мы можем его принудительно удалить.

С удалением ограничений все намного проще, чем с их созданием, для этого мы используем команду **DROP CONSTRAINT** в инструкции ALTER TABLE. Например, ниже мы удалим ограничения из таблиц TestTable7 и TestTable8.

```
ALTER TABLE TestTable7 DROP CONSTRAINT CK_TestTable7_C1;

ALTER TABLE TestTable7 DROP CONSTRAINT CK_TestTable7_C2;

ALTER TABLE TestTable8 DROP CONSTRAINT DF_C1;

ALTER TABLE TestTable8 DROP CONSTRAINT DF_C2;
```

При удалении ограничения уже не нужно указывать тип ограничения, достаточно указать его название и таблицу, к которой оно относится. После ALTER TABLE мы указываем таблицу, после DROP CONSTRAINT имя ограничения.

Вот мы с Вами и рассмотрели очень важную и в то же время сложную тему языка T-SQL – это ограничения. Рекомендую перечитать раздел, и потренироваться создавать те или иные ограничения, так как при проектировании базы данных (*или просто какого-то функционала*) Вы обязаны будете учитывать и использовать ограничения, а при реализации базы данных Вам обязательно нужно будет создавать какие-нибудь ограничения, а если Вы захотите обойтись без них, то в скором времени у Вас будет не база данных, а просто набор непонятной и несогласованной информации.

Глава 9 - Программирование на T-SQL

Все, что мы рассматривали до этого, относилось скорей к реализации возможностей языка SQL в языке T-SQL, а начиная с этой главы, мы уже будем рассматривать все те расширенные возможности языка T-SQL, которые реализованы в Microsoft SQL Server.

Язык T-SQL - это не просто язык, с помощью которого мы можем получать и управлять данными в базе данных, это полноценный язык программирования, в нем есть все присущие остальным языкам программирования возможности: переменные, условные конструкции, циклы, обработка ошибок и многое другое. В данной главе мы с Вами познакомимся со всеми этими возможностями, иными словами, будем учиться программировать на T-SQL!

Если Вам предстоит разрабатывать некие алгоритмы, реализовывать бизнес логику, то Вам обязательно нужно знать все то, что мы рассмотрим в данной главе. После ее прочтения Вы научитесь программировать в базе данных!

Итак, поехали!

Переменные

Переменные – это специальные объекты, которые временно хранят информацию (*значение*), которые мы сначала сохраняем, а потом используем.

Без переменных в программировании, на любом языке программирования, не обойтись!

Они нам нужны для того, чтобы мы могли сохранить нужную нам информацию, которая понадобится нам чуть позже в этой же инструкции.

Перед тем как использовать переменную, мы сначала должны ее объявить, это делается с помощью ключевого слова **DECLARE**. Перед названием переменной мы указываем символ **@**, затем пишем имя переменной, в пределах одной инструкции это имя должно быть уникальным, и задаем тип данных этой переменной (*см. Главу «Типы данных в SQL Server»*). Всеми этими действиями мы говорим SQL серверу, что это переменная, и мы хотим ее использовать в этой инструкции.

Давайте посмотрим, как выглядит инструкция объявления переменной.

```
DECLARE @TestVar INT
```

В данном случае мы объявили переменную TestVar с типом данных INT.

Переменная у нас есть, теперь нам нужно ее использовать, для этого мы должны присвоить ей значение, которое мы хотим сохранить, т.е. которое нам понадобится чуть позже в этой инструкции. По умолчанию после объявления переменной ей присваивается значение NULL.

Присвоить или задать значение переменной можно с помощью двух команд, первая - это команда SET.

```
DECLARE @TestVar INT
SET @TestVar = 10
```

В этой инструкции мы присвоили переменной TestVar значение 10.

Вторая – Вы ее уже знаете, это инструкция SELECT.

```
DECLARE @TestVar INT
SELECT @TestVar = 10
```

В данной инструкции мы сделали ровно то же самое, что и чуть выше. Кстати, с помощью SELECT можно присваивать значение сразу нескольким переменным, но для одной переменной рекомендовано использовать SET.

Для того чтобы сразу задать значение переменной во время ее объявления, так сказать, значение по умолчанию, можно после ее объявления написать знак = и указать значение, например:

```
DECLARE @TestVar INT = 10
```

Здесь мы не использовали команды SET или SELECT, но в переменной TestVar у нас уже сразу после объявления хранится значение 10.

Как объявить переменную, и присвоить ей значение, мы теперь знаем, осталось узнать, как это значение извлечь, для того чтобы использовать в инструкции.

А использовать это значение можно просто обращаясь к этой переменной. Например, в инструкции нам необходимо значение, которое у нас в переменной, умножить на 5, мы просто берем переменную и умножаем на 5.

```
DECLARE @TestVar INT

SET @TestVar = 10

SELECT @TestVar * 5 AS [Результат]
```

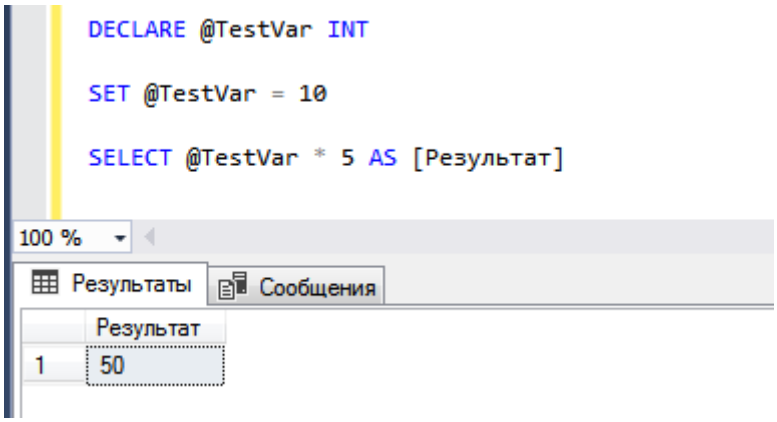


Рис. 77

В данной инструкции мы объявили переменную, присвоили ей значение 10, а затем умножили на 5, а результат вывели на экран с помощью инструкции SELECT (Рис. 77).

Переменные могут хранить не только одно значение, но и несколько, а именно целую таблицу, т.е. табличные данные, такие переменные называются - **табличные переменные.**

Табличные переменные имеют специальный тип данных TABLE. В целом табличные переменные похожи на временные таблицы, только создаются они путем объявления, а не с помощью инструкции CREATE TABLE, и имеют префикс @, а не #.

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

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

SELECT * FROM @TestTable
```

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

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

SELECT * FROM @TestTable
```

100 %

Результаты | Сообщения

	ProductId	CategoryId	ProductName	Price
1	1	1	Клавиатура	100,00
2	2	1	Мышь	50,00
3	3	2	Телефон	300,00

Рис. 78

В этой инструкции мы объявили переменную @TestTable с типом данных TABLE, и в скобочках указали структуру табличных данных, точно так же, как мы это делаем, когда создаем таблицу с помощью инструкции CREATE TABLE. Затем инструкцией INSERT мы вставили данные в табличную переменную, а потом извлекли их инструкцией SELECT.

Табличные переменные принудительно удалять не нужно, как, например, временные таблицы, они автоматически очищаются, когда пакет инструкций будет завершен. Если у Вас возникнет вопрос, что использовать в своих инструкциях: временные таблицы или табличные переменные, то рекомендация следующая, если данных много, и Вы будете их изменять, то лучше использовать временные таблицы, в подобных случаях они показывают более высокую производительность. Табличные переменные лучше использовать для хранения небольшого объема данных.

Сейчас мы с Вами рассмотрели переменные, которые мы можем сами создавать, они называется – **локальными переменными**. Они существуют только во время выполнения текущей инструкции, после выполнения которой мы к ним уже не сможем обратиться, они будут очищены и удалены. Но есть еще и **глобальные переменные**, к этим переменным мы можем обращаться уже в любой инструкции. Но таким переменным мы не можем присвоить свое значение, и их не нужно объявлять, они созданы SQL сервером и хранят системную информацию. Например:

```
SELECT @@SERVERNAME [Имя локального сервера],
       @@VERSION AS [Версия SQL сервера]
```

```
SELECT @@SERVERNAME [Имя локального сервера],
       @@VERSION AS [Версия SQL сервера]
```

100 %

Результаты | Сообщения

	Имя локального сервера	Версия SQL сервера
1	TESTSERVER\SQLEXPRESS	Microsoft SQL Server 2017 (RTM) - 14.0.1000.169 ...

Рис. 79

Как видите, такие переменные обозначаются уже с помощью двух символов @. В данном случае мы использовали две переменные @@SERVERNAME, которая возвращает имя локального сервера, и @@VERSION, которая возвращает версию SQL сервера.

Переменные мы с Вами рассмотрели, идем далее.

Комментарии

Сейчас мы рассмотрим еще одну возможность, которая есть во всех языках программирования – это возможность комментировать код.

Комментарии нужны для того, чтобы делать заметки в коде. Например, если инструкция очень большая или сложная, Вы можете просто забыть, для чего Вы написали тот или иной участок кода. Но если Вы сделаете комментарии, то Вам сразу станет понятно, что делает этот участок, и для чего Вы или кто-то другой его написали.

Например, Вы написали большую и сложную инструкцию, и забыли про неё, а через полгода Вам понадобилось внести изменения в эту инструкцию, по опыту я Вам могу сказать, что если инструкция действительно сложная и большая, то без комментариев Вам понадобится достаточно много времени на поиск нужного участка, которой необходимо изменить. Вам потребуется рассмотреть и понять каждый участок кода, чтобы найти тот, который Вам нужен. Даже если Вам нужно будет найти и изменить всего одну цифру, например, в переменной, без комментариев Вы еще подумаете, ту ли цифру Вы меняете или нет.

В языке T-SQL бывают: однострочные комментарии (--Текст) и многострочные комментарии (/*Текст*/).

```
--Объявление переменных: количество и сумма
DECLARE @Cnt INT, @Summa MONEY

SET @Cnt = 10
SET @Summa = 150

/*
     Выполняем операцию умножения.
     Пример многострочного комментария.
*/
SELECT @Cnt * @Summa AS [Результат]
```

В данном примере мы использовали и однострочный, и многострочный комментарий.

Совет 13

Всегда комментируйте свой код, свои SQL инструкции, там, где у Вас или у другого программиста, который будет читать это код, могут возникнуть вопросы.

Комментарии - это хорошо, это хороший стиль программирования! Однако стоит помнить, что непонятные комментарии - это плохо, и слишком много комментариев - это тоже не очень хорошо.

Операторы

Для того чтобы выполнять какие-нибудь операции над значением, в T-SQL есть специальные операторы, с которыми Вы, конечно же, знакомы из школьной программы математики.

- + Сложить – оператор сложения;
- - Вычесть – оператор вычитания;
- * Умножить – оператор умножения;
- / Делить – оператор деления;
- % Остаток – оператор возвращает целочисленный остаток при делении;
- = равно – оператор присваивания.

Операторы сравнения мы с Вами уже рассмотрели тогда, когда рассматривали условия WHERE (*помните: больше, меньше, IN и так далее*).

Пакеты

В Microsoft SQL Server все инструкции выполняются пакетами.

Пакет – это команды или инструкции SQL, которые передаются на SQL Server как одно целое, иными словами, SQL сервер их будет анализировать и компилировать, как единую инструкцию.

Например, помните, когда мы рассматривали переменные, я упоминал, что локальные переменные существуют только во время выполнения текущей инструкции, сейчас я могу уточнить, они существуют только во время выполнения текущего пакета.

Пакеты отделяются командой GO, однажды в этой книге я ее уже использовал. Если, например, Вы объявите переменную, затем напишите GO, т.е. скажите SQL серверу, что пакет завершен и сейчас после GO будет новый пакет, а после попытаетесь использовать переменную, у Вас возникнет ошибка. SQL сервер скажет Вам, что переменной с таким именем не существует, так как она была объявлена в другом пакете.

Если будет допущена синтаксическая ошибка хотя бы в одной инструкции, которая включена в пакет, то не выполнится весь пакет, так как SQL сервер компилирует весь пакет инструкции в так называемый план выполнения. После того как план сформирован, все инструкции последовательно выполняются согласно этому плану.

Также следует отметить, что некоторые инструкции SQL, например, CREATE VIEW, объединить с другими инструкциями в пакете нельзя, т.е. в данном случае CREATE должна быть первой инструкцией в пакете, а все, что после, SQL сервер рассматривает, как часть определения этой инструкции CREATE.

Команды условного выполнения

Это команды, с помощью которых Вы можете выполнять различные инструкции SQL в зависимости от выполнения или невыполнения определенных условий. Иными словами, если будет выполнено какое-нибудь условие, мы можем выполнить одну инструкцию, а если не будет, то другую.

IF…THEN

Эту конструкцию Вы, может, даже слышали - IF THEN, во всех языках программирования есть такая конструкция, просто синтаксис отличается.

В T-SQL синтаксис примерно следующий.

IF *Логическое выражение*
Инструкция 1
ELSE
Инструкция 2

Если логическое выражение истина, то выполняется инструкция 1, если нет, то инструкция 2, а если ключевое слово ELSE не указать и не определить инструкцию, то в случае невыполнения условия не выполнится ничего.

```
DECLARE @TestVar1 INT
DECLARE @TestVar2 VARCHAR(20)

SET @TestVar1 = 5

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

SELECT @TestVar2 AS [Значение TestVar1]
```

В данном примере мы объявили переменные, для первой задали значение 5, а затем, с помощью условной конструкции IF, мы проверяли значение данной переменной, а именно больше 0 это значение или нет.

Значение 5 больше 0, поэтому выполнилась первая инструкция, а именно присвоение второй переменой соответствующего текстового значения. Если в TestVar1 было бы значение меньше 0 или равным 0, то выполнилась вторая инструкция присваивания.

Например, в следующем случае давайте зададим переменной значение 0, но при этом уберем команду ELSE.

```
DECLARE @TestVar1 INT
DECLARE @TestVar2 VARCHAR(20)

SET @TestVar1 = 0

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

SELECT @TestVar2 AS [Значение TestVar1]
```

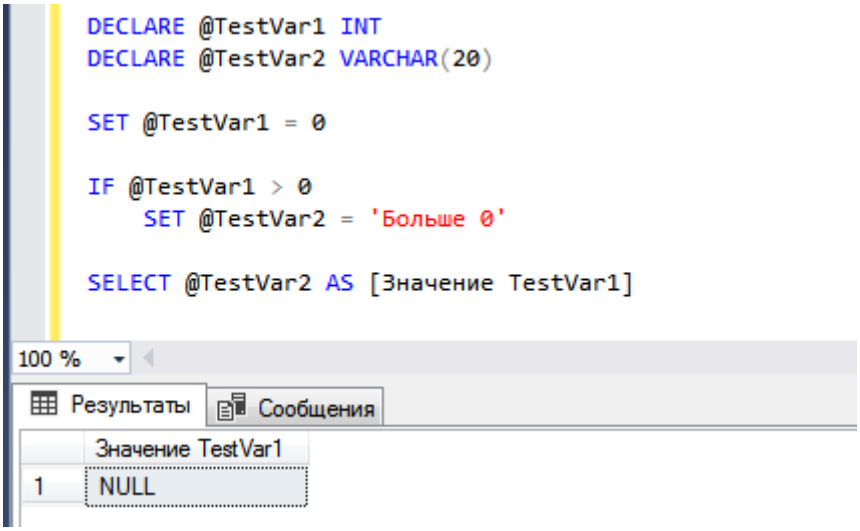


Рис. 80

В итоге в TestVar2 у нас ничего не записалось, так и остался NULL, так как наше условие не выполнилось.

Если Вам необходимо, чтобы в условии проверялся не один параметр, а несколько, то Вы можете комбинировать условия с помощью операторов AND или OR так же, как и в условии WHERE.

```
DECLARE @TestVar1 INT
DECLARE @TestVar2 VARCHAR(20)

SET @TestVar1 = -5

IF @TestVar1 > 0 OR @TestVar1 = -5
     SET @TestVar2 = 'Значение подходит'

SELECT @TestVar2 AS [Значение TestVar1]
```

В этом примере мы проверяем значение TestVar1 на то, является ли оно больше 0 или равняется -5, и если хоть одно из условий истина, то значение нам подходит, и инструкция присваивания выполняется.

IF EXISTS

Следующая условная конструкция IF EXISTS, она позволяет определить наличие записей в той или иной таблице. И на основе того, есть ли эти записи или нет, выполнить соответствующую инструкцию.

Например:

```
DECLARE @TestVar VARCHAR(20)

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

SELECT @TestVar AS [Наличие записей]
```

В данном примере мы с помощью IF EXISTS проверяем, есть ли записи в таблице TestTable, и на основе результата записываем в переменную TestVar соответствующее значение.

CASE

Если требуется выполнить несколько конструкций IF, то в T-SQL существует конструкция, упрощающая это действие, это CASE. Она подразумевает замену многократного использования конструкции IF.

```
DECLARE @TestVar1 INT
DECLARE @TestVar2 VARCHAR(20)

SET @TestVar1 = 1

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

SELECT @TestVar2 AS [Число]
```

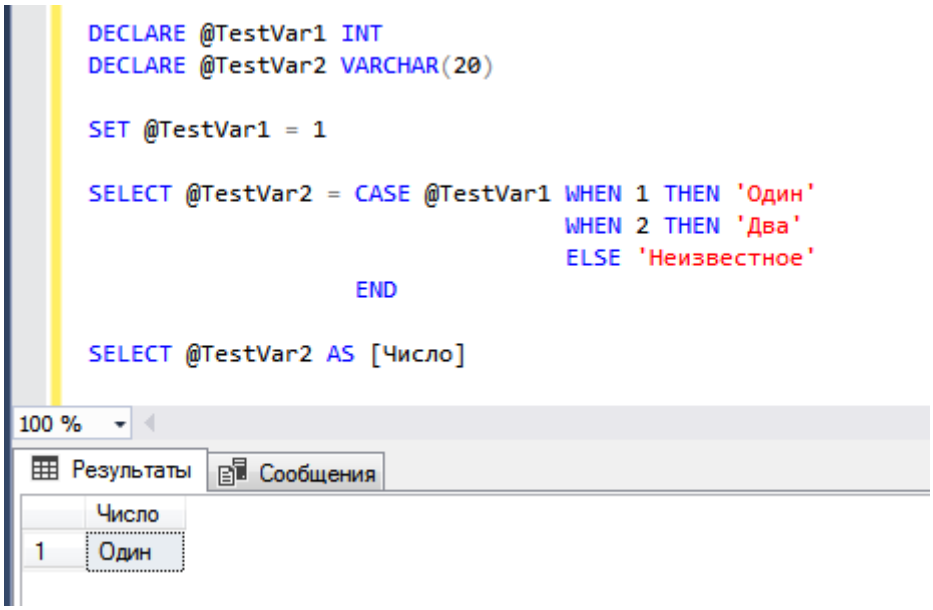


Рис. 81

Как видите, в данном случае мы уже используем инструкцию SELECT. После CASE мы написали переменную (*в обычном запросе это может быть столбец*), которую хотим проверять. Затем

мы начинаем проверять значения с помощью ключевого слова WHEN, и для этого указываем выражение для сравнения. После пишем THEN и выражение в случае выполнения условия. Таким образом, если TestVar1 = 1, то в TestVar2 запишется значение 'Один', если = 2, то 'Два', если не одно из условий не выполнилось, то 'Неизвестное', для этого мы указали ключевое слово ELSE. В заключении данной конструкции обязательно пишем END, это означает, что мы закончили CASE, иначе ошибка.

BEGIN...END

Эта конструкция означает блок команд или инструкций. Например, если нам нужно после IF выполнить не одну команду, а несколько, то мы должны заключить все инструкции в блок BEGIN...END. Так как после IF выполняется только самая первая инструкция, то вторая инструкция уже не будет являться частью IF.

BEGIN...END, как я уже сказал, позволяет нам указать столько инструкций, сколько нам нужно.

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

SET @TestVar1 = 5

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

END
SELECT @TestVar2 AS [Значение TestVar1],
        @TestVar3 AS [Значение TestVar2]
```

В данном случае мы проверяем значение переменной TestVar1 на то, чтобы оно не равнялось трем значениям (0, 1, 2), если условие выполняется, то выполнение инструкции переходит в блок BEGIN...END, и в нем последовательно выполняются все инструкции.

Циклы

Если Вам необходимо многократно выполнять одно и то же действие в зависимости от выполнения определённого условия, то Вам на помощь придут циклы.

Цикл в T-SQL – это повторное выполнение SQL инструкции или блока инструкций (*что такое блок Вы уже знаете*). К примеру, для определенных строк в таблице Вам необходимо выполнить какие-то сложные расчёты, и массово, т.е. с помощью обычного SQL запроса, это сделать не получается. Вручную обрабатывать каждую строку, т.е. для каждой записи выполнять определённый блок инструкций - это очень долго и не удобно. Или Вам над этими строками необходимо выполнять одно и то же действие до тех пор, пока Вы не достигните того результата, который Вам нужен, т.е. пока Ваше условие выполнится. Или Вам нужно выполнить одно и то же действие определенное количество раз, например, добавить в таблицу 1000 и более практически одинаковых записей, т.е. по сути выполнить 1000 раз инструкцию INSERT. Вы, конечно, можете написать одну инструкцию INSERT, но это инструкция будет очень большая, к тому же на ее написание у Вас уйдет немало времени. Вы также можете 1000 раз самостоятельно выполнить инструкцию INSERT, но это тоже очень долго.

Поэтому в T-SQL, как и любом другом языке программирования, есть возможность цикличного выполнения команд. Другими словами, мы можем написать одну инструкцию или блок, и запустить цикл, выполнение этого блока инструкций будет повторяться до тех пор, пока выполняется условие, в случае со строками, пока не обработаются все необходимые строки, в случае с выполнением одного и того же действия, пока не будет достигнут необходимый результат, в случае с выполнением определенного количества одних и тех же действий, пока это действие не выполнится столько раз, сколько мы указали.

Однократное выполнение всего блока инструкций называется **итерацией цикла**, т.е. когда весь блок выполнился, прошла одна итерация, блок начинает выполняться заново, начинается новая итерация. Иными словами, если нам нужно добавить 1000 строк с помощью цикла, будет выполнено 1000 итераций.

В отличие от других языков программирования в языке T-SQL есть только один тип цикла - это **WHILE с предусловием**, это означает, что команды начнутся, и будут повторяться до тех пор, пока выполняется условие перед началом цикла, т.е. сначала проверяется условие, затем выполняются инструкции или прекращается цикл, если условие не выполняется.

Инструкции в цикле могут быть очень сложными, мы можем там использовать и условия и даже вложенные циклы, при этом T-SQL нам позволяет управлять циклом. Иными словами, мы можем принудительно прекратить выполнение цикла, если, скажем, сработало какое-нибудь условие, или пропустить выполнение итерации, т.е. если сработало определенное условие, мы можем завершить текущую итерацию и перейти к следующей, не выполняя при этом оставшиеся инструкции в текущей итерации.

Для того чтобы принудительно выйти из цикла в T-SQL, используется ключевое слово **BREAK**. Для того чтобы пропустить итерацию - ключевое слово **CONTINUE**.

Теперь, я думаю, пора переходить к практике.

```
DECLARE @CountAll INT = 0

--Запускаем цикл
WHILE @CountAll < 10
BEGIN

     SET @CountAll = @CountAll + 1

END

SELECT @CountAll AS [Результат]
```

В данном примере мы объявили переменную, и сразу задали ей значение = 0. Затем мы проверяем, если значение в переменной меньше 10, то входим в цикл и выполняем блок инструкций. В инструкции мы увеличиваем значение переменной на 1 и сохраняем полученное значение обратно в переменную. Итерация закончена, перед следующей мы снова проверяем переменную, если значение в процессе прошедшей итерации не увеличилось до 10, то продолжаем цикл. Значение в переменной, как Вы понимаете, в нашем случае в каждой итерации изменяется, а именно увеличивается, в итоге, когда значение стало равняться 10, цикл прекратился, так как условие не выполнилось.

Теперь давайте посмотрим, как мы можем принудительно завершить цикл.

```
DECLARE @CountAll INT = 0

-- Запускаем цикл
WHILE @CountAll < 10
BEGIN

     SET @CountAll += 1

     IF @CountAll = 5
          BREAK
END

SELECT @CountAll AS [Результат]
```

В данном случае мы делаем ровно то же самое, прошу заметить, что здесь я использовал сокращенную запись присваивания с помощью составного оператора +=, он также прибавляет значение

к переменной и сохраняет его обратно в переменную (*записи равнозначные, в подобных случаях можете использовать ту, которая Вам больше нравится*).

Затем в блоке инструкций мы проверяем значение переменной, если оно равняется 5, то выходим из цикла, для этого мы указали ключевое слово BREAK.

Идем далее.

```
DECLARE @Cnt INT = 0
DECLARE @CountAll INT = 0

--Запускаем цикл
WHILE @CountAll < 10
BEGIN

     SET @CountAll += 1

     IF @CountAll = 5
          CONTINUE

     SET @Cnt += 1

END

SELECT @CountAll AS [CountAll],
        @Cnt AS [Cnt]
```

Этот пример показывает, как можно пропустить итерацию и перейти к следующей. Для демонстрации я объявил ещё одну переменную Cnt, и в блоке инструкций мы с этой переменной делаем точно такие же действия, как и с переменной CountAll, только тогда, когда значение в переменной CountAll у нас станет равняться 5, мы закончим итерацию и перейдем к следующей, т.е. все инструкции ниже не будут выполнены, и значение в переменной Cnt не увеличится. В результате мы видим, что итоговые значения переменных отличаются.

Совет 14

Всегда перед запуском новой SQL инструкции, в которой есть цикл, проверяйте условие, при котором цикл должен завершиться, для того чтобы избежать попадания в бесконечный цикл.

Команда PRINT

PRINT – это инструкция для вывода служебных сообщений клиентскому приложению. В среде Management Studio это сообщение отображается на вкладке «*Сообщения*» (*Messages*).

Обычно PRINT применяется для отладки кода на Transact-SQL. Иными словами, в своей SQL инструкции Вы можете использовать PRINT для того, чтобы узнать какое-нибудь промежуточное значение, которое используется в определенном участке кода инструкции. Если у Вас инструкция большая, а результат этой инструкции Вам непонятен, то PRINT может оказаться полезной командой для диагностики проблемы.

Для того чтобы использовать PRINT, Вам достаточно указать в качестве параметра текстовую строку, которую Вы хотите вывести в сообщении.

Например

```
DECLARE @TestVar INT = 1

IF @TestVar  > 0
     PRINT 'Значение переменной больше 0'
ELSE
     PRINT 'Значение переменной меньше или равно 0'
```

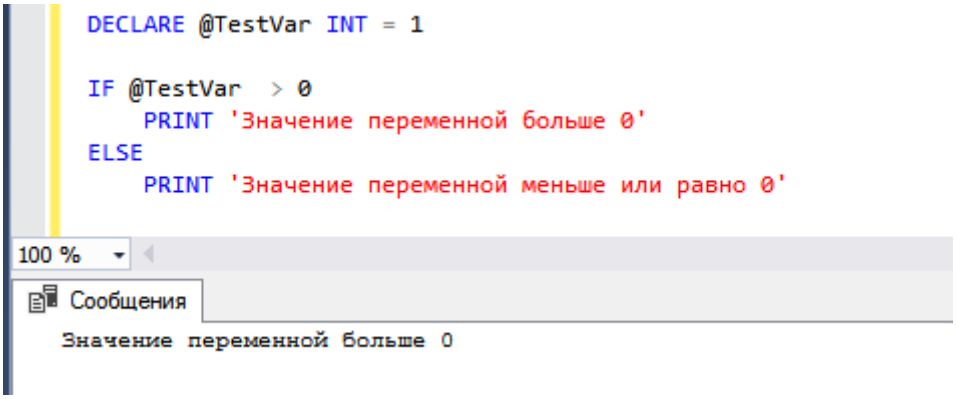


Рис. 82

В данном случае мы с помощью условной конструкции IF проверяем значение переменной @TestVar, и с помощью команды PRINT выводим соответствующее сообщение.

Команда RETURN

Команда RETURN предназначена для безусловного выхода из пакета или блока инструкций. С помощью RETURN мы можем прекратить выполнение инструкции из любой точки этой инструкции, все то, что будет идти после RETURN, выполняться не будет.

Данную команду полезно использовать тогда, когда Вам в процессе выполнения определенной инструкции необходимо ее полностью прекратить, в случае возникновения определённой ситуации.

Например

```
DECLARE @TestVar INT = 1

IF @TestVar < 0
     RETURN

SELECT @TestVar AS [Результат]
```

В этом примере мы проверяем значение переменной @TestVar, и если оно меньше 0, то мы полностью прекращаем выполнение инструкции, например, все действия ниже незачем выполнять, если @TestVar меньше 0, т.е. все команды абсолютно бессмысленны. Но если @TestVar больше или равно 0, то инструкцию необходимо выполнять далее, и в нашем случае оператор SELECT выведет на экран нам результат.

Команда GOTO

GOTO – эта команда осуществляет перевод выполнения инструкции в определенное место к определенной метке. Иными словами, если Вам вдруг нужно пропустить выполнение части инструкции или возвратиться назад к определённому участку кода, для того чтобы выполнить его еще раз, команда GOTO будет Вам полезна.

Для использования GOTO в своей инструкции Вам необходимо поставить метку или метки, к которым Вы хотите переходить в случае наступления нужного Вам события. Затем, если событие наступило, Вы пишете GOTO и выполняете переход к метке. Таким образом, инструкция, если Вы указали GOTO, продолжается, начиная с указанной метки.

С помощью GOTO можно даже организовать своего рода цикл, т.е. проверять выполнение определенного условия и в случае, если условие истина, то возвращаться назад к метке и повторять все действия заново.

```
DECLARE @TestVar INT = 0

METKA: --Устанавливаем метку

SET @TestVar += 1 --Увеличиваем значение переменной

--Проверяем значение переменной
IF @TestVar < 10
     --Если оно меньше 10, то возвращаемся назад к метке
     GOTO METKA

SELECT @TestVar AS [Результат]
```

Здесь мы установили метку METKA, и затем, когда нам нужно, мы осуществляли переход к этой метке с помощью команды GOTO. То есть мы проверяем значение переменной @TestVar и, если оно нас не устраивает, мы возвращаемся назад к метке, все, что после GOTO, не выполняется. Инструкция выполнилась только тогда, когда значение @TestVar нас устроило, и SELECT выполнился.

В следующем примере мы просто пропустим участок кода при выполнении определённого условия.

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

IF @TestVar <= 0
     GOTO METKA

SET @Rez = 10 / @TestVar

METKA: --Устанавливаем метку

SELECT @Rez AS [Результат]
```

Если значение переменной @TestVar меньше или равно 0, то следующий участок кода мы пропускаем и сразу переходим к указанной метке, если значение больше 0, то код продолжается, не обращая внимания на метки.

Честно скажу, GOTO я редко использую, тем более что рекомендуется использовать данную команду только в самых крайних случаях, т.е. если есть возможность обойтись без GOTO, обходитесь без нее, замените GOTO инструкцией IF или WHILE.

Команда WAITFOR

Бывают случаи, что процесс выполнения пакета SQL инструкций нужно приостановить на определенное время или до наступления заданного времени, иными словами, поставить на паузу выполнение инструкции. Язык T-SQL позволяет это сделать. Для этого используется команда WAITFOR, которая блокирует инструкцию до наступления заданного времени.

Для того чтобы сделать паузу на заданный период времени, нужно указать параметр DELAY у команды WAITFOR, для того чтобы приостановить выполнение пакета до указанного времени, нужно указать параметр TIME. Значения этих параметров задаются в формате hh:mi:ss

```
--Пауза на 5 секунд
WAITFOR DELAY '00:00:05'
     SELECT 'Продолжение выполнение инструкции' AS [Test]

--Пауза до 10 часов
WAITFOR TIME '10:00:00'
     SELECT 'Продолжение выполнение инструкции' AS [Test]
```

В первом случае показано, как приостановить инструкцию на определённое количество времени, в нашем случае на 5 секунд. Во втором мы приостанавливаем выполнение инструкции до 10 часов. Когда будете выполнять данный запрос, не забудьте его остановить (*просто нажать кнопку отменить*), или измените время на подходящее для Вас, а то инструкция **будет заблокирована до 10 часов утра!**

Обработка ошибок

В языке T-SQL, как и в других языках программирования, есть возможность отслеживать и перехватывать ошибки, которые могут возникнуть в процессе выполнения SQL инструкции. Здесь подразумевается не синтаксические ошибки, о которых нам сразу говорит SQL сервер, если допущена синтаксическая ошибка, инструкция даже не начнёт выполняться. Речь здесь идет о непредвиденных ошибках, которые потенциально могут возникнуть на том или ином участке кода.

Такие непредвиденные ситуации нужно отслеживать, чтобы избежать таких же непредвиденных результатов.

Например, все мы знаем, что осуществить деление на ноль не получится, поэтому мы можем тот участок кода, где может возникнуть такая ситуация, заключить в обработчик ошибок, который отследит эту ошибку и выполнит действия, которые мы предусмотрим на этот случай, а выполнение SQL инструкции не прекратится, хотя мы можем задать и такое развитие события.

В T-SQL для обработки ошибок используется конструкция **TRY...CATCH**, если кто знаком с другими языками программирования, то Вам эта конструкция должна быть знакома, так как она используется во многих языках программирования.

Суть данной конструкции следующая: код, в котором могут возникнуть ошибки, Вы заключаете в блок TRY, начало данного блока обозначается ключевыми словами BEGIN TRY, а окончание блока соответственно END TRY.

Затем Вы пишите блок CATCH, в котором указываете инструкции, которые необходимо выполнить в случае появления ошибки. Его границы обозначается словами BEGIN CATCH и END CATCH.

Пример использования конструкции TRY...CATCH для обработки ошибок.

```
--Начало блока обработки ошибок
BEGIN TRY
     --Инструкции, в которых могут возникнуть ошибки
     DECLARE @TestVar1 INT = 10,
             @TestVar2 INT = 0,
             @Rez INT

     SET @Rez = @TestVar1 / @TestVar2

END TRY
--Начало блока CATCH
BEGIN CATCH
     -- Действия, которые будут выполняться в случае возникновения
ошибки    SELECT ERROR_NUMBER() AS [Номер ошибки],
                 ERROR_MESSAGE() AS [Описание ошибки]
     SET @Rez = 0
END CATCH

SELECT @Rez AS [Результат]
```

```
--Начало блока обработки ошибок
BEGIN TRY
    --Инструкции, в которых могут возникнуть ошибки
    DECLARE @TestVar1 INT = 10,
            @TestVar2 INT = 0,
            @Rez INT

    SET @Rez = @TestVar1 / @TestVar2

END TRY
--Начало блока CATCH
BEGIN CATCH
    -- Действия, которые будут выполняться в случае возникновения ошибки
    SELECT ERROR_NUMBER() AS [Номер ошибки],
           ERROR_MESSAGE() AS [Описание ошибки]
    SET @Rez = 0
END CATCH

SELECT @Rez AS [Результат]
```

100 %

Результаты | Сообщения

	Номер ошибки	Описание ошибки
1	8134	Обнаружена ошибка: деление на ноль.

	Результат
1	0

Рис. 83

В данном примере операцию деления, в которой потенциально могут возникнуть ошибки, мы поместили в блок TRY, в блоке CATCH мы указали действия в случае ошибки, а именно мы выводим номер и описание ошибки с помощью встроенных системных функций **ERROR_NUMBER()** и **ERROR_MESSAGE()** (*о том, что такое функции и какие они бывают, мы с Вами поговорим в будущих главах*), а также присваиваем переменной с итоговым результатом значение 0. Следует отметить, что блок CATCH должен идти сразу же за блоком TRY.

Совет 15

Если в SQL инструкции используются сложные и важные алгоритмы, всегда используйте конструкцию обработки ошибок, тем самым Вы избежите непредвиденных результатов.

Вот мы с Вами и рассмотрели основные конструкции программирования на языке T-SQL, знание которых поможем Вам писать сложные, а главное правильные и нужные SQL инструкции, функции и процедуры, кстати, именно к этой части языка T-SQL мы сейчас и будем переходить, т.е. будем разбирать, что такое функции, процедуры, а также Вы познакомитесь и с другими не менее интересными объектами в SQL сервере.

Впереди Вас ждет еще очень много полезных вещей, без знания которых Вы не сможете полноценно программировать на языке T-SQL. В данной главе мы рассмотрели только общие конструкции программирования, но само программирование на языке T-SQL заключается не в этом, а в написании алгоритмов, реализации функционала, и в это входит как раз все, что мы рассмотрели до этого и все, что мы еще рассмотрим далее. Разработать какой-нибудь функционал без использования функций, процедур, триггеров просто не получится, так же, как и без конструкций, которые мы рассмотрели в данной главе, и уж тем более без знания языка SQL (*SELECT, UPDATE, INSERT, DELETE и так далее*).

Поэтому продолжаем.

Глава 10 - Функции в языке T-SQL

Данная глава посвящена очень полезным объектам в Microsoft SQL Server - функциям. Рекомендую уделить особое внимание функциям и хранимым процедурам, которые мы будем рассматривать в следующей главе, так как это механизмы, с помощью которых Вы будете реализовывать тот или иной функционал, ту или иную возможность. Без применения функций и процедур программировать на языке T-SQL у Вас не получится! Функционал практически любого приложения предполагает использование функций и процедур, даже если Вы не будете сами реализовывать эти функции или процедуры, то Вы однозначно будете пользоваться ими, поэтому очень важно понимать, как работают функции и процедуры.

После прочтения этой главы Вы будете иметь представление о том, как работают функции в SQL сервере, для чего они нужны и, конечно же, научитесь писать их сами!

Что такое функции в T-SQL и типы функций

Функция – это автоматизированный механизм получения определённой информации, иными словами, в функции заложен некий алгоритм, с помощью которого вычисляется итоговое значение.

Допустим, Вам нужно выполнить определенную SQL инструкцию, в которой заложен алгоритм определения стоимости товара, и в этом алгоритме задействовано много параметров, которые влияют на эту стоимость, соответственно, этот алгоритм предназначен для одной записи в таблице, т.е. одного товара. А теперь представьте, что Вам нужно вывести в запросе такую вот стоимость для 1000 товаров, конечно же, 1000 раз выполнять этот запрос для каждого товара не стоит, можно использовать подзапрос, но если Вам нужно будет в SQL инструкции несколько раз получать такую информацию или в другой инструкции Вам нужны точно такие же данные, то Вам придётся дублировать код, т.е. текст этого подзапроса. Это неудобно, тем более представьте, что если вдруг алгоритм необходимо изменить, то Вам, соответственно, нужно будет менять все инструкции, в которых использован этот алгоритм. Вместо этого, Вы можете написать функцию, в которой пропишете весь этот алгоритм один раз, и везде, где нужно применить этот алгоритм, просто будете вызывать эту функцию с соответствующими параметрами.

Функции предназначены для упрощения расчетов, если мы вынесли алгоритм в функцию, нам уже не нужно изменять все инструкции, в которых применяется этот алгоритм, нам достаточно внести изменения в код функции, чтобы она формировала итоговое значение с учетом этих изменений.

Функции в SQL сервере бывают двух типов:

- **Пользовательские** – это функции, которые пишем мы, т.е. пользователи;
- **Системные (*встроенные*)** – это функции, которые уже встроены в SQL сервер.

Пользовательские функции

Сначала предлагаю рассмотреть пользовательские функции, т.е. научится их писать, а потом уже перейти к системным функциям, которые расширяют возможности языка T-SQL и очень полезны.

Пользовательские функции в свою очередь делятся на функции, которые **возвращают одно скалярное значение**, и на функции, которые возвращают табличное значение, т.е. результирующий набор данных в виде таблицы, такие функции называются – **табличные функции.**

Создание

Функции можно создавать как с параметрами, так и без них. Параметры нужны нам для того, чтобы алгоритм функции менялся в зависимости от этих параметров или просто выполнял заложенный алгоритм для конкретной записи в таблице.

Например, давайте представим, что нам в запросах очень часто требуется выводить наименование товара, при этом в этих запросах данного поля нет, а есть только идентификатор товара. Так вот, чтобы вывести наименование, нам нужно выполнять или объединение с таблицей, в которой хранится данная информация, или выполнять подзапрос. Но оба этих способа требуют написания дополнительных конструкций на языке T-SQL, а также если подобная информация нужна нам в SQL инструкции в нескольких местах или в нескольких инструкциях, то эти способы оказываются не совсем удобны и не практичны.

Поэтому в данном случае лучше всего написать функцию, которая будет возвращать наименование товара по его идентификатору, идентификатор товара мы будем передавать в функцию в виде параметра. В итоге везде, где нам необходимо вывести наименование товара, мы будем просто обращаться к этой функции.

Пример создания функции, которая возвращает скалярное значение, т.е. одно значение.

```
CREATE FUNCTION TestFunction
     (
          @ProductId INT --Объявление входящих параметров
     )
RETURNS VARCHAR(100) --Тип возвращаемого результата
AS
BEGIN
     --Объявление переменных внутри функции
     DECLARE @ProductName VARCHAR(100);

     --Получение наименования товара по его идентификатору
     SELECT @ProductName = ProductName
     FROM TestTable
     WHERE ProductId = @ProductId

     --Возвращение результата
     RETURN @ProductName
END

GO

--Вызов функции. Получение наименования конкретного товара
SELECT dbo.TestFunction(1) AS [Наименование товара]

--Вызов функции. Передача в функцию параметра в виде столбца
SELECT ProductId,
       ProductName,
       dbo.TestFunction(ProductId) AS [Наименование товара]
FROM TestTable
```

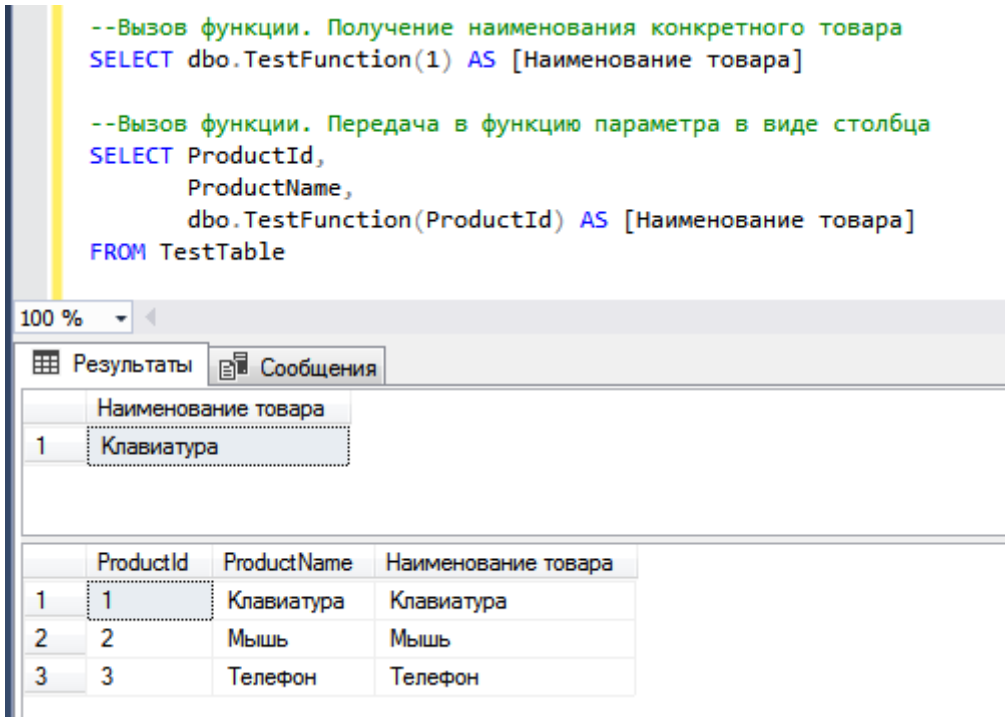


Рис. 84

Функция создается с помощью инструкции **CREATE FUNCTION**, затем мы указали название функции TestFunction. Далее в скобочках мы перечисляем параметры через запятую, с указанием типа данных данного параметра. В нашем случае параметр всего один - это ProductId. После этого мы пишем ключевое слово RETURNS, и указываем тип возвращаемого значения, т.е. какой тип данных будет иметь итоговое результирующее значение, в нашем случае это текстовая строка VARCHAR(100).

Далее мы пишем ключевое слово AS, и открываем блок инструкций словом BEGIN, в конце определения функции мы обязательно закрываем этот блок словом END. Внутри этого блока мы располагаем все наши инструкции, которые будут реализовывать наш алгоритм. В нашем случае мы просто обращаемся к таблице за наименованием товара по идентификатору, который мы получаем во входящем параметре (@ProductId). Входящие параметры внутри функции ведут себя как обычные переменные, соответственно, мы их можем использовать в своем коде так же, как и переменные, отличие от переменных в том, что их не нужно объявлять, мы их уже объявили, когда определяли входящие параметры.

С помощью команды **RETURN** мы возвращаем результат, и прекращаем выполнение функции.

Также в примере выше я показал, как обращаться к этой функции, т.е. просто пишем ее и в скобочках указываем, что будет выступать в качестве входящего параметра. Следует отметить, что тип данных должен соответствовать типу, который мы указали при создании функции, или хотя бы неявно преобразовываться. В указанном примере, конечно же, не совсем логично использовать такую функцию, так как именно в таблице TestTable хранится нужная нам информация, зато так наглядно видно, как работает эта функция.

Надеюсь, принцип того, как работает и создается скалярная функция, понятен. Если нет, то прочитайте заново описание, посмотрите комментарии, которые я указал в примере, ведь функции, как я уже говорил, Вы точно будете использовать, их количество в базе данных может достигать нескольких сотен, они очень полезны, можете даже прервать чтение и потренироваться создавать свои собственные тестовые функции.

Примечание! *В функциях нельзя использовать операции по изменению данных (INSERT, DELETE, UPDATE).*

Совет 16

Любой код, который полностью повторяется 2 и более раз в одной или нескольких SQL инструкциях, выносите в функцию, т.е. напишите функцию и везде, где нужно использовать этот код, вызывайте эту функцию.

В T-SQL есть возможность с помощью функции возвращать не одно значение, а целую таблицу, как я уже говорил, мы можем писать табличные функции. Они нужны нам для того, чтобы мы могли получать какие-то расчетные табличные данные, или просто сформированные с определенным условием на основе входящих параметров.

Например, нам периодически нужны товары из определённой категории, мы, конечно, можем написать представление, но в этом случае оно нам будет возвращать всегда данные только по одному условию, т.е. по одной категории, которую мы укажем в определении представления, а нам нужно получать данные из разных категорий, например, сейчас по одной, а завтра по другой. Поэтому в данном случае нам лучше написать табличную функцию, которая будет возвращать данные по товарам в категории, которую мы укажем во входящем параметре. Таким образом, мы получаем своего рода динамическое представление.

Пример создания функции, которая возвращает табличное значение, т.е. таблицу.

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

)
GO

--Пример обращения к табличной функции
SELECT * FROM FT_TestFunction(2)
```

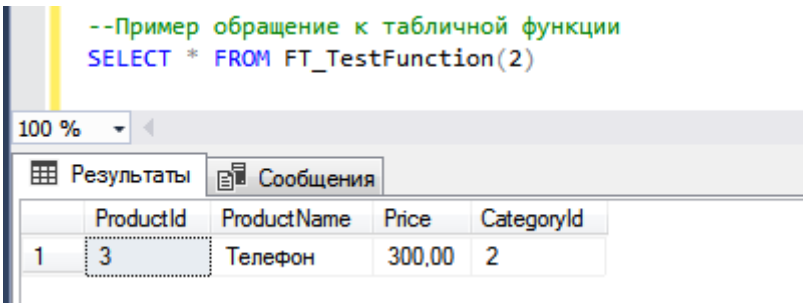


Рис. 85

Как видите, табличные функции создаются точно так же, как и обычные, т.е. инструкцией CREATE FUNCTION. Название функции FT_TestFunction, входящий параметр идентификатор категории (@CategoryId), он объявляются так же, как и параметры в скалярных функциях, в скобочках после названия. А вот далее идет ключевое отличие, в качестве типа возвращающего значения мы должны указать тип TABLE, иными словами, после команды RETURNS должно идти ключевое слово TABLE. Затем мы пишем AS и возвращаем результат командой RETURN, в скобочках после этой команды мы пишем необходимый нам запрос с использованием входящего параметра.

Обращаться к этой функции можно точно так же, как к таблице или представлению (Рис. 78), за исключением одного отличия, мы должны передать параметр в эту функцию. В примере видно, что нам возвращаются данные из категории с идентификатором 2.

Главным недостатком такого типа функций является то, что мы в ней не можем программировать, т.е. использовать условия, циклы или просто несколько инструкций. Такая табличная функция возвращает результирующий набор данных сформированного запроса.

Но T-SQL, как Вы уже, наверное, догадались, это очень мощная технология, поэтому мы можем написать табличную функцию, в которой можно будет программировать, для этого мы должны при указании типа возвращающего значения указать структуру возвращающихся данных, т.е. все столбцы, которые будут возвращаться, и их тип.

Давайте создадим еще одну похожую функцию, но только уже с использованием дополнительных конструкций языка T-SQL.

Пример создания табличной функции, которая включает несколько инструкций.

```
CREATE FUNCTION FT_TestFunction2
     (
          --Объявление входящих параметров
          @CategoryId INT,
          @Price MONEY

     )
     --Определяем результирующую таблицу
RETURNS @TMPTable TABLE (ProductId INT,
                         ProductName VARCHAR(100),
                         Price MONEY,
                         CategoryId INT
                         )
AS
BEGIN
     --Если указана отрицательная цена, то задаем цену равной 0
     IF @Price < 0
          SET @Price = 0

     --Заполняем данными результирующую таблицу
     INSERT INTO @TMPTable
          SELECT ProductId,
                 ProductName,
                 Price,
                 CategoryId
          FROM TestTable
          WHERE CategoryId = @CategoryId
            AND Price <= @Price
     --Возвращаем результат и прекращаем выполнение функции
     RETURN
END

GO
--Пример обращение к табличной функции
SELECT * FROM FT_TestFunction2(2, 200)
```

Мы создали функцию с названием FT_TestFunction2, и определили в ней два входящих параметра, @CategoryId и @Price. Затем, после команды RETURNS, мы объявляем результирующую таблицу, я ее назвал для примера @TMPTable, она содержит 4 столбца.

Как обычно, далее мы пишем ключевое слово AS, а затем открываем блок командой BEGIN, закрываем блок командой END в конце определения функции. Внутри данного блока мы уже можем полноценно программировать на T-SQL!

Для формирования результирующего набора данных нам необходимо заполнить таблицу @TMPTable, которую мы определили в качестве результата этой функции. Это делаем с помощью

инструкции INSERT (*напоминаю, что осуществлять модификацию данных в хранящихся в базе данных таблицах, в функциях, нельзя*).

После всех инструкций завершаем работу функции командой RETURN.

Чтобы подытожить тему создания функций, я перечислю несколько основных этапов создания функции, запомнив последовательность которых, Вы легко сможете писать их вручную, не тратя время на то, чтобы вспомнить, как синтаксически пишется функция, т.е. последовательность команд.

1 этап. Функции создаются с помощью инструкции CREATE FUNCTION.

2 этап. В случае необходимости объявляются входящие параметры.

3 этап. Должен быть определен итоговый результат (*тип данных результата*).

4 этап. Должно быть определено тело функции, другими словами, инструкции, реализующие алгоритм работы функции.

Запомнив эти 4 этапа, Вы по памяти будете писать свои собственные функции.

Теперь Вы знаете, как создаются функции. Сейчас давайте рассмотрим возможность изменения этих функций в случаях, когда изменен алгоритм работы функции или выявлена ошибка в ее работе.

Изменение

В функции, как и в таблицы, как и в представления, изменения вносятся с помощью инструкции ALTER. Суть заключается в следующем, если функция существует в базе данных, для ее изменения Вы вместо CREATE FUNCTION пишите **ALTER FUNCTION**, а все остальное меняете по необходимости, иными словами, все определение функции остается таким же, как и при ее создании, за исключением внесенных Вам изменений.

Помните, в самой первой функции, которую мы создали, мы получали наименование товара, давайте представим, что нам поставили задачу изменить эту функцию так, чтобы она возвращала наименование категории по идентификатору товара, а не наименование товара. Для этого нам необходимо всего лишь изменить тело функции, т.е. ее алгоритм.

Пример изменения функции.

```
ALTER FUNCTION TestFunction
     (
          @ProductId INT --Объявление входящих параметров
     )
RETURNS VARCHAR(100) --Тип возвращаемого результата
AS
BEGIN
     --Объявление переменных
     DECLARE @CategoryName VARCHAR(100);

     --Получение наименования категории товара по идентификатору
товара
     SELECT @CategoryName = T2.CategoryName
     FROM TestTable T1
     INNER JOIN TestTable2 T2 ON T1.CategoryId = T2.CategoryId
     WHERE T1.ProductId = @ProductId

     --Возвращение результата
     RETURN @CategoryName
END
GO

--Пример использования функции
SELECT ProductId,
       ProductName,
       dbo.TestFunction(ProductId) AS [CategoryName]
FROM TestTable
```

В данном случае мы изменили название переменной для наглядности, а также изменили запрос на получение необходимых данных. После того, как мы изменили функцию, мы можем сразу проверить ее работу. В примере мы написали запрос, в котором мы обращаемся к одной таблице, которая не содержит наименования категории, однако с помощью функции мы выводим это наименование для каждого товара, путем передачи идентификатора товара в эту функцию.

Таким образом, мы в основном запросе избегаем объединения с таблицей, в которой содержится наименование категории, за счет использования функции.

Совет 17

При внесении изменений в алгоритм работы функции помните о том, где и для каких целей эта функция используется, так как внесенные изменения отразятся во всех инструкциях, в которых задействована эта функция. Иными словами, не допускайте ситуаций, когда изменение алгоритма работы функции вносит корректные поправки в одни инструкции, для которых Вы и внесли эти изменения, а для других инструкций изменение алгоритма влечет некорректную работу этих SQL инструкций.

Удаление

В случае необходимости функции можно удалить, это делается так же, как и удаление всех остальных объектов в Microsoft SQL Server, с помощью команды **DROP**.

Давайте представим, что нам уже больше не нужна табличная функция FT_TestFunction2, для ее удаления мы напишем следующую инструкцию.

```
DROP FUNCTION FT_TestFunction2
```

Скалярные и любые другие пользовательские функции удаляются точно также, как видите, удалять всегда проще, чем создавать))

А сейчас давайте перейдем к функциям, которые уже есть в системе, и которые очень сильно помогают нам в разработке нужных нам алгоритмов - к системным функциям.

Системные функции

Многие алгоритмы, которые чаще всего требуются в повседневной работе, разработчики Misrosoft SQL Server уже реализовали в виде системных функций. Например, всем и всегда будет требоваться получить сумму значений в столбце (*помните, мы уже затрагивали тему агрегирующих функций*) или выполнить какие-нибудь операции над текстом в строке, или получить какую-нибудь системную информацию, все это можно сделать как раз с помощью системных функций, и сейчас мы кратко их рассмотрим. Сразу скажу, что системных функций очень много, и рассмотреть их все в данной книге просто не получится, да и не стоит. Я расскажу Вам про часто используемые функции сгруппированные по назначению.

Агрегатные функции

С данной группой функций Вы уже знакомы, это агрегатные функции или статистические, давайте для повторения я перечислю их еще раз.

- COUNT() – вычисляет количество значений в столбце (*значения NULL не учитываются*). Если написать COUNT(*), то будут учитываться все записи, т.е. все строки;
- SUM() – суммирует значения в столбце;
- MAX() – определяет максимальное значение в столбце;
- MIN() – определяет минимальное значение в столбце;
- AVG() – определяет среднее значение в столбце.

Строковые функции

Название данной группы функций говорит само за себя, эти функции работают со строками, иными словами, с текстовыми значениями в столбце или переменной.

LTRIM и RTRIM

Начну я с функций LTRIM и RTRIM. Нередко случается, что в текстовом столбце в таблице в начале или в конце строки, пользователи могут занести пробелы (*например, текстовое значение при вводе в базу было скопировано*), а они также являются символами, и при обработке подобных значений могут возникнуть некоторые проблемы. Поэтому в SQL сервере есть специальные функции, которые в запросе могут возвращать данные без этих пробелов, т.е. убрать все пробелы в начале, и в конце.

Функция LTRIM убирает все пробелы в начале строки, функция RTRIM в конце. Для примера давайте напишем инструкцию, в которой мы объявим текстовые переменные, присвоим им значения с пробелами и выведем результат без использования функций, и с использованием этих функций.

Во многие системные функции необходимо передавать параметры так же, как мы это делали, когда создавали свои собственные функции. В функции LTRIM и RTRIM, соответственно, нужно передать саму текстовую строку, из которой нужно удалить пробелы.

```
DECLARE @TestVar VARCHAR(100),
        @TestVar2 VARCHAR(100)

--Присвоение значений с пробелами
SELECT @TestVar = '          Текст',
       @TestVar2 = ' Текст          '

--Без использования функции
SELECT @TestVar AS TestVar,
       @TestVar2 AS TestVar2

--С использованием функций
SELECT LTRIM(@TestVar) AS TestVar,
       RTRIM(@TestVar2) AS TestVar2
```

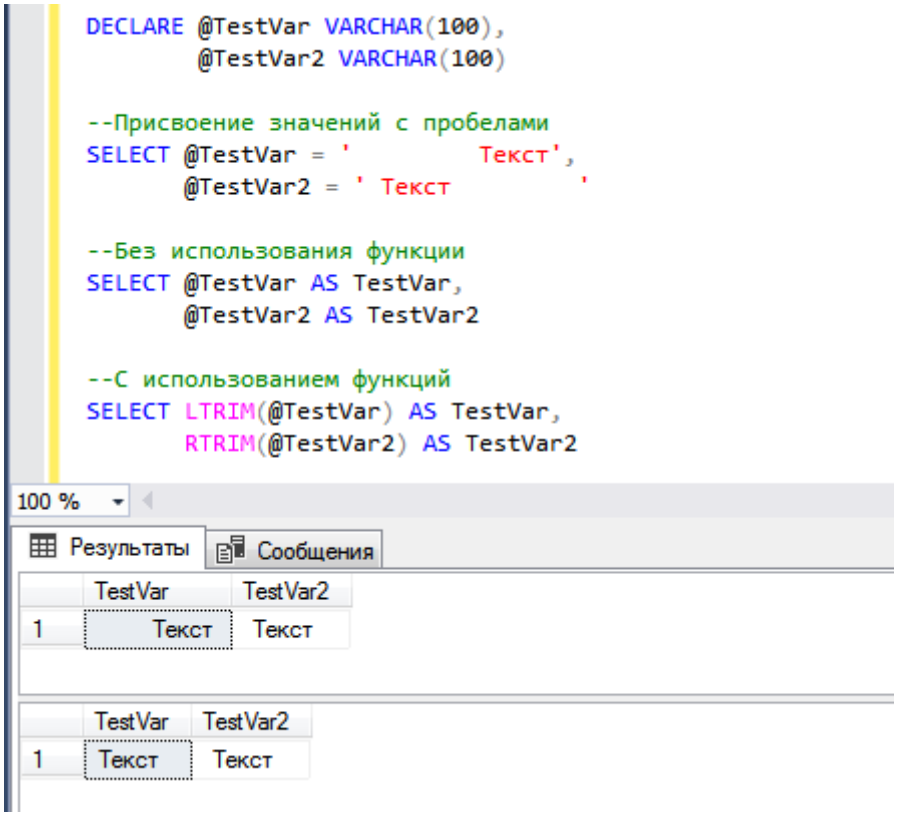


Рис. 86

Даже в среде Management Studio, когда Вы запустите этот запрос, Вы увидите визуальное отличие (Рис. 86).

LOWER и UPPER

Бывают случаи, что нужно значение всей строки получить в определённом регистре, в верхнем или в нижнем. Для этих целей в T-SQL есть встроенные функции LOWER и UPPER.

- UPPER - возвращает все символы указанной строки в верхнем регистре;
- LOWER - возвращает все символы указанной строки в нижнем регистре.

```
DECLARE @TestVar VARCHAR(100),
        @TestVar2 VARCHAR(100)

--Присвоение значения
SELECT @TestVar = 'ТеКст',
       @TestVar2 = 'ТЕкст'

--Без использования функции
SELECT @TestVar AS TestVar,
       @TestVar2 AS TestVar2

--С использованием функций
SELECT UPPER(@TestVar) AS TestVar,
       LOWER(@TestVar2) AS TestVar2
```

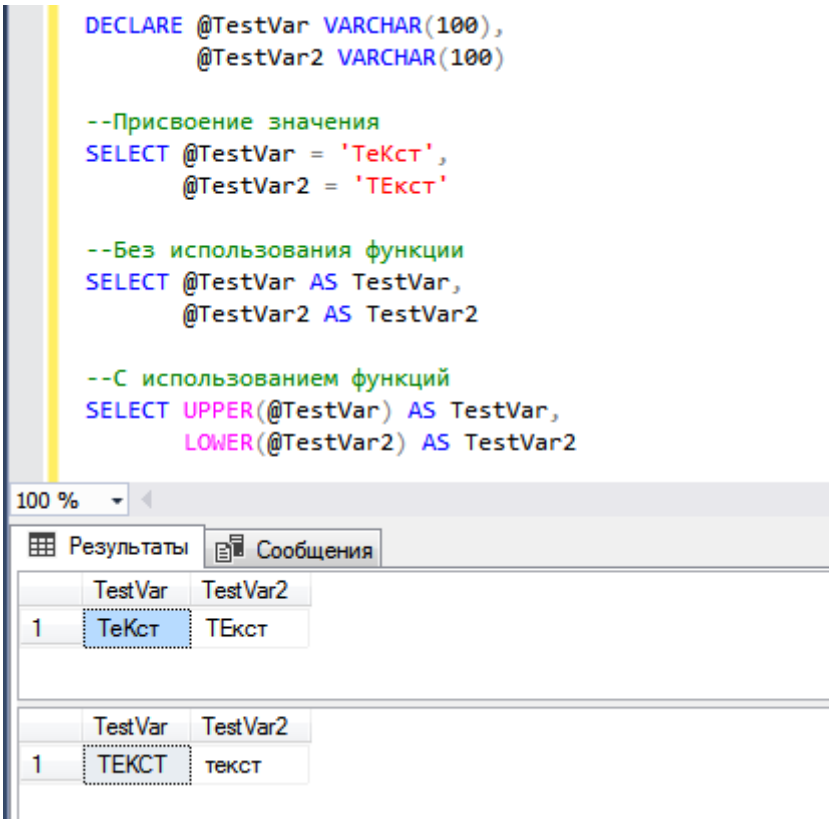


Рис. 87

В данном примере также наглядно видно, как работают данные функции.

LEN

Для того чтобы определить количество символов в строке, можно использовать функцию LEN. Например:

```
SELECT LEN('123456789') AS [Количество символов]
```

LEFT и RIGHT

Если Вам нужно получить не всю строку, а только первые несколько символов или последние несколько символов, можно использовать функции LEFT и RIGHT. LEFT возвращает символы слева, RIGHT, соответственно, справа.

Эти функции уже принимают два параметра, первый это строка, а второй количество символов, которые нужно получить.

```
DECLARE @TestVar VARCHAR(100),
        @TestVar2 VARCHAR(100)

--Присвоение значений
SELECT @TestVar = '1234567890',
       @TestVar2 = '1234567890'

--Выводим первые и последние 5 символов
SELECT LEFT(@TestVar, 5) AS TestVar,
       RIGHT(@TestVar2, 5) AS TestVar2
```

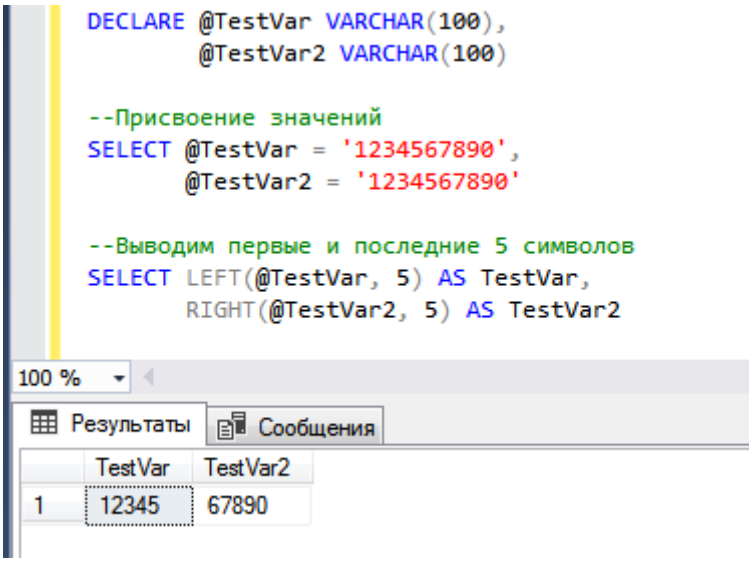


Рис. 88

SUBSTRING

С помощью функции SUBSTRING можно вывести определенную часть строки, например, начиная с указанной позиции, определенное количество символов. В данную функцию необходимо передать три параметра, первый - строка, второй – начальная позиция, третий - количество символов.

```
DECLARE @TestVar VARCHAR(100)

--Присвоение значения
SELECT @TestVar = '1234567890'

--Выводим 5 символов начиная с 3 символа
SELECT SUBSTRING(@TestVar, 3, 5) AS TestVar
```

Результат представлен на рисунке 89.

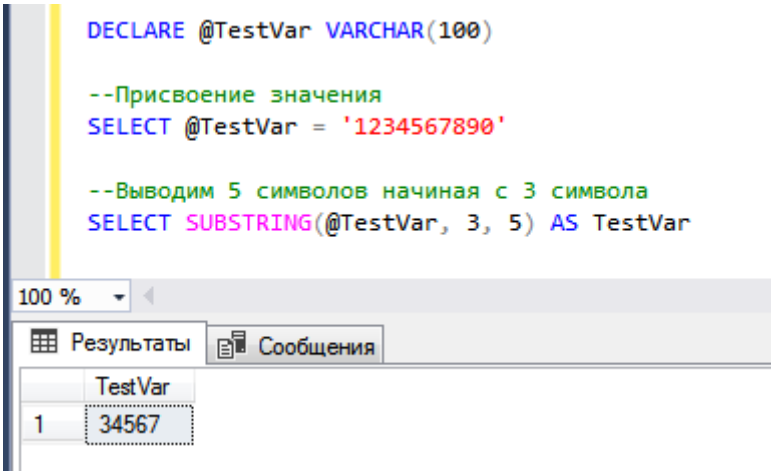


Рис. 89

В данном случае мы выводим часть строки, а именно всего 5 символов начиная с 3 символа.

Функции для работы с датой и временем

В SQL сервере есть функции, которые могут работать с датой и временем. Например, чтобы получить текущую дату, или часть даты (*день, месяц*) или прибавить к дате определённое количество дней, месяцев, часов и т.д.

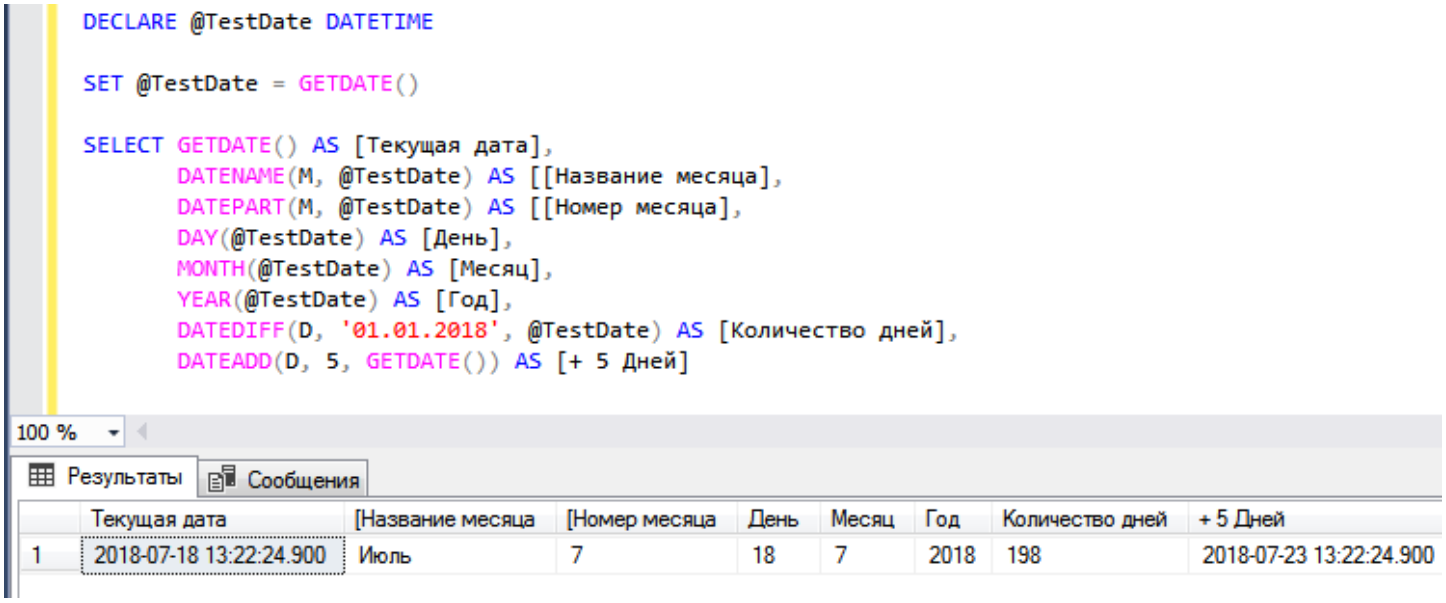
- **GETDATE** – функция возвращает текущее значение даты и времени на сервере, на котором запущен экземпляр SQL Server. Функция вызывается без параметров;
- **DATENAME** - возвращает название определенной части даты, например, название месяца. В функцию необходимо передать параметр, указывающий, название какой части необходимо вернуть, и, конечно же, параметр с самой датой;
- **DATEPART** – функция возвращает целое число, представляющее определенную часть даты, например, того же месяца, иными словами, также как DATENAME, только в виде числа, параметры такие же;
- **DAY** - возвращает целое число, представляющее день указанной в параметре даты;
- **MONTH** - функция возвращает целое число, представляющее месяц указанной в параметре даты;
- **YEAR** - возвращает целое число, представляющее год указанной в параметре даты;
- **DATEDIFF** – функция возвращает интервал времени между указанными датами. В качестве единицы измерения интервала может выступать час, день, месяц и так далее. Например, с помощью данной функции Вы можете узнать, сколько дней прошло между указанными датами. У Функции три параметра, первый - тип измерения, второй – начальная дата, третий - конечная дата;
- **DATEADD** – с помощью данной функции можно сформировать новое значение даты, например, прибавив к ней определённое количество дней или месяцев. У функции три параметра, первый - что прибавляем, второй – сколько, и третий – исходная дата.

Для демонстрации того, как работают эти функции, давайте напишем следующую SQL инструкцию.

```
DECLARE @TestDate DATETIME

SET @TestDate = GETDATE()

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



	Текущая дата	[Название месяца	[Номер месяца	День	Месяц	Год	Количество дней	+ 5 Дней
1	2018-07-18 13:22:24.900	Июль	7	18	7	2018	198	2018-07-23 13:22:24.900

Рис. 90

В данном примере мы объявляем переменную, и присваиваем ей значение равное текущей дате. Затем в инструкции SELECT мы вызываем функции для работы с датой и временем. В функции, в которые нужно передать параметры, мы в качестве параметра передаем переменную. В последнем случае (*DATEADD*) я для примера показал, что функции можно вкладывать друг в друга, т.е. необязательно в качестве параметра указывать переменную, можно и другую функцию, и столбец таблицы.

В функциях DATENAME и DATEPART значением «M» мы указали часть даты, т.е. месяц (*MONTH*). Можно использовать и «yyyy» - год, и «D» - день, и «hh» – час.

Математические функции

В T-SQL, конечно же, есть функции, которые позволяют производить различные математические расчеты, как простые, так и сложные. Вот некоторые математические функции:

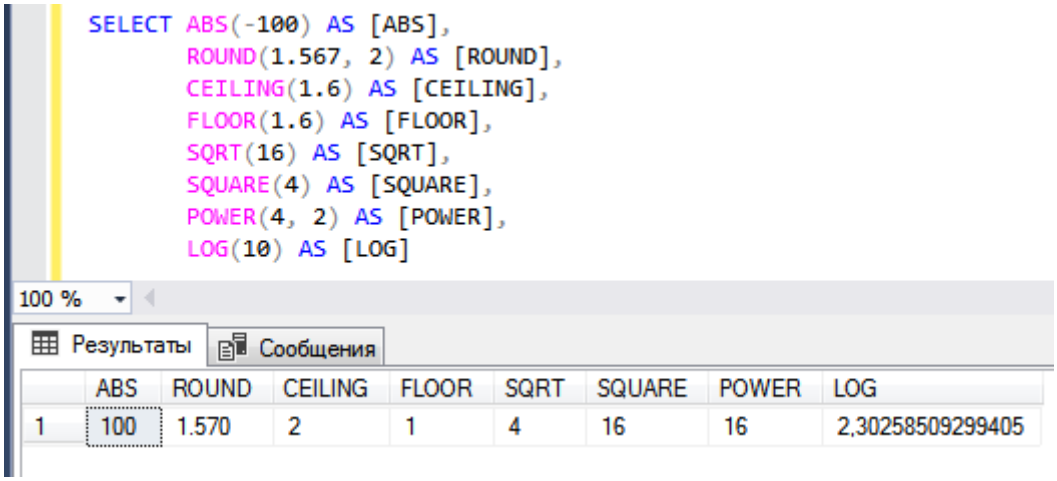
- **ABS** - функция, возвращает абсолютное (*положительное*) значение указанного числа;
- **ROUND** - функция возвращает числовое значение, округленное до указанной во входящем параметре точности;
- **CEILING** - функция возвращает наименьшее целое число, которое больше или равно числовому выражению, указанному во входящем параметре (*если по-простому, то округление до целого в большую сторону*);
- **FLOOR** - функция возвращает наибольшее целое число, меньшее или равное указанному числовому выражению (*округление до целого в меньшую сторону*);
- **SQRT** - функция возвращает квадратный корень указанного числа;

- **SQUARE** - функция возвращает квадрат указанного числа;
- **POWER** - функция возвращает возведенное в заданную в параметре степень указанное число;
- **LOG** - функция возвращает натуральный логарифм числового выражения.

Также есть и функции, которые могут возвращать тангенс (TAN), котангенс (COT), косинус (COS) и другие подобные, но они мне практически не требуются.

Пример использования математических функций.

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



	ABS	ROUND	CEILING	FLOOR	SQRT	SQUARE	POWER	LOG
1	100	1.570	2	1	4	16	16	2,30258509299405

Рис. 91

Функции метаданных

С помощью данной группы функций мы можем получать системную информацию о базе данных и объектах баз данных. Например, Вам нужно в своих SQL инструкциях использовать системные идентификаторы таблиц или представлений, при этом обращаться к системным представлениям, которые мы рассматривали в Главе 5, не очень удобно. В данных случаях лучше использовать функции метаданных.

- **DB_ID** - функция возвращает идентификатор базы данных, в качестве параметра можно указать имя базы данных;
- **DB_NAME** - функция возвращает имя базы данных, в качестве параметра можно указать идентификатор базы данных;
- **OBJECT_ID** - функция возвращает идентификатор объекта базы данных, в качестве параметра можно указать имя объекта;
- **OBJECT_NAME** - функция возвращает имя объекта базы данных, в качестве параметра можно указать идентификатор объекта.

Вот несколько вариантов использования представленных выше функций.

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

Я думаю, по названию столбцов все понятно, единственное, прокомментирую, что идентификатор (149575571), который я передаю в функцию OBJECT_NAME, в Вашей БД может и отсутствовать, в данном случае функция вернет NULL. Вы его можете заменить на то значение, которое вернет функция OBJECT_ID.

На самом деле существует гораздо больше функций метаданных, но на данном этапе Вам ими лучше голову не забивать, просто знайте, что они есть! Иногда они бывают незаменимы!

Прочие функции

В данной группе я выделил несколько функций, которыми Вы наверняка будете пользоваться.

- **ISNULL** – функция заменяет значение NULL указанным замещающим значением, имеет, соответственно, два параметра;
- **COALESCE** - функция возвращает первое выражение из списка параметров, которое не равняется NULL. Похожа на ISNULL, но в данном случае мы уже можем указывать больше чем два параметра. В некоторых случаях алгоритм их работы также отличается;
- **CAST** – функция преобразует выражение одного типа данных в другой;
- **HOST_NAME** – функция возвращает имя рабочей станции;
- **SUSER_SNAME** - функция возвращает имя входа;
- **USER_NAME** - функция возвращает имя пользователя базы данных.

Примеры.

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

Найти полный перечень и описание всех встроенных функций Вы можете в официальной документации к Microsoft SQL Server.

Для начала, я думаю, достаточно, главное теперь Вы знаете, как создаются функции и то, что существуют системные функции, и как их использовать.

Снова повторюсь, если Вам что-то непонятно, перечитайте главу и обязательно потренируйтесь в написании своих собственных SQL инструкций.

Глава 11 - Хранимые процедуры

Вот мы с Вами и дошли до объектов, с помощью которых реализуются все основные и сложные расчеты и алгоритмы в Microsoft SQL Server – это хранимые процедуры.

Мы с Вами рассмотрели функции, которые помогают нам в реализации наших алгоритмов, но сам алгоритм пишется в инструкциях, а все инструкции мы можем включить в процедуры, процедуры - это программы внутри базы данных.

Я говорил уже, что в функциях мы не можем выполнять операции модификации данных (*UPDATE, DELETE и т.д.*), в процедурах можем. В процедурах можно, наверное, все!

Если можно так выразиться, программирование на T-SQL – это реализация своих алгоритмов в виде хранимых процедур. Это, конечно, немного утрировано, но какая-то доля правды в этом есть, с помощью процедур мы реализуем все основные расчеты, проводимые в базе данных.

Поэтому я Вам советую уделить особое внимание этой главе. В данной главе мы рассмотрим основы написания хранимых процедур, после ее прочтения Вы самостоятельно сможете писать свои собственные процедуры и, соответственно, реализовывать свои алгоритмы!

Что это такое, и какие они бывают?

Хранимая процедура – это объект базы данных, в котором заложен алгоритм в виде SQL инструкций. Хранимые процедуры так же, как и функции, могут принимать входящие параметры, помимо этого они могут возвращать параметры, возвращать результирующий набор данных и выполнять операции модификации данных.

Хранимые процедуры использовать в обычной инструкции SELECT, как мы использовали функции, уже не получится, процедура - это отдельная инструкция, и ее необходимо вызывать, т.е. исполнять, а как это делается, мы рассмотрим чуть позже в этой главе.

Хранимые процедуры так же, как и функции, бывают **пользовательские и системные**. Как Вы уже, наверное, догадались, с помощью пользовательских процедур мы как раз и можем реализовывать свои собственные алгоритмы, а с помощью системных процедур мы можем получать дополнительную информацию об объектах базы данных, расширять стандартные возможности программирования на T-SQL, настраивать SQL сервер, и многое другое. Системные хранимые процедуры – это своего рода инструмент управления SQL сервером!

Пользовательские процедуры

Если в базе данных реализовывать бизнес логику, то это нужно будет делать именно с помощью пользовательских хранимых процедур, а все остальное - функции, представления и другие возможности языка T-SQL, всего лишь будут дополнением и вспомогательным инструментом реализации бизнес логики.

Допустим, Вам нужно выполнить какую-нибудь многошаговую операцию над данными, при этом Вам ее нужно автоматизировать, т.е. чтобы ее смогли выполнять обычные пользователи из клиентского приложения. Сделать это с помощью функций или представлений Вы не сможете, для этого Вам нужно написать соответствующую многошаговую SQL инструкцию, а, чтобы автоматизировать ее, Вам необходимо оформить данную инструкцию в виде хранимой процедуры. Пользователи в этом случае могут просто нажимать кнопку в клиентском приложении, в событии которого будет заложен вызов этой процедуры, таким образом, пользователь простым кликом мышки запускает сложный расчет или алгоритм, который Вы заложите в хранимой процедуре.

Надеюсь, важность и обязательность хранимых процедур в программировании на T-SQL понятна, поэтому давайте учиться их писать!

Создание хранимых процедур

Хранимые процедуры так же, как и другие объекты, создаются с помощью инструкции CREATE, а именно **CREATE PROCEDURE**.

Правила написания процедуры практически такие же, как и у функций. Сначала пишем команду создания и указываем название процедуры, затем определяем параметры процедуры, и, конечно же, определяем тело процедуры, т.е. инструкции, которые и будут реализовывать алгоритм работы процедуры.

В случае необходимости Вы можете сделать так, чтобы процедура вернула Вам табличные данные, как будто это вернул Вам обычный запрос SELECT. Это полезно, если Вы разрабатываете какой-нибудь отчет, который формирует данные по очень сложному алгоритму, требующему много различных расчетов, обычный запрос SELECT или представление Вам не помогут, а вот хранимая процедура очень даже поможет.

Предлагаю сразу перейти к практике. Давайте представим, что нам нужно добавить новый товар в определенную категорию, чтобы цена нового товара при этом была средней по данной категории. А также для демонстрации того, что процедура может возвращать табличные данные, давайте сделаем так, чтобы наша процедура возвращала нам все товары из категории, в которую мы добавили товар. Таким образом, мы добавляем новый товар, и сразу проверяем, что именно мы добавили.

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

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

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

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

GO

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

В данном примере мы создали процедуру TestProcedure, у которой два параметра, @CategoryId – категория, в которую необходимо добавить товар, @ProductName – название товара.

Затем мы написали ключевое слово AS, и начали писать тело процедуры, т.е. открыли блок инструкций словом BEGIN, после ключевого слова END, т.е. после закрытия данного блока, тело

хранимой процедуры будет завершено. Все входящие параметры хранимой процедуры в теле процедуры выглядят, как обычные переменные, и используются как переменные, только их объявлять командой DECLARE не нужно, мы их уже объявили во время определения входящих параметров.

В данной процедуре все инструкции я прокомментировал, но на текущей момент, что делают все эти инструкции, Вы уже должны знать! Даже что делают функции ROUND, AVG, LTRIM и RTRIM, если Вам непонятно, для чего я использовал здесь эти функции, настоятельно рекомендую перечитать главу, посвященную функциям. Единственное скажу, что LTRIM и RTRIM я использовал для того, чтобы привести входные данные к корректному виду, так как входящие данные могут содержать… ну Вы сами это уже должны знать.

В самом конце блока инструкций хранимой процедуры мы написали обычный запрос SELECT, именно он нам и возвращает табличные данные.

После того, как мы запустили инструкцию создания процедуры, SQL сервер ее проверил, скомпилировал и сохранил в виде отдельного объекта. При каждом вызове этой процедуры SQL сервер уже не будет повторно ее компилировать.

Командой GO я отделил пакет создания процедуры и пакет, в котором я вызываю эту процедуру. Как видите, вызывается процедура, иными словами, запускается на выполнение, командой **EXEC**. Но это сокращенная запись, полная команда **EXECUTE**. Другими словами, точно также можно было запустить процедуру командой EXECUTE.

Параметры в процедуру передаются после того, как Вы указали название этой процедуры, можно перечислять название параметров и указывать их значение, как в примере выше, но также можно и не указывать название параметров, при этом SQL сервер будут подставлять значения в параметры в том порядке, в котором они указаны. То есть, если параметр @CategoryId первый, то и значение для него необходимо указывать первым. Например, давайте в следующей инструкции запустим процедуру эквивалентным способом.

```
EXECUTE TestProcedure 1, 'Тестовый товар 2'
```

Главное при таком способе запуска процедур соблюдать последовательность, иначе можно просто передать некорректное значение параметра.

В теле процедуры, т.е. в блоке BEGIN…END, Вы можете писать практически все возможные SQL инструкции, закладывать абсолютно любой алгоритм с применением всех тех знаний, которыми Вы уже обладаете. Для практики самостоятельно создайте еще одну процедуру, которая будет обновлять цену товара по его идентификатору, при этом сделайте так, что, если новая цена товара вдвое превышает прежнюю цену, инструкция изменения цены не выполнялась, т.е. цена не изменялась.

Изменение хранимых процедур

Заложенный алгоритм в процедурах очень часто приходится дорабатывать, вносить изменения, поэтому Вы, конечно же, должны знать, как вносить эти изменения, сейчас мы это и рассмотрим.

Хранимые процедуры изменяются с помощью инструкции **ALTER**. Как и при изменении функций, для внесения изменения в процедуры мы вместо команды CREATE пишем ALTER, а все остальное определение (*код*) хранимой процедуры меняем по необходимости.

Давайте представим, что в созданную нами ранее процедуру TestProcedure необходимо внести определённые изменения, например, чтобы пользователь мог сам указывать конкретную цену товара при необходимости. Для этого нам нужно, соответственно, добавить еще один параметр, при этом он у нас будет необязательный, т.е. если мы его не укажем, то по умолчанию присвоится средняя цена. Также нам нужно будет немного доработать алгоритм процедуры.

```
--Изменяем процедуру
ALTER PROCEDURE TestProcedure
(
	@CategoryId INT,
	@ProductName VARCHAR(100),
	@Price MONEY = NULL -- Необязательный параметр
)
AS
BEGIN

	--Если цену не передали, то определяем среднюю цену
	IF @Price IS NULL
		SELECT @Price = ROUND(AVG(Price), 2)
		FROM TestTable
		WHERE CategoryId = @CategoryId

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

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

GO

--Вызываем процедуру
EXECUTE TestProcedure @CategoryId = 1,
                      @ProductName = 'Тестовый товар 3',
                      @Price = 100
```

Вы видите, что в процедуре добавился еще один параметр, он необязательный, о чем свидетельствует присвоение значения по умолчанию (= NULL). В коде процедуры мы добавили проверку, при которой если значение переменной @Price не задано (*это и есть наш параметр*), мы определяем среднюю цену и записываем ее в переменную @Price.

При вызове этой процедуры мы указали параметр @Price, в итоге у нас добавилась запись с указанной ценой. Если мы его не указали бы, то запись добавилась со средней ценой.

Совет 18

Если параметров в хранимой процедуре много, и некоторые из них являются необязательными, рекомендую при вызове таких процедур перечислять название параметров. Если параметров всего 1 или 2, то можно название параметров и не указывать, а использовать последовательность передачи значений.

Для закрепления полученных знаний, предлагаю выполнить Вам еще одно практическое задание. Измените созданную процедуру таким образом, чтобы все ее параметры были необязательными, т.е. чтобы мы могли запустить хранимую процедуру абсолютно без параметров, и она выполнилась, добавив тем самым некую шаблонную запись, например, запись в первой категории со значением наименования товара «*Не задано*» и средней ценой.

Удаление хранимых процедур

Если Вам хранимая процедура больше не нужна, Вы ее можете удалить, догадайтесь какой командой сделать это? Правильно, все той же DROP.

Например:

```
DROP PROCEDURE NameProcedure
```

Где, NameProcedure – это название процедуры.

При удалении процедуры следует помнить о том, что, если она используется в других процедурах или инструкциях, эти инструкции и процедуры будут завершаться с ошибкой, так как объект (*т.е. процедура*), на который они ссылаются, не существует, поэтому их также нужно подкорректировать.

Системные хранимые процедуры

Вы уже знаете, что есть и системные представления, и системные функции, и по всей логике, должны быть и системные хранимые процедуры, так оно и есть. С помощью системных процедур мы можем так же, как и с помощью представлений и функций, получать системную информацию, а также главной особенностью системных процедур является то, что с помощью них мы можем управлять SQL сервером и использовать его дополнительные возможности при программировании своих алгоритмов.

Абсолютно любой компонент, абсолютно любая возможность SQL сервера могут управляться с помощью системных хранимых процедур.

Системных хранимых процедур очень много, их даже больше, чем системных представлений! Посмотреть перечень системных процедур Вы можете в среде SQL Server Management Studio в обозревателе объектов «*База данных-> Программирование-> Хранимые процедуры-> Системные хранимые процедуры*».

Поэтому, конечно же, все их мы рассматривать не будем, для этого есть официальная документация.

Единственное, я покажу, как можно использовать системные хранимые процедуры.

Например, для того чтобы получить общую информацию о базе данных, включая данные о ее размере, о файлах, которые включаются в базу данных, Вы можете использовать системную процедуру **sp_helpdb**, в качестве параметра можно указать конкретную базу данных.

```
EXEC sp_helpdb 'TestDB'
```

Чтобы получить перечень таблиц, которые есть в БД, можно использовать процедуру **sp_tables**. В качестве параметра для этого передать значение TABLE, если передать VIEW, то процедура вернет перечень представлений.

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

Если Вам понадобиться переименовать объект базы данных, то Вы для этого можете использовать хранимую процедуру **sp_rename**. Например, в примере ниже мы переименовываем таблицу с именем TestTable_OldName и задаем ей новое имя TestTable_NewName.

```
EXEC sp_rename TestTable_OldName, TestTable_NewName
```

В общем, возможности системных хранимых процедур разнообразны, на текущий момент Вам достаточно знать об их существовании, изучать их все нет смысла, если Вы ими не будете пользоваться. Просто знайте, что, если Вам нужно что-то настроить в SQL сервере или получить какую-нибудь системную информацию, это скорей всего можно сделать с помощью системных процедур.

Глава 12 - Триггеры в T-SQL

На самом деле в данной главе мы продолжим разговаривать о хранимых процедурах, так как триггер - это всего лишь разновидность, специальный тип хранимой процедуры.

Сразу скажу, что без триггеров обойтись можно, т.е. умение их писать в программировании на T-SQL не является критичным, однако триггеры нам очень сильно помогают в реализации того или иного функционала, той или иной возможности, поэтому если Вы хотите стать хорошим программистом T-SQL, Вы должны уметь разрабатывать и использовать триггеры.

Что это такое? И зачем они нужны?

Триггер – в языке T-SQL это разновидность хранимой процедуры. Ключевым отличием триггера от хранимой процедуры является то, что он запускается не пользователем, а событием. Иными словами, хранимую процедуру мы запускаем командой EXECUTE, а триггер запускается, точнее срабатывает, на определенные события.

В качестве таких событий могут выступать операции INSERT, UPDATE или DELETE, другими словами, операции добавления, обновления или удаления данных.

Например, Вам нужно чтобы после (*или может быть вместо*) добавления новой записи в определенную таблицу, автоматически добавлялась новая запись в другую таблицу. Вы можете, конечно же, каждый раз при добавлении записей писать дополнительную инструкцию вставки в другую таблицу или даже разработать отдельную процедуру добавления таких данных, но, во-первых, это не удобно, во-вторых, это Вам не гарантирует того, что запись в дополнительную таблицу будет добавлена, ведь вторую инструкцию INSERT Вы или кто-то другой может просто забыть написать, а если при этом нарушаются какие-то важные бизнес требования, то у Вас возникнут серьёзные проблемы.

Самым простым решением в данной ситуации является создание триггера на событие INSERT. Например, Вы можете реализовать триггер, который будет срабатывать после вставки (*FOR или AFTER*) новой строки в таблицу, т.е. Вы посылаете инструкцию INSERT (*добавление записи*), SQL сервер определяет это событие, ждет завершения выполнения всех каскадных действий и проверки ограничений, и после этого запускает триггер, т.е. все SQL инструкции, определенные в нем. В триггерах Вы так же, как и в обычных хранимых процедурах, можете использовать практически все инструкции языка T-SQL, за исключением инструкций создания, изменения и удаления базы данных, и некоторых других.

Кроме операций INSERT, триггер может срабатывать и на операции UPDATE или DELETE. С помощью триггеров можно реализовать некий аудит данных, например, хранить все изменения, произошедшие в таблице, в другой, специально созданной для этого таблице.

Триггеры также можно использовать для обеспечения ссылочной целостности данных, для соблюдения бизнес-правил, для реализации очень сложных ограничений, которые нельзя реализовать с помощью ограничения CHECK, в общем, триггеры - это очень полезная возможность в программировании на языке T-SQL.

Триггеры в Microsoft SQL Server по моменту срабатывания делятся на:

- **FOR или AFTER** – триггер запускается после успешного выполнения инструкции, которая запускает триггер (*при разработке триггера можно использовать и ключевое слово FOR, и ключевое слово AFTER, они эквиваленты, т.е. триггер FOR аналогичен триггеру AFTER*);
- **INSTEAD OF** - в данном случае триггер запускается вместо инструкции, которая запускает триггер, тем самым переопределяя ее действие.

Временные таблицы inserted и deleted

В триггерах доступны две временные таблицы **inserted** и **deleted**. Их структура аналогична таблице, для которой создан триггер. Таблица inserted хранит новые значения строк, таблица deleted старые значения. Данные таблицы автоматически создаются SQL сервером, и мы к ним можем обращаться в триггере.

В таблице inserted добавляются строки во время операций INSERT и UPDATE, если со вставкой понятно (*т.е. там будут копии добавленных записей*), то во время обновления там будут строки с новыми значениями, на которые мы обновляем.

В таблице deleted добавляются строки во время операций DELETE и UPDATE, при удалении туда попадают удаленные строки, при обновлении - исходные значения строк до непосредственного обновления.

Иными словами, уже после выполненных операций по изменению данных, мы можем в триггере получить доступ и, соответственно, использовать значения, которые были в таблице до внесения этих изменений.

Используя эти таблицы, мы как раз и можем реализовать полный аудит данных, т.е. знать кто, когда и что именно изменил в таблице.

Создание триггеров на T-SQL

Триггеры создаются инструкцией **CREATE TRIGGER**. Для того чтобы понять, как работают, и как реализовываются триггеры, давайте рассмотрим реальный пример создания триггера, который будет делать следующее.

Допустим, Вам необходимо знать всю историю изменений в таблице, т.е. если значение в столбце было изменено или удалена запись, Вы хотите знать прежнее значение или какая именно строка была удалена, а также какой пользователь и когда внес эти изменения. Другими словами, мы напишем триггер для аудита всех изменений в таблице.

Для примера давайте используем нашу тестовую таблицу TestTable, т.е. мы хотим знать, кто и какие именно изменения вносит в эту таблицу.

Первое, что нам нужно сделать, это создать таблицу, в которой будут храниться все изменения. Для этого мы создаем таблицу, например, AutitTestTable, в которой будут столбцы для хранения как старых значений, так и новых значений всех столбцов таблицы TestTable.

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

Таблицы создавать Вы уже умеете, поэтому пояснять инструкцию выше я не буду, единственное скажу, что столбец DtChange – для хранения даты и времени изменения, UserName – для хранения имени пользователя, SQL_Command – для хранения типа операции.

Теперь давайте посмотрим на код создания триггера, Вы уже умеете читать код.

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

	IF EXISTS (SELECT * FROM inserted) AND EXISTS (SELECT * FROM deleted)
		SET @SQL_Command = 'UPDATE'

	IF NOT EXISTS (SELECT * FROM inserted) AND EXISTS (SELECT * FROM deleted)
		SET @SQL_Command = 'DELETE'

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

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

Этой инструкцией мы создали триггер с названием TRG_Audit_TestTable для таблицы TestTable. Затем мы указали, когда и на какое событие будет срабатывать данный триггер, т.е. ключевым словом AFTER мы говорим, что триггер должен срабатывать после успешного выполнения инструкции, а после перечисляем типы этих инструкций, в данном случае мы указали все INSERT, UPDATE, DELETE.

Далее уже привычное для нас слово AS, и открытие блока BEGIN…END, в котором мы, как и в обычных хранимых процедурах пишем все инструкции, которые мы хотим выполнить, когда сработает триггер.

В данном блоке мы сначала определяем, что это за инструкция, а затем на основе полученной информации осуществляем вставку в таблицу AutitTestTable.

Если происходит добавление или обновление записи, мы обращаемся к таблицам inserted и deleted, объединяя их по ProductId, так как при обновлении нам нужны и старые, и новые значения, в случае с INSERT это объединение нам не мешает, так как при этом основной источник у нас таблица inserted, таблицу deleted мы объединяем с помощью LEFT JOIN.

Если происходит удаление записи, мы обращаемся только к таблице DELETE, так как в inserted обращаться в этом случае нам не нужно, там пусто.

Как Вы, наверное, уже заметили, в триггере я использую функции GETDATE(), SUSER_SNAME(), которые возвращают соответственно дату изменения и имя пользователя, который внес эти изменения.

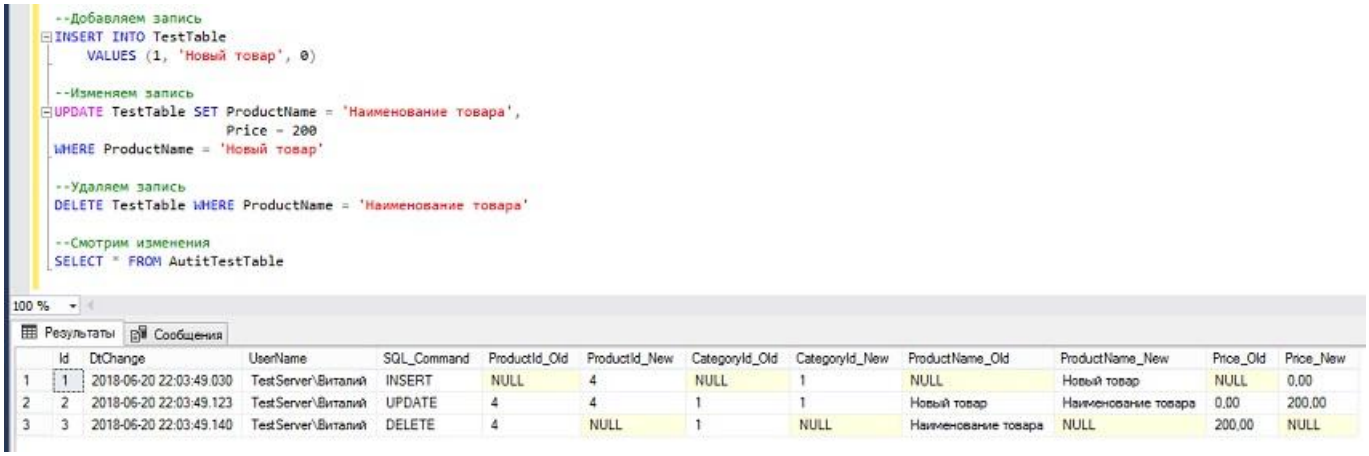
Нам осталось проверить работу только что созданного нами триггера, для этого выполните следующие запросы.

```
--Добавляем запись
INSERT INTO TestTable
     VALUES (1, 'Новый товар', 0)

--Изменяем запись
UPDATE TestTable SET ProductName = 'Наименование товара',
                     Price = 200
WHERE ProductName = 'Новый товар'

--Удаляем запись
DELETE TestTable WHERE ProductName = 'Наименование товара'

--Смотрим изменения
SELECT * FROM AutitTestTable
```



```
--Добавляем запись
INSERT INTO TestTable
    VALUES (1, 'Новый товар', 0)

--Изменяем запись
UPDATE TestTable SET ProductName = 'Наименование товара',
                     Price = 200
WHERE ProductName = 'Новый товар'

--Удаляем запись
DELETE TestTable WHERE ProductName = 'Наименование товара'

--Смотрим изменения
SELECT * FROM AutitTestTable
```

100 %

Результаты | Сообщения

	Id	DtChange	UserName	SQL_Command	ProductId_Old	ProductId_New	CategoryId_Old	CategoryId_New	ProductName_Old	ProductName_New	Price_Old	Price_New
1	1	2018-06-20 22:03:49.030	TestServer\Виталий	INSERT	NULL	4	NULL	1	NULL	Новый товар	NULL	0,00
2	2	2018-06-20 22:03:49.123	TestServer\Виталий	UPDATE	4	4	1	1	Новый товар	Наименование товара	0,00	200,00
3	3	2018-06-20 22:03:49.140	TestServer\Виталий	DELETE	4	NULL	1	NULL	Наименование товара	NULL	200,00	NULL

Рис. 92

У меня все отработало! А у Вас?

На практике изменений скорей всего будет много, да и столбцов в таблицах также будет намного больше, поэтому стоит, наверное, позаботиться об очистке старых записей таблицы AutitTestTable, так как ее размер будет очень интенсивно расти.

Совет 19

При разработке триггера, который реализует некие бизнес правила, всегда четко планируйте и тестируйте все действия триггера, в противном случае в определенных условиях, которые Вы не учтете, триггер может скрытно, т.е. незаметно для Вас, вносить некорректные изменения в базу данных, а когда Вы обнаружите эти некорректные данные или нарушение целостности данных, Вам нужно будет еще отследить причины этих изменений, ведь с первого взгляда явных причин Вы не увидите. Последствия таких ошибок в работе триггера могут быть очень серьезные.

Включение и отключение триггеров

В случае необходимости триггеры можно отключить, например, обычно для массовых операций некоторые триггеры отключают для быстродействия, а затем, после выполнения таких операций включают обратно. Инструкции включения и отключения триггеров, конечно же, можно включить в хранимую процедуру, для автоматизации этого процесса. В процессе отключения триггера сам объект триггер не удаляется из базы данных, он просто выключается, а когда триггер включается, он не создается заново, он просто активируется.

Для того чтобы отключить триггер, используется инструкция **DISABLE TRIGGER**, а для включения - **ENABLE TRIGGER**.

Например, давайте отключим триггер, созданный нами чуть ранее.

```
DISABLE TRIGGER TRG_Audit_TestTable ON TestTable;
```

Чтобы включить триггер, напишите следующую инструкцию.

```
ENABLE TRIGGER TRG_Audit_TestTable ON TestTable;
```

После инструкций DISABLE и ENABLE мы указываем название триггера и таблицу, для которой он создан с помощью ключевого слова ON. После инструкций DISABLE и ENABLE обязательно ставим символ точка с запятой, иначе будет ошибка.

Изменение триггеров на T-SQL

Вы, наверное, уже догадываетесь, что триггеры можно изменять, и какой именно инструкцией это можно сделать, все правильно - это можно сделать инструкцией **ALTER**. Как и изменение функций или процедур, мы просто вместо CREATE пишем ALTER, а все остальное определение триггера изменяем так, как нам нужно.

Например, давайте допустим, что нам теперь не нужно отслеживать операции удаления данных из таблицы, поэтому в триггер необходимо внести соответствующие изменения.

Для этого нам нужно убрать команду DELETE после ключевого слова AFTER, для того чтобы не отслеживать больше операции удаления, а также все SQL инструкции, которые связаны с удалением.

В итоге инструкция изменения триггера будет выглядеть следующим образом.

```
ALTER TRIGGER TRG_Audit_TestTable ON TestTable
      AFTER INSERT, UPDATE
AS
BEGIN
      DECLARE @SQL_Command VARCHAR(100);
      /*
      Определяем, что это за операция
      на основе наличия записей в таблицах inserted и deleted.
      На практике, конечно же, лучше делать отдельный триггер для каждой операции
      */

      IF EXISTS (SELECT * FROM inserted) AND EXISTS (SELECT * FROM deleted)
            SET @SQL_Command = 'UPDATE'
      ELSE
            SET @SQL_Command = 'INSERT'

      INSERT INTO AutitTestTable(DtChange, UserName, SQL_Command, ProductId_Old,
                                 ProductId_New,CategoryId_Old, CategoryId_New,
                                 ProductName_Old, ProductName_New, Price_Old, Price_New)
            SELECT GETDATE(), SUSER_SNAME(), @SQL_Command,
                  D.ProductId, I.ProductId,
                  D.CategoryId, I.CategoryId,
                  D.ProductName, I.ProductName,
                  D.Price, I.Price
            FROM inserted I
            LEFT JOIN deleted D ON I.ProductId = D.ProductId
END
```

Также отмечу, что в инструкции, где мы определяем операцию, мы внесли небольшие изменения, теперь мы только один раз запускаем проверку IF EXISTS, так как операций DELETE, т.е. третьего выбора, у нас просто не может быть.

Удаление триггеров на T-SQL

С инструкцией удаления как всегда все проще, пишем DROP TRIGGER и указываем его название.

```
DROP TRIGGER TRG_Audit_TestTable
```

С триггерами давайте заканчивать, надеюсь, Вам понятен принцип их работы. Если нет, то, как всегда, рекомендую потренироваться в разработке своих собственных триггеров.

Глава 13 - Курсоры в T-SQL

В данной главе мы с Вами рассмотрим очень интересную возможность языка T-SQL – это курсоры. Вы узнаете, что это такое, для чего они нужны, как они создаются, а также узнаете рекомендации по использованию курсоров, так как сразу скажу, что в некоторых ситуациях использовать курсоры нежелательно.

Что такое курсоры?

Курсор – это сохраненный результирующий набор данных, к которому можно обращаться. Простыми словами, курсор - это ссылка на результат запроса.

Курсоры в основном используются для того, чтобы выполнить для каждой строки определенного набора данных какие-то действия, расчет, применить определенный алгоритм. Курсоры можно использовать и в хранимых процедурах, и в триггерах, и в обычных SQL инструкциях.

Использование курсоров - это очень ресурсоемкий процесс для SQL сервера, инструкции, в которых используется курсор, выполняются медленнее тех, в которых операции выполняются на наборе данных, а не для каждой строки, поэтому просто так курсоры не рекомендуется использовать, применяйте их только там, где без них не обойтись.

Работа с курсорами

Я бы сказал, что курсоры - это целая конструкция языка T-SQL, так как для работы с курсорами необходимо выполнять несколько обязательных шагов, существует даже некая методика работы с курсорами:

1. Объявление переменных, которые будут использоваться и содержать данные, возвращенные курсором;
2. Объявление курсора, т.е. связывание курсора с инструкцией SELECT;
3. Открытие курсора, для того чтобы выполнить инструкцию SELECT и заполнить курсор;
4. Извлечение данных из курсора и работа с ними;
5. Закрытие курсора и освобождение всех занимаемых им ресурсов.

Давайте реализуем на практике данную методику, т.е. напишем пример использования курсора, и тем самым Вы увидите принцип работы курсора, а также принцип работы с этим курсором.

```
--1. Объявление переменных
DECLARE @ProductId INT,
        @ProductName VARCHAR(100),
        @Price MONEY

--2. Объявление курсора
DECLARE TestCursor CURSOR FOR
     SELECT ProductId, ProductName, Price
     FROM TestTable
     WHERE CategoryId = 1

--3. Открываем курсор
OPEN TestCursor

--4. Считываем данные из первой строки в курсоре
--и записываем их в переменные
FETCH NEXT FROM TestCursor INTO @ProductId, @ProductName, @Price

-- Запускаем цикл, выйдем из него, когда закончатся строки в курсоре
WHILE @@FETCH_STATUS = 0
BEGIN
     --На каждую итерацию цикла выполняем необходимые нам SQL
инструкции
     --Для примера изменяем цену по условию
     IF @Price < 100
          UPDATE TestTable SET Price = Price + 10
          WHERE ProductId = @ProductId

     --Считываем следующую строку курсора
     FETCH NEXT FROM TestCursor INTO @ProductId, @ProductName, @Price
END
--5. Закрываем курсор
CLOSE TestCursor
-- Освобождаем ресурсы
DEALLOCATE TestCursor
```

В данной инструкции я пометил начало каждого шага методики реализации курсора, а также прокомментировал весь код инструкции. Сейчас давайте я, как обычно, подробно распишу ключевые слова, все инструкции, которые относятся к курсору.

В начале мы объявили переменные, это Вам и так понятно, затем мы объявляем курсор, для этого пишем знакомое слово DECLARE, с помощью которого мы объявляем переменные, указываем название нашего курсора, в нашем примере это TestCursor. После названия курсора мы пишем ключевое слово CURSOR, говоря тем самым, что инструкцией DECLARE мы объявляем именно курсор. Затем с помощью команды FOR связываем курсор с инструкцией SELECT, после которой и пишем нужный нам запрос SELECT, к результату которого мы и будем обращаться.

Далее, для того чтобы обратиться к курсору, его необходимо открыть, для этого используется команда OPEN и указывается название нужного курсора.

Курсор открыт, значит, теперь мы можем извлечь данные из него. Это делается с помощью инструкции FETCH. Команда FETCH позволяет нам перемещаться по курсору, передвигая тем самым указатель по строкам в курсоре. Например, NEXT возвращает следующую строку за текущем указателем, в случае, если это начало курсора, как у нас, возвращается первая строка. Также существуют команды, позволяющие перемещать указатель на предыдущую строку (PRIOR), на первую

строку (FIRST), на последнюю строку (LAST), только для этого нужно указать параметр SCROLL, во время объявления курсора (CURSOR SCROLL FOR).

После команды, с помощью которой мы передвигаемся по курсору, в нашем случае NEXT, мы пишем ключевое слово FROM, и указываем название курсора. Затем пишем INTO, для того чтобы вставить все значения столбцов в курсоре, в том же порядке, в следующие переменные, и перечисляем эти переменные.

Теперь мы можем войти в цикл для того, чтобы последовательно пройтись по всем срокам в курсоре. Для определения того, что строки еще есть, мы используем функцию @@FETCH_STATUS, если она возвращает 0, значит, последняя инструкция извлечения данных закончилась успешно, когда она вернет не 0, выходим из цикла, так как строки закончились.

В блоке BEGIN…END цикла выполняем все инструкции, которые мы хотим выполнить для данной строки данных (*можно использовать и вызов процедур, и операции INSERT, DELETE и другие*).

Конечно, действия, которые мы выполняем в теле нашего цикла, можно было бы выполнить и без использования курсора, в данном случае я просто показал принцип работы с курсором. Перед окончанием блока BEGIN…END мы снова считываем следующую строку в курсоре командой FETCH, и пытаемся войти в следующую итерацию цикла (*если строка есть, то входим, если нет, то, соответственно, не входим*).

После того как курсор нам больше не нужен, мы его должны закрыть командой CLOSE, а если мы его больше вообще не будем использовать в данной инструкции, освобождаем все ресурсы командой DEALLOCATE. После этой команды, в случае если Вы захотите снова обратиться к этому курсору, Вам нужно будет его заново объявлять и открывать.

Совет 20

Если есть возможность выполнить задачу или реализовать алгоритм без использования курсоров, то делайте это без курсоров. Курсоры применяйте только в тех случаях, когда другого решения у Вас нет. Так как если в алгоритме работы приложения будет задействовано слишком много курсоров - это заметно замедлит работу приложения. Иными словами, используйте курсоры только в самых крайних случаях.

Глава 14 - Транзакции в T-SQL

Данная глава посвящена транзакциям. Если Вы будете реализовывать бизнес логику в Microsoft SQL Server, которая предполагает многошаговые операции, при этом каждый шаг в данной инструкции логически связан с другими шагами, то Вам обязательно нужно будет использовать транзакции. А что это такое, я сейчас и расскажу.

Введение

Транзакция – это команда или блок команд (*инструкций*), которые успешно завершаются как единое целое, при этом в базе данных все внесенные изменения фиксируются на постоянной основе, или отменяются, т.е. все изменения, внесенные любой командой, входящей в транзакцию, будут отменены. Другими словами, если одна команда или инструкция внутри транзакции завершилась с ошибкой, то все, что было отработано перед ней, также отменяется, даже если предыдущие команды завершились успешно. Транзакции - это отличный механизм обеспечения целостности данных.

У транзакции есть 4 очень важных свойства:

- **Атомарность** — все команды в транзакции либо полностью выполняются, и соответственно, фиксируются все изменения данных, либо ничего не выполняется и ничего не фиксируется;
- **Согласованность** — данные, в случае успешного выполнения транзакции, должны соблюдать все установленные правила в части различных ограничений, первичных и внешних ключей, определенных в базе данных;
- **Изоляция** — механизм предоставления доступа к данным. Транзакция изолирует данные, с которыми она работает, для того чтобы другие транзакции получали только согласованные данные;
- **Надежность** — все внесенные изменения фиксируются в журнале транзакций и данные считаются надежными, если транзакция была успешно завершена. В случае сбоя SQL Server сверяет данные, записанные в базе данных, с журналом транзакций, если есть успешно завершенные транзакции, которые не закончили процесс записи всех изменений в базу данных, они будут выполнены повторно. Все действия, выполненные не подтвержденными транзакциями, отменяются.

По сути, каждая отдельная инструкция языка T-SQL является транзакцией, это называется «*Автоматическое принятие транзакций*», но также есть и явные транзакции, это когда мы сами явно начинаем транзакцию и также явно заканчиваем ее, т.е. делаем все это с помощью специальных команд.

Для того чтобы понять, как работают транзакции, и для чего они нужны, давайте рассмотрим пример. Допустим, у Вас есть процедура, которая осуществляет перевод средств с одного счета на другой, соответственно, как минимум у Вас будет две операции в этой процедуре, снятие средств, и зачисление средств (*например, две инструкции UPDATE*). Но в каждой из этих операций может возникнуть ошибка и инструкция не выполнится. А теперь представьте, что первая инструкция снимает деньги, она выполнилась успешно, а вторая инструкция зачисляет деньги, и в ней возникла ошибка, без транзакции снятые деньги просто потеряются, так как они никуда не будут зачислены. Чтобы этого не допускать, все SQL инструкции, которые логически что-то объединяет, пишут внутри транзакции, и тогда, если наступит подобная ситуация, все изменения будут отменены, т.е. деньги вернутся обратно на счет.

Транзакции можно сочетать с обработкой и перехватом ошибок (*TRY...CATCH*), иными словами, Вы отслеживаете ошибки в Вашем блоке инструкций, и если они появляются, то в блоке CATCH Вы откатываете транзакцию, т.е. отменяете все изменения, которые были успешно выполнены до возникновения ошибки в транзакции.

Команды управления транзакциями

В T-SQL для управления транзакциями существуют следующие основные команды:

- **BEGIN TRANSACTION** (*можно использовать сокращённую запись BEGIN TRAN*) – команда служит для определения начала транзакции. В качестве параметра этой команде можно передать и название транзакции, полезно, если у Вас есть вложенные транзакции;
- **COMMIT TRANSACTION** (*можно использовать сокращённую запись COMMIT TRAN*) – с помощью данной команды мы сообщаем SQL серверу об успешном завершении транзакции, и о том, что все изменения, которые были выполнены, необходимо сохранить на постоянной основе;
- **ROLLBACK TRANSACTION** (*можно использовать сокращённую запись ROLLBACK TRAN*) – служит для отмены всех изменений, которые были внесены в процессе выполнения транзакции, например, в случае ошибки, мы откатываем все назад;
- **SAVE TRANSACTION** (*можно использовать сокращённую запись SAVE TRAN*) – данная команда устанавливает промежуточную точку сохранения внутри транзакции, к которой можно откатиться, в случае возникновения необходимости.

Предлагаю сразу рассмотреть пример использования транзакций.

```
BEGIN TRY
--Начало транзакции
BEGIN TRANSACTION

     --Инструкция 1
     UPDATE TestTable SET CategoryId = 2
     WHERE ProductId = 1

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

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

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

               --Прекращаем выполнение инструкции
               RETURN;
          END CATCH

--Если все хорошо. Сохраняем все изменения
COMMIT TRANSACTION
```

В данном примере у нас всего две инструкции, которые изменяют данные, но допустим, что они взаимосвязаны, т.е. они обе обязательно должны выполниться вместе или не выполниться также вместе.

Поэтому мы решили эти инструкции объединить в одну транзакцию, и отслеживать ошибки конструкцией TRY...CATCH.

Сначала мы открываем блок для обработки ошибок, затем открываем транзакцию BEGIN TRANSACTION, далее пишем все необходимые инструкции, которые мы хотим объединить в транзакцию.

После этого закрываем блок TRY, открываем блок CATCH, в котором в случае возникновения ошибки мы откатываем все изменения командой ROLLBACK TRANSACTION. Также мы принудительно завершаем нашу инструкцию командой RETURN.

Если ошибок нет, то в блок CATCH мы, соответственно, не попадаем, и у нас выполнится команда COMMIT TRANSACTION, которая сохранит все изменения.

В примере я намерено допускаю ошибку во второй инструкции, как Вы помните, CategoryId в таблице TestTable не может содержать значения NULL, поэтому возникнет ошибка и управление передастся в блок CATCH, и происходит откат изменений. Первая инструкция отработала нормально, но ее изменения не были сохранены, так как вторая инструкция выполнена с ошибкой.

Во время выполнения транзакции все данные, над которыми производятся изменения, блокируются, до завершения транзакции, так как, когда один процесс изменяет данные, другой процесс не может одновременно изменять их. В SQL сервер существует механизм, который блокирует (изолирует) данные во время выполнения транзакции. У данного механизма есть несколько уровней изоляции, каждый из которых определяет степень блокировки данных.

Уровни изоляции

READ UNCOMMITTED — самый низкий уровень, при котором SQL сервер разрешает так называемое «*грязное чтение*». Грязным чтением называют считывание неподтвержденных данных, иными словами, если транзакция, которая изменяет данные, не завершена, другая транзакция может получить уже измененные данные, хотя они еще не зафиксированы и могут отмениться.

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

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

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

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

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

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

Для того чтобы включить тот или иной уровень изоляции, для всей сессии необходимо выполнить команду

```
SET TRANSACTION ISOLATION LEVEL <название уровня изоляции>;
```

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

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

Транзакции, конечно же, это очень обширная тема языка T-SQL, но так как в данной книге мы говорим об основах, поэтому углубляться мы пока не будем, для старта этих знаний Вам будет достаточно!

Глава 15 - Работа с XML в T-SQL

Microsoft SQL Server поддерживает работу с XML данными, ведь XML – это очень удобный и распространённый формат данных, его легко использовать для обмена данными между приложениями. В T-SQL даже существует специальный тип данных XML, который предназначен для хранения XML данных.

Для того чтобы работать с XML, в языке T-SQL реализовано множество специальных конструкций, и в этой главе я Вас познакомлю с этими конструкциями.

Методы типа данных XML

Сначала давайте создадим тестовую таблицу, в которой будет столбец с типом XML.

```
CREATE TABLE TestTableXML(
     Id INT IDENTITY(1,1) NOT NULL,
     NameColumn VARCHAR(100) NOT NULL,
     XMLData XML NULL
 CONSTRAINT PK_TestTableXML PRIMARY KEY (Id)
)
```

Где, XMLData - это столбец, в котором будут храниться данные в формате XML. Как Вы понимаете, столбцы с типом XML добавляются точно так же, как и столбцы с обычным типом данных.

Для того чтобы добавить данные в таблицу, нужно написать обычный запрос на вставку данных (INSERT), только в столбец для XML данных вставлять, соответственно, данные в формате XML.

```
INSERT INTO TestTableXML (NameColumn, XMLData)
       VALUES('Текст','<Catalog>
                         <Name>Иван</Name>
                         <LastName>Иванов</LastName>
                       </Catalog>')

GO

SELECT * FROM TestTableXML
```

Рис. 93

Тип данных XML в Microsoft SQL Server имеет несколько методов (*функций*) работы над XML данными. Они реализуются по технологии XQuery, т.е. имеют такой же синтаксис, тому, кто знает XQuery, будет намного проще работать с XML данными на T-SQL.

Сейчас давайте кратко рассмотрим эти методы.

Метод query

query() – метод делает выборку в самом XML документе, который хранится в столбце с типом XML. Метод принимает один параметр, запрос к xml документу, т.е. что именно Вы хотите получить из xml.

Например, в нашей тестовой таблице TestTableXML есть корневой тег Catalog, а нам нужно получить только тег Name, для этого мы можем выполнить следующий запрос:

```
SELECT XMLData.query('/Catalog/Name') AS [Тег Name]
FROM TestTableXML
```

Результат

```
<Name>Иван</Name>
```

Для того чтобы использовать методы типа XML, мы после названия столбца с типом XML ставим точку и пишем название метода, в нашем случае query, в скобочках указываем то, что мы хотим получить.

Иногда необходимо получить данные из XML данных не в формате XML, а как обычные значения, иными словами, извлечь значения узла.

Метод value

value() – метод возвращает значение узла. Данный метод удобно использовать, когда Вам нужно в своих SQL инструкциях использовать определённое значение, извлеченное из XML документа. Метод имеет два параметра, первый - это откуда брать значение, а второй - какой тип Вы при этом хотите получить на выходе.

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

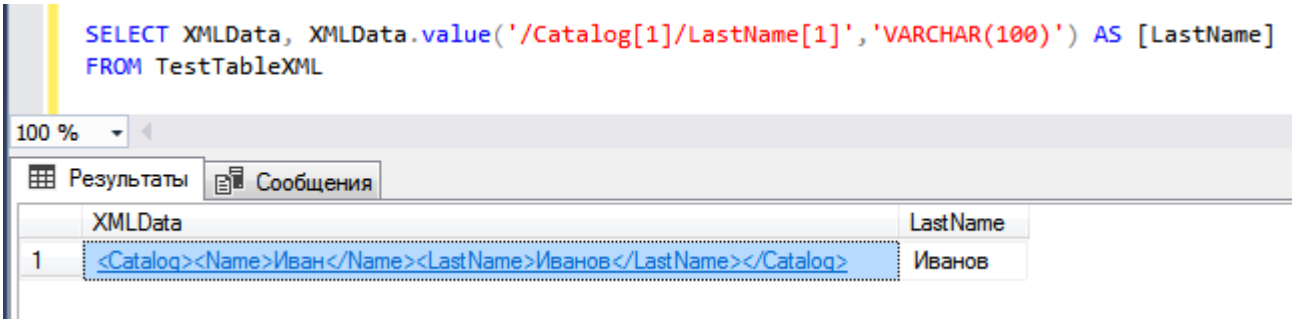


Рис. 94

В данном запросе мы получили значение первого узла LastName в первом элементе Catalog, тип значения мы указали VARCHAR(100).

Метод exist

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

Например, чтобы вывести все строки в XML данных, в которых есть элемент LastName, можно написать запрос.

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

Метод modify

modify() – этот метод изменяет XML данные. Его можно использовать только в инструкции UPDATE, в качестве параметра принимает новое значение, это инструкции по изменению XML документа.

Например, **удаление элемента**. Сейчас мы удалим элемент LastName.

```
UPDATE TestTableXML SET XMLData.modify('delete /Catalog/LastName')
GO
SELECT * FROM TestTableXML
```

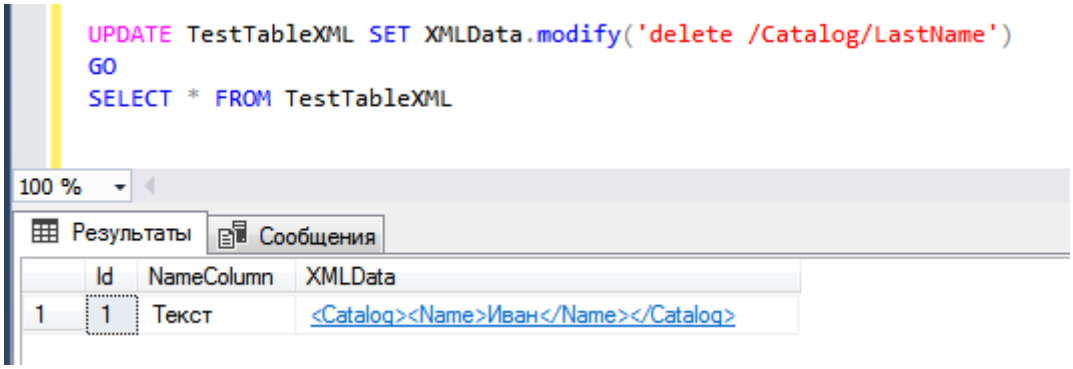


Рис. 95

Добавление элемента. В данном случае мы добавим элемент LastName.

```
UPDATE TestTableXML
SET XMLData.modify('insert <LastName>Иванов</LastName> as last into (/Catalog)[1] ')
GO
SELECT * FROM TestTableXML
```

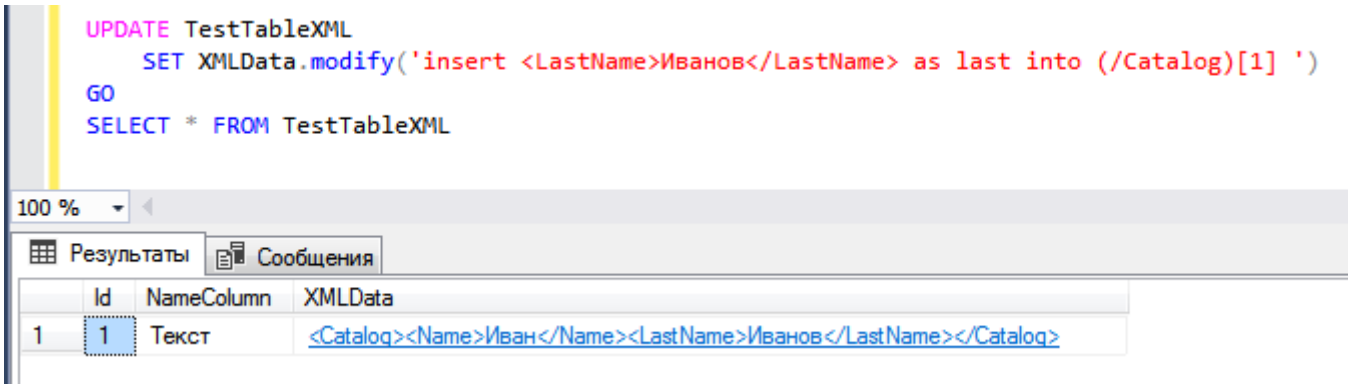


Рис. 96

Изменение значения в узле. В этом примере мы изменим значение узла Name на «Сергей».

```
UPDATE TestTableXML
SET XMLData.modify('replace value of(/Catalog/Name[1]/text())[1] with "Сергей" ')
GO
SELECT * FROM TestTableXML
```

```
UPDATE TestTableXML
    SET XMLData.modify('replace value of(/Catalog/Name[1]/text())[1] with "Сергей" ')
GO
SELECT * FROM TestTableXML
```

100 %

Результаты | Сообщения

	Id	NameColumn	XMLData
1	1	Текст	<Catalog><Name>Сергей</Name><LastName>Иванов</LastName></Catalog>

Рис. 97

Метод nodes

nodes() – метод используется для получения реляционных данных из конкретного XML документа.

```
DECLARE @XML_Doc XML;
SET @XML_Doc='<Root>
                <row id="1" Name="Иван"></row>
                <row id="2" Name="Сергей"></row>
              </Root>';
SELECT TMP.col.value('@id','INT') AS Id,
       TMP.col.value('@Name','VARCHAR(10)') AS Name
FROM   @XML_Doc.nodes('/Root/row') TMP(Col);
GO
```

```
DECLARE @XML_Doc XML;
SET @XML_Doc='<Root>
                <row id="1" Name="Иван"></row>
                <row id="2" Name="Сергей"></row>
              </Root>';
SELECT TMP.col.value('@id','INT') AS Id,
       TMP.col.value('@Name','VARCHAR(10)') AS Name
FROM @XML_Doc.nodes('/Root/row') TMP(Col);

GO
```

100 %

Результаты | Сообщения

	Id	Name
1	1	Иван
2	2	Сергей

Рис. 98

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

Конструкция FOR XML

Язык T-SQL позволяет сформировать XML данные на основе SQL запроса. Например, Вам потребовалось из данных, которые лежат в таблицах, сформировать XML документ и передать его в приложение. Это можно сделать с помощью конструкции **FOR XML.**

Для того чтобы сформировать XML данные, необходимо просто после инструкции SELECT определить конструкцию FOR XML.

У данной конструкции есть несколько режимов работы и специальных параметров (*директив*). Существуют следующие режимы:

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

Сейчас давайте рассмотрим несколько полезных параметров конструкции FOR XML:

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

Давайте рассмотрим несколько примеров. Все запросы возвращают XML документ, чтобы посмотреть его структуру, просто нажмите на результирующие данные.

```
--Пример 1
SELECT ProductId, ProductName, Price
FROM TestTable
ORDER BY ProductId
FOR XML RAW, TYPE
```

Результат

```
<row ProductId="1" ProductName="Клавиатура" Price="100.0000" />
<row ProductId="2" ProductName="Мышь" Price="50.0000" />
<row ProductId="3" ProductName="Телефон" Price="300.0000" />
```

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

```
--Пример 2
SELECT ProductId, ProductName, Price
FROM TestTable
ORDER BY ProductId
FOR XML RAW ('Product'), TYPE, ELEMENTS, ROOT ('Products')
```

Результат

```
<Products>
  <Product>
    <ProductId>1</ProductId>
    <ProductName>Клавиатура</ProductName>
    <Price>100.0000</Price>
  </Product>
  <Product>
    <ProductId>2</ProductId>
    <ProductName>Мышь</ProductName>
    <Price>50.0000</Price>
  </Product>
  <Product>
    <ProductId>3</ProductId>
    <ProductName>Телефон</ProductName>
    <Price>300.0000</Price>
  </Product>
</Products>
```

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

```
--Пример 3
SELECT TestTable.ProductId, TestTable.ProductName,
       TestTable2.CategoryName, TestTable.Price
FROM TestTable
LEFT JOIN TestTable2 ON TestTable.CategoryId = TestTable2.CategoryId
ORDER BY TestTable.ProductId
FOR XML AUTO, TYPE
```

Результат

```
<TestTable ProductId="1" ProductName="Клавиатура" Price="100.0000">
  <TestTable2 CategoryName="Комплектующие компьютера" />
</TestTable>
<TestTable ProductId="2" ProductName="Мышь" Price="50.0000">
  <TestTable2 CategoryName="Комплектующие компьютера" />
</TestTable>
<TestTable ProductId="3" ProductName="Телефон" Price="300.0000">
  <TestTable2 CategoryName="Мобильные устройства" />
</TestTable>
```

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

```
--Пример 4
SELECT TestTable.ProductId, TestTable.ProductName,
       TestTable2.CategoryName, TestTable.Price
FROM TestTable
LEFT JOIN TestTable2 ON TestTable.CategoryId = TestTable2.CategoryId
ORDER BY TestTable.ProductId
FOR XML AUTO, TYPE, ELEMENTS
```

Результат

```
<TestTable>
  <ProductId>1</ProductId>
  <ProductName>Клавиатура</ProductName>
  <Price>100.0000</Price>
  <TestTable2>
    <CategoryName>Комплектующие компьютера</CategoryName>
  </TestTable2>
</TestTable>
<TestTable>
  <ProductId>2</ProductId>
  <ProductName>Мышь</ProductName>
  <Price>50.0000</Price>
  <TestTable2>
    <CategoryName>Комплектующие компьютера</CategoryName>
  </TestTable2>
</TestTable>
<TestTable>
  <ProductId>3</ProductId>
  <ProductName>Телефон</ProductName>
  <Price>300.0000</Price>
  <TestTable2>
    <CategoryName>Мобильные устройства</CategoryName>
  </TestTable2>
</TestTable>
```

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

```
--Пример 5
SELECT ProductId AS "@Id", ProductName, Price
FROM TestTable
ORDER BY ProductId
FOR XML PATH ('Product'), TYPE, ROOT ('Products')
```

Результат

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

В этом примере мы уже используем расширенный режим **PATH**, при котором мы можем сами указывать, что у нас будет элементами, а что атрибутами.

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

Создавать XML данные на языке T-SQL на основе данных, которые лежат в наших таблицах, мы научились, теперь давайте научимся получать из XML данных привычные для нас табличные данные.

Конструкция OPENXML

Давайте представим, что Вашему приложению поступают данные в формате XML, в этих данных содержится информация, которую необходимо извлечь и сохранить в определенной структуре, для того чтобы впоследствии использовать или обрабатывать эту информацию обычным для нас способом на языке SQL. Для этих целей в языке T-SQL есть специальная конструкция, своего рода табличная функция, которая может на входе получить XML данные, а на выходе дать нам результирующий набор строк, как будто это обычный запрос на выборку. Функция называется **OPENXML**, она используется в секции FROM инструкции SELECT.

Это не просто табличная функция, это целая конструкция, так как, чтобы ее использовать, XML документ предварительно необходимо подготовить. Это делается с помощью системной хранимой процедуры **sp_xml_preparedocument**, которая проводит синтаксический анализ XML данных и возвращает дескриптор XML документа, который мы передаем в функцию OPENXML для извлечения данных.

После того как функция OPENXML отработает, дескриптор XML документа необходимо удалить системной процедурой **sp_xml_removedocument**, для того чтобы освободить память, конечно, можно этого и не делать, тогда в этом случае дескриптор будет существовать до конца сеанса (*чтобы избежать проблем с недостатком памяти, рекомендовано использовать процедуру для удаления дескриптора*).

В следующем примере мы сформируем XML документ, а затем извлечем из него данные в табличном виде. Чтобы сформировать XML документ, давайте используем конструкцию FOR XML (*так сказать для закрепления знаний*).

```
--Объявляем переменные
DECLARE @XML_Doc XML;
DECLARE @XML_Doc_Handle INT;

--Формируем XML документ
SET @XML_Doc = (
                    SELECT ProductId AS "@Id", ProductName, Price
                    FROM TestTable
                    ORDER BY ProductId
                    FOR XML PATH ('Product'), TYPE, ROOT ('Products')
                    );
--Подготавливаем XML документ
EXEC sp_xml_preparedocument @XML_Doc_Handle OUTPUT, @XML_Doc;

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

```
--Объявляем переменные
DECLARE @XML_Doc XML;
DECLARE @XML_Doc_Handle INT;

--Формируем XML документ
SET @XML_Doc = (
                SELECT ProductId AS "@Id", ProductName, Price
                FROM TestTable
                ORDER BY ProductId
                FOR XML PATH ('Product'), TYPE, ROOT ('Products')
                );
--Подготавливаем XML документ
EXEC sp_xml_preparedocument @XML_Doc_Handle OUTPUT, @XML_Doc;

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

100 %

Результаты | Сообщения

	ProductId	ProductName	Price
1	1	Клавиатура	100,00
2	2	Мышь	50,00
3	3	Телефон	300,00

Рис. 99

В этом примере сначала мы объявили переменные, затем сформировали XML документ, потом системной процедурой подготовили этот документ, для того чтобы из него можно было извлечь данные. У этой процедуры первый параметр выходной, т.е. она нам возвращает данные в виде значения параметра, чтобы его получить, мы указали переменную, в которую данное значение необходимо записать. Вторым параметром мы передали непосредственно сам XML документ.

Далее мы уже пишем запрос SELECT, и обращаемся к функции OPENXML. Ей мы передаем три параметра:

- Первый – дескриптор XML документа;
- Второй – шаблон на языке XPath, используемый для идентификации узлов, которые будут обрабатываться как строки;
- Третий – тип сопоставления между XML данными и реляционным, например, 0 – значение по умолчанию, используется атрибутивная модель сопоставления, 1 - использовать атрибутивную модель сопоставления, 2 - использовать сопоставление с использованием элементов.

После функции мы пишем ключевое слово WITH, и в скобочках перечисляем выходные столбцы и их тип данных. Так как ProductName и Price являются элементами, для них мы указываем только тип данных, а для того чтобы извлечь атрибут Id, мы дополнительно указали инструкцию с использованием символа @.

После того как мы извлекли данные, мы удалили дескриптор процедурой sp_xml_removedocument, у нее один параметр, т.е. сам дескриптор.

Как всегда, рекомендую Вам потренироваться писать свои запросы с использованием XML данных, особенно если Вам что-то непонятно. Перечитайте главу, проанализируйте примеры,

примените свои данные, только в этом случае Вы сможете использовать полученные знания на практике. Тяжело в изучении, легко на работе!

В языке T-SQL есть еще много интересных и полезных возможностей, в следующей главе мы рассмотрим некоторые из них.

Глава 16 - Дополнительные полезные возможности языка T-SQL

Полностью изучить все возможности и, тем более, нюансы языка T-SQL в одной книге, наверное, невозможно, я это понимал, когда читал книги по T-SQL других авторов, когда начал работать над этой книгой, я стал это понимать еще лучше. Я уже говорил, что цель данной книги - это не охват абсолютно всего, а донесение до Вас той информации и в том виде, которая будет Вам, как начинающим, полезна для старта программирования на T-SQL.

Основные моменты языка T-SQL на текущий момент мы уже рассмотрели, однако у меня присутствует огромное желание рассказать Вам еще о некоторых полезных возможностях и конструкциях языка T-SQL, которые обязательно Вам потребуются и пригодятся, но о которых мы с Вами еще не разговаривали. Без некоторых я уже не могу представить себе программирование на T-SQL, поэтому продолжаем изучение языка T-SQL!

Конструкция WITH – обобщенное табличное выражение

Начну с конструкции, которая, наверное, является у меня самой любимой в части упрощения написания кода, т.е. которую я постоянно использую, для того чтобы код сделать более читабельным, и чтобы в него впоследствии было легко вносить изменения. Я говорю о конструкции WITH, а именно об обобщенном табличном выражении (CTE).

Обобщенное табличное выражение (*Common Table Expression, сокращенно CTE*) – это результирующий набор данных, у которого есть имя, и к которому можно обращаться в запросе. Другими словами, у Вас есть SQL запрос (*SELECT*), который возвращает определённые данные на основе некого алгоритма, т.е. с объединением, с условием, с группировкой и так далее. И Вам нужно к этому результату обратиться, иными словами, также использовать его как часть более крупного SQL запроса. Вы, конечно, можете создать временную таблицу и заполнить ее этим набором данных, а потом уже использовать эту временную таблицу. Или, наверно, первое, что может прийти на ум - это использовать подзапрос, помните, как мы делали в секции FROM или JOIN, если нужно выполнить объединение. Но это очень сильно усложняет понимание кода всего запроса, с первого взгляда Вы просто не поймете, что делает этот запрос, если в нем будет куча подзапросов, объединений и других конструкций языка T-SQL.

Лично я для себя нашел отличное решение в подобных ситуациях, точнее оно просто есть в T-SQL – это конструкция WITH.

Конечно, изначально основной целью обобщённых табличных выражений была реализация возможности написания рекурсивных запросов, т.е. чтобы в запросе можно было обращаться к собственному результирующему набору данных, например, для формирования иерархических данных. Да, кстати, если Вам это нужно, в T-SQL это делается именно с помощью конструкции WITH.

Обобщенное табличное выражение - это своего рода представление, только оно не хранится как объект на сервере. Например, зачем создавать представление для одного конкретного запроса.

Если рекурсивные запросы лично я пишу редко, то запросы, в которых использую конструкцию WITH для повышения читабельности кода, я пишу достаточно часто, так как было уже отмечено, это очень сильно облегчает понимание логики всего SQL запроса.

Давайте рассмотрим пример CTE.

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

В данном случае мы написали обобщенное табличное выражение TestCTE. Для этого мы написали ключевое слово WITH, затем в скобочках перечислили названия столбцов, которые будут возвращаться при обращении к TestCTE, далее мы написали ключевое слово AS и в скобочках определили запрос, к результату которого мы хотим обращаться.

После этого мы можем обращаться к TestCTE как к представлению! Т.е. мы можем производить объединение с ним, анализировать его и так далее.

После определения обобщенного табличного выражения, т.е. сразу за TestCTE, должна идти инструкция SELECT, INSERT, UPDATE, MERGE или DELETE.

Перечисление столбцов после названия CTE (*в нашем случае после TestCTE*) можно и опустить, но в этом случае все столбцы в запросе должны возвращаться с определенным уникальным именем, в нашем примере у всех столбцов есть имена и они уникальные, поэтому мы смело можем написать и так.

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

Честно сказать, я практически никогда не перечисляю столбцы, так как в запросе всегда задаю псевдонимы для каждого столбца.

Конечно, данный пример CTE простой, но, если у Вас запросы сложные и большие, и Вам к их результату нужно еще обращаться, Вы просто не поймёте, что и для чего используется в данном запросе.

В обобщенном табличном выражении можно использовать несколько именованных запросов, тем самым разделять запрос на логические части, например:

```
WITH TestCTE1 AS --Первый запрос
     (
          --Представьте, что здесь запрос со своей сложной логикой
          SELECT ProductId, CategoryId, ProductName, Price
          FROM TestTable
     ), TestCTE2 AS -- Второй запрос
     (
          --Здесь также сложный запрос
          SELECT CategoryId, CategoryName
          FROM TestTable2
     )

--Работаем с результирующими наборами данных двух запросов
SELECT T1.ProductName, T2.CategoryName, T1.Price
FROM TestCTE1 T1
LEFT JOIN TestCTE2 T2 ON T1.CategoryId = T2.CategoryId
WHERE T1.CategoryId = 1
```

В нашем случае, конечно же, все можно было сделать гораздо проще, но, как я уже сказал, представьте, что у Вас запросы сложные и, если Вы бы использовали, например, подзапросы, было бы ничего не понятно, да что там говорить, вот пример точно таких же действий только без использования WITH.

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

Это, как я уж сказал, простые запросы в пару строк. Как мне кажется, с использованием WITH код даже в этом случае, намного читабельней и понятней.

Совет 21

Если в одном SQL запросе SELECT Вы прибегаете к использованию нескольких подзапросов в секциях FROM или JOIN, выносите данные подзапросы в обобщенное табличное выражение (*конструкция WITH*) – впоследствии это Вам значительно упростит понимание логики запроса.

Конструкция SELECT INTO

Данная конструкция очень полезна, если Вам нужно создавать таблицы, включая временные таблицы, на основе результирующего набора данных запроса SELECT.

Иными словами, у Вас есть запрос, который возвращает данные с определённой структурой, и Вам необходимо на основе этих данных и этой структуры создать новую таблицу. Если бы не было этой конструкции, Вам пришлось бы создавать таблицу вручную, т.е. перечислять все столбцы, их тип данных и так далее.

Например, Вам нужны итоговые объединённые данные нескольких таблиц в одной таблице.

```
SELECT T1.ProductName, T2.CategoryName, T1.Price
INTO TestTableDop
FROM TestTable T1
LEFT JOIN TestTable2 T2 ON T1.CategoryId = T2.CategoryId
WHERE T1.CategoryId = 1

SELECT * FROM TestTableDop
```

Для того чтобы создать таблицу на основе результирующего набора, необходимо непосредственно в запросе SELECT, после перечисления списка столбцов, т.е. перед секцией FROM, написать ключевое слово **INTO** и название новой таблицы.

В примере выше создалась таблица TestTableDop, в которой 3 столбца: ProductName, CategoryName и Price.

Оконные функции – предложение OVER

В языке T-SQL существует очень полезный и мощный инструмент для формирования различных аналитических отчетов – это инструкция ***OVER***.

OVER – это инструкция T-SQL, которая определяет окно для применения функций. «***Окно***» в Microsoft SQL Server – это контекст, в котором работает функция с определённым набором строк, относящихся к текущей строке.

Оконная функция – это функция, которая, соответственно, работает с окном, т.е. набором строк, и возвращает значение на основе неких вычислений.

Как я уже отметил, оконные функции используют в аналитических отчетах, например, для вычисления каких-то статистических значений (*суммы, скользящие средние, промежуточные итоги и так далее*) для каждой строки результирующего набора данных.

Честно скажу, оконные функции - это очень удобный и полезный функционал Microsoft SQL Server. Впервые поддержка оконных функций появилась в версии Microsoft SQL Server 2005, в которой была реализована базовая функциональность. В Microsoft SQL Server 2012 функционал оконных функций был расширен, и теперь он с лёгкостью решает много задач, которые до этого решались написанием дополнительного, в некоторых случаях, сложного, непонятного кода (*вложенные запросы и т.д.*), также этот код, в свою очередь, еще и снижал производительность.

Некоторые оконные функции очень специфичны, и на начальном этапе Вы с ними скорей всего не столкнетесь, поэтому я просто расскажу, что они делают, и приведу примеры их работы, углубляться мы не будем.

Сначала давайте посмотрим на общий упрощенный синтаксис предложения OVER.

Упрощенный синтаксис предложения OVER

```
Оконная функция (столбец для вычислений) OVER (
  [PARTITION BY столбец для группировки]
  [ORDER BY столбец для сортировки]
  [ROWS или RANGE выражение для ограничения строк в пределах группы]
 )
```

В выражении для ограничения строк в группе можно использовать следующие ключевые слова:

- **ROWS** – ограничивает строки;
- **RANGE** - логически ограничивает строки за счет указания диапазона значений в отношении к значению текущей строки;
- **UNBOUNDED PRECEDING** - указывает, что окно начинается с первой строки группы. Данная инструкция используется только как начальная точка окна;
- **UNBOUNDED FOLLOWING** – с помощью данной инструкции можно указать, что окно заканчивается на последней строке группы, соответственно, она может быть указана только как конечная точка окна;

- **CURRENT ROW** – инструкция указывает, что окно начинается или заканчивается на текущей строке, она может быть задана как начальная или как конечная точка;
- **BETWEEN** «*граница окна*» **AND** «*граница окна*» - указывает нижнюю и верхнюю границу окна, при этом верхняя граница не может быть меньше нижней границы;
- «*Значение*» **PRECEDING** – определяет число строк перед текущей строкой. Эта инструкция не допускается в предложении RANGE;
- «*Значение*» **FOLLOWING** - определяет число строк после текущей строки. Если FOLLOWING используется как начальная точка окна, то конечная точка должна быть также указана с помощью FOLLOWING. Эта инструкция не допускается в предложении RANGE.

Примечание! Чтобы указать выражение для дополнительного ограничения строк (*ROWS или RANGE*), в окне должна быть указана инструкция ORDER BY, пример мы рассмотрим чуть позже.

В T-SQL оконные функции можно подразделить на следующие группы:

- Агрегатные функции;
- Ранжирующие функции;
- Функции смещения;
- Аналитические функции.

В одной инструкции SELECT с одним предложением FROM можно использовать несколько оконных функций. Если инструкция PARTITION BY не указана, функция будет обрабатывать все строки результирующего набора. Некоторые функции не поддерживают инструкцию ORDER BY, ROWS или RANGE.

А сейчас давайте рассмотрим оконные функции каждой группы и посмотрим, как они работают.

Агрегатные оконные функции

Агрегатные оконные функции – это обычные агрегатные функции SUM, AVG, MAX, MIN, COUNT, которые используются с предложением OVER.

Обычно агрегатные функции используются в сочетании с инструкцией GROUP BY, которая группирует строки, это Вы уже знаете. Но их также можно использовать с предложением OVER, и в этом случае они будут вычислять значения в определённом окне *(наборе данных*) для каждой текущей строки, GROUP BY использовать при этом не нужно. Это очень удобно, если Вам необходимо получить какую-нибудь величину для каждой строки по отношению, например, к общей сумме.

В следующем примере продемонстрирована работа агрегатных оконных функций, результат представлен на рисунке 100.

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

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

100 %

Результаты | Сообщения

	ProductId	ProductName	CategoryId	Price	SUM	AVG	COUNT	MIN	MAX
1	1	Клавиатура	1	100,00	150,00	75,00	2	50,00	100,00
2	2	Мышь	1	50,00	150,00	75,00	2	50,00	100,00
3	3	Телефон	2	300,00	300,00	300,00	1	300,00	300,00

Рис. 100

После агрегатной функции мы написали ключевое слово OVER и в скобочках указали инструкцию PARTITION BY, для того чтобы сгруппировать данные (*сформировать наборы данных для каждой строки*).

На рисунке 100 Вы видите, у нас вывелись все строки, включая столбцы с агрегированными данными, сгруппированными по категории.

Ранжирующие оконные функции

Иногда при написании запросов требуется пронумеровать строки результирующего набора данных, т.е. чтобы в результате запроса был дополнительный столбец со значением, которое характеризует порядковый номер строки с учетом сортировки и группировки строк, иными словами, чтобы строки были определенным образом ранжированы. В основном это требуется для нумерации строк в группе или составления определённого рейтинга путем проставления ранга.

В T-SQL это можно реализовать с помощью специальных оконных ранжирующих функций: *ROW_NUMBER, RANK, DENSE_RANK и NTILE*.

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

Следующий пример демонстрирует, как можно пронумеровать все товары в каждой из категорий.

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

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

100 %

Результаты | Сообщения

	ROW_NUMBER	ProductId	CategoryId	ProductName	Price
1	1	1	1	Клавиатура	100,00
2	2	2	1	Мышь	50,00
3	1	3	2	Телефон	300,00

Рис. 101

Синтаксис такой же, как и у агрегатных оконных функций. После названия функции ROW_NUMBER () мы пишем ключевое слово OVER, затем в скобочках определяем группировку и сортировку. Соответственно, для группировки мы пишем PARTITION BY и указываем столбец, по которому группировать, для сортировки пишем ORDER BY и столбец, по которому сортировать. Иными словами, строки будут сгруппированы и отсортированы, и только после этого пронумерованы.

Инструкция PARTITION BY необязательная, ее можно и не писать, только в этом случае результирующий набор будет пронумерован полностью, т.е. не будет деления на группы, нумерация будет общая.

Синтаксис у всех ранжирующих функций одинаковый, т.е. инструкции PARTITION BY и ORDER BY есть у всех функций.

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

Пример.

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

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

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

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

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

Оконные функции смещения

Функции смещения – это функции, которые позволяют перемещаться и, соответственно, обращаться к разным строкам в наборе данных (окне) относительно текущей строки или просто обращаться к значениям в начале или в конце окна. Эти функции появились в Microsoft SQL Server 2012.

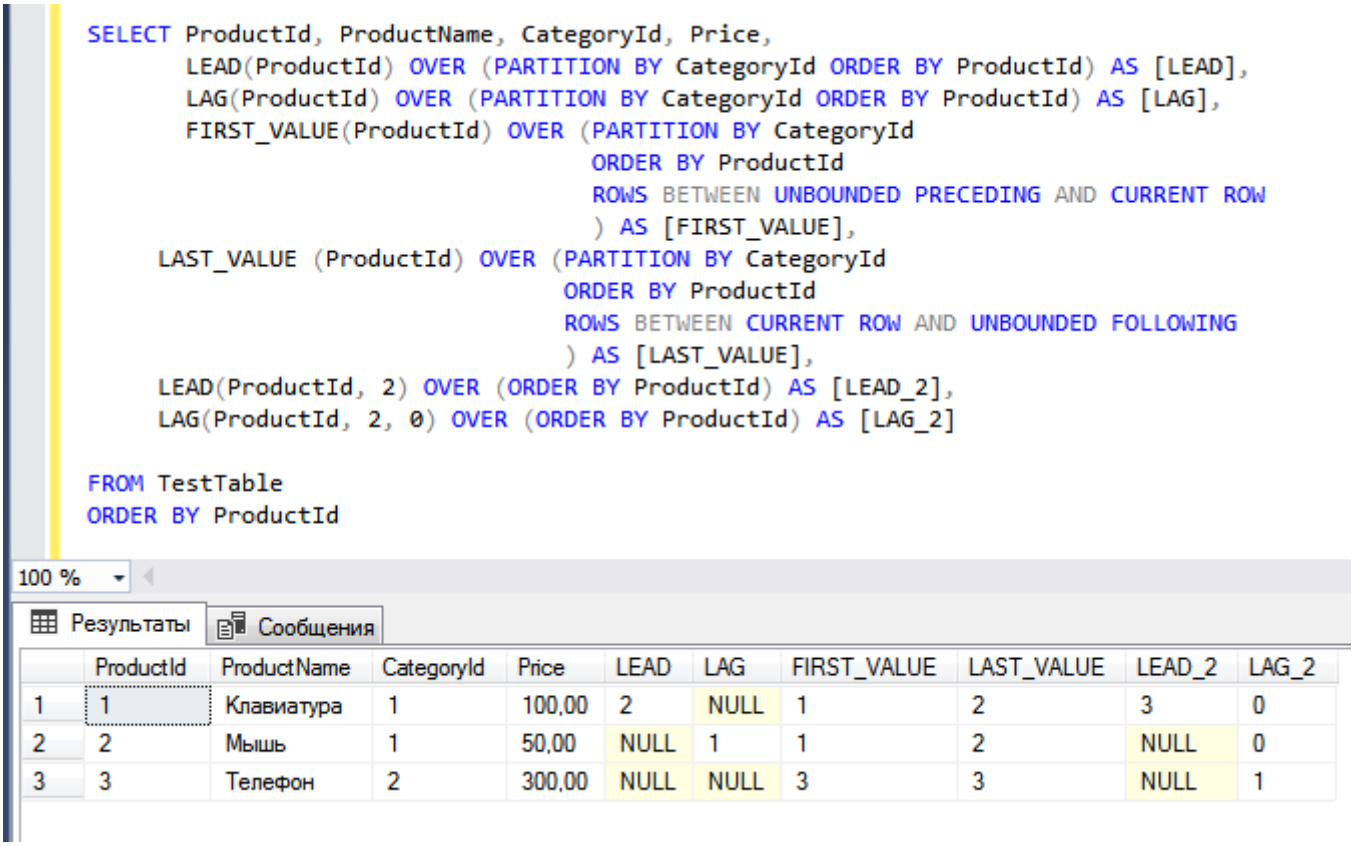
К функциям смещения в T-SQL относятся:

- **LEAD** – функция обращается к данным из следующей строки набора данных. Ее можно использовать, например, для того чтобы сравнить текущее значение строки со следующим. Имеет три параметра: столбец, значение которого необходимо вернуть (*обязательный параметр*), количество строк для смещения (*по умолчанию 1*), значение, которое необходимо вернуть, если после смещения возвращается значение NULL;
- **LAG** – функция обращается к данным из предыдущей строки набора данных. В данном случае функцию можно использовать для того, чтобы сравнить текущее значение строки с предыдущим. Имеет три параметра: столбец, значение, которого необходимо вернуть (*обязательный параметр*), количество строк для смещения (*по умолчанию 1*), значение, которое необходимо вернуть, если после смещения возвращается значение NULL;
- **FIRST_VALUE** - функция возвращает первое значение из набора данных, в качестве параметра принимает столбец, значение которого необходимо вернуть;
- **LAST_VALUE** - функция возвращает последнее значение из набора данных, в качестве параметра принимает столбец, значение которого необходимо вернуть.

Пример использования оконных функций смещения в T-SQL.

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

FROM TestTable
ORDER BY ProductId
```



	ProductId	ProductName	CategoryId	Price	LEAD	LAG	FIRST_VALUE	LAST_VALUE	LEAD_2	LAG_2
1	1	Клавиатура	1	100,00	2	NULL	1	2	3	0
2	2	Мышь	1	50,00	NULL	1	1	2	NULL	0
3	3	Телефон	2	300,00	NULL	NULL	3	3	NULL	1

Рис. 102

Данный запрос возвращает следующее [LEAD] и предыдущее [LAG] значение идентификатора товара в категории относительно текущей строки. Чтобы функции работали для каждой категории, мы указали инструкцию PARTITION BY CategoryId, инструкцией ORDER BY ProductId мы сортируем строки в окне по идентификатору.

Также запрос возвращает первое [FIRST_VALUE] и последнее [LAST_VALUE] значение идентификатора товара в категории, при этом в качестве примера я показал, как используется синтаксис дополнительного ограничения строк (*ROWS*).

Чтобы показать, как смещаться на несколько строк в запросе, я дополнительно вызвал функции LEAD и LAG, только уже с передачей необязательных параметров. Таким образом, в столбце [LEAD_2] происходит смещение на 2 строки вперед, а в столбце [LAG_2] на 2 строки назад, при этом в данном столбце будет выводиться 0, если после смещения нужной строки не окажется, для этого был указан третий необязательный параметр со значением 0.

Аналитические оконные функции

Данные функции также называют **функции распределения**, так как они возвращают информацию о распределении данных. Эти функции очень специфичны, и в основном используются для статистического анализа.

К аналитическим оконным функциям в T-SQL относятся:

- **CUME_DIST** - вычисляет и возвращает интегральное распределение значений в наборе данных. Иными словами, она определяет относительное положение значения в наборе;
- **PERCENT_RANK** - вычисляет и возвращает относительный ранг строки в наборе данных;
- **PERCENTILE_CONT** - вычисляет процентиль на основе постоянного распределения значения столбца. В качестве параметра принимает процентиль, который необходимо вычислить;
- **PERCENTILE_DISC** - вычисляет определенный процентиль для отсортированных значений в наборе данных. В качестве параметра принимает процентиль, который необходимо вычислить.

У функций PERCENTILE_CONT и PERCENTILE_DISC синтаксис немного отличается, столбец, по которому сортировать данные, указывается с помощью ключевого слова WITHIN GROUP.

Пример использования аналитических оконных функций в T-SQL.

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

Операторы PIVOT и UNPIVOT

На T-SQL можно формировать отчеты с использованием так называемых «*Сводных таблиц*». Данный функционал называют, например, в Excel «*Транспонирование*», в SQL такие запросы называют перекрестные запросы или запросы с использованием кросс табличных выражений. Данные запросы используются для того, чтобы развернуть результирующий набор данных, иными словами, значения в столбце, которые расположены в таблице, соответственно, по вертикали, мы можем выстроить по горизонтали, применяя при этом агрегацию и группировку данных, таким образом, происходит формирование некой сводной таблицы. Пользоваться отчетами, где данные представлены в таком виде, очень удобно, такие отчеты очень любит начальство, поэтому скорей всего отчеты со сводными таблицами Вы будете формировать.

В T-SQL для таких запросов используются специальный оператор PIVOT.

PIVOT - это оператор языка T-SQL, который поворачивает результирующий набор данных, преобразуя уникальные значения одного столбца в несколько столбцов.

Для того чтобы произвести обратное действие, т.е. преобразовать столбцы итогового набора данных в значения одного столбца, используется оператор **UNPIVOT**. Честно говоря, вот этот оператор мне требуется очень редко, в отличие от PIVOT, который в аналитике просто незаменим.

Недостатком данных операторов является то, что у них очень специфический синтаксис, и они требуют некой ручной работы, т.е., например, выходные столбцы у PIVOT должны быть перечислены вручную. Однако T-SQL - это все-таки язык программирования, и на нем можно реализовать хранимую процедуру, с помощью которой можно формировать динамические запросы PIVOT (*пример реализации можете найти на моем сайте info-comp.ru*).

Сейчас давайте рассмотрим примеры с использованием данных операторов. Для примера давайте сгруппируем данные в нашей тестовой таблице по категориям, и узнаем среднюю цену товаров в каждой категории, при этом название категорий мы расположим в столбцах. Также для сравнения

сначала я приведу пример с обычной группировкой, результат будет точно таким же, но названия категорий будут в одном столбце (Рис. 103).

```
--Обычная группировка
SELECT T2.CategoryName, AVG(T1.Price) AS AvgPrice
      FROM TestTable T1
      LEFT JOIN TestTable2 T2 ON T1.CategoryId = T2.CategoryId
GROUP BY T2.CategoryName

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

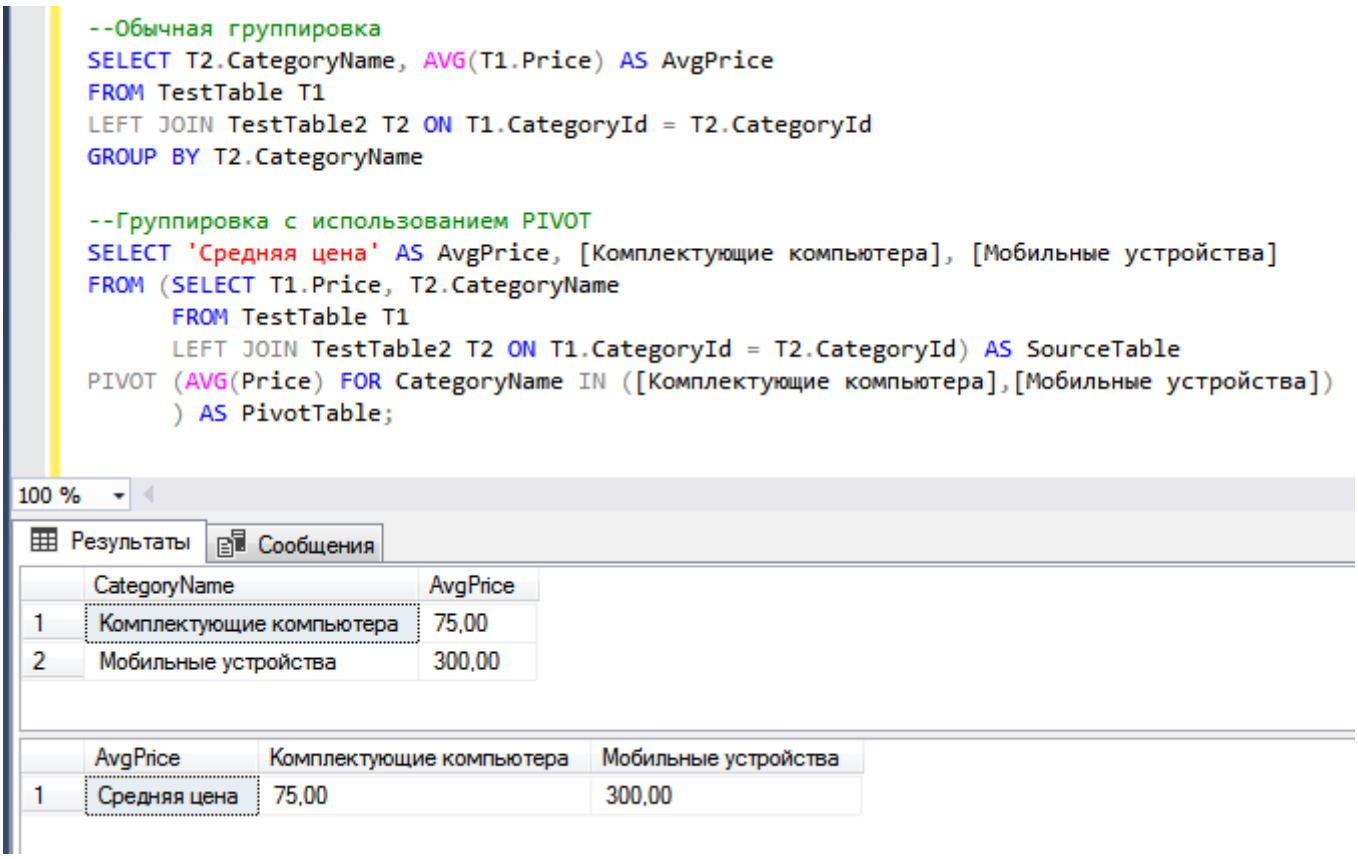


Рис. 103

Где,

- ✓ [Комплектующие компьютера], [Мобильные устройства] – это значения в столбце CategoryName, которые мы заранее должны знать;
- ✓ SourceTable – псевдоним выражения, в котором мы указываем исходный источник данных, например, вложенный запрос;
- ✓ PIVOT – вызов оператора PIVOT;
- ✓ AVG – агрегатная функция, в которую мы передаем столбец для анализа, в нашем случае Price;
- ✓ FOR – с помощью данного ключевого слова мы указываем столбец, содержащий значения, которые будут выступать именами столбцов, в нашем случае CategoryName;
- ✓ IN – оператор, с помощью которого мы перечисляем значения в столбце CategoryName;
- ✓ PivotTable - псевдоним сводной таблицы, его необходимо указывать обязательно.

Для того чтобы посмотреть на работу оператора UNPIVOT, нам нужны данные, которые мы можем использовать, подходящих данных, которые бы наглядно показали работу оператора, у нас нет.

Поэтому давайте создадим временную таблицу с подходящими данными, при этом для закрепления пройденного материала создадим мы эту таблицу с помощью конструкции SELECT INTO.

```
--Создаём временную таблицу с помощью SELECT INTO
SELECT 'Город' AS NamePar,
       'Москва' AS Column1,
       'Калуга' AS Column2,
       'Тамбов' AS Column3
INTO #TestUnpivot

--Смотрим, что получилось
SELECT * FROM #TestUnpivot

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

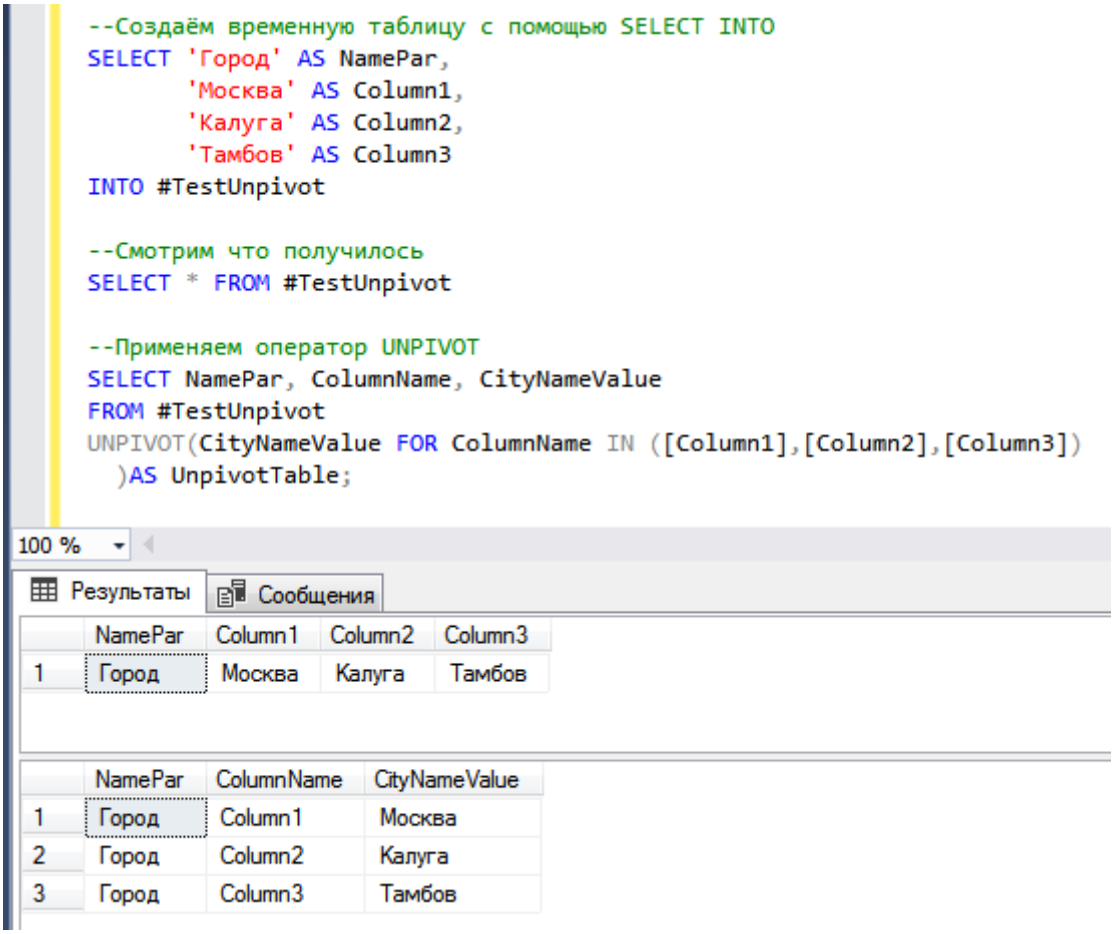


Рис. 104

Где,

- ✓ #TestUnpivot – таблица источник, в нашем случае временная таблица;
- ✓ CityNameValue – псевдоним столбца, который будет содержать значения наших столбцов;
- ✓ FOR – ключевое слово, с помощью которого мы указываем псевдоним для столбца, который будет содержать имена наших столбцов;
- ✓ ColumnName - псевдоним столбца, который будет содержать имена наших столбцов;
- ✓ IN – ключевое слово для указания имен столбцов.

Следует отметить то, что оператор UNPIVOT не восстанавливает данные, сгруппированные оператором PIVOT, он просто разворачивает итоговый набор данных.

Рекомендую Вам отнестись к оператору PIVOT с особой серьезностью, так как он действительно активно используется в аналитике, и значительно упрощает формирование отчетов. Поэтому перечитайте данный раздел, и потренируйтесь писать запросы с использованием данного оператора.

Ну а после этого переходим к другим, также очень полезным возможностям языка T-SQL, которые применяются в аналитике и формировании отчетов.

Аналитические операторы ROLLUP, CUBE и GROUPING SETS

Достаточно часто в отчетах, которые Вы будете разрабатывать на T-SQL, требуется выводить промежуточные итоги, так как это наглядно отображает имеющиеся сгруппированные данные в отчёте, такие отчеты обычно любят сотрудники бухгалтерии и, конечно же, руководство.

В языке T-SQL есть специальные операторы, которые помогают нам реализовывать отчеты с промежуточными итогами.

К таким операторам относятся: **ROLLUP, CUBE** и **GROUPING SETS**. Все эти операторы являются расширением секции GROUP BY, иными словами, когда мы пишем запросы с агрегацией и группировкой данных, мы можем использовать эти операторы для вывода промежуточных итогов.

ROLLUP

ROLLUP – оператор, который формирует промежуточные итоги для каждого указанного элемента и общий итог.

Например, нам нужно из нашей тестовой таблицы TestTable получить просуммированные данные в каждой категории. Это можно сделать с помощью группировки GROUP BY и статистической функции SUM, при этом нам нужно видеть и общий итог, для этого мы используем расширенные возможности GROUP BY, а именно оператор ROLLUP. Для сравнения я приведу два примера, первый без ROLLUP, второй с ROLLUP.

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

GO
--С использованием ROLLUP
SELECT CategoryId, SUM(Price) AS Summa
FROM TestTable
GROUP BY
ROLLUP (CategoryId)
```

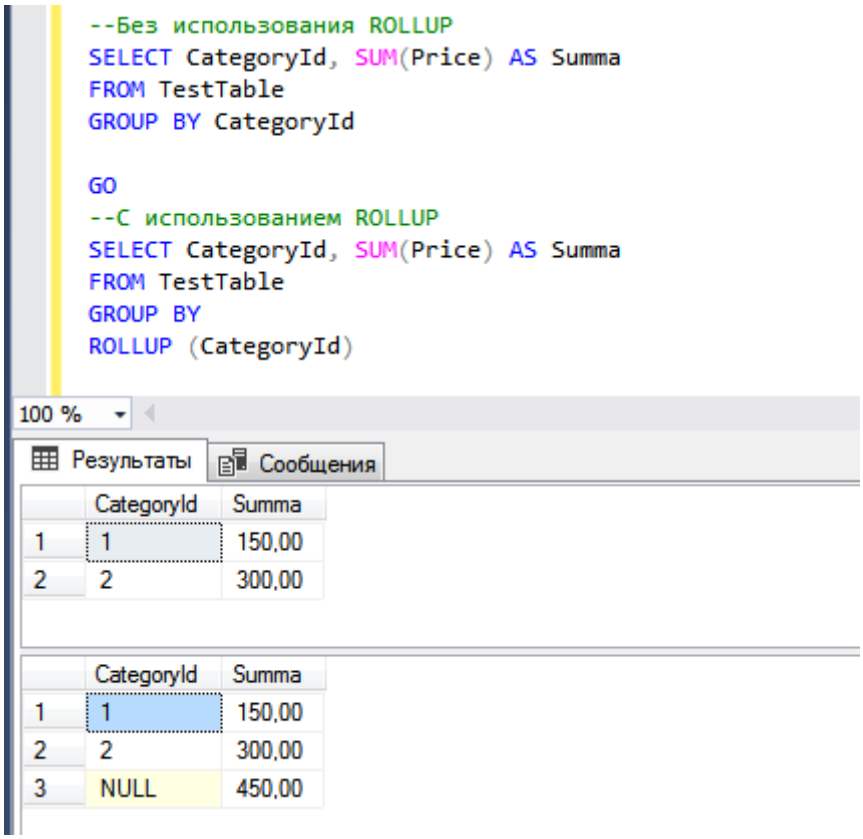


Рис. 105

Чтобы использовать оператор ROLLUP, мы просто после GROUP BY пишем данный оператор, и указываем столбец, по которому проводить группировку.

CUBE

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

В следующем примере мы добавим группировку еще по одному столбцу, в первом случае, промежуточные итоги у нас выведутся также для каждой категории (*CategoryId*), только добавится группировка по ProductName, во втором случае промежуточные итоги добавятся и для CategoryId, и для ProductName.

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

GO
--С использованием CUBE
SELECT ProductName, CategoryId, SUM(Price) AS Summa
FROM TestTable
GROUP BY
CUBE (CategoryId, ProductName)
```

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

GO
--С использованием CUBE
SELECT ProductName, CategoryId, SUM(Price) AS Summa
FROM TestTable
GROUP BY
CUBE (CategoryId, ProductName)
```

100 %

Результаты | Сообщения

	ProductName	CategoryId	Summa
1	Клавиатура	1	100,00
2	Мышь	1	50,00
3	NULL	1	150,00
4	Телефон	2	300,00
5	NULL	2	300,00
6	NULL	NULL	450,00

	ProductName	CategoryId	Summa
1	Клавиатура	1	100,00
2	Клавиатура	NULL	100,00
3	Мышь	1	50,00
4	Мышь	NULL	50,00
5	Телефон	2	300,00
6	Телефон	NULL	300,00
7	NULL	NULL	450,00
8	NULL	1	150,00
9	NULL	2	300,00

Рис. 106

GROUPING SETS

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

```
--С использованием GROUPING SETS
SELECT ProductName, CategoryId, SUM(Price) AS Summa
FROM TestTable
GROUP BY
GROUPING SETS (CategoryId, ProductName)

GO
--С использованием UNION ALL
SELECT ProductName, NULL AS CategoryId, SUM(Price) AS Summa
FROM TestTable
GROUP BY ProductName

UNION ALL

SELECT NULL AS ProductName, CategoryId, SUM(Price) AS Summa
FROM TestTable
GROUP BY CategoryId
```

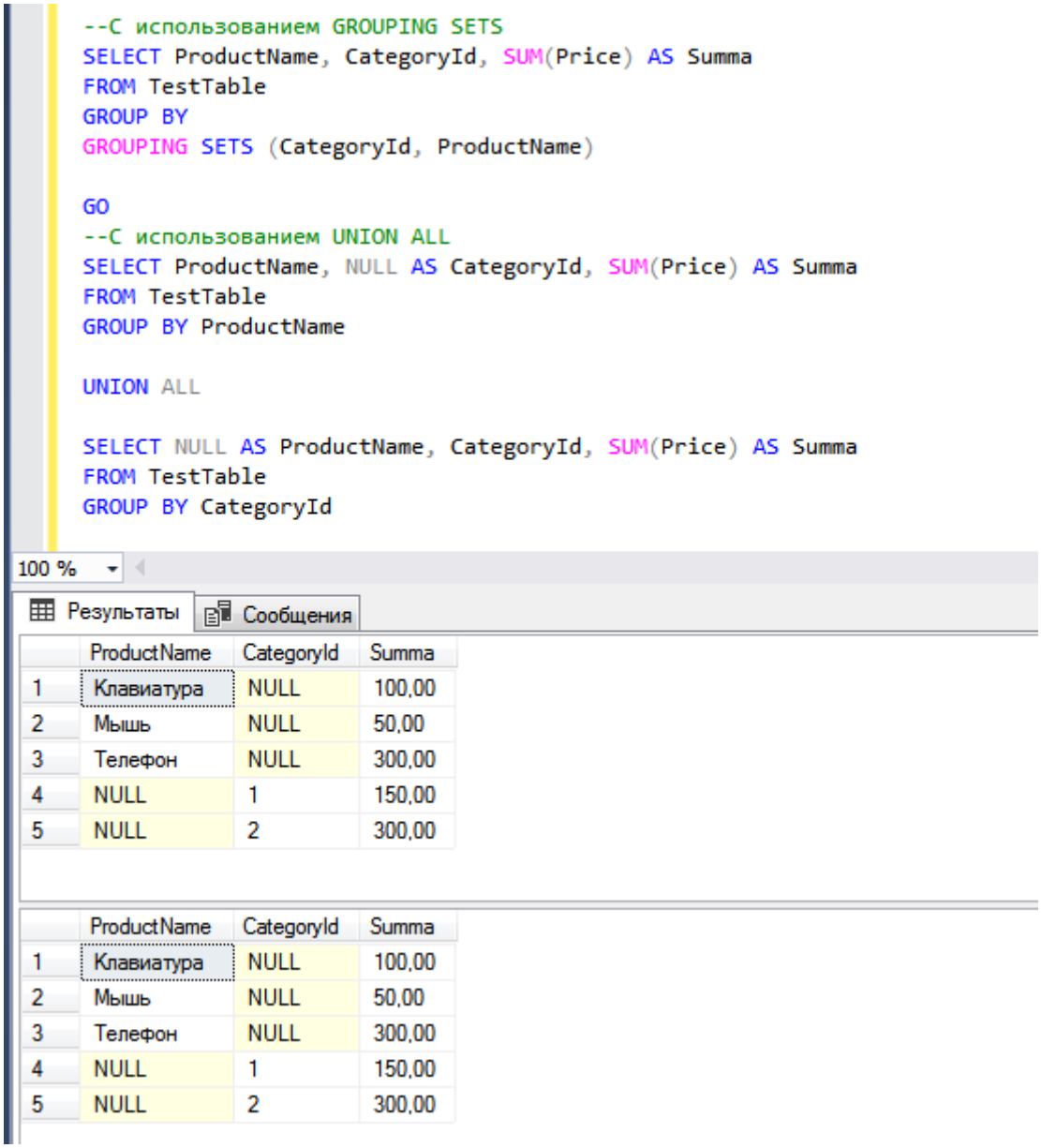


Рис. 107

Промежуточные итоги мы рассмотрели, идем далее.

Оператор APPLY

Табличные функции мы с Вами рассматривали, но иногда возникает необходимость запустить табличную функцию для каждой строки источника данных, и на выходе получить все результирующие наборы, объединённые в одном. Язык T-SQL позволяет это делать. Для этого существует специальный оператор **APPLY**.

Существует два типа оператора APPLY:

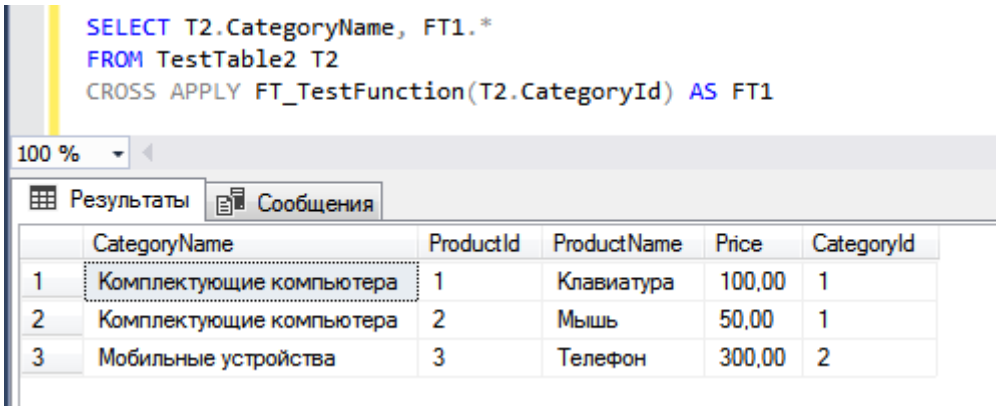
- **CROSS APPLY** - возвращает только строки из внешней таблицы, которые создает табличная функция;
- **OUTER APPLY** - возвращает и строки, которые формирует табличная функция, и строки со значениями NULL в столбцах, созданные табличной функцией. Например, табличная функция может не возвращать никаких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

Чтобы посмотреть, как работает оператор APPLY, нам нужна табличная функция, и она у нас есть. Помните, когда мы рассматривали табличные функции, мы создали функцию FT_TestFunction, которая возвращает список товаров в определенной категории, в качестве параметра она принимает идентификатор конкретной категории, данные из которой нам нужны.

А теперь давайте представим, что нам нужно получить именно с помощью этой функции все товары в нескольких категориях, например, все те категории, которые у нас есть в таблице TestTable2 (*таблица с категориями*).

Вызывать данную функцию с конкретным идентификатором несколько раз, конечно, не очень удобно, поэтому мы и будем использовать оператор APPLY.

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



	CategoryName	ProductId	ProductName	Price	CategoryId
1	Комплектующие компьютера	1	Клавиатура	100,00	1
2	Комплектующие компьютера	2	Мышь	50,00	1
3	Мобильные устройства	3	Телефон	300,00	2

Рис. 108

В данном случае функция FT_TestFunction была вызвана для каждой строки таблицы TestTable2. В функцию мы передавали идентификатор категории, в результате она вернула все наборы данных, которые были сформированы для каждой строки внешней таблицы, т.е. для каждой категории. В качестве примера я здесь показал, что можно использовать звездочку (*) и для псевдонима источника, а не только для всего списка выборки, иными словами, инструкция FT1.* говорит SQL серверу о том, что нам нужны все данные только из источника с псевдонимом FT1.

Как видите, оператор APPLY - это своего рода разновидность объединения данных.

А теперь, чтобы посмотреть на работу оператора OUTER APPLY, давайте добавим в таблицу TestTable2 еще одну запись, тем самым у нас возникнет ситуация, когда категория есть, а товаров в ней нет.

```
--Добавление новой строки в таблицу TestTable2
INSERT INTO TestTable2
          VALUES ('Новая категория');
```

После добавления записи сначала выполните тот же самый запрос с использованием типа CROSS, а затем с использованием OUTER, например.

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

Результат запроса с CROSS APPLY не изменился, а вот с OUTER APPLY у нас добавились дополнительные строки, правда со значениями NULL, которые вернула нам функция FT_TestFunction, вызванная с параметром, равным идентификатору новой категории.

Теперь в Вашем арсенале есть еще и оператор APPLY, который мы только что рассмотрели, впереди Вас ждет еще одна не менее интересная и полезная возможность языка T-SQL!

Получение данных из внешних источников, функции OPENDATASOURCE, OPENROWSET и OPENQUERY

В Microsoft SQL Server существует возможность запрашивать SQL запросом данные, которые расположены на другом сервере, или это может быть даже не SQL сервер, а просто файл Excel. Иными словами, мы можем написать SQL инструкцию, в которой одна таблица будет на одном сервере, другая на другом сервере, третья в файле Excel, четвертая в базе Access, а если подключить дополнительных поставщиков, то мы можем обращаться даже к СУБД Oracle. И все эти таблицы мы можем объединить, соединить, проанализировать из одного места одной инструкцией. Или, например, Вам просто нужно загружать данные в таблицу из другой базы данных или из обычного файла. Такие запросы называются **- Распределенные запросы.**

Все это можно реализовать с помощью языка T-SQL дополнительными конструкциями, о которых я Вам сейчас и расскажу.

OPENDATASOURCE – функция возвращает ссылку на источник данных, который может использоваться как часть четырехсоставного имени объекта.

OPENROWSET – подключается к источнику данных и выполняет необходимый запрос.

Для функций OPENDATASOURCE и OPENROWSET создавать отдельный объект в базе данных не требуется, в функции передаётся строка подключения.

Данные функции удобно использовать, если Вам требуется разово получить данные из определённых внешних источников, но если Вам требуется постоянно обращаться к таким источникам, то имеет смысл создать специальный объект в SQL Server, который будет настроен на данный источник, и к этому объекту Вы сможете обращаться практически как к обычной базе данных, не указывая при этом строку подключения, как в случае с OPENDATASOURCE и OPENROWSET.

Таким объектом в SQL Server выступает - «**Связанный сервер**» (*Linked Server*). Он настраивается один раз, т.е. при его создании Вы указываете все необходимые настройки, а затем просто обращаетесь к нему по имени, которое Вы укажете.

Для обращения к связанным серверам в SQL сервере существует специальная функция OPENQUERY. Можно, конечно же, обращаться к связанному серверу, не используя данную функцию, но рекомендуется работать со связанным сервером, используя функцию OPENQUERY.

OPENQUERY – функция обращается к связанному серверу и выполняет указанный запрос. На эту функцию можно даже ссылаться в инструкциях по модификации данных, т.е. мы можем изменять данные на связанном сервере.

В качестве примера давайте представим, что нам нужно обратиться к данным в файле Excel. Для этого я создал на диске D файл «TestExcel.xls» со структурой и данными, как в таблице TestTable, Вы тоже создайте.

Но сначала нам нужно включить возможность использования распределенных запросов на сервере, так как она по умолчанию выключена. Включается она с помощью системной хранимой процедуры **sp_configure**, которая отвечает за системные параметры сервера.

```
sp_configure 'show advanced options', 1;
RECONFIGURE;
GO
sp_configure 'Ad Hoc Distributed Queries', 1;
RECONFIGURE;
GO
```

Командой **RECONFIGURE** мы применяем новые параметры.
Теперь давайте подключимся к этому источнику разными способами.

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

Как видите, в функцию мы передаем строку подключения, а потом уже обращаемся к источнику, т.е. к таблице в Excel, которая расположена на листе 1, используя четырехсоставное имя объекта.

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

В данном примере мы уже в функцию передаем полную строку подключения, кстати, вместо [Лист1$] мы могли бы указать в апострофах SQL запрос, например.

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

Со связанным сервером сначала немного сложнее, точнее, его необходимо создать. А создается он с помощью системной процедуры **sp_addlinkedserver**, доступ к данному серверу настраивается процедурой **sp_addlinkedsrvlogin**. Связанный сервер также можно создать и с помощью среды SQL Server Management Studio (*Обозреватель объектов-> Объекты сервера-> Связанные серверы*).

```
--Создание связанного сервера
EXEC dbo.sp_addlinkedserver @server = 'TEST_EXCEL',
                            @srvproduct='OLE DB',
                            @provider='Microsoft.Jet.OLEDB.4.0',
                            @datasrc='D:\TestExcel.xls',
                            @provstr='Excel 8.0'
--Настройки безопасности (авторизации)
EXEC dbo.sp_addlinkedsrvlogin @rmtsrvname= 'TEST_EXCEL',
                            @useself= 'False',
                            @locallogin=NULL,
                            @rmtuser=NULL,
                            @rmtpassword=NULL
```

Связанный сервер создан, я его назвал TEST_EXCEL, теперь мы можем к нему обратиться.

```
--Обращение к связанному серверу
SELECT * FROM TEST_EXCEL...[Лист1$]

--Или с помощью OPENQUERY (рекомендуется)
SELECT * FROM OPENQUERY (TEST_EXCEL, 'SELECT * FROM [Лист$]')
```

Удалить связанный сервер можно с помощью системной хранимой процедуры.

```
--Удаление связанного сервера
EXEC sp_dropserver 'TEST_EXCEL', 'droplogins'
```

Выполнение динамических T-SQL инструкций

Microsoft SQL Server позволяет выполнять инструкции, сформированные динамически, иными словами, Вы можете сформировать текстовую строку, подставляя в нее переменные, и выполнить ее как T-SQL инструкцию, тем самым реализовав динамический код, так как значения переменных также можно формировать динамически на основе данных хранящихся в БД (*например, результат SQL запроса*).

Эта возможность очень полезна, если Вам необходимо, например, выполнять какой-нибудь запрос, где нужно указывать статические значение из базы данных (*неважно, в какой секции, будь то список выборки или условие*), которые на момент запуска инструкции, конечно же, Вам не известны, или известны, но их много и вручную писать очень долго. Ярким примером является оператор PIVOT, рассмотренный нами ранее, который требует вручную перечислять выходные столбцы, количество и название которых заранее нам могут быть просто неизвестны. Ссылку на пример реализации динамического запроса PIVOT я уже приводил в разделе, который посвящен оператору PIVOT.

Для того чтобы сформировать динамическую инструкцию в виде текстовой строки, можно использовать простую конкатенацию строк, в языке T-SQL это можно сделать, например, оператором + (плюс), только в данном случае выражения, участвующие в операции, должны иметь текстовый тип данных (*VARCHAR, например*).

Для выполнения T-SQL инструкций в виде текстовой строки используется все та же команда, которая запускает хранимые процедуры – это **EXEC** (*или EXECUTE*), только в качестве параметра ей передается текстовая строка. Кроме того, в Microsoft SQL Server реализована специальная системная хранимая процедура **sp_executesql**, которая делает в принципе то же самое, но ее работа более безопасна. Да, выполнение динамического SQL кода – **это не безопасно!** Если он выполняется с участием параметров, которые пришли от клиентского приложения. Так как в данные параметры злоумышленники могут внедрить вредоносный код в виде текста, а мы его просто исполним в БД, думая, что нам передали обычные параметры. Поэтому, если Вы будете использовать возможность динамического выполнения SQL инструкций совместно с клиентским приложением, тщательно проверяйте входящие параметры или используйте системную хранимую процедуру sp_executesql, в которой такие проверки уже есть.

Пример с использованием команды EXEC.

```
--Объявляем переменные
DECLARE @SQL_QUERY VARCHAR(200),
        @Var1 INT;

--Присваиваем значение переменным
SET @Var1 = 1;

--Формируем SQL инструкцию
SET @SQL_QUERY = 'SELECT * FROM TestTable WHERE ProductID = ' + CAST(@Var1
AS VARCHAR(10));

--Смотрим на итоговую строку
SELECT @SQL_QUERY AS [TEXT QUERY]

--Выполняем текстовую строку как SQL инструкцию
EXEC (@SQL_QUERY)
```

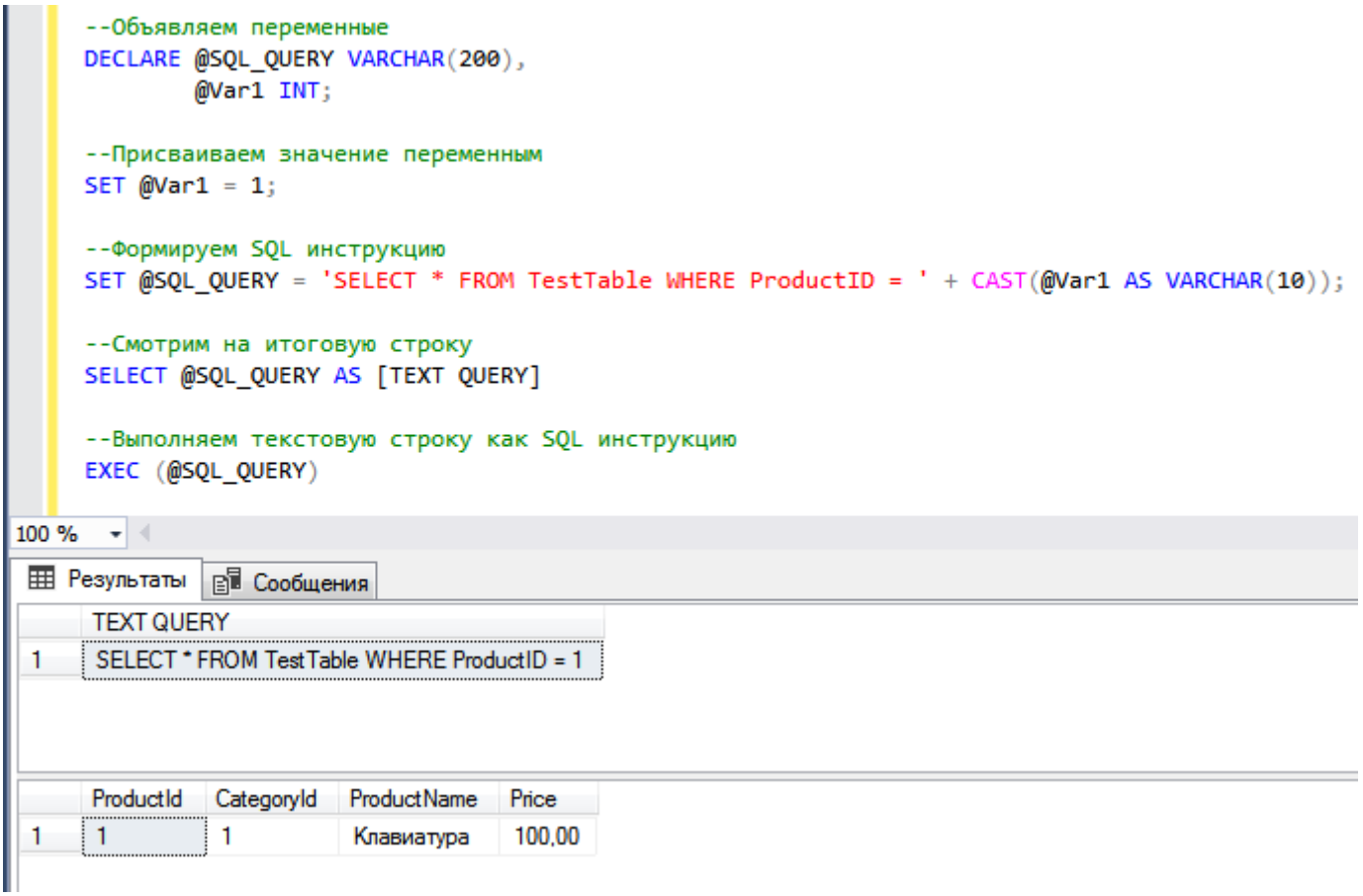


Рис. 109

В данном примере мы сформировали SQL запрос, сохранили его в переменной, и выполнили итоговый текст с помощью команды EXEC. Переменная @Var1 у нас имеет тип данных INT, поэтому, чтобы соединить ее со строкой, мы предварительно преобразовали значение в ней к типу данных VARCHAR. Для наглядности того, какой именно SQL запрос у нас получился, мы просто посмотрели, что у нас хранится в переменной @SQL_QUERY инструкцией SELECT.

Пример с использованием хранимой процедуры sp_executesql.

```
--Объявляем переменные
DECLARE @SQL_QUERY NVARCHAR(200);

--Формируем SQL инструкцию
SELECT @SQL_QUERY = N'SELECT * FROM TestTable WHERE ProductID = @Var1;';

--Смотрим на итоговую строку
SELECT @SQL_QUERY AS [TEXT QUERY]

--Выполняем текстовую строку как SQL инструкцию
EXEC sp_executesql @SQL_QUERY,--Текст SQL инструкции
                   N'@Var1 AS INT', --Объявление переменных в процедуре
                   @Var1 = 1 --Передаем значение для переменных
```

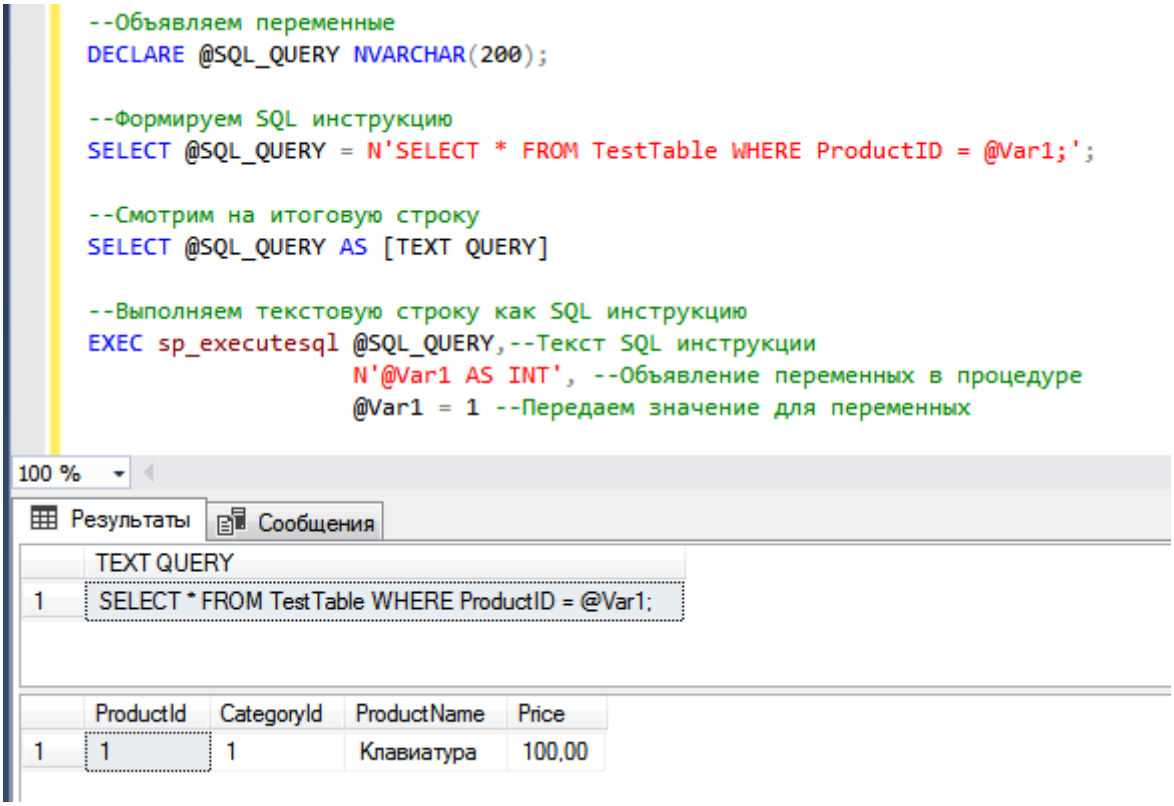


Рис. 110

В этом случае итоговый результат будет точно таким же, как и в примере с EXEC, только динамические значения, у нас это переменная @Var1, мы передали в виде параметра хранимой процедуры sp_executesql, саму переменную мы предварительно объявили также с помощью параметра этой процедуры.

Таким образом, первым параметром указывается текст SQL инструкции, в котором мы уже без конкатенации (оператора +) пишем название переменных в тех местах, где необходимо использовать динамическое значение. Вторым параметром - текст объявления переменных в инструкции, все последующие параметры – это передача значений для переменных в процедуру и, соответственно, подстановка в нашу инструкцию. Все параметры процедуры sp_executesql необходимо передавать в формате Unicode (*тип данных строк должен быть NVARCHAR*).

На этом с программированием на T-SQL мы закончили! Поздравляю Вас, основы T-SQL Вы знаете, но это еще не все. С помощью инструкций T-SQL мы еще можем администрировать сервер! Поэтому переходим к следующей главе, в которой я расскажу про часто используемые операции администрирования Microsoft SQL Server, и покажу, как это делается на T-SQL.

Глава 17 - Администрирование сервера и базы данных

В предыдущих главах данной книги я не затронул еще одну очень важную составляющую обучения работы с Microsoft SQL Server – это администрирование сервера. Поэтому в данной главе я кратко расскажу и покажу некоторые основные операции, с которыми Вы можете столкнуться при работе с SQL сервером. Это будет Вам полезно, так как администрировать SQL сервер можно исключительно с помощью T-SQL инструкций.

Детально рассмотреть все тонкости администрирования Microsoft SQL Server в рамках одной главы у нас не получится, так как данная тема очень обширная, поэтому если у Вас будет направление в работе больше в сторону именно администрирования, то рекомендую Вам почитать отдельные книги по Microsoft SQL Server, которые полностью посвящены администрированию.

Из данной главы Вы узнаете, как создавать пользователей, как назначать им права, как настраиваются параметры базы данных, как переключиться с одной базы на другую, как создавать архив базы данных, и как восстановить базу из этого архива в случае возникновения аварийной ситуации, как переместить базу данных на другой сервер, а также как сжать базу данных, в общем, даже в этой главе Вас ждет еще много чего интересного.

Безопасность

Пользоваться Microsoft SQL Server могут не все, а только те пользователи, которым разрешен доступ к серверу, к базе данных и к определённым объектам. Если нет разрешения, например, на подключение к экземпляру SQL Server, то пользователь, соответственно, не сможет вообще пользоваться SQL сервером, это сделано в целях безопасности, чтобы данные, хранящиеся на сервере, оставались в сохранности. Даже если у пользователя есть доступ к серверу, у него может отсутствовать доступ к определенным базам данных или даже к определенным объектам, например, пользователь работает с базой данных, но при этом ему запрещено удалять или изменять данные по бизнес правилам (*к примеру, это не входит в его обязанности*).

Для этого в Microsoft SQL Server реализована следующая схема:

- **Имя входа** - на сервере есть имена входа, с помощью которых можно подключаться к экземпляру SQL Server. Для того чтобы разрешить пользователю работать с SQL сервером, первое, что нужно сделать - это создать имя входа. Именем входа может выступать как созданная непосредственно на SQL сервере учетная запись или учетная запись в Windows, или доменная учетная запись. Смешанная проверка подлинности также возможна, т.е. можно создать и учетную запись на SQL сервере и использовать, например, доменные учетки. Имена входа Вы можете посмотреть в среде SQL Server Management Studio в обозревателе объектов, контейнер «*Безопасность*»;
- **Пользователь базы данных** – для того чтобы работать с базой данных, имя входа необходимо сопоставить с пользователем базы данных;
- **Роль сервера** – каждому имени входа назначается роль сервера, например, когда Вы устанавливали Microsoft SQL Server, Вам была назначена роль sysadmin, поэтому Вы можете без труда делать абсолютно все, что хотите с SQL сервером, но если имени входа назначена только роль public, то данная учетная запись может делать только то, что ей будет разрешено на уровне объектов сервера;
- **Роль базы данных** – каждому пользователю, для того чтобы предоставить тот или иной доступ к тому или иному объекту, необходимо назначить определенную роль базы данных. Роль базы данных – это своего рода группа прав доступа. Данные права даются уже на выполнение определенных действий. Существуют как предопределенные роли

базы данных, так и пользовательские, которые мы можем создать сами и включить в данную роль только те права, которые нам нужны. Предопределёнными ролями являются, например, роль db_owner, члены которой могут выполнять любые действия в базе данных, а члены роли db_datareader могут только читать данные, осуществлять изменение данных они уже не могут.

Таким образом, чтобы пользователь смог работать с SQL сервером, необходимо:

1. Создать имя входа;
2. Назначить роль сервера данной учетной записи;
3. Создать пользователя базы данных. Сопоставить имя входа с пользователем;
4. Дать пользователю определённые права, т.е. назначить роль базы данных.

Все вышеперечисленное можно реализовать с помощью графического интерфейса Management Studio, но так как книга посвящена языку T-SQL, давайте я покажу, как это делается с помощью T-SQL инструкций.

Для примера давайте создадим имя входа на самом SQL сервере, т.е. проверка подлинности будет осуществляться SQL Server, соответственно, при создании мы зададим пароль. Затем мы назначим этой учетной записи роль sysadmin (*например, добавился администратор сервера*), принудительно назначать серверную роль необязательно, по умолчанию имени входа назначается роль public. Затем мы создадим пользователя, и назначим ему роль db_owner.

```
--Создание имени входа
CREATE LOGIN [TestLogin] WITH PASSWORD='Pa$$w0rd',
                              DEFAULT_DATABASE=[TestDB]
GO
--Назначение роль сервера
EXEC sp_addsrvrolemember @loginame = 'TestLogin',
                              @rolename = 'sysadmin'
GO
--Создание пользователя базы данных и сопоставление с именем входа
CREATE USER [TestUser] FOR LOGIN [TestLogin]
GO
--Назначение пользователю роли базы данных (права доступа к объектам)
EXEC sp_addrolemember 'db_owner', 'TestUser'
GO
```

где,

- ✓ CREATE LOGIN – инструкция создания имени входа;
- ✓ sp_addsrvrolemember – системная хранимая процедура для назначения предопределенных ролей сервера;
- ✓ CREATE USER – инструкция создания пользователя базы данных;
- ✓ sp_addrolemember - системная хранимая процедура для назначения пользователям ролей базы данных.

Также следует пояснить, что параметр DEFAULT_DATABASE в инструкции CREATE LOGIN задает базу данных по умолчанию, которая связывается с именем входа. Параметр FOR LOGIN в инструкции CREATE USER сопоставляет имя входа с пользователем, который создается.

Совет 22

В целях безопасности и сохранности данных рекомендуется четко планировать права доступа для пользователей. Например, не назначать всем пользователям роль db_owner или серверную роль sysadmin, так как пользователи, сами того не зная, могут выполнить действия, которые повлекут за собой неправомерное или некорректное изменение данных или вовсе потерю данных. Последствия в таких случаях могут быть очень серьезные. Поэтому назначайте пользователям только те права, которые им необходимы для выполнения их непосредственной работы.

Команда USE

На одном экземпляре SQL Server могут располагаться несколько баз данных, и если Вы администратор всех баз данных или Вы тот человек, кто сопровождает все базы данных, то Вам необходимо будет переключаться между базами данных, например, для того чтобы в одной базе данных выполнить одну инструкцию, а в другой другую. Это делается с помощью с помощью команды USE, которая меняет контекст подключения на указанную базу данных. Этой командой мы не пользовались, так как работали в одной базе данных, которую мы создали еще в первой главе. Помните, также в первой главе я упоминал, что в Microsoft SQL Server есть и системные базы данных, давайте для примера переключимся на системную базу master, а затем обратно на нашу БД.

```
USE [master]
GO
--Инструкции, которые необходимо выполнить в базе данных master

USE [TestDB]
GO
--Инструкции, которые необходимо выполнить в базе данных TestDB
SELECT * FROM TestTable
```

Совет 23

Если у Вас на одном экземпляре SQL Server несколько баз данных, всегда проверяйте контекст подключения к базе данных перед запуском SQL инструкций. Также рекомендуется в инструкциях для администратора сервера принудительно использовать команду USE для точного указания базы данных, для которой предназначена эта инструкция.

Параметры базы данных

База данных – это объект на сервере, у которого есть параметры, они нужны для управления этой базой данных. Если при создании базы данных не были указаны параметры, они создаются со значением по умолчанию.

Для того чтобы изменить параметр базы данных, можно использовать команду SET в инструкции ALTER DATABASE, также это можно сделать и с помощью SQL Server Management Studio.

Параметров базы данных на самом деле много, сейчас для примера давайте изменим параметр, который позволяет перевести базу данных в режим «*Только для чтения*», иными словами, данные из базы данных можно будет только читать, вносить изменения - нет. Также следует отметить, что для изменения некоторых параметров требуется монопольный доступ к базе данных, т.е. не должно быть никаких подключений к этой базе данных.

Чтобы перевести нашу базу данных TestDB в режим «*Только для чтения*», выполните следующую инструкцию.

```
ALTER DATABASE [TestDB] SET READ_ONLY
```

Чтобы включить обратно возможность изменять данные, выполните вот эту инструкцию.

```
ALTER DATABASE [TestDB] SET READ_WRITE
```

Создание архива базы данных

База данных - это очень важная вещь! Все данные, которые расположены в базе, необходимо сохранить при любых обстоятельствах. Даже в тех случаях, когда у Вас вышел из строя жесткий диск, или администратор, например Вы сами, удалил случайно нужную базу данных. Поэтому для таких случаев базу данных необходимо резервировать, т.е. создавать BACKUP (*архив*) базы данных.

Архив базы данных делается инструкцией **BACKUP DATABASE.**

Например, давайте создадим архив нашей тестовой базы данных.

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

В данном случае мы указали следующие параметры:

- ✓ TO DISK - файл-носитель, в который будет сохранена архивная копия базы данных, он может содержать несколько архивов (*резервных наборов данных*);
- ✓ WITH – команда для указания параметров;
- ✓ NAME – параметр для указания имени резервного набора данных;
- ✓ STATS – данным параметром мы отображаем прогресс выполнения операции создания архива. В нашем случае SQL сервер будет выдавать сообщение после каждых выполненных 10 процентов операции.

Совет 24

Всегда создавайте архивы баз данных. Уделите время и разработайте план и схему резервирования базы данных, у инструкции BACKUP DATABASE много параметров, которые в этом Вам помогут. В итоге у Вас всегда должен быть актуальный архив базы данных на случай непредвиденных ситуаций, данный архив должен храниться в отдельном хранилище (*сервере*), а это хранилище или сервер должен располагаться в помещении, отличном от помещения, в котором расположен сервер с экземпляром SQL Server.

Восстановление базы данных из архива

Перед тем как приступить к рассмотрению операции восстановления базы данных, Вам необходимого знать, что у базы данных есть определенная «**Модель восстановления**» - это метод работы с журналом транзакций во время операций создания архива данных и восстановления базы данных из архива. **Журнал транзакций** – это файл, в который записываются все операции, выполненные в базе данных.

Существуют следующие модели восстановления:

- ➢ **Простая** – используется для баз данных, данные в которых изменяются неинтенсивно, и потеря данных с момента создания последней копии базы не является критичной. Например, копии создаются каждую ночь, если произошел сбой в середине дня, то все данные, которые были сделаны в течение этого дня, будут потеряны. Копия журнала транзакций при такой модели восстановления не создается;
- ➢ **Полная** – используется для баз данных, в которых необходима поддержка длительных транзакций. Это самая надежная модель восстановления, она позволяет восстановить базу

данных до точки сбоя, в случае наличия заключительного фрагмента журнала транзакций. В данном случае копию журнала транзакций необходимо делать по возможности как можно чаще. В журнал транзакций записываются все операции;

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

Определиться с выбором модели восстановления Вы должны на самом начальном этапе, когда будете разрабатывать план и схему резервирования базы данных. Давайте представим, что в нашей тестовой базе данных используется простая модель восстановления, поэтому ранее мы создали полный архив базы данных.

Теперь, когда у нас есть архив базы, мы всегда можем использовать его, для того чтобы восставить базу данных. Это делается инструкцией **RESTORE DATABASE**.

Для выполнения операции восстановления базы данных требуется монопольный доступ к базе данных, т.е. в ней никто не должен работать. Даже Ваш контекст подключения необходимо сменить инструкцией USE на любую другую базу, например, на системную master.

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

Где,

- ✓ FROM DISK – задает файл-носитель с резервными наборами данных;
- ✓ WITH – команда для указания параметров;
- ✓ FILE – порядковый номер резервного набора данных на носителе. Иными словами, если в файле у Вас несколько резервных наборов, то с помощью данного параметра Вы идентифицируете конкретный набор, который необходимо использовать для восстановления;
- ✓ STATS – данным параметром мы отображаем прогресс выполнения операции восстановления данных из архива.

Перемещение базы данных

При работе с базами данных у Вас может возникнуть ситуация, когда Вам нужно будет переместить базу данных, например, на новый сервер. И тут сразу возникает вопрос, как это сделать?

Первое, что Вы должны сделать - это получить монопольный доступ к базе данных, т.е. чтобы к базе не было активных подключений, даже Вашего.

Далее, Вам нужно отсоединить базу данных, переместить физические файлы базы данных в новое расположение, а затем присоединить базу к экземпляру SQL Server.

Для того чтобы отсоединить базу данных, можно использовать системную хранимую процедуру **sp_detach_db**, в качестве параметра она принимает имя базы данных.

Например, чтобы отсоединить базу TestDB от нашего экземпляра SQL Server, необходимо выполнить следующий запрос.

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

Если Вы обновите обозреватель объектов SSMS, то не обнаружите сейчас там базу данных TestDB, так как она отсоединена от данного экземпляра SQL Server. Физические файлы базы данных при этом остались, и мы теперь их можем переместить в нужное нам место, в нашем случае это файлы

TestDB.mdf и TestDB_log.ldf, которые у меня располагаются в каталоге D:\DataBase, у Вас они будут располагаться там, где Вы указали, когда устанавливали SQL Server (*см. Глава 1*).

Для примера давайте просто присоединим базу обратно. Это можно сделать инструкцией CREATE DATABASE, только в качестве параметра указать файлы базы данных и команду **FOR ATTACH**.

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

После выполнения данной инструкции база данных TestDB будет присоединена к экземпляру SQL Server.

Инструкциями отсоединения и присоединения Вы будете пользоваться редко, но знать эти инструкции Вы должны.

Сжатие базы данных

Спустя время размер базы данных растет, при этом фактический размер данных может быть значительно меньше. Это связано с тем, что при добавлении данных в базу физический размер файлов базы данных увеличивается в случае отсутствия в них свободного места, но при удалении данных он не уменьшается. Неиспользуемое пространство в таких случаях потом используется для хранения новых данных.

В случае необходимости высвободить дополнительное свободное место на жестком диске, можно удалить неиспользуемое пространство в файлах базы данных, данная операция называется – **сжатие базы данных**.

Для того чтобы сжать базу данных, можно использовать инструкцию **SHRINKDATABASE**.

```
DBCC SHRINKDATABASE('TestDB')
```

В данном случае будут сжаты все файлы базы данных, однако можно сжать только отдельные файлы. Для этих целей в SQL сервере есть инструкция **SHRINKFILE.**

```
DBCC SHRINKFILE ('TestDB_log', 5)
```

Сейчас мы сжали только файл журнала транзакций (*TestDB_log*), также в качестве параметра мы передали размер, до которого необходимо сжать, в нашем случае до 5 мегабайт.

Совет 25

Операцию сжатия базы данных не рекомендуется выполнять слишком часто, так как она может вызвать фрагментацию индексов и замедлить работу базы данных. Поэтому выполняйте операцию сжатия базы данных только в самых крайних случаях.

Основная практическая часть книги закончена, надеюсь, теперь Вы умеете программировать на T-SQL, писать различные запросы SQL, и выполнять основные операции администрирования с помощью инструкций T-SQL. Может быть, Вам что-то было непонятно, и это естественно. Может быть, я что-то Вам недостаточно полно объяснил, но при этом у Вас есть установленный SQL Server, есть примеры, скорей всего есть собственные данные и задачи, которые необходимо решить, используя язык T-SQL, поэтому не останавливайтесь в обучении, практикуйтесь, читайте книги. После того как Вы прочитали данную книгу и получили базовые знания (*честно скажу, даже больше, чем базовые!*)

языка, теперь Вы можете переходить к более сложным, углубленным областям языка T-SQL, и все новое, что Вы будете осваивать, Вам уже не будет казаться сложным и непонятным.

Последний совет, который я Вам дам, является ключевым. На протяжении всей книги я это всячески отмечал, только прислушавшись к данному совету, Вы станете первоклассным программистом T-SQL и специалистом в области баз данных, кстати, его можно применить в любой области для получения желаемых знаний. Он заключается в следующем.

Совет 26

Если Вы хотите стать профессионалом в своей области, никогда не останавливайтесь в обучении, практикуйтесь, читайте книги, проходите обучающие курсы, изучайте новые технологии в своей сфере, иными словами, станьте «*вечным студентом*» в хорошем смысле этого выражения. Если Вы прекратите работу и изучение какой-то сферы, например, языка T-SQL, уже через совсем непродолжительное время Вы забудете большую часть знаний и потеряете навык работы с этой технологией.

Это еще не все, впереди Вас ждет еще немного, но очень полезной информации о Microsoft SQL Server!

Глава 18 - Microsoft SQL Server во всей красе!

В качестве бонуса я хочу рассказать Вам о том, какие еще нам возможности предоставляет Microsoft SQL Server, так как я уже отмечал, это не просто СУБД – это целый комплекс приложений и компонентов. Может быть, конкретно к программированию на T-SQL это и не относится, но если Вы планируете работать с Microsoft SQL Server, то Вы просто обязаны знать, что умеет SQL Server. Может, Вы с этим и не будете работать, так как в средних и крупных компаниях существует четкое разделение: программист, администратор, аналитик SQL и так далее. Если компания небольшая, то вполне возможно все функции будете выполнять именно Вы, поэтому обязательно прочитайте данную главу, чтобы представлять возможности Microsoft SQL Server, и в случае необходимости использовать соответствующий функционал.

Детально, конечно же, я рассказывать про все не буду, так как каждая из представленных возможностей этой главы заслуживает отдельной книги, я всего лишь Вам расскажу об их существовании, и о том, для чего разработан весь этот функционал.

Итак, представляю Вам Microsoft SQL Server во всей красе!

SQL Server Reporting Services

SQL Server Reporting Services (SSRS) – это службы SQL сервера для построения отчетов. С помощью SSRS можно разрабатывать и формировать как простые табличные отчеты, так и интерактивные, графические и другие более сложные отчеты с использованием диаграмм и других отчетных элементов. Отчеты можно формировать на основе различных источников данных, иными словами, источником может выступать не только Microsoft SQL Server.

SSRS реализован как web-служба, что очень удобно и современно, доступ к отчетам предоставляется, соответственно, через веб-браузер. Однако, это же Misrosoft!, поэтому данный функционал можно интегрировать в свое приложение, используя специальные возможности, например, API интерфейс. Имеется возможность интегрировать SSRS с SharePoint.

SQL Server Reporting Services позволяет настраивать права доступа к отчетам, т.е., например, одной группе (*или конкретному пользователю*) Вы можете дать права на просмотр отчета, а другой - нет.

Все отчеты, сформированные с помощью служб Reporting Services можно экспортировать в разные форматы (*причем с сохранением визуального оформления*), например: Excel, Word, PDF, CSV, XML, TIFF, MHTML (*Web Archive*).

Таким образом, компания Microsoft включила в состав SQL Server отличное, комплексное решение построения отчетов, поэтому если Вам нужен сервис построения отчетов и при этом Вы используете Microsoft SQL Server, то я Вам рекомендую использовать именно службы Reporting Services!

SQL Server Integration Services

SQL Server Integration Services (SSIS) - это службы SQL Server, которые предназначены для автоматизации извлечения, трансформации и консолидации данных из одного типа источников данных в другой тип источника данных. Данный процесс называется **ETL** (*Extract, Transform, Load*) — извлечение, преобразование, загрузка. Также службы Integration Services можно использовать еще и для автоматизации других процессов, не связанных с транспортировкой данных, например, задачи, связанные с обслуживанием баз данных или какие-то действия с файлами в операционной системе, в общем SSIS - это своего рода платформа по автоматизации практически всего.

Вот основные возможности SQL Server Integration Services:

- Импорт и экспорт данных из одного источника данных в другой;
- Импорт данных из нескольких источников, их преобразование, объединение, фильтрация и доставка в один единый источник;
- Запуск исполняемых файлов (exe) или пакетных файлов (bat) в операционной системе;
- Запуск SQL инструкций или хранимых процедур в SQL сервере, а также задач, связанных с обслуживанием баз данных;
- Выполнение операций в файловой системе ОС. Например, создание, копирование, перемещение или удаление файлов и папок;
- И многое другое.

Все это можно комбинировать в одном пакете SSIS, иными словами, Вы можете создать пакет SSIS, в котором, допустим, Вы сначала делаете резервную копию базы данных, а также других источников (*например, файлов Excel*). Затем запускаете хранимую процедуру, которая обновляет данные в базе SQL сервера, потом осуществляется выгрузка данных из всех этих источников, происходит их объединение и доставка в удобном формате пользователю, а для того чтобы пользователь узнал, что его данные подготовлены можно отправить ему письмо по электронной почте.

В общем, возможности SSIS впечатляют!

SQL Server Analysis Services

SQL Server Analysis Services (SSAS) – это службы для работы с многомерными данными (OLAP). Многомерные данные позволяют проектировать, создавать и управлять сложными структурами, которые содержат детализирующие и статистические данные из нескольких источников данных, например, из реляционной базы данных SQL Server.

SSAS – разработаны для бизнес-анализа и хранения данных в формате, который позволяет быстро получить результат обработки большого объема данных.

Иным словами, OLAP – это хранилище данных, в котором хранится агрегированная информация большого массива данных. Данные из такой базы можно получить в десятки раз быстрей, чем из обычной базы.

Службы машинного обучения SQL Server

SQL сервер позволяет использовать для аналитики данных языки R или Python, таким образом, создавая интеллектуальные приложения в базе данных SQL Server. Простыми словами, пользовательское приложение может просто вызывать хранимую процедуру на SQL сервере, в которой будет исполняться код на языке R или Python, анализируя при этом данные в БД, не передавая их пользовательскому приложению. Возможность использования языка R добавилась в 2016 версии SQL Server, поддержка языка Python в 2017 версии.

Использование CLR сборок в SQL Server

Если Вам возможностей языка T-SQL недостаточно или просто Вы привыкли и специализируетесь на платформе .NET Framework, то Вы можете использовать для программирования в базе данных SQL сервера .NET-совместимые языки, такие как C# или VB.NET. Для этого в SQL Server есть возможность интеграции CLR сборок платформы .NET Framework. Таким образом, Вы можете разрабатывать хранимые процедуры, триггеры, функции и определяемые пользователем типы на языках C# или VB.NET.

In-Memory OLTP

In-Memory OLTP – это технология, которая позволят разместить таблицы с данными в памяти системы, при этом Вы можете обращаться к ним точно так же, как и к таблицам, которые расположены

на диске. Благодаря данной технологии можно в десятки раз увеличить производительность SQL инструкций. Функционал In-Memory OLTP доступен, начиная с 2014 версии SQL Server.

Полнотекстовый поиск

Полнотекстовый поиск – это поиск слов или фраз в текстовых данных. Такой тип поиска в основном используется для поиска текста в большом объёме данных, например, таблица с миллионом и более строк, так как он значительно быстрей обычного поиска с использованием оператора LIKE.

С помощью полнотекстового поиска можно реализовать своего рода поисковую систему документов по словам, фразам или словоформам в своей базе данных. Так как, помимо своей быстрой работы, полнотекстовый поиск обладает еще и возможностью ранжировать найденные документы, иными словами, выставлять ранг каждой найденной строке, таким образом, можно найти самые релевантные (*подходящие*) записи под Ваш запрос.

Поддержка технологии JSON

Начиная с SQL Server 2016, стало возможно встроенными средствами работать с форматом данных JSON, примерно также как мы можем работать с форматом XML, например, формировать данные в формате JSON и, соответственно, извлекать данные из JSON строки.

JSON (*Object Notation JavaScript*) – это текстовый формат данных, удобный для чтения, как человеком, так и компьютером. Он представляет собой текст в виде пар «*параметр-значение*», при этом мы можем передавать не только сами данные, но и их структуру.

Компонент Database Mail

Database Mail – это компонент, который предназначен для отправки электронных писем самим SQL сервером. Его можно использовать, например, для уведомления администратора SQL сервера (*допустим нас самих*) о различных событиях, произошедших на экземпляре SQL сервера (*выполнение задач, возникновение ошибок*).

Database Mail позволяет:

- Отправлять электронные письма без использования стороннего клиента;
- Вставлять в письмо результаты запроса;
- Вкладывать файлы в письмо;
- Указывать важность сообщения;
- Отправлять электронные письма нескольким адресатам, а также можно указывать адреса, на которые посылать копию.

Репликация SQL Server

Репликация – это технология, благодаря которой осуществляется копирование данных из одной базы в другую с целью синхронизации этих баз данных. Репликацию используют для того, чтобы обеспечить согласованность данных в различных местах их расположения, а также для повышения доступности и отказоустойчивости баз данных.

SQL Server Agent

SQL Server Agent – это служба SQL Server, которая предназначена для выполнения запланированных административных задач. Например, Вам нужно, чтобы в определенное время запускалась какая-нибудь хранимая процедура, SQL инструкция или операция обслуживания базы данных.

Заключение

Книгу, которая посвящена основам T-SQL и Microsoft SQL Server в целом, Вы прочитали, поздравляю Вас! Теперь у Вас есть базовые знания, позволяющие работать с SQL сервером и разрабатывать инструкции на языке T-SQL, но как уже было замечено – это всего лишь базовые знания, дающие Вам основу для дальнейшего, более углубленного развития.

После того, как я начал заниматься IT технологиями, я пришёл к выражению, которое стал очень сильно понимать, так как оно отражает действительность: «*Чем больше я узнаю, тем больше я понимаю, что ничего не знаю*». Поэтому, поверьте мне, если Вы прочитали одну книгу и считаете, что стали профессионалом - это не так. Всегда есть что-то, что Вы не знаете, и не сталкивались с этим, поэтому Вы должны постоянно стремиться узнать это, узнавать что-то новое! В данном процессе Ваш мозг активизируется, и Вы обязательно получите новую, и в большинстве случаев полезную для Вас информацию.

Если Вы работаете в небольшой компании, честно скажу, Вам даже тех знаний, которые есть в данной книге будет достаточно, для того чтобы хорошо выполнять свои обязанности, но если Вы планируете переходить в более крупную компанию или просто развивать текущую компанию, то обязательно продолжайте обучение.

В процессе работы у Вас, несомненно, будут возникать вопросы, у Вас будут появляться трудности, у Вас будут задачи, на решение которых Ваших знаний будет недостаточно. Поэтому открою Вам секрет, самую полную информацию по T-SQL и Microsoft SQL Server Вы можете найти только в официальной документации (BOL)! Ни одна книга Вам не даст информацию более детально, чем это представлено в официальной документации.

Вся документация по MS SQL Server
https://docs.microsoft.com/ru-ru/sql/

По T-SQL
https://docs.microsoft.com/ru-ru/sql/t-sql/

Просто читать документацию, конечно, сложно и скучно, поэтому и существуют книги, но найти ответ на возникший у Вас вопрос или найти решение какой-то конкретно проблемы в документации можно и, самое главное, нужно! Иными словами, первым делом ищите ответы в документации, если Вы их не нашли, или Вам что-то непонятно, то тогда обращайтесь уже к другим источникам.

Также хочу порекомендовать мой личный сайт – Info-Comp.ru, который в данной книге я уже упоминал. На нем собрано достаточно много полезной информации по T-SQL и SQL Server, кроме того, я там публикую статьи и на другие IT темы, поэтому если Вам понравилась книга, то контент на сайте Вам обязательно также понравится!

Также хотелось бы Вас попросить, если Вас это не затруднит и, если Вам действительно понравилась книга, оставить небольшой отзыв, хоть пару слов о том, что Вы чувствуете, думаете, умеете и знаете после прочтения данной книги. Для меня это будет самая высокая похвала и стимул двигаться вперед и создавать новые, еще более профессиональные и качественные продукты, заранее говорю Вам Спасибо! Оставить отзыв Вы можете на странице моего сайта, посвященной этой книге – http://info-comp.ru/t-sql-book.html.

В заключении, как и обещал, представляю Вам полный перечень моих личных советов по работе с языком T-SQL и Microsoft SQL Server.

Советы по T-SQL и Microsoft SQL Server

1. Всегда визуализируйте информацию в базе данных в виде таблиц, это поможет Вам лучше выстроить связь между таблицами, и написать необходимый SQL запрос.
2. Если Вам нужно хранить в столбце таблицы логический тип данных (*только TRUE или FALSE*), т.е. как Boolean в других языках программирования, то используйте тип данных BIT, не нужно использовать SMALLINT или INT.
3. Не используйте столбцы с типами float и real в условии SQL запросов, так как данные типы не хранят точных значений. Также не используйте float и real в финансовых приложениях, в операциях, связанных с округлением. Для этого лучше использовать decimal, money или smallmoney.
4. Разработку и отладку инструкций на модификацию данных лучше всего осуществлять на тестовых данных, чтобы в случае логической ошибки в запросе ничего страшного не произошло, а на «*боевых*» данных выполнять только готовые, проверенные, отлаженные инструкции.
5. При добавлении данных в таблицу всегда указывайте столбцы, в которые необходимо добавить данные. Это избавит Вас от путаницы и возможных проблем с результатом данной операции. Чем четче Вы будете давать инструкции SQL серверу, тем меньше у Вас будет ошибок!
6. Перед запуском инструкции UPDATE всегда проверяйте наличие и корректность условия, которое предварительно должно быть проверено на SELECT.
7. Запросы на изменение данных лучше всего проектировать так, чтобы они вставляли или изменяли как можно больше строк одной инструкцией, другими словами, если есть возможность заменить несколько инструкций обновления или добавления данных одной инструкцией, заменяйте, так эффективней по производительности.
8. При удалении данных Вы так же, как и при обновлении данных, должны четко понимать и протестировать условие на тестовых данных или с помощью инструкции SELECT, иными словами, Вы должны четко видеть, какие данные Вы сейчас удалите.
9. Всегда перед запуском массовой операции, которая требует серьезных расчетов на большом объеме данных, необходимо заранее провести обслуживание индексов, т.е. запустить процесс реорганизации и перестроения.
10. При создании таблицы всегда явно указывайте ограничение «NOT NULL» у всех столбцов, и по возможности столбцы не должны принимать значения NULL, иными словами, указывайте NOT NULL, только в исключительных случаях разрешайте хранения NULL значений.
11. В процессе планирования таблицы всегда думайте о том, какие именно данные Вам нужны в данной таблице, и на основе этого анализа создайте соответствующие ограничения, это избавит Вас от многочисленных проблем при добавлении данных и тем более, когда Вы будете анализировать эти данные, т.е. делать аналитические выборки.
12. При планировании структуры базы данных всегда выстраивайте связь между таблицами, которые логически связаны между собой, и создавайте соответствующие ограничения.
13. Всегда комментируйте свой код, свои SQL инструкции, там, где у Вас или у другого программиста, который будет читать это код, могут возникнуть вопросы.
14. Всегда перед запуском новой SQL инструкции, в которой есть цикл, проверяйте условие, при котором цикл должен завершиться, для того чтобы избежать попадания в бесконечный цикл.

15. Если в SQL инструкции используются сложные и важные алгоритмы, всегда используйте конструкцию обработки ошибок, тем самым Вы избежите непредвиденных результатов.
16. Любой код, который полностью повторяется 2 и более раз в одной или нескольких SQL инструкциях, выносите в функцию, т.е. напишите функцию, и везде, где нужно использовать этот код, вызывайте эту функцию.
17. При внесении изменений в алгоритм работы функции помните о том, где и для каких целей эта функция используется, так как внесенные изменения отразятся во всех инструкциях, в которых задействована эта функция. Иными словами, не допускайте ситуаций, когда изменение алгоритма работы функции вносит корректные поправки в одни инструкции, для которых Вы и внесли эти изменения, а для других инструкций изменение алгоритма влечет некорректную работу этих SQL инструкций.
18. Если параметров в хранимой процедуре много, и некоторые из них являются необязательными, рекомендую при вызове таких процедур перечислять название параметров. Если параметров всего 1 или 2, то можно название параметров и не указывать, а использовать последовательность передачи значений.
19. При разработке триггера, который реализует некие бизнес правила, всегда четко планируйте и тестируйте все действия триггера, в противном случае в определенных условиях, которые Вы не учтете, триггер может скрытно, т.е. незаметно для Вас, вносить некорректные изменения в базу данных, а когда Вы обнаружите эти некорректные данные или нарушение целостности данных, Вам нужно будет еще отследить причины этих изменений, ведь с первого взгляда явных причин Вы не увидите. Последствия таких ошибок в работе триггера могут быть очень серьезные.
20. Если есть возможность выполнить задачу или реализовать алгоритм без использования курсоров, то делайте это без курсоров. Курсоры применяйте только в тех случаях, когда другого решения у Вас нет. Так как если в алгоритме работы приложения будет задействовано слишком много курсоров, это заметно замедлит работу приложения. Иными словами, используйте курсоры только в самых крайних случаях.
21. Если в одном SQL запросе SELECT Вы прибегаете к использованию нескольких подзапросов в секциях FROM или JOIN, выносите данные подзапросы в обобщенное табличное выражение (*конструкция WITH*) – впоследствии это Вам значительно упростит понимание логики запроса.
22. В целях безопасности и сохранности данных, рекомендуется четко планировать права доступа для пользователей. Например, не назначать всем пользователям роль db_owner или серверную роль sysadmin, так как пользователи, сами того не зная, могут выполнить действия, которые повлекут за собой неправомерное или некорректное изменение данных или вовсе потерю данных. Последствия в таких случаях могут быть очень серьезные. Поэтому назначайте пользователям только те права, которые им необходимы для выполнения их непосредственной работы.
23. Если у Вас на одном экземпляре SQL Server несколько баз данных, всегда проверяйте контекст подключения к базе данных перед запуском SQL инструкций. Также рекомендуется в инструкциях для администратора сервера принудительно использовать команду USE для точного указания базы данных, для которой предназначена эта инструкция.
24. Всегда создавайте архивы баз данных. Уделите время и разработайте план и схему резервирования базы данных, у инструкции BACKUP DATABASE много параметров, которые в этом Вам помогут. В итоге у Вас всегда должен быть актуальный архив базы данных на случай непредвиденных ситуаций, данный архив должен храниться в отдельном хранилище (*сервере*), а это хранилище или сервер должен располагаться в помещении, отличном от помещения, в котором расположен сервер с экземпляром SQL Server.

25. Операцию сжатия базы данных не рекомендуется выполнять слишком часто, так как она может вызвать фрагментацию индексов и замедлить работу базы данных. Поэтому выполняйте операцию сжатия базы данных только в самых крайних случаях.
26. Если Вы хотите стать профессионалом в своей области, никогда не останавливайтесь в обучении, практикуйтесь, читайте книги, проходите обучающие курсы, изучайте новые технологии в своей сфере, иными словами, станьте «*вечным студентом*» в хорошем смысле этого выражения. Если Вы прекратите работу и изучение какой-то сферы, например, языка T-SQL, уже через совсем непродолжительное время Вы забудете большую часть знаний, и потеряете навык работы с этой технологией.

Все, что я хотел рассказать Вам в данной книге, я рассказал, удачи Вам, до свидания!