

Официальное учебное пособие для самоподготовки

Учебный курс

Microsoft®



Microsoft® SQL Server® 2012

Создание запросов

**Ицик Бен-Ган
Деян Сарка
Рон Талмейдж**

Экзамен 70-461



РУССКАЯ РЕДАКЦИЯ

Microsoft



Querying Microsoft SQL Server 2012

Exam 70-461
Training Kit

Itzik Ben-Gan
Dejan Sarka
Ron Talmage

Microsoft® SQL Server® 2012

Создание запросов

**Ицик Бен-Ган
Деян Сарка
Рон Талмейдж**

 **РУССКАЯ РЕДАКЦИЯ**

2014

УДК 004.6

ББК 32.973.26-018.2

Б46

Бен-Ган, И.

Б46 Microsoft® SQL Server® 2012. Создание запросов. Учебный курс Microsoft:
Пер. с англ. / И. Бен-Ган, Д. Сарка, Р. Талмейдж. — М.: Издательство «Русская редакция»,
2014. — 720 с.: ил. + CD-ROM

ISBN 978-5-7502-0432-8

Официальный учебный курс Microsoft рассматривает создание запросов в SQL Server 2012. Описаны создание объектов баз данных с помощью языка T-SQL, реализация типов данных, формирование вложенных и статистических запросов, запрос и управление XML-данными, модификация данных, устранение неполадок и оптимизация.

Книга является ценным справочником и позволяет самостоятельно подготовиться к сдаче экзамена 70-461 для получения сертификата MCSA: SQL Server 2012. На прилагаемом компакт-диске находится оригинальная английская версия книги, вопросы пробного экзамена и другие справочные материалы на английском языке.

Для квалифицированных пользователей и системных администраторов

УДК 004.6

ББК 32.973.26-018.2

© 2014, Translation Russian Edition Publishers.

Authorized Russian translation of the English edition of Training Kit (Exam 70-461): Querying Microsoft® SQL Server® 2012, 1e,
ISBN 978-0-7356-6605-4 © SolidQuality Global SL.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

© 2014, перевод ООО «Издательство «Русская редакция».

Авторизованный перевод с английского на русский язык произведения Training Kit (Exam 70-461): Querying Microsoft® SQL Server® 2012, 1e, ISBN 978-0-7356-6605-4 © SolidQuality Global SL.

Этот перевод оригинального издания публикуется и продаётся с разрешения O'Reilly Media, Inc., которая владеет или распоряжается всеми правами на его публикацию и продажу.

© 2014, оформление и подготовка к изданию, ООО «Издательство «Русская редакция».

Microsoft, а также товарные знаки, перечисленные в списке, расположеннем по адресу: <http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> являются товарными знаками или охраняемыми товарными знаками корпорации Microsoft в США и/или других странах. Все другие товарные знаки являются собственностью соответствующих фирм.

Все названия компаний, организаций и продуктов, а также имена лиц, используемые в примерах, вымышлены и не имеют никакого отношения к реальным компаниям, организациям, продуктам и лицам.

Ичик Бен-Ган, Деян Сарка, Рон Талмейдж

**Microsoft® SQL Server® 2012. Создание запросов.
Учебный курс Microsoft**

Перевод с английского языка Натальи Сержантовой



Подписано в печать 29.11.13.

Формат 70×100^{1/16}. Печать офсетная. Усл. печ. л. 58,05.

Тираж 1000 экз. Заказ №

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9 линия, 12/28

Оглавление

Учебный курс 70-461. Создание запросов в Microsoft SQL Server 2012	1
Введение	3
Системные требования	3
Требования к программному обеспечению и данным SQL Server	4
Требования к аппаратному и программному обеспечению	4
О прилагаемом компакт-диске.....	5
Установка заданий пробного экзамена	5
Использование пробного экзамена.....	5
Удаление ПО пробного экзамена	6
Благодарности	6
Ошибки и опечатки.....	6
Нас интересует ваше мнение.....	7
Оставайтесь на связи	7
Подготовка к экзамену	7
Глава 1. Основы построения запросов.....	8
ЗАНЯТИЕ 1. Основы языка T-SQL	9
Эволюция языка T-SQL.....	9
Использование языка T-SQL в соответствии с реляционной теорией	13
Использование правильной терминологии.....	18
ПРАКТИКУМ. Использование языка T-SQL в соответствии с реляционной теорией.....	18
Задание 1. Определите нереляционные элементы в запросе	18
Задание 2. Преобразуйте нереляционный запрос в реляционный	19
Резюме занятия	20
Закрепление материала	20
ЗАНЯТИЕ 2. Понимание логической обработки запросов	21
T-SQL как декларативный англо-подобный язык	21
Этапы логической обработки запросов	22
1. Обработка предложения <i>FROM</i>	23
2. Фильтрация строк на основании предложения <i>WHERE</i>	24
3. Группирование строк с помощью предложения <i>GROUP BY</i>	25
4. Фильтрация строк с помощью предложения <i>HAVING</i>	26
5. Обработка предложения <i>SELECT</i>	26
6. Управление сортировкой представления	28
ПРАКТИКУМ. Логическая обработка запроса	29
Задание 1. Устранение проблемы с группировкой	29
Задание 2. Устранение проблемы с присвоением псевдонима.....	30
Резюме занятия	30
Закрепление материала	31
УПРАЖНЕНИЯ	31
Упражнение 1. Важность знания теории	32
Упражнение 2. Собеседование на должность специалиста по анализу кода.....	32
Рекомендуемые упражнения	32
Просмотр общедоступных новостных групп по тематике T-SQL и анализ кода	32
Описание логической обработки запросов	33

Глава 2. Начало работы с инструкцией <i>SELECT</i>	34
ЗАНЯТИЕ 1. Использование предложений <i>FROM</i> и <i>SELECT</i>	34
Предложение <i>FROM</i>	35
Предложение <i>SELECT</i>	36
Разделение идентификаторов	39
ПРАКТИКУМ. Использование предложений <i>FROM</i> и <i>SELECT</i>	39
Задание 1. Составление простого запроса и использование псевдонимов таблиц.....	40
Задание 2. Использование псевдонимов столбцов и идентификаторов с разделителями.....	40
Резюме занятия	41
Закрепление материала	41
ЗАНЯТИЕ 2. Работа с типами данных и встроенными функциями.....	42
Выбор нужного типа данных	43
Выбор типов данных для ключей	48
Функции даты и времени	50
Текущая дата и время.....	51
Составляющие даты и времени.....	51
Функции добавления и вычитания даты.....	52
Смещение.....	52
Функции символьных типов данных.....	53
Объединение.....	53
Извлечение подстроки и ее позиция	54
Длина строки	55
Изменение строк.....	56
Форматирование строк	56
Выражение <i>CASE</i> и связанные с ним функции	56
ПРАКТИКУМ. Работа с типами данных и встроенными функциями.....	61
Задание 1. Применение конкатенации строк и использование функций даты и времени	61
Задание 2. Использование дополнительных функций даты и времени	61
Задание 3. Использование строковых данных и функций преобразования	62
Резюме занятия	62
Закрепление материала	63
УПРАЖНЕНИЯ	63
Упражнение 1. Анализ использования типов данных	63
Упражнение 2. Анализ использования функций	64
Рекомендуемые упражнения	64
Анализ типов данных в учебной базе данных.....	64
Анализ образцов кода из электронной документации по SQL Server 2012.....	65
Глава 3. Фильтрация и сортировка данных	66
ЗАНЯТИЕ 1. Фильтрация данных с помощью предикатов	67
Предикаты, троичная логика и аргументы поиска.....	67
Комбинирование предикатов.....	71
Фильтрация символьных данных	73
Фильтрация данных даты и времени.....	75
ПРАКТИКУМ. Фильтрация данных с помощью предикатов	77
Задание 1. Использование предложения <i>WHERE</i> для фильтрации строк со значением <i>NULL</i>	77
Задание 2. Использование предложения <i>WHERE</i> для фильтрации диапазона дат	78
Резюме занятия	79
Закрепление материала	79

ЗАНЯТИЕ 2. Сортировка данных.....	80
Как обеспечить порядок сортировки данных.....	80
Использование предложения <i>ORDER BY</i> для сортировки данных.....	82
ПРАКТИКУМ. Сортировка данных.....	87
Задание 1. Использование предложения <i>ORDER BY</i> с недетерминированной сортировкой	87
Задание 2. Использование предложения <i>ORDER BY</i> с детерминированной сортировкой	88
Резюме занятия	89
Закрепление материала	89
ЗАНЯТИЕ 3. Фильтрация данных с помощью предложений <i>TOP</i> и <i>OFFSET...FETCH</i>	90
Фильтрация данных с помощью предложения <i>TOP</i>	90
Фильтрация данных с помощью <i>OFFSET...FETCH</i>	94
ПРАКТИКУМ. Фильтрация данных с помощью <i>TOP</i> и <i>OFFSET...FETCH</i>	97
Задание 1. Использование конструкции <i>TOP</i>	97
Задание 2. Использование конструкции <i>OFFSET...FETCH</i>	98
Резюме занятия	100
Закрепление материала	100
УПРАЖНЕНИЯ	101
Упражнение 1. Рекомендации по улучшению производительности фильтрации и сортировки	101
Упражнение 2. Обучение разработчика-стажера	101
Рекомендуемые упражнения	102
Перечисление этапов логической обработки запросов и сравнение фильтров.....	102
Что такое детерминизм?	102
Глава 4. Комбинирование наборов данных.....	103
ЗАНЯТИЕ 1. Использование соединений.....	104
Перекрестные соединения	104
Внутренние соединения	106
Внешние соединения	110
Запросы с мультисоединениями	114
ПРАКТИКУМ. Использование соединений.....	116
Задание 1. Сопоставление клиентов и заказов с помощью внутреннего соединения	116
Задание 2. Сопоставление клиентов и заказов с помощью внешнего соединения.....	116
Резюме занятия	118
Закрепление материала	118
ЗАНЯТИЕ 2. Использование подзапросов, табличных выражений и оператора <i>APPLY</i>	119
Подзапросы	119
Независимые подзапросы	120
Коррелированные (связанные) подзапросы	121
Табличные выражения.....	123
Производные таблицы	124
Обобщенные табличные выражения	127
Представления и встроенные табличные функции	129
Оператор <i>APPLY</i>	131
Оператор <i>CROSS APPLY</i>	131
Оператор <i>OUTER APPLY</i>	133
ПРАКТИКУМ. Использование подзапросов, табличных выражений и оператора <i>APPLY</i>	135
Задание 1. Формирование списка продуктов с минимальной ценой за единицу в пределах категории	135

Задание 2. Формирование списка из N продуктов с минимальными ценами за единицу для каждого поставщика	136
Резюме занятия	138
Закрепление материала	138
ЗАНЯТИЕ 3. Использование операторов работы с наборами	139
Операторы <i>UNION</i> и <i>UNION ALL</i>	140
Оператор <i>INTERSECT</i>	142
Оператор <i>EXCEPT</i>	142
ПРАКТИКУМ. Применение операторов работы с наборами	144
Задание 1. Использование оператора работы с наборами <i>EXCEPT</i>	144
Задание 2. Использование оператора работы с наборами <i>INTERSECT</i>	145
Резюме занятия	145
Закрепление материала	145
УПРАЖНЕНИЯ	146
Упражнение 1. Анализ кода.....	146
Упражнение 2. Объяснение операторов работы с наборами	147
Рекомендуемые упражнения	147
Комбинирование наборов данных	147
Глава 5. Группирование и оконные функции.....	149
ЗАНЯТИЕ 1. Написание запросов для группировки данных	150
Работа с одиночным набором группирования.....	150
Работа с несколькими наборами группирования	155
ПРАКТИКУМ. Написание запросов группировки данных	160
Задание 1. Сбор статистической информации о клиентских заказах	160
Задание 2. Определение нескольких наборов группирования.....	161
Резюме занятия	162
Закрепление материала	162
ЗАНЯТИЕ 2. Сведение и отмена сведения данных.....	163
Сведение данных	164
Отмена сведения данных.....	167
ПРАКТИКУМ. Сведение данных	170
Задание 1. Сведение данных с помощью табличного выражения	170
Задание 2. Сведение данных и расчеты.....	171
Резюме занятия	172
Закрепление материала	172
ЗАНЯТИЕ 3. Использование оконных функций	173
Статистические оконные функции	174
Ранжирующие оконные функции	178
Оконные функции смещения	180
ПРАКТИКУМ. Использование оконных функций	182
Задание 1. Использование статистических оконных функций.....	182
Задание 2. Использование оконных ранжирующих функций и функций смещения.....	183
Резюме занятия	185
Закрепление материала	185
УПРАЖНЕНИЯ	186
Упражнение 1. Усовершенствование операций анализа данных	186
Упражнение 2. Собеседование на вакансии разработчика.....	187
Рекомендуемые упражнения	187
Логическая обработка запросов	187

Глава 6. Запросы с полнотекстовым поиском данных	189
ЗАНЯТИЕ 1. Создание полнотекстовых каталогов и индексов.....	190
Компоненты полнотекстового поиска	190
Создание и управление полнотекстовыми каталогами и индексами	193
ПРАКТИКУМ. Создание полнотекстового индекса	195
Задание 1. Создание таблицы и полнотекстовых компонентов	195
Задание 2. Установка семантической базы данных и создание полнотекстового индекса	198
Резюме занятия	199
Закрепление материала	200
ЗАНЯТИЕ 2. Использование предикатов <i>CONTAINS</i> и <i>FREETEXT</i>	200
Предикат <i>CONTAINS</i>	201
Предикат <i>FREETEXT</i>	202
ПРАКТИКУМ. Использование предикатов <i>CONTAINS</i> и <i>FREETEXT</i>	203
Задание 1. Использование предиката <i>CONTAINS</i>	203
Задание 2. Использование синонимов и предиката <i>FREETEXT</i>	205
Резюме занятия	207
Закрепление материала	207
ЗАНЯТИЕ 3. Использование табличных функций полнотекстового и семантического поиска	207
Использование функций полнотекстового поиска.....	208
Использование функций семантического поиска	209
ПРАКТИКУМ. Использование функций полнотекстового и семантического поиска	210
Задание 1. Использование функций полнотекстового поиска.....	210
Задание 2. Использование функций семантического поиска	211
Резюме занятия	212
Закрепление материала	212
УПРАЖНЕНИЯ	213
Упражнение 1. Расширение поиска.....	213
Упражнение 2. Использование семантического поиска.....	213
Рекомендуемые упражнения	213
Знакомство с динамическими административными представлениями, связанными с полнотекстовым поиском, и создание и восстановление резервных копий полнотекстовых каталогов и индексов	214
Глава 7. Запрос и управление XML-данными	215
ЗАНЯТИЕ 1. Возвращение результатов в виде XML с помощью предложения <i>FOR XML</i>	216
Введение в XML	216
Получение XML из реляционных данных	220
Режим <i>FOR XML RAW</i>	221
Режим <i>FOR XML AUTO</i>	221
Режим <i>FOR XML PATH</i>	224
Дробление XML на таблицы.....	226
ПРАКТИКУМ. Использование предложения <i>FOR XML</i>	228
Задание 1. Возвращение XML-документа.....	228
Задание 2. Возвращение XML-фрагмента.....	229
Резюме занятия	230
Закрепление материала	230
ЗАНЯТИЕ 2. Запрос XML-данных с помощью XQuery	231
Основные понятия XQuery.....	231
Типы данных XQuery	234
Функции XQuery.....	234
Навигация	236
Предикаты.....	238

Выражения <i>FLWOR</i>	239
ПРАКТИКУМ. Использование навигации XQuery/XPath.....	242
Задание 1. Использование выражения XPath.....	242
Задание 2. Использование выражения XPath с предикатами	244
Резюме занятия	245
Закрепление материала	245
ЗАНЯТИЕ 3. Использование типа данных XML.....	246
Когда используется тип данных XML	246
Методы типа данных XML	247
Использование типа данных XML для динамической схемы.....	249
XML-индексы	253
ПРАКТИКУМ. Использование методов типа данных XML.....	254
Задание 1. Использование методов <i>value()</i> и <i>exist()</i>	254
Задание 2. Использование методов <i>query()</i> , <i>nodes()</i> и <i>modify()</i>	255
Резюме занятия	257
Закрепление материала	257
УПРАЖНЕНИЯ	257
Упражнение 1. Создание отчетов из XML-данных.....	258
Упражнение 2. Динамическая схема.....	258
Рекомендуемые упражнения	258
Запрос XML-данных	258
Глава 8. Создание таблиц и обеспечение целостности данных	259
ЗАНЯТИЕ 1. Создание и изменение таблиц	259
Введение	260
Создание таблицы.....	261
Определение схемы базы данных	263
Именование таблиц и столбцов.....	264
Выбор типов данных для столбцов.....	266
Значение <i>NULL</i> и значения по умолчанию	267
Свойство идентификатора и порядковые номера.....	268
Вычисляемые столбцы.....	268
Сжатие таблиц	269
Изменение таблицы	270
Выбор индексов таблицы	270
ПРАКТИКУМ. Создание и изменение таблиц.....	271
Задание 1. Использование команды <i>ALTER TABLE</i> для добавления и изменения столбцов	271
Задание 2. Работа с <i>NULL</i> -столбцами в таблице	272
Резюме занятия	273
Закрепление материала	274
ЗАНЯТИЕ 2. Обеспечение целостности данных	274
Использование ограничений.....	275
Ограничения первичного ключа.....	276
Ограничения уникальности.....	277
Ограничения внешнего ключа	278
Проверочные ограничения.....	280
Ограничение по умолчанию.....	282
ПРАКТИКУМ. Обеспечение целостности данных.....	283
Задание 1. Работа с ограничениями первичного и внешнего ключа.....	283
Задание 2. Использование ограничений уникальности.....	285

Резюме занятия	286
Закрепление материала	286
УПРАЖНЕНИЯ	287
Упражнение 1. Работа с ограничениями таблиц	287
Упражнение 2. Использование ограничений уникальности и ограничений по умолчанию.....	287
Рекомендуемые упражнения	288
Создание таблиц и обеспечение целостности данных	288
Глава 9. Проектирование и создание представлений, встроенных функций и синонимов.....	289
ЗАНЯТИЕ 1. Проектирование и реализация представлений и встроенных функций.....	290
Введение	290
Представления.....	290
Синтаксис представлений базы данных	292
Параметры представления	292
Инструкции <i>SELECT</i> и <i>UNION</i> в представлении.....	293
Предложение <i>WITH CHECK OPTION</i>	293
Имена представлений.....	293
Ограничения в представлениях.....	294
Индексированные представления	294
Выполнение запросов из представлений.....	295
Изменение представления	295
Удаление представления.....	295
Модификация данных с помощью представления	296
Секционированные представления	296
Представления и метаданные.....	297
Встроенные функции.....	298
Параметры встроенной функции	300
ПРАКТИКУМ. Работа с представлениями и встроенными функциями	301
Задание 1. Построение представления для отчета.....	301
Задание 2. Преобразование представления во встроенную функцию	303
Резюме занятия	304
Закрепление материала	305
ЗАНЯТИЕ 2. Использование синонимов	306
Создание синонима.....	306
Удаление синонима.....	307
Уровень абстракции	308
Синонимы и ссылки на несуществующие объекты.....	308
Разрешения для работы с синонимами.....	308
Сравнение синонимов с другими объектами баз данных.....	309
ПРАКТИКУМ. Использование синонимов	310
Задание 1. Использование синонимов для создания более описательных имен отчетов	310
Задание 2. Использование синонимов для упрощения межбазовых запросов.....	312
Резюме занятия	313
Закрепление материала	313
УПРАЖНЕНИЯ	314
Упражнение 1. Сравнение представлений, встроенных функций и синонимов.....	314
Упражнение 2. Преобразование синонимов в другие объекты.....	314
Рекомендуемые упражнения	315
Проектирование и создание представлений, встроенных функций и синонимов	315

Глава 10. Вставка, обновление и удаление данных	316
ЗАНЯТИЕ 1. Вставка данных	316
Демонстрационные данные	317
Инструкция <i>INSERT VALUES</i>	317
Инструкция <i>INSERT SELECT</i>	319
Инструкция <i>INSERT EXEC</i>	320
Инструкция <i>SELECT INTO</i>	322
ПРАКТИКУМ. Вставка данных	325
Задание 1. Вставка данных о клиентах без заказов	325
Задание 2. Использование инструкции <i>SELECT INTO</i>	326
Резюме занятия	326
Закрепление материала	327
ЗАНЯТИЕ 2. Обновление данных	328
Демонстрационные данные	328
Инструкция <i>UPDATE</i>	329
Обновление с использованием объединения.....	330
Недетерминированная инструкция <i>UPDATE</i>	332
Инструкция <i>UPDATE</i> и табличные выражения	335
Инструкция <i>UPDATE</i> с использованием переменной	338
<i>UPDATE</i> и принцип единовременности.....	339
ПРАКТИКУМ. Обновление данных	340
Задание 1. Обновление данных с использованием соединений	340
Задание 2. Обновление данных с помощью обобщенного табличного выражения	341
Резюме занятия	342
Закрепление материала	342
ЗАНЯТИЕ 3. Удаление данных	343
Демонстрационные данные	343
Инструкция <i>DELETE</i>	344
Инструкция <i>TRUNCATE</i>	345
Инструкция <i>DELETE</i> на основе объединений	346
Инструкция <i>DELETE</i> с табличными выражениями	347
ПРАКТИКУМ. Удаление и усечение данных	348
Задание 1. Удаление данных с помощью соединений	348
Задание 2. Усечение данных	349
Резюме занятия	349
Закрепление материала	350
УПРАЖНЕНИЯ	351
Упражнение 1. Использование модификаций, поддерживающих оптимизированное введение журнала	351
Упражнение 2. Усовершенствование процесса обновления данных	351
Рекомендуемые упражнения	352
Сравнение инструкций <i>DELETE</i> и <i>TRUNCATE</i>	352
Глава 11. Другие виды модификации данных	353
ЗАНЯТИЕ 1. Использование объекта последовательности и свойства столбца <i>IDENTITY</i>	353
Использование свойства столбца <i>IDENTITY</i>	354
Использование объекта последовательности	358
ПРАКТИКУМ. Использование объекта последовательности	363
Задание 1. Создание последовательности с параметрами по умолчанию	363
Задание 2. Создание последовательности со значениями, отличными от значений по умолчанию	364

Резюме занятия	365
Закрепление материала	366
ЗАНЯТИЕ 2. Слияние данных	367
Использование инструкции <i>MERGE</i>	368
ПРАКТИКУМ. Использование инструкции <i>MERGE</i>	374
Задание 1. Использование инструкции <i>MERGE</i>	374
Задание 2. Выяснение роли предложения <i>ON</i> в инструкции <i>MERGE</i>	376
Резюме занятия	378
Закрепление материала	378
ЗАНЯТИЕ 3. Использование предложения <i>OUTPUT</i>	379
Работа с предложением <i>OUTPUT</i>	379
Инструкция <i>INSERT</i> с предложением <i>OUTPUT</i>	380
Инструкция <i>DELETE</i> с предложением <i>OUTPUT</i>	381
Инструкция <i>UPDATE</i> с предложением <i>OUTPUT</i>	382
Инструкция <i>MERGE</i> с предложением <i>OUTPUT</i>	382
Компонуемый DML	384
ПРАКТИКУМ. Использование предложения <i>OUTPUT</i>	386
Задание 1. Использование предложения <i>OUTPUT</i> в инструкции <i>UPDATE</i>	386
Задание 2. Использование компонуемого DML	387
Резюме занятия	388
Закрепление материала	389
УПРАЖНЕНИЯ	389
Упражнение 1. Лучшее решение для генерации ключей	390
Упражнение 2. Усовершенствование модификаций.....	390
Рекомендуемые упражнения	391
Сравнение старых и новых свойств	391
Глава 12. Реализация транзакций, обработка ошибок и динамический SQL	392
ЗАНЯТИЕ 1. Управление транзакциями и параллелизм	392
Основные понятия транзакций	393
Свойства транзакций ACID	394
Типы транзакций.....	396
Команды транзакций	396
Уровни и состояния транзакций.....	396
Режимы транзакций	397
Режим автоматической фиксации.....	397
Режим неявных транзакций.....	398
Режим явных транзакций.....	399
Вложенные транзакции.....	400
Разметка транзакции	401
Дополнительные параметры транзакции	402
Основные блокировки	403
Совместимость блокировок.....	404
Блокирование.....	405
Взаимоблокировка	405
Уровни изоляции транзакций	407
ПРАКТИКУМ. Реализация транзакций	410
Задание 1. Работа с режимами транзакций	410
Задание 2. Работа с блокированием и взаимоблокированием	411
Задание 3. Работа с уровнями изоляции транзакций.....	413
Резюме занятия	415
Закрепление материала	416

ЗАНЯТИЕ 2. Реализация обработки ошибок.....	417
Обнаружение и инициирование ошибок.....	417
Анализ сообщений об ошибке	418
Команда <i>RAISERROR</i>	419
Команда <i>THROW</i>	420
Функции <i>TRY_CONVERT</i> и <i>TRY_PARSE</i>	421
Обработка ошибок после их обнаружения	422
Неструктурированная обработка ошибок с помощью функции <i>@@ERROR</i>	422
Использование параметра <i>XACT_ABORT</i> с транзакциями	423
Структурированная обработка ошибок с помощью конструкции <i>TRY/CATCH</i>	423
Выбор между <i>THROW</i> и <i>RAISERROR</i> в блоках <i>TRY/CATCH</i>	425
Использование параметра <i>XACT_ABORT</i> с блоками <i>TRY/CATCH</i>	426
ПРАКТИКУМ. Использование обработки ошибок	427
Задание 1. Работа с неструктурной обработкой ошибок.....	427
Задание 2. Использование параметра <i>XACT_ABORT</i> для обработки ошибок.....	428
Задание 3. Структурированная обработка ошибок с помощью блоков <i>TRY/CATCH</i>	430
Резюме занятия	432
Закрепление материала	432
ЗАНЯТИЕ 3. Использование динамического SQL.....	433
Обзор динамического SQL.....	434
Использование динамического SQL	435
Генерация строк T-SQL	435
Инструкция <i>EXECUTE</i>	437
Внедрение кода SQL.....	438
Использование хранимой процедуры <i>sp_executesql</i>	440
ПРАКТИКУМ. Написание и тестирование динамического SQL	441
Задание 1. Генерация строки T-SQL и использование функции <i>QUOTENAME</i>	441
Задание 2. Предупреждение внедрения SQL-кода	442
Задание 3. Использование выходных параметров с процедурой <i>sp_executesql</i>	444
Резюме занятия	445
Закрепление материала	445
УПРАЖНЕНИЯ	446
Упражнение 1. Реализация обработки ошибок	446
Упражнение 2. Реализация транзакций.....	446
Рекомендуемые упражнения	447
Реализация обработки ошибок	447
Глава 13. Разработка и реализация процедур T-SQL.....	448
ЗАНЯТИЕ 1. Разработка и реализация хранимых процедур	449
Основные сведения о хранимых процедурах	449
Проверка существования хранимой процедуры	452
Параметры хранимой процедуры	452
Блок <i>BEGIN/END</i>	453
Инструкция <i>SET NOCOUNT ON</i>	453
Команда <i>RETURN</i> и коды возврата.....	454
Выполнение хранимых процедур	454
Входные параметры	454
Выходные параметры	455
Логика ветвления	456
Конструкция <i>IF/ELSE</i>	457
Конструкция <i>WHILE</i>	458

Команда <i>WAITFOR</i>	461
Инструкция <i>GOTO</i>	461
Разработка хранимых процедур	461
Результаты хранимых процедур.....	461
Вызов других хранимых процедур	462
Хранимые процедуры и обработка ошибок.....	462
Динамический SQL в хранимых процедурах.....	462
ПРАКТИКУМ. Реализация хранимых процедур	463
Задание 1. Создание хранимой процедуры для выполнения административных задач	463
Задание 2. Разработка хранимой процедуры <i>INSERT</i> для уровня доступа к данным	466
Резюме занятия	469
Закрепление материала	470
ЗАНЯТИЕ 2. Реализация триггеров.	471
Триггеры DML	471
Триггеры <i>AFTER</i>	472
Вложенные триггеры <i>AFTER</i>	475
Триггеры <i>INSTEAD OF</i>	475
Функции триггеров DML	476
ПРАКТИКУМ. Написание триггеров DML.....	477
Задание 1. Изучение содержимого вставленных и удаленных таблиц	477
Задание 2. Создание триггера <i>AFTER</i> для выполнения бизнес-правила.....	478
Резюме занятия	480
Закрепление материала	480
ЗАНЯТИЕ 3. Реализация определяемых пользователем функций	481
Основные сведения об определяемых пользователем функциях.....	482
Скалярные определяемые пользователем функции	482
Определяемые пользователем функции с табличным значением.....	484
Встроенная пользовательская функция с табличным значением.....	484
Многооператорная пользовательская функция с табличным значением	485
Ограничения для определяемых пользователем функций	486
Аргументы определяемой пользователем функции.....	486
Производительность в контексте определяемой пользователем функции	487
ПРАКТИКУМ. Написание определяемых пользователем функций	488
Задание 1. Создание скалярной определяемой пользователем функции для вычисления дисконтированной стоимости	488
Задание 2. Создание определяемых пользователем функций с табличным значением	489
Резюме занятия	490
Закрепление материала	491
УПРАЖНЕНИЯ	492
Упражнение 1. Реализация хранимых процедур и определяемых пользователем функций	492
Упражнение 2. Реализация триггеров	492
Рекомендуемые упражнения	493
Использование хранимых процедур, триггеров и определяемых пользователем функций.....	493
Глава 14. Использование инструментов анализа производительности запросов	494
ЗАНЯТИЕ 1. Основные понятия оптимизации запросов	494
Проблемы оптимизации запросов и оптимизатор запросов	495
Подсистема расширенных событий SQL Server, трассировка SQL и приложение SQL Server Profiler	500

ПРАКТИКУМ. Использование подсистемы расширенных событий.....	503
Задание 1. Подготовка инструкции T-SQL и создание сеанса расширенных событий	503
Задание 2. Использование сеанса расширенных событий	505
Резюме занятия	505
Закрепление материала	505
ЗАНЯТИЕ 2. Использование параметров сеанса инструкции <i>SET</i> и анализ планов запросов.....	506
Параметры сеанса инструкции <i>SET</i>	507
Планы выполнения	510
ПРАКТИКУМ. Использование параметров <i>SET</i> на уровне сеанса и планов выполнения	514
Задание 1. Подготовка данных.....	514
Задание 2. Проанализируйте запрос	515
Резюме занятия	516
Закрепление материала	516
ЗАНЯТИЕ 3. Использование динамических административных объектов	517
Введение в динамические административные объекты	518
Наиболее важные динамические административные объекты для настройки объектов.....	518
ПРАКТИКУМ. Использование связанных с индексом динамических административных объектов	521
Задание 1. Нахождение неиспользованных индексов	521
Задание 2. Нахождение отсутствующих индексов	522
Резюме занятия	523
Закрепление материала	523
УПРАЖНЕНИЯ	524
Упражнение 1. Анализ запросов	524
Упражнение 2. Непрерывный мониторинг.....	524
Рекомендуемые упражнения	524
Дополнительные сведения о расширенных событиях, планах выполнения и динамических административных объектах	524
Глава 15. Реализация индексов и статистика.....	526
ЗАНЯТИЕ 1. Реализация индексов.....	527
Кучи и сбалансированные деревья	527
Кучи.....	527
Кластеризованные индексы.....	532
Реализация некластеризованных индексов.....	540
Реализация индексированных представлений.....	545
ПРАКТИКУМ. Анализ некластеризованных индексов.....	547
Задание 1. Реализация некластеризованного индекса в куче	547
Задание 2. Реализация некластеризованного индекса на кластеризованной таблице	548
Резюме занятия	549
Закрепление материала	550
ЗАНЯТИЕ 2. Использование аргументов поиска	550
Поддержка запросов с индексами	550
Аргументы поиска	555
ПРАКТИКУМ. Использование логических операторов <i>OR</i> и <i>AND</i>	558
Задание 1. Поддержка логического оператора <i>OR</i>	558
Задание 2. Поддержка логического оператора <i>AND</i>	560
Резюме занятия	561
Закрепление материала	561

ЗАНЯТИЕ 3. Основные понятия статистики	561
Автоматически создаваемая статистика	562
Ручная поддержка статистики	566
ПРАКТИКУМ. Ручная поддержка статистики.....	567
Задание 1. Запрещение автоматического создания статистики	567
Задание 2. Наблюдение эффекта от отключения автоматического создания статистики.....	568
Резюме занятия	570
Закрепление материала	570
УПРАЖНЕНИЯ	570
Упражнение 1. Просмотр таблицы.....	570
Упражнение 2. Медленные обновления.....	571
Рекомендуемые упражнения	571
Узнайте больше об индексах и о том, как статистические данные влияют на выполнение запроса	571
Глава 16. Основные сведения о курсорах, наборах данных и временных таблицах....	572
ЗАНЯТИЕ 1. Сравнительная оценка использования решений на основе курсоров/итераций и решений на основе наборов данных.....	573
Значение выражения "на основе наборов"	573
Итерации для операций, которые должны выполняться построчно	574
Сравнение курсора и решений на основе наборов в задачах манипулирования данными.....	577
ПРАКТИКУМ. Сравнительная оценка решений на основе курсора и решений на основе наборов данных.....	581
Задание 1. Расчет статистического выражения с помощью курсора	581
Задание 2. Расчет статистического выражения с помощью решения на основе наборов данных	582
Резюме занятия	583
Закрепление материала	583
ЗАНЯТИЕ 2. Сравнение использования временных таблиц и табличных переменных	584
Область действия	584
Язык DDL и индексы.....	586
Физическое представление в базе данных tempdb	588
Транзакции	589
Статистика	590
ПРАКТИКУМ. Выбор оптимального временного объекта	593
Задание 1. Сравнение текущего количества заказов с количеством заказов за предыдущий год с использованием CTE	593
Задание 2. Сравнение текущего количества заказов с количеством заказов за прошлый год с использованием табличных переменных.....	594
Резюме занятия	595
Закрепление материала	596
УПРАЖНЕНИЯ	597
Упражнение 1. Рекомендации по повышению производительности для курсоров и временных объектов	597
Упражнение 2. Указать неточности в ответах	597
Рекомендуемые упражнения	598
Укажите различия	598
Глава 17. Основные сведения о дальнейших аспектах оптимизации.....	599
ЗАНЯТИЕ 1. Общие сведения об итераторах планов.....	600
Методы доступа	600

Алгоритмы соединений.....	607
Другие итераторы плана выполнения	610
ПРАКТИКУМ. Определение итераторов плана выполнения	613
Задание 1. Прогноз плана выполнения	613
Задание 2. Анализ плана выполнения.....	614
Резюме занятия	616
Закрепление материала	616
ЗАНЯТИЕ 2. Использование параметризованных запросов и пакетных операций	617
Параметризованные запросы	617
Обработка пакетов	622
ПРАКТИКУМ. Работа с параметризацией запросов и хранимыми процедурами	628
Задание 1. Работа с запросами, для которых SQL Server не использует повторно план выполнения	628
Задание 2. Изучение повторной компиляции хранимой процедуры.....	629
Резюме занятия	630
Закрепление материала	630
ЗАНЯТИЕ 3. Использование подсказок оптимизатора и структур планов.....	631
Подсказки оптимизатора.....	631
Структуры планов.....	636
ПРАКТИКУМ. Использование подсказок оптимизатора	639
Задание 1. Создание процедуры с указанием подсказки запроса <i>RECOMPILE</i>	639
Задание 2. Тестирование процедуры с указанием запроса <i>RECOMPILE</i>	639
Резюме занятия	640
Закрепление материала	640
УПРАЖНЕНИЯ	641
Упражнение 1. Оптимизация запроса	641
Упражнение 2. Табличная подсказка	641
Рекомендуемые упражнения	641
Анализ планов выполнения запросов и их принудительное выполнение	642
Ответы.....	643
Глава 1.....	643
Занятие 1. Закрепление материала	643
Занятие 2. Закрепление материала	644
Упражнения.....	645
Упражнение 1. Важность знания теории.....	645
Упражнение 2. Собеседование на должность специалиста по анализу кода	645
Глава 2.....	646
Занятие 1. Закрепление материала	646
Занятие 2. Закрепление материала	647
Упражнения.....	648
Упражнение 1. Анализ использования типов данных.....	648
Упражнение 2. Анализ использования функций	648
Глава 3.....	648
Занятие 1. Закрепление материала	648
Занятие 2. Закрепление материала	649
Занятие 3. Закрепление материала	650
Упражнения.....	651
Упражнение 1. Рекомендации по улучшению производительности фильтрации и сортировки.....	651
Упражнение 2. Обучение разработчика-стажера	652

Глава 4.....	652
Занятие 1. Закрепление материала	652
Занятие 2. Закрепление материала	653
Занятие 3. Закрепление материала	654
Упражнения.....	655
Упражнение 1. Анализ кода	655
Упражнение 2. Объяснение операторов работы с наборами.....	655
Глава 5.....	655
Занятие 1. Закрепление материала	655
Занятие 2. Закрепление материала	656
Занятие 3. Закрепление материала	657
Упражнения.....	658
Упражнение 1. Усовершенствование операций анализа данных	658
Упражнение 2. Прохождение собеседования на позицию разработчика	658
Глава 6.....	659
Занятие 1. Закрепление материала	659
Занятие 2. Закрепление материала	660
Занятие 3. Закрепление материала	660
Упражнения.....	661
Упражнение 1. Расширение поиска	661
Упражнение 2. Использование семантического поиска	662
Глава 7.....	662
Занятие 1. Закрепление материала	662
Занятие 2. Закрепление материала	663
Занятие 3. Закрепление материала	663
Упражнения.....	664
Упражнение 1. Создание отчетов из XML-данных	664
Упражнение 2. Динамическая схема	664
Глава 8.....	664
Занятие 1. Закрепление материала	664
Занятие 2. Закрепление материала	665
Упражнения.....	666
Упражнение 1. Работа с ограничениями таблиц	666
Упражнение 2. Использование ограничений уникальности и ограничений по умолчанию	666
Глава 9.....	667
Занятие 1. Закрепление материала	667
Занятие 2. Закрепление материала	667
Упражнения.....	668
Упражнение 1. Сравнение представлений, встроенных функций и синонимов	668
Упражнение 2. Преобразование синонимов в другие объекты	669
Глава 10.....	669
Занятие 1. Закрепление материала	669
Занятие 2. Закрепление материала	670
Занятие 3. Закрепление материала	671
Упражнения.....	672
Упражнение 1. Использование модификаций, поддерживающих оптимизированное ведение журнала	672
Упражнение 2. Усовершенствование процесса обновления данных	672
Глава 11.....	673
Занятие 1. Закрепление материала	673

Занятие 2. Закрепление материала	674
Занятие 3. Закрепление материала	674
Упражнения.....	675
Упражнение 1. Лучшее решение для генерации ключей	675
Упражнение 2. Усовершенствование модификаций	675
Глава 12.....	676
Занятие 1. Закрепление материала	676
Занятие 2. Закрепление материала	677
Занятие 3. Закрепление материала	678
Упражнения.....	679
Упражнение 1. Реализация обработки ошибок.....	679
Упражнение 2. Реализация транзакций	679
Глава 13.....	680
Занятие 1. Закрепление материала	680
Занятие 2. Закрепление материала	680
Занятие 3. Закрепление материала	681
Упражнения.....	682
Упражнение 1. Реализация хранимых процедур и определяемых	682
Упражнение 2. Реализация триггеров.....	683
Глава 14.....	683
Занятие 1. Закрепление материала	683
Занятие 2. Закрепление материала	684
Занятие 3. Закрепление материала	685
Упражнения.....	685
Упражнение 1. Анализ запросов.....	685
Упражнение 2. Непрерывный мониторинг	686
Глава 15.....	686
Занятие 1. Закрепление материала	686
Занятие 2. Закрепление материала	687
Занятие 3. Закрепление материала	687
Упражнения.....	688
Упражнение 1. Просмотр таблицы	688
Упражнение 2. Медленные обновления	688
Глава 16.....	688
Занятие 1. Закрепление материала	688
Занятие 2. Закрепление материала	689
Упражнения.....	690
Упражнение 1. Рекомендации по повышению производительности для курсоров и временных объектов	690
Упражнение 2. Указать неточности в ответах	691
Глава 17.....	691
Занятие 1. Закрепление материала	691
Занятие 2. Закрепление материала	692
Занятие 3. Закрепление материала	693
Упражнения.....	694
Упражнение 1. Оптимизация запроса.....	694
Упражнение 2. Табличная подсказка.....	694
Предметный указатель.....	695
Об авторах	700

Учебный курс 70-461. Создание запросов в Microsoft SQL Server 2012

Тема	Глава	Занятие
1. Создание объектов баз данных		
1.1. Создание и изменение таблиц с помощью синтаксиса T-SQL (простые инструкции)	8	1
1.2. Создание и изменение представлений (простые инструкции)	9 15	1 1
1.3. Проектирование представлений	9	1
1.4. Создание и модификация ограничений (простые инструкции)	8	2
1.5. Создание и изменение DML-триггеров	13	2
2. Работа с данными		
2.1. Запрос данных с помощью инструкции SELECT	1 2 3 4 5 6 8 9 12	1 2 Все занятия Все занятия 3 2 и 3 2 2 3
2.2. Реализация вложенных запросов	4 5 17	2 2 1
2.3. Реализация типов данных	2 3	2 1
2.4. Реализация статистических запросов	5	1 и 3
2.5. Запрос и управление XML-данными	7	Все занятия

(окончание)

Тема	Глава	Занятие
3. Модификация данных		
3.1. Создание и изменение хранимых процедур (простые инструкции)	13	Все занятия
3.2. Модификация данных с помощью инструкций INSERT, UPDATE и DELETE	10 11	Все занятия 3
3.3. Комбинирование наборов данных	2 4 11	2 3 2
3.4. Работа с функциями	2 3 6 13	2 1 3 3
4. Устранение неполадок и оптимизация		
4.1. Оптимизация запросов	12 14 15 17	Оба занятия Все занятия Все занятия Все занятия
4.2. Управление транзакциями	12	1
4.3. Сравнительная оценка построчных операций и операций на основе наборов	16	1
4.4. Реализация обработки ошибок	12 16	2 1

ПРИМЕЧАНИЕ Экзаменационные темы

Перечисленные здесь темы экзаменов актуальны на дату публикации этой книги. Темы экзаменов могут быть изменены в любое время без предварительного уведомления и по собственному усмотрению корпорации Microsoft. Для получения последнего списка тем экзамена посетите Web-сайт Microsoft Learning (Центр обучения) по адресу:

<http://www.microsoft.com/learning/en/us/exam.aspx?ID=70-461&locale=en-us>.

Введение

Этот учебный курс предназначен для специалистов в области информационных технологий (information technology, IT), которые работают с запросами данных в Microsoft SQL Server 2012. Он разработан для IT-профессионалов, планирующих сдавать сертификационный экзамен 70-461 "Querying Microsoft SQL Server 2012". Предполагается, что приступающие к изучению данного учебного курса имеют базовые представления об использовании языка запросов Transact-SQL (T-SQL) для получения данных в SQL Server 2012 и опыт работы с данным продуктом. Хотя эта книга поможет в подготовке к сдаче экзамена 70-461, ее можно рассматривать лишь как часть плана подготовки к экзамену, поскольку для успешной сдачи экзамена необходим практический опыт работы с Microsoft SQL Server 2012.

Рассматриваемый в рамках данного учебного курса и экзамена 70-461 материал относится к технологиям, используемым в Microsoft SQL Server 2012. Разделы данного курса включают все вопросы, необходимые для сдачи экзамена и перечисленные в разделе **Skills Measured** (Навыки) на странице <http://www.microsoft.com/learning/en/us/exam.aspx?ID=70-461&locale=en-us#tab2>.

Изучив этот курс, вы научитесь:

- создавать объекты базы данных;
- работать с данными;
- изменять данные;
- выполнять диагностику и оптимизацию кода T-SQL.

Сведения о том, в какой именно части книги обсуждается конкретная экзаменационная тема, см. на страницах тематического содержания.

Системные требования

Далее перечислены минимальные требования к системе на компьютере, который вы будете использовать для выполнения практических упражнений из книги и для запуска сопроводительного компакт-диска.

Требования к программному обеспечению и данным SQL Server

Далее приведены минимальные требования к программному обеспечению и данным SQL Server.

- **SQL Server 2012.** Необходим доступ к экземпляру SQL Server 2012 с учетной записью, дающей право на создание новых баз данных — предпочтительно относящейся к роли *sysadmin*. Для прохождения данного учебного курса можно использовать практически любой выпуск SQL Server (Standard, Enterprise, Business Intelligence или Developer), как 32-разрядный, так и 64-разрядный. При отсутствии доступа к работающему экземпляру в SQL Server можно установить его пробную версию, загрузив ее со страницы <http://www.microsoft.com/sqlserver/en/us/get-sql-server/try-it.aspx>.
- **Выбор режима установки SQL Server 2012.** В диалоговом окне **Feature Selection** (Выбор компонентов) программы установки SQL Server 2012 выберите по крайней мере следующие компоненты:
 - службы Database Engine Services;
 - Full-Text and Semantic Extractions for Search (Полнотекстовый и семантический поиск);
 - Documentation Components (Компоненты справки);
 - Management Tools - Basic (Средства управления - основные) (обязательно);
 - Management Tools - Complete (Средства управления - полный набор) (рекомендуется).
- **Демонстрационная база данных и исходный код TSQL2012.** Большинство заданий в этом учебном курсе выполняется с использованием демонстрационной базы данных TSQL2012. К данному учебному курсу прилагается архивный файл, содержащий исходные коды, упражнения и файл сценария с именем TSQL2012.sql, который следует использовать для создания демонстрационной базы данных. Архивный файл можно загрузить по адресу <http://go.microsoft.com/fwlink/?LinkId=263548> и с Web-страницы авторов курса по адресу <http://tsql.solidq.com/books/tk70461/>. Для удобства доступа к исходному коду создайте локальную папку с именем C:\TK70461\ (или любым другим именем) и распакуйте содержимое архивного файла в эту папку.

Требования к аппаратному и программному обеспечению

Минимальные требования к аппаратному обеспечению и операционной системе для установки и запуска SQL Server 2012 можно найти на странице [http://msdn.microsoft.com/ru-ru/library/ms143506\(v=sql.110\).aspx](http://msdn.microsoft.com/ru-ru/library/ms143506(v=sql.110).aspx).

О прилагаемом компакт-диске

В состав учебного курса входит сопроводительный компакт-диск, содержащий следующее.

- Пробный экзамен.** Проверить и закрепить полученные знания помогут электронные тесты, которые настраиваются в соответствии с вашими потребностями. Можно пройти пробный сертификационный экзамен 70-461. Для этого на диске содержится более 200 вопросов реального экзамена — их достаточно, чтобы познакомиться с процедурой опроса и оценить степень своей готовности к экзамену.
- Электронная версия книги.** На прилагаемом компакт-диске также находится электронная версия этой книги, которую можно носить с собой вместо печатной копии.

Установка заданий пробного экзамена

Чтобы установить программное обеспечение (ПО) с тренировочными тестами пробного экзамена с прилагаемого компакт-диска на жесткий диск своего компьютера, выполните следующие действия.

1. Вставьте прилагаемый компакт-диск в дисковод компьютера и примите условия лицензионного соглашения — откроется меню компакт-диска.

ПРИМЕЧАНИЕ Если меню не открывается

Если меню не открывается и лицензионное соглашение не выводится, возможно, на вашем компьютере отключена функция автозапуска. Дополнительные инструкции по установке см. в файле Readme.txt на компакт-диске.

2. Выберите пункт **Practice Tests** (Тренировочные тесты) и следуйте появившимся на экране инструкциям.

Использование пробного экзамена

Для запуска программного обеспечения с пробным экзаменом выполните следующие действия.

1. Нажмите кнопку **Start** (Пуск), а затем выберите последовательность команд **All Programs | Microsoft Press Training Kit Exam Prep** (Все программы | Microsoft Press Training Kit Exam Prep). Откроется окно со списком пробных экзаменов учебных курсов Microsoft, установленных на вашем компьютере.
2. Дважды щелкните на тренировочном экзамене, который хотите пройти.

После запуска тренировочного экзамена выберите режим его выполнения: **Certification Mode** (Режим сертификации), **Study Mode** (Режим обучения) или **Custom Mode** (Настраиваемый режим).

- Certification Mode** воспроизводит реальный сертификационный экзамен. В этом режиме необходимо ответить на предлагаемые вопросы за ограниченное время, при этом невозможно ни приостановить, ни перезапустить таймер.

- Study Mode** — тест без ограничения по времени с возможностью просмотра правильных ответов с пояснениями для каждого вопроса.
- Custom Mode** позволяет настраивать параметры теста.

Во всех режимах во время прохождения экзамена пользовательский интерфейс в основном остается одним и тем же, но в зависимости от выбранного режима различные параметры включаются или отключаются.

При просмотре ответа на отдельный вопрос тренировочного экзамена отображается раздел **References** (Справочные материалы), в котором указано, где в учебном курсе можно найти информацию, относящуюся к данному вопросу, и приводятся ссылки на другие источники информации. После того как вы нажмете кнопку **Test Results** (Результаты теста), чтобы оценить выполнение тренировочного теста целиком, можно перейти на вкладку **Learning Plan** (План обучения) и просмотреть список ссылок по всем темам.

Удаление ПО пробного экзамена

Для удаления программного обеспечения с пробным экзаменом из учебного курса используйте компонент **Program and Features** (Программы и компоненты) из Control Panel (Панель управления).

Благодарности

В создании книги принимает участие множество людей, а не только авторы, чьи имена вы видите на ее обложке. Мы хотим выразить признательность следующим людям за всю их работу, позволившую вам сейчас держать эту книгу в руках: Альберту Герберту (Herbert Albert), техническому редактору; Лиле Бен-Ган (Lilach Ben-Gan), руководителю проекта; Кену Джонсу (Ken Jones), рецензенту; Мелани Ярбру (Melanie Yarbrough), выпускающему редактору; Джейми Оделл (Jaime Odell), редактору; Марлен Ламберт (Marlene Lambert), руководителю проекта, Дженнни Кравер (Jeanne Craver), художнику; Джин Тренари (Jean Trenary), специалисту по верстке; Кэти Крауз (Kathy Krause), корректору; Кэрин Форсайт (Kerin Forsyth), редактору.

Ошибки и опечатки

Мы приложили все усилия, чтобы обеспечить точность информации в этой книге и на сопроводительном компакт-диске. Все ошибки, о которых стало известно после публикации книги, перечислены на странице по адресу:

<http://go.microsoft.com/fwlink/?LinkId=263549>

Если вы найдете ошибку, которая еще не отмечена, можно сообщить о ней на той же самой Web-странице. Если вам нужна дополнительная техническая поддержка, обращайтесь в Microsoft Press Book Support по адресу:

mspinput@microsoft.com

Имейте в виду, что по этому адресу не предоставляется техническая поддержка программного обеспечения корпорации Microsoft.

Нас интересует ваше мнение

Издательство Microsoft Press стремится удовлетворить запросы своих читателей, и ваши отзывы станут для нас самой главной наградой. Пожалуйста, сообщите нам свое мнение об этой книге на Web-странице <http://www.microsoft.com/learning/booksurvey>.

Опрос очень короткий, и мы непременно прочтем все ваши комментарии и предложения. Заранее спасибо за ваш отклик!

Оставайтесь на связи

Давайте продолжим наше общение! Нас можно найти в твиттере:

<http://twitter.com/MicrosoftPress>.

Подготовка к экзамену

Сертификационные экзамены корпорации Microsoft — отличная возможность карьерного роста и демонстрации уровня ваших профессиональных знаний и навыков. Предлагаемые экзамены проверяют производственный опыт и теоретические знания программного продукта. Несмотря на то, что производственный опыт нельзя заменить ничем, подготовка с помощью регулярных занятий и самостоятельного выполнения практических заданий сможет помочь вам в подготовке к экзамену. Мы советуем дополнить ваш план подготовки к экзамену комбинацией доступных учебных материалов и курсов. Например, вы можете использовать учебный комплект и другое учебное руководство при подготовке дома и записаться на Microsoft Official Curriculum (Официальный курс обучения Microsoft) для занятий в классе. Выберите наиболее подходящую для вас комбинацию.

ПРИМЕЧАНИЕ Сдача экзамена

Найдите одну минуту (ну хорошо, минуту и две секунды), чтобы просмотреть видеоролик "Passing a Microsoft Exam" (Сдача экзамена Microsoft) по адресу

<http://www.youtube.com/watch?v=Jp5qg2NhgZ0&feature=youtu.be>.

ГЛАВА 1

Основы построения запросов

Темы экзамена

- Работа с данными.
- Запрос данных с помощью инструкций `SELECT`.

Язык Transact-SQL (T-SQL) — это основной язык, используемый для управления данными и их обработки в Microsoft SQL Server. В данной главе обсуждаются основы создания запросов с помощью языка T-SQL. В ней описаны базовые понятия языка, терминология и концепции, которым необходимо следовать при написании кода на T-SQL. Далее будет рассмотрен процесс логической обработки запросов, знание которого необходимо для работы с этим языком.

ВАЖНО!

Вы прочли страницу 7?

На ней представлены полезные сведения о навыках, необходимых для сдачи данного экзамена.

В этой главе, в отличие от остальных глав данного учебного курса, напрямую не рассматриваются конкретные темы экзамена, за исключением обсуждения структуры инструкции `SELECT` — главной инструкции T-SQL, используемой для запроса данных. Тем не менее, представленная в этой главе информация исключительно важна для правильного понимания всей книги.

ПРЕЖДЕ ВСЕГО

Для полного освоения этой главы необходимо иметь:

- знания основных концепций работы с базами данных;
- опыт работы с SQL Server Management Studio (SSMS);
- базовые навыки написания кода на языке T-SQL;
- доступ к экземпляру SQL Server 2012 с установленной учебной базой данных TSQL2012 (подробное описание создания образца базы данных представлено во введении к данной книге).

Занятие 1. Основы языка T-SQL

Многие аспекты информатики, такие как языки программирования, развиваются на основе интуиции и современных тенденций. Без прочных основ их жизненный цикл может быть очень коротким, и даже если им удастся выжить, изменения в тенденциях часто ведут к их быстрому видоизменению. Язык T-SQL отличается от других языков прочными математическими основами. Для написания хорошего SQL-кода вовсе не обязательно быть математиком (хотя это не помешает), но имея четкое представление о базовых концепциях языка, вы сможете лучше понять сам язык. Разумеется, можно писать код T-SQL и без этих знаний — и он даже будет успешно работать, но это то же самое, что есть суп вилкой.

Изучив материал этого занятия, вы сможете:

- ✓ Сформулировать базовые понятия, на которых основывается язык T-SQL
- ✓ Объяснить важность использования T-SQL в соответствии с реляционной теорией
- ✓ Применять правильную терминологию при описании элементов, связанных с T-SQL

Продолжительность занятия — 40 минут.

Эволюция языка T-SQL

Как уже упоминалось, в отличие от многих объектов информационных технологий, язык T-SQL имеет строгие математические основы. Понимание ключевых компонентов этих основ поможет лучше разобраться с самим языком. Тогда при написании кода на T-SQL вы будете думать в терминах именно T-SQL, а не в терминах процедурных языков.

На рис. 1.1 показано развитие языка T-SQL от его математических основ.

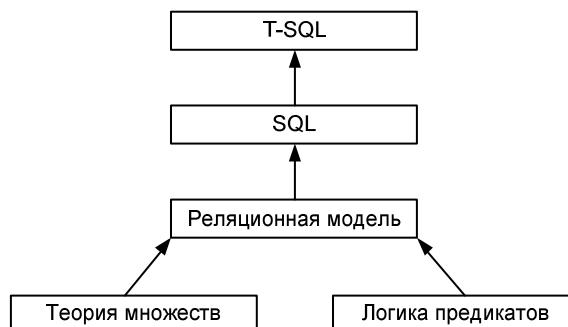


Рис. 1.1. Эволюция языка T-SQL

T-SQL является основным языком управления и обработки данных в системе управления реляционными базами данных (relational database management system, RDBMS) компании Microsoft — SQL Server, как при установке у заказчика, так и в "облачной" модели (Microsoft Windows Azure SQL Database). SQL Server также

поддерживает другие языки, такие как Microsoft Visual C# и Microsoft Visual Basic, но предпочтительным для управления и обработки данных является именно язык T-SQL.

T-SQL — это диалект стандартного языка SQL, который, в свою очередь, является одновременно стандартом Международной организации по стандартизации (International Organization for Standards, ISO) и Американского национального института стандартов (American National Standards Institute, ANSI). Эти два стандарта практически одинаковы для SQL. Данный стандарт SQL продолжает развиваться. Далее приведен список главных версий языка SQL до настоящего времени:

- SQL-86;
- SQL-89;
- SQL-92;
- SQL:1999;
- SQL:2003;
- SQL:2006;
- SQL:2008;
- SQL:2011.

Все ведущие поставщики баз данных, включая Microsoft, реализуют диалект SQL как главный язык для управления и обработки данных в их платформах баз данных. Таким образом, основные элементы языка выглядят одинаково. Но при этом каждый поставщик решает, какие функции реализовать, а какие — нет. Кроме того, стандарт допускает выбор некоторых аспектов при установке. Каждый поставщик также обычно реализует расширения к стандарту, если считает, что он не покрывает некую важную функциональность.

Следование стандарту при написании кода считается наилучшим решением. Это делает код более переносимым. Также более переносимыми становятся и ваши знания, поскольку для вас не составит труда начать работать на новых платформах. Если диалект, на котором вы пишете, поддерживает и стандартный, и нестандартный способы выполнения каких-либо операций, всегда следует выбирать стандартный способ. Нестандартный режим стоит выбирать только в том случае, если он дает заметное преимущество, не предоставляемое стандартным режимом.

Например, T-SQL поддерживает два оператора "не равно": `<>` и `!=`. Первый вариант является стандартным, второй — нет. Решение элементарно: используйте стандартный вариант!

В качестве примера, когда выбор стандартного или нестандартного варианта зависит от обстоятельств, рассмотрим следующее. T-SQL поддерживает несколько функций преобразования исходной величины в конечный (целевой) тип данных. К ним относятся функции `CAST` и `CONVERT`, первая из них является стандартной, вторая — нет. Нестандартная имеет аргумент стиля, не поддерживаемый функцией `CAST`. Так как `CAST` — стандартная функция, она должны быть указана как выбор по умолчанию для конвертации. Функцию `CONVERT` следует выбирать только тогда, когда необходимо использование аргумента стиля.

Еще один пример выбора стандартной формы — завершение инструкций T-SQL. В соответствии со стандартным SQL, инструкции должны завершаться точкой с запятой. Сейчас T-SQL не требует этого для всех инструкций, только в тех случаях, когда возможна неоднозначность элементов кода, как, например, в предложении `WITH` обобщенного табличного выражения (Common table expression, CTE). Необходимо следовать стандарту и завершать все инструкции точкой с запятой, даже если это не требуется текущей версией.

Основой стандартного языка SQL является *реляционная модель*, представляющая собой математическую модель для управления и обработки данных. Реляционная модель была создана и предложена Эдгаром Ф. Коддом в 1969 г. С тех пор она разрабатывается и развивается Крисом Дейтом, Хью Дарвеном и др.

Распространенное заблуждение заключается в том, что название "реляционная" связано с отношениями (связями) между таблицами (иными словами, с внешними ключами). В действительности истинным источником названия этой модели служит математическое понятие "*отношение*" (relation).

Отношение в реляционной модели — это то, что в SQL называется таблицей. Эти два понятия не являются синонимами. Можно сказать, что таблица является попыткой SQL представить отношение (в дополнение к переменной отношения, но это неважно в данном контексте). Возможно, это не очень успешная попытка. Хотя SQL основывается на реляционной модели, они имеют множество различий. Но важно иметь в виду, что понимая принципы данной модели, можно использовать SQL, или, точнее говоря, диалект, с которым вы работаете, в реляционном ключе. Более подробно об этом, в том числе рекомендации читателям, рассказано в разд. *"Использование языка T-SQL в соответствии с реляционной теорией"* далее в этой главе.

Вернемся к понятию отношения, которое SQL пытается представить с помощью таблицы: отношение имеет заголовок и тело. Заголовок — это набор атрибутов (которые SQL пытается представить в виде столбцов), каждый определенного типа. Атрибут определяется именем и именем типа. Тело отношения представляет собой множество кортежей (которые SQL пытается представить в виде строк). Каждый заголовок кортежа — это заголовок отношения. Каждое значение атрибута кортежа имеет соответствующий тип.

Чтобы разобраться в основах SQL, необходимо понять наиболее важные компоненты реляционной модели, теорию множеств и логику предикатов.

Помните, что заголовок отношения — это набор атрибутов, а его тело — множество кортежей. Что же такое множество? Согласно Георгу Кантору, основателю математической теории множеств, *множество* определяется следующим образом: "Под множеством понимается объединение в некое целое M определенных и хорошо различимых объектов нашего созерцания либо размышлений m , которые называются "элементами" множества M " (Джозеф У. Даубен. Георг Кантор. — Princeton University Press, 1990).

В этом определении заключено несколько очень важных понятий, которые, если в них разобраться, напрямую отразятся на процессе написания кода T-SQL. Напри-

мер, возьмем слово "целое". Множество должно рассматриваться как единое целое, т. е. взаимодействие осуществляется не с отдельными элементами множества, а с целым множеством.

Давайте рассмотрим термин "хорошо различимых" — множество не содержит дубликатов. Кодд заметил однажды: "Если нечто есть истина, оно не станет более истинным, если повторить это дважды". К примеру, множество $\{a, b, c\}$ можно рассматривать как равное множеству $\{a, a, b, c, c, c\}$.

Другой важный аспект множества не описан явно в вышеупомянутом определении Кантора, скорее, подразумевается — порядок элементов в множестве не имеет значения. Напротив, в последовательности (которая является упорядоченным множеством), например, порядок элементов важен. Комбинация принципов отсутствия дубликатов и независимости от порядка элементов в множестве означает, что множество $\{a, b, c\}$ равно множеству $\{b, a, c, c, a, c\}$.

Еще одна область математики, на которой основывается реляционная модель — это логика предикатов (логика первого порядка). *Предикат* — это выражение, которое, будучи соотнесенным с некоторым объектом, делает высказывание истинным или ложным. Например, "зарплата более 50 000 долларов" — это предикат. Можно вычислить этот предикат для конкретного работника, получив, таким образом, высказывание. Скажем, у некоего работника зарплата составляет 60 000 долларов. Оценив высказывание для этого работника, мы получим истинное значение. Иными словами, предикат — это параметризованное высказывание.



Реляционная модель использует предикаты в качестве одного из основных элементов. С помощью предикатов можно обеспечить целостность данных, фильтрацию данных и даже определить собственно модель данных.

Прежде всего, нужно определить предложения, которые следует сохранить в базе данных. Например, такое: заказ с идентификационным номером 10248 размещен на 12 февраля 2012 г. клиентом с идентификационным номером 7 и обрабатывается сотрудником с идентификационным номером 3. Затем следует создать предикат из этих утверждений, удалив данные и сохранив заголовок. Помните, что заголовок — это множество атрибутов, каждый из которых определяется именем и именем типа. В данном примере мы имеем `orderid INT`, `orderdate DATE`, `custid INT` и `empid INT`.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. На каких областях математики основывается реляционная модель?
2. В чем заключается разница между T-SQL и SQL?

Ответы на контрольные вопросы

1. Теория множеств и логика предикатов.
2. Язык SQL является стандартом, а T-SQL — это диалект и расширение, реализованные компанией Microsoft в СУБД SQL Server.

Использование языка T-SQL в соответствии с реляционной теорией

Как упоминалось в предыдущем разделе, язык T-SQL базируется на SQL, который, в свою очередь, основывается на реляционной модели. Однако во многих отношениях SQL, а следовательно, и T-SQL отходят от реляционной модели. Тем не менее, язык предоставляет достаточное количество инструментов, позволяющих, в случае хорошего понимания реляционной модели, использовать язык в реляционной манере и, таким образом, писать более правильный код.

К СВЕДЕНИЮ SQL и реляционная теория

Подробную информацию о различиях между SQL и реляционной моделью, а также о том, как использовать SQL в реляционной манере можно найти в книге C. J. Date "SQL and Relational Theory, Second Edition" (O'Reilly Media, 2011)¹. Это прекрасная книга, которую следует прочесть всем практически занимающимся базами данных.

Помните, что отношение имеет заголовок и тело. Заголовок представляет собой множество атрибутов, а тело — это множество кортежей. Помните, что, по определению, множество рассматривается как единое целое. Применительно к T-SQL это означает, что следует писать запросы, взаимодействующие с таблицами как единым целым. Необходимо стараться избегать использования итеративных конструкций, таких как курсоры и циклы, которые перебирают строки по одной. Также следует стараться не думать в итерационной манере, поскольку это неизбежно приведет к итерационным решениям.

Для людей с образованием в области процедурного программирования итерации являются естественным способом взаимодействия с данными (в файле, наборе записей, устройстве считывания данных). Поэтому использование курсоров и других итеративных конструкций в T-SQL является в некоторой степени дополнением к тому, что они уже знают. Однако с точки зрения реляционной модели, правильным является не обращаться к строкам по отдельности, напротив, нужно использовать реляционные операции и возвращать реляционные результаты. В T-SQL это означает написание запросов.

Также необходимо помнить, что множество не имеет дубликатов. Язык T-SQL не всегда следует этому правилу. Например, можно создать таблицу без ключа. В этом случае возможно наличие повторяющихся строк в таблице. Чтобы следовать реляционной теории, надо обеспечить уникальность в таблицах, например, используя первичный ключ или ограничение уникальности.

Даже если в таблице не разрешено дублирование строк, запрос к такой таблице может вернуть повторяющиеся строки как результат. В последующих главах вы найдете продолжение обсуждения дубликатов, здесь же предлагается пример в качестве иллюстрации. Рассмотрим следующий запрос:

```
USE TSQl2012;
SELECT country
FROM HR.Employees;
```

¹ Первое издание книги — Дейт К. Дж. SQL и реляционная теория. Как грамотно писать код на SQL. — СПб.: Символ-плюс, 2010.

Запрос направлен к учебной базе данных TSQL2012. Он возвращает атрибут `country` для сотрудников, хранящихся в таблице `HR.Employees`. В соответствии с реляционной моделью, реляционная операция к связи (отношению) должна вернуть связь. В таком случае это должно означать возвращение множества стран, в которых живут сотрудники, с акцентом на множество, как не имеющее дубликатов. Однако T-SQL не удаляет дубликаты по умолчанию. Далее приведен результат этого запроса.

```
Country
-----
USA
USA
USA
USA
UK
UK
UK
USA
UK
```

В действительности в языке T-SQL больший упор сделан на теорию мульти множеств, чем на теорию множеств. *Мульти множество* (также известное как неупорядоченное множество или супермножество) во многих отношениях подобно множеству, но допускает появление дубликатов. Как уже упоминалось, язык T-SQL предоставляет достаточно инструментов, позволяющих при желании следовать реляционной теории. Например, в этом языке есть предложение `DISTINCT`, позволяющее удалить дубликаты. Далее приведен отредактированный запрос.

```
SELECT DISTINCT country
FROM HR.Employees;
```

А вот результат отредактированного запроса:

```
Country
-----
UK
USA
```



Другой важнейший аспект множества — порядок элементов — не имеет значения. Поэтому, по идеи, строки в таблице следуют в произвольном порядке. Тогда, если выдается запрос к таблице и явно не указывается, что необходимо возвратить строки в определенном порядке, предполагается, что результат будет реляционным. Таким образом, не предполагается получение результата с каким-то определенным порядком строк, независимо от того, что вы знаете о физическом представлении данных, к примеру, когда данные индексированы.

Рассмотрим в качестве примера следующий запрос.

```
SELECT empid, lastname
FROM HR.Employees;
```

При выполнении этого запроса он возвратил следующий результат, который выглядит так, как будто данные отсортированы по столбцу `lastname`.

empid	lastname
5	Buck
8	Cameron
1	Davis
9	Dolgopyatova
2	Funk
7	King
3	Lew
4	Peled
6	Suurs

Даже если строки возвращены в другом порядке, результат все равно будет считаться правильным. SQL Server может выбирать между разными методами физического доступа, подразумевая, что нет необходимости обеспечивать определенный порядок в результате запроса. Например, SQL Server мог бы принять решение распараллелить запрос или сканировать данные в порядке их расположения в файле (что противоположно порядку индекса).

Если все-таки требуется обеспечить определенный порядок представления строк в результате, нужно добавить в запросе предложение `ORDER BY` следующим образом:

```
SELECT empid, lastname  
FROM HR.Employees  
ORDER BY empid;
```

В этом случае результат не является реляционным — это то, что в стандартном SQL называется *курсором*. Порядок строк в результате запроса гарантируется атрибутом `empid`. Результат этого запроса выглядит следующим образом:

empid	lastname
1	Davis
2	Funk
3	Lew
4	Peled
5	Buck
6	Suurs
7	King
8	Cameron
9	Dolgopyatova



Заголовок отношения — это множество атрибутов, которые должны идентифицироваться именем и именем типа. Для этих атрибутов не существует порядка. На-против, T-SQL отслеживает порядковые позиции столбцов на основании порядка их появления в определении таблицы. Когда создается запрос с использованием

SELECT*, гарантировано получение в результате запроса столбцов в порядке определения. T-SQL также позволяет ссылаться на порядковые позиции столбцов результата в предложении ORDER BY следующим образом:

```
SELECT empid, lastname  
FROM HR.Employees  
ORDER BY 1;
```

Кроме того, что эта практика не является реляционной, стоит подумать о возможности ошибки, если в какой-то момент вы измените список SELECT и забудете изменить соответственно список ORDER BY. Таким образом, рекомендуется всегда указывать названия атрибутов, которые необходимо упорядочить.

Язык T-SQL имеет еще одно отличие от реляционной модели, позволяющее определять результирующие столбцы на основании выражения без присвоения имени целевому столбцу. Например, следующий запрос допустим в T-SQL.

```
SELECT empid, firstname + ' ' + lastname  
FROM HR.Employees;
```

Этот запрос генерирует следующий результат:

empid	
1	Sara Davis
2	Don Funk
3	Judy Lew
4	Yael Peled
5	Sven Buck
6	Paul Suurs
7	Russell King
8	Maria Cameron
9	Zoya Dolgopyatova

Но в соответствии с реляционной моделью, все атрибуты должны иметь имена. Чтобы этот запрос был реляционным, необходимо назначить псевдоним (алиас) целевому атрибуту. Это можно сделать, используя предложение AS, как показано в следующем примере:

```
SELECT empid, firstname + ' ' + lastname AS fullname  
FROM HR.Employees;
```

Кроме того, T-SQL позволяет запросу возвращать несколько результирующих столбцов с одним и тем же именем. Например, рассмотрим объединение двух таблиц, T1 и T2, каждая из которых имеет столбец с именем keycol. В T-SQL возможно наличие списка SELECT, который выглядит следующим образом:

```
SELECT T1.keycol, T2.keycol ...
```

Чтобы результат был реляционным, все атрибуты должны иметь уникальные имена, поэтому необходимо использовать разные псевдонимы для выходных атрибутов, как в следующем примере:

```
SELECT T1.keycol AS key1, T2.keycol AS key2 ...
```



Что касается предикатов, в соответствии с *законом исключенного среднего* в математической логике, предикат может принимать значение "истина" или "ложь". Иными словами, предполагается, что предикаты используют двузначную логику. Однако Кодд хотел отразить в своей модели возможность для значений отсутствовать. Он упоминает два вида отсутствующих значений: отсутствующие, но применимые, и отсутствующие, но неприменимые. Возьмем в качестве примера атрибут сотрудника `mobilephone`. Отсутствующее, но применимое значение будет в том случае, если сотрудник имеет мобильный телефон, но не хочет предоставить информацию о нем, скажем, из соображений конфиденциальности. Отсутствующее, но неприменимое значение будет тогда, когда сотрудник просто не имеет мобильного телефона. По Кодду, язык, основанный на его модели, должен предоставлять два разных маркера для этих двух случаев. Еще раз: язык T-SQL, основанный на стандартном языке SQL, реализует только один маркер общего значения, который называется `NULL`, для любого вида отсутствующих значений. Это ведет уже к троичной логике предикатов. А именно, когда предикат сравнивает два значения, например, мобильные номера `mobilephone = '(425) 555-0136'`. Если присутствуют оба номера, результатом будет либо истина, либо ложь. Но если один из них равен `NULL`, результат даст значение троичной логики — неизвестную величину.

Следует заметить, что некоторые считают, будто правильная реляционная модель должна следовать двузначной логике. Но как уже упоминалось, создатель реляционной модели верил в идею поддержки отсутствующих значений и предикатов, что выходит за пределы двузначной логики. Особенно важно с точки зрения написания кода на языке T-SQL понять, что если база данных, к которой создаются запросы, поддерживает `NULL`-значения, их обработка далеко не тривиальна. Таким образом, следует очень хорошо понимать, что происходит, когда данные, которыми вы манипулируете с помощью различных конструкций запросов, таких как фильтрация, сортировка, группировка, объединение или пересечение, включают `NULL`-значения. Следовательно, при написании каждого фрагмента кода на T-SQL надо спросить себя, возможно ли присутствие `NULL`-значений в данных, с которыми вы будете взаимодействовать. Если да, вы должны быть уверены в том, что понимаете использование `NULL`-значений в вашем запросе и что ваши тесты специально направлены на проверку `NULL`-значений.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Назовите два аспекта, в которых T-SQL отклоняется от реляционной модели.
2. Объясните, как можно обойти эти два аспекта в первом вопросе и использовать T-SQL в реляционной манере.

Ответы на контрольные вопросы

1. Отношение имеет тело с определенным набором кортежей. Таблица не обязательно должна иметь ключ. T-SQL позволяет ссылаться на порядковые позиции столбцов в предложении `ORDER BY`.
2. Определите ключ в каждой таблице. Ссылайтесь на имена атрибутов, а не на их порядковые позиции в предложении `ORDER BY`.

Использование правильной терминологии

Понимание терминологии влияет на знания, поэтому следует приложить все необходимые усилия, чтобы понять и использовать правильную терминологию. Часто при обсуждении связанных с T-SQL тем используется неверная терминология. Но даже правильно понимая терминологию, необходимо хорошо чувствовать разницу между одними и теми же терминами в T-SQL и в реляционной модели.

Вот пример неправильной терминологии в T-SQL. Термины "поле" (field) и "запись" (record) для обозначения того, что в T-SQL называется "столбец" (column) и "строка" (row) соответственно. Поля и записи — это физические понятия. Поля — это то, что отображается в пользовательском интерфейсе в клиентских приложениях, а записи — это то, что имеется в файлах и курсорах. Таблицы являются логическими структурами и имеют логические строки и столбцы.

Еще один пример неверной терминологии связан с "NULL-величинами". NULL — это обозначение отсутствующего значения, а не собственно величины. Поэтому правильное использование этого термина — "NULL-маркер" или просто NULL.

Помимо использования правильной терминологии в T-SQL, важно понимать разницу между терминами, используемыми в T-SQL, и их реляционными эквивалентами. Вспомните из предыдущего раздела, что T-SQL пытается представить отношение как таблицу, кортеж как строку и атрибут как столбец, но при этом понятия T-SQL и их реляционные эквиваленты имеют определенные различия. Понимая эти различия, можно и нужно стараться использовать T-SQL с учетом реляционной теории.

Контрольные вопросы

- Почему термины "поле" и "запись" некорректны применительно к столбцу и строке?
- Почему термин "NULL-значение" некорректен?

Ответы на контрольные вопросы

- Потому что "поле" и "запись" описывают физические понятия, тогда как столбцы и строки являются логическими элементами таблицы.
- Потому что NULL не является величиной, напротив, это указание отсутствующего значения.

ПРАКТИКУМ Использование языка T-SQL в соответствии с реляционной теорией

В этом практикуме вы проверите ваши знания об использовании языка T-SQL в соответствии с реляционной теорией.

Задание 1. Определите нереляционные элементы в запросе

В этом задании дан запрос. Ваша задача — определить нереляционные элементы в запросе.

1. Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL201.
2. Введите следующий запрос в окне запросов и выполните его:

```
SELECT custid, YEAR(orderdate)
FROM Sales.Orders
ORDER BY 1, 2;
```

Вы получите следующий результат, представленный в сокращенном виде:

Custid	
1	2007
1	2007
1	2007
1	2008
1	2008
1	2008
2	2006
2	2007
2	2007
2	2008
...	

3. Проанализируйте код и выходной результат. Предполагается, что запрос возвратит для каждого клиента и года заказа ID этого клиента (`custid`) и год заказа (`YEAR(orderdate)`). Обратите внимание, что в запросе отсутствует требование упорядочения представления. Укажите нереляционные аспекты данного запроса.

Ответ. Запрос не присваивает псевдоним выражению `YEAR(orderdate)`, поэтому результирующий атрибут не имеет имени. Запрос может возвращать дубликаты. В запросе определено упорядочивание результата и используются порядковые позиции в предложении `ORDER BY`.

Задание 2. Преобразуйте нереляционный запрос в реляционный

Здесь в качестве исходного используйте запрос из *задания 1*. После того как определены нереляционные элементы в этом запросе, необходимо выполнить соответствующие преобразования, чтобы сделать запрос реляционным.

В шаге 3 *задания 1* вы перечислили нереляционные элементы последнего запроса. Внесите изменения в запрос так, чтобы он стал реляционным.

Для того чтобы сделать данный запрос реляционным, необходимо внести несколько изменений.

1. Определите имя атрибута, назначив псевдоним выражению `YEAR(orderdate)`.
2. Добавьте предложение `DISTINCT` для удаления дубликатов.
3. Также удалите предложение `ORDER BY` для возвращения реляционного результата.

4. Даже если бы существовало требование упорядочения представления (не в рассматриваемом примере), не следует использовать порядковые позиции; вместо этого нужно использовать имена атрибутов. Ваш код должен выглядеть следующим образом:

```
SELECT DISTINCT custid, YEAR(orderdate)
AS orderyear
FROM Sales.Orders;
```

Резюме занятия

- Язык T-SQL имеет строгие математические основы. Он основывается на стандартном SQL, который, в свою очередь, основывается на реляционной модели, в свою очередь основывающейся на теории множеств и логике предикатов.
- Важно понимать принципы реляционной модели и применять их при написании кода T-SQL.
- При описании концепций в T-SQL необходимо использовать правильную терминологию, поскольку это влияет на ваши знания.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Почему важно использовать стандартный SQL-код, где это возможно, и знать, какой код является стандартным, а какой — нет? (Выберите все подходящие варианты.)
 - А. Написание кода с использованием стандартного SQL не является важным.
 - Б. Стандартный SQL-код более переносим между платформами.
 - С. Стандартный SQL-код является более официальным.
 - Д. Знание, что такое стандартный SQL-код, делает ваши знания более платформенно-независимыми.
2. Что из нижеперечисленного не противоречит реляционной модели?
 - А. Использование порядковых позиций для столбцов.
 - Б. Возвращение дубликатов строк.
 - С. Не указание ключа в таблице.
 - Д. Обеспечение того, чтобы все атрибуты в результате запроса имели имена.
3. Какое взаимоотношение существует между SQL и T-SQL?
 - А. T-SQL является стандартным языком, а SQL — это диалект, используемый в Microsoft SQL Server.
 - Б. SQL является стандартным языком, а T-SQL — это диалект, используемый в Microsoft SQL Server.

- C. И SQL, и T-SQL являются стандартными языками.
- D. И SQL, и T-SQL являются диалектами, используемыми в Microsoft SQL Server.

Занятие 2. Понимание логической обработки запросов

У языка T-SQL имеется как физическая, так и логическая сторона. Логическая сторона — это концептуальная интерпретация запроса, которая объясняет, что является корректным результатом запроса. Физическая сторона — это обработка запроса ядром базы данных (компонентом Database Engine). Физическая обработка должна дать результат, определенный логической обработкой запроса. Для достижения этой цели ядро базы данных может использовать оптимизацию. Оптимизация способна изменить последовательность шагов логической обработки запроса или удалить их вовсе, но только при условии, что результат остается тем, который определен логической обработкой запроса. Данное занятие посвящено *логической обработке запросов* — концептуальной интерпретации запроса, определяющей правильный результат.



Изучив материал этого занятия, вы сможете:

- ✓ Понимать логику конструирования языка T-SQL
- ✓ Описать основные этапы логической обработки запросов
- ✓ Объяснить причины некоторых ограничений, принятых в T-SQL

Продолжительность занятия — 90 минут.

T-SQL как декларативный англо-подобный язык

Язык T-SQL, основой которого является стандартный SQL, является декларативным языком, подобным английскому языку. В этом языке "декларативный" означает, что вы определяете то, что вам нужно, в противоположность *императивным* языкам, которые также определяют, как достичь того, что вам нужно. Стандартный SQL описывает логическую интерпретацию декларативного запроса (часть "что"), тогда как за выяснение того, как физически обработать запрос (часть "как"), отвечает компонент Database Engine.

Поэтому важно при изучении логической обработки запросов не делать заключений, касающихся производительности. Причина этого в том, что логическая обработка только лишь определяет правильность запроса. При рассмотрении производительности запроса необходимо понимать, как работает оптимизация. Как уже упоминалось ранее, оптимизация может достаточно сильно отличаться от логической обработки запроса, поскольку разрешается делать изменения при условии, что будет получен именно тот результат, который определен логической обработкой запроса.

Интересно отметить, что стандартный язык SQL изначально так не назывался, его название звучало как SEQUEL, сокращение от "Structured English QUery Language" (структурированный английский язык запросов). Но затем, в результате несогласия с одной из авиакомпаний по поводу торговой марки, язык был переименован в SQL, что означает Structured Query Language (язык структурированных запросов). Тем не менее остается фактом, что инструкции составляются в англо-подобной манере. Например, рассмотрим инструкцию "Bring me a soda from the refrigerator" (принесите мне содовой воды из холодильника). Отметим, что в этой инструкции на английском языке объект стоит перед местоположением. Теперь рассмотрим следующий запрос на языке T-SQL:

```
SELECT shipperid, phone, companyname  
FROM Sales.Shippers;
```

Обратите внимание на сходство между последовательностью элементов в запросе и английским языком. Запрос сначала указывает список SELECT с атрибутами, которые необходимо возвратить, а затем идет предложение FROM с таблицей, к которой направлен запрос.

Не стоит задумываться о последовательности, в которой запрос должен быть логически интерпретирован. Например, как бы вы определили одни и те же инструкции роботу, а не человеку? Первоначальная инструкция принести содовой из холодильника, вероятно, была бы преобразована в нечто подобное следующему: "Подойдите к холодильнику; откройте дверь; достаньте содовую воду; принесите ее мне".

Аналогично, при логической обработке запроса сначала нужно знать, какая таблица запрашивается, прежде чем станет известно, какие атрибуты этой таблицы могут быть возвращены. Таким образом, вопреки последовательности в рассмотренном запросе, логическая обработка запроса должна выполняться следующим образом:

```
FROM Sales.Shippers  
SELECT shipperid, phone, companyname
```

Это элементарный пример запроса всего лишь с двумя предложениями. Разумеется, реальность может быть значительно сложнее. Если вы хорошо разберетесь в принципах логической обработки запросов, вы сможете объяснить многое в способах поведения языка, что в противном случае сделать очень сложно.

Этапы логической обработки запросов

Данный раздел рассматривает логическую обработку запросов и ее этапы. Не стоит беспокоиться, если некоторые принципы, обсуждаемые в этом разделе, не вполне понятны. Последующие главы данного учебного курса содержат более подробные описания, так что после ознакомления с ними изложенный в данном разделе материал станет более понятным. Чтобы полностью разобраться в этих принципах, ознакомьтесь с данным разделом сейчас и затем снова вернитесь к нему после изучения материала глав 2—5.

Главная инструкция, используемая для извлечения данных в языке T-SQL, — это инструкция SELECT. Далее приведены основные предложения, указанные в порядке, в котором они будут вводиться (известном как "порядок ввода с клавиатуры"):

1. SELECT.
2. FROM.
3. WHERE.
4. GROUP BY.
5. HAVING.
6. ORDER BY.

Но как уже упоминалось, последовательность логической обработки запросов, которая является последовательностью концептуальной интерпретации, иная. Она начинается с предложения `FROM`. Далее приведена логическая последовательность обработки запросов шести основных предложений запросов:

1. FROM.
2. WHERE.
3. GROUP BY.
4. HAVING.
5. SELECT.
6. ORDER BY.

Каждый этап работает с одной или более таблицами на входе и возвращает виртуальную таблицу на выходе. Выходная таблица одного этапа рассматривается как входная для следующего этапа. Это полностью согласуется с операциями на отношении, которые дают в результате отношение. Обратите внимание, если указано предложение `ORDER BY`, результат не является реляционным. Этот факт имеет последствия, которые будут обсуждены позже в данном учебном курсе, в *главах 3 и 4*.

В качестве примера рассмотрим следующий запрос:

```
SELECT country, YEAR(hiredate) AS yearhired, COUNT(*) AS numemployees
FROM HR.Employees
WHERE hiredate >= '20030101'
GROUP BY country, YEAR(hiredate)
HAVING COUNT(*) > 1
ORDER BY country, yearhired DESC;
```

Это запрос к таблице `HR.Employees`. Он выводит лишь служащих, которые были приняты на работу в 2003 году или позже. Служащие группируются по стране и году найма. Рассматриваются только группы с двумя и более сотрудниками. Для каждой группы запрос возвращает год найма на работу и количество работников, отсортированные по стране и году приема на работу в порядке убывания.

В следующих разделах представлено краткое описание того, что происходит на каждом этапе в соответствии с логической обработкой запроса.

1. Обработка предложения `FROM`

На первом этапе происходит оценивание предложения `FROM`. Здесь указываются запрашиваемые таблицы и, при необходимости, операторы таблиц, такие как опера-

торы соединения. Если нужно создать запрос к одной таблице, в этом предложении имя таблицы указывается как входная таблица. На выходе этой фазы будет таблица, содержащая все строки из входной таблицы. Именно так получилось в следующем запросе: на входе мы имеем таблицу HR.Employees (9 строк), и на выходе также таблица, содержащая все 9 строк (показана только часть атрибутов).

empid	hireid	country
1	2002-05-01	USA
2	2002-08-14	USA
3	2002-04-01	USA
4	2003-05-03	USA
5	2003-10-17	UK
6	2003-10-17	UK
7	2004-01-02	UK
8	2004-03-05	USA
9	2004-11-15	UK

2. Фильтрация строк на основании предложения WHERE

На следующем этапе выполняется фильтрация строк в соответствии с предикатом в предложении WHERE. Возвращаются только строки, для которых предикат имеет значение true.

СОВЕТ	Подготовка к экзамену
--------------	------------------------------

Строки, в которых предикат имеет значение "ложь" или возвращает неизвестное состояние, не возвращаются.

В данном запросе на этапе фильтрации WHERE фильтруются только строки для служащих, принятых на работу 1 января 2003 года или позже. На этом этапе возвращается 6 строк, которые являются входом для следующего этапа. Далее приведен результат данного этапа.

empid	hiredate	country
4	2003-05-03	USA
5	2003-10-17	UK
6	2003-10-17	UK
7	2004-01-02	UK
8	2004-03-05	USA
9	2004-11-15	UK

Типичная ошибка людей, не понимающих логической обработки запросов, — попытка обратиться в предложении WHERE к псевдониму столбца, определенному в предложении SELECT. Это недопустимо, потому что предложение WHERE вычисляется раньше предложения SELECT. Рассмотрим в качестве примера следующий запрос:

```
SELECT country, YEAR(hiredate) AS yearhired
FROM HR.Employees
WHERE yearhired >= 2003;
```

Запрос не выполняется и возвращает следующую ошибку.

```
Msg 207, Level 16, State 1, Line 3
Invalid column name 'yearhired'.
```

Если вы понимаете, что предложение `WHERE` оценивается перед предложением `SELECT`, вам понятно, что это неверный запрос, поскольку на этом этапе атрибут `yearhired` еще не существует. Вы можете указать выражение `YEAR(hiredate) >= 2003` в предложении `WHERE`. Будет еще лучше, если с целью оптимизации, которая обсуждается в *главах 3 и 15*, вы используете выражение `hiredate >= '20030101'`, как это сделано в исходном запросе.

3. Группирование строк с помощью предложения `GROUP BY`

На данном этапе определяется группа для каждой отдельной комбинации в сгруппированных элементах входной таблицы. Затем каждая входная строка сопоставляется с соответствующей группой. В рассматриваемом нами запросе строки группируются по стране и значению `YEAR(hiredate)`. На этом шаге из шести строк входной таблицы определяются четыре группы. Далее приведены эти группы, а также строки детализации, сопоставленные им (для наглядности избыточная информация удалена).

group country	group YEAR(hiredate)	detail empid	detail country	detail hiredate
UK	2003	5	UK	2003-10-17
		6	UK	2003-10-17
UK	2004	7	UK	2004-01-02
		9	UK	2004-11-15
USA	2003	4	USA	2003-05-03
USA	2004	8	USA	2004-03-05

Как видите, группе UK, 2003 сопоставлены две строки детализации, содержащие служащих с идентификаторами 5 и 6; группе UK, 2004 также соответствуют две строки детализации, для сотрудников с идентификаторами 7 и 9; группа USA, 2003 имеет одну сопоставленную ей строку с сотрудником номер 4; группе USA, 2004 тоже соответствует одна строка детализации, содержащая сотрудника номер 8.

Конечный результат этого запроса содержит одну строку, представляющую каждую группу (без учета фильтрации). Таким образом, выражения на всех этапах после данного этапа группировки имеют определенные ограничения. Все выражения, обрабатываемые на последующих этапах, должны гарантировать одно значение на группу. При ссылке на элемент из списка `GROUP BY` (например, `country`) такая гарантия уже имеется, и поэтому такая ссылка разрешена. Но если требуется ссылка на элемент, не являющийся частью нашего списка `GROUP BY` (к примеру, `empid`), он должен содержаться в какой-либо из агрегатных функций, таких как `MAX` или `SUM`. Это потому, что несколько значений возможны для элемента в пределах одной группы, и единственный способ гарантировать, что только одно из них будетозвращено, — агрегировать эти значения. Дополнительную информацию о группированных запросах можно найти в *главе 5*.

4. Фильтрация строк с помощью предложения *HAVING*

Этот этап также отвечает за фильтрацию строк на основе предиката, но он оценивается после того, как данные сгруппированы; следовательно, он оценивается по каждой группе и выполняет фильтрацию групп как единого целого. Как это принято в языке T-SQL, фильтрующий предикат может принимать значение "истина", "ложь" или "неизвестно". На этом этапе возвращаются только группы, для которых предикат принимает значение "истина". В этом случае предложение *HAVING* использует предикат `COUNT(*) > 1`, что означает фильтрацию по стране и году найма на работу только групп, имеющих более одного сотрудника. Если вы посмотрите на количество строк, сопоставленных каждой группе на предыдущем шаге, вы заметите, что остались только группы UK, 2003 и UK, 2004. Следовательно, результатом этого этапа будут следующие оставшиеся группы, показанные далее вместе с сопоставленными им строками детализации.

group country	group YEAR(hiredate)	detail empid	detail country	detail hiredate
UK	2003	5	UK	2003-10-17
		6	UK	2003-10-17
UK	2004	7	UK	2004-01-02
		9	UK	2004-11-15

Контрольный вопрос

- В чем заключается разница между предложениями *WHERE* и *HAVING*?

Ответ на контрольный вопрос

- Предложение *WHERE* оценивается до группировки строк, и поэтому оно оценивается построчно. Предложение *HAVING* оценивается после того, как строки сгруппированы, и, следовательно, оценивается для группы.

5. Обработка предложения *SELECT*

Пятым является этап, ответственный за обработку запроса *SELECT*. Интересно, что в процессе логической обработки запроса он оценивается почти последним. Это любопытно, учитывая, что предложение *SELECT* стоит первым в запросе.

Этот этап состоит из двух основных шагов. Первый шаг — оценка выражений в списке *SELECT* и создание результирующих атрибутов. Сюда входит присвоение атрибутам имен, если они являются производными от выражений. Помните, что если это группированный запрос, каждая группа представлена в результате единственной строкой. В нашем запросе после обработки фильтра *HAVING* остались 2 группы. Таким образом, данный шаг генерирует две строки. В этом случае список *SELECT* возвращает для каждой группы "страна — год заказа" строку, имеющую следующие атрибуты: `country`, `YEAR(hiredate)` с псевдонимом `yearhired` и `COUNT(*)` с псевдонимом `numemployees`.

Следующий шаг на этом этапе возникает, если указано предложение DISTINCT; в этом случае на данном шаге будут удалены дубликаты. Помните, что язык T-SQL основывается на теории мульти множеств в большей степени, чем на теории множеств, и поэтому если присутствие дубликатов в результате запроса возможно, именно вы отвечаете за их удаление с помощью предложения DISTINCT. В случае нашего запроса этот шаг недопустим. Далее приведен результат данного этапа запроса.

country	yearhired	numemployees
UK	2003	2
UK	2004	2

Если вы хотите вспомнить, как выглядит наш запрос, мы привели его здесь еще раз.

```
SELECT country, YEAR(hiredate) AS yearhired, COUNT(*) AS numemployees
FROM HR.Employees
WHERE hiredate >= '20030101'
GROUP BY country, YEAR(hiredate)
HAVING COUNT(*) > 1
ORDER BY country , yearhired DESC;
```

Пятый этап возвращает реляционный результат. Таким образом, последовательность строк не гарантирована. В случае данного запроса мы имеем предложение ORDER BY, обеспечивающее последовательность строк в результате запроса, но это будет рассмотрено при описании следующего этапа. Важно отметить, что выход этого этапа, который обрабатывает предложение SELECT, остается реляционным.

Также необходимо помнить, что этот этап присваивает псевдонимы столбцам: yearhired и numemployees. Это означает, что вновь создаваемые псевдонимы столбцов не видны предложениям, обрабатываемым на предыдущих этапах, таким как FROM, WHERE, GROUP BY и HAVING.

Обратите внимание, что псевдоним, созданный на этапе SELECT, даже невидим для других выражений, которые появляются в том же списке SELECT. Например, следующий запрос является неверным:

```
SELECT empid, country, YEAR(hiredate) AS yearhired, yearhired - 1
AS prevyear
FROM HR.Employees;
```

Этот запрос генерирует следующую ошибку:

```
Msg 207, Level 16, State 1, Line 1
Invalid column name 'yearhired'.
```

Причина, по которой это невозможно, заключается в том, что концептуально язык T-SQL оценивает все выражения, появляющиеся на том же самом этапе логической обработки запроса, одновременно. Обратите внимание на слово "концептуально". SQL Server не обязательно будет физически обрабатывать все выражения в один и тот же момент времени, но результат должен быть таким, как если бы он это делал. Такое поведение отличается от поведения других языков программирования, в ко-

торых выражения, как правило, рассчитываются справа налево, делая результат, сгенерированный в одном выражении, видимым выражению, появляющемуся справа от него. Но язык T-SQL отличается от них.

КОНТРОЛЬНЫЕ ВОПРОСЫ

- Почему нельзя обращаться к псевдониму столбца, определенному предложением SELECT в предложении WHERE?
- Почему нельзя обращаться к псевдониму столбца, определенному предложением SELECT в том же самом предложении SELECT?

Ответы на контрольные вопросы

- Потому что предложение WHERE логически оценивается на этапе, предшествующем этапу, который оценивает предложение SELECT.
- Потому что все выражения, которые появляются на одном и том же этапе логической обработки запроса, концептуально оцениваются в один и тот же момент времени.

6. Управление сортировкой представления

Данный этап возможен, если в запросе используется предложение ORDER BY. Этот этап отвечает за возвращение результата в определенном порядке представления в соответствии с выражениями из списка ORDER BY. Запрос указывает, что результирующие строки должны быть сначала отсортированы по стране (по умолчанию, в порядке возрастания) и затем по убыванию значения yearhired, давая следующий результат:

Country	yearhired	numemployees
UK	2004	2
UK	2003	2

Заметьте, что предложение ORDER BY является первым и единственным предложением, которое имеет право ссылаться на псевдонимы столбцов, определенные в предложении SELECT. Причина этого в том, что предложение ORDER BY — единственное, которое оценивается после предложения SELECT.

В отличие от предыдущих этапов, где результат был реляционным, выходные данные этого этапа не являются реляционными из-за гарантированной сортировки. Результат этого этапа в стандартном языке SQL называется курсором. Обратите внимание, использование в данном контексте термина "курсор" является концептуальным. Язык T-SQL также поддерживает объект, называемый курсором, который определяется на основании результата запроса, и это позволяет выбирать строки по одной в указанном порядке.

Вам может быть важно возвращение результата запроса в определенной последовательности для удобства представления или если вызывающая сторона должна использовать результат посредством некоторого механизма курсора, который выбирает строки по одной. Но следует помнить, что такая обработка не является реляционной. Если необходима обработка результата запроса в реляционной манере —

например, надо определить на основании запроса табличное выражение типа представления (подробнее об этом написано в главе 4), результат должен быть реляционным. Сортировка данных также может повысить затраты на обработку запроса. Если для вас не имеет значения последовательность, в которой возвращаются результирующие строки, можно избежать этих необязательных затрат, просто не добавляя предложение ORDER BY.

В запросе могут быть указаны параметры фильтра TOP или OFFSET-FETCH. Если они указаны, то же самое предложение ORDER BY, которое обычно используется для определения сортировки представления, также определяет, какие строки фильтровать для этих параметров. Важно отметить, что такой фильтр обрабатывается после того, как этап SELECT оценивает все выражения и удаляет дубликаты (при условии, что было указано предложение DISTINCT). Можно даже считать, что фильтры TOP и OFFSET-FETCH обрабатываются на своем собственном этапе с номером 7. В рассматриваемом запросе не указан такой фильтр, и, таким образом, эта фаза неприменима в этом случае.

ПРАКТИКУМ Логическая обработка запроса

В данном практикуме вам предстоит применить знания о логической обработке запросов.

Задание 1. Устранение проблемы с группировкой

В этом задании предлагается выполнить запрос с группировкой, который завершается ошибкой при попытке его выполнения, а также даны инструкции по исправлению этого запроса.

1. Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
2. Введите следующий запрос в окне запросов и выполните его.

```
SELECT custid, orderid  
FROM Sales.Orders  
GROUP BY custid;
```

Запрос должен возвратить для каждого клиента его ID и максимальное значение ID заказа для этого клиента, но вместо этого он завершился ошибкой. Попытайтесь выяснить, почему запрос не выполнился и что необходимо исправить, чтобы запрос возвратил желаемый результат.

3. Запрос не выполнился, поскольку величина orderid не указана ни в списке GROUP BY, ни в агрегатной функции. Существует несколько возможных значений orderid для клиента. Для исправления запроса нужно применить агрегатную функцию к атрибуту orderid. Задача запроса — возвратить максимальное значение orderid на клиента. Следовательно, нужно использовать агрегатную функцию MAX. Запрос должен выглядеть следующим образом:

```
SELECT custid, MAX(orderid) AS maxorderid  
FROM Sales.Orders  
GROUP BY custid;
```

Задание 2. Устранение проблемы с присвоением псевдонима

В этом задании вам предлагается другой запрос, завершившийся ошибкой, причиной которой на этот раз стала проблема с присвоением псевдонима. Так же как и в первом задании, представлены инструкции по исправлению этого запроса.

1. Очистите окно запросов, введите следующий запрос и выполните его.

```
SELECT shipperid, SUM(freight) AS totalfreight  
FROM Sales.Orders  
WHERE freight > 20000.00  
GROUP BY shipperid;
```

Предполагается, что запрос возвратит только грузоотправителей, для которых полная стоимость поставки больше 20 000, но вместо этого запрос возвратил пустой набор. Попробуйте определить, в чем заключается причина этого.

2. Помните, что фильтр WHERE оценивается построчно — не по группе. Запрос фильтрует индивидуальные заказы со стоимостью поставки более 20 000, а таких не существует. Для исправления запроса нужно применить фильтр по каждой группе грузоотправителей, а не по каждому заказу. Необходима фильтрация полной стоимости поставки на грузоотправителя. Достигнуть этого можно с помощью фильтра HAVING. Попробуйте устранить проблему, используя следующий запрос:

```
SELECT shipperid, SUM(freight) AS totalfreight  
FROM Sales.Orders  
GROUP BY shipperid  
HAVING totalfreight > 20000.00;
```

Но этот запрос также завершается ошибкой. Постарайтесь определить причину этого и что необходимо сделать для достижения желаемого результата.

3. На этот раз проблема заключается в том, что запрос пытается обратиться в предложении HAVING к псевдониму totalfreight, который определен в предложении SELECT. Предложение HAVING оценивается перед предложением SELECT, поэтому псевдоним столбца невидим для него. Для устранения неисправности нужно обратиться к выражению SUM(freight) в предложении HAVING следующим образом:

```
SELECT shipperid, SUM(freight) AS totalfreight  
FROM Sales.Orders  
GROUP BY shipperid  
HAVING SUM(freight) > 20000.00;
```

Резюме занятия

- Язык T-SQL разработан как декларативный язык, в котором инструкции представлены в англо-подобном виде. Таким образом, подобный вводу с клавиатуры порядок предложений в запросе начинается с предложения SELECT.

- Логическая обработка запросов является концептуальной интерпретацией запроса, определяющей корректный результат, и в отличие от подобного вводу с клавиатуры порядка предложений в запросе, она начинается с оценивания предложения `FROM`.
- Понимание логической обработки запросов является исключительно важным для правильного понимания языка T-SQL.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какой из указанных вариантов правильно представляет порядок логической обработки запросов для различных предложений запросов?
 - A. `SELECT > FROM > WHERE > GROUP BY > HAVING > ORDER BY`.
 - B. `FROM > WHERE > GROUP BY > HAVING > SELECT > ORDER BY`.
 - C. `FROM > WHERE > GROUP BY > HAVING > ORDER BY > SELECT`.
 - D. `SELECT > ORDER BY > FROM > WHERE > GROUP BY > HAVING`.
2. Какой вариант является неправильным? (Выберите все подходящие варианты.)
 - A. Ссылка на атрибут, который сгруппирован в предложении `WHERE`.
 - B. Ссылка на выражение в предложении `GROUP BY`; например, `GROUP BY YEAR(orderdate)`.
 - C. В запросе с группировкой ссылка в списке `SELECT` на атрибут, который не является частью списка `GROUP BY` и не входит в агрегатную функцию.
 - D. Ссылка на псевдоним, определенный в предложении `SELECT`, в предложении `HAVING`.
3. Что справедливо для результата запроса, не содержащего предложение `ORDER BY`?
 - A. Он является реляционным, если другие требования реляционности соблюdenы.
 - B. Он не может иметь дубликатов.
 - C. Гарантирует ту же последовательность строк в выходных данных, что и последовательность ввода.
 - D. Гарантирует ту же последовательность строк в выходных данных, что и в кластеризованном индексе.

Упражнения

В следующих упражнениях вы примените полученные знания о создании запросов в T-SQL. Ответы на эти вопросы можно найти в *приложении "Ответы" в конце книги*.

Упражнение 1. Важность знания теории

Вы и ваш коллега по команде вступили в дискуссию о важности понимания теоретических основ языка T-SQL. Ваш коллега возражает вам, что понимание этих основ не имеет никакого значения и для того, чтобы быть хорошим разработчиком и писать правильный код, достаточно изучить технические аспекты T-SQL. Ответьте на следующие вопросы, заданные вам вашим коллегой:

1. Можете ли вы привести пример элемента теории множеств, который может улучшить ваше понимание языка T-SQL?
2. Можете ли вы объяснить, почему понимание реляционной модели важно для тех, кто пишет код на языке T-SQL?

Упражнение 2. Собеседование на должность специалиста по анализу кода

Вы проходите собеседование на должность специалиста по анализу кода, чтобы помочь улучшить качество кода. Приложение, с которым работает компания, использует запросы, написанные неквалифицированными специалистами. Эти запросы имеют множество проблем, включая логические дефекты. Ваш интервьюер задал несколько вопросов и хочет получить краткий ответ, состоящий из нескольких фраз, на каждый вопрос. Ответьте на следующие вопросы, заданные вам вашим интервьюером:

1. Важно ли использовать стандартный код, когда это возможно, и почему?
2. У нас есть много запросов, использующих порядковые номера в предложении ORDER BY. Является ли это плохой практикой и, если так, почему?
3. Если в запросе нет предложения ORDER BY, в каком порядке будут возвращены записи?
4. Считаете ли вы правильным использование предложения DISTINCT в каждом запросе?

Рекомендуемые упражнения

Для успешного освоения экзаменационных тем, представленных в данной главе, выполните следующие задания.

Просмотр общедоступных новостных групп по тематике T-SQL и анализ кода

Для тренировки применения знаний об использовании языка T-SQL в реляционной манере вам необходимо проанализировать фрагменты кода, написанные другими.

- Задание 1.** Перечислите как можно больше примеров вариантов кодирования на языке T-SQL, которые не являются реляционными.
- Задание 2.** После того как вы создали список в задании 1, посетите общественный форум Microsoft по T-SQL по адресу <http://social.msdn.microsoft.com/>

[Forums/en/transactsql/threads](#). Проанализируйте фрагменты кода в обсуждениях по T-SQL. Попытайтесь определить случаи, где используются нереляционные элементы. Если такие найдутся, укажите, что нужно изменить, чтобы сделать их реляционными.

Описание логической обработки запросов

Чтобы лучше понимать логическую обработку запросов, выполните следующие задания.

- Задание 1.** Создайте документ, содержащий нумерованный список этапов логической обработки запросов в правильной последовательности. Дайте краткое описание того, что происходит на каждом шаге.
- Задание 2.** Создайте функциональную диаграмму, представляющую последовательность этапов логической обработки запросов с помощью Microsoft Visio, Microsoft PowerPoint или Microsoft Word.

ГЛАВА 2

Начало работы с инструкцией *SELECT*

Темы экзамена

- Работа с данными.
 - Запрос данных с помощью инструкций SELECT.
 - Реализация типов данных.
- Модификация данных.
 - Использование функций.

Предыдущая глава была посвящена знакомству с основными понятиями языка T-SQL. В данной главе мы сначала рассмотрим два основных предложения, используемых в запросах, — *FROM* и *SELECT*. Затем перейдем к типам данных, поддерживаемых Microsoft SQL Server, и обоснованию выбора подходящих типов данных для столбцов. Кроме того, мы поговорим об использовании встроенных скалярных функций, выражения CASE и вариаций, таких как ISNULL и COALESCE.

ПРЕЖДЕ ВСЕГО

Для полного освоения этой главы необходимо иметь:

- опыт работы с SQL Server Management Studio (SSMS);
- некоторый опыт в написании кода на языке T-SQL;
- доступ к экземпляру SQL Server 2012 с установленной учебной базой данных TSQL2012.

Занятие 1. Использование предложений *FROM* и *SELECT*

Предложения *FROM* и *SELECT* — две основные инструкции, появляющиеся почти в каждом запросе, целью которого является извлечение данных. В этом занятии

рассматривается назначение этих предложений, их использование и связанные с ними рекомендации.

Изучив материал этого занятия, вы сможете:

- ✓ Составлять запросы с использованием предложений FROM и SELECT
- ✓ Определять псевдонимы таблиц и столбцов
- ✓ Разбираться в оптимальных методах использования предложений FROM и SELECT

Продолжительность занятия — 30 минут.

Предложение *FROM*

В соответствии с логической обработкой запросов (см. подробное описание концепции в *главе 1*), предложение *FROM* — это первое предложение, которое должно оцениваться логически в запросе *SELECT*. У предложения *FROM* две главные роли:

- в этом предложении указываются таблицы, к которым будет направлен запрос;
- в этом предложении к входным таблицам могут быть применены табличные операторы, такие как соединения.

Данная глава посвящена первой роли. В *главах 4 и 5* рассматривается использование табличных операторов. Предполагая, что вы подключены к учебной базе данных TSQL2012, рассмотрим в качестве базового запроса следующий, использующий предложение *FROM* для того, чтобы указать, что таблица *HR.Employees* является запрашиваемой таблицей.

```
SELECT empid, firstname, lastname  
FROM HR.Employees;
```

Обратите внимание на использование для обращения к таблице имени, состоящего из двух частей. Первая часть (*HR*) — это название схемы, вторая часть (*Employees*) представляет собой имя таблицы. В некоторых случаях T-SQL поддерживает формат с опущенным именем схемы, например *FROM Employees*, при этом используется неявная схема процесса разрешения имен. Считается хорошей практикой всегда явно указывать имя схемы. В таком случае вам не грозит в итоге прийти к имени схемы, которую вы не планировали использовать, также можно избавиться от затрат, пусть даже небольших, сопровождающих процесс неявного разрешения имен.

В предложении *FROM* можно задавать запрашиваемым таблицам произвольные имена. Можно использовать для этого форму <таблица> <псевдоним> как в выражении *HR.Employees E* или <таблица> AS <псевдоним> как в выражении *HR.Employees AS E*. Последний формат лучше читается. Существует соглашение, в соответствии с которым при использовании псевдонимов следует употреблять короткие имена; как правило, это одна буква, которая каким-то образом обозначает запрашиваемую таблицу, например *E* для *Employees*.

Причины, по которым может возникнуть желание присваивать псевдонимы таблицам, раскрывается в *главе 4*. Пока вам достаточно знать, что язык T-SQL поддерживает такие псевдонимы таблиц и синтаксис для их присвоения.

Помните, что назначив таблице псевдоним, вы фактически переименовываете таблицу на время запроса. Первоначальное имя таблицы становится невидимым, доступен только ее псевдоним. Как правило, можно перед именем столбца, на который ссылаются, ставить имя таблицы, как в случае Employees.empid.

Однако если таблице Employees присвоен псевдоним E, ссылка Employees.empid является некорректной. Необходимо использовать выражение E.empid, как показано в следующем примере:

```
SELECT E.empid, firstname, lastname  
FROM HR.Employees AS E;
```

При попытке выполнения этого кода с использованием полного имени таблицы произойдет ошибка.

Как уже упоминалось ранее, в главе 4 подробно рассказано, почему необходимы псевдонимы таблиц.

Предложение **SELECT**

Предложение SELECT играет две главные роли в запросе:

- оценивает выражения, которые определяют выражения в результате запроса, присваивая им при необходимости псевдонимы;
- используя предложение DISTINCT, можно при необходимости избежать дублирования строк в результате запроса.

Начнем с первой роли. В качестве примера рассмотрим следующий запрос:

```
SELECT empid, firstname, lastname  
FROM HR.Employees;
```

Предложение FROM указывает, что входной таблицей в этом запросе является таблица HR.Employees. Предложение SELECT представляет только три атрибута из входного набора как возвращаемые атрибуты в результирующем наборе.

Язык T-SQL поддерживает использование звездочки (*) как альтернативы перечисления всех атрибутов из входных таблиц, но по ряду причин это считается плохим решением. Часто требуется возвратить только поднабор входных атрибутов, и использование звездочки может быть вызвано только ленью. Возвращая больше атрибутов, чем нужно, вы помешаете SQL Server использовать в отношении интересующего набора атрибутов то, что называют покрывающими индексами. Кроме того, через сеть отправляется больше данных, чем требуется, что вполне может оказаться негативное влияние на производительность системы. В дополнение к сказанному, определение базовой таблицы может измениться со временем, и даже если при первоначальном написании запроса * действительно представляла все требуемые атрибуты, позднее это может быть совсем не так. Учитывая эти и другие причины, мы рекомендуем всегда явно перечислять требуемые атрибуты.

В предложении SELECT можно присвоить собственные псевдонимы выражениям, которые определяют результирующие атрибуты. Существует несколько поддерживаемых форм создания псевдонимов: <выражение> AS <псевдоним> как в случае empid

AS employeeid, <выражение> <псевдоним> как в случае empid employeeid и <псевдоним> = <выражение> как в случае employeeid = empid.

РЕАЛЬНЫЙ МИР

Предпочтительный способ создания псевдонима

Мы считаем предпочтительным использование первой формы с предложением AS, поскольку она является стандартной и к тому же более читаемой. Вторая форма не только хуже читается, но и затрудняет нахождение ошибок в коде.

Рассмотрим следующий запрос:

```
SELECT empid, firstname lastname
FROM HR.Employees;
```

Разработчик, написавший запрос, предполагал, что будут возвращены атрибуты empid, firstname и lastname, но пропустил запятую между firstname и lastname. Запрос не выдал ошибку, но вернул следующий результат:

empid	lastname
1	Sara
2	Don
3	Judy
...	

Вопреки намерениям автора, SQL Server интерпретировал запрос как присвоение псевдонима lastname атрибуту firstname вместо того, чтобы возвратить оба. Если вы привыкли присваивать псевдонимы выражениям в форме с пробелом, локализовать такие ошибки вам будет сложно.

Вернемся к умышленному присвоению атрибутам псевдонимов и скажем, что оно используется в двух случаях. Один из них — переименование, когда необходимо, чтобы результирующий атрибут имел другое имя, нежели исходный атрибут. Например, если нужно дать результирующему атрибуту имя employeeid вместо empid, как показано в следующем примере:

```
SELECT empid AS employeeid, firstname, lastname
FROM HR.Employees;
```

Другой случай использования псевдонимов — назначить имя атрибуту, получившемуся из выражения, который в противном случае был бы безымянным. Например, предположим, что нужно сгенерировать результирующий атрибут из выражения путем конкатенации атрибута firstname, пробела и атрибута lastname. Для этого можно использовать следующий запрос:

```
SELECT empid, firstname + N' ' + lastname
FROM HR.Employees;
```

Результат будет нереляционным, т. к. результирующий атрибут не имеет имени.

empid	
1	Sara Davis
2	Don Funk
3	Judy Lew
...	

Присвоив выражению псевдоним, можно назначить имя результирующему атрибуту, сделав реляционным результат запроса, как в следующем примере:

```
SELECT empid, firstname + N' ' + lastname AS fullname
FROM HR.Employees;
```

Далее приведен сокращенный вариант результата этого запроса.

	fullname
1	Sara Davis
2	Don Funk
3	Judy Lew
...	

Напомним, в *главе 1* мы говорили о том, что если результат запроса может содержать дубликаты, T-SQL не будет их удалять без специального указания. Результат, содержащий дубликаты, считается нереляционным, поскольку наборы данных реляционного типа не должны иметь дубликатов. Поэтому, если дубликаты возможны в результате запроса и вы хотите их удалить, чтобы получить на выходе реляционный результат, этого можно добиться, добавив предложение DISTINCT, как показано в следующем примере:

```
SELECT DISTINCT country, region, city
FROM HR.Employees;
```

Таблица HR.Employees содержит девять строк и пять отдельных местоположений; следовательно, на выходе этого запроса имеется пять строк.

country	region	city
UK	NULL	London
USA	WA	Kirkland
USA	WA	Redmond
USA	WA	Seattle
USA	WA	Tacoma

Существует интересное различие между стандартным языком SQL и T-SQL с точки зрения минимальных требований запроса SELECT. В соответствии со стандартным языком SQL, запрос должен иметь как минимум предложения FROM и SELECT. Напротив, T-SQL поддерживает запрос SELECT, содержащий только предложение SELECT без предложения FROM. Можно себе представить такой запрос, как если бы он был направлен к воображаемой таблице, состоящей всего из одной строки. Например, приведенный далее запрос является неправильным с точки зрения стандартного SQL, но вполне корректен для T-SQL.

```
SELECT 10 AS col1, 'ABC' AS col2;
```

На выходе этого запроса будет одна строка с атрибутами, полученными из выражений имен, присвоенных с помощью псевдонимов.

col1	col2
10	ABC

Разделение идентификаторов

При ссылке на идентификаторы атрибутов, схем, таблиц и других объектов возможны случаи, когда требуется использовать разделители, и случаи, когда использование разделителей является необязательным.

T-SQL поддерживает обе формы разделения идентификаторов: стандартную, с использованием двойных кавычек, как в выражении "Sales"."Orders", а также частную форму с использованием квадратных скобок, как в выражении [Sales].[Orders].

Если идентификатор "регулярный", использование разделителей необязательно. В случае регулярного идентификатора он соответствует правилам форматирования идентификаторов. Эти правила говорят о том, что первым символом должна быть буква, определяемая стандартом Unicode 3.2 (a, ..., z, A, ..., Z и символы из других языков в кодировке Unicode), подчеркивание (_), значок "at" (@) или знак номера (#). Последующие символы могут включать буквы, арабские цифры, символ @, знак доллара (\$), знак номера или подчеркивание. Идентификатор не может быть зарезервированным ключевым словом языка T-SQL, не может содержать встроенные пробелы и не должен включать дополнительные символы.

Идентификатор, не соответствующий данным правилам, должен быть отделен разделителями. Например, атрибут 2006 считается нерегулярным идентификатором, потому что начинается с цифры, и значит, должен иметь разделители: "2006" или [2006]. На регулярный идентификатор, такой как y2006, можно ссылаться без разделителей, просто указав y2006, или при желании можно использовать разделители. Возможно, вы предпочтете не использовать разделители в регулярных идентификаторах, поскольку разделители засоряют код.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Назовите формы присвоения псевдонимов атрибутам в T-SQL.
2. Что такое нерегулярный идентификатор?

Ответы на контрольные вопросы

1. Формы присвоения псевдонимов: <выражение> AS <псевдоним>, <выражение> <псевдоним> и <псевдоним> = <выражение>.
2. Идентификатор, не соответствующий правилам форматирования идентификаторов, например, начинаящийся с цифры, включающий в себя пробел или являющийся зарезервированным символом T-SQL.

ПРАКТИКУМ Использование предложений FROM и SELECT

В этом практикуме вы проверите ваши знания об использовании предложений FROM и SELECT.

Задание 1. Составление простого запроса и использование псевдонимов таблиц

В этом задании вы попробуете свои силы в использовании предложений `FROM` и `SELECT`, включая применение псевдонимов таблиц.

1. Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
2. Чтобы попрактиковаться в написании простого запроса, использующего предложения `FROM` и `SELECT`, введите следующий запрос и выполните его.

```
USE TSQL2012;
SELECT shipperid, companyname, phone
FROM Sales.Shippers;
```

Инструкция `USE` гарантирует, что подключена целевая база данных TSQL2012.

Предложение `FROM` указывает, что запрашиваемой является таблица `Sales.Shippers`, а предложение `SELECT` выбирает атрибуты `shipperid`, `companyname` и `phone` из этой таблицы. Далее приведен результат этого запроса.

shipperid	companyname	phone
1	Shipper GVSUA	(503) 555-0137
2	Shipper ETYNR	(425) 555-0136
3	Shipper ZHISN	(415) 555-0138

3. Если в запросе участвуют две и более таблицы, и в другой таблице имеется атрибут `shipperid`, необходимо присвоить атрибуту `shipperid` префикс — имя таблицы, в результате чего получится `Shippers.shipperid`. Для краткости, можно использовать в качестве псевдонима более короткое имя, например `S`, и затем обращаться к этому атрибуту как `S.shipperid`. Далее приведен пример присвоения таблице псевдонима и использования нового имени таблицы в качестве префикса атрибута.

```
SELECT S.shipperid, companyname, phone
FROM Sales.Shippers AS S;
```

Задание 2. Использование псевдонимов столбцов и идентификаторов с разделителями

В этом задании вы попрактикуетесь в использовании псевдонимов столбцов, включая применение идентификаторов с разделителями.

В качестве отправной точки используйте запрос из шага 3 предыдущего задания.

1. Предположим, необходимо переименовать результирующий атрибут `phone` в `phone number`. Далее приведен пример присвоения псевдонима атрибуту с идентификатором `phone number` без использования разделителей.

```
SELECT S.shipperid, companyname, phone AS phone number
FROM Sales.Shippers AS S;
```

2. Этот код дает ошибку, поскольку номер телефона не является регулярным идентификатором, и поэтому должны использоваться разделители, как показано в следующем примере:

```
SELECT S.shipperid, companynname, phone AS [phone number]
FROM Sales.Shippers AS S;
```

3. Помните, что T-SQL поддерживает как собственный способ разделения идентификаторов с помощью квадратных скобок, так и стандартную форму, использующую двойные кавычки, например "phone number".

Резюме занятия

- Предложение `FROM` — это первое предложение, которое логически обрабатывается в запросе `SELECT`. В этом предложении указываются таблицы, к которым адресован запрос, и табличные операторы. В предложении `FROM` можно присваивать таблицам псевдонимы с именами по своему выбору и затем использовать псевдоним таблицы в качестве префикса к именам атрибутов.
- С помощью предложения `SELECT` можно указать выражения, определяющие результирующие атрибуты. Можно присваивать результирующим атрибутам собственные псевдонимы и таким образом получать реляционный результат. Если в результате запроса возможно получение дубликатов, их следует удалить с помощью предложения `DISTINCT`.
- При использовании регулярных идентификаторов в качестве имен объектов или атрибутов применение разделителей является необязательным. Если используются нерегулярные идентификаторы, разделители обязательны.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. В чем заключается важность назначения псевдонимов атрибутам в T-SQL? (Выберите все подходящие варианты.)
 - A. Возможность назначать атрибутам псевдонимы является всего лишь эстетическим свойством.
 - B. Выражение, полученное в результате вычисления, не имеет имени атрибута, если его не присвоить с использованием псевдонима, и это нереляционный вариант.
 - C. Язык T-SQL требует, чтобы все результирующие атрибуты запроса имели имена.
 - D. С помощью псевдонимов атрибута можно назначить результирующему атрибуту новое имя, если нужно, чтобы оно отличалось от начального имени атрибута.

2. Какие предложения, согласно T-SQL, являются обязательными в запросе SELECT?
 - A. Предложения FROM и SELECT.
 - B. Предложения SELECT и WHERE.
 - C. Предложение SELECT.
 - D. Предложения FROM и WHERE.
3. Какие из следующих методов считаются неправильными? (Выберите все подходящие варианты.)
 - A. Присвоение псевдонимов столбцам с помощью предложения AS.
 - B. Присвоение псевдонимов таблицам с помощью предложения AS.
 - C. Не присваивать псевдоним столбцу, если этот столбец является результатом вычисления.
 - D. Использование знака * в инструкции SELECT.

Занятие 2. Работа с типами данных истроенными функциями

При определении столбцов в процедурах, параметрах процедур и функций, а также переменных в T-SQL необходимо выбрать для них тип данных. Выбранный тип данных ограничивает поддерживаемые данные в дополнение к инкапсуляции воздействующего на данные поведения, представляя его посредством операторов и других средств. Поскольку типы данных — наиважнейший компонент данных, т. к. на них строится все, выбор того или иного типа данных имеет исключительно важное значение для приложения на различных уровнях. Поэтому эта область не должна рассматриваться как незначительная, напротив, она должна быть предметом пристального внимания и заботы. Это одна из причин, по которой эта тема рассматривается так рано в данном учебном курсе, хотя несколько его первых глав посвящены созданию запросов, и только более поздние главы связаны с определением данных, подобно созданию и изменению таблиц. Знание типов данных имеет важное значение как для определения данных, так и для манипулирования ими.

T-SQL поддерживает множество встроенных функций, которые могут использоваться для манипулирования данными. Поскольку функции выполняют операции на входных значениях и возвращают выходные значения, понимание встроенных функций идет рука об руку с пониманием типов данных.

Следует заметить, что эта глава не содержит исчерпывающей информации обо всех типах данных и всех функциях, поддерживаемых языком T-SQL — на это понадобились бы целая книга. В этой главе дается объяснение факторов, которые следует принимать во внимание при выборе типа данных, и ключевые понятия использования функций, как правило, в контексте определенного типа данных, например, данных типа "дата и время" или символьные данные. Подробную информацию и технические подробности о типах данных можно найти в электронной документации по SQL Server 2012 в разделе "Типы данных (Transact-SQL)" по адресу

[http://msdn.microsoft.com/ru-ru/library/ms187752\(v=SQL.110\).aspx](http://msdn.microsoft.com/ru-ru/library/ms187752(v=SQL.110).aspx). Для получения дополнительных сведений о встроенных функциях см. раздел "Встроенные функции (Transact-SQL)" по адресу [http://msdn.microsoft.com/ru-ru/library/ms174318\(v=SQL.110\).aspx](http://msdn.microsoft.com/ru-ru/library/ms174318(v=SQL.110).aspx).

Изучив материал этого занятия, вы сможете:

- ✓ Выбирать нужный тип данных
- ✓ Выбирать тип данных для ключей
- ✓ Работать с датой и временем, а также с символьными данными
- ✓ Работать с выражением CASE и связанными с ним функциями

Продолжительность занятия — 50 минут.

Выбор нужного типа данных

Выбор подходящих типов данных для атрибутов — возможно, одно из важнейших решений в отношении данных, которое вам предстоит принимать. SQL Server поддерживает множество типов данных из разных категорий: точные числовые типы данных (INT, NUMERIC), символьные строки (CHAR, VARCHAR), символьные строки в кодировке Unicode (NCHAR, NVARCHAR), приблизительные числовые типы данных (FLOAT, REAL), двоичные строки (BINARY, VARBINARY), дата и время (DATE, TIME, DATETIME2, SMALLDATETIME, DATETIME, DATETIMEOFFSET) и др. Из-за огромного числа возможностей эта задача может показаться сложной, но при условии следования определенным принципам можно точно и мастерски выбирать тип данных, что позволит иметь надежную, согласованную и эффективную базу данных.

Одна из самых сильных сторон реляционной модели — это значение, которое она придает целостности данных, как части самой модели, на многих уровнях. Одним из важных моментов при выборе нужного типа данных является необходимость помнить, что тип данных — это ограничение. То есть он имеет определенную область поддерживаемых величин и не допустит присутствия величин вне этой области. Например, тип данных DATE разрешает использование только допустимых дат. Попытка ввести данные, не являющиеся датой, как например 'abc' или '20120230', будет отвергнута. Если вы используете атрибут, который должен представлять дату, скажем, день рождения, и при этом возьмете тип данных INT или CHAR, вы ничего не выиграете от встроенной проверки допустимости дат. Тип INT не помешает использованию такой величины, как 99999999, а тип CHAR не помешает использованию такого значения, как '20120230'.

Подобно тому, что тип данных является ограничением, значение NOT NULL — также ограничение. Если предполагается, что атрибут не должен допускать значения NULL, важно использовать ограничение NOT NULL как часть определения этого атрибута. В противном случае значения NULL смогут проникнуть в ваш атрибут.

Кроме того, необходимо быть уверенным, что вы не путаете форматирование значения и его тип данных. Иногда, когда есть желание хранить данные в определенном формате, символьная строка используется для хранения дат. За форматирова-

ние значения отвечает приложение, в котором происходит представление данных. Вас не должен интересовать внутренний формат хранения данных, потому что тип данных является свойством значения, хранящегося в базе данных. Это имеет отношение к важному принципу реляционной модели, называемому *независимостью от физического представления данных*.



Тип данных формирует поведение. Используя неподходящий тип данных, вы терьете все свойственное этому типу данных поведение, инкапсулированное в форме поддерживающих его операторов и функций. В качестве простого примера можно взять оператор `+`, который для числовых типов данных представляет сложение, а для символьных строк — объединение (конкатенацию). Если выбран неподходящий тип данных для какого-то значения, иногда приходится конвертировать этот тип (явно или неявно), а иногда и немного искажать значение, чтобы можно было обрабатывать его так, как предполагалось.

Другой важный аспект при выборе подходящего типа данных — это размер. Часто одним из важнейших факторов, влияющих на производительность запроса, является количество задействованного ввода-вывода. Запрос, который читает меньше данных, как правило, работает быстрее. Чем больше используемый тип данных, тем больше памяти он занимает. В наше время таблицы, содержащие миллионы, если не миллиарды строк, довольно обычны.

Если умножить размер типа данных на количество строк в таблице, получаются весьма значительные числа. В качестве примера предположим, что у нас есть атрибут, представляющий экзаменационные баллы, которые являются целыми числами в диапазоне от 0 до 100. Использование для этого типа данных `INT` является излишеством. Он отводит 4 байта на значение, тогда как тип `TINYINT` использует только 1 байт и поэтому является более подходящим типом в данном случае. Аналогично для данных, представляющих даты, как правило, применяется тип `DATETIME`, который использует 8 байт памяти.

Если предполагается использовать дату без времени, следует применять тип `DATE`, который занимает только 3 байта. Если же значение будет представлять и дату, и время, нужно использовать `DATETIME2` или `SMALLDATETIME`. Первый требует от 6 до 8 байт памяти (в зависимости от точности) и дополнительно обеспечивает более широкий диапазон дат и повышенную, контролируемую точность. Последний использует только 4 байта, поэтому если поддерживаемый им диапазон дат и точность отвечают вашим потребностям, следует использовать этот тип данных. Говоря коротко, нужно использовать наименьший тип данных, удовлетворяющий вашим запросам. Хотя, конечно, следует думать не о сиюминутной ситуации, а о перспективе. Например, не стоит использовать тип `INT` для ключа таблицы, которая однажды вырастет до миллиардов строк. В таком случае надо остановиться на типе `BIGINT`.

Неправильным решением будет использование типа `INT` для атрибута, представляющего экзаменационные баллы, или типа `DATETIME` для значений даты и времени, которые требуют точности до минут, даже если думать о длительной перспективе.

Следует соблюдать осторожность в отношении неточных типов данных, таких как `FLOAT` и `REAL`. Два первых предложения в документации, описывающей эти типы

данных, дадут вам полное представление об их сущности: типы данных для работы с приближенными числами при использовании с числовыми данными с плавающей точкой. Данные с плавающей точкой являются приближенными; поэтому не все значения в диапазоне такого типа данных могут быть представлены точно. (Эти сведения можно найти в электронной документации для SQL Server 2012 в статье "Типы данных float and real Transact-SQL" по адресу <http://msdn.microsoft.com/ru-ru/library/ms173773.aspx>). Преимущество этих типов состоит в том, что они могут представлять очень большие и очень маленькие числа, помимо других числовых типов данных, которые поддерживает и может представлять SQL Server. Поэтому, если в научных целях представить очень большие или очень маленькие числа и не нужна очень большая точность, эти типы данных могут быть вполне подходящими. Они также достаточно экономичны (4 байта требуется под тип REAL и 8 байт под тип FLOAT). Но использовать их тогда, когда требуется большая точность, не нужно.

РЕАЛЬНЫЙ МИР

Проблемы с типом FLOAT

Мы помним случай, когда клиент использовал тип данных FLOAT для представления номеров штрихкода продуктов и был очень удивлен, когда при сканировании он получал не тот продукт. Также недавно мы получили запрос о конвертации величины с типом FLOAT в тип NUMERIC, дающий на выходе не ту величину, которая была введена. Вот этот запрос.

```
DECLARE @f AS FLOAT = '29545428.022495';
SELECT CAST(@f AS NUMERIC(28, 14)) AS value;
```

Можете догадаться, что получилось на выходе этого кода? Посмотрите.

```
Value
-----
29545428.02249500200000
```

Как мы уже говорили, некоторые величины не могут быть представлены точно.

Таким образом, необходимо быть уверенным в том, что используются нужные числовые типы данных, когда требуется представлять значения точно, и прибегать к использованию приблизительных числовых типов данных только в случаях, когда вы уверены, что это приемлемо для вашего приложения.

Еще один вопрос, который следует рассмотреть при выборе типа данных, — это выбор фиксированных типов (CHAR, NCHAR, BINARY) или динамических типов (VARCHAR, NVARCHAR, VARBINARY). Фиксированные типы используют память для указанного размера данных; например, тип CHAR(30) использует память на 30 символов, независимо от того, действительно вы используете 30 символов или меньше. Это означает, что обновления данных не потребуют физического расширения строки, и таким образом сдвиг данных не нужен. Итак, для атрибутов, которые часто обновляются, и где важное значение имеет производительность обновлений, необходимо использовать фиксированные типы данных. Обратите внимание, при использовании сжатия — особенно сжатия строк — SQL Server хранит фиксированные типы данных как переменные типы, но с меньшими накладными затратами.

Типы переменных используют память, необходимую на вводимые данные, плюс еще пару байтов на смещение (или 4 бита со сжатием строки). Поэтому при боль-

шом разнообразии размеров строк, если вы используете переменные типы, то можете сэкономить довольно много памяти. Как мы уже говорили, чем меньше используется памяти, тем меньше данных будет читать запрос, и тем быстрее он будет работать. Поэтому типы данных переменной длины, как правило, предпочтительны в случаях, когда производительность чтения имеет большое значение.

При использовании символьных строк возникает вопрос о выборе обычных типов символьных данных (`CHAR`, `VARCHAR`) или типов в кодировке Unicode (`NCHAR`, `NVARCHAR`). Первый тип данных использует 1 байт памяти на символ и поддерживает только один язык (на основании параметров сортировки), кроме английского. Последний тип данных использует 2 байта памяти на символ (без сжатия) и поддерживает несколько языков. Если будет необходима суррогатная пара, символ потребует 4 байта памяти. Поэтому когда используются данные на нескольких языках, и вам в ваших данных нужен только один язык, кроме английского, выгоднее использовать обычные символьные типы данных с более низкими требованиями к памяти. В случае многоязычных данных или если приложение по умолчанию работает с данными в кодировке Unicode, следует использовать типы данных в кодировке Unicode, чтобы избежать потери информации. Требования большего количества памяти для хранения данных в кодировке Unicode уменьшились, начиная SQL Server 2008 R2 с введением сжатия данных в Unicode.

При использовании типов данных, которые могут иметь соответствующую им длину, таких как `CHAR` и `VARCHAR`, T-SQL позволяет опустить длину, и в таком случае используется длина по умолчанию. Но значения по умолчанию могут быть различными в различных ситуациях. Считается наиболее правильным всегда явно указывать длину, как, например, в выражениях `CHAR(1)` или `VARCHAR(30)`.

При определении атрибутов, представляющих одни и те же данные в нескольких таблицах — особенно такие, которые затем могут быть использованы в качестве соединения столбцов (например, первичный ключ в одной таблице и внешний ключ в другой) — очень важно соблюдать согласованность типов данных. В противном случае при сравнении одного атрибута с другим SQL Server будет вынужден выполнить явную конвертацию типа одного атрибута в другой, что способно вызвать нежелательные последствия для производительности, такие как невозможность эффективного использования индексов.

Также необходимо быть уверенным, что при указании литерала какого-либо типа вы используете правильную форму. Например, литералы для обычных символьных строк отделяются с помощью одиночных кавычек, как в выражении `'abc'`, тогда как литералы символьных строк в кодировке Unicode разделяются с помощью заглавной буквы `N` и затем одиночных кавычек, как в случае `N'abc'`. Если в выражении присутствуют элементы с разными типами данных, SQL Server должен выполнить неявное преобразование, когда это возможно, а это может привести к снижению производительности. Обратите внимание, в некоторых случаях интерпретация литерала может не соответствовать тому, что вы можете представить себе интуитивно. Чтобы литерал был нужного вам типа, может потребоваться применить явное преобразование с помощью таких функций, как `CAST`, `CONVERT`, `PARSE` или

TRY_CAST, TRY_CONVERT и TRY_PARSE. Возьмем в качестве примера литерал 1, который рассматривается SQL Server как INT в любом контексте. Если нужно, чтобы литерал 1 рассматривался, скажем, как BIT, необходимо конвертировать тип этого литерала явно, как, например, в выражении CAST(1 AS BIT). Аналогично, литерал 4000000000 рассматривается как имеющий тип NUMERIC, а не BIGINT. В случае необходимости иметь для этого литерала последний тип данных, следует использовать преобразование CAST(4000000000 AS BIGINT). Разница между функциями без части TRY и их аналогами с этой частью состоит в том, что функции без TRY завершаются ошибкой, если значение невозможно преобразовать, тогда как функции с TRY возвращают в таком случае NULL. Например, следующий код завершится ошибкой:

```
SELECT CAST('abc' AS INT);
```

В свою очередь, этот код возвратит значение NULL:

```
SELECT TRY_CAST('abc' AS INT);
```

Что касается разницы между функциями CAST, CONVERT и PARSE, для функции CAST нужно указывать выражение и целевой тип; для функции CONVERT существует третий аргумент, представляющий стиль преобразования данных, который поддерживается для некоторых преобразований, таких как символьные строки и значения даты и времени. Например, выражение CONVERT(DATE, '1/2/2012', 101) преобразует литерал типа "символьная строка" в тип данных DATE с помощью стиля 101, представляющего стандарт США. Для функции PARSE можно указать культуру (язык и региональные параметры), используя любую культуру, поддерживаемую Microsoft .NET Framework. Например, выражение PARSE('1/2/2012' AS DATE USING 'en-US') анализирует входной литерал как DATE, используя культуру United States English.

При работе с выражениями, в состав которых входят операнды разных типов, SQL Server обычно преобразует тип данных с более низким приоритетом к типу данных с более высоким приоритетом. Рассмотрим в качестве примера выражение 1+'1'. Один операнд имеет тип данных INT, а другой — тип VARCHAR. Если вы заглянете в электронную документацию для SQL Server 2012 в раздел "Приоритет типов данных (Transact-SQL)" на страницу <http://msdn.microsoft.com/ru-ru/library/ms190309.aspx>, то найдете там, что тип данных INT имеет более высокий приоритет, чем VARCHAR; следовательно, SQL Server неявно преобразует значение '1' с типом VARCHAR в значение 1 с типом INT, и результатом выражения будет 2, а не строковое значение '11'. Разумеется, всегда можно взять инициативу в свои руки и использовать явное преобразование.

Если все операнды в выражении имеют один и тот же тип, этот же тип будет иметь и результат, что может оказаться нежелательным. Например, результатом выражения 5/2 в T-SQL будет целочисленное значение 2 типа INT, а не числовое значение 2.5 типа NUMERIC, поскольку оба операнда имеют целочисленный тип данных. Если бы у вас было два целочисленных столбца, например col1/col2, и вы хотели бы, чтобы частное от деления было числовым (тип NUMERIC), вам пришлось бы преобразовать эти столбцы явно, как в выражении CAST(col1 AS NUMERIC(12, 2)) /CAST(col2 AS NUMERIC(12, 2)).

Выбор типов данных для ключей

При определении интеллектуальных ("разумных") ключей в таблицах — а именно ключей на основе уже существующих атрибутов, полученных из приложения, — их типы не вызывают сомнений, т. к. вы эти типы уже определили для ваших атрибутов. Но если необходимо создать суррогатные ключи, которые добавляются с одной только целью — быть использованными в качестве ключей, — нужно определить подходящий тип данных для атрибута в дополнение к механизму генерации значения ключа. Практика показывает, что вы услышите много различных мнений о том, что является лучшим решением — некоторые основаны на теории, другие подкреплены эмпирическими доказательствами. Но для разных систем и разных нагрузок оптимальными могут быть различные решения. Более того, в некоторых системах скорость записи данных может быть приоритетной, тогда как в других приоритетом является скорость чтения данных. Одно решение может сделать более быстрой вставку данных, но более медленным их чтение; другое решение может дать совсем иной результат. В конечном счете, чтобы сделать разумный выбор, важно хорошо изучить теорию, опыт других людей, а также не забывать про тесты производительности на целевой системе. В связи с этим, очень подходящей кажется фраза из книги "Bubishi" Патрика Маккарти (Patrick McCarthy). Она гласит: "Мудрость заставляет работать знания".

Обратите внимание, в данном разделе рассматриваются такие понятия, как объекты последовательности, свойства идентификаторов столбцов и индексы, которые детально обсуждаются далее в этом учебном курсе. В главе 11 рассматриваются объекты последовательности и свойство `IDENTITY`, а в главе 15 — индексы. Вы можете вернуться к данному разделу после прочтения этих глав. Для генерации суррогатных ключей, как правило, используются следующие элементы.

- **Свойство столбца идентификаторов.** Свойство, которое автоматически генерирует ключи в атрибуте числового типа с масштабом 0; т. е. любого целочисленного типа (`TINYINT`, `SMALLINT`, `INT`, `BIGINT`) или типа `NUMERIC/DECIMAL` с масштабом 0.
- **Объект последовательности.** Независимый объект в базе данных, из которого можно получить новые объекты последовательности. Так же как и свойство столбца идентификаторов, он поддерживает любой числовой тип данных с масштабом 0. В отличие от него, он не привязан к конкретному столбцу; напротив, как уже было сказано, это независимый объект в базе данных. Также можно запросить новое значение объекта последовательности перед его использованием. Прочие его преимущества будут рассмотрены в главе 11.
- **Непоследовательные GUID.** Можно генерировать непоследовательные глобальные уникальные идентификаторы для их сохранения в атрибуте типа `UNIQUEIDENTIFIER`. Для генерации нового GUID вы можете использовать функцию T-SQL `NEWID`, вызывая его, например, с выражением по умолчанию, прикрепленным к столбцу. Его также можно генерировать где угодно — например, на клиенте, — с помощью программного интерфейса (API), который генерирует новый GUID. Уникальность идентификаторов GUID гарантирована в пространстве и времени.

- **Последовательные GUID.** Можно генерировать последовательные идентификаторы GUID с помощью функции T-SQL NEWSEQUENTIALID.
- **Пользовательские решения.** Если вы не хотите использовать для генерации ключей встроенные инструменты, предлагаемые SQL Server, следует разработать собственное пользовательское решение. В таком случае тип данных для ключа зависит от пользовательского решения.

COBET**Подготовка к экзамену**

Для успешной сдачи экзамена важно понимание встроенных инструментов, предлагаемых языком T-SQL для генерации суррогатных ключей, таких как объект последовательности, свойство столбца идентификаторов и функции NEWID и NEWSEQUENTIALID, а также их влияния на производительность.

При выборе генератора суррогатного ключа и типа данных для него немаловажное значение имеет размер типа данных. Чем он больше, тем больше требуется памяти и, следовательно, медленнее происходит чтение данных. Если выбирается тип данных INT, требуется 4 байта на значение, тип BIGINT требует 8 байт, тип UNIQUEIDENTIFIER — 16 байт и т. д. Требования к памяти для суррогатного ключа могут возрастать лавинообразно, если столбцы индексного ключа кластеризованного индекса используются некластеризованными индексами для внутренних целей как средство для нахождения строк в таблице. Так что если вы определите кластеризованный индекс для столбца x и некластеризованные индексы — один для столбца a, один для b и один для c, — некластеризованные индексы будут созданы на столбцах (a, x), (b, x) и (c, x) соответственно.

Другими словами, эффект умножается. Что касается выбора между последовательными ключами (с использованием идентификатора, последовательности или функции NEWSEQUENTIALID) и непоследовательными ключами (с помощью функции NEWID или пользовательского генератора случайных ключей), надо иметь в виду несколько важных моментов.

Начнем с последовательных ключей, где все строки добавляются в конец индекса. Когда страница заполнена, SQL Server выделяет новую страницу и заполняет ее. Это приводит к уменьшению фрагментации диска, что полезно для производительности чтения. Кроме того, вставки могут быть быстрее, когда одна сессия загружает данные и эти данные находятся на одном диске или на небольшом количестве дисков. Однако ситуация меняется, если речь идет о высокопроизводительных подсистемах хранения данных, имеющих множество дисководов. Загрузка данных из разных сеансов приведет к конфликту кратковременной блокировки страниц (кратковременные блокировки (latches)) — это объекты, используемые для синхронизации доступа к страницам базы данных) с самыми правыми страницами индексного связного списка уровня листьев. Этот ограничивающий фактор не позволяет использовать полную пропускную способность подсистемы хранения данных.

Обратите внимание, если вы решили остановиться на последовательных ключах и к тому же числового типа, можно всегда начинать с наименьших значений выбранного типа данных, чтобы использовать весь диапазон. Например, для типа INT нужно начинать не с 1, а с числа –2 147 483 648.



Рассмотрим непоследовательные ключи, как, например, случайные ключи, генерируемые с помощью функции `NEWID` или пользовательского решения. При попытке принудительно добавить строку в уже заполненную страницу SQL Server выполняет классическое разбиение страниц — он выделяет новую страницу и перемещает половину строк с исходной страницы на новую. Разбиение страниц — процесс затратный, вдобавок он приводит к фрагментации индекса. Фрагментация индекса может иметь негативное влияние на производительность операций чтения. Но с точки зрения производительности операции вставки, если подсистема хранения состоит из множества дисководов, и вы загружаете данные из нескольких сеансов, произвольный порядок может оказаться в действительности лучше, чем последовательный, несмотря на разбиения. Причина в том, что в правом конце индекса нет "горячих" точек, и пропускная способность подсистемы хранения данных используется лучше. Хороший пример, демонстрирующий эту стратегию, можно найти в блоге Томаса Кейзера (Thomas Kejser) по адресу <http://blog.kejser.org/2011/10/05/boosting-insert-speed-by-generating-scalable-keys/>.

Обратите внимание, последствия разбиения страниц и фрагментации индекса можно смягчить с помощью периодической перестройки индексов в процессе регулярного обслуживания системы, если на это, конечно, есть время.

Если, приняв во внимание вышеупомянутые причины, вы решите использовать ключи, генерированные в случайном порядке, вам все равно придется сделать выбор между индексами GUID и пользовательским генератором случайных ключей. Как уже упоминалось, идентификаторы GUID хранятся в типе данных `UNIQUEIDENTIFIER`, имеющем размер 16 байт — а это много. Но одно из главных преимуществ идентификаторов GUID заключается в том, что они могут генерироваться где угодно и не конфликтуют в пространстве и времени. Можно генерировать идентификаторы GUID не только на SQL Server с помощью функции `NEWID`, но и в любом другом месте, используя программные интерфейсы API. Вы также можете предложить собственное решение, которое генерирует меньшие случайные ключи. Выбранное решение может быть смесью встроенного средства и некоторых собственных доработок. Например, на странице <http://dangerousdba.blogspot.com/2011/10/day-sequences-saved-world.html> можно найти очень интересное решение, предложенное Вольфгангом 'Риком' Кучерой (Wolfgang 'Rick' Kutschera). Рик использует объект последовательности SQL Server, но инвертирует биты значений так, что вставка распределяется на конечной странице индекса.

В заключение этого раздела, посвященного ключам и их типам данных, напомним, что их много. Чем они меньше, тем, как правило, лучше, но следует принимать во внимание аппаратное обеспечение, которые вы используете, а также приоритеты производительности. Не следует забывать, что хотя очень важно тщательно провести предварительную оценку решения, его тестирование в целевой среде также играет большую роль.

Функции даты и времени

Язык T-SQL поддерживает ряд функций даты и времени, используемых для манипулирования датами и временем в данных. Поддержка функций даты и времени

продолжает совершенствоваться, и в двух последних версиях SQL Server добавлено несколько новых функций.

В данном разделе рассматриваются некоторые важные функции, поддерживаемые языком T-SQL, и представлены некоторые примеры. Полный список функций, а также технические детали и элементы синтаксиса приведены в электронной документации для SQL Server 2012 в разделе "Типы данных и функции даты и времени (Transact-SQL)" по адресу [http://msdn.microsoft.com/ru-ru/library/ms186724\(v=SQL.110\).aspx](http://msdn.microsoft.com/ru-ru/library/ms186724(v=SQL.110).aspx).

Текущая data и время

Одна из важных категорий функций — категория, возвращающая текущую дату и время. К этой категории относятся функции GETDATE, CURRENT_TIMESTAMP, GETUTCDATE, SYSDATETIME, SYSUTCDATETIME и SYSDATETIMEOFFSET.

Функция GETDATE — это собственная функция языка T-SQL, возвращающая текущую дату и время на экземпляре SQL Server, к которому вы подключены, с типом данных DATETIME. Функция CURRENT_TIMESTAMP делает то же самое, но она является стандартной, и, следовательно, ее рекомендуется использовать. Функции SYSDATETIME и SYSDATETIMEOFFSET подобны, но они возвращают значения с более точными типами данных — DATETIME2 и DATETIMEOFFSET (включая смещение) соответственно. Обратите внимание, не существует встроенных функций для возвращения текущей даты или времени; для получения такой информации надо просто привести функцию SYSDATETIME к DATE или TIME соответственно. Например, для получения текущей даты можно использовать выражение CAST(SYSDATETIME() AS DATE). Функция GETUTCDATE возвращает текущую дату и время в терминах UTC как тип данных DATETIME; функция SYSUTCDATETIME делает то же самое, но только возвращает результат с более точным типом данных DATETIME2.

Составляющие даты и времени

В этом разделе рассматриваются функции даты и времени, которые либо извлекают часть из значения даты и времени (как, например, DATEPART), либо строят значение даты и времени из части (как, например, DATEFROMPARTS).

С помощью функции DATEPART можно извлечь желаемую часть, такую как год, минуты или наносекунды из входного значения даты и времени, и возвратить эту часть в виде целочисленного значения. Например, выражение DATEPART(month, '20120212') возвращает 2. Язык T-SQL предоставляет функции YEAR, MONTH и DAY как сокращения функции DATEPART, не требуя указывать нужную часть. Функция DATENAME подобна функции DATEPART, но она возвращает, в противоположность целочисленному значению, имя части выражения в виде символьной строки. Обратите внимание, эта функция зависит от языка. Это означает, что если действующий язык сессии us_english, выражение DATENAME(month, '20120212') возвращает 'February', а для итальянского языка она возвратит 'febbraio'.

Язык T-SQL содержит ряд функций, которые строят желаемое выражение даты и времени из его числовых частей. Такая функция имеется для каждого из

шести возможных типов даты и времени: DATEFROMPARTS, DATETIME2FROMPARTS, DATETIMEFROMPARTS, DATETIMEOFFSETFROMPARTS, SMALLDATETIMEFROMPARTS и TIMEFROMPARTS. Например, чтобы построить значение DATE из его частей, нужно использовать такое выражение, как DATEFROMPARTS(2012, 02, 12).

Наконец, функция EOMONTH вычисляет соответствующую дату конца месяца для входной величины даты и времени. Например, предположим, что сегодня 12 февраля 2012 г. Выражение EOMONTH(SYSDATETIME()) тогда возвращает дату '2012-02-29'. Эта функция поддерживает необязательный входной параметр, указывающий, сколько месяцев следует добавить к результату.

Функции добавления и вычитания даты

Язык T-SQL поддерживает функции сложения и вычитания даты и времени, которые называются DATEADD и DATEDIFF.

Функция DATEADD используется очень часто. С ее помощью можно прибавить требуемое количество единиц указанной части к указанной величине даты и времени. Например, выражение DATEADD(year, 1, '20120212') прибавляет один год к входной дате 12 февраля 2012 г.

Функция DATEDIFF — еще одна часто используемая функция; она возвращает разность между двумя значениями даты и времени в единицах запрашиваемой части. Например, выражение DATEDIFF(day, '20110212', '20120212') вычисляет разность в днях между 12 февраля 2011 г. и 12 февраля 2012 г. и возвращает значение 365. Обратите внимание, эта функция рассматривает только запрашиваемую и более высокого уровня часть в иерархии даты и времени¹, но не более низкого. Например, выражение DATEDIFF(year, '20111231', '20120101') рассматривает только часть года и поэтому возвращает значение 1. Месяц и день во входном выражении ее не интересуют.

Смещение

Язык T-SQL поддерживает две функции, работающие со значениями даты и времени со смещением: SWITCHOFFSET и TODATETIMEOFFSET.

С помощью функции SWITCHOFFSET можно возвратить входное значение DATETIMEOFFSET с требуемым смещением. Например, рассмотрим выражение SWITCHOFFSET(SYSDATETIMEOFFSET(), '-08:00'). Независимо от смещения экземпляра системы, к которому вы подключены, вы хотите получить значение текущей даты и времени со смещением '-08:00'. Если системное смещение составляет, скажем, '-05:00', функция сделает поправку на эту величину и вычтет 3 часа из входного значения.

Функция TODATETIMEOFFSET используется с другой целью. Ее применяют для построения значения DATETIMEOFFSET из двух входных величин: первая — значение

¹ Дату можно представить, например, так: dd.mm.yyyy (день.месяц.год). В отношении части mm: dd является более низкой частью в уровне иерархии даты, а yyyy — более высокой частью. — Ред.

даты и времени, которое ничего не знает о смещении, и вторая — собственно смещение. Эту функцию можно использовать для миграции даты, не поддерживающей смещение, когда вы храните значение локальной даты и времени в одном атрибуте, а смещение в другом, в дату, поддерживающую смещение. Пусть имеется локальное время и дата в атрибуте с именем `dt` и смещение в атрибуте с именем `theoffset`. Вы добавляете атрибут с именем `dto` типа DATETIMEOFFSET в таблицу. Затем вы обновляете выражение `TODATETIMEOFFSET(dt, theoffset)` с новым атрибутом и после этого отбрасываете исходные атрибуты `dt` и `theoffset` из таблицы.

Следующий код демонстрирует использование обеих функций:

```
SELECT
SWITCHOFFSET('20130212 14:00:00.0000000 -08:00', '-05:00') AS [SWITCHOFFSET],
TODATETIMEOFFSET('20130212 14:00:00.0000000', '-08:00') AS [TODATETIMEOFFSET];
```

Далее приведены выходные данные этого кода.

SWITCHOFFSET	TODATETIMEOFFSET
2013-02-12 17:00:00.0000000 -05:00	2013-02-12 14:00:00.0000000 -08:00

Функции символьных типов данных

Язык T-SQL в действительности не был предназначен для поддержки функции сложного манипулирования символьными строками, поэтому в нем не найдется большого количества таких функций. В этом разделе описаны функции работы с символьными строками, поддерживаемые языком T-SQL и собранные в категории.

Объединение

Объединение символьных строк — очень часто используемая операция. Язык T-SQL поддерживает два способа объединения строк: первый — с использованием оператора "плюс" (+), а второй — с помощью функции `CONCAT`.

Далее приведен пример объединения строк в запросе с помощью оператора +.

```
SELECT empid, country, region, city,
country + N',' + region + N',' + city AS location
FROM HR.Employees;
```

Вот как выглядит результат этого запроса.

empid	country	region	city	location
1	USA	WA	Seattle	USA,WA,Seattle
2	USA	WA	Tacoma	USA,WA,Tacoma
3	USA	WA	Kirkland	USA,WA,Kirkland
4	USA	WA	Redmond	USA,WA,Redmond
5	UK	NULL	London	NULL
6	UK	NULL	London	NULL
7	UK	NULL	London	NULL
8	USA	WA	Seattle	USA,WA,Seattle
9	UK	NULL	London	NULL

Обратите внимание, если хоть одно из входных значений равно NULL, оператор + возвращает NULL. Это стандартное поведение, изменить которое можно выключением параметра сеанса CONCAT_NULL_YIELDS_NULL, хотя использование нестандартного поведения не рекомендуется. Если есть необходимость заменить значение NULL пустой строкой, существует несколько способов сделать это программными средствами. Один из способов — использование функции COALESCE(<exp>, ''). Например, в приведенных данных только регион (region) может принимать значение NULL, так что можно использовать следующий запрос для замены ' + region пустой строкой, если регион имеет значение NULL.

```
SELECT empid, country, region, city,
country + COALESCE( N', ' + region, N'') + N', ' + city AS location
FROM HR.Employees;
```

Другой способ — использование функции CONCAT, которая, в отличие от оператора +, заменяет входное значение NULL пустой строкой. Посмотрите, как выглядит этот запрос.

```
SELECT empid, country, region, city,
CONCAT(country, N', ' + region, N', ' + city) AS location
FROM HR.Employees;
```

Далее приведены выходные данные этого запроса.

empid	country	region	city	location
1	USA	WA	Seattle	USA, WA, Seattle
2	USA	WA	Tacoma	USA, WA, Tacoma
3	USA	WA	Kirkland	USA, WA, Kirkland
4	USA	WA	Redmond	USA, WA, Redmond
5	UK	NULL	London	UK, London
6	UK	NULL	London	UK, London
7	UK	NULL	London	UK, London
8	USA	WA	Seattle	USA, WA, Seattle
9	UK	NULL	London	UK, London

Заметьте, что на этот раз, если регион (region) имел значение NULL, он был заменен пустой строкой (в location).

Извлечение подстроки и ее позиция

В данном разделе рассматриваются функции, которые можно использовать для извлечения подстроки из строки и указания позиции подстроки в строке.

Функция SUBSTRING позволяет извлечь подстроку из строки, представленной первым аргументом, начиная с позиции, которая указана вторым аргументом, и длины выхода, указанной третьим аргументом. Например, функция SUBSTRING('abcde', 1, 3) возвращает 'abc'. Если третий аргумент указывает на позицию, которая больше, чем длина строки, функция не выдает ошибку, она просто извлекает подстроку до конца строки.

Функции LEFT и RIGHT извлекают запрашиваемое количество символов с левого или правого края входной строки соответственно. Например, функция LEFT('abcde', 3) возвращает подстроку 'abc', а функция RIGHT('abcde', 3) возвратит 'cde'.

Функция CHARINDEX возвращает позицию первого вхождения строки, представленной первым аргументом, в строке, представленной вторым аргументом. Например, функция CHARINDEX(' ', 'Itzik Ben-Gan') ищет первое вхождение пробела во втором входном аргументе и возвращает 5 в данном случае. Заметьте, можно указать третий аргумент, обозначающий позицию начала поиска.

В одном и том же выражении можно объединять или вкладывать функции. Например, пусть мы имеем запрос к таблице с атрибутом fullname в формате '<first> <last>', и нужно написать выражение, которое извлекает первую часть полного имени. Можно использовать следующее выражение.

```
LEFT(fullname, CHARINDEX(' ', fullname) - 1)
```

Язык T-SQL также поддерживает функцию PATINDEX, которая, как и CHARINDEX, может использоваться для определения первой позиции строки внутри другой строки. Но если с помощью функции CHARINDEX вы ищете постоянную строку, то функция PATINDEX выполняет поиск шаблона. Шаблон очень похож на шаблон LIKE, который вам, вероятно, знаком, где можно использовать подстановочные символы, такие как % для любой строки, _ для одного символа и квадратные скобки ([]), представляющие один символ из определенного списка или диапазона. Если вы не знакомы с конструкцией этих шаблонов, прочтите разделы "PATINDEX (Transact-SQL)" и "LIKE (Transact-SQL)" в электронной документации по SQL Server 2012 по адресам [http://msdn.microsoft.com/ru-ru/library/ms188395\(v=SQL.110\).aspx](http://msdn.microsoft.com/ru-ru/library/ms188395(v=SQL.110).aspx) и [http://msdn.microsoft.com/ru-ru/library/ms179859\(v=SQL.110\).aspx](http://msdn.microsoft.com/ru-ru/library/ms179859(v=SQL.110).aspx). Например, выражение PATINDEX('%[0-9]%', 'abcd123efgh') ищет первое вхождение цифры (символ в диапазоне 0—9) во втором выражении и возвращает позицию 6.

Длина строки

Язык T-SQL содержит две функции, которые можно использовать для измерения длины входного значения, — LEN и DATALENGTH.

Функция LEN возвращает длину входной строки в виде определенного количества символов. Обратите внимание, она возвращает количество символов, а не байтов, независимо от того, является входное значение строкой в стандартной кодировке или в кодировке Unicode. Например, выражение LEN(N'xyz') возвращает 3. Если имеются хвостовые пробелы, функция LEN их удаляет.

Функция DATALENGTH возвращает длину входного значения в виде числа байтов. Это означает, например, что если на входе — символьная строка в кодировке Unicode, то будет занято 2 байта на символ. Например, функция DATALENGTH(N'xyz') возвращает 6. Также заметьте, что в отличие от функции LEN функция DATALENGTH не удаляет хвостовые пробелы.

Изменение строк

Язык T-SQL поддерживает ряд функций, которые можно использовать для применения изменений к входным строкам. К ним относятся функции REPLACE, REPLICATE и STUFF.

С помощью функции REPLACE можно заменить во входной строке, являющейся первым аргументом, все вхождения строки, являющейся вторым аргументом. Например, функция REPLACE('.1.2.3.', '.', '/') заменяет все вхождения точки (.) следующим (/), возвращая строку '/1/2/3/'.

Функция REPLICATE позволяет повторить входную строку необходимое количество раз. Например, выражение REPLICATE('0', 10) повторяет строку '0' десять раз, возвращая '0000000000'.

Функция STUFF производит действия над входной строкой, представленной в качестве первого аргумента; затем, из позиции символа, указанной вторым аргументом, удаляет число символов, указанное третьим аргументом. Потом она вставляет в эту позицию строку, указанную четвертым аргументом. Например, функция STUFF('x,y,z', 1, 1, '') удаляет первый символ во входной строке и возвращает 'x,y,z'.

Форматирование строк

В этом разделе рассматриваются функции, которые можно использовать для применения форматирования к входной строке. Это функции UPPER, LOWER, LTRIM, RTRIM и FORMAT.

Суть первых четырех функций очевидна (верхний регистр входных данных, нижний регистр входных данных, отсечение ведущих пробелов и отсечение концевых пробелов). Обратите внимание, здесь нет функции TRIM, которая удаляет и ведущие, и концевые пробелы; чтобы этого достигнуть, нужно вложить вызов одной функции в другую, как в случае RTRIM(LTRIM(<param>)).

С помощью функции FORMAT можно форматировать входное значение на основе строки форматирования и дополнительно указать культуру (язык и региональные параметры) в качестве третьего входного параметра там, где это имеет смысл. Можно использовать любой формат строк, поддерживаемый .NET Framework. (Подробную информацию можно найти в разделах "FORMAT (Transact-SQL)" и "Типы форматирования" по адресам [http://msdn.microsoft.com/ru-ru/library/hh213505\(v=sql.110\).aspx](http://msdn.microsoft.com/ru-ru/library/hh213505(v=sql.110).aspx) и <http://msdn.microsoft.com/ru-ru/library/26etazsy.aspx>.) Например, функция FORMAT(1759, '0000000000') форматирует входное число в символьную строку с фиксированным размером в 10 символов и с ведущими нулями и возвращает '0000001759'.

Выражение CASE и связанные с ним функции

Язык T-SQL поддерживает выражение с именем CASE и несколько связанных с ним функций, которые можно использовать для применения условной логики для определения возвращаемого значения. Многие неправильно называют выражение CASE

инструкцией. Инструкция выполняет некоторые действия или управляет последовательностью выполнения кода, но это совсем не то, что делает выражение CASE; выражение CASE возвращает значение и, следовательно, это выражение.

Выражение CASE имеет две формы — простую и поисковую. Далее приведен пример простой формы выражения CASE к учебной базе данных TSQL2012.

```
SELECT productid, productname, unitprice, discontinued,
CASE discontinued
    WHEN 0 THEN 'No'
    WHEN 1 THEN 'Yes'
    ELSE 'Unknown'
END AS discontinued_desc
FROM Production.Products;
```

Простая форма этого выражения сравнивает *входное выражение* (в данном случае это атрибут discontinued) с несколькими возможными скалярными величинами WHEN (в данном случае 0 и 1) и возвращает результирующее выражение (в данном случае 'No' и 'Yes' соответственно), соответствующее первому совпадению. Если совпадений нет и указано предложение ELSE, возвращается выражение else (в данном случае 'Unknown'). Если же предложение ELSE отсутствует, по умолчанию выдается ELSE NULL. Далее приведен сокращенный вариант выходного набора для данного запроса.

productid	productname	unitprice	discontinued	discontinued_desc
1	Product HHYDP	18.00	0	No
2	Product RECZE	19.00	0	No
3	Product IMEHJ	10.00	0	No
4	Product KSBRM	22.00	0	No
5	Product EPEIM	21.35	0	Yes
...				

Поисковая форма выражения CASE является более гибкой. Вместо сравнения входного выражения с несколькими возможными выражениями она использует предикаты в предложениях WHEN, и первый предикат, который принимает значение "истина", определяет, когда и какое выражение должно возвращаться. Если ни один предикат не принимает значение "истина", выражение CASE возвращает предложение ELSE. Посмотрите следующий пример:

```
SELECT productid, productname, unitprice,
CASE
    WHEN unitprice < 20.00 THEN 'Low'
    WHEN unitprice < 40.00 THEN 'Medium'
    WHEN unitprice >= 40.00 THEN 'High'
    ELSE 'Unknown'
END
AS pricerange
FROM Production.Products;
```



В этом примере выражение CASE возвращает описание диапазона цен единицы товара. Если цена единицы товара ниже 20 долларов, возвращается значение 'Low', когда она выше или равна 20 долларам или ниже 40 долларов, возвращается значение 'Medium', а когда она равна 40 далларам или выше, то возвращается значение 'High'. Для надежности здесь есть предложение ELSE; если на входе значение NULL, выражение ELSE возвратит значение 'Unknown'. Заметьте, что второму предикату WHEN не нужно явно проверять, равна ли величина 20 долларам или более. Это потому, что предикаты WHEN оцениваются по порядку, и первый предикат WHEN не дает значение истины. Далее приведен выходной набор данного запроса в сокращенном виде.

productid	productname	unitprice	pricerange
1	Product HYDPP	18.00	Low
2	Product RECZE	19.00	Low
3	Product IMEHJ	10.00	Low
4	Product KSBRM	22.00	Medium
5	Product EPEIM	21.35	Medium

Язык T-SQL поддерживает несколько функций, которые можно рассматривать как сокращенные варианты выражения CASE. Это стандартные функции COALESCE и NULLIF, а также нестандартные функции ISNULL, IIF и CHOOSE.

Функция COALESCE принимает список выражений на вход и возвращает первое выражение, не равное NULL, или NULL, если все выражения имеют значение NULL. Например, функция COALESCE(NULL, 'x', 'y') возвращает 'x'. В более общем смысле, функция

COALESCE(<exp1>, <exp2>, ..., <expn>)

аналогична следующему:

```
CASE
    WHEN <exp1> IS NOT NULL THEN <exp1>
    WHEN <exp2> IS NOT NULL THEN <exp2>
    ...
    WHEN <expn> IS NOT NULL THEN <exp2>
    ELSE NULL
END
```

Типичным использованием функции COALESCE является замена значения NULL чем-либо другим. Например, функция COALESCE(region, '') возвращает значение региона, если это не NULL, и пустую строку, если это NULL.

Язык T-SQL поддерживает нестандартную функцию ISNULL, которая сходна со стандартной функцией COALESCE, но несколько более ограничена в том смысле, что она поддерживает только два входных значения. Так же как и функция COALESCE, она возвращает первое входное значение, не равное NULL. Таким образом, вместо COALESCE(region, '') можно использовать выражение (region, ''). Как правило, рекомендуется использовать стандартные возможности, за исключением случаев,

когда нестандартные функции представляют интерес с точки зрения гибкости или производительности. Функция `ISNULL` фактически является более ограниченной, чем функция `COALESCE`, поэтому, как правило, рекомендуется прибегать к функции `COALESCE`.

Существует несколько тонких различий между функциями `COALESCE` и `ISNULL`, которые могут быть вам интересны. Одно из различий — определение входными данными типа выходных данных. Рассмотрим следующий код:

```
DECLARE  
    @x AS VARCHAR(3) = NULL,  
    @y AS VARCHAR(10) = '1234567890';  
SELECT COALESCE(@x, @y) AS [COALESCE], ISNULL(@x, @y) AS [ISNULL];
```

Вот как выглядит выход этого кода.

COALESCE	ISNULL
-----	-----
1234567890	123

Заметьте, что тип выражения `COALESCE` определяется возвращаемым элементом, тогда как тип выражения `ISNULL` определяется первыми выходными данными.

Еще одно различие между функциями `COALESCE` и `ISNULL` наблюдается при использовании инструкции `SELECT INTO`, которая подробно рассматривается в главе 11. Предположим, список `SELECT` инструкции `SELECT INTO` содержит выражение `COALESCE(col1, 0) AS newcol1` в сравнении с `ISNULL(col1, 0) AS newcol1`. Если исходный атрибут `col1` определен как `NOT NULL`, оба выражения в выходной таблице дадут атрибут, определенный как `NOT NULL`. Однако, если исходный атрибут `col1` определен как допускающий значения `NULL`, функция `COALESCE` создаст результатирующий атрибут, допускающий значения `NULL`, тогда как функция `ISNULL` создаст атрибут, не допускающий значения `NULL`.

COBET

Подготовка к экзамену

Функции `COALESCE` и `ISNULL` могут влиять на производительность при комбинировании наборов; например, при использовании объединений или при фильтрации данных. Рассмотрим пример, в котором имеются две таблицы — `T1` и `T2`, и необходимо объединить их на основе совпадений между `T1.col1` и `T2.col1`. Атрибуты разрешают значения `NULL`. Обычно сравнение между двумя значениями `NULL` дают неизвестную величину, и это приводит к тому, что строки отбрасываются. Вы хотите рассматривать два пустых значения как равные. В таком случае лучше, что можно сделать — использовать функции `COALESCE` или `ISNULL` для замены `NULL` значением, которое точно не может появиться в данных. Например, если атрибуты целочисленные и вы знаете, что имеете только положительные значения в данных (можно даже установить ограничения для этого), можно использовать предикат `COALESCE(T1.col1, -1) = COALESCE(T2.col1, -1)` или `ISNULL(T1.col1, -1) = ISNULL(T2.col1, -1)`. Проблема в том, что поскольку вы манипулируете атрибутами, которые сравниваете, SQL Server не может обеспечить упорядочение индекса. Это может привести к эффективному использованию доступных индексов. Рекомендуется применять более длинную форму: `T1.col1 = T2.col1 OR (T1.col1 IS NULL AND T2.col1 IS NULL)`. Ее SQL Server понимает как просто сравнение, которое рассматривает пустые значения как равные. С помощью этой формы SQL Server может эффективно использовать индексацию.

Язык T-SQL также поддерживает стандартную функцию `NULLIF`. Эта функция принимает два входных выражения и возвращает `NULL`, если они равны, или первое входное выражение, если они не равны. Например, рассмотрим выражение `NULLIF(col1, col2)`. Если `col1` равно `col2`, функция возвращает `NULL`; в противном случае она возвращает значение `col1`.

Что касается функций `IIF` и `CHOOSE`, эти нестандартные функции T-SQL были добавлены, чтобы упростить миграцию из платформ Microsoft Access. Поскольку эти функции не являются стандартными и имеются простые стандартные альтернативы, такие как выражения `CASE`, использовать их не рекомендуется. Однако, при миграции из Access в SQL Server, эти функции могут быть полезны с точки зрения более гладкой миграции, а затем постепенно можно провести рефакторинг кода так, чтобы перейти на стандартные функции. С помощью функции `IIF` можно возвращать одно значение, если входной предикат принимает значение "истина", и другое — в противном случае. Эта функция имеет следующий синтаксис:

```
IIF(<predicate>, <true_result>, <false_or_unknown_result>)
```

Данное выражение эквивалентно следующему:

```
CASE WHEN <predicate> THEN <true_result> ELSE <false_or_unknown_result> END
```

Например, выражение `IIF(orderyear = 2012, qty, 0)` возвращает значение в атрибуте `qty`, если атрибут `orderyear` равен 2012, и ноль в противном случае.

Функция `CHOOSE` позволяет получить позицию и список выражений и возвращает выражение в указанной позиции. Эта функция имеет следующий синтаксис:

```
CHOOSE(<pos>, <exp1>, <exp2>, ..., <expn>)
```

Например, выражение `CHOOSE(2, 'x', 'y', 'z')` возвращает 'y'. Еще раз, вполне естественно заменить выражение `CHOOSE` его логическим эквивалентом — выражением `CASE`; но смысл поддержки выражения `CHOOSE`, так же как и `IIF`, — использовать временное решение для упрощения миграции из приложения Access в SQL Server.

КОНТРОЛЬНЫЕ ВОПРОСЫ

- Будете ли вы использовать тип данных `FLOAT` для представления цены единицы товара?
- В чем заключается разница между функциями `NEWID` и `NEWSEQUENTIALID`?
- Какая функция возвращает значение текущей даты и времени с типом данных `DATETIME2`?
- В чем заключается разница между оператором `+` и функцией `CONCAT` при объединении символьных строк?

Ответы на контрольные вопросы

- Нет, поскольку тип данных `FLOAT` — это приблизительный тип данных и не может представлять все значения точно.
- Функция `NEWID` генерирует значения идентификатора GUID в произвольном порядке, тогда как функция `NEWSEQUENTIALID` генерирует идентификаторы GUID, которые увеличиваются последовательно.

3. Функция SYSDATETIME.
4. Оператор + по умолчанию выставляет результирующее значение NULL при входном значении NULL, тогда как функция CONCAT воспринимает значения NULL как пустые строки.

ПРАКТИКУМ Работа с типами данных и встроенными функциями

В данном практикуме вам предстоит проверить ваши знания о типах данных и функциях. Вы будете запрашивать данные из существующей таблицы и манипулировать существующими атрибутами с помощью функций. Вам даны задания, в которых требуется написать запросы для решения поставленных задач. Рекомендуется сначала попробовать написать запросы самостоятельно и затем сравнить полученные результаты с предложенным решением.

Задание 1. Применение конкатенации строк и использование функций даты и времени

В этом задании вы потренируетесь объединять строки и применять функции даты и времени.

1. Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
2. Напишите запрос к таблице HR.Employees, который возвращает ID сотрудника, его полное имя (объедините атрибуты `firstname`, пробел и `lastname`) и год рождения (с применением функции к атрибуту `birthdate`). Далее приведен вариант запроса, решающего эту задачу.

```
SELECT empid,
       firstname + N' ' + lastname AS fullname,
       YEAR(birthdate) AS birthyear
  FROM HR.Employees;
```

Задание 2. Использование дополнительных функций даты и времени

В этом задании вы попрактикуетесь в использовании дополнительных функций даты и времени.

Напишите выражение, которое вычисляет дату последнего дня текущего месяца. Также напишите выражение, которое вычисляет последний день текущего года. Разумеется, есть масса способов решить эту задачу. Далее приведен один из вариантов вычисления последнего дня последнего месяца.

```
SELECT EOMONTH(SYSDATETIME()) AS end_of_current_month;
```

А в следующем примере приведен один из способов вычисления конца текущего года.

```
SELECT DATEFROMPARTS(YEAR(SYSDATETIME()), 12, 31) AS end_of_current_year;
```

С помощью функции `YEAR` можно извлечь текущий год, а затем предоставить текущий год с номером месяца 12 и количеством дней 31 функции `DATEFROMPARTS` для построения последнего дня текущего года.

Задание 3. Использование строковых данных и функций преобразования

В этом задании вы будете тренироваться в использовании строковых данных и функций преобразования.

1. Напишите запрос к таблице `Production.Products`, который возвращает существующий числовый ID продукта, а также ID продукта в формате строки фиксированной длины в 10 знаков с ведущими нулями. Например, для продукта с ID равным 42, нужно возвратить строку '`0000000042`'. Один из способов решения этой задачи — использовать следующий код:

```
SELECT productid,
       RIGHT(REPLICATE('0', 10) + CAST(productid AS VARCHAR(10)), 10)
  AS str_productid
  FROM Production.Products;
```

2. Используя функцию `REPLICATE`, сгенерируйте строку, состоящую из 10 нулей. Далее объедините символьную форму ID продукта. Затем извлеките 10 самых правых символов из результирующей строки.

Вы можете предложить более простой способ решения той же задачи с помощью новых функций, появившихся в SQL Server 2012? Значительно проще решить эту задачу с помощью функции `FORMAT`, как показано в следующем примере:

```
SELECT productid,
       FORMAT(productid, 'd10') AS str_productid
  FROM Production.Products;
```

Резюме занятия

- Выбор типа данных для атрибутов оказывает исключительно важное влияние на функциональность и производительность кода T-SQL, взаимодействующего с данными — и еще в большей степени это справедливо для атрибутов, используемых в ключах. Поэтому выбору типов данных следует уделять очень большое внимание.
- Язык T-SQL поддерживает множество функций, которые можно использовать для манипулирования данными даты и времени, символьными строковыми данными и другими типами данных. Помните, что язык T-SQL в основном был предназначен для обработки данных, а не для форматирования или подобных задач. Таким образом, в этих областях, как правило, можно получить только базовую поддержку. Подобные задачи, как правило, лучше всего выполнять на клиенте.
- Язык T-SQL предоставляет выражение `CASE`, которое позволяет возвращать значение с использованием условной логики, а также множество функций, которые можно рассматривать сокращенными вариантами выражения `CASE`.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Почему важно использовать соответствующие типы для атрибутов?
 - A. Потому что тип атрибута позволяет управлять форматированием значений.
 - B. Потому что тип атрибута ограничивает значения до определенной области поддерживаемых значений.
 - C. Потому что тип атрибута предотвращает появление дубликатов.
 - D. Потому что тип атрибута предотвращает появление значений NULL.
2. Какую из перечисленных далее функций вы будете использовать для генерации суррогатных ключей? (Выберите все подходящие ответы.)
 - A. NEWID.
 - B. NEWSEQUENTIALID.
 - C. GETDATE.
 - D. CURRENT_TIMESTAMP.
3. В чем состоит разница между простым выражением CASE и поисковым выражением CASE?
 - A. Простое выражение CASE используется, когда модель восстановления базы данных проста, поисковое выражение CASE используется, когда зарегистрировано полное или неполное восстановление базы данных.
 - B. Простое выражение CASE сравнивает входное выражение с несколькими возможными выражениями в предложениях WHEN, а поисковое выражение CASE использует независимые предикаты в предложении WHEN.
 - C. Простое выражение CASE можно использовать где угодно в запросе, а поисковое выражение CASE — только в предложении WHERE.
 - D. Простое выражение CASE можно использовать где угодно в запросе, а поисковое выражение CASE — только в фильтрах запроса (ON, WHERE, HAVING).

Упражнения

В следующем упражнении вы примените полученные знания об инструкции SELECT. Ответы на вопросы можно найти в *приложении "Ответы" в конце книги*.

Упражнение 1. Анализ использования типов данных

Вы приглашены в качестве консультанта, чтобы решить проблемы с производительностью в существующей системе. Изначально система была разработана с помощью SQL Server 2005 и недавно была обновлена до версии SQL Server 2012.

Скорость записи в системе очень низка, но производительность более чем достаточная. Производительность записи не является приоритетной задачей. Но зато высокий приоритет имеет производительность чтения, которая считается неудовлетворительной на данный момент. Одна из главных задач консультации — представить рекомендации, которые смогут помочь увеличить производительность чтения. У вас назначена встреча с представителями заказчика, и они просят ваших рекомендаций в отношении различных возможностей улучшения ситуации. Одна из интересующих их областей — использование типов данных. Вам надо ответить на следующие вопросы заказчика:

1. Мы используем множество атрибутов, представляющих даты, таких как дата заказа, дата выставления счета-фактуры и т. д., и в данный момент мы используем для этого тип данных `DATETIME`. Рекомендуете ли вы придерживаться существующего типа данных или заменить его другим? Можете дать еще какие-то аналогичные рекомендации?
2. Мы имеем собственное решение для секционирования таблиц, поскольку используем стандартный выпуск SQL Server. Мы также используем суррогатный ключ типа `UNIQUEIDENTIFIER` с функцией `NEWID`, вызываемой выражением ограничения по умолчанию в качестве первичного ключа для таблиц. Этот подход выбран потому, что мы не хотим, чтобы возникали конфликты между ключами в разных таблицах. Этот первичный ключ так же используется, как кластеризованный индексный ключ. Можете ли вы дать рекомендации относительно нашего выбора ключей?

Упражнение 2. Анализ использования функций

Та же компания, которая пригласила вас для анализа использования типов данных, просит вас проанализировать использование у нее функций. Вам задают следующий вопрос.

- Наше приложение до сих пор работало с SQL Server, но из-за недавнего слияния с другой компанией мы также вынуждены поддерживать другие платформы баз данных. Какие рекомендации можете вы нам дать с точки зрения использования функций?

Рекомендуемые упражнения

Выполните следующие задания, которые помогут вам успешно справиться с заданиями экзамена, представленными в данной главе.

Анализ типов данных в учебной базе данных

Чтобы проверить свои знания в области типов данных, проанализируйте типы данных в учебной базе данных TSQL2012.

- Задание 1.** С помощью **Object Explorer** (Обозреватель объектов) в SSMS найдите учебную базу данных TSQL2012. Проанализируйте выбор типов данных для различных атрибутов и попытайтесь обосновать этот выбор. Также оцените,

являются ли выбранные варианты оптимальными, и подумайте, есть ли возможность улучшения некоторых из них.

- **Задание 2.** Обратитесь в электронной документации к разделу "Приоритет типов данных (Transact-SQL)" по адресу <http://msdn.microsoft.com/ru-ru/library/ms190309.aspx>. Укажите порядок приоритета типов данных INT, DATETIME и VARCHAR. Попытайтесь обосновать этот порядок приоритета, установленный компанией Microsoft.

Анализ образцов кода из электронной документации по SQL Server 2012

- **Задание 1.** Найдите раздел "Типы данных и функции даты и времени (Transact-SQL)" в электронной документации по SQL Server 2012 по адресу [http://msdn.microsoft.com/ru-ru/library/ms186724\(v=SQL.110\).aspx](http://msdn.microsoft.com/ru-ru/library/ms186724(v=SQL.110).aspx). Перейдите оттуда по ссылкам на статьи о функциях, которые кажутся вам полезными. В этих статьях перейдите в секцию "Примеры". Проанализируйте эти примеры.
- **Задание 2.** Так же как и в задании 1, найдите в электронной документации по SQL Server 2012 раздел "Строковые функции (Transact-SQL)" по адресу [http://msdn.microsoft.com/ru-ru/library/ms181984\(v=SQL.110\).aspx](http://msdn.microsoft.com/ru-ru/library/ms181984(v=SQL.110).aspx). Перейдите оттуда по ссылкам на функции, которые кажутся вам полезными. В этих статьях перейдите в секцию "Примеры". Проанализируйте и выполните примеры, убедитесь в том, что поняли их.

ГЛАВА 3

Фильтрация и сортировка данных

Темы экзамена

- Работа с данными.
 - Запрос данных с помощью предложения SELECT.
 - Реализация типов данных.
- Модификация данных.
 - Работа с функциями.

Фильтрация и сортировка данных — самые главные и наиболее распространенные аспекты при работе с запросами данных. Практически в каждом создаваемом запросе необходимо выполнять фильтрацию данных, и многие запросы работают с сортировкой данных. Традиционный способ фильтрации данных в языке T-SQL основывается на предикатах. Однако T-SQL также поддерживает другой подход к фильтрации данных — выбор определенного количества строк и сортировка. На нем основывается использование в T-SQL параметров TOP и OFFSET...FETCH.

Что касается сортировки, хотя она и производит впечатление тривиальной задачи при создании запросов, в действительности с ней связано большое количество недоразумений и неясностей, которые мы попробуем объяснить в данной главе.

ПРЕЖДЕ ВСЕГО

Для полного освоения этой главы необходимо иметь:

- опыт работы с SQL Server Management Studio (SSMS);
- некоторый опыт в написании кода на языке T-SQL;
- доступ к экземпляру SQL Server 2012 с установленной учебной базой данных TSQL2012.

Занятие 1. Фильтрация данных с помощью предикатов

Язык T-SQL поддерживает в запросах три предложения, которые позволяют фильтровать данные на основе предикатов. Это предложения ON, WHERE и HAVING. Предложения ON и HAVING рассматриваются далее в этой книге. Описание предложения ON является частью обсуждения объединений (joins) в главе 4, а предложение HAVING рассматривается в разделе главы 5, посвященном группировке данных. Данное занятие этой главы как раз посвящено фильтрации данных с помощью предложения WHERE.

Изучив материал этого занятия, вы сможете:

- ✓ Использовать предложение WHERE для фильтрации данных с помощью предикатов
- ✓ Правильно фильтровать данные, содержащие значение NULL
- ✓ Использовать аргументы поиска для эффективной фильтрации данных
- ✓ Комбинировать предикаты и логические операторы
- ✓ Понимать значение троичной логики для фильтрации данных
- ✓ Выполнять фильтрацию символьных данных
- ✓ Выполнять фильтрацию значений даты и времени

Продолжительность занятия — 60 минут.

Предикаты, троичная логика и аргументы поиска

В самых первых SQL-запросах, которые вы составляли, вам, скорее всего, уже приходилось использовать предложение WHERE для фильтрации данных с помощью предикатов. Сначала кажется, что в этом нет ничего сложного и непонятного. Но со временем по мере проникновения в тайны T-SQL становится ясно, что фильтрация имеет не такие уж простые и очевидные стороны. Например, необходимо понять, как предикаты взаимодействуют со значениями NULL и как ведут себя фильтры на основе таких предикатов. Также нужно понимать, как строить предикаты для максимального увеличения эффективности запросов. Для всего этого совершенно необходимо быть хорошо знакомым с концепцией *аргумента поиска*.

В данной главе используется таблица HR.Employees из учебной базы данных TSQL2012. Далее приведена эта таблица (показаны только нужные в данной главе столбцы).

empid	firstname	lastname	country	region	city
1	Sara	Davis	USA	WA	Seattle
2	Don	Funk	USA	WA	Tacoma
3	Judy	Lew	USA	WA	Kirkland
4	Yael	Peled	USA	WA	Redmond
5	Sven	Buck	UK	NULL	London
6	Paul	Suurs	UK	NULL	London

7	Russell	King	UK	NULL	London
8	Maria	Cameron	USA	WA	Seattle
9	Zoya	Dolgopyatova	UK	NULL	London

Начнем с простого примера и рассмотрим следующий запрос, который фильтрует только сотрудников из Соединенных Штатов Америки (USA).

```
SELECT empid, firstname, lastname, country, region, city
FROM HR.Employees
WHERE country = N'USA';
```

В главе 1 мы уже говорили о том, что предикат — это логическое выражение. Когда значения NULL не разрешены в данных (в этом случае столбец `country` определен как не допускающий значения NULL), предикат может принимать значение "истина" или "ложь". Тип логики, используемый в таком случае, известен как *двоичная логика*. Фильтр `WHERE` возвращает только строки, для которых предикат возвращает значение "истина". Далее приведен результат этого запроса.

empid	firstname	lastname	country	region	city
1	Sara	Davis	USA	WA	Seattle
2	Don	Funk	USA	WA	Tacoma
3	Judy	Lew	USA	WA	Kirkland
4	Yael	Peled	USA	WA	Redmond
8	Maria	Cameron	USA	WA	Seattle

Однако если значения NULL могут присутствовать в данных, все несколько усложняется. Давайте рассмотрим столбцы местоположения клиента — `country`, `region` и `city` в таблице `Sales.Customers`. Предположим, эти столбцы отражают иерархию местоположения на основе организации продаж. Для некоторых стран, таких как Соединенные Штаты, применимы все три столбца, как, например:

```
Country: USA
Region: WA
City: Seattle
```

Но другие страны, такие как Великобритания, имеют только две возможные характеристики — страна и город. В подобных случаях столбец `region` имеет значение NULL; например:

```
Country: UK
Region: NULL
City: London
```

Далее рассмотрим запрос, выполняющий фильтрацию только сотрудников из штата Вашингтон (WA).

```
SELECT empid, firstname, lastname, country, region, city
FROM HR.Employees
WHERE region = N'WA';
```

Возвращаясь к главе 1, напомним, что если данные могут содержать NULL-значения, предикат может принимать значение "истина", "ложь" или "неизвестно". Этот тип





логики называется *троичной логикой*. При использовании в предикате, подобном приведенному в предыдущем примере, оператора проверки на равенство будет получено значение "истина", если оба операнда — не NULL-значения и равны, например, `WA` и `WA`. Будет получено значение "ложь", если оба операнда не NULL и не равны, например, `OR` и `WA`. Пока что все просто. Сложнее ситуация, когда появляются NULL-маркеры. Будет получено неизвестное значение, если хотя бы один operand — NULL; например, `NULL` и `WA` или даже `NULL` и `NULL`.

Как уже упоминалось, фильтр `WHERE` возвращает строки, для которых предикат принимает значение "истина", т. е. отбрасываются варианты со значениями "ложь" и "неизвестно". Таким образом, этот запрос возвращает только сотрудников, для которых регион (`region`) не NULL и равен `WA`, как показано в следующем примере:

<code>empid</code>	<code>firstname</code>	<code>lastname</code>	<code>country</code>	<code>region</code>	<code>city</code>
1	Sara	Davis	USA	WA	Seattle
2	Don	Funk	USA	WA	Tacoma
3	Judy	Lew	USA	WA	Kirkland
4	Yael	Peled	USA	WA	Redmond
8	Maria	Cameron	USA	WA	Seattle

Такое поведение может показаться вполне очевидным, но давайте рассмотрим случай, когда надо возвратить только сотрудников, не проживающих в штате Вашингтон (`WA`). Для этого вы напишите следующий запрос:

```
SELECT empid, firstname, lastname, country, region, city
FROM HR.Employees
WHERE region <> N'WA';
```

Запустив этот запрос, вы получите пустой набор на выходе:

<code>empid</code>	<code>firstname</code>	<code>lastname</code>	<code>country</code>	<code>region</code>	<code>city</code>

Вам понятен такой результат?

Как оказалось, все сотрудники, не проживающие в штате Вашингтон, из Великобритании (`UK`); напомним, что величина `region` для населенных пунктов в `UK` имеет значение `NULL`, чтобы обозначить, что это недопустимо. Даже если вам совершенно ясно, что любой человек из `UK` не проживает в штате Вашингтон, это не понятно T-SQL. Для T-SQL `NULL` представляет пустое значение, которое может быть допустимым, оно может иметь значение `WA` точно так же, как и любое другое. Поэтому T-SQL не может с уверенностью решить, что регион отличается от `WA`. Например, если бы в таблице был сотрудник из штата Нью-Йорк (`NY`), соответствующая строка была бы возвращена.

Учитывая, что в таблице `Employees` `NULL`-регион представляет отсутствующий и недопустимый регион, как заставить T-SQL возвратить сведения о таких сотрудниках при поиске местоположений, для которых регион отличается от `WA`?

Для таких предикатов, как `region <> N'WA'` или `region = NULL`, необходимо помнить, что два значения `NULL` не считаются равными друг другу. Фактически, резуль-

тат выражения `NULL = NULL` неизвестен, а не "истина". В языке T-SQL есть предикат `IS NULL` для возвращения значения "истина", когда проверяемый операнд имеет значение `NULL`. Аналогично, предикат `IS NOT NULL` возвращает значение "истина", если проверяемый операнд не равен `NULL`. Таким образом, для решения этой проблемы можно использовать запрос в следующем виде:

```
SELECT empid, firstname, lastname, country, region, city
FROM HR.Employees
WHERE region <> N'WA'
OR region IS NULL;
```

Далее приведен результат этого запроса:

empid	firstname	lastname	country	region	city
5	Sven	Buck	UK	NULL	London
6	Paul	Suurs	UK	NULL	London
7	Russell	King	UK	NULL	London
9	Zoya	Dolgopyatova	UK	NULL	London

Использование фильтров в запросе имеет значение и с точки зрения производительности. Прежде всего, выполняя фильтрацию строк в запросе (а не на клиенте), мы снижаем нагрузку на сеть. Кроме того, учитывая информацию фильтров в запросе, SQL Server способен оценить возможность использования индексов для эффективного получения данных без полного сканирования таблицы. Важно заметить, что для эффективного использования индекса предикат должен иметь форму, известную как *аргумент поиска* (search argument, SARG). В главе 15 подробно рассматривается индексирование и использование аргументов поиска. В данной главе только кратко описана сама концепция и даны простые примеры.

Предикат в форме "*столбец оператор значение*" или "*значение оператор столбец*" может быть аргументом поиска. Например, такие предикаты, как `col1 = 10` и `col1 > 10`, являются аргументами поиска. Манипулирование фильтруемыми столбцами в большинстве случаев не позволяет предикатам быть аргументами поиска. Примером манипулирования фильтруемым столбцом может быть применение к нему функции, скажем, `F(col1) = 10`, где `F` — некая функция. Это правило имеет исключения, но их немного.

Например, предположим, что имеется хранимая процедура, принимающая входной параметр `@dt`, представляющий входную дату отгрузки (shipped date). Предполагается, что эта процедура должна возвратить заказы, которые были отгружены на дату ввода. Если бы столбец `shippeddate` не разрешал значения `NULL`, для выполнения этой задачи нужно было бы использовать следующий запрос:

```
SELECT orderid, orderdate, empid
FROM Sales.Orders
WHERE shippeddate = @dt;
```

Однако столбец `shippeddate` разрешает значения `NULL`; они представляют заказы, которые еще не были отгружены. Когда пользователям понадобятся все заказы, которые еще не отгружены, пользователи поставят в качестве даты ввода `NULL`, и за-

прос должен будет справиться с этой задачей. Помните, что при сравнении двух NULL-значений вы получаете неизвестную величину, и строка отсеивается. Значит, данная форма предиката не может правильно обработать входные NULL-значения. Но можно использовать функцию COALESCE или ISNULL, чтобы заменить значения NULL значением, которое не существует в данных, как в следующем примере:

```
SELECT orderid, orderdate, empid  
FROM Sales.Orders  
WHERE COALESCE(shippeddate, '19000101') = COALESCE(@dt, '19000101');
```

Проблема заключается в том, что, хотя такой запрос возвращает правильный результат — даже в случае значения NULL на входе — предикат не является аргументом поиска. Это означает, что SQL Server не может эффективно использовать индекс для столбца shippeddate. Для того чтобы сделать предикат аргументом поиска, следует избегать манипулирования фильтруемым столбцом и переписать предикат следующим образом:

```
SELECT orderid, orderdate, empid  
FROM Sales.Orders  
WHERE shippeddate = @dt  
OR (shippeddate IS NULL AND @dt IS NULL);
```

СОВЕТ**Подготовка к экзамену**

Для успешного прохождения экзамена необходимо хорошо понимать влияние функций COALESCE и ISNULL на производительность.

Любопытно, что стандартный SQL имеет предикат с именем IS NOT DISTINCT FROM, который имеет то же значение, что и предикат, использованный в последнем запросе (возвращает истину, когда обе части равны или когда обе имеют значение NULL, в противном случае возвращает ложь). К сожалению, T-SQL не поддерживает этот предикат.

В другом примере манипулирования фильтруемый столбец входит в выражение, например, $col1 - 1 \leq @n$. Иногда можно переписать предикат в форме, являющейся аргументом поиска, что сделает возможным эффективное использование индексирования. Например, последний предикат можно переписать с помощью простого математического выражения — $col1 \leq @n + 1$.

Словом, если предикат включает манипулирование фильтруемым столбцом и имеются альтернативные способы выразить его без этого манипулирования, это может увеличить вероятность эффективного использования индексирования. Далее в разд. "Фильтрация символьных данных" и "Фильтрация значений даты и времени" мы рассмотрим еще несколько подобных примеров. И как уже говорилось ранее, более подробно эта тема освещена в главе 15.

Комбинирование предикатов

Предикаты можно объединять в предложении WHERE с помощью логических операторов AND и OR. Можно также инвертировать предикаты, используя логический опе-

ратор NOT. Этот раздел мы начнем с описания важных аспектов инвертирования предикатов и затем обсудим их комбинирование.

Инверсия значений "истина" и "ложь" совершенно понятна — "NOT true" дает значение "ложь", а "NOT false" — "истина". Что может показаться странным, так это инверсия неизвестного значения (unknown) — инверсия неизвестного значения (NOT unknown) остается неизвестным значением.

Вспомните запрос, рассмотренный ранее в этой главе, который возвращает всех сотрудников из штата Вашингтон. Этот запрос использовал в предложении WHERE предикат `region = N'WA'`. Предположим, вам нужно возвратить сотрудников, не проживающих в штате WA, и для этого вы используете предикат `NOT region = N'WA'`. Понятно, что случаи, которые возвращают значение "ложь" для положительного предиката (скажем, когда регион равен NY), возвращают значение "истина" для предиката с отрицанием. Так же понятно и то, что в случае, когда возвращается значение "истина" для положительного предиката (к примеру, регион равен WA), возвращают значение "ложь" для предиката с отрицанием. Однако если регион имеет значение `NULL`, и положительный предикат, и предикат с отрицанием возвращают неизвестное значение, и такая строка отбрасывается. Поэтому правильный способ для вас включить случаи с `NULL` в результат — если вы уверены, что вам это нужно — использовать оператор `IS NULL` как в выражении `NOT region = N'WA' OR region IS NULL`.

Что касается комбинирования предикатов, следует знать несколько интересных вещей. Существуют правила приоритета, которые определяют порядок логической оценки различных предикатов. Оператор NOT выполняется раньше операторов AND и OR, а оператор AND раньше оператора OR. Например, предположим, что фильтр WHERE в нашем запросе имеет следующую комбинацию предикатов.

```
WHERE col1 = 'w' AND col2 = 'x' OR col3 = 'y' AND col4 = 'z'
```

Поскольку оператор AND выполняется раньше оператора OR, вы получите следующий эквивалент:

```
WHERE (col1 = 'w' AND col2 = 'x') OR (col3 = 'y' AND col4 = 'z')
```

Если попытаться выразить эти операторы как псевдофункции, эта комбинация операторов будет эквивалентна выражению `OR(AND(col1 = 'w', col2 = 'x'), AND(col3 = 'y', col4 = 'z'))`.

Поскольку скобки имеют более высокий приоритет по сравнению со всеми операторами, всегда можно использовать их для того, чтобы полностью контролировать логический порядок оценки, который вам нужен, как показано в следующем примере:

```
WHERE col1 = 'w' AND (col2 = 'x' OR col3 = 'y') AND col4 = 'z'
```

Повторим еще раз, с псевдофункцией такая комбинация операторов с учетом скобок эквивалентна выражению `AND(col1 = 'w', OR(col2 = 'x', col3 = 'y'), col4 = 'z')`.

Как говорилось в главе 1, все выражения, которые появляются на одном и том же этапе логической обработки запросов — например, этапе WHERE — в принципе оце-

ниваются в один и тот же момент времени. Например, рассмотрим следующий предикат фильтра:

```
WHERE propertytype = 'INT' AND CAST(propertyval AS INT) > 10
```

Пусть запрашиваемая таблица содержит разные значения свойств. Столбец `propertytype` представляет тип свойства (`INT`, `DATE` и т. д.), а столбец `propertyval` содержит само значение в строке символов. Если `propertytype` — `'INT'`, значение в столбце `propertyval` можно преобразовать в `INT`; в противном случае это делать не обязательно.

Можно предположить, что если правила приоритетности не указывают другого, предикаты будут оцениваться слева направо, и такое вычисление будет иметь место, когда это возможно. Иными словами, если для первого предиката `propertytype = 'INT'` принимает значение "ложь", SQL Server не будет оценивать второй предикат `CAST(propertyval AS INT) > 10`, потому что результат уже известен. Опираясь на это заключение, можно ожидать, что этот запрос никогда не завершится ошибкой, пытаясь преобразовать то, что не может быть преобразовано.

В реальности, тем не менее, все обстоит иначе. SQL Server действительно поддерживает принцип сокращенной обработки; однако в соответствии с принципом единовременности, существующем в T-SQL, выражения вовсе не обязательно будут оцениваться слева направо. Исходя из соображений стоимости, SQL Server может решить начать со второго выражения и затем, если оно даст значение "истина", оценить первое выражение. Это означает, что если имеются строки в таблице, в которых столбец `propertytype` отличается от `'INT'` и в этих строках значение `propertyval` не может быть преобразовано в `INT`, запрос может быть не выполнен из-за ошибки преобразования.

Эту проблему можно решить несколькими способами. Самый простой — использовать функцию `TRY_CAST` вместо функции `CAST`. Когда выражение на входе не может быть преобразовано в целевой тип, функция `TRY_CAST` возвращает значение `NULL`, а не завершается ошибкой. Сравнение значения `NULL` с чем угодно дает неизвестное значение. Наконец, вы получите правильный результат, не позволяя запросу завершиться ошибкой. Поэтому предложение `WHERE` должно быть переписано следующим образом:

```
WHERE propertytype = 'INT' AND TRY_CAST(propertyval AS INT) > 10
```

Фильтрация символьных данных

Во многих отношениях фильтрация символьных данных не отличается от фильтрации других типов данных. В этом разделе рассматриваются некоторые аспекты, типичные именно для символьных данных: правильная форма литералов и предикат `LIKE`.

Как говорилось в главе 2, литерал имеет тип. При написании выражения, в котором есть операнды разных типов, SQL Server должен будет применить неявное преобразование для выравнивания типов данных. В зависимости от обстоятельств, неявные преобразования могут иногда вызывать снижение производительности.

Важно знать надлежащую форму литералов разных типов и быть уверенным, что используется правильная форма. Классический пример использования неправильных типов литералов связан с символьными строками в кодировке Unicode (типы NVARCHAR и NCHAR). Правильная форма для литерала символьной строки в Unicode — использовать в качестве префикса литерала прописную букву N и отделить литерал одиночными кавычками; например, N'literal'. В случае регулярного литерала символьной строки нужно только отделить литерал одиночными кавычками, например, 'literal'. Очень распространенная плохая традиция — задавать регулярный литерал символьной строки, когда фильтруемый столбец имеет тип Unicode, как в следующем примере:

```
SELECT empid, firstname, lastname
FROM HR.Employees
WHERE lastname = 'Davis';
```

Поскольку столбец и литерал имеют разные типы, SQL Server неявно преобразует один тип операнда в другой. В данном примере, к счастью, SQL Server преобразует тип литерала в тип столбца, поэтому по-прежнему можно эффективно применить индексирование. Однако могут быть случаи, когда неявное преобразование снижает производительность. Хорошей практикой считается использование правильной формы, как в этом примере.

```
SELECT empid, firstname, lastname
FROM HR.Employees
WHERE lastname = N'Davis';
```

Язык T-SQL имеет предикат `LIKE`, который можно использовать для фильтрации символьных строковых данных (регулярных или в кодировке Unicode) с помощью сопоставления с шаблоном. Далее приведена форма предиката с использованием `LIKE`.

`<column> LIKE <pattern>`

Предикат `LIKE` поддерживает подстановочные символы, которые можно использовать в шаблонах. В табл. 3.1 представлены допустимые подстановочные символы, их значения и примеры использования.

Таблица 3.1. Подстановочные символы, используемые в шаблонах `LIKE`

Подстановочный знак	Значение	Пример
% (знак процента)	Любая строка, включая пустую	'D%': строка, начинающаяся с D
_ (подчеркивание)	Один символ	'_D%': строка, в которой второй символ D
[<список символов>]	Один символ из списка	'[AC]%' : строка, в которой первый символ А или С
[<диапазон символов>]	Один символ из диапазона	'[0-9]%' : строка, в которой первый символ — цифра
[^<список или диапазон символов>]	Один символ, не входящий в список или диапазон	'[^0-9]%' : строка, в которой первый символ — не цифра

В качестве примера, представим, что вам нужно возвратить всех сотрудников, чьи фамилии начинаются с буквы D. Вам надо использовать такой запрос:

```
SELECT empid, firstname, lastname  
FROM HR.Employees  
WHERE lastname LIKE 'D%';
```

Этот запрос возвращает следующий результат:

empid	firstname	lastname
1	Sara	Davis
9	Zoya	Dolgopyatova

Если необходимо выполнить поиск символа, который является подстановочным знаком, можно указать его после символа, который вы определите как Escape-символ с помощью ключевого слова `ESCAPE`. Например, выражение `col1 LIKE '!_%' ESCAPE '!'` выполняет поиск строки, которая начинается со знака подчеркивания (_), используя в качестве Escape-символа восклицательный знак (!).

ВАЖНО!

Производительность предиката `LIKE`

Когда шаблон `LIKE` начинается с известного префикса, например `col LIKE 'ABC%`, SQL Server в принципе может эффективно использовать индекс для фильтруемого столбца; иными словами, SQL Server способен спокойно выполнять упорядочение по индексу. Если же шаблон начинается с подстановочного знака, например `col LIKE '%ABC%`, SQL Server уже не может полагаться на упорядочение по индексу. Также при поиске строки, которая начинается с известного префикса (скажем, ABC), надо быть уверенным, что используется предикат `LIKE`, как в случае `col LIKE 'ABC%`, поскольку эта форма считается аргументом поиска. Напомним, что применение обработки к фильтруемому столбцу не позволяет предикату быть аргументом поиска. Например, форма `LEFT(col, 3) = 'ABC'` не является аргументом поиска и не дает SQL Server возможности эффективно использовать индекс.

Фильтрация данных даты и времени

При фильтрации данных даты и времени следует принимать во внимание несколько моментов, которые имеют отношение и к правильности кода, и к его производительности. Следует подумать о таких вещах, как отобразить литералы, фильтровать диапазоны и использовать аргументы поиска.

Начнем с литералов. Предположим, нужно создать запрос к таблице `Sales.Orders` и возвратить только заказы, размещенные 12 февраля 2007 года. Для этого используйте следующий запрос:

```
SELECT orderid, orderdate, empid, custid  
FROM Sales.Orders  
WHERE orderdate = '02/12/07';
```

Для американца эта форма, скорее всего, означает 12 февраля 2007 года. Но для британца эта форма скорее значит 2 декабря 2007 года. Для японца она может означать 7 декабря 2002 года. Вопрос в том, как SQL Server интерпретирует это значение, когда он преобразует эту символьную строку в тип даты и времени для выравнивания этих данных с типом фильтруемого столбца? Оказывается, это зависит от

языка входа в систему при запуске кода. Каждый вход в систему имеет язык по умолчанию, который с ним ассоциируется, и этот язык по умолчанию устанавливает различные параметры сессии от имени выполнившего вход в систему, включая параметр DATEFORMAT. Вход в систему с us_english имеет настройку DATEFORMAT, установленную в mdy, для британца это будет dmy, а для японца — ymd. Проблема в том, как разработчику надо записать дату, если он хочет быть понятым именно так, как предполагает, независимо от того, кто запустил код?

Существуют два главных подхода. Один — использовать форму, которая считается не зависящей от языка. Например, форма '20070212' всегда интерпретируется как ymd, вне зависимости от языка. Обратите внимание, форма '2007-02-12' считается не зависящей от языка только для типов данных DATE, DATETIME2 и DATETIMEOFFSET. К сожалению, исторически сложилось так, что эта форма считается зависящей от языка для типов DATETIME и SMALLDATETIME. Преимущество формы без разделителей состоит в том, что она не зависит от языка для всех типов данных даты и времени. Поэтому мы рекомендуем писать запрос подобно следующему:

```
SELECT orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE orderdate = '20070212';
```

ПРИМЕЧАНИЕ Сохранение дат в столбце DATETIME

Фильтруемый столбец orderdate имеет тип данных DATETIME, представляющий и дату, и время. Литерал, указанный в фильтре, имеет только дату. Когда SQL Server преобразует литерал в тип данных фильтруемого столбца, он устанавливает время на полночь, если время не указано. Если надо, чтобы фильтр возвратил все строки с указанной датой, нужно удостовериться, что все значения сохранены со значением времени " полночь".

Другой подход — использовать функции CONVERT или PARSE, чтобы установить, как SQL Server должен интерпретировать указанный вами литерал. Функция CONVERT поддерживает номер стиля, представляющий стиль преобразования, а функция PARSE поддерживает указание имени культуры. В главе 2 представлены описания обеих функций.

Еще один важный аспект фильтрации данных даты и времени — постараться, когда это возможно, использовать аргументы поиска. Например, пусть нужно отфильтровать только заказы, размещенные в феврале 2007 года. Можно использовать функции YEAR и MONTH, как в следующем примере:

```
SELECT orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE YEAR(orderdate) = 2007 AND MONTH(orderdate) = 2;
```

Однако поскольку в данном случае выполняются манипуляции с фильтруемым столбцом, предикат не может рассматриваться как аргумент поиска и, следовательно, SQL Server не сможет полагаться на упорядочение по индексу. Следует переписать этот предикат в виде диапазона, как показано в следующем примере:

```
SELECT orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE orderdate >= '20070201' AND orderdate < '20070301';
```

Теперь, когда фильтруемый столбец не подвергается манипуляциям, предикат считается аргументом поиска и SQL Server потенциально может полагаться на упорядочение по индексу.

Если у вас возникает вопрос, почему здесь диапазон дат выражен с помощью операторов "больше или равно" (\geq) и "меньше" ($<$), а не оператором BETWEEN. Для этого есть причины. Если при использовании оператора BETWEEN столбец содержит и дату, и время, что будет использоваться в виде конечного значения? Как вы понимаете, для разных типов существуют различные точности. Более того, предположим, у вас тип данных DATETIME и вы используете следующий предикат:

```
WHERE orderdate BETWEEN '20070201' AND '20070228 23:59:59.999'
```

Точность этого типа данных равняется 3,33 мс. Часть миллисекунд в конечной части не кратна единице точности времени, так что SQL Server округляет значение до полуночи 1 марта 2007 г. В результате вы можете получить заказы, которые не ожидаете увидеть в выходном наборе. Подводя итог, используйте вместо оператора BETWEEN операторы \geq и $<$, и все будет работать правильно во всех случаях, со всеми типами даты и времени, независимо от того, применима часть, представляющая время, или нет.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем заключаются преимущества с точки зрения производительности при использовании фильтра WHERE?
2. Какая форма предиката фильтра может опираться на упорядочивание по индексу?

Ответы на контрольные вопросы

1. Нагрузку на сеть можно уменьшить с помощью выполнения фильтрации на сервере баз данных, а не на клиенте, и можно использовать индексы, чтобы избежать полного сканирования задействованных баз данных.
2. Аргумент поиска (SARG).

ПРАКТИКУМ Фильтрация данных с помощью предикатов

В этом практикуме вы проверите ваши знания об использовании предикатов для фильтрации данных.

Задание 1. Использование предложения WHERE для фильтрации строк со значением NULL

В этом задании вам нужно использовать предложение WHERE, чтобы отфильтровать непоставленные заказы в таблице Sales.Order.

1. Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.

2. От вас требуется написать запрос, который возвращает заказы, которые еще не поставлены. В этих заказах есть значение `NULL` в столбце `shippeddate`. Для начала используйте следующий запрос:

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE shippeddate = NULL;
```

Но при выполнении этого кода вы получите пустой результирующий набор.

orderid	orderdate	custid	empid
---------	-----------	--------	-------

Причина в том, что когда выражение сравнивает два значения `NULL`, результатом является неизвестное значение, и строка отбрасывается.

3. Перепишите предикат фильтра, чтобы в нем использовался оператор `IS NULL` вместо знака равенства (`=`), как в примере далее.

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE shippeddate IS NULL;
```

На этот раз вы получите правильный результат, приведенный далее в сокращенном виде.

orderid	orderdate	custid	empid
11008	2008-04-08 00:00:00.000	20	7
11019	2008-04-13 00:00:00.000	64	6
11039	2008-04-21 00:00:00.000	47	1
...			

Задание 2. Использование предложения WHERE для фильтрации диапазона дат

В этом задании вы будете использовать предложение `WHERE` для фильтрации заказов из таблицы `Sales.Orders` в пределах определенного диапазона дат.

1. Вам нужно возвратить все заказы, которые были размещены между 11 февраля 2008 года и 12 февраля 2008 года. Столбец `orderdate`, который вы будете фильтровать, имеет тип данных `DATETIME`. В соответствии с текущими данными в таблице все значения `orderdate` имеют время, установленное на полночь. Но давайте представим, что это не так — какое-то время имеет значение, отличное от полночи. Для начала используйте предикат `BETWEEN`, как показано в следующем примере:

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE orderdate BETWEEN '20080211' AND '20080212 23:59:59.999';
```

Поскольку число 999 не кратно единице точности типа данных `DATETIME` (3,33 мс), конечное значение в диапазоне округляется до следующей полночи, и в результате запроса будут строки с датой 13 февраля, которые нам не нужны.

orderid	orderdate	custid	empid
10881	2008-02-11 00:00:00.000	12	4
10887	2008-02-13 00:00:00.000	29	8
10886	2008-02-13 00:00:00.000	34	1
10884	2008-02-12 00:00:00.000	45	4
10883	2008-02-12 00:00:00.000	48	8
10882	2008-02-11 00:00:00.000	71	4
10885	2008-02-12 00:00:00.000	76	6

2. Чтобы решить проблему, исправьте диапазон в фильтре так, чтобы там были операторы `>=` и `<`, как в следующем запросе:

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE orderdate >= '20080211' AND orderdate < '20080213';
```

На этот раз результат будет верным.

Резюме занятия

- Используя предложение `WHERE`, можно выполнять фильтрацию данных с помощью предикатов. Предикаты в языке T-SQL используют трехуровневую логику. Предложение `WHERE` возвращает случаи, когда предикат принимает значение "истина" и отбрасывает все прочие.
- Фильтрация данных с предложением `WHERE` позволяет снизить нагрузку на сеть и может потенциально разрешать использование индексации для минимизации ввода-вывода. Для эффективного использования индексов важно представлять предикаты в виде аргументов поиска.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Что означает термин "троичная логика" в T-SQL?
 - A. Три возможных логических результирующих значения предиката: истина, ложь и `NULL`.
 - B. Три возможных логических результирующих значения предиката: истина, ложь и неизвестное значение.
 - C. Три возможных логических результирующих значения предиката: 1, 0 и `NULL`.
 - D. Три возможных логических результирующих значения предиката: -1, 0 и 1.
2. Какие из перечисленных литералов зависят от языка для типа данных `DATETIME`? (Выберите все подходящие варианты.)
 - A. '2012-02-12'.
 - B. '02/12/2012'.

- C. '12/02/2012'.
D. '20120212'.
3. Какие из перечисленных предикатов являются аргументами поиска? (Выберите все подходящие варианты.)
A. DAY(orderdate) = 1.
B. companyname LIKE 'A%'.
C. companyname LIKE '%A%'.
D. companyname LIKE '%A'.
E. orderdate >= '20120212' AND orderdate < '20120213'.

Занятие 2. Сортировка данных

Считается, что сортировка данных — это тривиальная задача, но это вовсе не так. Она служит причиной множества ошибок и неточностей в языке T-SQL. В данном занятии рассмотрены критически важные различия между отсортированными и неотсортированными данными. Кроме того, описаны предоставляемые T-SQL инструменты для сортировки данных.

Изучив материал этого занятия, вы сможете:

- ✓ Использовать предложение ORDER BY для определения порядка строк в результирующем наборе запроса
- ✓ Объяснить разницу между запросом, содержащим и не содержащим предложение ORDER BY
- ✓ Управлять направлением сортировки по возрастанию и по убыванию
- ✓ Следовать лучшим рекомендациям использования сортировки
- ✓ Определять ограничения сортировки при использовании инструкции DISTINCT
- ✓ Выполнять сортировку по псевдонимам, присвоенным в предложении SELECT

Продолжительность занятия — 30 минут.

Как обеспечить порядок сортировки данных

Возможно, одним из самых запутанных аспектов языка T-SQL можно считать понимание, когда результат запроса гарантированно будет возвращен в определенном порядке и когда — нет. Точное понимание этого аспекта языка напрямую связано с его математической основой — а именно, с математической теорией множеств. Если вы поймете это на самых начальных стадиях написания кода на T-SQL, вам будет значительно проще, чем тем, кто имеет неверное представление об этом.

В качестве примера рассмотрим следующий код:

```
SELECT empid, firstname, lastname, city, MONTH(birthdate) AS birthmonth  
FROM HR.Employees  
WHERE country = N'USA' AND region = N'WA';
```

Можно ли гарантировать, что строки возвращаются в определенном порядке и каков этот порядок?

Кто-то скажет, что строки будут возвращены неотсортированными; кто-то предположит упорядочение по первичному ключу; кто-то предположит сортировку по кластерному индексу; другие решат, что ни один из видов сортировки не гарантирован.

Если вы помните, мы говорили в *главе 1*, что таблица в T-SQL представляет отношение; отношение — это набор, и набор не имеет порядка расположения элементов в нем. Исходя из этого, пока вы явно не укажете в запросе противное, результат этого запроса не имеет гарантированного порядка расположения элементов. Например, наш вопрос при запуске дает вот такой результат:

empid	firstname	lastname	city	birthmonth
1	Sara	Davis	Seattle	12
2	Don	Funk	Tacoma	2
3	Judy	Lew	Kirkland	8
4	Yael	Peled	Redmond	9
8	Maria	Cameron	Seattle	1

Может показаться, что результат отсортирован по значению `empid`, но это совсем не гарантировано. Еще больше может запутать то, что запущенный несколько раз запрос возвращает результат в той же последовательности; но еще раз, это вовсе не гарантировано. Когда система управления базой данных (в данном случае SQL Server) обрабатывает запрос, она знает, что может возвратить результат в любой последовательности, поскольку точные инструкции для возвращения данных в каком-то определенном порядке отсутствуют. Может случиться, что из соображений оптимизации или любых других ядро SQL Server примет решение обработать данные в определенном порядке *в данном случае*. Даже имеется некоторая вероятность, что такие последовательности будут повторяться, если физические обстоятельства не изменятся. Но существует огромная разница между тем, что может случиться из соображений оптимизации или иных и что в действительности гарантированно случится.

Ядро базы данных может — и иногда это делает — изменить вариант выбора, способный повлиять на порядок, в котором возвращаются строки, зная, что может свободно это сделать. Примеры таких изменений выбора включают изменения в распределении данных, доступность физических структур, таких как индексы, а также доступность ресурсов — центрального процессора и памяти. Кроме того, с изменениями в движке после обновления до более новой версии продукта или даже после установки очередного пакета обновлений (service pack), обстоятельства оптимизации могут измениться. В свою очередь, такие изменения могут затронуть, кроме прочего, порядок строк в результате запроса.

Одним словом, следует еще раз повторить: запрос, не содержащий явных инструкций для возвращения строк в определенном порядке, не может гарантировать нужный порядок строк в результате. Если такая гарантия действительно нужна, един-

ственний способ ее иметь — добавить в запрос предложение ORDER BY. Именно этому посвящен следующий раздел.

Использование предложения **ORDER BY** для сортировки данных

Единственный способ действительно гарантировать возвращение строк запросом в определенном порядке — добавить предложение ORDER BY.

Например, если вам нужно возвратить информацию о сотрудниках, живущих в штате Вашингтон в США, отсортированную по городам, нужно указать столбец city в предложении ORDER BY следующим образом:

```
SELECT empid, firstname, lastname, city, MONTH(birthdate) AS birthmonth
FROM HR.Employees
WHERE country = N'USA' AND region = N'WA'
ORDER BY city;
```

Далее приведен результат этого запроса.

empid	firstname	lastname	city	birthmonth
3	Judy	Lew	Kirkland	8
4	Yael	Peled	Redmond	9
8	Maria	Cameron	Seattle	1
1	Sara	Davis	Seattle	12
2	Don	Funk	Tacoma	2

Если не указан порядок сортировки, по умолчанию используется сортировка по возрастанию. Можно задать его явно и указать параметр ASC для города, но это тоже самое, что не указывать последовательность сортировки вообще. Чтобы получить упорядочение по убыванию, надо явно указать параметр DESC, как в следующем примере:

```
SELECT empid, firstname, lastname, city, MONTH(birthdate) AS birthmonth
FROM HR.Employees
WHERE country = N'USA' AND region = N'WA'
ORDER BY city DESC;
```

На этот раз запрос выводит строки, отсортированные по городу в порядке убывания.

empid	firstname	lastname	city	birthmonth
2	Don	Funk	Tacoma	2
1	Sara	Davis	Seattle	12
8	Maria	Cameron	Seattle	1
4	Yael	Peled	Redmond	9
3	Judy	Lew	Kirkland	8

Столбец city не является уникальным для фильтруемой страны и региона, поэтому упорядочение строк для одного и того же города (например, Seattle) не гарантиро-

вано. В таком случае говорят, что упорядочение недетерминированное. Так же как запрос без предложения ORDER BY не гарантирует порядок результирующих строк в общем, запрос, содержащий предложение ORDER BY, в случае когда город не является уникальным, не может гарантировать порядок сортировки для одного и того же названия города. К счастью, в списке ORDER BY можно указать несколько выражений, разделенных запятыми. Один из вариантов использования этой возможности — применить для упорядочения разрыв цепочки. Например, можно задать empid в качестве вторичного столбца сортировки, как показано в примере.

```
SELECT empid, firstname, lastname, city, MONTH(birthdate) AS birthmonth
FROM HR.Employees
WHERE country = N'USA' AND region = N'WA'
ORDER BY city, empid;
```

Далее приведен результат этого запроса.

empid	firstname	lastname	city	birthmonth
3	Judy	Lew	Kirkland	8
4	Yael	Peled	Redmond	9
1	Sara	Davis	Seattle	12
8	Maria	Cameron	Seattle	1
2	Don	Funk	Tacoma	2

Теперь список ORDER BY уникален; а значит, упорядочение является детерминированным. До тех пор, пока базовые данные не изменятся, в дополнение к порядку представления результатов гарантирована их повторяемость. Направление сортировки можно указать по принципу "выражение-за-выражением", например, ORDER BY col1 DESC, col2, col3 DESC (col1 по убыванию, затем col2 по возрастанию, потом col3 по убыванию).

С помощью языка T-SQL можно сортировать по порядковым номерам столбцов в списке SELECT, но это считается плохой практикой. Рассмотрим следующий запрос в качестве примера:

```
SELECT empid, firstname, lastname, city, MONTH(birthdate) AS birthmonth
FROM HR.Employees
WHERE country = N'USA' AND region = N'WA'
ORDER BY 4, 1;
```

В этом примере вы хотите упорядочить строки по четвертому выражению в списке SELECT (city) и затем по первому (empid). В данном примере это равносильно использованию выражения ORDER BY city, empid. Однако это считается плохой практикой по некоторым соображениям. Во-первых, T-SQL отслеживает порядковые позиции столбцов в таблице, в дополнение к позициям в результате запроса, но это не является реляционным. Вспомните, что заголовок отношения — это набор атрибутов, а набор не имеет порядка. Также при использовании порядковых номеров очень просто забыть о соответствующем изменении позиций после внесения изменений в список SELECT. Например, пусть вы решили применить изменения в списке SELECT, но при этом забыли изменить соответственно список ORDER BY. Получился нижеприведенный запрос.

```
SELECT empid, city, firstname, lastname, MONTH(birthdate) AS birthmonth
FROM HR.Employees
WHERE country = N'USA' AND region = N'WA'
ORDER BY 4, 1;
```

Теперь запрос упорядочивает данные по параметрам `lastname` и `empid`, вместо того, чтобы упорядочивать по значениям `city` и `empid`. Подводя итог, правильным считается ссылаться на имена столбцов или на выражения с именами столбцов, но не на порядковые номера.

Заметьте, вы можете упорядочить результирующие строки по элементам, которые не будете возвращать. Например, следующий запрос возвращает для каждого условного сотрудника его идентификатор (`ID`) и город (`city`), упорядочивая результирующие строки по дню рождения сотрудника.

```
SELECT empid, city
FROM HR.Employees
WHERE country = N'USA' AND region = N'WA'
ORDER BY birthdate;
```

Далее приведен результат этого запроса.

empid	city
4	Redmond
1	Seattle
2	Tacoma
8	Seattle
3	Kirkland

Конечно, результат был бы более значительным, если бы вы включили атрибут `birthdate`, но если вы считаете это ненужным, это все равно абсолютно правильно. Правило гласит, что можно упорядочивать результирующие строки по элементам, которые не входят в список `SELECT`, когда результирующие строки там разрешены. Правило меняется, если также указано предложение `DISTINCT` — и вполне обоснованно. При использовании предложения `DISTINCT` удаляются дубликаты; результирующие строки не обязательно соотносятся с исходными строками в соотношении "один-к-одному", скорее, как "один-ко-многим". Например, попробуйте объяснить, почему этот запрос неправильный.

```
SELECT DISTINCT city
FROM HR.Employees
WHERE country = N'USA' AND region = N'WA'
ORDER BY birthdate;
```

Может существовать несколько служащих — с разными днями рождения — в одном и том же городе. Но вы возвращаете в результате запроса только одну строку для каждого конкретного города. И если взять один город (скажем, Сиэтл) с некоторыми сотрудниками, какая из дат рождения сотрудников должна быть применена как параметр сортировки? Запрос не возьмет ни одну из них — он просто не будет выполнен.

Таким образом, если используется предложение DISTINCT в списке ORDER BY могут быть только элементы, присутствующие в списке SELECT, как в следующем примере:

```
SELECT DISTINCT city
FROM HR.Employees
WHERE country = N'USA' AND region = N'WA'
ORDER BY city;
```

Теперь запрос составлен совершенно правильно и возвращает следующий результат:

```
city
-----
Kirkland
Redmond
Seattle
Tacoma
```

О предложении ORDER BY следует также знать и то, что оно в принципе оценивается после предложения SELECT — в отличие от многих других предложений. Это означает, что псевдонимы столбцов, присвоенные в предложении SELECT, фактически видны предложению ORDER BY. Например, следующий запрос использует функцию MONTH для возвращения месяца даты рождения, присваивая выражению столбца псевдоним birthmonth. Затем запрос ссылается на этот псевдоним столбца birthmonth **прямо в предложении ORDER BY**.

```
SELECT empid, firstname, lastname, city, MONTH(birthdate) AS birthmonth
FROM HR.Employees
WHERE country = N'USA' AND region = N'WA'
ORDER BY birthmonth;
```

Запрос возвращает следующий результат:

empid	firstname	lastname	city	birthmonth
8	Maria	Cameron	Seattle	1
2	Don	Funk	Tacoma	2
3	Judy	Lew	Kirkland	8
4	Yael	Peled	Redmond	9
1	Sara	Davis	Seattle	12

Другой непростой аспект упорядочения данных — обработка значений NULL. Напомним, что значение NULL представляет отсутствующее значение, поэтому сравнивая NULL с любым другим значением, получаем неизвестный результат. То же самое относится и к сравнению двух значений NULL. Поэтому вопрос о поведении значений NULL с точки зрения сортировки вряд ли можно считать тривиальным. Должны ли они сортироваться вместе с другими значениями? Если да, должны ли они сортироваться до или после не-NUL-значений? Стандартный язык SQL говорит о том, что NULL-значения должны сортироваться вместе со всеми, но оставляет на усмотрение реализации решать, сортировать их до или после не-NUL-значений. Для SQL

Server было решено сортировать их перед не-NULL-значениями (при возрастающем направлении сортировки). В качестве примера приведем запрос, возвращающий для каждого заказа его идентификатор (ID) и дату отгрузки, упорядоченные по дате отгрузки.

```
SELECT orderid, shippeddate
FROM Sales.Orders
WHERE custid = 20
ORDER BY shippeddate;
```

Помните, что недоставленные заказы имеют NULL в столбце `shippeddate`, следовательно, они сортируются раньше отгруженных заказов, как видно из результата запроса.

orderid	shippeddate
11008	NULL
11072	NULL
10258	2006-07-23 00:00:00.000
10263	2006-07-31 00:00:00.000
10351	2006-11-20 00:00:00.000
...	

Стандартный SQL поддерживает параметры `NOLIMIT FIRST` и `NOLIMIT LAST` для того, чтобы можно было контролировать, как сортируются значения `NULL`, но T-SQL не поддерживает эту возможность. В качестве эксперимента посмотрите, знаете ли вы, как отсортировать заказы по дате отгрузки в возрастающем порядке, но с сортировкой значений `NULL` последними (*подсказка*: можно указать выражения в предложении `ORDER BY`; подумайте, как использовать выражение `CASE` для достижения этой цели).

Итак, напомним, запрос без предложения `ORDER BY` возвращает реляционный результат (по крайней мере, с точки зрения упорядочения) и, следовательно, не гарантирует никакой порядок. Единственный способ гарантировать порядок данных — использование предложения `ORDER BY`. В соответствии со стандартным SQL, запрос, содержащий `ORDER BY`, в принципе возвращает *курсор*, а не отношение.

Индексация рассматривается позднее в данном учебном курсе, сейчас достаточно сказать, что создание правильных индексов способно помочь SQL Server избежать необходимости в сортировке данных для выполнения запроса `ORDER BY`. Без правильных индексов SQL Server должен выполнять сортировку, и она может быть дорогой процедурой, особенно при работе с большими наборами данных. Если у вас нет необходимости в возвращении данных в отсортированном виде, убедитесь, что вы не используете предложение `ORDER BY` во избежание ненужных затрат.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как можно гарантировать последовательность строк в результате запроса?
2. В чем состоит разница между результатом запроса с предложением `ORDER BY` и без него?

Ответы на контрольные вопросы

1. Единственный способ — добавить предложение ORDER BY.
2. Без предложения ORDER BY результат будет реляционным (с точки зрения сортировки); при наличии предложения ORDER BY результат в принципе представляет собой то, что стандартно называется курсором.

ПРАКТИКУМ Сортировка данных

В данном практикуме вам предстоит применить знания о сортировке данных с помощью предложения ORDER BY.

Задание 1. Использование предложения ORDER BY с недетерминированной сортировкой

В этом задании вам предстоит использовать предложение ORDER BY для сортировки данных и попробовать свои силы в недетерминированной сортировке.

1. Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
2. Вам нужно написать запрос, который возвращает заказы для клиента с номером 77. Используйте следующий запрос:

```
SELECT orderid, empid, shipperid, shippeddate  
FROM Sales.Orders  
WHERE custid = 77;
```

Вы получите следующий результат:

orderid	empid	shipperid	shippeddate
10992	1	3	2008-04-03 00:00:00.000
10805	2	3	2008-01-09 00:00:00.000
10708	6	2	2007-11-05 00:00:00.000
10310	8	2	2006-09-27 00:00:00.000

Обратите внимание, поскольку вы не указали предложение ORDER BY, нет никакой гарантии, что строки будут возвращены в последовательности, указанной в предыдущем коде. Единственное, в чем можно быть уверенными, это то, что вы получите данный конкретный набор строк.

3. Вам нужно исправить ваш запрос таким образом, чтобы строки были отсортированы по значению shipperid. Добавьте предложение ORDER BY, как показано в примере.

```
SELECT orderid, empid, shipperid, shippeddate  
FROM Sales.Orders  
WHERE custid = 77  
ORDER BY shipperid;
```

Теперь запрос возвращает следующий результат:

orderid	empid	shipperid	shippeddate
10708	6	2	2007-11-05 00:00:00.000
10310	8	2	2006-09-27 00:00:00.000
10992	1	3	2008-04-03 00:00:00.000
10805	2	3	2008-01-09 00:00:00.000

Теперь вы можете гарантировать, что строки будут возвращены отсортированными по shipperid, но является ли сортировка детерминированной? Например, можете ли вы сказать с уверенностью, какой будет последовательность строк с одинаковым значением идентификатора shipperid? Ответ — нет.

Задание 2. Использование предложения ORDER BY с детерминированной сортировкой

В этом задании вам предстоит использовать предложение ORDER BY для упорядочения данных и попробовать себя в детерминированной сортировке.

1. Вы будете использовать запрос, который написали в п. 3 предыдущего задания. От вас требуется добавить вторичную сортировку по дате отгрузки в убывающем порядке. Добавьте shippeddate DESC в предложение ORDER BY, как написано далее.

```
SELECT orderid, empid, shipperid, shippeddate
FROM Sales.Orders
WHERE custid = 77
ORDER BY shipperid, shippeddate DESC;
```

Теперь запрос возвращает следующий результат:

orderid	empid	shipperid	shippeddate
10708	6	2	2007-11-05 00:00:00.000
10310	8	2	2006-09-27 00:00:00.000
10992	1	3	2008-04-03 00:00:00.000
10805	2	3	2008-01-09 00:00:00.000

В отличие от п. 3, сейчас есть гарантия, что строки с одним и тем же идентификатором грузоотправителя (shipperid) будут отсортированы по дате отгрузки в убывающем порядке. Является теперь сортировка детерминированной? Можете ли вы точно сказать, каким будет порядок строк с одинаковым идентификатором грузоотправителя и датой отгрузки? Ответ — по-прежнему нет, поскольку комбинация столбцов shipperid и shippeddate не уникальна, и неважно, к каким мыслям приводят вас текущие значения, которые вы видите в таблице. Формально в результате этого запроса может быть несколько строк с одинаковым значением shipperid и shippeddate.

2. Вам следует переписать запрос из п. 1 так, чтобы гарантировать детерминированную сортировку. Вам необходимо определить разрыв цепочки. Например,

укажите в качестве схемы разрешения конфликтов orderid DESC, как написано в следующем примере:

```
SELECT orderid, empid, shipperid, shippeddate
FROM Sales.Orders
WHERE custid = 77
ORDER BY shipperid, shippeddate DESC, orderid DESC;
```

Теперь при наличии связывания значений shipperid и shippeddate строка с большим значением orderid будет сортироваться первой.

Резюме занятия

- Запросы, как правило, возвращают реляционный результат там, где сортировка не гарантирована. Если вам необходимо гарантировать упорядочение представления, нужно в запросе добавить предложение ORDER BY.
- С помощью предложения ORDER BY можно указать список выражений для первичной сортировки, вторичной сортировки и т. д. Для каждого выражения можно указать параметры ASC и DESC для упорядочения по возрастанию или убыванию, при этом по умолчанию принимается сортировка по возрастанию.
- Даже если указано предложение ORDER BY, в результате все равно может получиться недетерминированная сортировка. Чтобы она была детерминированной, список ORDER BY должен быть уникальным.
- Можно использовать порядковые номера выражений из списка SELECT в предложении ORDER BY, но это считается плохой практикой.
- Можно выполнять сортировку по элементам, появляющимся в списке SELECT, если предложение DISTINCT также не определено.
- Поскольку считается, что предложение ORDER BY обрабатывается после предложения SELECT, можно ссылаться на псевдонимы, присвоенные в предложении SELECT внутри предложения ORDER BY.
- При сортировке SQL Server считает значения NULL ниже, чем значения не-NUL, и равными друг другу. Это означает, что при упорядочивании по возрастанию они сортируются все вместе перед не-NUL-маркерами.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Если в запросе отсутствует предложение ORDER BY, в каком порядке будутозвращены строки?
 - A. В произвольном порядке.
 - B. С сортировкой по первичному ключу.

- С. С сортировкой по кластерному индексу.
- D. В порядке размещения.
2. Вы хотите, чтобы результирующие строки были отсортированы по убыванию значения `orderdate` и затем по убыванию значения `orderid`. Какое из приведенных далее предложений даст желаемый результат?
- A. ORDER BY `orderdate, orderid DESC`.
- B. ORDER BY DESC `orderdate, DESC orderid`.
- C. ORDER BY `orderdate DESC, orderid DESC`.
- D. DESC ORDER BY `orderdate, orderid`.
3. Вы хотите, чтобы результирующие строки были отсортированы по возрастанию значения `orderdate` и затем по возрастанию значения `orderid`. Какие из приведенных далее предложений дадут желаемый результат? (Укажите все возможные варианты.)
- A. ORDER BY ASC(`orderdate, orderid`).
- B. ORDER BY `orderdate, orderid ASC`.
- C. ORDER BY `orderdate ASC, orderid ASC`.
- D. ORDER BY `orderdate, orderid`.

Занятие 3. Фильтрация данных с помощью предложений *TOP* и *OFFSET...FETCH*

Первое занятие было посвящено фильтрации данных, второе — сортировке данных. В этом занятии в некотором смысле смешиваются вопросы фильтрации и сортировки данных. Часто необходимо отфильтровать данные с учетом конкретной сортировки и указанного количества строк. Возьмем, к примеру, запросы "возвратить три самых последних заказа" и "возвратить пять самых дорогих продукта". Фильтр имеет некоторую спецификацию сортировки и требуемое количество строк. T-SQL предоставляет две возможности для решения таких задач фильтрации: первая — `TOP`, собственное предложение T-SQL, и вторая — стандартное предложение `OFFSET...FETCH`, которое было представлено в SQL Server 2012.

Изучив материал этого занятия, вы сможете:

- ✓ Фильтровать данные с помощью предложения `TOP`
- ✓ Фильтровать данные с помощью предложения `OFFSET...FETCH`

Продолжительность занятия — 30 минут.

Фильтрация данных с помощью предложения *TOP*

Предложение `TOP` позволяет фильтровать запрашиваемое количество или процент строк в результате запроса на основе указанной сортировки. Элемент `TOP` указыва-

ется в предложении `SELECT` перед запрашиваемым количеством строк в круглых скобках (тип данных `BIGINT`). Спецификация сортировки в фильтре `TOP` основывается на том же самом предложении `ORDER BY`, которое обычно используется для сортировки представления.

Например, следующий пример возвратит три самых последних заказа.

```
SELECT TOP (3) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

Здесь цифра 3 указана как число строк, которые нужно отфильтровать, а выражение `orderdate DESC` — как спецификация сортировки. Итак, вы получаете 3 строки с самыми последними датами заказа. Далее приведен результат этого запроса.

orderid	orderdate	custid	empid
11077	2008-05-06 00:00:00.000	65	1
11076	2008-05-06 00:00:00.000	9	4
11075	2008-05-06 00:00:00.000	68	8

ПРИМЕЧАНИЕ Параметр `TOP` и круглые скобки

Язык T-SQL поддерживает указание числа строк, которое надо отфильтровать, с помощью параметра `TOP` в запросах `SELECT` без использования скобок, но только лишь с целью обратной совместимости. Правильный синтаксис предполагает использование круглых скобок.

Вместо числа строк можно также указать процентное соотношение строк, которые надо отфильтровать. Для этого укажите величину `FLOAT` в диапазоне от 0 до 100 в скобках и ключевое слово `PERCENT` после скобок, как показано в следующем примере:

```
SELECT TOP (1) PERCENT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

Параметр `PERCENT` задает следующую целую часть результирующего количества строк, если это не целое число. В нашем примере без использования предложения `TOP` количество строк в результирующем наборе равно 830. Фильтрация 1% дает 8,3, и следующая целая часть этого числа равна 9; следовательно, этот запрос возвратит 9 строк.

orderid	orderdate	custid	empid
11076	2008-05-06 00:00:00.000	9	4
11077	2008-05-06 00:00:00.000	65	1
11075	2008-05-06 00:00:00.000	68	8
11074	2008-05-06 00:00:00.000	73	7
11070	2008-05-05 00:00:00.000	44	2
11071	2008-05-05 00:00:00.000	46	1
11073	2008-05-05 00:00:00.000	58	2

```
11072      2008-05-05 00:00:00.000  20      4
11067      2008-05-04 00:00:00.000  17      1
```

Предложение `TOP` не ограничено постоянным вводом, наоборот, оно позволяет указывать произвольное выражение. С практической точки зрения эта возможность особенно важна, когда необходимо передать параметр переменной в качестве входных данных, как в приведенном далее примере кода.

```
DECLARE @n AS BIGINT = 5;

SELECT TOP (@n) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

Этот запрос генерирует следующий результат:

orderid	orderdate	custid	empid
11076	2008-05-06 00:00:00.000	9	4
11077	2008-05-06 00:00:00.000	65	1
11075	2008-05-06 00:00:00.000	68	8
11074	2008-05-06 00:00:00.000	73	7
11070	2008-05-05 00:00:00.000	44	2

В большинстве случаев предложение `TOP` должно поддерживаться какой-либо спецификацией сортировки, но как оказывается, предложение `ORDER BY` не является обязательным. Например, следующий запрос является правильным технически.

```
SELECT TOP (3) orderid, orderdate, custid, empid
FROM Sales.Orders;
```

Однако этот запрос не является детерминированным. Он фильтрует 3 строки, но без всякой гарантии, какие именно три строки будут возвращены. Вы в итоге получите любые 3 строки, которые SQL Server выбрал первыми, и это зависит от оптимизации. Например, наш запрос дал следующий результат:

orderid	orderdate	custid	empid
11011	2008-04-09 00:00:00.000	1	3
10952	2008-03-16 00:00:00.000	1	1
10835	2008-01-15 00:00:00.000	1	1

Но нет гарантии, что те же самые строки будут возвращены, когда вы запустите этот запрос в следующий раз. Если вы действительно хотите получить три произвольных строки, стоит добавить предложение `ORDER BY` с выражением (`SELECT NULL`), чтобы дать знать другим людям, что это ваш осознанный выбор, а не недоразумение. Далее показано, как будет выглядеть ваш запрос.

```
SELECT TOP (3) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY (SELECT NULL);
```

Заметьте, что даже если вы используете предложение ORDER BY, чтобы запрос был полностью детерминированным, сортировка должна быть уникальной. Например, снова рассмотрим первый запрос из данного раздела.

```
SELECT TOP (3) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

Столбец orderdate не является уникальным, поэтому упорядочение при наличии связей будет произвольным. Когда этот запрос был запущен, система возвратила следующий результат:

orderid	orderdate	custid	empid
11077	2008-05-06 00:00:00.000	65	1
11076	2008-05-06 00:00:00.000	9	4
11075	2008-05-06 00:00:00.000	68	8

Но что если существуют другие строки в результирующем наборе без параметра TOP, которые имеют такую же дату заказа, что и последняя строка в нашем примере? Мы не всегда заботимся о том, чтобы результат был детерминированным или повторяемым, но если это необходимо, можно использовать две возможности. Одна — попросить включить все связи с последней строкой, добавив аргумент WITH TIES, как на примере далее.

```
SELECT TOP (3) WITH TIES orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

Конечно, в результате может быть возвращено больше строк, чем мы запрашивали, как показывает результат этого запроса.

orderid	orderdate	custid	empid
11077	2008-05-06 00:00:00.000	65	1
11076	2008-05-06 00:00:00.000	9	4
11075	2008-05-06 00:00:00.000	68	8
11074	2008-05-06 00:00:00.000	73	7

Другая возможность гарантировать детерминизм — разбить связи с помощью разрыва цепочки, который сделает сортировку уникальной. Например, в случае связей в дате заказа предположим, что вы хотели, чтобы строки с большим идентификатором заказа (orderid) "выиграли". Для этого добавьте выражение orderid DESC в предложение ORDER BY, как показано на следующем примере:

```
SELECT TOP (3) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC, orderid DESC;
```

Далее приведен результат этого запроса.

orderid	orderdate	custid	empid
11077	2008-05-06 00:00:00.000	65	1

11076	2008-05-06 00:00:00.000	9	4
11075	2008-05-06 00:00:00.000	68	8

Запрос теперь стал детерминированным, и гарантирована повторяемость запроса, до тех пор, пока базовые данные не изменятся.

В заключение этого раздела хотелось бы заметить, что предложение `TOP` можно также использовать в инструкциях модификации для ограничения количества манипулируемых строк, но модификации рассматриваются далее в этом учебном курсе.

Фильтрация данных с помощью `OFFSET...FETCH`

Конструкция `OFFSET...FETCH` — это конструкция фильтрации, которую, наподобие предложения `TOP`, можно использовать для фильтрации данных на основе указанного количества строк и упорядочения. Но в отличие от `TOP`, это стандартная конструкция, которая также имеет возможность пропуска данных, что делает ее полезной для оперативного листания.

Предложения `OFFSET` и `FETCH` появляются сразу после предложения `ORDER BY` и, фактически, в языке T-SQL они требуют присутствия предложения `ORDER BY`. Сначала указывается предложение `OFFSET`, определяя, сколько строк нужно пропустить (0, если ни одной); затем при желании указывается предложение `FETCH`, предписывающая, сколько строк надо отфильтровать. Например, следующий запрос задает сортировку на основе даты заказа по убыванию, далее — идентификатора заказа по убыванию, а затем он пропускает 50 строк и выбирает следующие 25 строк.

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC, orderid DESC
OFFSET 50 ROWS FETCH NEXT 25 ROWS ONLY;
```

Далее приведен сокращенный результат этого запроса.

orderid	orderdate	custid	empid
11027	2008-04-16 00:00:00.000	10	1
11026	2008-04-15 00:00:00.000	27	4
...			
11004	2008-04-07 00:00:00.000	50	3
11003	2008-04-06 00:00:00.000	78	3

Предложение `ORDER BY` теперь играет две роли: первая — сообщить параметру `OFFSET...FETCH`, какие строки надо отфильтровать. Другая роль — определение сортировки представления в запросе.

Как уже говорилось ранее, в языке T-SQL параметр `OFFSET...FETCH` требует присутствия предложения `ORDER BY`. Также в T-SQL — в противоположность стандартному SQL, — предложение `FETCH` требует присутствия предложения `OFFSET`. Поэтому, если вы хотите отфильтровать какие-либо строки и ничего не пропускать, надо все равно указать предложение `OFFSET` со значением `0 ROWS`.

Чтобы сделать синтаксис более понятным, можно использовать попеременно ключевые слова `NEXT` или `FIRST`. При пропуске некоторого количества строк может быть более интуитивно понятным использование ключевых слов `FETCH NEXT` для того, чтобы указать, сколько строк надо отфильтровать. Но если не надо пропускать строки, более интуитивно понятным может быть использование ключевого слова `FETCH FIRST`, как показано в следующем примере:

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC, orderid DESC
OFFSET 0 ROWS FETCH FIRST 25 ROWS ONLY;
```

В аналогичных целях можно использовать форму в единственном числе `ROW` или во множественном числе `ROWS` попеременно для указания числа строк, которые надо пропустить или отфильтровать. Не будет ошибкой, если вы напишете `FETCH NEXT 1 ROWS` или `FETCH NEXT 25 ROW`. Использование подходящей формы остается на ваше усмотрение, так же как в английском языке.

В T-SQL предложение `FETCH` требует присутствия предложения `OFFSET`, а предложение `OFFSET` не требует наличия предложения `FETCH`. Другими словами, указывая предложение `OFFSET`, вы просите пропустить некоторое количество строк; затем, указав предложение `FETCH`, вы просите вернуть все оставшиеся строки. Например, в следующем запросе требуется пропустить 50 строк, возвратив все остальные.

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC, orderid DESC
OFFSET 50 ROWS;
```

Далее приведен сокращенный вариант выходного набора данных.

orderid	orderdate	custid	empid
11027	2008-04-16 00:00:00.000	10	1
11026	2008-04-15 00:00:00.000	27	4
...			
10249	2006-07-05 00:00:00.000	79	6
10248	2006-07-04 00:00:00.000	85	5

(780 row(s) affected)

Как уже ранее упоминалось, конструкция `OFFSET...FETCH` требует присутствия предложения `ORDER BY`. Но что, если нужно отфильтровать определенное количество строк в произвольном порядке? Для этого можно указать выражение `(SELECT NULL)` в предложении `ORDER BY`, как показано на примере далее.

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY (SELECT NULL)
OFFSET 0 ROWS FETCH FIRST 3 ROWS ONLY;
```

Этот код просто отфильтровывает три произвольных строки. Далее приведен результат, полученный при запуске этого кода в системе.

orderid	orderdate	custid	empid
11011	2008-04-09 00:00:00.000	1	3
10952	2008-03-16 00:00:00.000	1	1
10835	2008-01-15 00:00:00.000	1	1

С помощью предложений `OFFSET` и `FETCH` можно в качестве входных данных использовать выражения. Это очень удобно, когда требуется динамически вычислять входные значения. Например, представьте, что вы реализуете возможность постраничного просмотра, где пользователю возвращается одна страница строк за один раз. Пользователь отправляет вашей процедуре или функции в качестве входных параметров номер страницы (`@pagenum parameter`) и размер страницы (`@pagesize parameter`). Это означает, что вам нужно пропустить количество строк, равное `@pagenum` минус 1, умноженное на `@pagesize`, и выбрать следующие `@pagesize` строк. Реализовать это можно с помощью следующего кода (с использованием локальных переменных для простоты):

```
DECLARE @pagesize AS BIGINT = 25, @pagenum AS BIGINT = 3;

SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC, orderid DESC
OFFSET (@pagenum - 1) * @pagesize ROWS FETCH NEXT @pagesize ROWS ONLY;
```

Используя эти входные данные, код возвратит следующий результат:

orderid	orderdate	custid	empid
10477	2007-03-17 00:00:00.000	60	5
10476	2007-03-17 00:00:00.000	35	8
...			
10454	2007-02-21 00:00:00.000	41	4
10453	2007-02-21 00:00:00.000	4	1

(25 row(s) affected)

Вы можете изменить входные значения по своему усмотрению и посмотреть, как изменится результат.

Поскольку конструкция `OFFSET...FETCH` является стандартной, а `TOP` — нет, в случаях, когда они логически эквивалентны, рекомендуется использовать более привычный. Кроме того, `OFFSET...FETCH` имеет преимущество перед параметром `TOP` — он поддерживает возможность пропуска данных. Однако `OFFSET...FETCH` не поддерживает имеющиеся у `TOP` возможности, такие как `PERCENT` и `WITH TIES`.

С точки зрения производительности, необходимо оценить индексацию столбцов `ORDER BY` для поддержки конструкций `TOP` и `OFFSET...FETCH`. Такая индексация служит очень простой цели — индексировать фильтруемые столбцы, — и может помочь избежать сканирования ненужных данных, а также сортировки.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как гарантировать детерминированные результаты с помощью конструкции TOP?
2. Каковы преимущества использования OFFSET...FETCH по сравнению с TOP?

Ответы на контрольные вопросы

1. Либо возвращая все связи с помощью параметра WITH TIES, либо с использованием уникальной сортировки для разрыва связей.
2. Конструкция OFFSET...FETCH является стандартной, тогда как TOP — нет; кроме того, OFFSET...FETCH поддерживает возможность пропуска данных, а TOP — нет.

**ПРАКТИКУМ Фильтрация данных
с помощью TOP и OFFSET...FETCH**

В этом практикуме вам предстоит проверить ваши знания о фильтрации данных с помощью конструкций TOP и OFFSET...FETCH.

Задание 1. Использование конструкции TOP

В этом задании вы будете использовать конструкцию TOP для фильтрации данных.

1. Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
2. Перед вами поставлена задача написать запрос к таблице Production.Products, возвращающий 5 наиболее дорогих продуктов 1-й категории. Напишите следующий запрос:

```
SELECT TOP (5) productid, unitprice
FROM Production.Products
WHERE categoryid = 1
ORDER BY unitprice DESC;
```

Вы получите следующий результирующий набор:

Productid	unitprice
38	263.50
43	46.00
2	19.00
1	18.00
35	18.00

Запрос возвращает нужный результат, за исключением того, что нет никакой обработки связей. Иными словами, упорядочение продуктов с одинаковой ценой не является детерминированным.

3. Вас просят представить решения для превращения предыдущего запроса в детерминированный: одно решение с использованием связей, другое — с разрывом связей. Во-первых, рассмотрите вариант, который включает все связи, с помощью параметра WITH TIES. Добавьте этот параметр следующим образом.

```
SELECT TOP (5) WITH TIES productid, unitprice
FROM Production.Products
WHERE categoryid = 1
ORDER BY unitprice DESC;
```

Вы получите следующий результат, включающий связи.

productid	unitprice
38	263.50
43	46.00
2	19.00
1	18.00
39	18.00
35	18.00
76	18.00

- Используйте следующий вариант, который разрывает связи, используя параметр `productid` в убывающем порядке.

```
SELECT TOP (5) productid, unitprice
FROM Production.Products
WHERE categoryid = 1
ORDER BY unitprice DESC, productid DESC;
```

Этот запрос дает следующий результат:

productid	unitprice
38	263.50
43	46.00
2	19.00
76	18.00
39	18.00

Задание 2. Использование конструкции `OFFSET...FETCH`

В этом задании вы попробуете свои силы в использовании конструкции `OFFSET...FETCH` для фильтрации данных.

- Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQSL2012.
- Вам необходимо написать несколько запросов, которые просматривают продукты, по 5 за один раз, с сортировкой по цене единицы товара, используя идентификатор продукта (`productid`) для разрыва соединений. Начните с написания запроса, который возвращает пять первых продуктов.

```
SELECT productid, categoryid, unitprice
FROM Production.Products
ORDER BY unitprice, productid
OFFSET 0 ROWS FETCH FIRST 5 ROWS ONLY;
```

Вы могли бы использовать либо ключевое слово FIRST, либо NEXT, предположим, что вы решили использовать ключевое слово FIRST, поскольку это более естественно, если не надо пропускать строки. Этот запрос генерирует следующий результат:

productid	categoryid	unitprice
33	4	2.50
24	1	4.50
13	8	6.00
52	5	7.00
54	6	7.45

3. Далее, напишите запрос, который возвращает следующие 5 строк (с 6 по 10), используя следующий пример:

```
SELECT productid, categoryid, unitprice
FROM Production.Products
ORDER BY unitprice, productid
OFFSET 5 ROWS FETCH NEXT 5 ROWS ONLY;
```

На этот раз используйте ключевое слово NEXT, поскольку вам нужно пропустить несколько строк. Этот запрос генерирует следующий результат:

productid	categoryid	unitprice
75	1	7.75
23	5	9.00
19	3	9.20
45	8	9.50
47	3	9.50

4. Аналогично, напишите запрос, который возвращает строки с 11 по 15:

```
SELECT productid, categoryid, unitprice
FROM Production.Products
ORDER BY unitprice, productid
OFFSET 10 ROWS FETCH NEXT 5 ROWS ONLY;
```

Этот запрос генерирует следующий результат:

productid	categoryid	unitprice
41	8	9.65
3	2	10.00
21	3	10.00
74	7	10.00
46	8	12.00

То же самое надо сделать для последующих строк.

Резюме занятия

- С помощью конструкций `TOP` и `OFFSET...FETCH` можно фильтровать даты на основе указанного количества строк и сортировки.
- Предложение `ORDER BY`, которое обычно используется в запросе для сортировки представления, также используется конструкциями `TOP` и `OFFSET...FETCH`, чтобы указать, какие строки надо фильтровать.
- Параметр `TOP` — собственная возможность языка T-SQL, которую можно использовать для указания количества или процентного соотношения строк, подлежащих фильтрации.
- Можно сделать запрос `TOP` детерминированным двумя способами: первый — используя параметр `WITH TIES` для возвращения всех связей, второй — используя уникальную сортировку для разрыва связей.
- `OFFSET...FETCH` — это стандартная конструкция, подобная параметру `TOP`, поддерживаемая SQL Server 2012. В отличие от `TOP` она позволяет задать количество строк, которые надо пропустить, прежде чем указать число строк, которое следует отфильтровать. Поэтому она может использоваться для оперативной разбивки данных на страницы.
- Как `TOP`, так и `OFFSET...FETCH` поддерживают в качестве входных данных выражения, а не только константы.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Вы выполняете запрос с выражением `TOP (3)`. Какой из следующих вариантов наиболее точно описывает, сколько строк будет возвращено?
 - A. Меньше трех строк.
 - B. Три строки или менее.
 - C. Три строки.
 - D. Три строки или более.
 - E. Более трех строк.
 - F. Меньше трех, три или более трех строк.
2. Вы выполняете запрос с выражением `TOP (3) WITH TIES` и неуникальной сортировкой. Какой из следующих вариантов наиболее точно описывает, сколько строк будет возвращено?
 - A. Меньше трех строк.
 - B. Три строки или менее.
 - C. Три строки.

- D. Три строки или более.
- E. Более трех строк.
- F. Меньше трех, три или более трех строк.
3. Какое из следующих `OFFSET...FETCH` выражений корректно в T-SQL? (Укажите все подходящие варианты.)
- A. `SELECT ... ORDER BY orderid OFFSET 25 ROWS.`
- B. `SELECT ... ORDER BY orderid FETCH NEXT 25 ROWS ONLY.`
- C. `SELECT ... ORDER BY orderid OFFSET 25 ROWS FETCH NEXT 25 ROWS ONLY.`
- D. `SELECT ... <no ORDER BY> OFFSET 0 ROWS FETCH FIRST 25 ROWS ONLY.`

Упражнения

В следующих упражнениях вы примените полученные знания фильтрации и сортировки данных. Ответы на вопросы можно найти в *приложении "Ответы" в конце книги*.

Упражнение 1. Рекомендации по улучшению производительности фильтрации и сортировки

Вы приглашены в качестве консультанта на пивоваренный завод, использующий SQL Server 2012, чтобы помочь решить проблемы с производительностью. Вы выполняете трассировку обычной нагрузки на систему и обнаруживаете очень медленное выполнение запросов. Вы видите очень большую нагрузку на сеть. Вы видите, что множество запросов возвращает все строки клиенту и затем клиент выполняет фильтрацию. Запросы, которые фильтруют данные, часто выполняют обработку фильтруемых столбцов. Все запросы содержат предложение `ORDER BY`, и когда вы об этом спрашиваете, вам говорят, что в действительности это не требуется, но разработчики привыкли так делать — на всякий случай. Вы обнаружили множество дорогостоящих операций сортировки. Клиент хочет получить от вас рекомендации по улучшению производительности и задает вам следующие вопросы:

1. Можно ли сделать что-нибудь, чтобы улучшить способ выполнения сортировки?
2. Наносит ли какой-либо ущерб использование `ORDER BY`, даже если данные не обязательно должны возвращаться в отсортированном виде?
3. Дайте, пожалуйста, рекомендации по использованию запросов с конструкциями `TOP` и `OFFSET...FETCH`?

Упражнение 2. Обучение разработчика-стажера

Вы обучаете разработчика-стажера фильтрации и сортировке данных на языке T-SQL. Разработчику не все понятно, и он задает вам вопросы. Ответьте на следующие вопросы, используя все свои знания:

1. Когда я пытаюсь сослаться на псевдоним столбца, который определил в списке `SELECT` в предложении `WHERE`, то получаю ошибку. Объясните, пожалуйста, почему это запрещено и как разрешить эту проблему?
2. Кажется, ссылка на псевдоним столбца в предложении `ORDER BY` поддерживается. Почему?
3. Почему так получилось, что компания Microsoft сделала обязательным указывать предложение `ORDER BY` при использовании конструкции `OFFSET...FETCH`, но не при использовании `TOP`? Означает ли это, что только запросы `TOP` могут иметь недетерминированную сортировку?

Рекомендуемые упражнения

Для успешного освоения экзаменационных тем, представленных в данной главе, выполните следующие задания.

Перечисление этапов логической обработки запросов и сравнение фильтров

Для тренировки применения знаний о логической обработке запросов перечислите элементы, которые вы уже изучили, в правильной последовательности.

- **Задание 1.** В этой главе вы изучили использование предложения `WHERE` для фильтрации данных с помощью предикатов, предложение `ORDER BY` для сортировки данных, и конструкции `TOP` и `OFFSET...FETCH` как другой способ фильтрации данных. Объединив свои знания с полученными в главе 1, перечислите элементы запроса `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`, `TOP` и `OFFSET...FETCH` в правильном порядке логической обработки запросов. Помните, что поскольку `TOP` и `OFFSET...FETCH` не могут использоваться вместе в одном запросе, вам нужно создать два таких списка.
- **Задание 2.** Перечислите возможности, которые имеет фильтр `OFFSET...FETCH` и недоступны для конструкции `TOP` в SQL Server 2012, и наоборот.

Что такое детерминизм?

Вспомните, что детерминированный запрос — это запрос, который имеет только один правильный результат. Чтобы продемонстрировать свои знания детерминизма в запросах, приведите примеры детерминированных и недетерминированных запросов.

- **Задание 1.** Приведите примеры запросов с детерминированной и недетерминированной сортировкой. Опишите своими словами, что требуется для получения детерминированной сортировки данных.
- **Задание 2.** Приведите примеры детерминированных и недетерминированных запросов с использованием конструкций `TOP` и `OFFSET...FETCH`. Объясните, как можно обеспечить детерминизм в обоих случаях.

ГЛАВА 4

Комбинирование наборов данных

Темы экзамена

- Работа с данными.
 - Запрос данных с помощью предложения `SELECT`.
 - Реализация подзапросов.
- Модификация данных.

T-SQL предоставляет разные способы для комбинирования данных из нескольких таблиц, которые рассмотрены в данной главе. Мы обсудим соединения (*joins*), подзапросы, табличные выражения, оператор `APPLY` и операторы работы с наборами.

ПРЕЖДЕ ВСЕГО

Для полного освоения этой главы необходимо иметь:

- опыт работы с SQL Server Management Studio (SSMS);
- некоторый опыт в написании кода на языке T-SQL;
- доступ к экземпляру SQL Server 2012 с установленной учебной базой данных TSQl2012;
- понимание фильтрации и сортировки данных.

Кроме того, прежде чем запускать запросы в данной главе, добавьте нового поставщика в таблицу `Production.Suppliers`, запустив следующий код:

```
USE TSQl2012;
INSERT INTO Production.Suppliers (companyname, contactname, contacttitle,
    address, city, postalcode, country, phone)
VALUES (N'Supplier XYZ', N'Jiru', N'Head of Security',
    N'42 Sekimai Musashino-shi', N'Tokyo', N'01759', N'Japan',
    N'(02) 4311-2609');
```

Этот поставщик не имеет связанных с ним продуктов в таблице `Production.Products` и используется в примерах для демонстрации отсутствия соответствий.

Занятие 1. Использование соединений

Часто данные, которые нужно запросить, распределены по нескольким таблицам. Чем более нормализованной является среда, тем более таблиц, как правило, вы используете. Обычно таблицы связаны по ключам, таким как первичный ключ на одной стороне и первичный ключ на другой. Вы также можете использовать соединения, чтобы запрашивать данные из разных таблиц и сопоставлять строки, которые должны быть связаны. В этой главе мы обсудим различные виды соединений, поддерживаемых в языке T-SQL: пересечение, внутреннее соединение и внешнее соединение.

Изучив материал этого занятия, вы сможете:

- ✓ Составлять запросы, которые используют перекрестные объединения, внутренние соединения и внешние соединения
- ✓ Объяснять разницу между предложениями `ON` и `WHERE`
- ✓ Составлять запросы, которые сочетают в себе разные типы соединений
- ✓ Выполнять фильтрацию значений даты и времени

Продолжительность занятия — 60 минут.

Перекрестные соединения

Перекрестное соединение (cross join) — это простейший и поэтому не самый используемый тип соединения. Это соединение выполняет то, что называют *декартовым произведением* двух входных таблиц. Иными словами, он выполняет умножение двух таблиц, создавая строку для каждой комбинации строк из обеих таблиц. Если в таблице T_1 имеется m строк, а в таблице T_2 — n строк, результатом перекрестного соединения этих таблиц будет виртуальная таблица с $m \times n$ строками. На рис. 4.1 показан пример перекрестного соединения таблиц.



В левой таблице три строки со значениями ключей `A`, `B` и `C`. В правой таблице — четыре строки со значениями ключей `B1`, `C1`, `C2` и `D1`. Результатом является таблица из 12 строк, содержащая все возможные комбинации строк из этих двух входных таблиц.

Рассмотрим пример из учебной базы данных TSQL2012. Эта база данных включает в себя таблицу `dbo.Nums`, которая содержит столбец `n` с последовательностью целых чисел, начиная с 1. Ваша задача — использовать эту таблицу для генерации результирующей таблицы со строкой для каждого дня недели (1—7) и номера смены (1—3) при условии трех смен в день. В дальнейшем этот результат может использоваться для предоставления информации о видах работ в разных сменах в различные дни недели. При условии 7 дней в неделе и 3-х смен в день результат должен содержать 21 строку.

Далее приведен запрос, решающий эту задачу с помощью перекрестного объединения двух экземпляров таблицы `Nums` — одно представляет дни недели (и имеет псевдоним `D`), а другое представляет смены (с псевдонимом `S`).

```

SELECT D.n AS theday, S.n AS shiftno
FROM dbo.Nums AS D
CROSS JOIN dbo.Nums AS S
WHERE D.n <= 7
AND S.N <= 3
ORDER BY theday, shiftno;

```

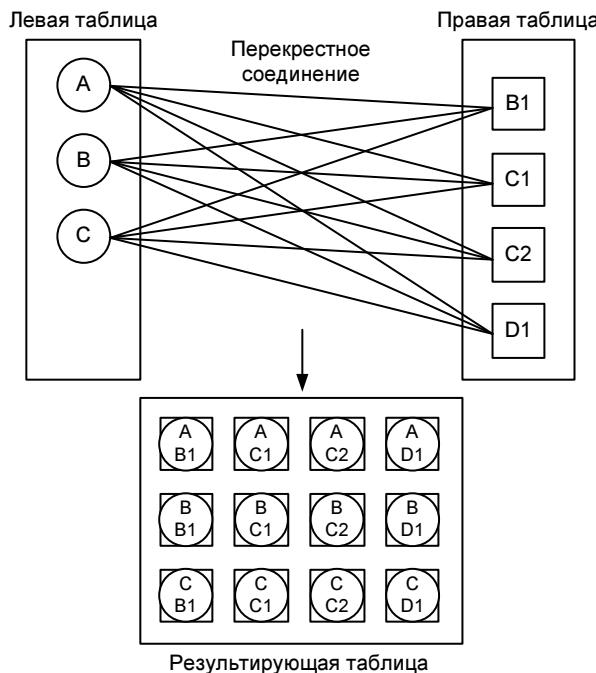


Рис. 4.1. Перекрестное объединение таблиц

А вот как выглядит результат этого запроса.

theday	shiftno
1	1
1	2
1	3
2	1
2	2
2	3
3	1
3	2
3	3
4	1
4	2
4	3
5	1
5	2

5	3
6	1
6	2
6	3
7	1
7	2
7	3

Таблица `Tums` содержит 100 строк. В соответствии с логической обработкой запросов, первым шагом обработки нашего запроса является оценка предложения `FROM`. Перекрестное объединение действует в предложении `FROM`, выполняя декартово перемножение двух экземпляров `Nums` и создавая таблицу с 10 млрд строк (не стоит беспокоиться, это чисто концептуально). Затем предложение `WHERE` отфильтровывает только строки, в которых столбец `D.n` меньше или равен 7 и столбец `S.n` меньше или равен 3. После применения фильтра результат имеет 21 удовлетворяющую условиям строку. Затем предложение `SELECT` возвращает столбец `D.n`, назвав его `theday`, и столбец `S.n`, назвав его `shiftno`.

К счастью, SQL Server не обязан буквально следовать логической обработке запросов, пока он возвращает правильный результат. Именно это является задачей оптимизации — возвращать результат как можно быстрее. SQL Server знает, что если перекрестное объединение стоит перед фильтром, сначала можно оценить фильтры (что особенно эффективно, когда фильтры поддерживаются индексами), а затем сопоставлять оставшиеся строки.

Обратите внимание на важность использования псевдонимов таблиц в данном объединении. С одной стороны, удобно ссылаться на таблицу, используя более короткое имя. Но для самосоединения, как в нашем случае, использование псевдонимов столбцов обязательно. Если не присвоить разные псевдонимы двум экземплярам одной таблицы, в конце концов, будет получен неправильный результат, поскольку будут существовать двойные имена столбцов, даже если имя таблицы включить в имя столбца в качестве префикса.

Давая столбцам разные псевдонимы, можно однозначно ссылаться на столбцы, используя формат `псевдоним_таблицы.имя_столбца`, как в случае с `D.n` и `S.n`.

Также следует заметить, что в дополнение к поддержке синтаксиса перекрестных соединений с ключевыми словами `CROSS JOIN`, и стандартный язык SQL, и T-SQL поддерживают более старый синтаксис, где между именами таблиц ставится запятая, как в случае `FROM T1, T2`. Однако по многим причинам рекомендуется придерживаться современного синтаксиса; это позволит уменьшить количество ошибок и писать более согласованный код.

Внутренние соединения

С помощью *внутренних соединений* можно сопоставлять строки из двух таблиц по предикату, как правило, сравнивая значение первичного ключа в одной таблице с внешним ключом в другой. На рис. 4.2 показан пример внутреннего соединения.



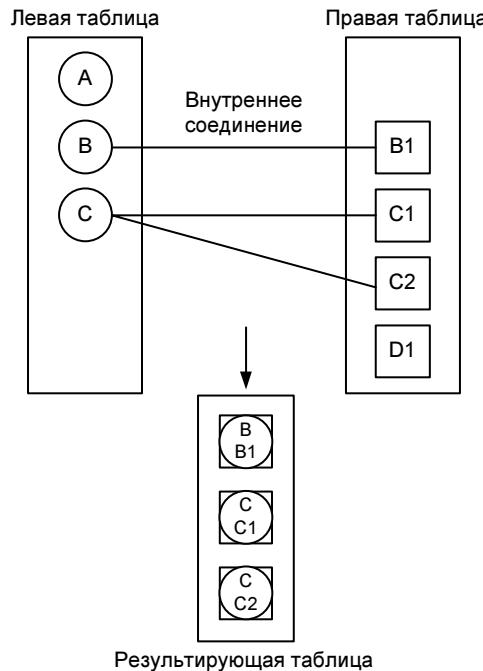


Рис. 4.2. Внутреннее соединение

Буквами обозначены первичный ключ в левой таблице и внешний ключ в правой таблице. С учетом того, что данное соединение является объединением по эквивалентности (используя предикат с оператором равенства, как в случае `lefttable.keycol = righttable.keycol`), внутреннее соединение возвращает только совпадающие строки, для которых предикат принимает значение "истина". Строки, для которых предикат принимает значение "ложь" или неизвестное значение, отбрасываются.

Например, следующий пример возвращает поставщиков из Японии (`Japan`) и продукты, которые они поставляют.

```
SELECT S.companyname AS supplier, S.country,
       P.productid, P.productname, P.unitprice
  FROM Production.Suppliers AS S
    INNER JOIN Production.Products AS P
      ON S.supplierid = P.supplierid
 WHERE S.country = N'Japan';
```

Далее приведен результат этого запроса.

supplier	country	productid	productname	unitprice
Supplier QOVFD	Japan	9	Product AOZBW	97.00
Supplier QOVFD	Japan	10	Product YHXGE	31.00
Supplier QOVFD	Japan	74	Product BKAZJ	10.00
Supplier QWUSF	Japan	13	Product POXFU	6.00

Supplier QWUSF	Japan	14	Product PWCJB	23.25
Supplier QWUSF	Japan	15	Product KSZOI	15.50

Обратите внимание, предикат совпадения соединения указан в предложении `ON`. Он сопоставляет поставщиков и продукты, имеющие одинаковые ID поставщика. Строки с любой стороны, не находящие соответствие в другой таблице, отбрасываются. Например, поставщики из Японии, которым не соответствует ни один продукт, отбрасываются.

СОВЕТ Подготовка к экзамену

Часто таблицы объединяют по внешнему ключу. Например, имеется внешний ключ, определенный для столбца `supplierid` в таблице `Production.Products` (ссылающаяся таблица). Также важно помнить, что когда определяется первичный ключ или уникальная константа, SQL Server создает уникальный индекс для столбцов ограничения, чтобы обеспечить свойство уникальности ограничения. Но при определении внешнего ключа SQL Server не создает никаких индексов на столбцах внешнего ключа. Такие индексы способны улучшить производительность соединений по их отношениям. Поскольку SQL Server не создает такие индексы автоматически, вы должны самостоятельно определить, где они могут быть полезны, и создать их. Таким образом, при настройке индексов следует проанализировать столбцы внешнего ключа и оценить преимущества создания на них индексов.

Что касается последнего запроса, еще раз обращаем ваше внимание на удобство использования коротких псевдонимов таблиц, когда требуется ссылаться на неоднозначные имена столбцов, такие как `supplierid`. Заметьте, запрос использует псевдонимы столбцов даже перед однозначными именами столбцов, такими как `s.country`. Это необязательно, если имя столбца однозначно, но даже в таком случае считается хорошим тоном с точки зрения понятности.

Часто задают вопрос: "В чем разница между предложениями `ON` и `WHERE`, и существует ли разница, в каком предложении указывают предикат?" Для внутренних соединений — нет, не существует. Оба предложения выполняют одну и ту же задачу фильтрации данных. Оба фильтруют только строки, для которых предикат принимает значение "истина", и отбрасываются строки, предикат для которых принимает значение "ложь" или "неизвестно". С точки зрения логической обработки запроса, предложение `WHERE` оценивается сразу после предложения `FROM`, так что по существу это эквивалентно объединению предикатов с помощью оператора `AND`. SQL Server знает это и поэтому может сам изменить последовательность, в которой он оценивает предикаты на практике, причем это очень зависит от оценки затрат ресурсов.

Принимая во внимание все сказанное, вы можете следующим образом изменить расположение предикатов в предыдущем запросе, указав оба в предложении `ON`; исходный смысл его от этого не изменится.

```
SELECT S.companyname AS supplier,
       S.country,
       P.productid, P.productname, P.unitprice
  FROM Production.Suppliers AS S
 INNER JOIN Production.Products AS P
    ON S.supplierid = P.supplierid
   AND S.country = N'Japan';
```

Однако для многих людей вполне естественно указывать предикат, по которому сравниваются столбцы с обеих сторон, в предложении `ON`, а предикат, который

фильтрует столбцы на одной стороне, — в предложении WHERE. Но, как мы уже говорили, для внутренних соединений это не имеет значения. При обсуждении внешних соединений в следующем разделе вы увидите, что в этом случае предложения ON и WHERE играют разные роли; там нужно, в зависимости от потребностей, выяснить, какое предложение подходит для каждого из предикатов.

Другой пример использования внутреннего соединения представлен в следующем запросе, который объединяет два экземпляра таблицы HR.Employees для того, чтобы найти соответствие между сотрудниками и их руководителями (руководитель тоже является сотрудником, следовательно, возможно самосоединение).

```
SELECT E.empid,
       E.firstname + N' ' + E.lastname AS emp,
       M.firstname + N' ' + M.lastname AS mgr
  FROM HR.Employees AS E
 INNER JOIN HR.Employees AS M
    ON E.mgrid = M.empid;
```

В результате запроса получим следующее:

empid	emp	mgr
2	Don Funk	Sara Davis
3	Judy Lew	Don Funk
4	Yael Peled	Judy Lew
5	Sven Buck	Don Funk
6	Paul Suurs	Sven Buck
7	Russell King	Sven Buck
8	Maria Cameron	Judy Lew
9	Zoya Dolgopyatov	Sven Buck

Обратите внимание на предикат соединения: ON E.mgrid = M.empid. Экземпляр сотрудника имеет псевдоним E, а менеджер — псевдоним M. Для нахождения правильных совпадений, ID менеджера конкретного сотрудника должен быть равен ID сотрудника этого менеджера.

Только 8 строк было возвращено, хотя всего в таблице 9 строк. Причина в том, что президент (Sara Davis с ID = 1) не имеет менеджера, и поэтому ее столбец mgrid имеет значение NULL. Помните, что внутреннее соединение не возвращает строки, для которых не находятся совпадения.

Как и в случае перекрестного соединения, и стандартный SQL, и T-SQL поддерживают прежний синтаксис внутренних соединений, где между именами таблиц ставится запятая, а затем все предикаты в предложении WHERE. Но как уже говорилось, считается хорошим тоном придерживаться нового синтаксиса с использованием ключевого слова JOIN. При использовании старого синтаксиса, если вы забыли указать предикат соединения, в конечном итоге вы получите незапланированное перекрестное соединение. Если же используется новый синтаксис, внутреннее соединение синтаксически некорректно без предложения ON, так что если вы забудете указать предикат соединения, парсер (синтаксический анализатор) выдаст ошибку.

Поскольку внутреннее соединение — наиболее часто используемый тип соединения, стандарт языка сделал его значением по умолчанию, если указано только ключевое слово `JOIN`. Поэтому выражение `T1 JOIN T2` эквивалентно выражению `T1 INNER JOIN T2`.



Внешние соединения

С помощью *внешнего соединения* можно запросить сохранение всех строк из одной или обеих сторон соединения без учета того, имеются ли соответствующие строки в другой стороне. Это осуществляется с помощью предиката `ON`.

Ключевые слова `LEFT OUTER JOIN` (или кратко `LEFT JOIN`) используются, чтобы сохранить левую таблицу. Такое соединение возвращает то же самое, что обычно возвращает внутреннее соединение, т. е. совпадения (назовем их внутренними строками). Кроме этого, соединение также возвращает строки из левой части, которые не имеют совпадений в правой части (назовем их внешними строками) со значениями `NULL` в качестве заполнителей в правой таблице. На рис. 4.3 показан пример левого внешнего соединения.

В отличие от внутреннего соединения, левая строка с ключом `A` возвращена, хотя она не имеет совпадений в правой части. Она возвращена со значением `NULL` в качестве заполнителя в правой части.

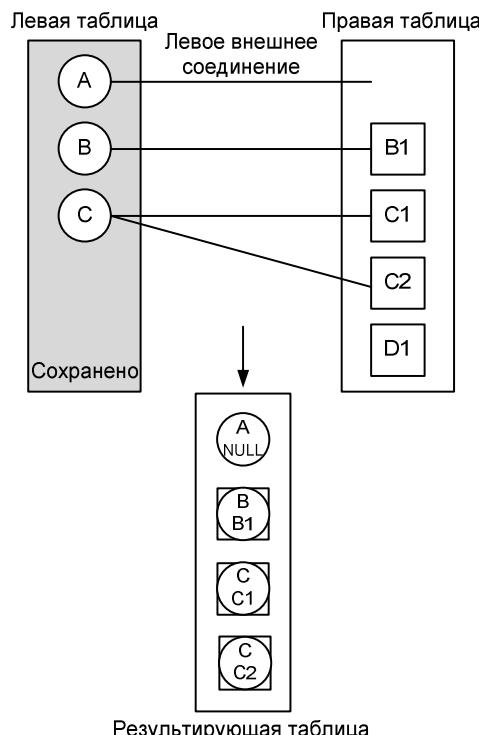


Рис. 4.3. Левое внешнее соединение

Рассмотрим запрос, который возвращает поставщиков из Японии и продукты, которые они поставляют, включая поставщиков из Японии, не имеющих соответствующих им продуктов.

```
SELECT S.companyname AS supplier,
       P.productid, P.productname, P.unitprice
  FROM Production.Suppliers AS S
 LEFT OUTER JOIN Production.Products AS P
    ON S.supplierid = P.supplierid
 WHERE S.country = N'Japan';
```

В результате запроса получим следующее:

supplier	country	productid	productname	unitprice
Supplier QOVFD	Japan	9	Product AOZBW	97.00
Supplier QOVFD	Japan	10	Product YHXGE	31.00
Supplier QOVFD	Japan	74	Product BKAZJ	10.00
Supplier QWUSF	Japan	13	Product POXFU	6.00
Supplier QWUSF	Japan	14	Product PWCJB	23.25
Supplier QWUSF	Japan	15	Product KSZOI	15.50
Supplier XYZ	Japan	NULL	NULL	NULL

Поскольку таблица Production.Suppliers является сохраняемой стороной запроса, поставщик XYZ возвращен, хотя он не имеет соответствующих ему продуктов. Как вы помните, внутреннее соединение отбросило этого поставщика.

Важно понимать, что в случае внешнего соединения предложения `ON` и `WHERE` играют разные роли и поэтому не являются взаимозаменяемыми. Предложение `WHERE` по-прежнему играет роль фильтра — а именно, оно сохраняет строки, дающие значение "истина", и отбрасывает строки, дающие значение "ложь" или "неизвестно". Так, в нашем запросе предложение `WHERE` фильтрует только поставщиков из Японии, поэтому поставщики из других стран просто не попадают в выходной набор.

Однако предложение `ON` не является просто фильтром, напротив, его основная задача — сопоставление данных. Иными словами, строка из сохраненной стороны будет возвращена независимо от того, найдет предикат `ON` совпадение или нет. Таким образом, предикат `ON` только определяет, какие строки из несохраненной стороны совпадают со строками в сохраненной таблице, но не определяет, возвращать эти строки или нет. В нашем запросе предложение `ON` сопоставляет строки с обеих сторон путем сравнения значений `supplierid` в них. Так как это предикат сопоставления (в противоположность фильтру), объединение не отбрасывает поставщиков, вместо этого оно только определяет, какие продукты соответствуют каждому поставщику. Но даже в соответствии с предикатом `ON` поставщик не имеет соответствий, он все равно будет возвращен. Другими словами, по отношению к сохраненной стороне соединения, предложение `ON` не является конечным; конечным будет предложение `WHERE`. Поэтому если вы сомневаетесь, указывать предикат в предложении `ON` или `WHERE`, задайте себе вопрос: для чего будет использоваться предикат — для фильтрации или сопоставления? Будет ли он конечным или нет?

Как вы думаете, что произойдет, если вы укажете и предикат, который сравнивает ID поставщика с обеих сторон, и предикат, сравнивающий страну поставщика со значением Japan в предложении ON? Попробуйте сделать так:

```
SELECT S.companyname AS supplier, S.country,
       P.productid, P.productname, P.unitprice
  FROM Production.Suppliers AS S
    LEFT OUTER JOIN Production.Products AS P
      ON S.supplierid = P.supplierid
     AND S.country = N'Japan';
```

Посмотрите, чем отличается результат (показанный здесь в сокращенном виде), и можете ли вы объяснить своими словами, что запрос возвратил в этом случае.

supplier	country	productid	productname	unitprice
Supplier SWRXU	UK	NULL	NULL	NULL
Supplier VHQZD	USA	NULL	NULL	NULL
Supplier STUAZ	USA	NULL	NULL	NULL
Supplier QOVFD	Japan	9	Product AOZBW	97.00
Supplier QOVFD	Japan	10	Product YHXGE	31.00
Supplier QWUSF	Japan	15	Product BKAZJ	10.00
Supplier EQPNC	Spain	NULL	NULL	NULL
...				

Теперь, когда оба предиката появились в предложении ON, они занимаются сопоставлением данных. Это означает, что возвращены все поставщики — даже не из Японии. Но для того, чтобы сопоставить продукт с поставщиком, ID поставщика на обеих сторонах должны совпадать, а страна поставщика должна быть Японией.

Вернемся к запросу, который сопоставлял сотрудников и их руководителей: вспомните, внутреннее соединение удалило строку со сведениями о президенте компании, поскольку не был найден соответствующий менеджер. Если нужно включить строку для президента компании, следует использовать внешнее соединение, сохраняющее сторону, которая представляет сотрудников (E) следующим образом:

```
SELECT E.empid,
       E.firstname + N' ' + E.lastname AS emp,
       M.firstname + N' ' + M.lastname AS mgr
  FROM HR.Employees AS E
    LEFT OUTER JOIN HR.Employees AS M
      ON E.mgrid = M.empid;
```

На этот раз результат запроса содержит строку о президенте компании.

empid	emp	mgr
1	Sara Davis	NULL
2	Don Funk	Sara Davis
3	Judy Lew	Don Funk
4	Yael Peled	Judy Lew

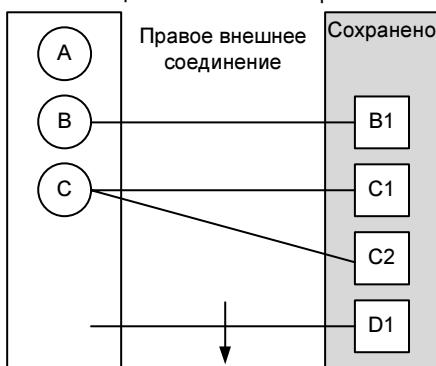
5	Sven Buck	Don Funk
6	Paul Suurs	Sven Buck
7	Russell King	Sven Buck
8	Maria Cameron	Judy Lew
9	Zoya Dolgopyatov	Sven Buck

Так же как левое внешнее соединение можно использовать для сохранения левой стороны, правое внешнее соединение можно использовать для сохранения правой стороны. Для этого надо указать ключевые слова `RIGHT OUTER JOIN` (или кратко `RIGHT JOIN`). На рис. 4.4 показан пример правого внешнего соединения.

Язык T-SQL также поддерживает полное внешнее соединение (`FULL OUTER JOIN` или кратко `FULL JOIN`), которое сохраняет обе стороны. Это соединение иллюстрирует пример на рис. 4.5.

Левая таблица

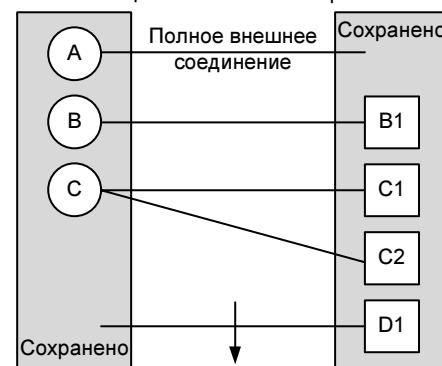
Правая таблица



Результирующая таблица

Левая таблица

Правая таблица



Результирующая таблица

Рис. 4.4. Правое внешнее соединение

Рис. 4.5. Полное внешнее соединение

Полное внешнее соединение возвращает внутренние строки, которые обычно возвращаются внутренним соединением, плюс строки слева, которые не имеют совпадений в правой стороне, со значениями `NULL` в качестве заполнителей с правой стороны, плюс строки справа, которые не имеют совпадений в левой стороне, со значениями `NULL` в качестве заполнителей с левой стороны.

Запросы с мультисоединениями

Важно помнить, что в языке T-SQL соединение в принципе имеет место между двумя таблицами в каждый момент времени. Запрос с несколькими соединениями оценивает соединения слева направо. Таким образом, результат одного запроса используется в качестве левого входа для следующего соединения. Если вы не разберетесь в этом, то можете столкнуться с логическими ошибками, особенно если присутствуют внешние соединения (в случае внутренних и перекрестных соединений порядок не может влиять на результат).

Предположим в качестве примера, что вы хотите возвратить всех поставщиков из Японии и соответствующие им продукты, если такие имеются. Для этого нужно внешнее соединение между таблицами Production.Suppliers и Production.Products, сохраняющее поставщиков (Suppliers). Но вы также хотите включить информацию о категории продукта, поэтому добавляете внутреннее соединение с Production.Categories следующим образом:

```
SELECT S.companyname AS supplier,
       S.country,
       P.productid, P.productname, P.unitprice,
       C.categoryname
  FROM Production.Suppliers AS S
 LEFT OUTER JOIN Production.Products AS P
    ON S.supplierid = P.supplierid
 INNER JOIN Production.Categories AS C
    ON C.categoryid = P.categoryid
 WHERE S.country = N'Japan';
```

Взгляните на результат этого запроса.

supplier	country	productid	productname	unitprice	categoryname
Supplier QOVFD	Japan	9	Product AOZBW	97.00	Meat/Poultry
Supplier QOVFD	Japan	10	Product YHXGE	31.00	Seafood
Supplier QOVFD	Japan	74	Product BKAZJ	10.00	Produce
Supplier QWUSF	Japan	13	Product POXFU	6.00	Seafood
Supplier QWUSF	Japan	14	Product PWCJB	23.25	Produce
Supplier QWUSF	Japan	15	Product KSZOI	15.50	Condiments

Поставщик Supplier XYZ из Японии был отброшен — можете объяснить, почему?

Первое соединение включает внешние строки (поставщики, не имеющие продуктов), но ставит значения NULL в атрибутах продуктов для этих строк. Затем соединение с Production.Categories сравнило значения NULL в categoryid во внешних строках со значениями categoryid в таблице Production.Categories и отбросило эти строки, т. е. внутреннее соединение, которое следовало за внешним соединением, аннулировало внешнюю часть соединения.

Существует множество способов решить эту проблему, но, возможно, наиболее естественным будет использование интересной возможности этого языка — выделить некоторые соединения в их собственную независимую логическую фазу. Что вам действительно нужно, так это левое внешнее соединение между таблицей

Production.Suppliers и результатом внутреннего соединения между Production.Productis и таблицей Production.Categories. Вы можете сформулировать ваш запрос точно так же, как приведенный далее.

```
SELECT S.companyname AS supplier, S.country,
       P.productid, P.productname, P.unitprice,
       C.categoryname
  FROM Production.Suppliers AS S
 LEFT OUTER JOIN
  (Production.Products AS P
    INNER JOIN Production.Categories AS C
      ON C.categoryid = P.categoryid)
  ON S.supplierid = P.supplierid
 WHERE S.country = N'Japan';
```

Теперь в результате запроса остались поставщики из Японии, не имеющие продуктов.

supplier	country	productid	productname	unitprice	categoryname
Supplier QOVFD	Japan	9	Product AOZBW	97.00	Meat/Poultry
Supplier QOVFD	Japan	10	Product YHXGE	31.00	Seafood
Supplier QOVFD	Japan	74	Product BKAZJ	10.00	Produce
Supplier QWUSF	Japan	13	Product POXFU	6.00	Seafood
Supplier QWUSF	Japan	14	Product PWCJB	23.25	Produce
Supplier QWUSF	Japan	15	Product KSZOI	15.50	Condiments
Supplier XYZ	Japan	NULL	NULL	NULL	NULL

Такое свойство языка может действительно сбить с толку, но, к счастью, есть решение этой проблемы.

Примечательно, что внешние соединения имеют только один стандартный синтаксис — с использованием ключевого слова `JOIN` и предложения `ON`. По существу, именно введение внешних соединений в этот стандарт привело к изменению синтаксиса, реализующему необходимость разделения между предложениями, в которых указывается предикат сопоставления (`ON`), и предикатом фильтрации (`WHERE`). Затем, возможно, из соображений согласованности, в стандарт была добавлена поддержка аналогичного синтаксиса, основанного на использовании ключевого слова `JOIN` для перекрестных и внутренних соединений.

КОНТРОЛЬНЫЕ ВОПРОСЫ

- В чем заключается разница между старым и новым синтаксисом перекрестных соединений?
- Назовите разные типы внешних соединений.

Ответы на контрольные вопросы

- В новый синтаксис входят ключевые слова `CROSS JOIN` между именами таблиц, тогда как старый синтаксис использует для этого запятую.
- Левое, правое и полное соединения.

ПРАКТИКУМ Использование соединений

В этом практикуме вы проверите ваши знания об использовании соединений.

Задание 1. Сопоставление клиентов и заказов с помощью внутреннего соединения

В этом задании вы будете практиковаться в сопоставлении клиентов и заказов с использованием внутренних соединений.

1. Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
2. Напишите запрос, который сопоставляет клиентов соответствующим им заказам, возвращая только совпадения. От вас не требуется возвращать клиентов, не имеющих соответствующих им заказов.

Запустите следующий запрос:

```
USE TSQL2012;
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
INNER JOIN Sales.Orders AS O
ON C.custid = O.custid;
```

Этот запрос генерирует такой результат:

custid	companyname	ordered	orderdate
85	Customer ENQZT	10248	2006-07-04 00:00:00.000
79	Customer FAPSM	10249	2006-07-05 00:00:00.000
34	Customer IBVRG	10250	2006-07-08 00:00:00.000
...			

Задание 2. Сопоставление клиентов и заказов с помощью внешнего соединения

В этом задании вы будете практиковаться в сопоставлении клиентов и заказов с использованием внешних соединений.

1. Возьмите запрос, который вы составили в п. 2 предыдущего задания. Перепиши-те его таким образом, чтобы включить клиентов без заказов. Измените тип соединения (чтобы это было левое внешнее соединение) следующим образом:

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
LEFT OUTER JOIN Sales.Orders AS O
ON C.custid = O.custid;
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE orderdate BETWEEN '20080211' AND '20080212 23:59:59.999';
```

Теперь на выходе запроса есть клиенты, у которых нет заказов, со значениями NULL в атрибутах заказа.

custid	companyname	ordered	orderdate
85	Customer ENQZT	10248	2006-07-04 00:00:00.000
79	Customer FAPSM	10249	2006-07-05 00:00:00.000
34	Customer IBVRG	10250	2006-07-08 00:00:00.000
...			
22	Customer DTDMN	NULL	NULL
57	Customer WVAXS	NULL	NULL

(832 rows affected)

2. Возвратите только клиентов без заказов. Для этого добавьте к предыдущему запросу предложение WHERE, которое фильтрует строки, содержащие значение NULL в ключе с несохраненной стороны (O.orderid), следующим образом:

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
LEFT OUTER JOIN Sales.Orders AS O
ON C.custid = O.custid
WHERE O.orderid IS NULL;
```

Результат запроса показывает, что два клиента не имеют заказов.

custid	companyname	ordered	orderdate
22	Customer DTDMN	NULL	NULL
57	Customer WVAXS	NULL	NULL

3. Напишите запрос, который возвращает всех клиентов, но находит соответствующие им заказы, только если они были размещены в феврале 2008 г. Поскольку как сравнение ID клиента для самого клиента с ID клиента для заказа, так и диапазон дат считаются частью логики сопоставления, то оба сравнения должны быть представлены в предложении ON следующим образом:

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
LEFT OUTER JOIN Sales.Orders AS O
ON C.custid = O.custid
AND O.orderdate >= '20080201'
AND O.orderdate < '20080301';
```

Этот запрос возвращает 110 строк; далее представлена часть результата запроса.

custid	companyname	ordered	orderdate
1	Customer NRZBB	NULL	NULL
2	Customer MLTDN	NULL	NULL
3	Customer KBUDE	NULL	NULL
4	Customer HFBZG	10864	2008-02-02 00:00:00.000

```
5      Customer HGVLZ 10866    2008-02-03 00:00:00.000
5      Customer HGVLZ 10875    2008-02-06 00:00:00.000
...
```

Если вы укажете предикат диапазона дат в предложении `WHERE`, клиенты, которые не разместили заказы в этом месяце, будут отброшены, а это не то, что вам нужно.

Резюме занятия

- Перекрестные соединения возвращают декартово произведение строк обеих сторон.
- Внутренние соединения сопоставляют строки с помощью предиката и возвращают только совпадения.
- Внешние соединения сопоставляют строки с помощью предиката и возвращают как совпадения, так и несовпадения из таблиц, помеченных как сохраненные.
- Запросы с мультикоединениями содержат несколько соединений. В них возможно присутствие разных типов соединений. Логический порядок обработки соединений можно контролировать с помощью круглых скобок или изменением положения предложения `ON`.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. В чем заключается разница между предложениями `ON` и `WHERE`?
 - A. Предложение `ON` использует двоичную логику, а предложение `WHERE` — троичную.
 - B. Предложение `ON` использует троичную логику, а предложение `WHERE` — двоичную.
 - C. Во внешних соединениях предложение `ON` определяет фильтрацию, а предложение `WHERE` — сопоставление данных.
 - D. Во внешних соединениях предложение `ON` определяет сопоставление данных, а предложение `WHERE` — фильтрацию.
2. Какие ключевые слова можно опустить в новом стандартном синтаксисе соединений без изменения значения соединения? (Выберите все подходящие варианты.)
 - A. `JOIN`.
 - B. `CROSS`.
 - C. `INNER`.
 - D. `OUTER`.

3. Какой синтаксис рекомендуется использовать для перекрестных и внутренних соединений и почему?
- A. Синтаксис с ключевым словом `JOIN`, поскольку это соответствует синтаксису внешнего запроса и вызывает меньше ошибок.
- B. Синтаксис с запятой между именами таблиц, поскольку это соответствует синтаксису внешнего запроса и вызывает меньше ошибок.
- C. Рекомендуется избегать использования перекрестных и внутренних соединений.
- D. Рекомендуется использовать только символы нижнего регистра и опускать ключевые слова по умолчанию, как в случае `JOIN` вместо `INNER JOIN`, поскольку они увеличивают потребление энергии.

Занятие 2. Использование подзапросов, табличных выражений и оператора `APPLY`

Язык T-SQL поддерживает вложенные запросы (подзапросы). Это очень удобный компонент языка, который можно использовать для того, чтобы ссылаться на результаты одного запроса из другого. Нет никакой необходимости хранить результат одного запроса в переменной, чтобы иметь возможность ссылаться на этот результат из другого запроса. Данное занятие посвящено различным типам вложенных запросов. В нем также рассматривается использование табличных выражений, которые являются именованными запросами. Наконец, в данном занятии рассматривается табличный оператор `APPLY`.

Изучив материал этого занятия, вы сможете:

- ✓ Использовать независимые подзапросы и связанные (коррелированные) подзапросы
- ✓ Использовать подзапросы, которые возвращают скалярные, многозначные и табличные результаты
- ✓ Использовать в запросах производные таблицы и обобщенные табличные выражения (common table expressions, CTE)
- ✓ Создавать и использовать представления и встроенные табличные функции
- ✓ Использовать оператор `APPLY`

Продолжительность занятия — 60 минут.

Подзапросы

Вложенные запросы (*подзапросы*) могут быть независимыми, т. е. независимыми от внешнего запроса, или связанными, т. е. имеющими ссылку на столбец из таблицы во внешнем запросе. Что касается результата вложенного запроса, он может быть скалярным, многозначным или табличным.

В этом разделе мы начнем с рассмотрения более простых независимых запросов и затем перейдем к связанным вложенными (коррелированными) подзапросам.



Независимые подзапросы

Независимые подзапросы — это вложенные запросы, которые не имеют зависимостей от внешнего запроса. При желании можно выделить внутренний запрос и запустить его независимо. Благодаря этому устранение неполадок при работе с независимыми подзапросами проще, чем при работе со связанными подзапросами.

Как мы уже говорили, вложенный запрос может возвращать разные типы результатов. Он может возвращать единственное значение, несколько значений или даже целую результирующую таблицу. Подзапросы, возвращающие табличное значение, или табличные выражения, обсуждаются в разд. "Представления и встроенные табличные функции" далее в этой главе.

Вложенные запросы, которые возвращают единственное значение, или скалярные вложенные запросы, можно использовать там, где предполагается наличие выражения с единственным значением, как например на одной стороне сравнения. Например, в следующем запросе используется независимый подзапрос для возвращения продуктов с минимальной ценой за единицу.

```
SELECT productid, productname, unitprice
FROM Production.Products
WHERE unitprice = (SELECT MIN(unitprice)
                    FROM Production.Products);
```

Далее приведен результат этого запроса.

productid	productname	unitprice
33	Product ASTMN	2.50

Как видно из примера, подзапрос возвращает минимальную цену единицы продукта из таблицы Production.Products. Затем внешний запрос возвращает информацию о продуктах, имеющих минимальную цену. Попробуйте выделить только внутренний запрос и выполнить его — вы убедитесь в том, что это возможно.

Обратите внимание, если подзапрос, который должен быть скалярным, возвращает в действительности более одного значения, то это приведет к сбою при выполнении кода. Если скалярный подзапрос возвращает пустой набор, он конвертируется в значение NULL.

Подзапрос также может возвращать несколько значений в виде столбца. Такой подзапрос можно использовать там, где ожидается многозначный результат, например при использовании предиката IN. Следующий запрос использует многозначный подзапрос для возвращения продуктов, полученных от поставщиков из Японии.

```
SELECT productid, productname, unitprice
FROM Production.Products
WHERE supplierid IN
      (SELECT supplierid
       FROM Production.Suppliers
       WHERE country = N'Japan');
```

Далее приведен результат этого запроса.

productid	productname	unitprice
9	Product AOZBW	97.00
10	Product YHXGE	31.00
74	Product BKAZJ	10.00
13	Product POXFU	6.00
14	Product PWCJB	23.25
15	Product KSZOI	15.50

Внутренний запрос возвращает идентификаторы (ID) поставщиков из Японии. Затем внешний запрос возвращает информацию о продуктах, у которых ID поставщика входит в набор, возвращенный подзапросом. Предикат IN в подзапросе можно инвертировать по общему правилу, поэтому если необходимо возвратить продукты, поставщики которых не проживают в Японии, нужно просто заменить IN на NOT IN.

Коррелированные (связанные) подзапросы

Коррелированные (связанные) подзапросы — это подзапросы, в которых внутренний запрос имеет ссылку на столбец таблицы во внешнем запросе. Их использование несколько сложнее по сравнению с независимыми подзапросами, поскольку невозможно просто выделить внутреннюю часть и запустить ее отдельно.



В качестве примера предположим, что нужно возвратить продукты, имеющие минимальную цену за единицу товара в каждой категории. Можно использовать коррелированный подзапрос для возвращения минимальной цены за единицу для всех продуктов, у которых идентификатор категории (`categoryid`) равен идентификатору категории во внешней строке (корреляция), как показано в примере ниже.

```
SELECT categoryid, productid, productname, unitprice
FROM Production.Products AS P1
WHERE unitprice = (SELECT MIN(unitprice)
                   FROM Production.Products AS P2
                   WHERE P2.categoryid = P1.categoryid);
```

Этот запрос генерирует следующий результат:

categoryid	productid	productname	unitprice
1	24	Product QOGNU	4.50
2	3	Product IMEHJ	10.00
3	19	Product XKXDO	9.20
4	33	Product ASTMN	2.50
5	52	Product QRSRF	7.00
6	54	Product QAQRL	7.45
7	74	Product BKAZJ	10.00
8	13	Product POXFU	6.00

Обратите внимание, что внутренний и внешний запросы ссылаются на разные экземпляры одной и той же таблицы Production.Products. Чтобы подзапрос мог различать их, разным экземплярам надо назначить различные псевдонимы. Запрос назначает псевдоним P1 внешнему экземпляру таблицы и P2 внутреннему экземпляру; используя эти псевдонимы таблицы в качестве префикса, можно ссылаться на столбцы явным образом. Подзапрос использует связь (корреляцию) в предикате P2.categoryid = P1.categoryid, означающую, что отфильтровываются только продукты, у которых ID категории такой же, как в соответствующей строке внешнего запроса. Таким образом, для внешней строки с ID категории, равным 1, внутренний запрос возвращает минимальную цену из всех продуктов, у которых ID категории равен 1; если во внешней строке ID категории равен 2, внутренний запрос возвращает минимальную цену за единицу продукта для всех продуктов, у которых ID категории равен 2, и т. д.

Еще одним примером связанного подзапроса может служить следующий запрос, возвращающий клиентов, разместивших свои заказы 12 февраля 2007 г.

```
SELECT custid, companyname
FROM Sales.Customers AS C
WHERE EXISTS
(SELECT *
 FROM Sales.Orders AS O
 WHERE O.custid = C.custid
 AND O.orderdate = '20070212');
```

Этот запрос генерирует следующий результат:

custid	companyname
5	Customer HGVLZ
66	Customer LHANT

Предикат EXISTS принимает подзапрос на вход и возвращает значение "истина", когда подзапрос возвращает хотя бы одну строку, в противном случае — "ложь". В данном случае подзапрос возвращает заказы, размещенные клиентом, ID которого равно ID клиента во внешней строке (связь) и у которых дата заказа равняется 12 февраля 2007 г. Так что внешний запрос возвращает клиента, только если существует хотя бы один заказ, размещенный этим клиентом в указанную дату.

Как предикат, EXISTS не обязан возвращать результирующий набор, он возвращает значение "истина" или "ложь" в зависимости от того, возвращает ли запрос какие-либо строки. По этой причине оптимизатор запросов SQL Server игнорирует список SELECT в запросе, и поэтому что бы вы ни указали, это не повлияет на варианты оптимизации, такие как выбор индекса.

Предикат EXISTS можно инвертировать, так же как и любой другой предикат. Приведенный далее запрос инвертирует предикат предыдущего запроса, возвращая клиентов, которые не размещали заказы 12 февраля 2007 г.

```
SELECT custid, companyname
FROM Sales.Customers AS C
```

```
WHERE NOT EXISTS
(SELECT *
   FROM Sales.Orders AS O
  WHERE O.custid = C.custid
    AND O.orderdate = '20070212');
```

Этот запрос генерирует следующий результат, представленный в усеченном виде.

custid	companyname
72	Customer AHPOP
58	Customer AHXHT
25	Customer AZJED
18	Customer BSVAR
91	Customer CCFIZ
...	

Табличные выражения

Табличные выражения — это именованные запросы. Вы пишете внутренний запрос, который возвращает реляционный результирующий набор, даете ему имя и вызываете его из внешнего запроса. Язык T-SQL поддерживает четыре формы табличных выражений:



- производные таблицы;
- обобщенные табличные выражения (common table expression, CTE);
- представления;
- встроенные табличные функции.

Две первых формы видны только инструкции, которая их определяет. Что касается двух последних, определение табличного выражения сохраняется в базе данных как объект. Таким образом, оно является повторно используемым, и вы также можете контролировать доступ к объекту с помощью разрешений.

Обратите внимание, поскольку предполагается, что табличное выражение представляет отношение, внутренний запрос, определяющий выражение, должен быть реляционным. Это означает, что все возвращаемые внутренним запросом столбцы должны иметь имена (используйте псевдонимы, если столбец является результатом выражения), а также все имена столбцов должны быть уникальными. Кроме того, внутренний запрос не может содержать предложение ORDER BY (помните, у множества нет упорядочения). Последнее правило имеет исключение: если во внутреннем запросе используется параметр TOP или OFFSET...FETCH, предложение ORDER BY имеет значение, не относящееся к упорядочиванию представления; напротив, оно является частью спецификации фильтра. Поэтому если внутренний запрос использует предложение TOP или OFFSET...FETCH, предложение ORDER BY также допустимо. Но тогда у внешнего запроса не гарантировано упорядочивание представления, если он не имеет собственного предложения ORDER BY.

ВАЖНО!**Оптимизация табличных выражений**

Важно помнить, что с точки зрения производительности, когда SQL Server оптимизирует запросы, содержащие табличные выражения, он сначала извлекает логику табличного выражения и таким образом непосредственно взаимодействует с базовыми таблицами. Он не сохраняет результат табличного выражения во внутренней рабочей таблице и затем не взаимодействует с этой рабочей таблицей. Это означает, что табличные выражения никак не влияют на производительность — ни хорошо, ни плохо, — просто никак.

Теперь, когда вам понятны требования внутреннего запроса, можно перейти к изучению различных форм табличных выражений, поддерживаемых языком T-SQL.

Производные таблицы

Производная таблица — это, возможно, форма табличного выражения, которая более всего похожа на подзапрос, только это подзапрос, возвращающий в качестве результата целую таблицу. Для производной таблицы следует определить внутренний запрос в круглых скобках в предложении `FROM` внешнего запроса и указать имя производной таблицы после этих скобок.

Прежде чем продемонстрировать использование производных таблиц, мы рассмотрим в этом разделе запрос, который возвращает определенный желаемый результат. Затем опишем требование, которое не может быть выполнено непосредственно в этом запросе, и покажем, как можно этого добиться с помощью производной таблицы (или любого другого табличного выражения типичного для такого случая).

Рассмотрим следующий запрос, вычисляющий номера строк для продуктов, разделенных по категориям и упорядоченных по параметрам `unitprice` и `productid`.

```
SELECT
    ROW_NUMBER() OVER(PARTITION BY categoryid
                      ORDER BY unitprice, productid) AS rownum,
    categoryid, productid, productname, unitprice
FROM Production.Products;
```

Этот запрос возвращает следующий результат, показанный здесь в сокращенном виде.

rownum	categoryid	productid	productname	unitprice
1	1	24	Product QOGNU	4.50
2	1	75	Product BWRLG	7.75
3	1	34	Product SWNJV	14.00
4	1	67	Product XLXQF	14.00
5	1	70	Product TOONT	15.00
...				
1	2	3	Product IMEHJ	10.00
2	2	77	Product LUNZZ	13.00
3	2	15	Product KSZOI	15.50
4	2	66	Product LQMGN	17.00
5	2	44	Product VJIEO	19.45
...				

О функции `ROW_NUMBER()`, так же как и о прочих оконных функциях, вы узнаете в главе 5. Пока достаточно сказать, что функция `ROW_NUMBER()` вычисляет уникальные целые числа с приращением, начиная с 1, на основании указанной сортировки, по возможности, в пределах некоторой группы строк. Как видно из результата запроса, функция `ROW_NUMBER()` генерирует уникальные целые числа с приращением, начиная с 1, на основании сортировки значений `unitprice` и `productid`, внутри каждой группы, определенной значением `categoryid`.

Функция `ROW_NUMBER()` — и вообще оконные функции — разрешены только в предложениях `SELECT` и `ORDER BY` запроса. А что, если вы хотите отфильтровать строки на основе результата такой функции? Например, предположим, что вам нужно возвратить только строки, у которых номер строки меньше или равен 2; а именно, в каждой категории вам необходимо возвратить два продукта с самыми низкими ценами за единицу, используя идентификатор продукта `productid` как определяющий параметр. Ссыльаться на функцию `ROW_NUMBER()` в предложении `WHERE` запроса не разрешается. Также следует помнить, что в соответствии с логической обработкой запросов нельзя ссылаться на псевдоним столбца, который был назначен в списке `SELECT` в предложении `WHERE`, потому что предложение `WHERE` по определению оценивается раньше предложения `SELECT`.

Обойти эти ограничения можно с помощью табличного выражения. Нужно написать запрос, такой же как предыдущий, который вычисляет оконную функцию в предложении `SELECT`, и присвоить псевдоним столбца результирующему столбцу. Затем надо определить табличное выражение на основании этого запроса и сделать ссылку на псевдоним столбца в предложении `WHERE` внешнего запроса следующим образом:

```
SELECT categoryid, productid, productname, unitprice
FROM (SELECT ROW_NUMBER() OVER(PARTITION BY categoryid
                               ORDER BY unitprice, productid) AS rownum,
            categoryid, productid, productname, unitprice
         FROM Production.Products) AS D
WHERE rownum <= 2;
```

Результат этого запроса представлен здесь в сокращенном виде.

categoryid	productid	productname	unitprice
1	24	Product QOGNU	4.50
1	75	Product BWRLG	7.75
2	3	Product IMEHJ	10.00
2	77	Product LUNZZ	13.00
3	19	Product XKXDO	9.20
3	47	Product EZZPR	9.50
...			

Как видите, производная таблица определена в предложении `FROM` внешнего запроса в круглых скобках, за которыми следует имя производной таблицы. Тогда внешний запрос имеет право ссылаться на псевдонимы столбцов, которые были присвоены внутренним запросом. Это классическое использование табличных выражений.

При работе с производными таблицами доступны два типа присвоения псевдонимов столбцам: *встроенный* и *внешний*. В случае встроенного типа указывается псевдоним столбца как часть выражения, как в случае <выражение> AS псевдоним. В последнем запросе использовался встроенный тип для назначения псевдонима `rownum` выражению, содержащему функцию `ROW_NUMBER()`. При использовании внешней формы присвоения псевдонима не указываются псевдонимы результирующих столбцов как часть выражений для столбцов, вместо этого всем целевым столбцам даются имена сразу после имени производной таблицы, как в случае `FROM (...) AS D(rownum, categoryid, productid, productname, unitprice)`. При использовании внешней формы необходимо указать имена всех целевых столбцов, а не только тех, которые являются результатами вычислений.

Существует пара проблем, связанных с использованием производных таблиц, которые обусловлены тем, что производная таблица определяется в предложении `FROM` внешнего запроса. Первая проблема возникает, когда нужно ссылаться на одну производную таблицу из другой. В таком случае приходится прибегать к вложенным производным таблицам, а вложенность часто усложняет логику, затрудняя следование ей и увеличивая возможность появления ошибок. Рассмотрим следующую общую форму вложенности производных таблиц:

```
SELECT ...
FROM (SELECT ...
      FROM (SELECT ...
            FROM T1
            WHERE ...) AS D1
      WHERE ...) AS D2
WHERE ...;
```

Другая проблема производных таблиц имеет отношение к свойству "одновременности" языка. Напомним, что все выражения, появляющиеся в одной и той же фазе логической обработки запроса, концептуально оцениваются в один и тот же момент времени. Это справедливо и для табличных выражений. В результате имя, присвоенное производной таблице, не видно другим элементам, которые появляются в той же самой фазе логической обработки запроса, в которой было определено имя производной таблицы. Это означает, что если вы хотите объединить несколько экземпляров одной производной таблицы, вы не сможете это сделать. У вас не будет другого выбора, как продублировать код, определяя несколько производных таблиц на основании одного и того же запроса. Общая форма такого запроса выглядит следующим образом:

```
SELECT ...
FROM (SELECT ...
      FROM T1) AS D1
INNER JOIN
  (SELECT ...
    FROM T1) AS D2
ON ...;
```

Производные таблицы D1 и D2 основываются на одном и том же запросе. Это повторение кода увеличивает возможность возникновения ошибок, когда нужно переписывать внутренние запросы.

Обобщенные табличные выражения

Обобщенное табличное выражение (common table expression, CTE) подобно производной таблице в том смысле, что это именованное табличное выражение, видимое только инструкции, которая его определяет. Так же как запрос к производной таблице, запрос к CTE включает три основных части:



- внутренний запрос;
- имя, которое присваивается запросу и его столбцам;
- внешний запрос.

Однако в случае CTE, отличается расположение этих трех частей. Вспомните, для производных таблиц внутренний запрос появляется в предложении `FROM` внешнего запроса, т. е. одно в середине другого. В случае CTE, сначала вы даете имя CTE, потом задаете внутренний запрос и затем внешний запрос — более модульный подход.

```
WITH <CTE_name>
AS ( <inner_query> )
<outer_query>;
```

Вспомните пример из раздела о производных таблицах, где для каждой категории продукта надо было возвратить два продукта с минимальной ценой за единицу. Далее показано, как можно реализовать эту же задачу с помощью CTE.

```
WITH C AS
(SELECT
    ROW_NUMBER() OVER(PARTITION BY categoryid
                       ORDER BY unitprice, productid) AS rounum,
    categoryid, productid, productname, unitprice
   FROM Production.Products)
SELECT categoryid, productid, productname, unitprice
  FROM C
 WHERE rounum <= 2;
```

Как видите, принцип подобен производным таблицам, за исключением того, что внутренний запрос не определен в середине внешнего запроса; вместо этого, сначала определен внутренний запрос — от начала и до конца, затем внешний запрос — от начала до конца. Такая структура позволяет составлять более чистый код, который проще понимать.

Вам не нужно создавать вложенные CTE, как в случае с производными таблицами. Если необходимо определить несколько CTE, просто разделите их запятыми. Каждый из них может ссылаться на ранее определенные CTE, и внешний запрос может ссылаться на все CTE. После завершения внешнего запроса все CTE, определенные

в инструкции WITH, удаляются. Отказ от вложенных СТЕ упрощает следование логике запроса и таким образом снижает возможность появления ошибок. Например, если нужно сослаться на один СТЕ из другого, можно использовать следующую общую форму:

```
WITH C1 AS
( SELECT ...
  FROM T1
  WHERE ... ),
C2 AS
( SELECT ...
  FROM C1
  WHERE ... )
SELECT ...
FROM C2
WHERE ...;
```

Поскольку имя СТЕ назначено до начала внешнего запроса, можно сослаться на несколько экземпляров того же самого имени СТЕ, что невозможно в случае производных таблиц. Общая форма запроса выглядит следующим образом:

```
WITH C AS
( SELECT ...
  FROM T1 )
SELECT ...
FROM C AS C1
INNER JOIN C AS C2
ON ...;
```

СТЕ также имеют рекурсивную форму. Тело рекурсивного запроса содержит два или более запросов, обычно разделенных оператором UNION ALL. По крайней мере, один запрос в теле СТЕ, известный как закрепленный элемент, — это запрос, который возвращает реляционный результат. Закрепленный запрос вызывается только один раз. Кроме того, хотя бы один запрос в теле СТЕ, называемый рекурсивным элементом, имеет ссылку на имя СТЕ. Этот запрос вызывается неоднократно, до тех пор, пока он не возвратит пустой результирующий набор. В каждой итерации ссылка на имя СТЕ из рекурсивного элемента представляет предыдущий результирующий набор. Ссылка на имя СТЕ из внешнего запроса представляет объединенные результаты вызова закрепленного элемента и всех вызовов рекурсивного элемента.

Например, следующий код использует рекурсивный СТЕ для возвращения цепочки управления для определенного сотрудника, вплоть до исполнительного директора (CEO).

```
WITH EmpsCTE AS
( SELECT empid, mgrid, firstname, lastname, 0 AS distance
  FROM HR.Employees
 WHERE empid = 9
```

```

UNION ALL

SELECT M.empid, M.mgrid, M.firstname, M.lastname,
       S.distance + 1 AS distance
  FROM EmpsCTE AS S
  JOIN HR.Employees AS M
    ON S.mgrid = M.empid )
SELECT empid, mgrid, firstname, lastname, distance
  FROM EmpsCTE;

```

Этот код возвращает следующий результат:

empid	mgrid	firstname	lastname	distance
9	5	Zoya	Dolgopyatova	0
5	2	Sven	Buck	1
2	1	Don	Funk	2
1	NULL	Sara	Davis	3

Как видите, закрепленный элемент возвращает строку для сотрудника с номером 9. Затем рекурсивный элемент вызывается многократно, и каждый раз объединяет предыдущий результирующий набор с таблицей `HR.Employees` для возвращения непосредственного руководителя сотрудника из предыдущего захода.

Рекурсивный запрос останавливается, как только он возвратит пустой набор — в нашем случае, не найдя менеджера по СЕО. Затем внешний запрос возвращает объединенные результаты вызова закрепленного элемента (строка для сотрудника с идентификатором 9) и все вызовы рекурсивного элемента (все руководители выше по позиции сотрудника с ID = 9).

Представления и встроенные табличные функции

Как вы узнали из предыдущих разделов, производные таблицы и СТЕ — это табличные выражения, которые видны только для инструкции, в которой они определены. После выполнения инструкции такое табличное выражение удаляется. Следовательно, производные таблицы и СТЕ не являются повторно используемыми. Чтобы иметь возможность повторного использования, необходимо сохранить определение табличного выражения как объект в базе данных, а для этого можно использовать либо представления, либо встроенные функции, возвращающие табличное значение (табличные функции). Поскольку они являются объектами базы данных, можно управлять доступом к ним с помощью разрешений.

Главное различие между представлениями и встроенными табличными функциями заключается в том, что первые не принимают входные параметры, а вторые — принимают. В качестве примера предположим, что нам нужно сохранить определение запроса, выполняющего вычисление номера строки, который рассматривался как пример в предыдущих разделах. Для этого надо создать следующее представление:

```

IF OBJECT_ID('Sales.RankedProducts', 'V') IS NOT NULL
DROP VIEW Sales.RankedProducts;

```

```

GO
CREATE VIEW Sales.RankedProducts
AS
SELECT ROW_NUMBER() OVER(PARTITION BY categoryid
                        ORDER BY unitprice, productid) AS rownum,
       categoryid, productid, productname, unitprice
FROM Production.Products;
GO

```

Обратите внимание, в базе данных сохраняется не результирующий набор представления, а только лишь его определение. Теперь, когда наше представление сохранено, объект становится повторно используемым. Когда бы ни понадобилось вызвать это представление, оно доступно при условии, что имеются необходимые разрешения на вызов представления.

```

SELECT categoryid, productid, productname, unitprice
FROM Sales.RankedProducts
WHERE rownum <= 2;

```

Что касается встроенных табличных функций, они в принципе очень похожи на представления; но, как уже упоминалось, они поддерживают входные параметры. Так что если вам нужно определить нечто, подобное представлению с параметрами, более всего для этого подходит встроенная табличная функция. Рассмотрим в качестве примера рекурсивный СТЕ из разд. *"Обобщенные табличные выражения"*, который возвращал всю управляющую цепочку менеджеров для указанного сотрудника вплоть до генерального директора. Предположим, вы хотите инкапсулировать логику того запроса в табличное выражение для повторного использования, а также параметризовать входные данные сотрудника вместо использования константы 9. Этого можно достигнуть с помощью встроенной табличной функции со следующим определением:

```

IF OBJECT_ID('HR.GetManagers', 'IF') IS NOT NULL DROP FUNCTION HR.GetManagers;
GO
CREATE FUNCTION HR.GetManagers(@empid AS INT) RETURNS TABLE
AS
RETURN
    WITH EmpsCTE AS
    (
        SELECT empid, mgrid, firstname, lastname, 0 AS distance
        FROM HR.Employees
        WHERE empid = @empid
        UNION ALL
        SELECT M.empid, M.mgrid, M.firstname, M.lastname,
               S.distance + 1 AS distance
        FROM EmpsCTE AS S
        JOIN HR.Employees AS M
        ON S.mgrid = M.empid )

```

```
SELECT empid, mgrid, firstname, lastname, distance
FROM EmpsCTE;
GO
```

Заметьте, заголовок присваивает функции имя (`HR.GetManagers`), определяет входной параметр (`@empid AS INT`) и указывает, что функция возвращает табличный результат (определенный возвращаемым запросом). Функция затем использует предложение `RETURN`, возвращающее результат рекурсивного запроса, а закрепленный элемент рекурсивного CTE отфильтровывает сотрудника, ID которого равно входному ID сотрудника. При вызове этой функции вы передаете входной ID определенного сотрудника, как показано в следующем примере:

```
SELECT *
FROM HR.GetManagers(9) AS M;
```

Оператор **APPLY**

Оператор **APPLY** можно использовать для применения табличного выражения, заданного в качестве правого входа, к каждой строке табличного выражения, заданного как левый вход. По сравнению с объединением, оператор **APPLY** интересен тем, что правое табличное выражение может быть связано с левой таблицей; другими словами, внутренний запрос в правом табличном выражении может иметь ссылку на элемент из левой таблицы. Тогда получается, что правое табличное выражение оценивается отдельно для каждой левой строки. Это означает, что вы можете заменить использование курсоров в некоторых случаях оператором **APPLY**.

Например, пусть у вас есть запрос, который исполняет некоторую логику для определенного клиента. Предположим, что вам нужно применить логику этого запроса к каждому клиенту из таблицы `Sales.Customers`. Можно было бы использовать курсор, чтобы перебирать клиентов, и в каждой итерации вызывать запрос для текущего клиента. Вместо этого можно использовать оператор **APPLY**, представляющий таблицу `Sales.Customers` как левый вход, а табличное выражение на основании вашего запроса — как правый вход. Вы можете связать ID клиента во внутреннем запросе правого табличного выражения с ID клиента из левой таблицы.

Две формы оператора **APPLY** — **CROSS** и **OUTER** — описаны в следующих подразделах.

Оператор **CROSS APPLY**

Оператор **CROSS APPLY** обрабатывает левое и правое табличные выражения как входные. Правое табличное выражение может иметь связь с элементом из левой таблицы. Правое табличное выражение применяется к каждой строке из левого входа. По сравнению с оператором **OUTER APPLY**, специфика оператора **CROSS APPLY** состоит в том, что если правое табличное выражение возвращает пустой набор для левой строки, такая левая строка не возвращается. На рис. 4.6 показан пример оператора **CROSS APPLY**.

Буквы **X**, **Y** и **Z** обозначают ключевые значения их левой таблицы. Буква **F** обозначает табличное выражение, представленное как правый вход. И в круглых скобках вы видите ключевое значение из левой строки, переданное как связанный (корре-

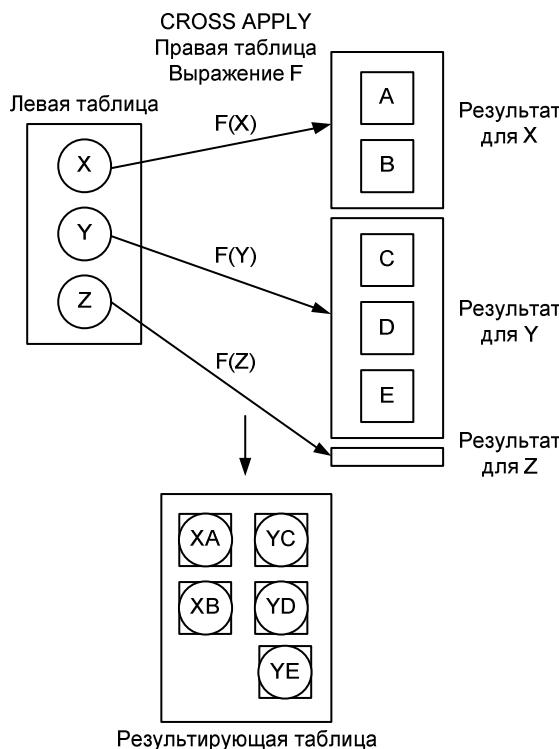


Рис. 4.6. Оператор CROSS APPLY

лированный) элемент. В правой части рисунка представлен результат, возвращенный из правого табличного выражения для каждой левой строки. В нижней части показан результат табличного оператора `CROSS APPLY`, где левые строки сопоставлены с соответствующими правыми строками, которые были возвращены для них. Обратите внимание, левая строка, которая возвращает пустой набор из правого табличного выражения, не возвращается. Это относится к строке с ключевым значением `Z`.

В качестве более показательного примера предположим, что вы пишете запрос, который возвращает два продукта с минимальной ценой за единицу для определенного поставщика — скажем, поставщик с номером 1.

```
SELECT productid, productname, unitprice
FROM Production.Products
WHERE supplierid = 1
ORDER BY unitprice, productid
OFFSET 0 ROWS FETCH FIRST 2 ROWS ONLY;
```

Этот запрос генерирует следующий результат:

productid	productname	unitprice
3	Product IMEHJ	10.00
1	Product HHYDP	18.00

Далее, предположим, что вам нужно применить эту логику каждому поставщику из Японии, который представлен в таблице Production.Suppliers. Вы не хотите использовать курсор для перебора всех поставщиков по одному и вызова отдельного запроса для каждого. Вместо этого, вы можете использовать оператор CROSS APPLY, как показано в следующем примере:

```
SELECT S.supplierid, S.companyname AS supplier, A.*  
FROM Production.Suppliers AS S  
CROSS APPLY (SELECT productid, productname, unitprice  
              FROM Production.Products AS P  
             WHERE P.supplierid = S.supplierid  
             ORDER BY unitprice, productid  
             OFFSET 0 ROWS FETCH FIRST 2 ROWS ONLY) AS A  
WHERE S.country = N'Japan';
```

Этот запрос генерирует следующий результат:

supplierid	supplier	productid	productname	unitprice
4	Supplier QOVFD	74	Product BKAZJ	10.00
4	Supplier QOVFD	10	Product YHXGE	31.00
6	Supplier QWUSF	13	Product POXFU	6.00
6	Supplier QWUSF	15	Product KSZOI	15.50

Как видно в запросе, левым входом для оператора APPLY является таблица Production.Suppliers с отфильтрованными поставщиками только из Японии. Правое табличное выражение — это связанная производная таблица, возвращающая два продукта с минимальными ценами за единицу для левого поставщика.

Поскольку оператор APPLY применяет правое табличное выражение к каждому поставщику слева, вы получаете два продукта с минимальной ценой для каждого поставщика из Японии. Так как оператор CROSS APPLY не возвращает левые строки, для которых правое табличное выражение возвращает пустой набор, поставщики из Японии, которые не имеют соответствующих продуктов, отбрасываются.

Оператор *OUTER APPLY*

Оператор OUTER APPLY делает то же самое, что и оператор CROSS APPLY, и кроме этого включает в результат строки с левой стороны, которые возвращают пустой набор с правой стороны. Значения NULL используются как заменители для результирующих столбцов с правой стороны. Иными словами, оператор OUTER APPLY сохраняет записи левой стороны. В некотором смысле, разница между операторами OUTER APPLY и CROSS APPLY подобна разнице между левым внешним соединением (LEFT OUTER JOIN) и внутренним соединением (INNER JOIN).

На рис. 4.7 представлен пример оператора OUTER APPLY.

Обратите внимание, на этот раз левая строка с ключевым значением Z сохранена.

Вспомним пример, возвращающий два продукта с минимальными ценами за единицу для каждого поставщика из Японии: если вы используете оператор OUTER

APPLY вместо оператора CROSS APPLY, вы сохраните левую сторону. Далее приведен переписанный запрос.

```
SELECT S.supplierid, S.companyname AS supplier, A.*  
FROM Production.Suppliers AS S  
    OUTER APPLY (SELECT productid, productname, unitprice  
                  FROM Production.Products AS P  
                 WHERE P.supplierid = S.supplierid  
                 ORDER BY unitprice, productid  
                 OFFSET 0 ROWS FETCH FIRST 2 ROWS ONLY) AS A  
WHERE S.country = N'Japan';
```

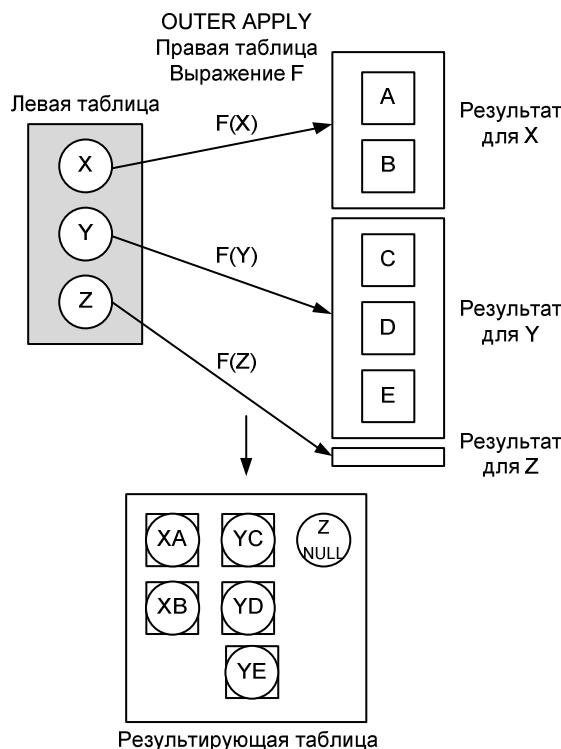


Рис. 4.7. Оператор OUTER APPLY

Вот как выглядит результат этого запроса:

	supplierid	supplier	productid	productname	unitprice
4		Supplier QOVFD	74	Product BKAZJ	10.00
4		Supplier QOVFD	10	Product YHXGE	31.00
6		Supplier QWUSF	13	Product POXFU	6.00
6		Supplier QWUSF	15	Product KSZOI	15.50
30		Supplier XYZ	NULL	NULL	NULL

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем заключается разница между независимым и коррелированным (связанным) подзапросами?
2. В чем заключается разница между операторами **APPLY** и **JOIN**?

Ответы на контрольные вопросы

1. Независимые подзапросы не зависят от внешнего запроса, тогда как связанные подзапросы имеют ссылку на элемент из таблицы во внешнем запросе.
2. При использовании оператора **JOIN** оба входа представляют статические отношения. В операторе **APPLY** левая сторона представляет собой статическое отношение, а правая часть может быть табличным выражением со связями с элементами из левой таблицы.

ПРАКТИКУМ Использование подзапросов, табличных выражений и оператора **APPLY**

В данном практикуме вам предстоит применить знания о подзапросах, табличных выражениях и операторе **APPLY**.

Задание 1. Формирование списка продуктов с минимальной ценой за единицу в пределах категории

В этом задании необходимо написать решение, использующее СТЕ для возвращения списка продуктов с наименьшей ценой в пределах категории товара.

1. Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
2. В качестве первого шага напишите запрос к таблице `Production.Products`, который группирует продукты по идентификатору категории `categoryid` и возвращает минимальное значение цены продукта `unitprice` для каждой категории. Следующий запрос реализует данный шаг:

```
SELECT categoryid, MIN(unitprice) AS mn
FROM Production.Products
GROUP BY categoryid;
```

Вы получите такой результат:

categoryid	mn
1	4.50
2	10.00
3	9.20
4	2.50
5	7.00
6	7.45
7	10.00
8	6.00

3. Следующий шаг вашего решения — определить СТЕ на основе предыдущего запроса и затем объединить СТЕ с таблицей Production.Products для того, чтобы возвратить для каждой категории продукты с минимальной ценой за единицу. Этот шаг можно реализовать с помощью следующего кода:

```
WITH CatMin AS
( SELECT categoryid, MIN(unitprice) AS mn
  FROM Production.Products
 GROUP BY categoryid )
SELECT P.categoryid, P.productid, P.productname, P.unitprice
  FROM Production.Products AS P
 INNER JOIN CatMin AS M
    ON P.categoryid = M.categoryid
   AND P.unitprice = M.mn;
```

Этот код представляет законченное решение, возвращающее желаемый результат.

categoryid	productid	productname	unitprice
1	24	Product QOGNU	4.50
2	3	Product IMEHJ	10.00
3	19	Product XKXDO	9.20
4	33	Product ASTMN	2.50
5	52	Product QSRXF	7.00
6	54	Product QAQRL	7.45
7	74	Product BKAZJ	10.00
8	13	Product POXFU	6.00

Задание 2. Формирование списка из N продуктов с минимальными ценами за единицу для каждого поставщика

В этом задании вам предстоит использовать операторы CROSS APPLY и OUTER APPLY.

1. Определите встроенную табличную функцию, которая принимает на вход ID поставщика (@supplierid), а также количество продуктов (@n), и возвращает @n продуктов с минимальными ценами для входного поставщика. При наличии одинаковых элементов в цене продукта используйте ID продукта в качестве уточняющего параметра.

Для определения функции используйте следующий код:

```
IF OBJECT_ID('Production.GetTopProducts', 'IF') IS NOT NULL
DROP FUNCTION
Production.GetTopProducts;
GO
CREATE FUNCTION Production.GetTopProducts(@supplierid AS INT, @n AS BIGINT)
RETURNS TABLE
AS
```

```

RETURN
SELECT productid, productname, unitprice
FROM Production.Products
WHERE supplierid = @supplierid
ORDER BY unitprice, productid
OFFSET 0 ROWS FETCH FIRST @n ROWS ONLY;
GO

```

2. Запросите эту функцию, чтобы протестировать ее, указывая ID поставщика равным 1 и число 2 для возвращения двух продуктов с минимальными ценами за единицу для входного поставщика.

```
SELECT * FROM Production.GetTopProducts(1, 2) AS P;
```

Этот код генерирует следующий результат:

productid	productname	unitprice
3	Product IMEHJ	10.00
1	Product HHYDP	18.00

3. Далее, возвратите для каждого поставщика из Японии два продукта с минимальными ценами за единицу. Для этого используйте оператор CROSS APPLY с Production.Suppliers в качестве источника данных с левой стороны и функцией Production.GetTopProducts в качестве правой стороны, как показано далее в примере.

```

SELECT S.supplierid, S.companyname AS supplier, A.*
FROM Production.Suppliers AS S
CROSS APPLY Production.GetTopProducts(S.supplierid, 2) AS A
WHERE S.country = N'Japan';

```

Этот код генерирует следующий результат:

supplierid	supplier	productid	productname	unitprice
4	Supplier QOVFD	74	Product BKAZJ	10.00
4	Supplier QOVFD	10	Product YHXGE	31.00
6	Supplier QWUSF	13	Product POXFU	6.00
6	Supplier QWUSF	15	Product KSZOI	15.50

4. В предыдущем шаге вы использовали оператор CROSS APPLY, и потому поставщики из Японии, не имеющие связанных с ними продуктов, были отброшены. Пусть вам нужно возвратить и их тоже. Тогда вы должны сохранить левую сторону, и чтобы добиться этого, используйте оператор OUTER APPLY, как показано в примере далее.

```

SELECT S.supplierid, S.companyname AS supplier, A.*
FROM Production.Suppliers AS S
OUTER APPLY Production.GetTopProducts(S.supplierid, 2) AS A
WHERE S.country = N'Japan';

```

На этот раз выходной набор содержит и поставщиков без продуктов.

supplierid	supplier	productid	productname	unitprice
4	Supplier QOVFD	74	Product BKAZJ	10.00
4	Supplier QOVFD	10	Product YHXGE	31.00
6	Supplier QWUSF	13	Product POXFU	6.00
6	Supplier QWUSF	15	Product KSZOI	15.50
30	Supplier XYZ	NULL	NULL	NULL

5. После этого запустите следующий код для очистки данных:

```
IF OBJECT_ID('Production.GetTopProducts', 'IF') IS NOT NULL
    DROP FUNCTION Production.GetTopProducts;
```

Резюме занятия

- С помощью подзапросов можно вкладывать запросы друг в друга. Можно использовать как независимые, так и связанные подзапросы. Подзапросы могут возвращать в качестве результата одно значение, несколько значений или таблицу.
- Язык T-SQL поддерживает четыре вида табличных выражений, которые являются именованными выражениями запросов. Производные таблицы и CTE — это виды табличных выражений, доступных только для инструкции, в которой они определены. Представления и встроенные табличные функции представляют собой повторно используемые табличные выражения, определения которых сохраняются в базе данных в виде объектов. Представления не поддерживают входные параметры, тогда как встроенные табличные функции поддерживают их.
- Оператор `APPLY` обрабатывает два табличных выражения как входные наборы. Он применяет правое табличное выражение к каждой строке из левой стороны. Внутренний запрос в правом табличном выражении может иметь связи с элементами из левой таблицы. Оператор `APPLY` имеет два варианта. Вариант `CROSS APPLY` не возвращает левые строки, которые возвращают пустой набор из правой стороны. Оператор `OUTER APPLY` сохраняет левую сторону и поэтому возвращает строки из набора данных с левой стороны, когда правая сторона возвращает пустой набор. Значения `NULL` используются для замещения атрибутов правой стороны во внешних строках.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Что происходит, когда скалярный подзапрос возвращает более одного значения?
 - А. Запрос дает ошибку при выполнении.
 - Б. Возвращается первое значение.

- С. Возвращается последнее значение.
D. Результат конвертируется в NULL.
2. В чем преимущество использования СТЕ над производными таблицами? (Выберите все подходящие варианты.)
- A. СТЕ лучше работают, чем производные таблицы.
B. СТЕ не имеют вложенности; код более модульный, что облегчает следование логике кода.
C. В отличие от производных таблиц, можно ссылаться на несколько экземпляров одного и того же имени СТЕ, избегая повторения кода.
D. В отличие от производных таблиц, СТЕ могут быть использованы всеми инструкциями в сессии, а не только инструкцией, в которых они определены.
3. В чем заключается разница между результатами выражения `T1 CROSS APPLY T2` и `T1 CROSS JOIN T2` (правое табличное выражение не имеет связей с левым)?
- A. Оператор `CROSS APPLY` фильтрует только строки, в которых значения столбцов с одинаковым именем равны; оператор `CROSS JOIN` просто возвращает все комбинации.
B. Если `T1` содержит строки, а `T2` — нет, оператор `CROSS APPLY` возвращает пустой набор, а оператор `CROSS JOIN` все равно возвратит строки из `T1`.
C. Если `T1` содержит строки, а `T2` — нет, оператор `CROSS APPLY` все равно возвратит строки из `T1`, а оператор `CROSS JOIN` возвратит пустой набор.
D. Между ними нет разницы.

Занятие 3. Использование операторов работы с наборами

Операторы работы с наборами обрабатывают два результирующих набора запросов, сравнивая полные строки в этих результатах. В зависимости от результата сравнения и используемого оператора для работы с наборами, оператор определяет, возвращать строку или нет. Язык T-SQL поддерживает три оператора работы с наборами: `UNION`, `INTERSECT` и `EXCEPT`; также в нем поддерживается оператор `UNION ALL` для работы с наборами типа `multiset`.

Общий формат кода, использующего оператор работы с наборами, выглядит следующим образом:

```
<query 1>
<set operator>
<query 2>
[ORDER BY <order_by_list>]
```

Существует несколько правил, которым необходимо следовать при использовании операторов работы с наборами.

- Поскольку полные строки сопоставляются между результирующими наборами, количество столбцов в запросах должно совпадать и типы соответствующих столбцов должны быть сравнимыми (неявно конвертируемыми).
- Операторы работы с наборами рассматривают два значения NULL как равные при сравнении. Это довольно необычно по сравнению с предложениями фильтрации, такими как WHERE и ON.
- Поскольку речь идет об операторах работы с наборами, а не операторах курсора, отдельные запросы не могут иметь предложений ORDER BY.
- При желании можно добавить предложение ORDER BY, которое определяет порядок сортировки результата, возвращаемого оператором работы с наборами.
- Имена столбцов для результирующих столбцов определяются первым запросом.

Изучив материал этого занятия, вы сможете:

- ✓ Объединять результаты запросов с помощью операторов UNION и UNION ALL
- ✓ Создавать пересечение результатов запросов с помощью оператора INTERSECT
- ✓ Получать разницу между результатами запросов с помощью оператора EXCEPT

Продолжительность занятия — 30 минут.

Операторы UNION и UNION ALL

Оператор работы с наборами UNION объединяет результаты двух входных запросов. Как *оператор работы с наборами*, UNION имеет неявное свойство DISTINCT, означающее, что он не возвращает дубликаты строк. На рис. 4.8 проиллюстрирован оператор UNION с помощью диаграммы Венна.

В качестве примера использования оператора UNION рассмотрим следующий запрос, который возвращает местоположения сотрудников, клиентов или и тех, и других.

```
SELECT country, region, city
FROM HR.Employees
UNION
SELECT country, region, city
FROM Sales.Customers;
```

Этот запрос генерирует следующий результат, приведенный здесь в сокращенном виде.

country	region	city
UK	NULL	London
USA	WA	Kirkland
USA	WA	Seattle
...		
(71 row(s) affected)		



Рис. 4.8. Оператор UNION

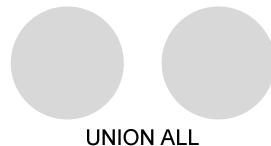


Рис. 4.9. Оператор UNION ALL

В таблице HR.Employees имеется 9 строк, а таблица Sales.Customers — 91 строку, но в объединенном результате представлено 71 различное местоположение; следовательно, оператор UNION возвращает 71 строку.

Если вы хотите сохранить дубликаты, например, чтобы позже сгруппировать строки и подсчитать количество вхождений, то вместо оператора UNION следует использовать оператор UNION ALL. Оператор UNION ALL объединяет результаты двух входных запросов, но не пытается удалить дубликаты. На рис. 4.9 представлен оператор UNION ALL с помощью диаграммы Венна.

Следующий запрос объединяет местоположения сотрудников и местоположения клиентов с помощью оператора UNION ALL.

```
SELECT country, region, city
FROM HR.Employees
UNION ALL
SELECT country, region, city
FROM Sales.Customers;
```

Поскольку оператор UNION ALL не пытается удалить дубликаты строк, результат содержит 100 строк (9 сотрудников + 91 клиент).

country	region	city
USA	WA	Seattle
USA	WA	Tacoma
USA	WA	Kirkland
USA	WA	Redmond
UK	NULL	London
UK	NULL	London
UK	NULL	London
...		

(100 row(s) affected)

ВАЖНО!**Сравнение операторов UNION и UNION ALL**

Если наборы, которые вы объединяете, не пересекаются и появление дубликатов не ожидается, операторы UNION и UNION ALL возвратят одинаковый результат. Но с точки зрения производительности в подобном случае лучше применять оператор UNION ALL, поскольку при использовании оператора UNION SQL Server может попытаться удалить дубликаты, что приведет к ненужным затратам.

Оператор *INTERSECT*

Оператор *INTERSECT* возвращает только различные строки, общие для обоих наборов. Другими словами, если строка хотя бы один раз появляется в первом наборе и хотя бы один раз — во втором, она появится один раз в результате оператора *INTERSECT*. На рис. 4.10 представлен оператор *INTERSECT* с помощью диаграммы Венна.

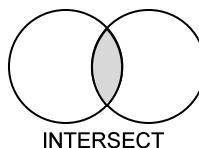


Рис. 4.10. Оператор *INTERSECT*

В качестве примера возьмем следующий код, который использует оператор *INTERSECT* для возвращения различных местоположений, которые являются местоположениями и сотрудника, и клиента (местоположения, где имеется как минимум один сотрудник и один клиент).

```
SELECT country, region, city
FROM HR.Employees
INTERSECT
SELECT country, region, city
FROM Sales.Customers;
```

Этот запрос генерирует следующий результат:

country	region	city
UK	NULL	London
USA	WA	Kirkland
USA	WA	Seattle

Заметьте, что местоположение (*UK*, *NULL*, *London*) было возвращено, т. к. оно появляется с обеих сторон. При сравнении значений *NULL* в столбце *region* в строках с обеих сторон, оператор *INTERSECT* считает их равными. Также обратите внимание, что не важно, сколько раз то же самое местоположение появляется с каждой стороны; пока оно появляется хотя бы один раз с обеих сторон, оно возвращается один раз в результирующем наборе.

Оператор *EXCEPT*

Оператор *EXCEPT* позволяет получить разность наборов данных. Он возвращает отличающиеся строки, которые появляются в первом запросе и не появляются во втором. Иными словами, если строка хотя бы раз появляется в первом запросе и ни разу во втором, она один раз возвращается в выходном наборе. Иллюстрация оператора *EXCEPT* с помощью диаграммы Венна приведена на рис. 4.11.

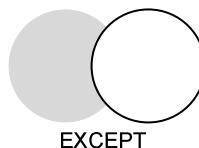


Рис. 4.11. Оператор EXCEPT

Следующий запрос, который можно использовать в качестве примера использования оператора EXCEPT, возвращает местоположения сотрудников, не являющиеся местоположениями клиентов.

```
SELECT country, region, city
FROM HR.Employees
EXCEPT
SELECT country, region, city
FROM Sales.Customers;
```

Этот запрос генерирует следующий результат:

country	region	city
USA	WA	Redmond
USA	WA	Tacoma

В случае операторов UNION и INTERSECT, порядок входных запросов не имеет значения. Однако для оператора EXCEPT выражения <query 1> EXCEPT <query 2> и <query 2> EXCEPT <query 1> имеют разное значение.

Наконец, операторы работы с наборами имеют приоритет: INTERSECT имеет более высокий приоритет, чем операторы UNION и EXCEPT, а операторы UNION и EXCEPT оцениваются слева направо на основании их позиций в выражении. Рассмотрим следующие операторы работы с наборами:

```
<query 1> UNION <query 2> INTERSECT <query 3>;
```

Во-первых, имеет место пересечение между запросом 2 и запросом 3, и затем — объединение результата пересечения и запроса 1. Вы всегда можете задавать приоритет с помощью круглых скобок. Поэтому, если вы хотите, чтобы сначала было выполнено объединение, надо использовать следующую запись запроса:

```
(<query 1> UNION <query 2>) INTERSECT <query 3>;
```

По окончании запустите следующий код для очистки данных:

```
DELETE FROM Production.Suppliers WHERE supplierid > 29;
IF OBJECT_ID('Sales.RankedProducts', 'V') IS NOT NULL DROP VIEW
Sales.RankedProducts;
IF OBJECT_ID('HR.GetManagers', 'IF') IS NOT NULL DROP FUNCTION HR.GetManagers;
```

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие операторы работы с наборами поддерживает язык T-SQL?
2. Назовите два требования к запросам, участвующим в операторе работы с наборами.

Ответы на контрольные вопросы

1. Операторы работы с наборами UNION, INTERSECT и EXCEPT, а также оператор UNION ALL работы с наборами типа multiset.
2. Количество столбцов в двух запросах должно быть одинаковым, и соответствующие столбцы должны иметь совместимые типы данных.

ПРАКТИКУМ Применение операторов работы с наборами

В данном практикуме вам предстоит проверить ваши знания об операторах работы с наборами.

Задание 1. Использование оператора работы с наборами EXCEPT

В этом задании вы определите связи между клиентами и сотрудниками через заказы с помощью оператора работы с наборами EXCEPT.

1. Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
2. Напишите запрос, который возвращает сотрудников, обрабатывающих заказы для клиента с номером 1, но не клиента с номером 2. Для этого используйте оператор работы с наборами EXCEPT, как показано в следующем примере:

```
SELECT empid  
FROM Sales.Orders  
WHERE custid = 1
```

EXCEPT

```
SELECT empid  
FROM Sales.Orders  
WHERE custid = 2;
```

Первый запрос возвращает сотрудников, обрабатывающих заказы для клиента с номером 1, а второй запрос возвращает сотрудников, обрабатывающих заказы для клиента с номером 2. Поскольку используется оператор EXCEPT между первым и вторым запросом, вы получаете сотрудников, обрабатывающих заказы только клиента с номером 1, но не с номером 2, как и требовалось. Помните, что оператор EXCEPT не возвращает дубликаты строк, поэтому вам не стоит беспокоиться о сотруднике, появляющемся более одного раза в выходном наборе. Ваш код возвратит ID следующих сотрудников:

```
empid  
-----  
1  
6
```

Задание 2. Использование оператора работы с наборами *INTERSECT*

В этом задании вы попробуете свои силы в определении связей между клиентами и сотрудниками через заказы с помощью оператора работы с наборами *INTERSECT*.

Используя ту же таблицу *Sales.Orders*, которую вы использовали в предыдущем задании, возвратите сотрудников, обрабатывающих заказы и для клиента с номером 1, и для клиента с номером 2. Для этого используйте те же самые два входных запроса, но на этот раз добейтесь "пересечения" результатов с помощью оператора *INTERSECT*, как показано в нижеприведенном примере.

```
SELECT empid  
FROM Sales.Orders  
WHERE custid = 1  
  
INTERSECT  
  
SELECT empid
```

```
FROM Sales.Orders  
WHERE custid = 2;
```

Этот код возвращает следующий результат:

```
empid
```

```
-----  
3  
4
```

Резюме занятия

- Операторы работы с наборами сравнивают полные строки в результирующих наборах двух запросов.
- Оператор *UNION* объединяет входные наборы, возвращая несовпадающие строки.
- Оператор *UNION ALL* объединяет входные наборы, не удаляя дубликаты.
- Оператор *INTERSECT* возвращает только строки, которые появляются в обоих входных наборах, возвращая несовпадающие строки.
- Оператор *EXCEPT* возвращает строки, которые появляются в первом наборе, но не во втором, возвращая несовпадающие строки.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в приложении "Ответы" в конце книги.

1. Какой из следующих операторов удаляет дубликаты из результата? (Выберите все подходящие варианты.)

- A. UNION.
 - B. UNION ALL.
 - C. INTERSECT.
 - D. EXCEPT.
2. В каком операторе порядок сортировки во входном запросе имеет значение?
- A. UNION.
 - B. UNION ALL.
 - C. INTERSECT.
 - D. EXCEPT.
3. Какое из приведенных далее выражений является эквивалентом выражения $<\text{query } 1> \text{ UNION } <\text{query } 2> \text{ INTERSECT } <\text{query } 3> \text{ EXCEPT } <\text{query } 4>$?
- A. $(<\text{query } 1> \text{ UNION } <\text{query } 2>) \text{ INTERSECT } (<\text{query } 3> \text{ EXCEPT } <\text{query } 4>)$.
 - B. $<\text{query } 1> \text{ UNION } (<\text{query } 2> \text{ INTERSECT } <\text{query } 3>) \text{ EXCEPT } <\text{query } 4>$.
 - C. $<\text{query } 1> \text{ UNION } <\text{query } 2> \text{ INTERSECT } (<\text{query } 3> \text{ EXCEPT } <\text{query } 4>)$.
 - D. $<\text{query } 1> \text{ UNION } (<\text{query } 2> \text{ INTERSECT } <\text{query } 3> \text{ EXCEPT } <\text{query } 4>)$.

Упражнения

В следующих упражнениях вы примените полученные знания о комбинировании наборов данных. Ответы на вопросы можно найти в *приложении "Ответы" в конце книги*.

Упражнение 1. Анализ кода

Вас попросили проанализировать код в системе, в которой существуют проблемы сопровождения кода, а также производительности. Вы обнаружили следующие результаты и должны определить, что рекомендовать клиенту.

1. Вы находите множество запросов, которые используют несколько уровней вложенности производных таблиц, затрудняя следование логике запросов. Вы также обнаруживаете множество запросов, которые объединяют несколько производных таблиц, основанных на одном и том же запросе, а также выясняете, что некоторые запросы повторяются в нескольких местах в коде. Что вы можете по рекомендовать клиенту для уменьшения сложности и повышения сопровождения кода?
2. В процессе анализа кода вы устанавливаете множество случаев, в которых используются курсоры для поочередного доступа к экземплярам определенного элемента (такого как клиент, сотрудник, грузоотправитель); далее код вызывает запрос для каждого из этих элементов, сохраняя результат во временной таблице; затем код просто возвращает все строки из этой временной таблицы. Клиент

испытывает проблемы и с сопровождением, и с производительностью существующего кода. Что вы можете порекомендовать?

3. Вы установили проблемы производительности, связанные с соединениями. Вы выяснили, что в системе нет индексов, созданных явно; существуют только индексы, созданные по умолчанию с помощью первичного ключа и ограничений уникальности. Что вы можете порекомендовать?

Упражнение 2. Объяснение операторов работы с наборами

Вы читаете лекцию об операторах работы с наборами на конференции. В конце занятия вы предоставляете аудитории возможность задать вопросы. Ответьте на следующие вопросы, заданные вам участниками лекции.

1. У нас в системе есть множество представлений, использующих оператор `UNION` для комбинирования непересекающихся множеств из разных таблиц. При вызове этих представлений заметно снижение производительности. Что вы посоветуете для улучшения производительности?
2. Укажите, пожалуйста, преимущества использования операторов работы с наборами, таких как `INTERSECT` и `EXCEPT`, по сравнению с применением внутренних и внешних соединений?

Рекомендуемые упражнения

Для успешного освоения экзаменационных тем, представленных в данной главе, выполните следующие задания.

Комбинирование наборов данных

Для тренировки применения знаний о комбинировании наборов данных используйте соединения, подзапросы и операторы работы с наборами в учебной базе данных TSQL2012.

- **Задание 1.** В этом задании вам предстоит выполнить соединение таблиц в учебной базе данных TSQL2012. Вы укажете, как связаны разные таблицы с помощью связей "внешний ключ — уникальный ключ" и напишете соединения для сопоставления строк связанных таблиц. Для того чтобы найти базу данных TSQL2012, используйте **Object Explorer** (Обозреватель объектов) в SSMS. Раскройте папку с таблицей `Sales.Orders` и затем папку `Keys`. Дважды щелкните кнопкой мыши на разных внешних ключах и затем раскройте элемент `Tables and Columns Specifications` (Спецификация таблиц и столбцов) для определения ссылающихся и тех, на которые ссылаются, таблиц и столбцов. Создайте запросы с соединениями для сопоставления строк между таблицей `Sales.Orders` и всеми связанными таблицами на основе указанных связей и убедитесь в том, что в ва-

шем запросе вы возвращаете столбцы из всех связанных таблиц. Для лучшего освоение соединений вы можете проделать то же самое с другими таблицами.

- **Задание 2.** В этом задании вам нужно указать строки, которые появляются в одной таблице, но не имеют соответствий в другой. Вам поставлена задача возвратить ID сотрудников из таблицы HR.Employees, которые не обрабатывали заказы (в таблице Sales.Orders) 12 февраля 2008 г. Напишите три разных решения, используя соединения, подзапросы и операторы работы с наборами. Для проверки правильности своих решений вы должны возвратить ID сотрудников 1, 2, 3, 5, 7 и 9.

ГЛАВА 5

Группирование и оконные функции

Темы экзамена

- Работа с данными.
 - Запрос данных с помощью инструкций SELECT.
 - Реализация подзапросов.
 - Реализация статистических запросов.

Данная глава посвящена операциям анализа данных. Функция анализа данных — это функция, применяемая к набору строк и возвращающая одно значение. Примером такой функции является статистическая функция `SUM`. Функцией анализа данных может быть либо функция группирования, либо оконная функция. Эти два вида различаются по способу определения набора строк, обрабатываемых функцией. Можно использовать сгруппированные запросы для определения сгруппированных таблиц, а затем групповая функция применяется к каждой группе. Или можно использовать оконные запросы, которые определяют оконные таблицы, и затем применить оконную функцию к каждому окну.

Занятия в данной главе посвящены группирующими запросам, а также сведению (*pivoting*) и отмене сведения (*unpivoting*) данных. Сведение данных можно рассматривать как специальную форму группирования, а отмена сведения — это операция, противоположная сведению. Также в этой главе описаны оконные запросы.

ПРЕЖДЕ ВСЕГО

Для полного освоения этой главы необходимо иметь:

- опыт работы с SQL Server Management Studio (SSMS);
- некоторый опыт в написании кода на языке T-SQL;
- понимание комбинирования данных;
- доступ к экземпляру SQL Server 2012 с установленной учебной базой данных TSQL2012.

Занятие 1. Написание запросов для группировки данных

Запросы для группировки данных (или группирующие запросы) можно использовать для того, чтобы задавать группы данных, а затем для каждой группы выполнить вычисления аналитических данных. Данные группируются в соответствии с набором атрибутов, известным как *набор группирования (группирующий набор)*. В стандартных запросах T-SQL определяется одиночный набор группирования, т. е. данные в них группируются только одним способом. В последних версиях языка T-SQL представлена поддержка функциональности, позволяющей определять несколько наборов группирования в одном запросе. В этом занятии мы начнем с рассмотрения запросов, в которых определяется одиночный набор группирования, а затем перейдем к запросам с несколькими наборами группирования.

Изучив материал этого занятия, вы сможете:

- ✓ Группировать данные с помощью одиночного набора группирования
- ✓ Использовать групповые функции
- ✓ Группировать данные с помощью нескольких наборов группирования

Продолжительность занятия — 60 минут.

Работа с одиничным набором группирования

Группирующие запросы позволяют организовать запрашиваемые строки в группы и применять вычисления для анализа данных, такие как статистические функции, к этим группам. Запрос становится сгруппированным, если используется групповая функция, предложение `GROUP BY` или оба.

Запрос, вызывающий групповую функцию, но не содержащий предложения `GROUP BY`, организует все строки в одну группу. В качестве примера рассмотрим следующий запрос:

```
USE TSQL2012;
SELECT COUNT(*) AS numorders
FROM Sales.Orders;
```

Этот запрос генерирует следующий результат:

```
numorders
-----
830
```

Поскольку предложение `GROUP BY` отсутствует в этом запросе, все строки, запрашиваемые в таблице `Sales.Orders`, объединены в одну группу, и функция `COUNT(*)` затем подсчитывает число строк в этой группе. Запросы с группированием возвращают одну результирующую строку на группу, и поскольку данный запрос определяет только одну группу, он возвращает лишь одну строку в результирующем наборе.

Используя явное предложение GROUP BY, можно группировать строки с помощью заданного группирующего набора выражений. Например, следующий запрос группирует строки по ID грузоотправителя (shipperid) и подсчитывает количество строк (в данном случае заказов) на каждую отдельную группу.

```
SELECT shipperid, COUNT(*) AS numorders
FROM Sales.Orders
GROUP BY shipperid;
```

Этот запрос генерирует следующий результат:

shipperid	numorders
1	249
2	326
3	255

Запрос определил три группы, поскольку имеются три разных идентификатора грузоотправителя (shipperid). Набор группирования может состоять из нескольких элементов. Например, следующий запрос группирует строки по ID грузоотправителя shipperid и году отгрузки shippedyear.

```
SELECT shipperid, YEAR(shippeddate) AS shippedyear,
       COUNT(*) AS numorders
  FROM Sales.Orders
 GROUP BY shipperid, YEAR(shippeddate);
```

Этот запрос генерирует следующий результат:

shipperid	shippedyear	numorders
1	2008	79
3	2008	73
1	NULL	4
3	NULL	6
1	2006	36
2	2007	143
2	NULL	11
3	2006	51
1	2007	130
2	2008	116
2	2006	56
3	2007	125

Обратите внимание, мы получили группу для каждой отдельной комбинации ID грузоотправителя и года отгрузки, существующей в данных, даже если год отгрузки имеет значение NULL. Вспомните, NULL в столбце shippeddate представляет неотгруженные заказы, поэтому значение NULL в столбце shippedyear представляет группу неотгруженных заказов для соответствующего грузоотправителя.

Если необходимо отфильтровать целые группы, требуется опция фильтрации, которая оценивается на уровне группы — в отличие от предложения WHERE, оценивае-

мого на уровне строки. Для этой цели язык T-SQL предоставляет предложение `HAVING`. Так же как предложение `WHERE`, предложение `HAVING` использует предикат, но оценивает предикат по группе, а не по строке. Это означает, что можно ссылаться на агрегатные вычисления, потому что данные уже сгруппированы.

Например, пусть вам нужно сгруппировать только поставленные заказы по ID грузоотправителя `shipperid` и году отгрузки `shippedyear`, а также отфильтровать только группы с менее чем 100 заказами. Для этого можно использовать следующий запрос:

```
SELECT shipperid, YEAR(shippeddate) AS shippedyear,
       COUNT(*) AS numorders
  FROM Sales.Orders
 WHERE shippeddate IS NOT NULL
 GROUP BY shipperid, YEAR(shippeddate)
 HAVING COUNT(*) < 100;
```

Этот запрос генерирует следующий результат:

shipperid	shippedyear	numorders
1	2008	79
3	2008	73
1	2006	36
3	2006	51
2	2006	56

Заметьте, запрос фильтрует только отгруженные заказы в предложении `WHERE`. Этот фильтр применяется на уровне строки до того, как данные сгруппированы. Затем запрос группирует данные по ID грузоотправителя и году отгрузки. После этого предложение `HAVING` фильтрует только группы, которые содержат количество строк (заказов), меньшее 100. Наконец, предложение `SELECT` возвращает ID грузоотправителя, год отгрузки и количество заказов на каждую остающуюся группу.

Язык T-SQL поддерживает несколько статистических функций. К ним относятся функция `COUNT(*)` и несколько общих функций для работы с наборами (как они сгруппированы в стандартном SQL), таких как `COUNT`, `SUM`, `AVG`, `MIN` и `MAX`. Общие функции наборов применяются к выражению и игнорируют значения `NULL`.

Следующий запрос в дополнение к некоторым общим функциям наборов, включая функцию `COUNT`, вызывает функцию `COUNT(*)`.

```
SELECT shipperid,
       COUNT(*) AS numorders,
       COUNT(shippeddate) AS shippedorders,
       MIN(shippeddate) AS firstshipdate,
       MAX(shippeddate) AS lastshipdate,
       SUM(val) AS totalvalue
  FROM Sales.OrderValues
 GROUP BY shipperid;
```

Этот запрос генерирует следующий результат (даты отформатированы для удобства чтения):

shipperid	numorders	shippedorders	firstshipdate	lastshipdate	totalvalue
3	255	249	2006-07-15	2008-05-01	383405.53
1	249	245	2006-07-10	2008-05-04	348840.00
2	326	315	2006-07-11	2008-05-06	533547.69

Обратите внимание на разницу между результатами функций COUNT(shippeddate) и COUNT(*). Первая игнорирует значения NULL в столбце shippeddate, и поэтому дает количество заказов меньшее или равное количеству заказов, полученных во второй функции.

Используя общие функции наборов, можно работать с отдельными вхождениями, указывая предложение DISTINCT перед выражением, как в следующем примере:

```
SELECT shipperid, COUNT(DISTINCT shippeddate) AS numshippingdates
FROM Sales.Orders
GROUP BY shipperid;
```

Этот запрос генерирует следующий результат:

shipperid	numshippingdates
1	188
2	215
3	198

Опция DISTINCT доступна не только в функции COUNT, но также в других общих функциях наборов. Однако, как правило, она используется с функцией COUNT.

С точки зрения логической обработки запросов, предложение GROUP BY оценивается после предложений FROM и WHERE и перед предложениями HAVING, SELECT и ORDER BY. Поэтому последние три предложения работают уже со сгруппированной таблицей, и таким образом выражения, которые они поддерживают, ограничены. Каждая группа представлена только одной результирующей строкой; поэтому все выражения, которые появляются в этих запросах, должны гарантировать единственное результирующее значение на группу. Нет никаких проблем при ссылке непосредственно на элементы, появляющиеся в предложении GROUP BY, поскольку каждый из них возвращает только одно отличающееся значение на группу. Но если нужно ссылаться на элементы из базовых таблиц, которых нет в списке GROUP BY, к ним необходимо применить статистическую (агрегатную) функцию. Это нужно для того, чтобы обеспечить возвращение выражением только одного значения на группу. В качестве примера рассмотрим следующий запрос, который выполняется с ошибкой.

```
SELECT S.shipperid, S.companyname, COUNT(*) AS numorders
FROM Sales.Shippers AS S
JOIN Sales.Orders AS O
    ON S.shipperid = O.shipperid
GROUP BY S.shipperid;
```

Запрос генерирует следующую ошибку:

```
Msg 8120, Level 16, State 1, Line 1
Column 'Sales.Shippers.companyname' is invalid in the select list because it is
not
contained in either an aggregate function or the GROUP BY clause.
```

Хотя вы знаете, что не может быть более одного отличающегося названия компании на каждый отличающийся ID грузоотправителя, этого не знает T-SQL. Поскольку столбец S.companyname не появляется в списке GROUP BY и не содержится в статистической функции, он разрешен в предложениях HAVING, SELECT и ORDER BY.

Можно найти несколько обходных путей для решения этой задачи. Одно из решений — добавить столбец S.companyname в список GROUP BY следующим образом:

```
SELECT S.shipperid, S.companyname,
       COUNT(*) AS numorders
  FROM Sales.Shippers AS S
 INNER JOIN Sales.Orders AS O
    ON S.shipperid = O.shipperid
 GROUP BY S.shipperid, S.companyname;
```

Этот запрос генерирует следующий результат:

shipperid	companyname	numorders
1	Shipper GVSUA	249
2	Shipper ETYNR	326
3	Shipper ZHISN	255

Другой способ решения этого вопроса — применить статистическую функцию, такую как MAX, для возвращения нужного столбца, как в следующем запросе:

```
SELECT S.shipperid,
       MAX(S.companyname) AS companyname,
       COUNT(*) AS numorders
  FROM Sales.Shippers AS S
 INNER JOIN Sales.Orders AS O
    ON S.shipperid = O.shipperid
 GROUP BY S.shipperid;
```

В этом случае статистическая функция выглядит неестественно, поскольку не может быть более одного отдельного имени компании на каждый отдельный ID грузоотправителя (shipperid). Поэтому предлагаемый первым вариантом план выглядит более оптимистично и к тому же более естественно.

Третий обходной путь — сначала сгруппировать и агрегировать строки таблицы Orders, определить табличное выражение с помощью сгруппированного запроса и затем объединить это табличное выражение с таблицей Shippers для получения названий компаний-грузоотправителей. Далее приведен код такого решения.

```
WITH C AS
( SELECT shipperid, COUNT(*) AS numorders
```

```
FROM Sales.Orders
GROUP BY shipperid )
SELECT S.shipperid, S.companyname, numorders
FROM Sales.Shippers AS S
INNER JOIN C
    ON S.shipperid = C.shipperid;
```

SQL Server обычно оптимизирует третий вариант решения так же, как первый. Первое решение можно считать предпочтительным, поскольку оно содержит намного меньше кода.

ЗАМЕЧАНИЕ

SQL Server 2012 позволяет создавать определяемые пользователем статистические выражения (user defined aggregate, UDA), используя код .NET в среде CLR (Common Language Runtime). SQL Server 2012 предоставляет встроенные определяемые пользователем статистические выражения для пространственных типов данных GEOMETRY и GEOGRAPHY, а также дает возможность создавать новые выражения, используя пространственные типы данных как входные данные. Более подробную информацию можно найти в электронной документации по SQL Server 2012.

Работа с несколькими наборами группирования

В языке T-SQL можно определить несколько наборов группирования в одном запросе. Иными словами, один запрос можно использовать для группирования данных более чем одним способом. Язык T-SQL поддерживает три выражения, которые позволяют задавать несколько наборов группирования: GROUPING SETS, CUBE и ROLLUP. Они используются в предложении GROUP BY.

Выражение GROUPING SETS можно использовать для создания списка наборов группирования, которые требуется определить в запросе. Следующий запрос определяет четыре набора группирования.

```
SELECT shipperid, YEAR(shippeddate) AS shipyear, COUNT(*) AS numorders
FROM Sales.Orders
WHERE shippeddate IS NOT NULL – исключить неотгруженные заказы
GROUP BY GROUPING SETS
(
( shipperid, YEAR(shippeddate)),
( shipperid ) ,
( YEAR(shippeddate) ) ,
( ) )
;
```

Наборы группирования представлены списком, разделенным запятыми внутри внешней пары круглых скобок, принадлежащих предложению GROUPING SETS. Внутренние круглые скобки используются для вложения каждого набора группирования. Если они не указаны, каждый отдельный элемент рассматривается как отдельный набор группирования.

В этом запросе определены четыре набора группирования. Один из них — пустой набор группирования, который определяет одну группу, содержащую все строки

для вычисления итоговых статистических функций. Запрос генерирует следующий результат.

shipperid	shipyear	numorders
1	2006	36
2	2006	56
3	2006	51
NULL	2006	143
1	2007	130
2	2007	143
3	2007	125
NULL	2007	398
1	2008	79
2	2008	116
3	2008	73
NULL	2008	268
NULL	NULL	809
3	NULL	249
1	NULL	245
2	NULL	315

Этот выходной набор объединяет результаты группирования и агрегирования данных четырех различных наборов группирования. Как видно из результата, значения `NULL` используются для подстановки в строках, в которых элемент не является частью набора группирования. Например, в результирующих строках, которые находятся в соответствии с набором группирования (`shipperid`), результирующий столбец `shipyear` имеет значение `NULL`. Аналогично, в строках, которые соответствуют набору группирования (`YEAR(shippeddate)`), столбец `shipperid` устанавливается в `NULL`.

Тот же результат может быть достигнут написанием четырех отдельных групповых запросов — каждый для определения только одного набора группирования — и объединением их результатов с помощью оператора `UNION ALL`. Однако такое решение потребует написания значительно большего количества кода и не будет так эффективно оптимизировано, как запрос, использующий выражение `GROUPING SETS`.

Язык T-SQL поддерживает два дополнительных выражения с именами `CUBE` и `ROLLUP`, которые можно рассматривать как упрощенные варианты выражения `GROUPING SETS`. Выражение `CUBE` принимает список выражений на вход и определяет все возможные наборы группирования, которые могут быть сгенерированы из входов, включая пустой набор группирования. Например, следующий запрос является логическим эквивалентом предыдущего запроса, использовавшего выражение `GROUPING SETS`.

```
SELECT shipperid, YEAR(shippeddate) AS shipyear, COUNT(*) AS numorders
FROM Sales.Orders
GROUP BY CUBE(shipperid, YEAR(shippeddate));
```

Выражение CUBE определяет все четыре возможных набора группирования из двух входов:

1. (shipperid, YEAR(shippeddate)).
2. (shipperid).
3. (YEAR(shippeddate)).
4. ().

Выражение ROLLUP также представляет собой сокращенный вариант выражения GROUPING SETS, но оно используется, когда существует иерархическая структура, сформированная входными элементами. В таком случае только поднабор возможных наборов группирования представляет реальный интерес. Рассмотрим, например, иерархию местоположений, созданную в этом заказе элементами shipcountry, shipregion и shipcity. Имеет смысл свертывать данные только в одном направлении, производя статистические вычисления для следующих наборов группирования:

1. (shipcountry, shipregion, shipcity).
2. (shipcountry, shipregion).
3. (shipcountry).
4. ().

Другие наборы группирования просто не интересны. Например, хотя одно и то же название города может появиться в разных местах в мире, нет смысла агрегировать все вхождения этого названия — независимо от региона и страны.

Поэтому при образовании элементами иерархической структуры используется выражение ROLLUP, что позволяет избежать вычисления ненужных статистических выражений. Далее приведен пример запроса, использующего выражение ROLLUP на основании вышеупомянутой иерархии.

```
SELECT shipcountry, shipregion, shipcity, COUNT(*) AS numorders
FROM Sales.Orders
GROUP BY ROLLUP(shipcountry, shipregion, shipcity);
```

Этот запрос генерирует следующий результат (показанный здесь в сокращенном виде):

shipcountry	shipregion	shipcity	numorders
<hr/>			
Argentina	NULL	Buenos Aires	16
Argentina	NULL	NULL	16
Argentina	NULL	NULL	16
...			
USA	AK	Anchorage	10
USA	AK	NULL	10
USA	CA	San Francisco	4
USA	CA	NULL	4

USA	ID	Boise	31
USA	ID	NULL	31
...			
USA	NULL	NULL	122
...			
NULL	NULL	NULL	830

Как уже упоминалось, значения NULL используются для подстановки, когда элемент не входит в набор группирования. Если все сгруппированные столбцы в базовой таблице запрещают использование значений NULL, можно указать строки, связанные с одиночным набором группирования, с помощью уникальной комбинации значений NULL и не-NULL в этих столбцах. Проблема возникает при определении строк, связанных с одиночным набором группирования, когда сгруппированный столбец разрешает значения NULL — как в случае со столбцом shipregion. Как объяснить, представляет ли значение NULL в результате замещающее значение (означающее "все регионы") или исходное значение NULL из таблицы (означающее "неприменимый регион")? В языке T-SQL имеются две функции, позволяющие решить эту проблему: GROUPING и GROUPING_ID.

Функция GROUPING принимает единственный элемент на вход и возвращает 0, если этот элемент входит в состав набора группирования, и 1 — когда не входит в него. Следующий запрос демонстрирует использование функции GROUPING:

```
SELECT
    shipcountry, GROUPING(shipcountry) AS grpcountry,
    shipregion , GROUPING(shipregion) AS grpregion,
    shipcity   , GROUPING(shipcity)   AS grpcity,
    COUNT(*) AS numorders
FROM Sales.Orders
GROUP BY ROLLUP(shipcountry, shipregion, shipcity);
```

Этот запрос генерирует следующий результат (показанный здесь в сокращенном виде):

shipcountry	grpcountry	shipregion	grpcountry	shipcity	grpcountry	numorders
Argentina	0	NULL	0	Buenos Aires	0	16
Argentina	0	NULL	0	NULL	1	16
Argentina	0	NULL	1	NULL	1	16
...						
USA	0	AK	0	Anchorage	0	10
USA	0	AK	0	NULL	1	10
USA	0	CA	0	San Francisco	0	4
USA	0	CA	0	NULL	1	4
USA	0	ID	0	Boise	0	31
USA	0	ID	0	NULL	1	31
...						
USA	0	NULL	1	NULL	1	122
...						
NULL	1	NULL	1	NULL	1	830

Теперь можно определить набор группирования, выполнив поиск значений "0" в элементах, входящих в состав набора группирования, и значений "1" в остальных.

Другая функция, которую можно использовать для определения наборов группирования — `GROUPING_ID`. Она принимает на вход список сгруппированных столбцов и возвращает целое число, представляющее битовую карту. Самый правый бит представляет самое правое входное значение. Этот бит равен 0, когда соответствующий элемент является частью набора группирования, и 1 — в противном случае. Каждый бит представляет значение 2^{n-1} , где n — позиция бита; таким образом, самый правый бит представляет 1, следующий слева от него 2, следующий 4, затем 8 и т. д. Результирующее целое число является суммой значений, представляющих элементы, не входящие в набор группирования, потому что их биты установлены. Далее приведен запрос, демонстрирующий использование этой функции.

```
SELECT GROUPING_ID(shipcountry, shipregion, shipcity) AS grp_id,
       shipcountry, shipregion, shipcity,
       COUNT(*) AS numorders
  FROM Sales.Orders
 GROUP BY ROLLUP( shipcountry, shipregion, shipcity );
```

Этот запрос генерирует следующий результат (показанный здесь в сокращенном виде):

grp_id	shipcountry	shipregion	shipcity	numorders
0	Argentina	NULL	Buenos Aires	16
1	Argentina	NULL	NULL	16
3	Argentina	NULL	NULL	16
...				
0	USA	AK	Anchorage	10
1	USA	AK	NULL	10
0	USA	CA	San Francisco	4
1	USA	CA	NULL	4
0	USA	ID	Boise	31
1	USA	ID	NULL	31
...				
3	USA	NULL	NULL	122
...				
7	NULL	NULL	NULL	830

Последняя строка выходного набора представляет пустой набор группирования — ни один из трех элементов не является частью набора группирования. Таким образом, соответствующие биты (значения 1, 2 и 4) установлены. Сумма значений, представляющих эти биты, равна 7.

СОВЕТ

Подготовка к экзамену

Можно указать несколько выражений `GROUPING SETS`, `CUBE` и `ROLLUP`, разделенных запятыми, в предложении `GROUP BY`. Этим достигается эффект умножения. Например, выражение `CUBE(a, b, c)` определяет 8 наборов группирования, а выражение `ROLLUP(x, y, z)` определяет 4 набора группирования. Разделив их запятой, как в случае `CUBE(a, b, c), ROLLUP(x, y, z)`, вы их перемножаете и получаете в результате 32 набора группирования.

КОНТРОЛЬНЫЕ ВОПРОСЫ

- Что делает запрос группирующим запросом?
- Какие выражения можно использовать для определения нескольких наборов группировки в одном запросе?

Ответы на контрольные вопросы

- Использование статистической функции, предложения GROUP BY или и того, и другого.
- Выражения GROUPING SETS, CUBE и ROLLUP.

ПРАКТИКУМ Написание запросов группировки данных

В этом практикуме вы проверите ваши знания об использовании группирующих запросов. Вы напишете запросы, которые определяют одиночный набор группирования, а также несколько наборов группирования.

Задание 1. Сбор статистической информации о клиентских заказах

В этом задании вам предстоит группировать и агрегировать данные, содержащие клиентов и заказы. Прежде чем рассматривать предложенное решение, попробуйте самостоятельно написать запрос.

- Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
- Напишите запрос, который вычисляет количество заказов на каждого клиента для всех клиентов из Испании.

Для решения этой задачи сначала нужно объединить таблицы Sales.Customers и Sales.Orders с использованием связи между идентификатором клиента в таблице клиентов и в таблице заказов. Затем следует отфильтровать только строки, в которых страна клиента — Испания. После этого необходимо сгруппировать полученные строки по ID клиента. Поскольку в обеих входных таблицах есть столбец custid, нужно снабдить столбец псевдонимом исходной таблицы. Например, при использовании столбца таблицы Sales.Customers, которой вы установите псевдоним C, в предложении GROUP BY его надо указывать как C.custid. Наконец, вам нужно возвратить ID клиента и количество строк в списке SELECT. Вот как выглядит запрос:

```
USE TSQL2012;
SELECT C.custid, COUNT(*) AS numorders
FROM Sales.Customers AS C
INNER JOIN Sales.Orders AS O
    ON C.custid = O.custid
WHERE C.country = N'Spain'
GROUP BY C.custid;
```

Этот запрос генерирует следующий результат:

custid	numorders
8	3
29	5
30	10
69	5

3. В выходной набор запроса добавьте информацию о городе. Для начала попробуйте просто добавить в список SELECT величину C.city, как в этом примере.

```
SELECT C.custid, C.city, COUNT(*) AS numorders
FROM Sales.Customers AS C
INNER JOIN Sales.Orders AS O
    ON C.custid = O.custid
WHERE C.country = N'Spain'
GROUP BY C.custid;
```

Появится следующая ошибка:

```
Msg 8120, Level 16, State 1, Line 1
Column 'Sales.Customers.city' is invalid in the select list because it is
not
contained in either an aggregate function or the GROUP BY clause.
```

4. Предложите решение, которое позволило бы возвращать город.

Одно из решений — добавить город в предложение GROUP BY, как в этом запросе.

```
SELECT C.custid, C.city, COUNT(*) AS numorders
FROM Sales.Customers AS C
INNER JOIN Sales.Orders AS O
    ON C.custid = O.custid
WHERE C.country = N'Spain'
GROUP BY C.custid, C.city;
```

Этот запрос генерирует следующий результат:

custid	city	numorders
8	Madrid	3
29	Barcelona	5
30	Sevilla	10
69	Madrid	5

Задание 2. Определение нескольких наборов группирования

В этом задании вам предстоит определить несколько наборов группирования.

За отправную точку возьмите запрос, который вы написали в п. 4 предыдущего задания. Кроме расчета по каждому клиенту, возвращаемого этим запросом, включите в выходной набор общее число заказов. В выходном наборе сначала должно быть показано число заказов по каждому клиенту, а затем — общее число заказов.

Можно использовать выражение GROUPING SETS для определения двух наборов группирования: одного для (C.custid, C.city) и второго — для пустого набора группирования (()). Чтобы расчеты по клиентам выводились раньше общего числа заказов, отсортируйте данные с помощью функции GROUPING(C.custid). Вот как выглядит такой запрос:

```
SELECT C.custid, C.city, COUNT(*) AS numorders
FROM Sales.Customers AS C
INNER JOIN Sales.Orders AS O
ON C.custid = O.custid
WHERE C.country = N'Spain'
GROUP BY GROUPING SETS ( (C.custid, C.city), () )
ORDER BY GROUPING(C.custid);
```

Этот запрос генерирует следующий результат:

custid	city	numorders
8	Madrid	3
29	Barcelona	5
30	Sevilla	10
69	Madrid	5
NULL	NULL	23

Резюме занятия

- Язык T-SQL позволяет группировать данные и выполнять операции анализа данных на группах.
- Можно применять статистические функции, такие как COUNT, SUM, AVG, MIN и MAX, к группам.
- Стандартные групповые запросы определяют только один набор группирования.
- Новую функциональность языка T-SQL можно использовать для определения нескольких наборов группирования в одном запросе с помощью выражений GROUPING SETS, CUBE и ROLLUP.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какие ограничения налагаются групповые запросы?
 - A. Если запрос является групповым, обязательно должна вызываться статистическая функция.
 - B. Если в запросе используется статистическая функция, должно присутствовать предложение GROUP BY.

- С. Элементы предложения `GROUP BY` также должны быть указаны в предложении `SELECT`.
- D. При ссылке на элемент из запрашиваемых таблиц в предложениях `HAVING`, `SELECT` или `ORDER BY`, он должен либо появиться в списке `GROUP BY`, либо содержаться в статистической функции.
2. Каково назначение функций `GROUPING` и `GROUPING_ID`? (Выберите все подходящие варианты.)
- A. Эти функции можно использовать в предложении `GROUP BY` для группировки данных.
- B. Эти функции можно использовать, чтобы указать, замещает ли значение `NULL` в результате запроса элемент, не являющийся частью набора группирования, или это оригинальное значение `NULL` из таблицы.
- C. Эти функции можно использовать, чтобы уникальным образом определить набор группирования, с которым связана результирующая строка.
- D. Эти функции можно использовать для сортировки данных на основании ассоциации набора группирования, т. е. сначала детализация, а затем агрегирование.
3. В чем заключается разница между статистической функцией `COUNT(*)` и базовой функцией `COUNT(<expression>)`?
- A. Функция `COUNT(*)` подсчитывает строки; функция `COUNT(<expression>)` подсчитывает строки, в которых `<expression>` не равно значению `NULL`.
- B. Функция `COUNT(*)` подсчитывает столбцы; функция `COUNT(<expression>)` подсчитывает строки.
- C. Функция `COUNT(*)` возвращает тип данных `BIGINT`; функция `COUNT(<expression>)` возвращает тип данных `INT`.
- D. Между этими функциями не существует разницы.

Занятие 2. Сведение и отмена сведения данных

Сведение данных — это особый случай группирования и агрегирования данных. Отмена сведения в некотором смысле представляет собой операцию, обратную сведению данных. Язык T-SQL поддерживает собственные операторы для обоих. Первая часть этого занятия посвящена сведению данных, вторая — отмене сведения данных.

Изучив материал этого занятия, вы сможете:

- ✓ Использовать оператор `PIVOT` для сведения данных
- ✓ Использовать оператор `UNPIVOT` для отмены сведения данных

Продолжительность занятия — 40 минут.

Сведение данных

Сведение — это технология группирования и агрегирования данных путем трансформации их из состояния строк в состояние столбцов. Во всех запросах сведения необходимо указать три элемента.

- Что вы хотите видеть в строках? Этот элемент называется *on rows*, или *группирующий элемент* (grouping element).
- Что вы хотите видеть в столбцах? Этот элемент называется *on cols*, или *распределяющий элемент* (spreading element).
- Что вы хотите видеть на пересечении каждого отдельного значения строки и столбца? Этот элемент известен как *данные* (data), или *агрегатный элемент* (aggregation element).



Предположим в качестве примера, что мы хотим запросить таблицу Sales.Orders. Нам нужно возвратить строку для каждого отдельного ID клиента (группирующий элемент), столбец для каждого отдельного ID грузоотправителя (распределяющий элемент) и на пересечении каждого клиента и грузоотправителя мы хотим видеть сумму затрат на транспортировку (агрегатный элемент). Язык T-SQL позволяет решить такую задачу сведения данных с помощью табличного оператора PIVOT. В общем, рекомендуемая форма запроса сведения выглядит следующим образом:

```
WITH PivotData AS
(
    SELECT
        <grouping column>,
        <spreading column>,
        <aggregation column>
    FROM <source table>
)
SELECT <select list>
FROM PivotData
PIVOT(<aggregate function>(<aggregation column>)
      FOR <spreading column> IN (<distinct spreading values>)) AS P;
```

Эта форма состоит из нижеприведенных пунктов.

- Вы должны определить табличное выражение (как, например, PivotData), которое возвращает три элемента, участвующие в сведении данных. Не рекомендуется запрашивать исходную базовую таблицу напрямую; скоро мы объясним причину этого.
- Вы должны запустить внешний запрос к табличному выражению и применить оператор PIVOT к этому табличному выражению. Оператор PIVOT возвращает табличный результат. Этой таблице нужно присвоить псевдоним, например, P.
- Спецификация оператора PIVOT начинается с указания статистической функции, примененной к агрегатному элементу, в нашем примере это SUM(freight).
- Затем вы укажете предложение FOR, за которым следует распределяющий столбец (spreading column), в данном примере это shipperid.

- После этого следует указать предложение `IN`, за которым идет список отличающихся значений, которые появляются в распределяющем элементе, разделенных запятыми. Величины, которые были значениями в распределяющем столбце (в данном случае это ID грузоотправителей), становятся именами столбцов в результирующей таблице. Таким образом, элементы списка должны быть обозначены как идентификаторы столбцов. Помните, что если идентификатор столбца является нерегулярным выражением, он должен иметь разделители. Поскольку идентификаторы грузоотправителей являются целыми числами, они должны иметь разделители: [1], [2], [3].

Согласно этому рекомендуемому синтаксису запросов сведения, следующий запрос выполняет задачу нашего примера (возвратить ID клиентов в строках, ID грузоотправителей в столбцах и общую стоимость транспортировки (фрахт) на пересечениях).

```
WITH PivotData AS
( SELECT
    custid,      -- группирующий столбец
    shipperid,   -- распределяющий столбец
    freight      -- агрегатный столбец
  FROM Sales.Orders )
SELECT custid, [1], [2], [3]
FROM PivotData
PIVOT(SUM(freight) FOR shipperid IN ([1],[2],[3])) AS P;
```

Этот запрос генерирует следующий результат (показанный здесь в сокращенном виде):

custid	1	2	3
1	95.03	61.02	69.53
2	43.90	NULL	53.52
3	63.09	116.56	88.87
4	41.95	358.54	71.46
5	189.44	1074.51	295.57
6	0.15	126.19	41.92
7	217.96	215.70	190.00
8	16.16	175.01	NULL
9	341.16	419.57	597.14
10	129.42	162.17	502.36
...			

(89 row(s) affected)

Если внимательно посмотреть на спецификацию оператора `PIVOT`, можно заметить, что мы указали агрегатный и распределяющий элементы, но не указали группирующий элемент. Группирующий элемент указан методом исключения — это то, что осталось от запрашиваемой таблицы, кроме агрегатного и распределяющего элементов. Поэтому рекомендуется подготовить табличное выражение для опера-

тора сведения, возвращающее только 3 элемента, участвующих в задаче сведения. Если запросить базовую таблицу напрямую (в данном случае это `Sales.Orders`), все столбцы таблицы, кроме агрегатных (`freight`) и распределяющих (`shipperid`) столбцов, неявно станут группирующими элементами. Это относится даже к столбцу первичного ключа `orderid`.

Поэтому вместо того, чтобы получить одну строку на клиента, вы в конечном итоге получите строку на заказ. Можно в этом убедиться, выполнив следующий код:

```
SELECT custid, [1], [2], [3]
FROM Sales.Orders
PIVOT(SUM(freight) FOR shipperid IN ([1],[2],[3])) AS P;
```

Этот запрос генерирует следующий результат (показанный здесь в сокращенном виде):

custid	1	2	3
85	NULL	NULL	32.38
79	11.61	NULL	NULL
34	NULL	65.83	NULL
84	41.34	NULL	NULL
76	NULL	51.30	NULL
34	NULL	58.17	NULL
14	NULL	22.98	NULL
68	NULL	NULL	148.33
88	NULL	13.97	NULL
35	NULL	NULL	81.91
...			

(830 row(s) affected)

Вы получите 830 строк, потому что в таблице `Sales.Orders` имеется 830 строк. Определив табличное выражение, как было показано в рекомендуемом решении, вы можете контролировать, какие столбцы будут использоваться в качестве группирующих столбцов. Если вы возвратите столбцы `custid`, `shipperid` и `freight` в табличном выражении и используете два последних в качестве распределяющего и агрегатного элементов соответственно, оператор `PIVOT` неявно решит, что элемент `custid` — это группирующий элемент. Поэтому данные будут сгруппированы по `custid`, и в результате будет возвращена одна строка на каждого клиента.

Необходимо знать несколько ограничений, относящихся к оператору `PIVOT`.

- Агрегатный и распределяющий элементы не должны быть прямыми результатами выражений; они должны быть именами столбцов из запрашиваемой таблицы. Однако можно применить выражения в запросе, определяющем табличное выражение, присвоить псевдонимы этим выражениям и затем использовать эти псевдонимы в операторе `PIVOT`.
- Функция `COUNT(*)` не разрешена в виде статистической (агрегатной) функции, используемой оператором `PIVOT`. Если нужно выполнить расчет количества, сле-

дует использовать общую статистическую функцию COUNT(<имя_столбца>). Простое решение этой ситуации — определить фиктивный столбец в табличном выражении в виде константы, как в случае 1 AS agg_col, и затем в операторе PIVOT применить статистическую функцию к этому столбцу: COUNT(agg_col).

- Оператор PIVOT может использовать только одну статистическую функцию.
- Предложение IN оператора PIVOT принимает статический список распределяемых значений. Он не поддерживает подзапрос на входе. Вы заранее должны знать, какие отличающиеся значения находятся в распределяющем столбце, и указать их в предложении IN. Если этот список заранее неизвестен, можно использовать динамический SQL для того, чтобы составить и выполнить строку запроса после того, как запрошены отличающиеся значения в данных. Подробно динамический SQL рассматривается в главе 12.

Отмена сведения данных

Отмена сведения данных может рассматриваться как процедура, обратная сведению данным. Исходной точкой служат некоторые сведенные данные. При отмене сведения данных входные данные разворачиваются из состояния столбцов в состояние строк. Точно так же как язык T-SQL поддерживает собственный табличный оператор PIVOT для сведения данных, он поддерживает собственный оператор UNPIVOT для выполнения отмены сведения. Подобно оператору PIVOT, оператор UNPIVOT реализован как табличный оператор, который используется в предложении FROM. Этот оператор обрабатывает входную таблицу, которая расположена слева от него и может быть результатом других табличных операторов, таких как объединение.

Выходом оператора UNPIVOT является табличный результат, который может использоваться в качестве входа для других табличных операторов, находящихся справа от него.

Для демонстрации отмены сведения данных используйте в качестве примера тренировочную таблицу Sales.FreightTotals.

Следующий код создает пример данных и выполняет запрос, чтобы показать содержимое таблицы.

```
USE TSQL2012;
IF OBJECT_ID('Sales.FreightTotals') IS NOT NULL DROP TABLE Sales.FreightTotals;
GO
WITH PivotData AS
( SELECT
    custid, -- группирующий столбец
    shipperid, -- распределяющий столбец
    freight -- агрегатный столбец
  FROM Sales.Orders )
SELECT *
  INTO Sales.FreightTotals
```

```
FROM PivotData
PIVOT( SUM(freight) FOR shipperid IN ([1],[2],[3]) ) AS P;
SELECT * FROM Sales.FreightTotals;
```

Этот код генерирует следующие выходные данные, показанные здесь в сокращенном виде.

custid	1	2	3
1	95.03	61.02	69.53
2	43.90	NULL	53.52
3	63.09	116.56	88.87
4	41.95	358.54	71.46
5	189.44	1074.51	295.57
6	0.15	126.19	41.92
7	217.96	215.70	190.00
8	16.16	175.01	NULL
9	341.16	419.57	597.14
10	129.42	162.17	502.36
...			

Как видим, исходная таблица имеет одну строку для каждого клиента и один столбец для каждого грузоотправителя (грузоотправители 1, 2 и 3). Пересечение каждого клиента и грузоотправителя содержит полную стоимость отгрузки. Задача отмены сведения данных заключается в возвращении строки для каждого клиента и грузоотправителя, содержащего ID клиента, в одном столбце; ID грузоотправителя во втором столбце и стоимость перевозки (*freight*) в третьем столбце.

Отмена сведения всегда берет набор исходных столбцов и разворачивает их в несколько строк, генерируя два выходных столбца: один для хранения исходных значений столбцов и другой — для имен исходных столбцов. Исходные столбцы уже существуют, поэтому их имена должны быть известны. Но эти два целевых столбца создаются процессом отмены сведения, поэтому для них нужно выбрать имена. В нашем примере исходные столбцы — это [1], [2] и [3]. Что касается имен целевых столбцов, нужно решить, как их назвать. В данном случае столбец значений удобно назвать *freight*, а столбец имен — *shipperid*. Помните, в каждой задаче отмены сведения необходимо указать следующие три элемента:

- набор исходных столбцов, для которых нужно выполнить отмену сведения (в данном случае [1], [2], [3]);
- имя, которое будет присвоено столбцу с полученными значениями (в данном случае *freight*);
- имя, которое вы хотите присвоить столбцу целевых имен (в данном случае *shipperid*).

После определения этих трех элементов используйте следующий формат запроса для решения задачи отмены сведения данных.

```
SELECT <column list>, <names column>, <values column>
FROM <source table>
UNPIVOT(<values column> FOR <names column> IN(<source columns>)) AS U;
```

Используя этот синтаксис, следующий запрос решает поставленную задачу.

```
SELECT custid, shipperid, freight
FROM Sales.FreightTotals
UNPIVOT(freight FOR shipperid IN([1],[2],[3])) AS U;
```

Этот запрос генерирует следующий результат (показанный здесь в сокращенном виде):

custid	shipperid	freight
1	1	95.03
1	2	61.02
1	3	69.53
2	1	43.90
2	3	53.52
3	1	63.09
3	2	116.56
3	3	88.87
4	1	41.95
4	2	358.54
4	3	71.46
...		

Кроме отмены сведения данных, оператор UNPIVOT отфильтровывает строки, содержащие значения NULL в столбце значений (в нашем случае это freight). Предполагается, что они представляют неподходящие случаи. От присутствия значений NULL невозможно было избавиться в источнике, если столбец был применимым по крайней мере еще для одного клиента. Но после отмены сведения данных нет причины хранить строку для определенной пары "клиент — грузоотправитель", если она не применима, т. е. если этот грузоотправитель не отгружал заказы для этого клиента.

Если говорить о типах данных, столбец имен определен символьной строкой в кодировке Unicode (NVARCHAR (128)). Столбец значений имеет тот же тип данных, что исходные столбцы, для которых выполнялась отмена сведения данных. Поэтому тип данных всех столбцов, для которых выполняется отмена сведения данных, должен быть одинаковым.

После завершения работы запустите следующий код для очистки данных:

```
IF OBJECT_ID('Sales.FreightTotals') IS NOT NULL DROP TABLE Sales.FreightTotals;
```

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем заключается разница между операторами PIVOT и UNPIVOT?
2. В виде каких конструкций реализованы операторы PIVOT и UNPIVOT?

Ответы на контрольные вопросы

1. Оператор PIVOT выполняет ротацию данных из состояния строк в состояние столбцов; оператор UNPIVOT разворачивает данные из столбцов в строки.
2. Операторы PIVOT и UNPIVOT реализуются как табличные операторы.

ПРАКТИКУМ Сведение данных

В данном практикуме вам предстоит применить знания о сведении данных.

Задание 1. Сведение данных с помощью табличного выражения

В этом задании вам нужно выполнить сведение данных с помощью табличного выражения.

1. Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
2. Напишите запрос PIVOT к таблице Sales.Orders, который возвращает максимальную дату отгрузки для каждого года заказа, а также ID грузоотправителя. Нужно возвратить годы в строках, ID грузоотправителей (1, 2 и 3) в столбцах и максимальную дату отгрузки в части данных.

Сначала попробуйте решить задачу с помощью следующего запроса:

```
SELECT YEAR(orderdate) AS orderyear, [1], [2], [3]
FROM Sales.Orders
PIVOT( MAX(shippeddate) FOR shipperid IN ([1],[2],[3]) ) AS P;
```

Предполагается, что в результате вы должны получить три строки для 2006, 2007 и 2008 гг., но вместо этого вы получите 830 строк, что равняется числу заказов в таблице.

3. Попробуйте объяснить, почему вы получили нежелательный результат, и предложите решение.
4. Причиной нежелательного результата является то, что вы напрямую запрашивали таблицу Sales.Orders. SQL Server определил, какие столбцы надо группировать, методом исключения; группирующие столбцы — все столбцы, которые вы не указали в качестве распределяющих (в данном случае shipperid), и агрегатные (в данном случае shippeddate). Все оставшиеся столбцы, включая ordered, неявно становятся частью списка группирования. Поэтому вы получили строку на заказ, а не строку на год. Для решения проблемы определите табличное выражение, которое содержит только группирующий, распределяющий и агрегатный столбцы, и представьте это табличное выражение как вход в запросе PIVOT. Решение должно выглядеть следующим образом:

```
WITH PivotData AS
( SELECT YEAR(orderdate) AS orderyear, shipperid, shippeddate
  FROM Sales.Orders )
SELECT orderyear, [1], [2], [3]
FROM PivotData
PIVOT( MAX(shippeddate) FOR shipperid IN ([1],[2],[3]) ) AS P;
```

Далее приведен результат запроса, в котором данные представлены в кратком формате.

orderyear	1	2	3
2007	2008-01-30	2008-01-21	2008-01-09
2008	2008-05-04	2008-05-06	2008-05-01
2006	2007-01-03	2006-12-30	2007-01-16

Задание 2. Сведение данных и расчеты

В этом задании вам нужно применить агрегатную функцию COUNT при сведении данных. Как и в предыдущем задании, вы будете использовать таблицу Sales.Orders из учебной базы TSQl2012.

- Напишите запрос PIVOT, который возвращает одну строку для каждого отличного ID клиента, столбец для каждого отличного ID грузоотправителя и подсчет заказов на пересечении "клиент — грузоотправитель".

Подготовьте табличное выражение, которое возвращает только столбцы custid и shipperid из таблицы Sales.Orders, и представьте это табличное выражение в качестве входа оператора PIVOT.

Для начала попробуйте использовать статистическую функцию COUNT(*), как показано в следующем запросе:

```
WITH PivotData AS
(
    SELECT
        custid,      -- группирующий столбец
        shipperid -- распределяющий столбец
    FROM Sales.Orders
)
SELECT custid, [1], [2], [3]
FROM PivotData
PIVOT(COUNT(*)) FOR shipperid IN ([1],[2],[3])) AS P;
```

Поскольку оператор PIVOT не поддерживает статистическую функцию COUNT(*), вы получите следующее сообщение об ошибке.

```
Msg 102, Level 15, State 1, Line 10
Incorrect syntax near '*'.
```

- Подумайте, как обойти эту проблему.

Для решения проблемы нужно использовать общую функцию COUNT(<имя столбца>), но помните, что на входе статистической функции не может быть результат выражения; это должно быть имя столбца, существующее в запрашиваемой таблице. Поэтому один из вариантов — использовать распределяющий столбец (spreading column) в качестве столбца агрегирования, как в случае COUNT(shipperid). Другая возможность — создать ложный столбец из константного выражения в табличном выражении и затем использовать этот столбец в качестве входа для функции COUNT, как показано далее.

```
WITH PivotData AS
(
    SELECT
        custid,      -- группирующий столбец
```

```
shipperid,    -- распределяющий столбец
1 AS aggcol  -- агрегатный столбец
FROM Sales.Orders  )
SELECT custid, [1], [2], [3]
FROM PivotData
PIVOT( COUNT(aggcol) FOR shipperid IN ([1],[2],[3]) ) AS P;
```

Этот запрос генерирует желаемый результат.

custid	1	2	3
1	4	1	1
2	1	0	3
3	2	3	2
4	1	8	4
5	5	9	4
6	1	3	3
7	5	3	3
8	1	2	0
9	6	7	4
10	3	3	8
...			

Резюме занятия

- Сведение данных — это особая форма группирования и агрегирования данных, когда данные разворачиваются из состояния строк в состояние столбцов.
- При сведении данных необходимо указать три вещи: группирующий элемент, распределяющий элемент и агрегатный элемент.
- T-SQL поддерживает собственный оператор `PIVOT`, используемый для сведения данных входной таблицы.
- Отмена сведения данных трансформирует данные из состояния столбцов в состояние строк.
- Чтобы выполнить отмену сведения данных, необходимо указать три вещи: исходные столбцы, для которых нужно выполнить отмену сведения, столбец целевых имен и столбец целевых значений.
- T-SQL поддерживает собственный оператор с именем `UNPIVOT`, который применяется для отмены сведения данных входной таблицы.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Как определяет оператор PIVOT, что является группирующим элементом?
 - A. Это элемент, указанный на входе функции GROUPING.
 - B. Это определяется методом исключения — элементы из запрашиваемой таблицы, которые не были указаны как распределяющие и агрегатные элементы.
 - C. Это элемент, указанный в предложении GROUP BY.
 - D. Это первичный ключ.
2. Что из перечисленного ниже не разрешено в спецификации оператора PIVOT? (Выберите все подходящие варианты.)
 - A. Определение вычисления в качестве входа для статистической функции.
 - B. Определение вычисления в качестве распределяющего элемента.
 - C. Использование подзапроса в предложении IN.
 - D. Использование нескольких статистических функций.
3. Какой тип данных имеет столбец целевых значений в результате оператора UNPIVOT?
 - A. INT.
 - B. NVARCHAR(128).
 - C. SQL_VARIANT.
 - D. Тип данных исходных столбцов, к которым применяется отмена сведения.

Занятие 3. Использование оконных функций

Как и групповые функции, оконные функции также позволяют выполнять вычисления для анализа данных. Они различаются по способу определения набора строк, обрабатываемых функцией. В групповых функциях группирующие запросы используются для того, чтобы организовать строки запросов в группы, после чего групповые функции применяются к каждой группе. Вы получаете одну результирующую строку на группу, а не на базовую строку. Таким образом, вы определяете набор строк на функцию — и затем возвращаете одно результирующее значение на каждую базовую строку и функцию. Набор строк для функции, с которыми предстоит работать, определяется с помощью выражения OVER.

В этом занятии рассматриваются три вида оконных функций: агрегатная, ранжирующая и функция смещения.

Изучив материал этого занятия, вы сможете:

- ✓ Использовать статистические оконные функции, ранжирующие функции и функции смещения
- ✓ Определять секционирование, упорядочение и кадрирование в оконных функциях

Продолжительность занятия — 60 минут.

Статистические оконные функции

Статистические (агрегатные) оконные функции — это то же самое, что групповые агрегатные функции (например, SUM, COUNT, AVG, MIN и MAX), за исключением того, что статистические оконные функции применяются к окну строк, определенному предложением OVER.

Одно из преимуществ использования оконных функций по сравнению с группирующими запросами в том, что оконные запросы не скрывают детализацию, они возвращают одну строку на каждую строку базового запроса. Это означает, что можно использовать в одном запросе, и даже в одном выражении, детализированные и агрегатные элементы. С помощью предложения OVER можно определить набор строк, с которым будет работать функция, для каждой базовой строки. Другими словами, оконный запрос определяет окно строк на каждую функцию и строку базового запроса.

Как уже упоминалось, для определения окна строк функции используется предложение OVER. Окно определяется применительно к текущей строке. Если используются пустые круглые скобки, предложение OVER представляет полностью результирующий набор базового запроса. Например, выражение `SUM(val) OVER()` представляет общее количество всех строк в базовом запросе. Предложение секционирования окна используется для ограничения этого окна. Например, выражение `SUM(val) OVER(PARTITION BY custid)` представляет итоговую сумму для конкретного клиента. Если в текущей строке идентификатор клиента равен 1, предложение OVER clause отфильтрует только те строки из результирующего набора базового запроса, в которых ID клиента равен 1; следовательно, выражение возвратит итоговую сумму для клиента с номером 1.

Далее приведен пример запроса к представлению `Sales.OrderValues`, возвращающий для каждого заказа ID клиента, ID заказа и стоимость заказа; используя оконную функцию, запрос также возвращает общую сумму всех заказов и сумму заказов для клиента.

```
SELECT custid, orderid, val,
       SUM(val) OVER(PARTITION BY custid) AS custtotal,
       SUM(val) OVER() AS grandtotal
  FROM Sales.OrderValues;
```

Этот запрос генерирует следующий результат (показанный здесь в сокращенном виде):

custid	orderid	val	custtotal	grandtotal
1	10643	814.50	4273.00	1265793.22
1	10692	878.00	4273.00	1265793.22
1	10702	330.00	4273.00	1265793.22
1	10835	845.80	4273.00	1265793.22
1	10952	471.20	4273.00	1265793.22
1	11011	933.50	4273.00	1265793.22
2	10926	514.40	1402.95	1265793.22

```

2      10759    320.00  1402.95   1265793.22
2      10625    479.75  1402.95   1265793.22
2      10308    88.80   1402.95   1265793.22
...

```

Общая сумма (`grandtotal`), разумеется, одинакова для всех строк. Сумма по клиенту одинакова для всех строк, имеющих одинаковый ID клиента.

В одном и том же выражении можно одновременно использовать строки детализации и оконные агрегаты. Например, следующий запрос вычисляет для каждого заказа процент стоимости текущего заказа от суммарной стоимости по клиенту, а также процент от полной стоимости.

```

SELECT custid, orderid, val,
       CAST(100.0 * val / SUM(val) OVER(PARTITION BY custid) AS NUMERIC(5, 2)) AS pctcust,
       CAST(100.0 * val / SUM(val) OVER() AS NUMERIC(5, 2)) AS pcttotal
  FROM Sales.OrderValues;

```

Этот запрос генерирует следующий результат (показанный здесь в сокращенном виде):

custid	orderid	val	pctcust	pcttotal
1	10643	814.50	19.06	0.06
1	10692	878.00	20.55	0.07
1	10702	330.00	7.72	0.03
1	10835	845.80	19.79	0.07
1	10952	471.20	11.03	0.04
1	11011	933.50	21.85	0.07
2	10926	514.40	36.67	0.04
2	10759	320.00	22.81	0.03
2	10625	479.75	34.20	0.04
2	10308	88.80	6.33	0.01
...				

Сумма всех процентов от общей суммы равна 100. Сумма всех процентов от общей стоимости по клиенту равна 100 для каждой секции строк, включающей одного и того же клиента.

Оконные агрегатные функции поддерживают еще одну возможность фильтрации, называемую кадрированием. Смысл ее заключается в том, что определяется упорядочение внутри секции с помощью предложения оконного кадра и затем, на основе этого упорядочения, можно заключить набор строк между двумя границами. Границы определяются с помощью предложения оконного кадра. Предложение оконного кадра требует, чтобы предложение упорядочивания окна обязательно присутствовало, поскольку множество не имеет порядка, а без него ограничение строк между двумя границами бессмысленно.

В предложении оконного кадра указываются единицы измерения (`ROWS` или `RANGE`) и экстент оконного кадра (смещение границ по отношению к текущей строке). Ис-

пользуя предложение `ROWS`, можно указать границы как один из следующих параметров:

- `UNBOUNDED PRECEDING` или `FOLLOWING`, означающие начало или конец секции соответственно;
- `CURRENT ROW`, представляющий текущую строку;
- `<н> ROWS PRECEDING` или `FOLLOWING`, означающие *n* строк перед или после текущей строки соответственно.

Предположим, вам нужно направить запрос к представлению `Sales.OrderValues` и вычислить промежуточные значения суммы от начала деятельности текущего клиента и до текущего заказа. Необходимо использовать статистическую функцию `SUM`. Вам необходимо секционировать окно по значению `custid`. Вам также нужно упорядочить это окно по значениям `orderdate` и `orderid`. Затем нужно наложить кадр на строки с начала секции (`UNBOUNDED PRECEDING`) до текущей строки. Ваш запрос должен выглядеть следующим образом:

```
SELECT custid, orderid, orderdate, val,
       SUM(val) OVER(PARTITION BY custid
                      ORDER BY orderdate, orderid
                      ROWS BETWEEN UNBOUNDED PRECEDING
                               AND CURRENT ROW) AS runningtotal
  FROM Sales.OrderValues;
```

Этот запрос генерирует следующий результат (показанный здесь в сокращенном виде):

custid	orderid	orderdate	val	runningtotal
1	10643	2007-08-25	814.50	814.50
1	10692	2007-10-03	878.00	1692.50
1	10702	2007-10-13	330.00	2022.50
1	10835	2008-01-15	845.80	2868.30
1	10952	2008-03-16	471.20	3339.50
1	11011	2008-04-09	933.50	4273.00
2	10308	2006-09-18	88.80	88.80
2	10625	2007-08-08	479.75	568.55
2	10759	2007-11-28	320.00	888.55
2	10926	2008-03-04	514.40	1402.95
...				

Обратите внимание, как накапливаются значения с начала клиентской секции и до текущей строки. Кстати, вместо полной формы ограничителя кадра `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` можно использовать краткую форму `ROWS UNBOUNDED PRECEDING`, сохраняя тот же смысл.

Используя оконные агрегатные функции для выполнения таких вычислений, как промежуточная сумма, вы получите значительно лучшую производительность по сравнению с объединениями или подзапросами и групповыми агрегатными функ-

циями. Оконные функции способны давать хорошую оптимизацию — особенно при использовании UNBOUNDED PRECEDING в качестве первого разделителя.

С точки зрения логической обработки запросов, результат запроса будет получен, когда достигается фаза SELECT — после того, как будут обработаны фазы FROM, WHERE, GROUP BY И HAVING.

Поскольку предполагается, что оконные функции оперируют результирующим набором базового запроса, они разрешены только в предложениях SELECT И ORDER BY. Если необходимо ссылаться на результат оконной функции в предложении, оцениваемом перед предложением SELECT, нужно использовать табличное выражение. Оконная функция вызывается в предложении SELECT внутреннего запроса, присваивая выражению псевдоним столбца. Затем можно ссылаться на этот столбец во внешнем запросе во всех предложениях.

Например, пусть вам нужно отфильтровать результат запроса, возвратив только те строки, в которых промежуточная сумма меньше 1000,00. Следующий код решает эту задачу путем определения обобщенного табличного выражения (common table expression, CTE), используя предыдущий запрос и затем выполняя фильтрацию во внешнем запросе.

```
WITH RunningTotals AS
( SELECT custid, orderid, orderdate, val,
       SUM(val) OVER(PARTITION BY custid
                      ORDER BY orderdate, orderid
                     ROWS BETWEEN UNBOUNDED PRECEDING
                           AND CURRENT ROW) AS runningtotal
    FROM Sales.OrderValues )
SELECT *
  FROM RunningTotals
 WHERE runningtotal < 1000.00;
```

Этот запрос генерирует следующий результат (показанный здесь в сокращенном виде):

custid	orderid	orderdate	val	runningtotal
1	10643	2007-08-25	814.50	814.50
2	10308	2006-09-18	88.80	88.80
2	10625	2007-08-08	479.75	568.55
2	10759	2007-11-28	320.00	888.55
3	10365	2006-11-27	403.20	403.20
...				

Еще один пример ограничителя оконного кадра: если было бы нужно, чтобы кадр включал только последние 3 строки, следовало бы использовать формат ROWS BETWEEN 2 PRECEDING AND CURRENT ROW.

Что касается ограничителя оконного кадра RANGE, согласно стандартному языку SQL, он позволяет определять ограничители на основе логических смещений от ключа сортировки текущей строки. Помните, что предложение ROWS определяет

ограничители — число строк от текущей строки, — на основе физических смещений. Однако в SQL Server 2012 представлена очень узкая реализация опции RANGE, поддерживающая в качестве ограничителей только UNBOUNDED PRECEDING или FOLLOWING и CURRENT ROW. Различие между предложениями ROWS и RANGE при использовании одинаковых ограничителей таково: первое не включает одноранговые строки (связанные строки в терминах ключа сортировки), тогда как второй включает.

ВАЖНО! Сравнение предложений ROWS и RANGE

В SQL Server 2012 предложение ROWS обычно оптимизируется значительно лучше, чем RANGE при использовании тех же ограничителей. Если окно определено с помощью предложения упорядочивания окна, но без предложения оконного кадра, по умолчанию принимается RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. Поэтому, если не требуется специальное поведение, обеспеченное опцией RANGE, которая включает одноранговые строки, убедитесь в том, что вы явно используете опцию ROWS.

Ранжирующие оконные функции

С помощью ранжирующих оконных функций можно ранжировать строки внутри секций, используя указанное упорядочение. Как в случае других оконных функций, если не указано предложение секционирования окна, полный результат базового запроса рассматривается в качестве одной секции. Предложение порядка окна является обязательным. Ранжирующие оконные функции не поддерживают предложение оконного кадра. T-SQL поддерживает четыре ранжирующие оконные функции: ROW_NUMBER, RANK, DENSE_RANK и NTILE.

Следующий запрос демонстрирует использование этих функций.

```
SELECT custid, orderid, val,
       ROW_NUMBER() OVER(ORDER BY val) AS rownum,
       RANK()          OVER(ORDER BY val) AS rnk,
       DENSE_RANK()    OVER(ORDER BY val) AS densernk,
       NTILE(100)      OVER(ORDER BY val) AS ntile100
  FROM Sales.OrderValues;
```

Этот запрос генерирует следующий результат (показанный здесь в сокращенном виде):

custid	orderid	val	rownum	rnk	densernk	ntile100
12	10782	12.50	1	1	1	1
27	10807	18.40	2	2	2	1
66	10586	23.80	3	3	3	1
76	10767	28.00	4	4	4	1
54	10898	30.00	5	5	5	1
88	10900	33.75	6	6	6	1
48	10883	36.00	7	7	7	1
41	11051	36.00	8	7	7	1
71	10815	40.00	9	8	8	1

38	10674	45.00	10	10	9	2
53	11057	45.00	11	10	9	2
75	10271	48.00	12	12	10	2
...						

ВАЖНО!**Упорядочение представления или упорядочение окна**

Поскольку рассматриваемый запрос не содержит предложение ORDER BY для представления данных, то нет гарантии, что строки будут представлены в каком-то определенном порядке. Предложение порядка окна определяет только упорядочение для вычисления оконной функции. Если вы вызываете оконную функцию в запросе, но не указываете предложение ORDER BY для представления, невозможно гарантировать, что строки будут представлены в том же порядке, как для оконной функции. Если такая гарантия необходима, следует добавить предложение ORDER BY для представления данных.

Функция ROW_NUMBER вычисляет уникальный последовательный номер строки, начиная с 1, в секции окна на основании сортировки окна. Поскольку рассматриваемый в примере запрос не содержит предложение секционирования окна, эта функция рассматривает результирующий набор как одну секцию; следовательно, функция присваивает уникальные номера строк для всего результирующего набора запроса.

Обратите внимание, если упорядочение не является уникальным, функция ROW_NUMBER будет недетерминированной. Например, в результате данного примера две строки имеют одинаковое значение упорядочения равное 36.00, но эти строки получают разные номера строк (`rownum`). Причина этого в том, что функция должна генерировать уникальные номера внутри секции. Поскольку разрыв связей не задается явно, выбор, какая строка получает больший номер строки, является случайным (зависящим от оптимизации). Если нужен детерминированный результат (с гарантией повторяющихся результатов), необходимо добавить опцию разрыва связей. Например, чтобы упорядочение было уникальным, можно добавить первичный ключ, как в случае ORDER BY `val, orderid`.

Функции RANK и DENSE_RANK отличаются от функции ROW_NUMBER в том смысле, что они присваивают одно и то же ранжирующее значение всем строкам, имеющим одинаковое значение сортировки. Функция RANK возвращает число строк в секции, которые имеют более низкое значение сортируемого столбца, чем текущее, плюс 1. Например, рассмотрим строки в результате нашего учебного примера, которые имеют значение `val`, равное 45.00. Девять предшествующих строк имеют значения, меньшие 45.00; следовательно, текущая строка получает ранг 10 (9 + 1).

Функция DENSE_RANK возвращает количество предшествующих отличающихся значений упорядочиваемого столбца с более низким значением, чем текущее, плюс 1. Например, те же строки, которые получили ранг 10, получают плотный ранг (dense rank), равный 9. Причина этого в том, что эти строки имеют значение сортировки равное 45.00, и при этом имеется 9 различных значений сортировки, более низких, чем 45.00. Поскольку функция RANK рассматривает строки, а функция DENSE_RANK — отличающиеся значения, первая может иметь промежутки между результирующими значениями сортировки, а последняя нет. Так как функции RANK и DENSE_RANK вычисляют одинаковое ранжирующее значение для строк с одинаковым значением

сортировки, обе функции, детерминированные даже тогда, когда упорядочение не является уникальным. Фактически при использовании уникальной сортировки обе функции возвращают тот же самый результат, что и функция `ROW_NUMBER`. Поэтому обычно эти функции интересны тогда, когда сортировка не является уникальной.

Функция `NTILE` позволяет организовать строки внутри секции в запрашиваемое количество групп одинакового размера на основе указанной сортировки. Желаемое количество групп задается как вход для этой функции. В нашем учебном запросе мы запрашивали 100 групп. В результирующем наборе — 830 строк, и, следовательно, базовый размер группы составляет $830/100 = 8$ с остатком 30. Поскольку есть остаток, равный 30, первые 30 групп содержат по дополнительной строке. А именно, группы 1—30 будут иметь по 9 строк, и все оставшиеся группы (от 31 до 100) — по 8 строк. Посмотрите, в результате этого запроса первым 9 строкам (в соответствии с сортировкой по столбцу `val`) присвоен номер группы 1, затем следующим 9 строкам присвоен номер группы 2 и т. д. Подобно функции `ROW_NUMBER` функция `NTILE` недетерминированная, если сортировка не является уникальной. Для гарантии детерминизма следует определить уникальную сортировку.

СОВЕТ

Подготовка к экзамену

Как говорилось при обсуждении статистических оконных функций, оконные функции разрешены только в предложениях `SELECT` и `ORDER BY` запроса. Если нужно на них сослаться в других предложениях, например в предложении `WHERE`, следует использовать табличное выражение, такое как `CTE`. Тогда нужно вызвать оконную функцию в предложении `SELECT` внутреннего запроса, присвоив выражению псевдоним столбца. Затем можно сослаться на этот псевдоним столбца в предложении `WHERE` внешнего запроса. Вы можете попробовать этот способ в заданиях к этому занятию.

Оконные функции смещения

Оконные функции смещения возвращают элемент строки, которая находится на указанном смещении от текущей строки в секции окна или от первой либо последней строки оконного кадра. T-SQL поддерживает следующие оконные функции смещения: `LAG`, `LEAD`, `FIRST_VALUE` и `LAST_VALUE`.

Функции `LAG` и `LEAD` используют смещение относительно текущей строки, а функции `FIRST_VALUE` и `LAST_VALUE` работают с первой или последней строкой окна соответственно.

Функции `LAG` и `LEAD` поддерживают предложения секционирования и упорядочения окна. Но они не поддерживают предложение оконного кадра. Функция `LAG` возвращает элемент строки в текущей секции, который является запрашиваемым числом строк перед данной строкой (на основании сортировки окна), при этом смещение по умолчанию принимается равным 1. Функция `LEAD` возвращает элемент строки, который находится на запрашиваемом смещении после текущей строки.

Рассмотрим в качестве примера следующий запрос, который использует функции `LAG` и `LEAD` для возвращения с каждым заказом значения заказа предыдущего клиента, а также значения заказа следующего клиента.

```

SELECT custid, orderid, orderdate, val,
       LAG(val) OVER(PARTITION BY custid
                      ORDER BY orderdate, orderid) AS prev_val,
       LEAD(val) OVER(PARTITION BY custid
                      ORDER BY orderdate, orderid) AS next_val
FROM Sales.OrderValues;

```

Этот запрос генерирует следующий результат (показанный здесь в сокращенном виде):

custid	orderid	orderdate	val	prev_val	next_val
1	10643	2007-08-25	814.50	NULL	878.00
1	10692	2007-10-03	878.00	814.50	330.00
1	10702	2007-10-13	330.00	878.00	845.80
1	10835	2008-01-15	845.80	330.00	471.20
1	10952	2008-03-16	471.20	845.80	933.50
1	11011	2008-04-09	933.50	471.20	NULL
2	10308	2006-09-18	88.80	NULL	479.75
2	10625	2007-08-08	479.75	88.80	320.00
2	10759	2007-11-28	320.00	479.75	514.40
2	10926	2008-03-04	514.40	320.00	NULL
...					

Поскольку явное смещение не указано, обе функции используют в качестве него значение по умолчанию, равное 1. При желании иметь смещение, отличное от 1, следует указать его вторым аргументом, как в случае `LAG(val, 3)`.

Заметьте, что если строка не существует на запрашиваемом смещении, функция возвращает по умолчанию значение `NULL`. Если в этом случае нужно возвращать другое значение, его следует указать в качестве третьего элемента, как в случае `LAG(val, 3, 0)`.

Функции `FIRST_VALUE` и `LAST_VALUE` возвращают выражение значения из первой или последней строки в оконном кадре соответственно. Эти функции, естественно, поддерживают предложения секционирования, сортировки и оконного кадра. Например, следующий запрос возвращает, наряду с каждым заказом, значения первого и последнего заказа клиента.

```

SELECT custid, orderid, orderdate, val,
       FIRST_VALUE(val) OVER(PARTITION BY custid
                                  ORDER BY orderdate, orderid
                                  ROWS BETWEEN UNBOUNDED PRECEDING
                                         AND CURRENT ROW) AS first_val,
       LAST_VALUE(val) OVER(PARTITION BY custid
                                 ORDER BY orderdate, orderid
                                 ROWS BETWEEN CURRENT ROW
                                         AND UNBOUNDED FOLLOWING) AS last_val
FROM Sales.OrderValues;

```

Этот запрос генерирует следующий результат (показанный здесь в сокращенном виде):

custid	orderid	orderdate	val	first_val	last_val
1	11011	2008-04-09	933.50	814.50	933.50
1	10952	2008-03-16	471.20	814.50	933.50
1	10835	2008-01-15	845.80	814.50	933.50
1	10702	2007-10-13	330.00	814.50	933.50
1	10692	2007-10-03	878.00	814.50	933.50
1	10643	2007-08-25	814.50	814.50	933.50
2	10926	2008-03-04	514.40	88.80	514.40
2	10759	2007-11-28	320.00	88.80	514.40
2	10625	2007-08-08	479.75	88.80	514.40
2	10308	2006-09-18	88.80	88.80	514.40
...					

ВАЖНО!

Кадр по умолчанию

и производительность параметра RANGE

Напомним, что когда оконный кадр применим к функции, но явно не указано предложение оконного кадра, по умолчанию используется выражение RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. Для улучшения производительности, как правило, рекомендуется избегать опции RANGE; для этого требуется явное задание предложения ROWS. Также, если вы хотите получить первую строку в секции, использование функции FIRST_VALUE с рамкой по умолчанию, по меньшей мере, даст правильный результат. Однако если вы хотите получить последнюю строку в секции, использование функции LAST_VALUE с рамкой по умолчанию не даст желаемого результата, поскольку последняя строка в рамке по умолчанию — это текущая строка. Таким образом, используя функцию LAST_VALUE, нужно явно задавать рамку окна, для того чтобы получить то, что вам нужно. И если вам требуется элемент из последней строки секции, вторым разделителем в рамке должен быть UNBOUNDED FOLLOWING.

КОНТРОЛЬНЫЕ ВОПРОСЫ

- Какие предложения поддерживаются различными видами оконных функций?
- Что представляют разделители UNBOUNDED PRECEDING и UNBOUNDED FOLLOWING?

Ответы на контрольные вопросы

- Предложения секционирования, упорядочения и кадрирования.
- Начало и конец секции соответственно.

ПРАКТИКУМ Использование оконных функций

В данном практикуме вам предстоит проверить ваши знания об оконных функциях.

Задание 1. Использование статистических оконных функций

В этом задании вам требуется написать запросы с помощью статистических оконных функций. Прежде чем обратиться к предложенному решению, попробуйте написать собственное.

1. Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
2. Напишите запрос к представлению `Sales.OrderValues`, который возвращает для каждого клиента и заказа значение скользящего среднего для трех последних заказов клиента.

Ваше решение должно быть подобно следующему запросу:

```
SELECT custid, orderid, orderdate, val,
       AVG(val) OVER(PARTITION BY custid
                      ORDER BY orderdate, orderid
                     ROWS BETWEEN 2 PRECEDING
                     AND CURRENT ROW) AS movingavg
  FROM Sales.OrderValues;
```

Этот запрос генерирует следующий результат (показанный здесь в сокращенном виде):

custid	orderid	orderdate	val	movingavg
1	10643	2007-08-25	814.50	814.500000
1	10692	2007-10-03	878.00	846.250000
1	10702	2007-10-13	330.00	674.166666
1	10835	2008-01-15	845.80	684.600000
1	10952	2008-03-16	471.20	549.000000
1	11011	2008-04-09	933.50	750.166666
2	10308	2006-09-18	88.80	88.800000
2	10625	2007-08-08	479.75	284.275000
2	10759	2007-11-28	320.00	296.183333
2	10926	2008-03-04	514.40	438.050000
...				

Задание 2. Использование оконных ранжирующих функций и функций смещения

В этом задании вам даны задачи, требующие написания запросов с помощью оконных ранжирующих функций и функций смещения. Вам необходимо отфильтровать строки на основании результата оконной функции и написать выражения, которые создают смесь элементов детализации и оконных функций.

1. В качестве следующего задания напишите запрос к таблице `Sales.Orders` и отфильтруйте три заказа с наибольшим значением затрат на транспортировку по каждому грузоотправителю, используя `orderid` в качестве критерия отбора.

Вам следует использовать функцию `ROW_NUMBER` для фильтрации желаемых строк. Но помните, что вы не можете ссылаться на оконные функции прямо в предложении `WHERE`.

Решение вопроса — определить табличное выражение на основании запроса, который вызывает функцию `ROW_NUMBER` и присваивает выражению псевдоним столбца. Затем вы можете выполнить фильтрацию во внешнем запросе с по-

мощью этого псевдонима столбца. Далее приведен запрос, содержащий законченное решение.

```
WITH C AS
( SELECT shipperid, orderid, freight,
    ROW_NUMBER() OVER(PARTITION BY shipperid
                      ORDER BY freight DESC, orderid) AS rounum
  FROM Sales.Orders )
SELECT shipperid, orderid, freight
FROM C
WHERE rounum <= 3
ORDER BY shipperid, rounum;
```

Этот запрос генерирует следующий результат:

	shipperid	orderid	freight
1	1	10430	458.78
1	1	10836	411.88
1	1	10658	364.15
2	2	10372	890.78
2	2	11030	830.75
2	2	10691	810.05
3	3	10540	1007.64
3	3	10479	708.95
3	3	11032	606.19

2. Ваше последнее задание — сделать запрос к представлению `Sales.OrderValues`. Вам необходимо рассчитать разницу между текущей стоимостью заказа и стоимостью предыдущего заказа данного клиента, а также разницу между текущей стоимостью заказа и стоимостью следующего заказа данного клиента.

Для получения стоимости предыдущего и последующего заказов вы можете использовать функции `LAG` и `LEAD` соответственно. Затем вы можете вычесть результаты этих функций из значений столбца `val` для получения желаемых разностей. Далее приведен запрос, представляющий законченное решение этой задачи.

```
SELECT custid, orderid, orderdate, val,
       val - LAG(val) OVER(PARTITION BY custid
                           ORDER BY orderdate, orderid) AS diffprev,
       val - LEAD(val) OVER(PARTITION BY custid
                           ORDER BY orderdate, orderid) AS diffnext
  FROM Sales.OrderValues;
```

Этот запрос генерирует следующий результат (представленный здесь в сокращенном виде):

	custid	orderid	orderdate	val	diffprev	diffnext
1	1	10643	2007-08-25	814.50	NULL	-63.50
1	1	10692	2007-10-03	878.00	63.50	548.00

1	10702	2007-10-13	330.00	-548.00	-515.80
1	10835	2008-01-15	845.80	515.80	374.60
1	10952	2008-03-16	471.20	-374.60	-462.30
1	11011	2008-04-09	933.50	462.30	NULL
2	10308	2006-09-18	88.80	NULL	-390.95
2	10625	2007-08-08	479.75	390.95	159.75
2	10759	2007-11-28	320.00	-159.75	-194.40
2	10926	2008-03-04	514.40	194.40	NULL

...

Резюме занятия

- Оконные функции выполняют аналитические вычисления данных. Они обрабатывают набор строк, определенный для каждой базовой строки, с помощью предложения `OVER`.
- Язык T-SQL поддерживает статистические, ранжирующие оконные функции, а также оконные функции смещения. Все оконные функции поддерживают предложения секционирования и сортировки. Статистические оконные функции, кроме `FIRST_VALUE` и `LAST_VALUE`, также поддерживают предложение оконного кадра.
- В отличие от группирующих запросов, которые скрывают строки детализации и возвращают только одну строку на группу, оконные запросы не скрывают эти строки. Они возвращают одну строку на каждую строку в базовом запросе и позволяют смешивать элементы детализации и оконные функции в один и тех же выражениях.

К СВЕДЕНИЮ Оконные функции

Более подробную информацию об оконных функциях, их оптимизации и практическом применении можно найти в книге "Microsoft SQL Server 2012. Высокопроизводительный код T-SQL. Оконные функции"¹.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какие границы используют по умолчанию оконные функции, когда указано предложение упорядочивания окна, но не указано явное предложение оконного кадра? (Выберите все подходящие варианты.)
 - A. `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`.
 - B. `ROWS UNBOUNDED PRECEDING`.

¹ Бен-Ган И. Microsoft SQL Server 2012. Высокопроизводительный код T-SQL. Оконные функции: Пер. с англ. — М.: Издательство "Русская редакция"; СПб.: БХВ-Петербург, 2013. — 256 стр. : ил.

- C. RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.
- D. RANGE UNBOUNDED PRECEDING.
2. Что вычисляют функции RANK и DENSE_RANK?
- A. Функция RANK возвращает количество строк, которые имеют меньшее значение упорядочения (при сортировке по возрастанию), чем текущее; функция DENSE_RANK возвращает число отличающихся значений упорядочения, меньших, чем текущее.
- B. Функция RANK возвращает число строк, имеющих меньшее значение упорядочения, чем текущее, плюс 1; функция DENSE_RANK возвращает число отличных значений упорядочения, меньших, чем текущее, плюс 1.
- C. Функция RANK возвращает число строк, имеющих меньшее значение упорядочения, чем текущее, минус 1; функция DENSE_RANK возвращает число отличных значений упорядочения, меньших, чем текущее, минус 1.
- D. Обе функции возвращают одинаковый результат, если сортировка не является уникальной.
3. Почему оконные функции разрешены только в предложениях запроса SELECT и ORDER BY?
- A. Поскольку предполагается, что они воздействуют на результат базового запроса, который получается, когда логическая обработка запроса достигает фазы SELECT.
- B. Потому что у компании Microsoft не было времени для их реализации в других предложениях.
- C. Потому что никогда не возникает необходимость фильтрации или группировки данных на основании результата оконных функций.
- D. Потому что в других предложениях эти функции рассматриваются как функции-лазейки (программы несанкционированного доступа, door-backdoor functions).

Упражнения

В следующих упражнениях вы примените полученные знания о группировании и оконных функциях. Ответы на вопросы можно найти в *приложении "Ответы" в конце книги*.

Упражнение 1. Усовершенствование операций анализа данных

Вы работаете аналитиком данных в финансовой компании, которая использует SQL Server 2012 для работы с базами данных. Компания недавно обновила СУБД с версии SQL Server 2000. Вы часто используете запросы T-SQL к базе данных компа-

ний для анализа данных. До сих пор вы были ограничены требованием, чтобы код был совместим с SQL Server 2000, в основном использующий соединения, подзапросы и сгруппированные запросы. Ваши запросы часто были сложными и медленно работали. Сейчас вы оцениваете возможность использования функциональности, доступной в SQL Server 2012.

1. Вам часто приходится вычислять такие вещи, как промежуточные суммы, выполнять расчеты с начала года по текущую дату и скользящее среднее. Что вы теперь можете выбрать для их обработки? Чего нужно остерегаться для обеспечения хорошей производительности?
2. Иногда вам нужно создавать сводные отчеты, где данные преобразуются из строк в столбцы или наоборот. До сих пор вы импортировали данные в Microsoft Excel и решали эти задачи там, но теперь вы предпочитаете делать это в T-SQL. Что вы рассматриваете как средство решения этой задачи? С чем нужно соблюдать осторожность при использовании рассматриваемой вами функциональности?
3. Во многих запросах вам нужно вычислять интервалы времени, т. е. указывать время, прошедшее между предыдущим и текущим событием или между текущим и последующим событием. До сих пор для этого вы использовали вложенные запросы. Что вы рассматриваете сейчас в качестве альтернативы?

Упражнение 2. Собеседование на вакансию разработчика

Вы проходите собеседование на вакансию разработчика T-SQL. Ответьте на следующие вопросы, задаваемые вам проводящим собеседование специалистом.

1. Опишите разницу между функциями `ROW_NUMBER` и `RANK`.
2. Опишите разницу между единицами рамки окна `ROWS` и `RANGE`.
3. Почему вы не можете ссылаться на оконную функцию в предложении `WHERE` запроса и как можно обойти эту ситуацию?

Рекомендуемые упражнения

Для успешного освоения экзаменационных тем, представленных в данной главе, выполните следующие задания.

Логическая обработка запросов

Чтобы проверить знания в области логической обработки запросов, укажите порядок, в котором оцениваются различные предложения запроса. Также перечислите предложения, в которых разрешены вычисления, изученные в данной главе.

- Задание 1.** К этому моменту вы должны быть знакомы со всеми основными предложениями запроса `SELECT`: `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`, `TOP` и `OFFSET...FETCH`. Укажите порядок, в котором эти предложения оценивают-

ся в соответствии с логической обработкой запросов. Также укажите предложения, в которых выполняются операторы PIVOT и UNPIVOT. Наконец, укажите предложения с разрешенными групповыми функциями и предложения, в которых разрешены оконные функции.

- **Задание 2.** Подумайте и укажите логические преимущества, которые имеют агрегатные оконные функции по сравнению с групповыми статистическими функциями и статистическими функциями, вычисляемыми в подзапросах.

ГЛАВА 6

Запросы с полнотекстовым поиском данных

Темы экзамена

- Работа с данными.
 - Запрос данных с помощью инструкций SELECT.
- Модификация данных.
 - Работа с функциями.

Трудно представить себе поиск в Интернете без современных средств поиска, таких как Bing или Google. Однако большинство современных приложений по-прежнему ограничивает пользователей возможностью точного поиска. Для конечных пользователей даже оператор `LIKE` стандартного языка SQL недостаточно эффективен для приближенного поиска. Кроме того, многие документы хранятся в современных базах данных; конечные пользователи, возможно, захотят иметь эффективные инструменты поиска по содержимому документов.

Microsoft SQL Server 2012 расширяет поддержку полнотекстового поиска, который был по сути доступен в предыдущих редакциях. В этой главе рассматривается полнотекстовый поиск, а также семантический поиск в базе данных SQL Server.

ПРЕЖДЕ ВСЕГО

Для полного освоения этой главы необходимо иметь:

- понимание реляционных принципов баз данных;
- опыт работы с SQL Server Management Studio (SSMS);
- некоторый опыт в написании кода на языке T-SQL;
- доступ к экземпляру SQL Server 2012 с установленной учебной базой данных TSQL2012;
- полнотекстовый поиск (Full-Text Search), установленный на вашем экземпляре SQL Server.

Занятие 1. Создание полнотекстовых каталогов и индексов

Полнотекстовый поиск (full-text search) позволяет выполнять приблизительный поиск в базах данных SQL Server 2012. Прежде чем начать использовать полнотекстовые предикаты и функции, необходимо создать *полнотекстовые индексы* внутри *полнотекстовых каталогов*. После создания полнотекстовых индексов на символьных столбцах в базе данных вы можете выполнять поиск:



- простых выражений* (simple term), т. е. одного или нескольких определенных слов или фраз;
- префиксных выражений* (prefix terms), которые являются выражениями, с которых начинаются слова или фразы;
- производных выражений* (generation terms), означающих словоформы (inflectional forms) слов;
- выражений с учетом расположения* (proximity terms), или слов либо фраз, расположенных рядом с другими словами или фразами;
- выражения тезауруса* (thesaurus terms), или синонимы слова;
- взвешенные выражения* (weighted terms), представляющие собой слова или фразы, использующие пользовательское значение веса;
- статистический семантический поиск* (statistical semantic search) или поиск ключевых фраз в документе;
- подобные документы*, в которых подобие определяется с помощью выражений семантического ключа.



Изучив материал этого занятия, вы сможете:

- ✓ Создавать полнотекстовые каталоги и индексы
- ✓ Разрешать статистическое семантическое индексирование

Продолжительность занятия — 60 минут.

Компоненты полнотекстового поиска

Чтобы начать использовать полнотекстовый поиск, необходимо хорошо понимать его компоненты. Прежде всего, следует проверить, установлен ли компонент Full-Text Search (Полнотекстовый поиск), выполнив следующий запрос:

```
SELECT SERVERPROPERTY('IsFullTextInstalled');
```

Если компонент Full-Text Search не установлен, вы должны перезапустить программу установки SQL Server.

Можно создавать полнотекстовые индексы на столбцах типов CHAR, VARCHAR, NCHAR, NVARCHAR, TEXT, NTEXT, IMAGE, XML и VARBINARY(MAX). Помимо применения полнотек-

стовых индексов к символьным данным SQL Server, можно сохранять целые документы в двоичных или XML-столбцах и использовать полнотекстовые запросы на этих документах. Столбцы с типом данных VARBINARY (MAX), IMAGE или XML требуют дополнительного столбца, в котором хранится расширение файла (docx, pdf или xlsx) документа в каждой строке.

Для документов нужны соответствующие фильтры. Фильтры, которые в терминологии полнотекстового поиска называются iFilters, извлекают текстовую информацию и удаляют в документе форматирование. Можно проверить, какие фильтры установлены в вашем экземпляре SQL Server с помощью следующего запроса:

```
EXEC sys.sp_help_fulltext_system_components 'filter';
```

Кроме системной хранимой процедуры, для проверки, какой фильтр установлен на вашем экземпляре SQL Server, можно использовать представление каталога sys.fulltext_document_types следующим образом:

```
SELECT document_type, path  
FROM sys.fulltext_document_types;
```

По умолчанию используется множество популярных форматов. Можно установить дополнительные фильтры, такие как фильтры для форматов документов Microsoft Office 2010. Загрузить пакет фильтров Microsoft Office 2010 можно по этому адресу: <http://www.microsoft.com/en-us/download/details.aspx?id=17062>.

После установки пакета фильтров на компьютер, необходимо зарегистрировать эти фильтры в SQL Server с помощью следующей команды:

```
EXEC sys.sp_fulltext_service 'load_os_resources', 1;
```

Возможно, после этого потребуется перезапустить SQL Server. После этого проверьте, были ли фильтры успешно установлены, снова применив системную процедуру sys.sp_help_fulltext_system_components.

Средства разбиения по словам (word breakers) и парадигматические модули (stemmers) выполняют лингвистический анализ полнотекстовых данных. Поскольку правила изменяются от языка к языку, средства разбиения по словам и парадигматические модули зависят от языка. *Средство разбиения по словам* определяет отдельные слова (*токены*). Токены вставляются в полнотекстовый индекс в сжатом формате. *Парадигматический модуль* генерирует гибкие формы слова на основании правил данного языка. Для выяснения, какие языки поддерживаются SQL Server, можно использовать следующий запрос:

```
SELECT lcid, name  
FROM sys.fulltext_languages  
ORDER BY name;
```

Парадигматические модули зависят от языка. Если вы используете локализованную версию SQL Server, компонент SQL Server Setup устанавливает в качестве полнотекстового языка по умолчанию язык вашего экземпляра, если этот язык поддерживается на вашем экземпляре. Если же нет или вы используете нелокализованную версию SQL Server, полнотекстовым языком по умолчанию устанавливается анг-



лийский язык. Для каждого полнотекстового индексированного столбца можно указать свой язык. Изменить язык по умолчанию можно с помощью системной процедуры sys.sp_configure.

Средства разбиения по словам также зависят от языка. Если средство разбиения по словам не существует для языка вашего экземпляра SQL Server, применяется нейтральное средство разбиения по словам, которое использует только нейтральные символы для разбиения текста на отдельные слова.

Представьте себе, что у вас есть документы по SQL Server. Фраза "SQL Server", вероятно, появляется в каждом документе. Такая фраза не поможет вам с поиском, однако она приведет к чрезмерному увеличению полнотекстового индекса. Предотвратить это можно с помощью создания *стоп-списков*, содержащих *стоп-слова*. Проверить текущие стоп-слова и стоп-списки в вашей базе данных можно с помощью следующих запросов:

```
SELECT stoplist_id, name  
FROM sys.fulltext_stoplists;  
SELECT stoplist_id, stopword, language  
FROM sys.fulltext_stopwords;
```

Полнотекстовые запросы могут выполнять поиск не только слов, которые указаны в запросе; они также могут искать синонимы слов. SQL Server ищет синонимы в файле *тезауруса*. Каждый язык имеет свой XML-файл тезауруса. Файлы тезауруса для экземпляра по умолчанию находятся по адресу SQL_Server_install_path\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSSQL\FTDATA\.

Можно вручную редактировать каждый файл тезауруса и конфигурировать следующие элементы:

- diacritics_sensitive** (с учетом диакритических знаков). Установите этот элемент в значение 0, если язык не использует диакритические знаки, или в значение 1, если использует;
- expansion** (расширение). Этот элемент используется для добавления расширяющих слов к исходному слову. Например, можно добавить расширяющее слово "автор" к слову "писатель", чтобы искать также и слово "автор", когда конечный пользователь ищет слово "писатель";
- replacement** (замена). Этот элемент следует использовать для определения замещающих слов или терминов для определенного слова или термина. Например, "Windows 2008" может быть замещением для "Win 2k8". В таком случае SQL Server будет искать "Windows 2008", хотя в качестве элемента поиска будет использоваться "Win 2k8".

После редактирования файла тезауруса для определенного языка его нужно загрузить с помощью вызова следующей системной процедуры:

```
EXEC sys.sp_fulltext_load_thesaurus_file 1033;
```

Параметр этой процедуры указывает ID языка; в данном случае это 1033, что означает язык US English.



Полнотекстовые запросы могут также выполнять поиск в свойствах документа. В каких свойствах документа выполнять поиск, зависит от фильтра документа. Можно создать *список свойств поиска* (search property list) для определения доступных для поиска свойств ваших документов. Вы можете включить в него свойства, которые определенный фильтр может извлекать из документа.

COBET**Подготовка к экзамену**

Хотя полнотекстовый поиск не входит в число тем экзамена, могут быть косвенные вопросы на эту тему. Помните, что полнотекстовые предикаты могут входить в предложение WHERE запроса.

Создание и управление полнотекстовыми каталогами и индексами

После того как вся инфраструктура полнотекстового поиска настроена, можно начинать его использовать. Полнотекстовые индексы хранятся в полнотекстовых каталогах. Полнотекстовый каталог — это виртуальный объект, контейнер для полнотекстовых индексов. Как виртуальный объект, он не принадлежит ни к одной файловой группе.

Далее приведен синтаксис для создания полнотекстового каталога.

```
CREATE FULLTEXT CATALOG catalog_name
    [ON FILEGROUP filegroup ]
    [IN PATH 'rootpath']
    [WITH <catalog_option>]
    [AS DEFAULT]
    [AUTHORIZATION owner_name ]
<catalog_option> ::= ACCENT_SENSITIVITY = {ON|OFF}
```

Параметры ON FILEGROUP и IN PATH предназначены для обеспечения обратной совместимости с SQL Server 2008 и более ранними версиями и не влияют на SQL Server 2012; вам их использовать не нужно. Параметр ACCENT_SENSITIVITY определяет, учитывают ли полнотекстовые индексы в этом каталоге диакритические знаки или нет. Если вы позже измените этот параметр, вам нужно будет перестроить все полнотекстовые индексы в данном каталоге.

Для того чтобы перестроить полнотекстовый каталог, используйте инструкцию ALTER FULLTEXT CATALOG, а для его удаления — инструкцию DROP FULLTEXT CATALOG.

После того как полнотекстовый каталог создан, можно создать соответствующие полнотекстовые индексы. Далее приведен синтаксис для создания полнотекстового индекса.

```
CREATE FULLTEXT INDEX ON table_name
    [ ( { column_name
        [ TYPE COLUMN type_column_name ]
        [ LANGUAGE language_term ]
        [ STATISTICAL_SEMANTICS ]
    } [,...n] ) ]
```

```

KEY INDEX index_name
[ ON <catalog_filegroup_option> ]
[ WITH [ ( ) <with_option> [,...n] ( ) ] ]
[ ; ]

<catalog_filegroup_option>::=
{ fulltext_catalog_name
| ( fulltext_catalog_name, FILEGROUP filegroup_name)
| ( FILEGROUP filegroup_name, fulltext_catalog_name)
| ( FILEGROUP filegroup_name )

}

<with_option>::=
{ CHANGE_TRACKING [ = ] { MANUAL | AUTO | OFF [, NO POPULATION ] }
| STOPLIST [ = ] { OFF | SYSTEM | stoplist_name }
| SEARCH PROPERTY LIST [ = ] property_list_name
}

```

Большинство параметров интуитивно понятны. Вы узнаете о них в практикумах к этому занятию. Далее перечислены некоторые из параметров:

- **KEY INDEX *index_name*.** Это имя индекса уникального ключа по таблице. Это должен быть уникальный столбец с одним ключом, не допускающий неопределенных (*NULL*) значений. Рекомендуется использовать тип данных *integer*.
- **CHANGE_TRACKING [=] {MANUAL | AUTO | OFF [, NO POPULATION]}.** Этот параметр указывает, будет ли SQL Server обновлять полнотекстовый индекс автоматически. Для отслеживания изменений SQL Server использует механизм отслеживания изменений.
- **STATISTICAL_SEMANTICS.** Этот параметр создает дополнительные индексы ключевых фраз и подобия документов, являющиеся частью статистического семантического индексирования.

Последний из перечисленных параметров, **STATISTICAL_SEMANTICS**, заслуживает более подробного объяснения. Статистический семантический поиск дает возможность глубже анализировать документы путем извлечения и индексирования статистически релевантных ключевых фраз. Полнотекстовый поиск использует эти ключевые фразы для идентификации и индексирования подобных или связанных документов. Запрос к этим семантическим индексам осуществляется с помощью трех функций набора строк T-SQL для извлечения результатов в виде структурированных данных. Вы будете использовать эти функции в практикумах данной главы. Семантический поиск расширяет функциональность полнотекстового поиска. Он дает возможность запрашивать значение документа. Например, можно запросить индекс ключевых фраз для построения классификации документов. Или можно запросить индекс подобия документов для определения резюме, соответствующих описанию вакансии. Семантический поиск предоставляет возможность создавать собственное решение для интеллектуального анализа текста. Семантический поиск может представлять особый интерес в сочетании с компонентами интеллектуального анализа данных службы SQL Server Integration Services (SSIS).



Чтобы использовать семантический поиск, у вас должен быть установлен компонент Full Text Search (Полнотекстовый поиск). Кроме этого, необходимо установить базу данных семантической статистики (Semantic Language Statistics Database). Вы будете ее устанавливать в практикуме к данному занятию.

КОНТРОЛЬНЫЙ ВОПРОС

- Можно сохранить индексы из одного полнотекстового каталога в разных файловых группах?

Ответ на контрольный вопрос

- Да. Полнотекстовый каталог — это виртуальный объект; полнотекстовые индексы представляют собой физические объекты. Вы можете сохранить каждый полнотекстовый индекс из одного и того же каталога в разных файловых группах.

ПРАКТИКУМ Создание полнотекстового индекса

В этом практикуме вам нужно создать таблицу, заполнить ее документами и текстовыми данными, создать полнотекстовый каталог и индекс на этой таблице. Данный практикум предполагает, что на вашем экземпляре сервера в качестве языка по умолчанию установлен US English (английский США).

Задание 1. Создание таблицы и полнотекстовых компонентов

В этом задании вам нужно создать демонстрационную таблицу, заполнить ее каким-либо демо-текстом и затем создать стоп-слова и стоп-список и свойства поиска документа.

1. Запустите SQL Server Management Studio (SSMS) и подключитесь к вашему экземпляру SQL Server.
2. Откройте новое окно запроса, нажав кнопку **New Query** (Новый запрос).
3. Измените контекст на базу данных TSQL2012.
4. Проверьте, установлен ли компонент Full-Text Search, запустив следующий запрос:

```
SELECT SERVERPROPERTY('IsFullTextInstalled');
```

5. Если компонент Full-Text Search не установлен, запустите программу SQL Server Setup (Установка SQL Server) и установите этот компонент. Также установите пакеты фильтров Microsoft Office 2010.
6. Создайте таблицу, которую вы будете использовать для полнотекстового поиска. Создайте ее в схеме `dbo` и назовите `Documents`. Используйте данные табл. 6.1 для столбцов вашей таблицы `dbo.Documents`.

Таблица 6.1. Информация о столбцах таблицы *dbo.Documents*

Имя столбца (Column name)	Тип данных (Data type)	Возможность NULL-значений (Nullability)	Комментарий
id	INT	NOT NULL	IDENTITY, PRIMARY KEY
title	NVARCHAR(100)	NOT NULL	Названия документов, которые вы будете импортировать
doctype	NCHAR (4)	NOT NULL	Типы документов, которые вы будете импортировать
docexcerpt	NVARCHAR(1000)	NOT NULL	Фрагменты документов, которые вы будете импортировать
doccontent	VARBINARY (MAX)	NOT NULL	Документы, которые вы будете импортировать

Используйте следующий код для создания таблицы:

```
CREATE TABLE dbo.Documents
( id INT IDENTITY(1,1)      NOT NULL,
  title NVARCHAR(100)        NOT NULL,
  doctype NCHAR(4)           NOT NULL,
  docexcerpt NVARCHAR(1000)  NOT NULL,
  doccontent VARBINARY(MAX)  NOT NULL,
  CONSTRAINT PK_Documents PRIMARY KEY CLUSTERED(id) );
```

7. Импортируйте 4 документа, включенные в папку для данной книги. Если это папка C:\TK70461, можно использовать следующий код; в противном случае, измените папку в функциях OPENROWSET соответствующим образом.

```
INSERT INTO dbo.Documents (title, doctype, docexcerpt, doccontent)
SELECT N'Columnstore Indices and Batch Processing',
       N'docx',
       N'You should use a columnstore index on your fact tables,
       putting all columns of a fact table in a columnstore index.
       In addition to fact tables, very large dimensions could benefit
       from columnstore indices as well.
       Do not use columnstore indices for small dimensions. ',
       bulkcolumn
FROM OPENROWSET(BULK 'C:\TK70461\ColumnstoreIndicesAndBatchProcessing.docx',
                SINGLE_BLOB) AS doc;
INSERT INTO dbo.Documents (title, doctype, docexcerpt, doccontent)
SELECT N'Introduction to Data Mining',
       N'docx',
       N'Using Data Mining is becoming more a necessity for every company
       and not an advantage of some rare companies anymore. ',
       bulkcolumn
FROM OPENROWSET(BULK 'C:\TK70461\IntroductionToDataMining.docx',
                SINGLE_BLOB) AS doc;
```

```

INSERT INTO dbo.Documents (title, doctype, docexcerpt, doccontent)
SELECT N'Why Is Bleeding Edge a Different Conference',
N'docx',
N'During high level presentations attendees encounter many
questions. For the third year, we are continuing with
the breakfast Q&A session. It is very popular, and for
two years now, we could not accommodate enough time for
all questions and discussions! ',
bulkcolumn
FROM OPENROWSET(BULK
'C:\TK70461\WhyIsBleedingEdgeADifferentConference.docx',
SINGLE_BLOB) AS doc;
INSERT INTO dbo.Documents (title, doctype, docexcerpt, doccontent)
SELECT N'Additivity of Measures',
N'docx',
N'Additivity of measures is not exactly a data warehouse
design problem. However, you have to realize which
aggregate functions you will use in reports for which
measure, and which aggregate functions you will use
when aggregating over which dimension.',
bulkcolumn
FROM OPENROWSET(BULK 'C:\TK70461\AdditivityOfMeasures.docx',
SINGLE_BLOB) AS doc;

```

8. Создайте список свойств поиска с именем WordSearchPropertyList. Добавьте в список свойство Authors. Свойства документов имеют предопределенные уникальные идентификаторы GUID и целочисленные идентификаторы (Integer ID). Список некоторых, наиболее известных из них можно найти в электронной документации по SQL Server 2012 в разделе "Поиск идентификаторов GUID наборов свойств и целочисленных идентификаторов свойств для свойств поиска" по адресу <http://msdn.microsoft.com/ru-ru/library/ee677618.aspx>. GUID свойства Authors документов Office равен F29F85E0-4FF9-1068-AB91-08002B27B3D9, а его целочисленный индекс (integer ID) равен 4. Используйте следующий код:

```

CREATE SEARCH PROPERTY LIST WordSearchPropertyList;
GO
ALTER SEARCH PROPERTY LIST WordSearchPropertyList
ADD 'Authors'
WITH (PROPERTY_SET_GUID = 'F29F85E0-4FF9-1068-AB91-08002B27B3D9',
PROPERTY_INT_ID = 4,
PROPERTY_DESCRIPTION = 'System.Authors - authors of a given item.');

```

9. Создайте список стоп-слов с именем SQLStopList. Добавьте в него слово *SQL*, указав английский язык. Используйте следующий код:

```

CREATE FULLTEXT STOPLIST SQLStopList;
GO
ALTER FULLTEXT STOPLIST SQLStopList
ADD 'SQL' LANGUAGE 'English';

```

10. Проверьте список стоп-слов и запомните ID стоп-списка. Используйте следующий запрос:

```
SELECT w.stoplist_id, l.name, w.stopword, w.language
FROM sys.fulltext_stopwords AS w
INNER JOIN sys.fulltext_stoplists AS l
ON w.stoplist_id = l.stoplist_id;
```

11. Используйте динамическое административное представление `sys.dm_fts_parser` для того, чтобы проверить, как полнотекстовый поиск выполняет синтаксический анализ (parsing) строк, в соответствии с вашим стоп-списком, информацией тезауруса, разбиением по словам в выбранном языке и парадигматической моделью данного языка. Например, следующие два запроса проверяют, как строка разбита на слова и какие словоформы слова может использовать полнотекстовый поиск. Обратите внимание на параметры динамического административного представления: первый представляет анализируемую символьную строку, второй — идентификатор языка (1033 для US English), третий — идентификатор стоп-списка, который получили из предыдущего запроса, и наконец, четвертый — флагок, показывающий, должен ли синтаксический анализ учитывать диакритические знаки.

```
SELECT *
FROM sys.dm_fts_parser
(N'"Additivity of measures is not exactly a data warehouse design problem.
However, you have to realize which aggregate functions you will use
in reports for which measure, and which aggregate functions
you will use when aggregating over which dimension."', 1033, 5, 0);
SELECT *
FROM sys.dm_fts_parser
('FORMSOF(INFLECTIONAL,' + 'function' + ')', 1033, 5, 0);
```

Задание 2. Установка семантической базы данных и создание полнотекстового индекса

В этом задании вам предстоит установить семантическую базу данных и затем создать полнотекстовый индекс.

1. Проверьте, установлен ли компонент Semantic Language Statistics Database (база данных семантической статистики языка). Если следующий запрос не возвращает строку, этот компонент надо установить.

```
SELECT * FROM sys.fulltext_semantic_language_statistics_database;
```

Чтобы установить базу данных семантической статистики языка, запустите пакет `SemanticLanguageDatabase.msi` из папки `x64\Setup` (если вы используете 64-разрядный экземпляр) или `x86\Setup` (если у вас 32-битный экземпляр) с установочного устройства SQL Server.

2. Проверьте, имеет ли учетная запись службы SQL Server разрешения **Read** и **Write** на папку, в которой вы установили файлы компонента Semantic Language

Statistics Database. Папка по умолчанию — C:\Program Files\Microsoft Semantic Language Database. Если эта база данных установлена в папке по умолчанию, можно подключить ее с помощью следующей команды:

```
CREATE DATABASE semanticsdb ON
    (FILENAME = 'C:\Program Files\Microsoft Semantic Language
Database\semanticsdb.mdf'),
    (FILENAME = 'C:\Program Files\Microsoft Semantic Language
Database\semanticsdb_log.ldf')
FOR ATTACH;
```

3. После подключения базы данных зарегистрируйте ее, используя следующий код:

```
EXEC sp_fulltext_semantic_register_language_statistics_db
@dbname = N'semanticsdb';
```

4. Убедитесь в том, что база данных семантической статистики была успешно установлена, повторив запрос из п. 1. На этот раз запрос должен возвратить строку.
5. Наконец, пришло время создать каталог. Назовите его DocumentsFtCatalog. Используйте следующий код:

```
CREATE FULLTEXT CATALOG DocumentsFtCatalog;
```

6. Теперь создайте полнотекстовый индекс. Вы должны проиндексировать столбцы docexcerpt и doccontent. Установите свойство отслеживания изменений для заполнения индекса в значение AUTO. Используйте следующий код:

```
CREATE FULLTEXT INDEX ON dbo.Documents
( docexcerpt Language 1033,
  doccontent TYPE COLUMN doctype
  Language 1033
  STATISTICAL_SEMANTICS )
KEY INDEX PK_Documents
ON DocumentsFtCatalog
WITH STOPLIST = SQLStopList,
  SEARCH PROPERTY LIST = WordSearchPropertyList,
  CHANGE_TRACKING AUTO;
```

Резюме занятия

- ❑ Можно создавать полнотекстовые каталоги с помощью механизмов полнотекстового поиска и семантического поиска SQL Server.
- ❑ Можно улучшить полнотекстовый поиск, добавив стоп-слова в стоп-списки, расширив тезаурус разрешив поиск по свойствам документа.
- ❑ Можно использовать объект динамического управления sys.dm_fts_parser для того, чтобы проверить, как полнотекстовый поиск разбивает документ на слова, создает словоформы слов и т. д.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в приложении "Ответы" в конце книги.

1. Какие элементы полнотекстового поиска можно использовать для предотвращения индексирования лишних слов (noisy words)? (Выберите все подходящие варианты.)
 - A. Стоп-слова.
 - B. Тезаурус.
 - C. Парадигматический модуль.
 - D. Стоп-список.
2. Какую базу вы должны установить для того, чтобы разрешить семантический поиск?
 - A. msdb.
 - B. distribution.
 - C. semanticsdb.
 - D. tempdb.
3. Как можно создать синонимы для искомых слов?
 - A. Редактировать файл тезауруса.
 - B. Создать таблицу тезауруса.
 - C. Использовать стоп-слова и для синонимов.
 - D. Полнотекстовый поиск не поддерживает синонимы.

Занятие 2. Использование предикатов *CONTAINS* и *FREETEXT*

SQL Server поддерживает два очень мощных предиката для ограничения результирующего набора запроса с использованием полнотекстовых индексов. Эти два предиката — *CONTAINS* и *FREETEXT*. Оба они поддерживают разные виды поиска терминов. Кроме этих предикатов, SQL Server поддерживает две табличные функции для полнотекстового поиска и три табличные функции для семантического поиска. В этом занятии мы рассмотрим два указанных предиката, а в следующем — пять табличных функций.

Изучив материал этого занятия, вы сможете:

- ✓ Использовать в запросах предикат *CONTAINS*
- ✓ Использовать предикат *FREETEXT*

Продолжительность занятия — 40 минут.

Предикат **CONTAINS**

Предикат **CONTAINS** предоставляет возможность выполнять поиск:

- слов и фраз в тексте;
- точных или примерных (fuzzy) совпадений;
- словоформ слова;
- текста, в котором искомое слово находится рядом с другим искомым словом;
- синонимов искомого слова;
- префикса слова или выражения.

Также можно добавить собственный пользовательский вес к искомым словам. Предикат **CONTAINS** используется в предложении **WHERE** инструкции T-SQL.

Все подробности об этом предикате можно найти в электронной документации по SQL Server 2012 в разделе "CONTAINS (Transact-SQL)" по адресу <http://msdn.microsoft.com/ru-ru/library/ms187787.aspx>. Далее приведены наиболее важные формы запросов с предикатом **CONTAINS** в псевдокоде, где **FTcolumn** представляет столбец с полнотекстовой индексацией, а '**SearchWord?**' обозначает искомое слово или фразу:

- `SELECT...FROM...WHERE CONTAINS(FTcolumn, 'SearchWord1')`

Это простейшая форма. Она выполняет поиск строк, в которых **FTcolumn** содержит точное совпадение с '**SearchWord1**'. Это простое выражение.

- `SELECT...FROM...WHERE CONTAINS(FTcolumn, 'SearchWord1 OR SearchWord2')`

Выполняется поиск строк, в которых **FTcolumn** содержит полное совпадение со словом '**SearchWord1**' или со словом '**SearchWord2**'. Также можно использовать логические операторы **AND** и **NOT** и изменять порядок оценивания операторов в выражении с помощью круглых скобок.

- `SELECT...FROM...WHERE CONTAINS(FTcolumn, '"SearchWord1 SearchWord2")')`

Выполняется поиск строк, в которых **FTcolumn** содержит полное совпадение с фразой "**SearchWord1 SearchWord2**".

- `SELECT...FROM...WHERE CONTAINS(FTcolumn, '"SearchWord1*")')`

Выполняется поиск строк, в которых **FTcolumn** содержит хотя бы одно слово, начинающееся с '**SearchWord1**'. Это префиксное выражение.

- `SELECT...FROM...WHERE CONTAINS(FTcolumn, 'NEAR(SearchWord1, SearchWord2)')`

Выполняется поиск строк, в которых **FTcolumn** содержит **SearchWord1** и **SearchWord2**. Это простейшее настраиваемое выражение с учетом расположения. В этой простейшей версии выполняется поиск только вхождений обоих слов, без учета расстояния и между ними и порядка следования. Результат подобен поиску простого выражения, где два слова или фразы объединены с помощью логического оператора **AND**.

- `SELECT...FROM...WHERE CONTAINS(FTcolumn, 'NEAR((SearchWord1, SearchWord2), DISTANCE)')`

Здесь выполняется поиск строк, в которых `FTcolumn` содержит `SearchWord1` и `SearchWord2`. Порядок искомых слов не имеет значения, тогда как расстояние представляет собой целое число, указывающее, какое максимальное количество слов, не включающихся в поиск, может быть между искомыми словами, чтобы определить строку для результирующего набора.

- `SELECT...FROM...WHERE CONTAINS(FTcolumn, 'NEAR((SearchWord1, SearchWord2), DISTANCE, FLAG)')`

Выполняется поиск строк, в которых `FTcolumn` содержит `SearchWord1` и `SearchWord2`. Два выражения поиска должны быть ближе друг другу, чем указанное расстояние. Флаг может принимать значение `TRUE` или `FALSE`; по умолчанию установлено значение `FALSE`. Если флаг установлен в `TRUE`, тогда порядок искомых выражений имеет значение; слово `SearchWord1` должно стоять в тексте перед словом `SearchWord2`.

- `SELECT...FROM...WHERE CONTAINS(FTcolumn, 'FORMSOF(INFLECTIONAL, SearchWord1)')`

Это формат предиката с использованием *сформированного* слова. Выполняется поиск строк, в которых `FTcolumn` содержит любую словоформу слова `SearchWord1`.

- `SELECT...FROM...WHERE CONTAINS(FTcolumn, 'FORMSOF(THESAURUS, SearchWord1)')`

Это опять формат предиката с использованием *сформированного* слова. Выполняется поиск строк, в которых `FTcolumn` содержит или слово `SearchWord1`, или любой его синоним, определенный в файле тезауруса.

- `SELECT...FROM...WHERE CONTAINS(FTcolumn, 'ISABOUT(SearchWord1 WEIGHT(W1), SearchWord2 WEIGHT(W2))')`

Это *взвешенное* слово. Веса влияют на ранг возвращенных документов. Но поскольку предикат `CONTAINS` не выполняет ранжирование результатов, эта форма не влияет на результат. Взвешенная форма полезна в функции `CONTAINSTABLE`.

- `SELECT...FROM...WHERE CONTAINS(PROPERTY(FTcolumn, 'PROPERTYNAME'), 'SearchWord1')`

Это поиск по свойствам документа. Вам нужно иметь документы с известными свойствами. В таком запросе выполняется поиск строк в документах, имеющих свойство `PropertyName`, содержащее значение `SearchWord1`.

Предикат *FREETEXT*

Предикат `FREETEXT` является менее определенным и поэтому возвращает большее число строк, чем предикат `CONTAINS`. Он выполняет поиск значений, совпадающих со смыслом фразы, а не с точным значением слов. Когда вы используете предикат `FREETEXT`, ядро выполняет разбиение на слова искомой фразы, генерирует словоформы (но не парадигмы) и определяет список расширений и замен для слов в выражениях поиска на основании слов из тезауруса. Формат этого предиката значи-

тельно проще, чем предиката `CONTAINS`: `SELECT...FROM...WHERE FREETEXT(FTcolumn, 'SearchWord1 SearchWord2')`. Здесь вы выполняете поиск строк, в которых `FTcolumn` включает любые словоформы и любые определенные синонимы слов `SearchWord1` и `SearchWord2`.

COBET**Подготовка к экзамену**

Предикат `FREETEXT` является менее избирательным, чем предикат `CONTAINS`, и поэтому он обычно возвращает больше строк, чем предикат `CONTAINS`.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как выполнить поиск синонимов слова с помощью предиката `CONTAINS`?
2. Какой предикат является более определенным, `CONTAINS` или `FREETEXT`?

Ответы на контрольные вопросы

1. Нужно использовать синтаксис
`CONTAINS(FTcolumn, 'FORMSOF(THESAURUS, SearchWord1)')`.
2. Предикат `CONTAINS` применяется для более определенного поиска.

ПРАКТИКУМ Использование предикатов `CONTAINS` и `FREETEXT`

После создания всех компонентов, необходимых для решений полнотекстового поиска, самое время начать использовать полнотекстовый поиск.

Задание 1. Использование предиката `CONTAINS`

В этом задании вы будете использовать предикат `CONTAINS`. Кроме этого вам нужно будет отредактировать и использовать файл тезауруса.

1. Если вы закрыли SQL Server Management Studio (SSMS), запустите ее и подключитесь к вашему экземпляру SQL Server. Откройте новое окно запроса, нажав кнопку **New Query** (Новый запрос).
2. Подключитесь к базе данных TSQL2012.

3. Найдите все строки, в которых столбец `docexcerpt` таблицы `dbo.Documents` содержит слово `data`. Используйте следующий запрос:

```
SELECT id, title, docexcerpt
FROM dbo.Documents
WHERE CONTAINS(docexcerpt, N'data');
```

4. Найдите все строки, в которых столбец `docexcerpt` таблицы `dbo.Documents` содержит слово `data` или слово `index`. Используйте следующий запрос:

```
SELECT id, title, docexcerpt
FROM dbo.Documents
WHERE CONTAINS(docexcerpt, N'data OR index');
```

5. Найдите все строки, в которых столбец docexcerpt таблицы dbo.Documents содержит слово *data*, но не содержит слово *mining*. Используйте следующий запрос:

```
SELECT id, title, docexcerpt  
FROM dbo.Documents  
WHERE CONTAINS(docexcerpt, N'data AND NOT mining');
```

6. Найдите все строки, в которых столбец docexcerpt таблицы dbo.Documents содержит слово *data* или слова *fact* и *warehouse*. Используйте следующий запрос:

```
SELECT id, title, docexcerpt  
FROM dbo.Documents  
WHERE CONTAINS(docexcerpt, N'data OR (fact AND warehouse)');
```

7. Найдите все строки, в которых столбец docexcerpt таблицы dbo.Documents содержит фразу *data warehouse*. Используйте следующий запрос:

```
SELECT id, title, docexcerpt  
FROM dbo.Documents  
WHERE CONTAINS(docexcerpt, N'"data warehouse"');
```

8. Найдите все строки, в которых столбец docexcerpt таблицы dbo.Documents содержит слова, начинающиеся с префикса *add*. Используйте следующий запрос:

```
SELECT id, title, docexcerpt  
FROM dbo.Documents  
WHERE CONTAINS(docexcerpt, N'"add*"'');
```

9. Найдите все строки, в которых столбец docexcerpt таблицы dbo.Documents содержит слово *problem* где угодно рядом со словом *data*. Используйте следующий запрос:

```
SELECT id, title, docexcerpt  
FROM dbo.Documents  
WHERE CONTAINS(docexcerpt, N'NEAR(problem, data)'');
```

10. Найдите все строки, в которых столбец docexcerpt таблицы dbo.Documents содержит слово *problem* где угодно рядом со словом *data*. Попробуйте сделать это с помощью запроса, выполняющего поиск фрагментов, в которых эти слова расположены на расстоянии сначала менее пяти слов, а затем менее одного слова, не участвующего в поиске. Первый из двух приведенных далее запросов должен возвратить одну строку, второй — ни одной.

```
SELECT id, title, docexcerpt  
FROM dbo.Documents  
WHERE CONTAINS(docexcerpt, N'NEAR((problem, data),5)'');
```

```
SELECT id, title, docexcerpt  
FROM dbo.Documents  
WHERE CONTAINS(docexcerpt, N'NEAR((problem, data),1)'');
```

11. Найдите все строки, в которых столбец docexcerpt таблицы dbo.Documents содержит слово *problem* где угодно рядом со словом *data*. Попробуйте сделать это

с помощью запроса, выполняющего поиск фрагментов, в которых эти слова расположены на расстоянии менее пяти слов, не участвующих в поиске. Но теперь укажите, что слово *problem* должно стоять перед словом *data*. Используйте следующий запрос:

```
SELECT id, title, docexcerpt  
FROM dbo.Documents  
WHERE CONTAINS(docexcerpt, N'NEAR((problem, data), 5, TRUE)');
```

12. Найдите все строки, в которых столбец *docexcerpt* таблицы *dbo.Documents* содержит слово *presentation*. Попробуйте написать запрос, который ищет полное совпадение, и запрос, который ищет любые словоформы этого слова. Первый из приведенных далее запросов не должен возвратить ни одной строки, второй — одну строку.

```
SELECT id, title, docexcerpt  
FROM dbo.Documents  
WHERE CONTAINS(docexcerpt, N'presentation');
```

```
SELECT id, title, docexcerpt  
FROM dbo.Documents  
WHERE CONTAINS(docexcerpt, N'FORMSOF(INFLECTIONAL, presentation)');
```

Задание 2. Использование синонимов и предиката *FREETEXT*

В этом задании вам нужно отредактировать и использовать файл тезауруса для добавления синонима. Напишите запрос *PIVOT*, который возвращает одну строку для каждого отличного ID клиента, столбец для каждого отличного ID грузоотправителя и подсчет заказов на пересечении "клиент — грузоотправитель".

1. Используйте редактор Notepad (Блокнот) для модификации файла тезауруса для английского языка US English. Добавьте синоним *necessity* для слова *need*. Вам нужно отредактировать файл *tsenu.xml*, расположенный при установке по умолчанию в папке C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSSQL\FTData. Если при установке был задан иной путь или вы используете экземпляр сервера не по умолчанию, тогда в общем виде путь должен быть таким: *SQL_Server_install_path\Microsoft SQL Server\MSSQL11.Instance_id\MSSQL\FTData*. Очистите XML-комментарии в этом файле.

После редактирования содержимое файла должно выглядеть следующим образом:

```
<XML ID="Microsoft Search Thesaurus">  
<thesaurus xmlns="x-schema:tsSchema.xml">  
  <diacritics_sensitive>0</diacritics_sensitive>  
  <expansion>  
    <sub>Internet Explorer</sub>  
    <sub>IE</sub>  
    <sub>IE5</sub>  
  </expansion>
```

```

<replacement>
    <pat>NT5</pat>
    <pat>W2K</pat>
    <sub>Windows 2000</sub>
</replacement>
<expansion>
    <sub>run</sub>
    <sub>jog</sub>
</expansion>
<expansion>
    <sub>need</sub>
    <sub>necessity</sub>
</expansion>
</thesaurus> </XML>

```

2. Загрузите файл тезауруса для языка US English.

```
EXEC sys.sp_fulltext_load_thesaurus_file 1033;
```

3. Найдите все строки, в которых столбец docexcerpt таблицы dbo.Documents содержит слово *need* или его синоним. Попытайтесь сделать это с помощью запроса, который ищет точные совпадения, и с помощью запроса, который ищет синонимы слова. Первый из двух приведенных далее запросов должен не возвратить ни одной строки, второй — одну.

```

SELECT id, title, docexcerpt
FROM dbo.Documents
WHERE CONTAINS(docexcerpt, N'need');

```

```

SELECT id, title, docexcerpt
FROM dbo.Documents
WHERE CONTAINS(docexcerpt, N'FORMSOF(THESAURUS, need)');

```

4. Найдите все строки, в которых столбец doccontent таблицы dbo.Documents содержит свойство Authors, значение которого включает слово "Dejan". Используйте следующий запрос:

```

WITH PivotData
SELECT id, title, docexcerpt
FROM dbo.Documents
WHERE CONTAINS(PROPERTY(doccontent, 'Authors'), 'Dejan');

```

5. Наконец, найдите все строки, в которых столбец docexcerpt таблицы dbo.Documents содержит любое из слов *data*, *presentation* или *need*. Слова могут быть в любой словоформе. Также найдите синонимы этих слов. Используйте следующий запрос:

```

SELECT id, title, doctype, docexcerpt
FROM dbo.Documents
WHERE FREETEXT(docexcerpt, N'data presentation need');

```

Резюме занятия

- Предикат CONTAINS можно использовать для избирательного поиска.
- Предикат FREETEXT можно использовать для более общего поиска.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Что из нижеперечисленного не является частью предиката CONTAINS?
 - A. FORMSOF.
 - B. THESAURUS.
 - C. NEAR.
 - D. PROPERTY.
 - E. TEMPORARY.
2. Какое из перечисленных выражений с учетом расположения задает расстояние и порядок следования слов?
 - A. NEAR((SearchWord1, SearchWord2), 5, TRUE).
 - B. NEAR((SearchWord1, SearchWord2), CLOSE, ORDER).
 - C. NEAR((SearchWord1, SearchWord2), 5).
 - D. NEAR(SearchWord1, SearchWord2).
3. Что можно искать с помощью предиката CONTAINS? (Выберите все подходящие варианты.)
 - A. Словоформы слова.
 - B. Синонимы искомого слова.
 - C. Переводы слова.
 - D. Текст, в котором слово поиска находится рядом с другим словом.
 - E. Префикс слова или фразы.

Занятие 3. Использование табличных функций полнотекстового и семантического поиска

В предыдущем занятии вы узнали, что выражения в полнотекстовом поиске могут быть взвешены для изменения ранга документов. Но увидеть ранг документа с помощью предиката CONTAINS невозможно. Вы должны получить таблицу документов (или идентификаторов документов) и их ранг. Эта таблица возвращается функциями CONTAINSTABLE и FREETEXTTABLE. Кроме того, в практикуме к занятию 1 вы установили базу данных семантической статистики (Semantic Language Statistics

Database). Сейчас мы займемся выполнением семантического поиска посредством трех табличных функций: SEMANTICKEYPHRASETABLE, SEMANTICSIMILARITYDETAILSTABLE и SEMANTICSIMILARITYTABLE.

Изучив материал этого занятия, вы сможете:

- ✓ Использовать табличные функции полнотекстового поиска
- ✓ Использовать табличные функции семантического поиска

Продолжительность занятия — 30 минут.

Использование функций полнотекстового поиска

Функции CONTAINSTABLE и FREETEXTTABLE возвращают два столбца: KEY и RANK. Столбец KEY представляет собой уникальный ключ индекса, используемого в предложении KEY INDEX инструкции CREATE FULLTEXT INDEX. Столбец RANK возвращает порядковый номер между 0 и 1000. Это значение ранга. Оно говорит о том, насколько строка соответствует критерию поиска. Этот номер всегда релевантен запросу; он указывает относительный порядок значимости (релевантности) для определенного набора строк. Меньшее значение соответствует меньшей релевантности (степени соответствия). Действительные величины не имеют значения; они могут меняться даже при следующем запуске данного запроса.

Вычисление ранга является достаточно сложным. SQL Server принимает во внимание частотность (частоту встречаемости) термина, т. е. частоту нахождения искомого слова в документе, количество слов в документе, выражения с учетом расположения (в предложении NEAR), вес (предложение ISABOUT), количество индексированных строк и т. д. Вычисления отличаются для функции CONTAINSTABLE и для функции FREETEXTTABLE, поскольку последняя не поддерживает большинство параметров, поддерживаемых первой функцией, таких как учет расположения и вес терминов.

Сокращенно синтаксис функции CONTAINSTABLE выглядит следующим образом:

```
CONTAINSTABLE(table, { column_name | (column_list) | * },
  '<contains_search_codition>' [, LANGUAGE language_term]
  [, top_n_by_rank] )
```

Условия поиска те же самые, что и в предикате CONTAINS. Можно использовать простой термин, префикс, сформированное выражение, выражение с учетом расположения или взвешенный термин. Значение `top_n_by_rank` — это целое число, указывающее, что только *n* строк с наивысшим рангом должно быть возвращено в наборе строк. Этот параметр может иметь значение с точки зрения производительности, поскольку запрос может возвращать огромные наборы строк.

Синтаксис функции FREETEXTTABLE выглядит следующим образом:

```
FREETEXTTABLE(table, { column_name | (column_list) | * },
  'freetext_string' [, LANGUAGE language_term] [, top_n_by_rank] )
```

Здесь показан полный синтаксис, потому что он очень простой.

ПРИМЕЧАНИЕ Язык разбиения по словам, морфологического поиска, поиска по тезаурусу и стоп-слов

Стоит подробнее рассмотреть, что означает аргумент `язык`. Это язык, который SQL Server использует для разбиения по словам, морфологического поиска, поиска по тезаурусу и удаления стоп-слов из запроса. Если не указано никакое значение, используется язык полнотекстового поиска столбца. Этот параметр может быть полезен при сохранении документов, написанных на разных языках, в одном столбце. Локальный идентификатор документа (locale identifier, LCID) определяет, какой язык использует SQL Server для индексации содержимого. Когда запрашивается такой столбец, использование аргумента `язык` может повысить качество нахождения соответствий. Можно также использовать аргумент `язык` в предикатах `CONTAINS` и `FREETEXT` и в функции `CONTAINSTABLE`. Можно задать его как целое число, представляющее LCID, или как строковую переменную, представляющую псевдоним языка. Кроме того, можно даже указать LCID как шестнадцатеричную строку.

Использование функций семантического поиска

Существуют три табличные функции, разрешающие семантический поиск. Синтаксис первой из них, `SEMANTICKEYPHRASETABLE`, выглядит следующим образом:

`SEMANTICKEYPHRASETABLE
(таблица, { столбец | (список_столбцов) | * } [, ключ])`

Эта функция возвращает таблицу с ключевыми фразами, связанными со столбцами с полнотекстовой индексацией из списка `список_столбцов`. Параметр `ключ` указывает уникальный ключ из индекса, используемый в предложении `KEY INDEX` инструкции `CREATE FULLTEXT INDEX`. Если он опущен, SQL Server возвращает ключевые фразы для всех строк.

СОВЕТ Подготовка к экзамену

Семантический поиск доступен только с помощью табличных функций; он не поддерживает никакие предикаты предложения `WHERE` запроса.

Синтаксис второй функции семантического поиска, `SEMANTICSIMILARITYDETAILSTABLE`, выглядит следующим образом:

`SEMANTICSIMILARITYDETAILSTABLE
(таблица, столбец-источник, ключ-источник,
соответствующий_столбец, соответствующий_ключ)`

Эта функция возвращает таблицу с ключевыми фразами, являющимися общими для двух документов. Исходный документ задается с помощью аргумента `ключ-источник`, который также является уникальным ключом из индекса, используемого в предложении `KEY INDEX` инструкции `CREATE FULLTEXT INDEX`, и с помощью аргумента `столбец-источник`, который представляет имя столбца с полнотекстовой индексацией.

Последняя функция семантического поиска — `SEMANTICSIMILARITYTABLE` — представлена далее.

`SEMANTICSIMILARITYTABLE(таблица, { столбец | (список_столбцов) | * }, ключ)`

Эта функция возвращает таблицу с документами, имеющими семантическое подобие с искомым документом, указанным параметром `ключ`. Параметр `ключ` указывает

уникальный ключ из индекса, используемого в предложении KEY INDEX инструкции CREATE FULLTEXT INDEX. Этую функцию можно применять для определения, какие документы имеют наибольшее сходство с заданным документом.

КОНТРОЛЬНЫЙ ВОПРОС

- Сколько функций полнотекстового и семантического поиска поддерживает SQL Server?

Ответ на контрольный вопрос

- SQL Server поддерживает две функции полнотекстового и три функции семантического поиска.

ПРАКТИКУМ Использование функций полнотекстового и семантического поиска

В данном практикуме вам предстоит использовать табличные функции полнотекстового и семантического поиска.

Задание 1. Использование функций полнотекстового поиска

В этом задании вы будете выполнять запрос данных, используя функции CONTAINSTABLE и FREETEXTTABLE.

1. Если вы закрыли SQL Server Management Studio (SSMS), запустите ее и подключитесь к вашему экземпляру SQL Server. Откройте новое окно запроса, нажав кнопку **New Query** (Новый запрос).
2. Подключитесь к базе данных TSQL2012.
3. Напишите запрос, использующий функцию CONTAINSTABLE для ранжирования документов по нахождению в них (включению) слов *data* или *level* в столбце docexcerpt. Используйте следующий запрос:

```
SELECT D.id, D.title, CT.[RANK], D.docexcerpt
FROM CONTAINSTABLE(dbo.Documents, docexcerpt,
N'data OR level') AS CT
INNER JOIN dbo.Documents AS D
ON CT.[KEY] = D.id
ORDER BY CT.[RANK] DESC;
```

4. Напишите запрос, использующий функцию FREETEXTTABLE для ранжирования документов по нахождению в них (включению) слов *data* или *level* в столбце docexcerpt. Сравните результат с результатом предыдущего запроса. Используйте следующий запрос:

```
SELECT D.id, D.title, FT.[RANK], D.docexcerpt
FROM FREETEXTTABLE (dbo.Documents, docexcerpt,
N'data level') AS FT
```

```

    INNER JOIN dbo.Documents AS D
        ON FT.[KEY] = D.id
    ORDER BY FT.[RANK] DESC;

```

5. Напишите запрос, который извлекает ранг документов на основании нахождения в них слов *data* или *level* в столбце *docexcerpt*. Назначьте слову *data* вес 0,8, а слову *level* — вес 0,2. Сравните результат с результатами первого запроса с использованием функции *CONTAINSTABLE* в этом упражнении. Используйте следующий запрос:

```

SELECT D.id, D.title, CT.[RANK], D.docexcerpt
FROM CONTAINSTABLE
    (dbo.Documents, docexcerpt,
     N'ISABOUT(data weight(0.8), level weight(0.2))') AS CT
    INNER JOIN dbo.Documents AS D
        ON CT.[KEY] = D.id
    ORDER BY CT.[RANK] DESC;

```

6. Напишите запрос, который извлекает ранг документов на основании нахождения в них слов *data* или *row* в столбце *doccontent*. Слова должны находиться на расстоянии менее 30 терминов поиска. Используйте следующий запрос:

```

SELECT D.id, D.title, CT.[RANK]
FROM CONTAINSTABLE (dbo.Documents, doccontent,
                    N'NEAR((data, row), 30)') AS CT
    INNER JOIN dbo.Documents AS D
        ON CT.[KEY] = D.id
    ORDER BY CT.[RANK] DESC;

```

7. Протестируйте предыдущий запрос с разным расстоянием между терминами поиска.

Задание 2. Использование функций семантического поиска

В этом задании вы будете выполнять запрос данных, используя функции *SEMANTICKEYPHRASETABLE*, *SEMANTICSIMILARITYDETAILSTABLE* и *SEMANTICSIMILARITYTABLE*.

1. Напишите запрос, который извлекает 20 наиболее важных фраз семантического поиска в документах в столбце *doccontent*. Используйте следующий запрос:

```

SELECT TOP (20) D.id, D.title, SKT.keyphrase, SKT.score
FROM SEMANTICKEYPHRASETABLE
    (dbo.Documents, doccontent) AS SKT
    INNER JOIN dbo.Documents AS D
        ON SKT.document_key = D.id
    ORDER BY SKT.score DESC;

```

2. Возвратите все документы, кроме документа с ID, равным 1, упорядоченные по семантическому сходству с документом в столбце *doccontent* с ID, равным 1. Используйте следующий запрос:

```

SELECT SST.matched_document_key, D.title, SST.score
FROM SEMANTICSIMILARITYTABLE

```

```
(dbo.Documents, doccontent, 1) AS SST  
INNER JOIN dbo.Documents AS D  
    ON SST.matched_document_key = D.id  
ORDER BY SST.score DESC;
```

3. Возвратите ключевые фразы семантического поиска, общие для документа с ID, равным 1, и документа с ID, равным 4. Отсортируйте фразы по порогу подобия (similarity score). Используйте следующий запрос:

```
SELECT SSDT.keyphrase, SSDT.score  
FROM SEMANTICSIMILARITYDETAILSTABLE  
    (dbo.Documents, doccontent, 1, doccontent, 4) AS SSDT  
ORDER BY SSDT.score DESC;
```

4. Очистите базу данных.

```
DROP TABLE dbo.Documents;  
DROP FULLTEXT CATALOG DocumentsFtCatalog;  
DROP SEARCH PROPERTY LIST WordSearchPropertyList;  
DROP FULLTEXT STOPLIST SQLStopList;
```

5. Выйдите из SSMS.

Резюме занятия

- Функции полнотекстового поиска полезны для ранжирования результатов.
- Функции семантического подобия дают возможность глубоко проникать в содержимое документов. Они позволяют находить ключевые фразы и сравнивать документы.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какую функцию следует использовать для ранжирования документов на основании учета расположения слов относительно друг друга?
 - A. CONTAINSTABLE.
 - B. FREETEXTTABLE.
 - C. SEMANTICKEYPHRASETABLE.
 - D. SEMANTICSIMILARITYTABLE.
 - E. SEMANTICSIMILARITYDETAILSTABLE.
2. Какую функцию можно использовать для нахождения документа, который семантически наиболее подобен указанному документу?
 - A. CONTAINSTABLE.
 - B. FREETEXTTABLE.

- C. SEMANTICKEYPHRASETABLE.
 - D. SEMANTICSIMILARITYTABLE.
 - E. SEMANTICSIMILARITYDETAILSTABLE.
3. Какая функция возвращает таблицу с ключевыми фразами, связанными со столбцом с полнотекстовой индексацией?
- A. CONTAINSTABLE.
 - B. FREETEXTTABLE.
 - C. SEMANTICKEYPHRASETABLE.
 - D. SEMANTICSIMILARITYTABLE.
 - E. SEMANTICSIMILARITYDETAILSTABLE.

Упражнения

В следующих упражнениях вы примените полученные знания о запросе полнотекстовых данных и использовании семантического поиска. Ответы на вопросы можно найти в приложении "Ответы" в конце книги.

Упражнение 1. Расширение поиска

После развертывания бизнес-приложения (LOB) у клиента вы обнаружили, что оно недостаточно удобно для использования. Конечные пользователи должны выполнять множество операций поиска; при этом они всегда должны знать точную фразу, которую ищут.

1. Как можно улучшить взаимодействие пользователей с приложением?
2. Как нужно изменить запросы для поддержки улучшенного взаимодействия пользователей с приложением?

Упражнение 2. Использование семантического поиска

Вы должны проанализировать некоторые документы Microsoft Word с целью найти документы, семантически схожие с документом, который вы получили от руководителя. Вам необходимо представить быстрое и простое решение этой проблемы.

1. Следует ли вам для решения этой проблемы создать приложение Microsoft .NET или использовать запросы на языке T-SQL?
2. Если вы выберете T-SQL, какую функцию этого языка вы будете использовать?

Рекомендуемые упражнения

Для успешного освоения экзаменационных тем, представленных в данной главе, выполните следующие задания.

Знакомство с динамическими административными представлениями, связанными с полнотекстовым поиском, и создание и восстановление резервных копий полнотекстовых каталогов и индексов

С полнотекстовыми индексами связаны некоторые виды административной работы. Для краткого знакомства с такой административной работой вам следует познакомиться с информацией, представленной в динамических административных представлениях, которые связаны с полнотекстовым и семантическим поиском, и научиться создавать резервные копии полнотекстовых каталогов и индексов.

□ **Задание 1.** Чтобы хорошо разобраться в вопросах полнотекстового поиска, проверьте информацию, которую вы найдете в следующих динамических административных представлениях:

- sys.dm_fts_active_catalogs;
- sys.dm_fts_fdhosts;
- sys.dm_fts_index_keywords_by_document;
- sys.dm_fts_index_keywords_by_property;
- sys.dm_fts_index_keywords;
- sys.dm_fts_index_population;
- sys.dm_fts_memory_buffers;
- sys.dm_fts_memory_pools;
- sys.dm_fts_outstanding_batches;
- sys.dm_fts_parser;
- sys.dm_fts_population_ranges;
- sys.dm_fts_semantic_similarity_population.

□ **Задание 2.** Создание и восстановление резервных копий — это типичная работа администратора баз данных. Вы должны иметь общее представление о том, как включать полнотекстовые каталоги и индексы в резервную копию. Статья электронной документации по SQL Server 2012 "Создание резервных копий и восстановление полнотекстовых каталогов и индексов", расположенная по адресу <http://msdn.microsoft.com/ru-ru/library/ms142511.aspx>, объясняет, как выполнять следующие задачи:

- нахождение полнотекстовых индексов в полнотекстовом каталоге;
- нахождение файловой группы или файла, который содержит полнотекстовый индекс;
- создание резервной копии файловых групп, содержащих полнотекстовые индексы.

ГЛАВА 7

Запрос и управление XML-данными

Темы экзамена

- Работа с данными.
 - Запрос XML-данных и управление ими.

Microsoft SQL Server 2012 обеспечивает всестороннюю поддержку XML. Сюда входит создание XML из реляционных данных с помощью запроса и дробление XML-данных в реляционный табличный формат. Кроме того, в SQL Server есть собственный тип данных `xml`. Вы можете сохранять XML-данные, ограничивать их с помощью XML-схем, индексировать их посредством специализированных XML-индексов, манипулировать ими с помощью методов типа данных `xml`. Все методы XML-данных языка T-SQL принимают в качестве параметра строку XQuery. XQuery (сокращенное название XML Query Language) — это стандартный язык, используемый для запроса XML-данных и манипулирования ими.

В этой главе вы узнаете, как использовать все упомянутые выше возможности работы с XML. Кроме этого, вы поймете, почему стоит использовать XML в реляционной базе данных.

ВАЖНО!

Использование термина "XML" в данной главе

В этой главе термин "XML" используется для обозначения как открытого стандарта, так и типа данных языка T-SQL.

ПРЕЖДЕ ВСЕГО

Для полного освоения этой главы необходимо иметь:

- понимание принципов реляционных баз данных;
- опыт работы с SQL Server Management Studio (SSMS);
- некоторый опыт в написании кода на языке T-SQL;
- доступ к экземпляру SQL Server 2012 с установленной учебной базой данных TSQL2012.

Занятие 1. Возвращение результатов в виде XML с помощью предложения FOR XML

XML является широко используемым стандартом для обмена данными, вызова методов Web-сервисов, работы с файлами конфигурации и т. д. Мы начнем это занятие с краткого введения в XML. После этого вы узнаете, как с помощью запроса создать XML-файл, используя различные варианты предложения `FOR XML`. Завершается занятие информацией о том, как дробить XML-данные на реляционные таблицы с помощью функции набора строк `OPENXML`.

Изучив материал этого занятия, вы сможете:

- ✓ Описывать XML-документы
- ✓ Конвертировать реляционные данные в XML
- ✓ Дробить XML на таблицы

Продолжительность занятия — 40 минут.

Введение в XML

Это занятие знакомит вас с XML с помощью примеров. Далее приведен пример XML-документа, созданного с помощью предложения `FOR XML` инструкции `SELECT`.

```
<CustomersOrders>
  <Customer custid="1" companyname="Customer NRZBB">
    <Order orderid="10692" orderdate="2007-10-03T00:00:00" />
    <Order orderid="10702" orderdate="2007-10-13T00:00:00" />
    <Order orderid="10952" orderdate="2008-03-16T00:00:00" />
  </Customer>
  <Customer custid="2" companyname="Customer MLTDN">
    <Order orderid="10308" orderdate="2006-09-18T00:00:00" />
    <Order orderid="10926" orderdate="2008-03-04T00:00:00" />
  </Customer>
</CustomersOrders>
```

ПРИМЕЧАНИЕ Сопутствующий код

Запрос, в результате которого получен XML-код в предыдущем примере, а также другие запросы для других примеров, представлены в прилагаемых к курсу файлах кода.

Как вы видите, XML использует теги, чтобы поименовать части *XML-документа*. Эти части называются *элементами*. Каждый открывающий тег, такой как `<Customer>`, должен иметь соответствующий закрывающий тег, в данном случае это `</Customer>`. Если какой-либо элемент не имеет вложенных элементов, его нотация может быть сокращена до единственного тега, обозначающего начало и конец элемента, как например `<Order ... />`. Элементы могут быть вложенными. Теги не могут чередоваться; закрывающий тег родительского элемента должен располагаться после закрывающего тега последнего вложенного элемен-



та. Если каждый открывающий тег имеет соответствующий ему закрывающий тег и если теги правильно вложены друг в друга, считается, что это XML-документ *правильного формата*.

XML-документы являются *упорядоченными*. Это не означает, что они упорядочены по какому-то определенному значению элемента; это означает, что позиция элементов имеет значение. Например, элемент со значением `orderid`, равным 10702 в предыдущем примере, является вторым элементом `Order` под первым элементом `Customer`.

XML представляет собой *чувствительный к регистру текст в кодировке Unicode*. Никогда не следует забывать, что в XML учитывается регистр символов. Кроме того, некоторые символы в XML, такие как `<`, который представляет тег, обрабатываются как *элементы разметки* и имеют особое значение. Если эти символы нужно включить в значения XML-документа, они должны быть экранированы с помощью амперсандса (`&`) и специального кода, за которым следует точка с запятой (`;`), как показано в табл. 7.1.

Таблица 7.1. Специальные символы в XML-документах

Символ	Замещающий текст
<code>&</code> (амперсанд)	<code>&amp;</code>
<code>"</code> (кавычка)	<code>&quot;</code>
<code><</code> (меньше чем)	<code>&lt;</code>
<code>></code> (больше чем)	<code>&gt;</code>
<code>'</code> (апостроф)	<code>&apos;</code>

В качестве альтернативы можно использовать специальную XML-секцию `CDATA`, которая записывается как `<! [CDATA[...]]>`. Три точки здесь можно заменить любой символьной строкой, которая не содержит строковый литерал `]]>`; в этом случае специальные символы в строке не будут интерпретированы (*parsed*), как элементы разметки XML.

Инструкции по обработке (*processing instructions*), являющиеся информацией для приложений, обрабатывающих XML, записываются подобно элементам, между символами "меньше" (`<`) и "больше" (`>`), и начинаются и заканчиваются вопросительным знаком (?), как в случае `<?PItarget data?>`. Ядро, обрабатывающее XML — например, компонент SQL Server Database Engine — получает эти инструкции.

Кроме элементов и инструкций по обработке, XML может содержать компоненты в формате `<!-- Эта комментарий -->`.

Наконец, XML может включать в себя пролог в начале документа, указывающий версию XML и кодировку документа, как например `<?xml version="1.0" encoding="ISO-8859-15"?>`.

Кроме XML-документов можно также использовать XML-фрагменты. Единственная разница между документом и фрагментом состоит в том, что документ имеет один корневой узел, как например <CustomersOrders> в предыдущем примере. Если вы удалите этот узел, то получите следующий XML-фрагмент:

```
<Customer custid="1" companyname="Customer NRZBB">
  <Order orderid="10692" orderdate="2007-10-03T00:00:00" />
  <Order orderid="10702" orderdate="2007-10-13T00:00:00" />
  <Order orderid="10952" orderdate="2008-03-16T00:00:00" />
</Customer>
<Customer custid="2" companyname="Customer MLTDN">
  <Order orderid="10308" orderdate="2006-09-18T00:00:00" />
  <Order orderid="10926" orderdate="2008-03-04T00:00:00" />
</Customer>
```

Если вы удалите второго клиента, то снова получите XML-документ, поскольку он опять будет иметь один корневой узел.

Как видно из примеров, элементы могут иметь *атрибуты*. Атрибуты имеют собственные имена, и их значения заключены в кавычки. Это *атрибутивное* (attribute-centric) представление. Однако можно записать XML иначе; каждый атрибут может быть вложенным элементом исходного элемента. Это называется *элементным* (element-centric) представлением.

Наконец, имена элементов не должны быть уникальными, поскольку на них можно ссылаться по их позиции; но для того, чтобы различать элементы, принадлежащие к разным предметным областям, разным отделам или разным компаниям, можно добавить *пространства имен*. Можно объявить пространства имен, используемые в корневом элементе XML-документа. Для каждого отдельного пространства имен можно также использовать псевдоним. Затем псевдонимы пространств имен вы можете поставить перед именами элементов в качестве префиксов. Далее приведен пример элементного XML-документа, в котором используются пространства имен; данные в нем те же самые, что и в первом примере этого занятия.

```
<CustomersOrders xmlns:co="TK461-CustomersOrders"> <co:Customer>
  <co:custid>1</co:custid>
  <co:companyname>Customer NRZBB</co:companyname>
  <co:Order>
    <co:orderid>10692</co:orderid>
    <co:orderdate>2007-10-03T00:00:00</co:orderdate>
  </co:Order>
  <co:Order>
    <co:orderid>10702</co:orderid>
    <co:orderdate>2007-10-13T00:00:00</co:orderdate>
  </co:Order>
  <co:Order>
    <co:orderid>10952</co:orderid>
    <co:orderdate>2008-03-16T00:00:00</co:orderdate>
```

```
</co:Order>
</co:Customer>
<co:Customer>
  <co:custid>2</co:custid>
  <co:companyname>Customer MLTDN</co:companyname>
  <co:Order>
    <co:orderid>10308</co:orderid>
    <co:orderdate>2006-09-18T00:00:00</co:orderdate>
  </co:Order>
  <co:Order>
    <co:orderid>10926</co:orderid>
    <co:orderdate>2008-03-04T00:00:00</co:orderdate>
  </co:Order>
</co:Customer>
</CustomersOrders>
```

XML является очень гибким языком. Как вы можете заметить, существует всего несколько правил для создания XML-документа правильного формата. В XML-документе реальные данные перемешаны с *метаданными*, такими как имена элементов и атрибутов. Поскольку XML является текстом, он очень удобен для обмена данными между разными системами и даже между разными платформами. Однако при обмене данными важно, чтобы метаданные были фиксированными. Если бы вам нужно было импортировать документ с клиентскими заказами, как в предыдущих примерах, каждые 2 минуты, вы совершенно определенно захотели бы автоматизировать процесс импорта. Представьте себе, сколько вам пришлось бы выполнить работы, если бы метаданные изменились с каждым новым импортом. Например, пусть элемент *Customer* переименован в элемент *Client*, а элемент *Order* — в элемент *Purchase*. Или представьте, что атрибут *orderdate* (или элемент) вдруг изменяет свой тип данных из *timestamp* на *integer*. Вы быстро пришли бы к заключению, что хотели бы иметь более фиксированную *схему* для импортируемых XML-документов.

Существует множество разных стандартов для описания метаданных XML-документов. В настоящий момент наиболее широко используемым описанием метаданных является описание с помощью *XSD-документов* (XML Schema Description). XSD-документы представляют собой XML-документы, которые описывают метаданные других XML-документов. Схема XSD-документа предопределена. С помощью стандарта XSD можно указывать имена элементов, типы данных и количество вхождений элемента, ограничения и т. д. В следующем примере показана XSD-схема, описывающая элементную версию клиентов и их заказов.



```
<xsd:schema targetNamespace="TK461-CustomersOrders">
  xmlns:schema="TK461-CustomersOrders"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:sqltypes="http://schemas.microsoft.com/sqlserver/2004/sqltypes"
  elementFormDefault="qualified">
  <xsd:import namespace="http://schemas.microsoft.com/sqlserver/2004/sqltypes"/>
```

```
schemaLocation=http://schemas.microsoft.com/sqlserver/2004/sqltypes/sqltypes.xsd
/>
<xsd:element name="Customer">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="custid" type="sqltypes:int" />
      <xsd:element name="companyname">
        <xsd:simpleType>
          <xsd:restriction base="sqltypes:nvarchar"
            sqltypes:localeId="1033"
            sqltypes:sqlCompareOptions="IgnoreCase
              IgnoreKanaType IgnoreWidth"
            sqltypes:sqlSortId="52">
            <xsd:maxLength value="40" />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element ref="schema:Order" minOccurs="0"
        maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="Order">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="orderid" type="sqltypes:int" />
      <xsd:element name="orderdate" type="sqltypes:datetime" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

Когда выполняется проверка соответствия XML-документа схеме, это называется *проверкой правильности* (валидацией) документа. Говорится, что документ с предопределенной схемой — *типовизированный* (typed) XML-документ.

Получение XML из реляционных данных

Инструкция T-SQL `SELECT` позволяет создавать все XML-документы, показанные в этом занятии. В данном разделе описано, как преобразовать результирующий набор запроса в XML с помощью предложения `FOR XML` инструкции `SELECT` языка T-SQL. Вы узнаете о наиболее полезных параметрах и директивах этого предложения; подробное описание полного синтаксиса этого предложения можно найти в электронной документации для SQL Server 2012 в разделе "FOR XML (SQL Server)" по адресу <http://msdn.microsoft.com/ru-ru/library/ms178107.aspx>.

Режим FOR XML RAW

Первый параметр для создания XML из результата запроса — это режим `RAW`. Созданный XML довольно близок к реляционному (табличному) представлению данных. В режиме `RAW` каждая строка из возвращенного набора строк преобразуется в один элемент с именем `row`, а столбцы преобразуются в атрибуты этого элемента. Далее приведен пример XML-документа, созданного с помощью режима `FOR XML RAW`.

```
<row custid="1" companyname="Customer NRZBB" orderid="10692"
      orderdate="2007-10-03T00:00:00" />
<row custid="1" companyname="Customer NRZBB" orderid="10702"
      orderdate="2007-10-13T00:00:00" />
<row custid="1" companyname="Customer NRZBB" orderid="10952"
      orderdate="2008-03-16T00:00:00" />
<row custid="2" companyname="Customer MLTDN" orderid="10308"
      orderdate="2006-09-18T00:00:00" />
<row custid="2" companyname="Customer MLTDN" orderid="10926"
      orderdate="2008-03-04T00:00:00" />
```

Можно расширить режим `RAW`, переименовав элемент `row`, добавив корневой элемент, включив в XML пространства имен и сделав элементным XML. Далее приведен пример расширенного XML, созданного с использованием режима `FOR XML RAW`.

```
<CustomersOrders>
  <Order custid="1" companyname="Customer NRZBB" orderid="10692"
        orderdate="2007-10-03T00:00:00" />
  <Order custid="1" companyname="Customer NRZBB" orderid="10702"
        orderdate="2007-10-13T00:00:00" />
  <Order custid="1" companyname="Customer NRZBB" orderid="10952"
        orderdate="2008-03-16T00:00:00" />
  <Order custid="2" companyname="Customer MLTDN" orderid="10308"
        orderdate="2006-09-18T00:00:00" />
  <Order custid="2" companyname="Customer MLTDN" orderid="10926"
        orderdate="2008-03-04T00:00:00" />
</CustomersOrders>
```

Как видите, теперь это документ, а не фрагмент. Он более похож на "настоящий" XML; однако он не содержит никаких дополнительных уровней вложенности. Клиент с `custid` равным 1 повторяется три раза, по одному разу для каждого заказа; было бы лучше, если бы он появлялся только один раз и включал заказы как вложенные элементы. Получить XML, который проще читать, можно с помощью режима `FOR XML AUTO`, описанного в следующем разделе.

Режим FOR XML AUTO

Режим `FOR XML AUTO` прост в использовании и позволяет получить хорошие XML-документы с вложенными элементами. В режимах `AUTO` и `RAW` можно применять ключевое слово `ELEMENTS` для генерации элементного XML. Предложение `WITH NAMESPACES`, предшествующее части `SELECT` запроса, определяет пространства имен и

псевдонимы в возвращаемом XML. До сих пор вы видели только XML-результаты. В практикуме к этому занятию вы будете создавать запросы, которые дают подобный результат. Но для того, чтобы дать вам лучшее представление о том, как выглядит SELECT с предложением FOR XML, мы предлагаем следующий пример:

```
WITH XMLNAMESPACES('TK461-CustomersOrders' AS co) S
SELECT [co:Customer].custid AS [co:custid],
       [co:Customer].companyname AS [co:companyname],
       [co:Order].orderid AS [co:orderid],
       [co:Order].orderdate AS [co:orderdate]
  FROM Sales.Customers AS [co:Customer]
 INNER JOIN Sales.Orders AS [co:Order]
    ON [co:Customer].custid = [co:Order].custid
 WHERE [co:Customer].custid <= 2
   AND [co:Order].orderid % 2 = 0
 ORDER BY [co:Customer].custid, [co:Order].orderid
FOR XML AUTO, ELEMENTS, ROOT('CustomersOrders');
```

Псевдонимы таблиц и столбцов T-SQL в запросе используются для создания имен элементов с префиксами в виде имен пространств. Двоеточие применяется в XML для разделения имени пространства от имени элемента. Предложение WHERE запроса ограничивает выходные данные до двух клиентов, при этом извлекается только каждый второй заказ для каждого клиента. На выходе мы получаем довольно правильный элементный XML-документ.

```
<CustomersOrders xmlns:co="TK461-CustomersOrders">
  <co:Customer>
    <co:custid>1</co:custid>
    <co:companyname>Customer NRZBB</co:companyname>
    <co:Order>
      <co:orderid>10692</co:orderid>
      <co:orderdate>2007-10-03T00:00:00</co:orderdate>
    </co:Order>
    <co:Order>
      <co:orderid>10702</co:orderid>
      <co:orderdate>2007-10-13T00:00:00</co:orderdate>
    </co:Order>
    <co:Order>
      <co:orderid>10952</co:orderid>
      <co:orderdate>2008-03-16T00:00:00</co:orderdate>
    </co:Order>
  </co:Customer>
  <co:Customer> <co:custid>2</co:custid>
    <co:companyname>Customer MLTDN</co:companyname>
    <co:Order>
      <co:orderid>10308</co:orderid>
      <co:orderdate>2006-09-18T00:00:00</co:orderdate>
    </co:Order>
```

```
<co:Order>
  <co:orderid>10926</co:orderid>
  <co:orderdate>2008-03-04T00:00:00</co:orderdate>
</co:Order>
</co:Customer>
</CustomersOrders>
```

Обратите внимание, использование надлежащего предложения ORDER BY имеет очень важное значение. С помощью инструкции SELECT языка T-SQL фактически выполняется форматирование возвращаемого XML. Без предложения ORDER BY порядок возвращенных строк непредсказуем, поэтому можно получить запутанный XML-документ с многократно повторяющимся элементом, каждый раз содержащим только лишь часть вложенных элементов.

СОВЕТ

Подготовка к экзамену

В запросе предложение FOR XML стоит после предложения ORDER BY.

Не только предложение ORDER BY имеет важное значение; порядок столбцов в предложении SELECT также влияет на возвращаемый XML. SQL Server использует порядок столбцов для определения вложенности элементов. Порядок следования столбцов должен придерживаться взаимоотношения "один-ко-многим". Клиент может иметь множество заказов; поэтому столбцы customer должны в запросе стоять раньше столбцов order.

Вам может не понравиться, что вы должны заботиться о порядке следования столбцов; для отношения порядок столбцов и строк не имеет значения. Тем не менее следует признать, что результат запроса не является отношением; это текст в формате XML, и части вашего запроса используются для форматирования этого текста.

В режимах RAW и AUTO можно также возвращать XSD-схему создаваемого документа. Эта схема находится внутри возвращаемого XML, перед реальными XML-данными; поэтому она называется *встроенной схемой*. Для возвращения XSD-схемы используется директива XMLSCHEMA. Эта директива принимает параметр, который определяет целевое пространство имен. Если нужна только схема баз данных, просто следует включить в запрос условие WHERE с предикатом, который не удовлетворяет никакой строке. Следующий запрос возвращает схему XML, сгенерированную в предыдущем запросе.



```
SELECT [Customer].custid AS [custid],
       [Customer].companyname AS [companyname],
       [Order].orderid AS [orderid],
       [Order].orderdate AS [orderdate]
  FROM Sales.Customers AS [Customer]
 INNER JOIN Sales.Orders AS [Order]
    ON [Customer].custid = [Order].custid
 WHERE 1 = 2
   FOR XML AUTO, ELEMENTS,
   XMLSCHEMA('TK461-CustomersOrders');
```

Далее приведен пример — XSD-документ.

```
<xsd:schema targetNamespace="TK461-CustomersOrders"
    xmlns:schema="TK461-CustomersOrders"
    xmlns:xsd=http://www.w3.org/2001/XMLSchema
    xmlns:sqltypes=http://schemas.microsoft.com/sqlserver/2004/sqltypes
    elementFormDefault="qualified">
    <xsd:import namespace=http://schemas.microsoft.com/sqlserver/2004/sqltypes
    schemaLocation=http://schemas.microsoft.com/sqlserver/2004/sqltypes/sqltypes.xsd
/>
    <xsd:element name="Customer">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="custid" type="sqltypes:int" />
                <xsd:element name="companyname">
                    <xsd:simpleType>
                        <xsd:restriction base="sqltypes:nvarchar"
                            sqltypes:localeId="1033"
                            sqltypes:sqlCompareOptions="IgnoreCase IgnoreKanaType
                            IgnoreWidth"
                            sqltypes:sqlSortId="52"> <xsd:maxLength value="40" />
                        </xsd:restriction>
                    </xsd:simpleType>
                </xsd:element>
                <xsd:element ref="schema:Order" minOccurs="0"
                    maxOccurs="unbounded" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="Order">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="orderid" type="sqltypes:int" />
                <xsd:element name="orderdate" type="sqltypes:datetime" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Режим FOR XML PATH

С помощью двух последних возможностей предложения FOR XML — параметров EXPLICIT и PATH — можно вручную определять возвращаемый XML. Используя эти два параметра, вы получаете полный контроль над возвращаемым XML-документом. Режим EXPLICIT предназначен только для обеспечения обратной совместимости; он использует собственный синтаксис T-SQL для формирования

XML. Режим PATH использует стандартные выражения XML XPath для определения элементов и атрибутов создаваемого XML. В этом разделе рассматривается режим PATH; если необходимы дополнительные сведения о режиме EXPLICIT, обратитесь к электронной документации по SQL Server 2012 и статье "Использование режима EXPLICIT совместно с предложением FOR XML" по адресу <http://msdn.microsoft.com/ru-ru/library/ms189068.aspx>.

В режиме PATH имена и псевдонимы столбцов служат как выражения XPath. Выражения XPath определяют путь к элементу в генерируемом XML. Путь представлен в виде иерархической структуры; уровни разделяются с помощью слеша (/). По умолчанию каждый столбец становится элементом; если нужно сгенерировать атрибутивный XML, перед именем псевдонима должен быть префикс в виде символа "at" (@).

Вот как выглядит пример простого запроса XPath:

```
SELECT Customer.custid AS [@custid],  
       Customer.companyname AS [companyname]  
  FROM Sales.Customers AS Customer  
 WHERE Customer.custid <= 2  
 ORDER BY Customer.custid  
FOR XML PATH ('Customer'), ROOT('Customers');
```

Этот запрос возвращает следующий результат:

```
<Customers>  
  <Customer custid="1">  
    <companyname>Customer NRZBB</companyname>  
  </Customer>  
  <Customer custid="2">  
    <companyname>Customer MLTDN</companyname>  
  </Customer>  
</Customers>
```

Если необходимо создать XML с вложенными элементами для дочерних таблиц, в режиме PATH нужно использовать вложенные запросы (подзапросы) в части SELECT запроса. Вложенные запросы должны возвращать скалярное значение в предложении SELECT. Однако мы знаем, что родительская строка может иметь несколько дочерних строк; клиент может иметь несколько заказов. Скалярное значение получается посредством возвращения XML из вложенного запроса. Затем результат возвращается как единственное скалярное XML-значение. Вложенные XML из-под запроса форматируются с помощью предложения FOR XML, так же как XML форматируется во внешнем запросе. Кроме этого, нужно использовать директиву TYPE предложения FOR XML для получения значения типа данных XML, а не XML в виде текста, что не может быть получено из внешнего запроса.

Вы будете создавать XML с вложенными элементами, используя предложение FOR XML PATH, в практикуме к данному занятию.

КОНТРОЛЬНЫЙ ВОПРОС

- Как с помощью инструкции `SELECT` получить XSD-схему вместе с XML-документом?

Ответ на контрольный вопрос

- Для этого надо использовать директиву `XMLSHEMA` предложения `FOR XML`.



Дробление XML на таблицы

Вы только что узнали, как создавать XML из реляционных данных. Конечно, можно выполнять и обратную операцию: преобразовывать XML в таблицы. Преобразование XML в реляционные таблицы называется *дроблением* (shredding) XML. Достичь этого можно с помощью метода `nodes()` типа данных `xml`; этот метод описан в занятии 3. Начиная с SQL Server 2000, дробление можно выполнять с помощью функции набора данных `OPENXML`.

Функция `OPENXML` позволяет получить набор строк XML-документов с помощью *объектной модели документа* (Document Object Model, DOM) XML-документов. Прежде чем выполнять синтаксический разбор DOM, его нужно для этого подготовить. Чтобы подготовить представление DOM XML, следует вызвать системную хранимую процедуру `sys.sp_xml_preparedocument`. После дробления документа нужно удалить представление DOM с помощью системной процедуры `sys.sp_xml_removedocument`.

Функция `OPENXML` использует следующие параметры:

- дескриптор XML DOM-документа, возвращаемый процедурой `sp_xml_preparedocument`;
- выражение XPath для нахождения узлов, которые нужно сопоставить строкам возвращаемого набора данных;
- описание возвращаемого набора данных;
- сопоставление между XML-узлами и столбцами набора строк.

Дескриптор документа является целым числом. Это простейший параметр. Выражение XPath определяется как шаблон строки, который указывает, как XML-узлы преобразуются в строки. Путь к узлу используется как шаблон; узлы, расположенные ниже выбранного узла, определяют строки возвращаемого набора строк.

Можно сопоставить XML-элементы или атрибуты строкам и столбцам с помощью предложения `WITH` функции `OPENXML`. В этом предложении можно указать существующую таблицу, которая используется как шаблон для возвращаемого набора строк, или таблицу можно задать с помощью синтаксиса, подобного используемому в инструкции `CREATE TABLE` T-SQL.

Функция `OPENXML` принимает необязательный третий параметр `flags`, который позволяет задать сопоставление между XML-данными и реляционным набором данных. Значение параметра 1 обозначает атрибутивную модель сопоставления, 2 — сопоставление с использованием элементов, 3 — оба сопоставления. Но значение

параметра `flags`, равное 3, не задокументировано, поэтому не рекомендуется его использовать. Значение `flags`, равное 8, можно совмещать со значениями 1 и 2 с помощью побитового логического оператора `OR` для получения возможности применения одновременно атрибутивной модели сопоставления и сопоставления с использованием элементов. XML из следующих примеров OPENXML использует атрибуты и элементы, например `custid` является атрибутом, а `companynname` — элементом. Назначение этого несколько усложненного XML — показать разницу между атрибутивной моделью сопоставления и сопоставлением с использованием элементов. Приведенный далее код дробит один и тот же XML три раза, чтобы продемонстрировать разницу между различными видами сопоставления с использованием следующих значений параметра `flags`: 1, 2 и 11 (8 + 1 + 2); все три запроса используют одно и то же описание набора строк в предложении `WITH`.

```
DECLARE @DocHandle AS INT;
DECLARE @XmlDocument AS NVARCHAR(1000);
SET @XmlDocument = N'
<CustomersOrders>
    <Customer custid="1">
        <companynname>Customer NRZBB</companynname>
        <Order orderid="10692">
            <orderdate>2007-10-03T00:00:00</orderdate>
        </Order>
        <Order orderid="10702">
            <orderdate>2007-10-13T00:00:00</orderdate>
        </Order>
        <Order orderid="10952">
            <orderdate>2008-03-16T00:00:00</orderdate>
        </Order>
    </Customer>
    <Customer custid="2">
        <companynname>Customer MLTDN</companynname>
        <Order orderid="10308">
            <orderdate>2006-09-18T00:00:00</orderdate>
        </Order>
        <Order orderid="10926">
            <orderdate>2008-03-04T00:00:00</orderdate>
        </Order>
    </Customer> </CustomersOrders>';
-- Создание внутреннего представления
EXEC sys.sp_xml_preparedocument @DocHandle OUTPUT, @XmlDocument;
-- Атрибутивное сопоставление
SELECT *
FROM OPENXML (@DocHandle, '/CustomersOrders/Customer', 1)
    WITH (custid INT, companynname NVARCHAR(40));
-- Элементное сопоставление
SELECT *
```

```
FROM OPENXML (@DocHandle, '/CustomersOrders/Customer',2)
    WITH (custid INT, companyname NVARCHAR(40));
-- Атрибутивное и элементное сопоставления
-- Комбинирование флага 8 с флагами 1 и 2
SELECT *
FROM OPENXML (@DocHandle, '/CustomersOrders/Customer',11)
    WITH (custid INT, companyname NVARCHAR(40));
-- Удаление DOM
EXEC sys.sp_xml_removedocument @DocHandle;
GO
```

Далее приведен результат выполнения этих трех запросов.

custid	companyname
1	NULL
2	NULL
custid	companyname
NULL	Customer NRZBB
NULL	Customer MLTDN
custid	companyname
1	Customer NRZBB
2	Customer MLTDN

Как видите, мы получим атрибуты с атрибутивным сопоставлением, элементы с сопоставлением с использованием элементов и комбинацию этих двух типов сопоставления. Метод `nodes()` типа данных `xml` более эффективен для дробления XML-документа только один раз и поэтому является предпочтительным способом дробления XML-документов в таком случае. Однако если один и тот же документ надо дробить несколько раз, как показано в примере из трех запросов для функции `OPENXML`, тогда более быстрым вариантом может быть представление DOM один раз, использование функции `OPENXML` несколько раз и удаление представления DOM.

ПРАКТИКУМ Использование предложения FOR XML

В этом практикуме вы будете создавать XML-документы из реляционных данных. Вам нужно возвратить XML-данные в виде документа и в виде фрагмента.

Задание 1. Возвращение XML-документа

В этом задании вам нужно возвратить XML в виде документа из реляционных данных.

1. Запустите SQL Server Management Studio (SSMS) и подключитесь к вашему экземпляру SQL Server.
2. Откройте новое окно запроса, нажав кнопку **New Query** (Новый запрос).

3. Измените контекст на базу данных TSQL2012.
4. Возвратите клиентов с их заказами в виде XML в режиме RAW. Возвратите столбцы custid и companyname из таблицы Sales.Customers и столбцы orderid и orderdate из таблицы Sales.Orders. Можно использовать следующий запрос:

```
SELECT Customer.custid, Customer.companyname,
       [Order].orderid, [Order].orderdate
  FROM Sales.Customers AS Customer
    INNER JOIN Sales.Orders AS [Order]
      ON Customer.custid = [Order].custid
 ORDER BY Customer.custid, [Order].orderid
 FOR XML RAW;
```

5. Проверьте полученный результат:
6. Усовершенствуйте XML, созданный предыдущим запросом, изменив режим RAW на AUTO. Сделайте результат элементным, используя TK461-CustomersOrders в качестве пространства имен и CustomersOrders как корневой элемент. Можно использовать следующий код.

```
WITH XMLNAMESPACES('TK461-CustomersOrders' AS co)
SELECT [co:Customer].custid AS [co:custid],
       [co:Customer].companyname AS [co:companyname],
       [co:Order].orderid AS [co:orderid],
       [co:Order].orderdate AS [co:orderdate]
  FROM Sales.Customers AS [co:Customer]
    INNER JOIN Sales.Orders AS [co:Order]
      ON [co:Customer].custid = [co:Order].custid
 ORDER BY [co:Customer].custid, [co:Order].orderid
 FOR XML AUTO, ELEMENTS, ROOT('CustomersOrders');
```

Задание 2. Возвращение XML-фрагмента

В этом задании вам нужно возвратить XML, отформатированный как фрагмент, из реляционных данных.

1. Возвратите третий XML-код в виде фрагмента, а не документа. Возвратите верхний элемент Customer с атрибутами custid и companyname. Возвратите вложенный элемент Order с атрибутами orderid и orderdate. Используйте предложение FOR XML PATH для явного форматирования XML. Можно воспользоваться следующим кодом:

```
SELECT Customer.custid AS [@custid],
       Customer.companyname AS [@companyname],
       (SELECT [Order].orderid AS [@orderid],
              [Order].orderdate AS [@orderdate]
         FROM Sales.Orders AS [Order]
        WHERE Customer.custid = [Order].custid
          AND [Order].orderid % 2 = 0
        FOR XML PATH('Order')) AS [Order]
```

```
ORDER BY [Order].orderid
FOR XML PATH('Order'), TYPE)
FROM Sales.Customers AS Customer
WHERE Customer.custid <= 2
ORDER BY Customer.custid
FOR XML PATH('Customer');
```

2. Проверьте полученный результат.

Резюме занятия

- Предложение FOR XML инструкции SELECT языка T-SQL можно использовать для получения XML.
- Используйте функцию OPENXML для дробления XML на таблицы.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какие режимы FOR XML справедливы? (Выберите все подходящие варианты.)
 - FOR XML AUTO.
 - FOR XML MANUAL.
 - FOR XML DOCUMENT.
 - FOR XML PATH.
2. Какую директиву предложения FOR XML нужно использовать для получения элементного XML?
 - ATTRIBUTES.
 - ROOT.
 - ELEMENTS.
 - XMLSHEMA.
3. Какую опцию предложения FOR XML можно использовать для ручного форматирования возвращаемого XML? (Выберите все подходящие варианты.)
 - FOR XML AUTO.
 - FOR XML EXPLICIT.
 - FOR XML RAW.
 - FOR XML PATH.

Занятие 2. Запрос XML-данных с помощью XQuery

XQuery — это стандартный язык для просмотра экземпляров XML и возвращения XML. Он значительно богаче, чем выражение XPath, являющееся более старым стандартом, который можно использовать только для простой навигации. Используя язык XQuery, можно перемещаться так же, как и с помощью XPath; но кроме этого вы можете выполнять перебор узлов, формировать возвращаемый экземпляр XML и многое другое.



Для работы с языком запросов нужен механизм обработки запросов. Компонент SQL Server Database Engine обрабатывает XQuery внутри инструкций T-SQL с помощью методов типа данных XML. Не все свойства XQuery поддерживаются в SQL Server. Например, определяемые пользователем функции XQuery не поддерживаются в SQL Server, поскольку там уже имеется язык T-SQL с доступными функциями CLR. В дополнение, T-SQL поддерживает нестандартные расширения к языку XQuery, называемые XML DML, которые можно использовать для модификации элементов и атрибутов в XML-данных. Поскольку тип данных XML является большим объектом, возможно значительное ограничение производительности, если единственным способом модификации XML-величины будет замена всей этой величины.

В данном занятии рассматривается только использование XQuery для извлечения данных; дополнительные сведения о типах данных XML приведены в занятии 3. В этом занятии мы будем использовать только переменные типа данных XML и метод query типа данных XML. Метод query принимает строку XQuery как параметр и возвращает XML, который вы формулируете в XQuery.

Реализация XQuery в SQL Server выполнена в соответствии со стандартом консорциума WWW (World Wide Web Consortium, W3C) и снабжена расширениями для поддержки модификации данных. Дополнительные сведения о W3C можно найти в Интернете по адресу <http://www.w3.org/>, а новости и дополнительные ресурсы по языку XQuery находятся по адресу <http://www.w3.org/XML/Query/>.

Изучив материал этого занятия, вы сможете:

- ✓ Использовать выражение XPath для перемещения по узлам экземпляра XML
- ✓ Использовать предикаты XQuery
- ✓ Использовать выражение XQuery FLWOR

Продолжительность занятия — 60 минут.

Основные понятия XQuery

Язык XQuery, так же как и XML, чувствителен к регистру символов. Поэтому если вы будете править примеры вручную, то должны писать запросы точно так, как они написаны в этой главе. Например, если вы напишете `Data()` вместо `data()`, будет выдано сообщение об ошибке, говорящее о том, что функции `Data()` не существует.

XQuery возвращает последовательности, которые могут включать в себя *атомарные* значения или *комплексные* значения (XML-узлы). Любой узел, такой как элемент, атрибут, текст, обрабатывающая инструкция, комментарий или документ, может быть включен в последовательность. Разумеется, можно форматировать последовательности, чтобы получить XML-документ правильного формата. Следующий код демонстрирует различные последовательности, возвращенные из простого экземпляра XML с помощью трех XML-запросов.

```
DECLARE @x AS XML;
SET @x=N'
<root>
  <a>1<c>3</c><d>4</d></a>
  <b>2</b>
</root>';
SELECT
  @x.query('*') AS Complete_Sequence,
  @x.query('data(*)') AS Complete_Data,
  @x.query('data(root/a/c)') AS Element_c_Data;
```

Вот как выглядят возвращенные последовательности:

Complete_Sequence	Complete_Data	Element_c_Data
<root><a>1<c>3</c><d>4</d>2</root>	1342	3

Первое выражение XQuery — простейшее возможное выражение *path*, которое выбирает все из экземпляра XML; второе использует функцию *data()* для извлечения атомарных значений данных из всего документа; третье использует функцию *data()* для извлечения атомарных значений данных только из элемента.

Каждый идентификатор в XQuery — это полное имя (qualified name) или QName. QName состоит из локального имени и необязательного префикса пространства имен. В предыдущем примере *root*, *a*, *b*, *c* и *d* являются именами QName; но у них нет префиксов пространств имен. В SQL Server предопределены следующие стандартные пространства имен:

- *xs* — пространство имен для XML-схемы (универсальный идентификатор ресурса или URI, <http://www.w3.org/2001/XMLSchema>);
- *xsi* — пространство имен экземпляра XML-схемы, которое используется для связи XML-схемы с документами экземпляра (<http://www.w3.org/2001/XMLSchema-instance>);
- *xdt* — пространство имен для типов данных XPath и XQuery (<http://www.w3.org/2004/07/xpath-datatypes>);
- *fn* — пространство имен функций (<http://www.w3.org/2004/07/xpath-functions>);
- *sqltypes* — пространство имен, которое обеспечивает сопоставление типов данных SQL Server (<http://schemas.microsoft.com/sqlserver/2004/sqltypes>);
- *xml* — пространство имен XML по умолчанию ([http://www.w3.org/XML/1998\(namespace\)](http://www.w3.org/XML/1998(namespace))).

Эти пространства имен можно использовать в запросах без их дополнительного определения. Вы можете определить собственные типы данных в *прологе*, который относится к началу XQuery. Пролог отделяется от тела запроса точкой с запятой. Кроме того, в T-SQL можно объявлять пространства имен, используемые в выражениях XQuery заранее, в предложении WITH команды SELECT языка T-SQL. Если XML использует одно пространство имен, можно также объявить его как пространство имен по умолчанию для всех элементов в прологе XQuery.

Также можно включить в выражение XQuery комментарии — это текст между круглыми скобками и двоеточиями (: this is a comment :). Не следует путать их с узлами комментариев в XML-документе; это комментарий вашего XQuery, и он не влияет на возвращаемый XML. Следующий код показывает все три метода объявления пространств имен и использует комментарии XQuery. Он извлекает заказы для первого клиента из экземпляра XML.

```
DECLARE @x AS XML;
SET @x='<CustomersOrders xmlns:co="TK461-CustomersOrders">
<co:Customer co:custid="1" co:companyname="Customer NRZBB">
    <co:Order co:orderid="10692" co:orderdate="2007-10-03T00:00:00" />
    <co:Order co:orderid="10702" co:orderdate="2007-10-13T00:00:00" />
    <co:Order co:orderid="10952" co:orderdate="2008-03-16T00:00:00" />
</co:Customer>
<co:Customer co:custid="2" co:companyname="Customer MLTDN">
    <co:Order co:orderid="10308" co:orderdate="2006-09-18T00:00:00" />
    <co:Order co:orderid="10926" co:orderdate="2008-03-04T00:00:00" />
</co:Customer>
</CustomersOrders>';
-- Пространство имен во вводном элементе XQuery
SELECT @x.query('
(: explicit namespace :)
declare namespace co="TK461-CustomersOrders";
//co:Customer[1]/*') AS [Explicit namespace];
-- Пространство имен по умолчанию во вводном элементе XQuery
SELECT @x.query('
(: default namespace :)
declare default element namespace "TK461-CustomersOrders";
//Customer[1]/*') AS [Default element namespace];
-- Пространство имен, определенное в T-SQL-предложении SELECT WITH
WITH XMLNAMESPACES('TK461-CustomersOrders' AS co)
SELECT @x.query('
(: namespace declared in T-SQL :)
//co:Customer[1]/*') AS [Namespace in WITH clause];
```

Далее приведены выходные данные в сокращенном виде.

Explicit namespace

```
<co:Order xmlns:co="TK461-CustomersOrders" co:orderid="10692" co:orderd
```

```

Default element namespace
-----
<Order xmlns="TK461-CustomersOrders" xmlns:p1="TK461-Customers"
Namespace in WITH clause
-----
<co:Order xmlns:co="TK461-CustomersOrders" co:orderid="10692" co:orderd

```

ПРИМЕЧАНИЕ Пространство имен по умолчанию

Если используется пространство имен элемента по умолчанию, оно не включается для элементов в результирующий XML; оно включается для атрибутов. Поэтому только первый и третий запросы полностью эквивалентны. Кроме того, когда используется пространство имен элемента по умолчанию, нельзя определить аббревиатуру собственного пространства имен.

В этих запросах используется относительный путь для нахождения элемента `Customer`. Прежде чем рассматривать все различные способы навигации в XQuery, вам нужно прочесть о самых важных типах данных и функциях XQuery, описанных в двух последующих разделах.

Типы данных XQuery

XQuery использует около 50 предопределенных типов данных. Кроме того, в реализации SQL Server также имеется пространство имен `sqltypes`, которое определяет типы данных SQL Server. Вы уже знаете о типах данных SQL Server. Не стоит уделять слишком много внимания типам данных XQuery; вам никогда не придется использовать большинство из них. В данном разделе перечислены лишь наиболее важные типы данных без их детального рассмотрения.

Типы данных XQuery делятся на типы узлов и атомарные типы. Типы узлов включают в себя атрибут, комментарий, элемент, пространство имен, текст, инструкцию по обработке и узел документа. Наиболее важные атомарные типы, которые можно использовать в запросах, — это `xs:boolean`, `xs:string`, `xs:QName`, `xs:date`, `xs:time`, `xs:datetime`, `xs:float`, `xs:double`, `xs:decimal` и `xs:integer`.

Вам необходимо лишь бегло ознакомиться с этим значительно сокращенным списком. Важно понять, что в языке XQuery существует собственная система типов, он имеет все наиболее часто применяемые типы, которые вам могут понадобиться, и вы можете использовать в определенных функциях только определенные типы данных. Поэтому пришло время представить несколько важных функций языка XQuery.

Функции XQuery

Наряду с множеством типов данных, в языке XQuery также имеются десятки функций. Они объединены в несколько категорий. Функция `data()`, ранее использовавшаяся в этой главе, — это функция метода доступа к данным. К наиболее часто используемым и полезным функциям языка XQuery, поддерживаемым SQL Server, относятся следующие:

- ❑ числовые функции (numeric functions) — `ceiling()`, `floor()` и `round()`;
- ❑ строковые функции (string functions) — `concat()`, `contains()`, `substring()`, `string-length()`, `lower-case()` и `upper-case()`;

- функции над значениями типа Boolean и функции логического конструктора (boolean и boolean constructor functions) — not(), true() и false();
- функции на узлах (nodes functions) — local-name() и namespace-uri();
- статистические функции (aggregate functions) — count(), min(), max(), avg() и sum();
- функции метода доступа данных (data accessor functions) — data() и string();
- функции расширения XQuery (SQL server extension functions) — sql:column() и sql:variable().

Можно легко понять, что делает функция и какие типы данных она поддерживает, по именам функций и категорий. Полный список функций и их подробное описание можно найти в электронной документации по SQL Server 2012 в разделе "Функции XQuery для типа данных xml" по адресу <http://msdn.microsoft.com/ru-ru/library/ms189254.aspx>.

Следующий запрос использует статистические функции count() и max() для извлечения информации о заказах для каждого клиента в XML-документе.

```
DECLARE @x AS XML;
SET @x='
<CustomersOrders>
  <Customer custid="1" companyname="Customer NRZBB">
    <Order orderid="10692" orderdate="2007-10-03T00:00:00" />
    <Order orderid="10702" orderdate="2007-10-13T00:00:00" />
    <Order orderid="10952" orderdate="2008-03-16T00:00:00" />
  </Customer>
  <Customer custid="2" companyname="Customer MLTDN">
    <Order orderid="10308" orderdate="2006-09-18T00:00:00" />
    <Order orderid="10926" orderdate="2008-03-04T00:00:00" />
  </Customer>
</CustomersOrders>';
SELECT @x.query('
for $i in //Customer
return
  <OrdersInfo>
    { $i/@companyname }
    <NumberOfOrders>
      { count($i/Order) }
    </NumberOfOrders>
    <LastOrder>
      { max($i/Order/@orderid) }
    </LastOrder>
  </OrdersInfo>
');
```

Как видите, этот пример XQuery является более сложным, чем предыдущий. Этот запрос использует итерации, известные как выражения XQuery FLWOR, и форматиру-

ет возвращенный XML в части `return` запроса. Выражения `FLWOR` обсуждаются далее в этом разделе. Сейчас рассматривайте этот запрос как пример использования статистических функций в языке XQuery. Далее приведен результат этого запроса.

```
<OrdersInfo companyname="Customer NRZBB">
  <NumberOfOrders>3</NumberOfOrders>
  <LastOrder>10952</LastOrder>
</OrdersInfo>
<OrdersInfo companyname="Customer MLTDN">
  <NumberOfOrders>2</NumberOfOrders>
  <LastOrder>10926</LastOrder>
</OrdersInfo>
```

Навигация

В языке XQuery имеется множество способов навигации по XML-документу. В действительности в этой книге не хватит места для полного описания всех возможностей навигации в XQuery; следует понимать, что далеко не все они рассмотрены в данном разделе. Основной подход — это использование выражения XPath. С помощью XQuery можно указать абсолютный или относительный путь из текущего узла. Язык XQuery принимает во внимание текущую позицию в документе; это означает, что вы можете ссылаться на путь относительно текущего узла, на который вы переместились в предыдущем выражении пути. Каждый путь состоит из последовательности шагов, перечисленных слева направо. Полный путь может иметь следующий вид:

```
Node-name/child::element-name[@attribute-name=value]
```

Шаги разделены слешами; таким образом, пример пути, описанный здесь, имеет 2 шага. Во втором шаге мы видим подробно, из каких частей может состоять шаг. Каждый шаг может состоять из трех частей.

- *Ось (axis)* указывает направление перемещения. В данном примере значение оси равно `child::`, что указывает на дочерние узлы по отношению к узлу из предыдущего шага.
- *Тест узла (node test)* указывает критерий для выбора узлов. В нашем примере тестом (проверкой) узла является `element-name`; он выбирает только узлы с именем `element-name`.
- *Предикат (predicate)* еще более сужает поиск. В данном примере есть один предикат: `[@attribute-name=value]`, который выбирает только узлы, которые имеют атрибут с именем `attribute-name` и значением этого атрибута `value`, такие как `[@orderid=10952]`.

Обратите внимание, в примере предиката есть ссылка на ось `attribute::`; символ @ является обозначением выражения оси `attribute::`. Это выглядит несколько запутанно; нужно просто рассматривать навигацию в XML-документе в четырех направлениях: вверх (по иерархии), вниз (по иерархии), здесь (в текущем узле) и

вправо (на уровне текущего контекста, для нахождения атрибутов). В табл. 7.2 описаны оси, поддерживаемые в SQL Server.

Таблица 7.2. Оси, поддерживаемые в SQL Server

Ось	Обозначение	Описание
child::		Возвращает потомка текущего узла контекста. Это ось по умолчанию; ее можно пропустить. Направление — вниз
descendant::		Извлекает всех потомков узла контекста. Направление — вниз
self::		Извлекает узел контекста. Направление — здесь
descendant-or-self::	//	Извлекает узел контекста и всех его потомков. Направление — здесь и затем вниз
attribute::	@	Извлекает указанный атрибут узла контекста. Направление — вправо
parent::	..	Извлекает родителя узла контекста. Направление — вверх

Test (проверка) узла стоит после указанной вами оси. Тест узла может быть таким же простым, как тест имени. Указывая имя, вы говорите о том, что вам нужны узлы с этим именем. Также можно использовать подстановочные символы. Звездочка (*) означает, что вам нужен любой основной узел с любым именем. Основной узел — это разновидность узла для оси. Основной узел — это атрибут, если ось представлена как attribute::, или это элемент для всех других осей. Можно также сузить поиск с помощью подстановочных символов. Если вам нужны все основные узлы пространства имен prefix, используйте выражение prefix:*. Если вам нужны все основные узлы с именем local-name, без учета, к какому пространству имен они принадлежат, используйте выражение *:local-name.

Можно также выполнить проверку типов узлов, что поможет запросить узлы, которые не являются основными. Можно использовать следующие проверки типов узлов:

- comment() — позволяет выбирать узлы комментариев;
- node() — значение True для узлов любого типа. Не следует путать с подстановочным символом *; звездочка означает любой основной узел, тогда как node() означает вообще любой узел;
- processing-instruction() — позволяет выбрать узел инструкций по обработке;
- text() — позволяет извлечь текстовые узлы или узлы без тегов.

СОВЕТ

Подготовка к экзамену

Навигация по XML может быть весьма непростой; убедитесь в том, что вы досконально разобрались в описании полного пути.



Предикаты

К основным предикатам относятся числовые предикаты и предикаты со значениями типа Boolean. Числовые предикаты просто выбирают узлы по позиции. Они заключаются в скобки. Например, `/x/y[1]` означает первый дочерний элемент у каждого элемента `x`. Также можно использовать круглые скобки для того, чтобы применить числовой предикат ко всему результату пути. Например, `(/x/y)[1]` означает первый элемент всех узлов, выбранных по `x/y`.

Предикаты со значениями типа Boolean выбирают все узлы, для которых предикат принимает значение `true`. Язык XQuery поддерживает логические операторы `and` и `or`. Но вас может удивить то, как работают операторы сравнения. Они обрабатывают и атомарные значения, и последовательности. Что касается последовательностей, если одно атомарное значение в последовательности приводит к значению `true` выражения, все выражение принимает также значение `true`. Посмотрите на следующий пример:

```
DECLARE @x AS XML = N'';
SELECT @x.query('(1, 2, 3) = (2, 4)');      -- true
SELECT @x.query('(5, 6) < (2, 4)');        -- false
SELECT @x.query('(1, 2, 3) = 1');           -- true
SELECT @x.query('(1, 2, 3) != 1');          -- true
```

Первое выражение принимает значение `true`, поскольку число 2 есть в обеих последовательностях. Второе выражение принимает значение `false`, потому что ни одно из атомарных значений из первой последовательности не имеет значения меньше, чем любое значение из второй последовательности. Третье выражение принимает значение `true`, поскольку имеется атомарное значение в последовательности слева, которое равно атомарному значению справа. Четвертое выражение имеет значение `true`, потому что существует атомарное значение в последовательности слева, которое не равно атомарному значению справа. Не правда ли, интересный результат? Последовательность `(1, 2, 3)` и равна, и не равна атомарному значению 1. Если это вас смущает, используйте *операторы сравнения значений*. (Знакомые символьные операторы из предыдущего примера называются *общими операторами сравнения* в XQuery.) Операторы сравнения значений не воздействуют на последовательности, они воздействуют на единичные величины. В следующем примере показано использование операторов сравнения значений.

```
DECLARE @x AS XML = N'';
SELECT @x.query('(5) lt (2)');            -- false
SELECT @x.query('(1) eq 1');              -- true
SELECT @x.query('(1) ne 1');              -- false
GO
DECLARE @x AS XML = N'';
SELECT @x.query('(2, 2) eq (2, 2)');    -- ошибка
GO
```

Обратите внимание, последний запрос, который находится в отдельном пакете, выдает ошибку, потому что он пытается использовать оператор сравнения значений

на последовательностях. В табл. 7.3 перечислены общие операторы сравнения и соответствующие им операторы сравнения значений.

Таблица 7.3. Общие операторы сравнения и операторы сравнения значений

Общий оператор	Оператор сравнения значений	Описание
=	eq	Равно
!=	ne	Не равно
<	lt	Меньше
<=	le	Меньше или равно
>	gt	Больше
>=	ge	Больше или равно

Язык XQuery также поддерживает условные выражения `if..then..else` со следующим синтаксисом:

```
if (<exp1>
then
    <exp2>
else
    <exp3>
```

Заметьте, выражение `if..then..else` не используется для изменения хода выполнения программы запроса XQuery. Оно более походит на функцию, которая оценивает параметр логического выражения и возвращает одно выражение или другое в зависимости от значения логического выражения. Оно больше похоже на выражение CASE языка T-SQL, чем на его же инструкцию IF.

Следующий код демонстрирует использование условного выражения.

```
DECLARE @x AS XML = N'<Employee empid="2">
    <FirstName>fname</FirstName>
    <LastName>lname</LastName>
</Employee>';
DECLARE @v AS NVARCHAR(20) = N'FirstName';
SELECT @x.query('if (sql:variable("@v")="FirstName") then
    /Employee/FirstName
else
    /Employee/LastName') AS FirstOrLastName;
GO
```

В этом случае предполагаемым результатом является имя сотрудника с ID, равным 2. Если бы вы изменили значение переменной `@v`, результатом запроса была бы фамилия этого сотрудника.

Выражения FLWOR

Настоящая сила языка XQuery заключена в его так называемых выражениях FLWOR. FLWOR — это аббревиатура слов for, let, where, order by и return. Выражение FLWOR

можно использовать для выполнения итераций на последовательности, возвращенное выражением XPath. Хотя обычно выполняются итерации на последовательности узлов, выражения FLWOR можно использовать для выполнения итераций на любой последовательности. Можно ограничить обрабатываемые узлы с помощью предиката, сортировать узлы и форматировать возвращаемый XML. Инструкция FLWOR состоит из следующих частей.

- **for.** С помощью предложения `for` итераторы-переменные привязываются к входным последовательностям. Входными последовательностями могут быть либо последовательности узлов, либо последовательности атомарных значений. Последовательности атомарных значений создаются с помощью литералов и функций.
- **let.** С помощью необязательного предложения `let` выполняется присвоение значения переменной для определенной итерации. Используемое для присвоения выражение может возвращать последовательность узлов или последовательность атомарных значений.
- **where.** С помощью необязательного предложения `where` выполняется фильтрация итерации.
- **order by.** С помощью предложения `order by` можно контролировать порядок, в котором обрабатываются элементы входной последовательности. Порядок контролируется на основе атомарных значений.
- **return.** Предложение `return` оценивается один раз на итерацию, и результаты возвращаются клиенту в порядке итерации. Это предложение форматирует результирующий XML.

Далее приведен пример использования всех предложений FLWOR.

```
DECLARE @x AS XML;
SET @x = N'
<CustomersOrders>
  <Customer custid="1">
    <!-- Комментарий 111 -->
    <companyname>Customer NRZBB</companyname>
    <Order orderid="10692">
      <orderdate>2007-10-03T00:00:00</orderdate>
    </Order>
    <Order orderid="10702">
      <orderdate>2007-10-13T00:00:00</orderdate>
    </Order>
    <Order orderid="10952">
      <orderdate>2008-03-16T00:00:00</orderdate>
    </Order>
  </Customer>
  <Customer custid="2">
    <!-- Комментарий 222 -->
    <companyname>Customer MLTDN</companyname>
```

```
<Order orderid="10308">
    <orderdate>2006-09-18T00:00:00</orderdate>
</Order>
<Order orderid="10952">
    <orderdate>2008-03-04T00:00:00</orderdate>
</Order>
</Customer>
</CustomersOrders>';
SELECT @x.query('for $i in CustomersOrders/Customer/Order
    let $j := $i/orderdate
    where $i/@orderid < 10900
    order by ($j)[1]
    return
        <Order-orderid-element>
            <orderid>{data($i/@orderid)}</orderid>
            {$j}
        </Order-orderid-element>')
AS [Filtered, sorted and reformatted orders with let clause];
```

Как видно из предложения `for`, запрос выполняет итерации по всем узлам `Order` с помощью переменной-итератора и возвращает эти узлы. Имя переменной-итератора в XQuery должно начинаться со знака доллара (`$`). Предложение `where` ограничивает обработанные узлы `order` только теми, у которых атрибут `orderid` меньше 10900.

Выражение, переданное предложению `order by`, должно возвращать значения типа, совместимого с оператором `gt` языка XQuery. Как вы помните, оператор `gt` ожидает атомарные значения. Этот запрос сортирует возвращаемый XML по элементу `orderdate`. Хотя мы имеем один элемент `orderdate` на заказ, XQuery не знает об этом и рассматривает `orderdate` как последовательность, а не атомарное значение. Числовой предикат указывает первый элемент `orderdate` заказа как значение для сортировки. Без этого числового предиката вы получили бы здесь ошибку.

Предложение `return` формирует возвращаемый XML. Оно преобразует атрибут `orderid` в элемент, создав этот атрибут вручную и выбрав только значение атрибута с функцией `data()`. Оно возвращает также элемент `orderdate` и заворачивает оба в элемент `Order-orderid-element`. Обратите внимание на фигурные скобки, окружающие выражения, которые извлекают значение элемента `orderid` и элемента `orderdate`. Язык XQuery оценивает выражение в фигурных скобках; без них все обрабатывалось бы как строковый литерал, и таким был бы и возвращенный результат.

Предложение `let` присваивает имя выражению `$i/orderdate`. Это выражение повторяется дважды в запросе, в предложениях `order by` и `return`. Для задания имени выражению необходимо использовать переменную, отличную от `$i`. Язык XQuery вставляет это выражение каждый раз, когда выполняется ссылка на новую переменную. Далее приведен результат запроса.

```
<Order-orderid-element>
  <orderid>10308</orderid>
  <orderdate>2006-09-18T00:00:00</orderdate>
</Order-orderid-element>
<Order-orderid-element>
  <orderid>10692</orderid>
  <orderdate>2007-10-03T00:00:00</orderdate>
</Order-orderid-element>
<Order-orderid-element>
  <orderid>10702</orderid>
  <orderdate>2007-10-13T00:00:00</orderdate>
</Order-orderid-element>
```

Контрольные вопросы

- Что делает предложение `return` выражений FLWOR?
- Каким должен быть результат выражения `(12, 4, 7) != 7?`

Ответы на контрольные вопросы

- В предложении `return` форматируется результирующий XML-код запроса.
- Результат должен иметь значение `true`.

ПРАКТИКУМ Использование навигации XQuery/XPath

В данном практикуме вы будете использовать выражения XPath для навигации внутри XQuery. Вы начнете с простого выражения пути и затем будете использовать более сложные выражения пути с предикатами.

Задание 1. Использование выражения XPath

В этом задании вы будете использовать простые выражения XPath для возвращения поднаборов XML-данных.

- Если вы закрыли SQL Server Management Studio (SSMS), запустите ее и подключитесь к вашему экземпляру SQL Server. Откройте новое окно запроса, нажав кнопку **New Query** (Новый запрос).
- Подключитесь к базе данных TSQL2012.
- Используйте следующий экземпляр XML для тестирования навигации.

```
DECLARE @x AS XML;
SET @x = N'
<CustomersOrders>
  <Customer custid="1">
    <!-- Комментарий 111 -->
    <companynamen>Customer NRZBB</companynamen>
    <Order orderid="10692">
```

```
<orderdate>2007-10-03T00:00:00</orderdate>
</Order>
<Order orderid="10702">
    <orderdate>2007-10-13T00:00:00</orderdate>
</Order>
<Order orderid="10952">
    <orderdate>2008-03-16T00:00:00</orderdate>
</Order>
</Customer>
<Customer custid="2">
    <!-- Комментарий 222 -->
    <companyname>Customer MLTDN</companyname>
    <Order orderid="10308">
        <orderdate>2006-09-18T00:00:00</orderdate>
    </Order>
    <Order orderid="10952">
        <orderdate>2008-03-04T00:00:00</orderdate>
    </Order>
</Customer>
</CustomersOrders>';
```

4. Напишите запрос, который выбирает узлы *Customer* с дочерними узлами. Выберите только основные узлы (в данном контексте это элементы). Результат должен быть подобен следующему, приведенному здесь в сокращенном виде.

1. Principal nodes

```
<companyname>Customer NRZBB</companyname><Order
orderid="10692"><orderdate>2007-
```

Для получения желаемого результата используйте следующий запрос:

```
SELECT @x.query('CustomersOrders/Customer/*')
    AS [1. Principal nodes];
```

5. Теперь возвратите все узлы, а не только основные узлы. Результат должен быть подобен следующему, приведенному здесь в сокращенном виде.

2. All nodes

```
<!-- Комментарий 111 --><companyname>Customer NRZBB</companyname><Order
orderid="106"
```

Используйте следующий запрос для получения желаемого результата.

```
SELECT @x.query('CustomersOrders/Customer/node()')
    AS [2. All nodes];
```

6. Возвратите только узлы комментариев. Результат должен быть подобен следующему:

3. Comment nodes

```
<!-- Комментарий 111 --><!-- Комментарий 222 -->
```

Для получения желаемого результата используйте следующий запрос:

```
SELECT @x.query('CustomersOrders/Customer/comment()')
AS [3. Comment nodes];
```

Задание 2. Использование выражения XPath с предикатами

В этом задании вам нужно использовать выражения XPath с предикатами для возвращения фильтрованных поднаборов XML-данных.

- Используйте следующий экземпляр XML (тот же самый, что и в предыдущем задании) для тестирования навигации.

```
DECLARE @x AS XML;
SET @x = N'
<CustomersOrders>
    <Customer custid="1">
        <!-- Комментарий 111 -->
        <companyname>Customer NRZBB</companyname>
        <Order orderid="10692">
            <orderdate>2007-10-03T00:00:00</orderdate>
        </Order>
        <Order orderid="10702">
            <orderdate>2007-10-13T00:00:00</orderdate>
        </Order>
        <Order orderid="10952">
            <orderdate>2008-03-16T00:00:00</orderdate>
        </Order>
    </Customer>
    <Customer custid="2">
        <!-- Комментарий 222 -->
        <companyname>Customer MLTDN</companyname>
        <Order orderid="10308">
            <orderdate>2006-09-18T00:00:00</orderdate>
        </Order>
        <Order orderid="10952">
            <orderdate>2008-03-04T00:00:00</orderdate>
        </Order>
    </Customer>
</CustomersOrders>';
```

- Возвратите все заказы для клиента 2. Результат должен быть подобен следующему, приведенному здесь в сокращенном виде.

4. Customer 2 orders

```
<Order orderid="10308"><orderdate>2006-09-18T00:00:00</orderdate></Order><Order
```

Для получения желаемого результата используйте следующий запрос:

```
SELECT @x.query('//Customer[@custid=2]/Order')
AS [4. Customer 2 orders];
```

3. Возвратите все заказы с номером заказа 10952, без учета клиента. Результат должен быть подобен следующему, приведенному здесь в сокращенном виде.

5. Orders with orderid=10952

```
<Order orderid="10952"><orderdate>2008-03-
16T00:00:00</orderdate></Order><Order
```

Для получения желаемого результата используйте следующий запрос:

```
SELECT @x.query('//Order[@orderid=10952]')
AS [5. Orders with orderid=10952];
```

4. Возвратите второго клиента, имеющего хотя бы один заказ. Результат должен быть подобен следующему, приведенному здесь в сокращенном виде.

6. 2nd Customer with at least one Order

```
<Customer custid="2"><!-- Комментарий 222 --><companyname>Customer
MLTDN</companyname
```

Для получения желаемого результата используйте следующий запрос:

```
SELECT @x.query('(/CustomersOrders/Customer/Order/parent::Customer) [2]')
AS [6. 2nd Customer with at least one Order];
```

Резюме занятия

- Язык XQuery можно использовать в запросах T-SQL, чтобы запрашивать XML-данные.
- Язык XQuery поддерживает собственные типы данных и функции.
- Выражение XPath используется для навигации по экземпляру XML.
- Настоящая сила XPath заключена в выражениях FLWOR.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какое из перечисленных не является предложением FLWOR?
 - A. for.
 - B. let.
 - C. where.
 - D. over.
 - E. return.

2. Какой тест типа узла может быть использован для извлечения всех узлов экземпляра XML?
 - A. Звездочка (*).
 - B. comment().
 - C. node().
 - D. text().
3. Какое условное выражение поддерживается в XQuery?
 - A. IIF.
 - B. if..then..else.
 - C. CASE.
 - D. switch.

Занятие 3. Использование типа данных XML

XML — это стандартный формат для обмена данными между разными приложениями и платформами. Он широко применяется и поддерживается почти всеми современными технологиями. Базы данных просто обязаны работать с XML. Хотя XML может быть сохранен как простой текст, представление в виде простого текста означает полное отсутствие знаний о структуре, встроенной в XML-документ. Можно выполнить декомпозицию текста, сохранить его в нескольких связанных таблицах и использовать реляционные технологии для манипулирования данными. Реляционные структуры довольно статичны, и изменять их непросто. Подумайте о динамических или быстро изменяющихся структурах XML. Сохранение XML-данных в собственном типе данных XML решает эти проблемы, разрешая функциональность, добавленную к этому типу данных и обеспечивающую поддержку разнообразных XML-технологий.

Изучив материал этого занятия, вы сможете:

- ✓ Использовать тип данных XML и его методы
- ✓ Индексировать XML-данные

Продолжительность занятия — 45 минут.

Когда используется тип данных XML

Схема базы данных иногда часто меняется. Подумайте о ситуациях, в которых вам нужно поддерживать множество различных схем для одного и того же типа событий. В SQL Server имеется множество таких случаев. Триггеры языка описания данных (data definition language, DDL) и расширенные события — вот хорошие примеры этого. Существуют десятки различных событий DDL. Каждое событие возвращает различные сведения о событиях; каждое событие возвращает данные с разными схемами. Решение сделать так, чтобы триггеры DDL возвращали ин-

формацию о событии в XML-формате с помощью функции `eventdata()`, было осознанным проектным решением. Информация о событиях в формате XML довольно просто подлежит изменению. К счастью, используя такую архитектуру, SQL Server сможет легко распространить поддержку новых DDL-событий на следующие версии.

Другим интересным примером поддержки внутреннего XML являются инструкции `Showplan XML`. С помощью инструкций `SET SHOWPLAN_XML` и `SET STATISTICS XML` можно генерировать информацию плана выполнения в XML-формате. Представьте себе размер приложений и инструментов, которым нужна информация плана выполнения — теперь запросите ее и выполните синтаксический разбор не составляет труда. Можно даже заставить оптимизатор использовать указанный план выполнения, представив XML-план в подсказке к запросу `USE PLAN`.

Еще одна точка применения XML — представление разреженных данных. Данные становятся разреженными, если имеется множество значений `NULL`, когда некоторые столбцы неприменимы ко всем строкам. Стандартные решения такой ситуации предлагают подтипы данных или реализацию открытой модели схемы в реляционной среде. Но решение на основе XML — наиболее простое для реализации. Решение, предлагающее подтипы данных, может привести к созданию множества новых таблиц. В SQL Server 2008 появились разреженные столбцы и фильтруемые индексы. Разреженные столбцы могут быть еще одним решением использования атрибутов, которые неприменимы ко всем строкам в таблице. В разреженных столбцах обеспечивается оптимизированное хранение значений `NULL`. Если появляется необходимость их индексировать, можно эффективно использовать фильтруемые индексы для индексации только известных значений; таким образом оптимизируется таблица и индексируются сохраненные данные. Кроме этого, можно иметь доступ ко всем разреженным столбцам сразу с помощью набора столбцов. Набор столбцов — это XML-представление всех разреженных столбцов, которое даже может обновляться. Однако использование разреженных столбцов и набора столбцов дает схему более сложной по сравнению со схемой, в которой применяются явные XML-столбцы.

Существуют и другие причины для использования XML-модели. XML по определению поддерживает иерархические и упорядоченные данные. Если вашим данным присуще упорядочение, возможно принятие решения о хранении их как XML. Вы также можете получить XML-документы от вашего бизнес-партнера, возможно, без необходимости дробить документ на таблицы. Тогда более резонным может быть сохранение полного XML-документа в базе данных без его дробления.

Методы типа данных XML

В разделе, посвященном введению в XQuery, данной главы вы уже познакомились с типом данных `xml`. XQuery был параметром для метода `query()` этого типа данных. Тип данных `xml` включает в себя 5 методов, которые принимают XQuery как параметр. Эти методы поддерживают запрос (метод `query()`), извлечение атомарных значений (метод `value()`), проверку существования (метод `exist()`), модифи-

кацию разделов в XML-данных (метод `modify()`) в противоположность перезаписи всего документа и дробление XML-данных на несколько строк в результирующем наборе (метод `nodes()`). Вы будете использовать методы типа данных `xml` в практикуме к этому занятию.

Метод `value()` типа данных `xml` возвращает скалярную величину, поэтому он может быть указан в любом месте, где разрешены скалярные значения, например в списке `SELECT` запроса. Обратите внимание, метод `value()` принимает выражение XQuery в качестве первого входного параметра. Вторым параметром служит возвращенный тип данных SQL Server. Метод `value()` должен возвращать скалярное значение; поэтому необходимо указать позицию элемента в последовательности, которую вы просматриваете, даже если вы знаете, что она только одна.

Метод `exist()` можно использовать для проверки, существует ли указанный узел в экземпляре XML. Типичное использование этого выражения — в предложении `WHERE` запросов T-SQL. Метод `exist()` возвращает бит — флагок, который представляет значение `true` или `false`. Он может возвращать следующие значения:

- 1, представляющую значение `true`, если выражение XQuery в запросе возвращает непустой результат. Это означает, что искомый узел существует в экземпляре XML;
- 0, представляющий значение `false`, если выражение XQuery возвращает пустой результат;
- `NULL`, если экземпляр XML имеет значение `NULL`.

Метод `query()`, как понятно из его названия, используется для запроса XML-данных. Вы уже знаете об этом методе из предыдущего занятия данной главы. Он возвращает экземпляр нетипизированного значения XML.

Тип данных `xml` — это большой тип объекта. Количество данных, хранящихся в столбце этого типа, может быть очень большим. Может быть непрактично заменять все значение, когда в действительности все, что требуется — это только изменить его небольшую часть, например скалярное значение какого-либо вложенного элемента. Тип данных `xml` в SQL Server предоставляет метод `modify()`, в принципе подобный методу `WRITE`, который может быть использован в инструкции `UPDATE` языка для типа данных `VARCHAR(MAX)` и других типов `MAX`. Метод `modify()` вызывается в инструкции `UPDATE` языка T-SQL.

Стандарт W3C не поддерживает модификацию данных с помощью языка XQuery. Однако SQL Server имеет собственные расширения языка для поддержки модификации данных в языке XQuery. Язык XQuery в SQL Server поддерживает три ключевых слова языка манипулирования данными (DML) для модификации данных: `insert`, `delete` и `replace value of`.

Метод `nodes()` полезен, когда требуется раздробить XML-значение на реляционные данные. Таким образом, его назначение то же самое, что у функции набора данных `OPENXML`, описанной в занятии 1 этой главы. Но использование метода `nodes()`, как правило, намного быстрее, чем подготовка DOM с вызовом хранимой процедуры `sp_xml_preparedocument`, выполнения инструкции `SELECT...FROM` функции `OPENXML` и

вызыва sp_xml_removedocument. Метод nodes() подготавливает DOM внутренним образом, в процессе выполнения инструкции SELECT языка T-SQL. Подход OPENXML может быть быстрее, если DOM подготавливается один раз и затем дробится несколько раз в одном и том же пакете.

Результатом метода nodes() является результирующий набор, который содержит логические копии оригинальных экземпляров XML. В этих логических копиях узел контекста каждого экземпляра строки ставится в соответствие одному узлу, указанному выражением XQuery, что означает, что вы получаете строку на каждый отдельный узел, начиная с начальной точки, определенной выражением XQuery. Метод nodes() возвращает копии значений XML, так что вы должны использовать дополнительные методы для извлечения скалярных значений из них. Метод nodes() должен вызываться для каждой строки в таблице. С помощью оператора APPLY языка T-SQL можно вызывать выражение из правой таблицы для каждой строки выражения левой таблицы в части FROM.

Использование типа данных XML для динамической схемы

В данном занятии вы на примере узнаете, как использовать тип данных XML внутри базы данных. Этот пример показывает, как можно сделать схему реляционной базы данных динамической. В примере используется таблица Products базы данных TSQL2012.

Представьте, что вам нужно сохранить некоторые определенные атрибуты только для напитков и какие-то другие атрибуты только для приправ. Например, только для напитков нужно сохранить процент рекомендуемого дневного потребления (recommended daily allowance, RDA) витаминов и краткое описание только для приправ, чтобы указать их общий тип (такие как сладкий, острый или соленый). Можно добавить столбец с типом данных XML в таблицу Production.Products базы данных TSQL2012; для нашего примера назовем его additionalattributes. Поскольку другие категории продуктов не имеют дополнительных атрибутов, этот столбец должен допускать значения null. Следующий код изменяет таблицу Production.Products добавлением этого столбца.

```
ALTER TABLE Production.Products  
ADD additionalattributes XML NULL;
```

Прежде чем вставлять данные в новый столбец, можно создать ограничения для значений этого столбца. Вам следует использовать типизированный XML, XML, проверенный на соответствие схеме. С помощью XML-схемы вы можете ограничить возможные узлы, типы данных этих узлов и т. д. В SQL Server вы можете проверить XML-данные на соответствие коллекции XML-схем. Это именно то, что нужно для динамической схемы; если бы вы могли проверить XML-данные на соответствие только одной схеме, то не могли бы использовать тип данных XML для решения на основе динамической схемы, поскольку экземпляры XML были бы ограничены одной схемой. Проверка на соответствие коллекции схем дает возмож-

ность поддерживать различные схемы для напитков и приправ. Если бы вам понадобилось проверить соответствие XML-значений только одной схеме, то следовало бы определить лишь одну схему в коллекции.

Схему коллекций нужно создать с помощью инструкции CREATE XML SCHEMA COLLECTION языка T-SQL. Вы должны предоставить на вход XML-схему, XSD-документ. Создание схемы — это задача, которую следует выполнять очень тщательно. Если вы сделаете в схеме ошибку, могут быть приняты неправильные данные, а правильные — наоборот, отклонены.

Простейший способ создания XML-схем — сначала создать реляционные таблицы, а затем использовать параметр XMLSCHEMA предложения FOR XML. Сохраните результирующее XML-значение (схему) в переменной, предоставьте ту переменную на вход инструкции CREATE XML SCHEMA COLLECTION. Следующий код создает вспомогательные пустые таблицы для напитков и приправ, а затем выполняет инструкцию SELECT с использованием предложения FOR XML для создания XML-схемы из этих таблиц. Далее он сохраняет схемы в переменной и создает коллекцию схем из этой переменной. Наконец, после того как коллекция схем создана, код уничтожает вспомогательные таблицы.

```
-- Вспомогательные таблицы
CREATE TABLE dbo.Beverages
( percentvitaminsRDA INT );
CREATE TABLE dbo.Condiments
( shortdescription NVARCHAR(50) );
GO

-- Сохранение схемы в переменной и создание коллекции
DECLARE @mySchema NVARCHAR(MAX);
SET @mySchema = N'';
SET @mySchema = @mySchema +
(SELECT *
FROM Beverages
FOR XML AUTO, ELEMENTS, XMLSCHEMA('Beverages'));
SET @mySchema = @mySchema +
(SELECT *
FROM Condiments
FOR XML AUTO, ELEMENTS, XMLSCHEMA('Condiments'));
SELECT CAST(@mySchema AS XML);
CREATE XML SCHEMA COLLECTION dbo.ProductsAdditionalAttributes AS @mySchema;
GO

-- Удаление вспомогательных таблиц
DROP TABLE dbo.Beverages, dbo.Condiments;
GO
```

Следующий шаг — преобразование XML-столбца из состояния правильного формата (well-formed) в проверенный на соответствие схеме.

```
ALTER TABLE Production.Products
ALTER COLUMN additionalattributes
XML (dbo.ProductsAdditionalAttributes);
```

Информацию о коллекциях схем можно получить с помощью запроса представлений каталога sys.xml_schema_collections, sys.xml_schema_namespaces, sys.xml_schema_components и некоторых других представлений в схеме sys с именами, которые начинаются с xml_schema_. Однако коллекция схем хранится в SQL Server в табличном формате, а не как XML. Имеет смысл выполнить ту же проверку на соответствие схеме также и на стороне клиента. Зачем посыпать данные на сервер, если система управления реляционной базой данных (RDBMS) отклонит их? Можно также выполнить проверку на соответствие коллекции схем с помощью кода Microsoft .NET, пока у вас есть схемы. Поэтому имеет смысл сохранить схемы, которые вы создали с помощью T-SQL, также в файлах на файловой системе. Если вы забыли сохранить схемы в файловой системе, вы все равно можете извлечь их из коллекции схем SQL Server с помощью системной функции xml_schema_namespace. Обратите внимание, что схема, возвращенная этой функцией, может быть лексически такой же, как оригинальная схема, использованная при создании вашей коллекции схем. Комментарии, аннотации и пробелы теряются. Но сохраняются все свойства схемы, использованные для проверки на соответствие.

Следует обратить внимание еще на одну проблему, прежде чем использовать новый тип данных. Как избежать привязки ошибочной схемы к продукту заданной категории? Например, как предотвратить привязку схемы напитков к приправам? Эта проблема решается с помощью триггера; но предпочтительно наличие декларативного и проверочного ограничений. Именно поэтому в коде добавлены пространства имен к схемам. Вам нужно проверить, является ли пространство имен тем же самым, что имя категории продукта. Вы не можете использовать методы типа данных XML внутри ограничений. Вам нужно создать две дополнительные функции: одна извлекает XML-пространство имен XML-столбца additional_attributes, а другая — имя категории продукта. В проверочном ограничении можно проанализировать, равны ли возвращенные значения обеих функций. Далее приведен код, в котором создаются обе функции и добавляется проверочное ограничение для таблицы Production.Products.

```
-- Функция для извлечения пространства имен
CREATE FUNCTION dbo.GetNamespace (@chkcol XML)
RETURNS NVARCHAR(15)
AS
BEGIN
    RETURN @chkcol.value('namespace-uri((/*)[1])', 'NVARCHAR(15)')
END;
GO

-- Функция для извлечения имени категории
CREATE FUNCTION dbo.GetCategoryName (@catid INT)
RETURNS NVARCHAR(15)
AS
BEGIN
    RETURN
        (SELECT categoryname
        FROM Production.Categories)
```

```

        WHERE categoryid = @catid)
END;
GO
-- Добавление ограничения
ALTER TABLE Production.Products ADD CONSTRAINT ck_Namespace
    CHECK (dbo.GetNamespace(additionalattributes) =
        dbo.GetCategoryName(categoryid));
GO

```

Инфраструктура подготовлена. Теперь можно попробовать вставить какие-нибудь верные XML-данные в новый столбец.

```

-- Напитки
UPDATE Production.Products
    SET additionalattributes = N'<Beverages xmlns="Beverages">
        <percentvitaminsRDA>27</percentvitaminsRDA>
        </Beverages>'

WHERE productid = 1;
-- Специи
UPDATE Production.Products
    SET additionalattributes = N'<Condiments xmlns="Condiments">
        <shortdescription>very sweet</shortdescription>
        </Condiments>'

WHERE productid = 3;

```

Чтобы удостовериться, что проверочное ограничение на соответствие схеме работает, вам надо попробовать вставить также какие-то неверные данные.

```

-- Стока вместо типа int
UPDATE Production.Products
    SET additionalattributes = N'<Beverages xmlns="Beverages">
        <percentvitaminsRDA>twenty seven</percentvitaminsRDA>
        </Beverages>'

WHERE productid = 1;
-- Неправильное пространство имен
UPDATE Production.Products
    SET additionalattributes = N'<Condiments xmlns="Condiments">
        <shortdescription>very sweet</shortdescription>
        </Condiments>'

WHERE productid = 2;
-- Неправильный элемент
UPDATE Production.Products
    SET additionalattributes = N'<Condiments xmlns="Condiments">
        <unkownelement>very sweet</unkownelement>
        </Condiments>'

WHERE productid = 3;

```

Вы должны получить сообщение об ошибке для всех трех инструкций UPDATE. Вы можете проверить данные с помощью инструкции SELECT. По окончании вы можете очистить базу данных TSQL2012 с помощью следующего кода:

```
ALTER TABLE Production.Products
    DROP CONSTRAINT ck_Namespace;
ALTER TABLE Production.Products
    DROP COLUMN additionalattributes;
DROP XML SCHEMA COLLECTION dbo.ProductsAdditionalAttributes;
DROP FUNCTION dbo.GetNamespace;
DROP FUNCTION dbo.GetCategoryName;
GO
```

Контрольный вопрос

- Какой метод типа данных XML вы будете использовать для извлечения скалярных значений из экземпляра XML?

Ответ на контрольный вопрос

- Метод value() типа данных XML извлекает скалярные значения из экземпляра XML.

XML-индексы

Тип данных XML — это действительно большой объектный тип. Каждое значение столбца может содержать до 2 Гбайт данных. Последовательный просмотр XML-данных может быть недостаточно эффективным способом извлечения простого скалярного значения. Для реляционных данных можно создать индекс на фильтруемых столбцах, разрешая операцию поиска в индексе вместо сканирования таблицы. Аналогично можно индексировать XML-столбцы с помощью специализированных XML-индексов. Первый индекс, который вы открываете на XML-столбце, является первичным XML-индексом. Этот индекс содержит разделенное материализованное представление значений XML. Для каждого XML-значения в столбце индекс создает несколько строк данных. Количество строк в индексе примерно равно числу узлов в значении XML. Такой индекс может ускорить поиск нужного элемента с помощью метода exist(). После создания первичного XML-индекса можно создать до трех вторичных XML-индексов других типов.

- PATH. Этот вторичный XML-индекс особенно полезен, если в запросах указано выражение пути. Он ускоряет метод exist() лучше, чем первичный XML-индекс. Такой индекс также ускоряет запросы, которые используют метод value() для полностью определенного пути.
- VALUE. Этот вторичный XML-индекс полезен, если запросы основаны на значениях и путь не полностью определен или содержит подстановочные символы.
- PROPERTY. Этот вторичный XML-индекс особенно полезен для запросов, которые извлекают одно или более значений из отдельных экземпляров XML с помощью метода value().

Первичный XML-индекс должен быть создан первым. Он может быть создан только на таблицах с кластеризованным первичным ключом.

ПРАКТИКУМ Использование методов типа данных XML

В данном практикуме вам предстоит использовать методы типа данных XML.

Задание 1. Использование методов `value()` и `exist()`

В этом задании вы будете использовать методы `value()` и `exist()` типа данных XML.

- Если вы закрыли SQL Server Management Studio (SSMS), запустите ее и подключитесь к вашему экземпляру SQL Server. Откройте новое окно запроса, нажав кнопку **New Query** (Новый запрос).
- Подключитесь к базе данных TSQL2012.
- Используйте следующий экземпляр XML для методов типа данных XML.

```
DECLARE @x AS XML;
SET @x = N'
<CustomersOrders>
    <Customer custid="1">
        <!-- Комментарий 111 -->
        <companyname>Customer NRZBB</companyname>
        <Order orderid="10692">
            <orderdate>2007-10-03T00:00:00</orderdate>
        </Order>
        <Order orderid="10702">
            <orderdate>2007-10-13T00:00:00</orderdate>
        </Order>
        <Order orderid="10952">
            <orderdate>2008-03-16T00:00:00</orderdate>
        </Order>
    </Customer>
    <Customer custid="2">
        <!-- Комментарий 222 -->
        <companyname>Customer MLTDN</companyname>
        <Order orderid="10308">
            <orderdate>2006-09-18T00:00:00</orderdate>
        </Order>
        <Order orderid="10952">
            <orderdate>2008-03-04T00:00:00</orderdate>
        </Order>
    </Customer>
</CustomersOrders>';
```

- Напишите запрос, который извлекает имя первого клиента как скалярную величину. Результат должен быть подобен приведенному здесь.

First Customer Name

Customer NRZBB

Используйте следующий запрос для получения желаемого результата.

```
SELECT @x.value('(/CustomersOrders/Customer/companyname)[1]',  
'NVARCHAR(20)')  
AS [First Customer Name];
```

5. Теперь проверьте, существуют ли узлы companyname и address под узлом Customer. Результат должен быть подобен приведенному здесь.

Company Name Exists	Address Exists
1	0

Используйте следующий запрос для получения желаемого результата.

```
SELECT @x.exist('/CustomersOrders/Customer/companyname')  
AS [Company Name Exists],  
@x.exist('/CustomersOrders/Customer/address')  
AS [Address Exists];
```

Задание 2. Использование методов query(), nodes() и modify()

В этом задании вы будете использовать методы query(), nodes() и modify() типа данных XML.

1. Используйте следующий экземпляр XML (тот же, что и в предыдущем задании) для тестирования методов типа данных XML.

```
DECLARE @x AS XML;  
SET @x = N'  
<CustomersOrders>  
  <Customer custid="1">  
    <!-- Комментарий 111 -->  
    <companyname>Customer NRZBB</companyname>  
    <Order orderid="10692">  
      <orderdate>2007-10-03T00:00:00</orderdate>  
    </Order>  
    <Order orderid="10702">  
      <orderdate>2007-10-13T00:00:00</orderdate>  
    </Order>  
    <Order orderid="10952">  
      <orderdate>2008-03-16T00:00:00</orderdate>  
    </Order>  
  </Customer>  
  <Customer custid="2">  
    <!-- Комментарий 222 -->  
    <companyname>Customer MLTDN</companyname>  
    <Order orderid="10308">  
      <orderdate>2006-09-18T00:00:00</orderdate>  
    </Order>
```

```

<Order orderid="10952">
    <orderdate>2008-03-04T00:00:00</orderdate>
</Order>
</Customer>
</CustomersOrders>';

```

2. Возвратите все заказы для клиента со значением `custid=1` (первый клиент в XML-документе) в виде XML. Результат должен быть подобен приведенному здесь.

```

<Order orderid="10692">
    <orderdate>2007-10-03T00:00:00</orderdate>
</Order>
<Order orderid="10702">
    <orderdate>2007-10-13T00:00:00</orderdate>
</Order>
<Order orderid="10952">
    <orderdate>2008-03-16T00:00:00</orderdate>
</Order>

```

Используйте следующий запрос для получения желаемого результата.

```

SELECT @x.query('//Customer[@custid=1]/Order')
    AS [Customer 1 orders];

```

3. Разделите всю информацию о заказах для клиента со значением `custid=1` (первый клиент в XML-документе). Результат должен быть подобен приведенному здесь.

Order Id	Order Date
10692	2007-10-03 00:00:00.000
10702	2007-10-13 00:00:00.000
10952	2008-03-16 00:00:00.000

Используйте следующий запрос для получения желаемого результата.

```

SELECT T.c.value('./@orderid[1]', 'INT') AS [Order Id],
    T.c.value('./@orderdate[1]', 'DATETIME') AS [Order Date]
FROM @x.nodes('//Customer[@custid=1]/Order')
    AS T(c);

```

4. Обновите имя первого клиента и затем извлеките новые данные. Результат должен быть подобен приведенному здесь.

```
First Customer New Name
```

```
New Company Name
```

Используйте следующий запрос для получения желаемого результата.

```

SET @x.modify('replace value of
    /CustomersOrders[1]/Customer[1]/companynam[1]/text() [1]
    with "New Company Name"';

```

```
SELECT @x.value('(/CustomersOrders/Customer/companyname)[1]',  
'NVARCHAR(20)')  
AS [First Customer New Name];
```

5. Выходите из SSMS.

Резюме занятия

- Тип данных XML полезен для множества сценариев внутри реляционной базы данных.
- Можно проверять XML-экземпляры на соответствие коллекции схем.
- Можно работать с XML-данными с помощью методов типа данных XML.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какой из перечисленных ниже методов не является методом типа данных XML?
 - A. `merge()`.
 - B. `nodes()`.
 - C. `exist()`.
 - D. `value()`.
2. Какой тип XML-индексов вы можете создавать?
 - A. PRIMARY.
 - B. PATH.
 - C. ATTRIBUTE.
 - D. PRINCIPALNODES.
3. Какой метод типа данных XML используется для дробления XML-данных в табличный формат?
 - A. `modify()`.
 - B. `nodes()`.
 - C. `exist()`.
 - D. `value()`.

Упражнения

В следующих упражнениях вы примените полученные знания о запросе и управлении XML-данными. Ответы на вопросы можно найти в *приложении "Ответы" в конце книги*.

Упражнение 1. Создание отчетов из XML-данных

Компания, в которой вы работаете консультантом, использует Web-сайт для получения отзывов о своих продуктах от клиентов. Эти обзоры продуктов хранятся в XML-столбце с именем reviewsXML таблицы ProductReviews. XML-столбец проверен на соответствие схеме и содержит, помимо прочего, элементы `firstname`, `lastname` и `datereviewed`. Компания хочет генерировать отчет с именами авторов отзывов и датами отзыва. Кроме этого, поскольку уже имеется много очень длинных отзывов, компания обеспокоена производительностью этого отчета.

1. Как вы можете получить данные, необходимые для отчета?
2. Что вы можете сделать для максимального повышения производительности отчета?

Упражнение 2. Динамическая схема

Вы должны представить решение для динамической схемы для таблицы Products в вашей компании. Все продукты имеют одинаковые основные атрибуты, такие как ID продукта, имя продукта и заявленная цена товара. Но при этом различные группы продуктов имеют разные дополнительные атрибуты. Помимо динамической схемы для переменной части атрибутов, вы должны обеспечить по крайней мере основные ограничения, такие как типы данных, для этих переменных атрибутов.

1. Как бы вы сделали схему таблицы Products динамической?
2. Как вы можете обеспечить выполнение хотя бы основных ограничений?

Рекомендуемые упражнения

Для успешного освоения экзаменационных тем, представленных в данной главе, выполните следующие задания.

Запрос XML-данных

В демонстрационной базе данных AdventureWorks2012 есть таблица HumanResources.JobCandidate. Она содержит столбец типа данных XML с именем Resume.

- Задание 1.** Найдите все имена и фамилии в этом столбце.
- Задание 2.** Найдите всех кандидатов из Чикаго.
- Задание 3.** Возвратите различные штаты, найденные во всех резюме.

ГЛАВА 8

Создание таблиц и обеспечение целостности данных

Темы экзамена

- Создание объектов базы данных.
 - Создание и изменение таблиц с использованием синтаксиса T-SQL (простые инструкции).
 - Создание и модификация ограничений (простые инструкции).

Таблицы — это основной способ хранения данных в Microsoft SQL Server. Для того чтобы использовать таблицы, необходимо овладеть искусством их создания, а также научиться добавлять ограничения для защиты целостности сохраняемых данных. В этой главе вы узнаете, как создавать и изменять таблицы и как использовать ограничения для обеспечения целостности данных между таблицами.

ПРЕЖДЕ ВСЕГО

Для полного освоения этой главы необходимо иметь:

- понимание реляционных принципов баз данных;
- опыт работы с SQL Server Management Studio (SSMS);
- некоторый опыт в написании кода на языке T-SQL;
- доступ к экземпляру SQL Server 2012 с установленной учебной базой данных TSQL2012.

Занятие 1. Создание и изменение таблиц

Поскольку SQL Server хранит данные в таблицах, жизненно важно понимать команды T-SQL, предназначенные для создания и изменения таблиц. В этом занятии вы познакомитесь с этими командами и их параметрами.

Изучив материал этого занятия, вы сможете:

- ✓ Использовать инструкцию CREATE TABLE для создания таблицы
- ✓ Понимать, как задавать типы данных для столбцов
- ✓ Использовать инструкцию ALTER TABLE для изменения некоторых свойств столбцов
- ✓ Создать таблицу со сжатием

Продолжительность занятия — 45 минут.

Введение

В SQL Server таблица — это основной способ, используемый для хранения данных. Каждая таблица принадлежит одной базе данных, поэтому, когда данные хранятся в таблице, SQL Server защищает их посредством создания и восстановления резервной копии, а также транзакционного режима работы, описываемых следующим образом.

- При создании резервной копии базы данных все ее таблицы архивируются, а при восстановлении базы данных из архива все эти таблицы восстанавливаются с теми же данными, которые они имели на момент архивирования.
- Когда вы выполняете запрос данных в базе данных, эти данные в конечном итоге находятся либо в текущей таблице, либо в другой таблице, на которую ссылается запрос.
- Даже системные данные сохраняются в SQL Server в специально зарезервированных для этого таблицах, называемых системными таблицами.

В SQL Server таблицы, содержащие данные, часто называются базовыми таблицами в отличие от других таблиц и объектов, которые могут быть получены из таблиц, таких как представления или запросы. Базовая таблица является постоянной в том смысле, что определение и содержимое таблицы остается в базе данных даже после остановки и перезапуска SQL Server.

К другим вариантам таблиц, рассмотренным в данном учебном курсе, относятся следующие:

- *временные таблицы* — это базовые таблицы, которые существуют в базе данных tempdb столько, сколько длится сессия или область действия, которые на них ссылаются (дополнительную информацию можно найти в главе 16);
- *табличные переменные* — это переменные, которые могут хранить данные только на протяжении пакета T-SQL (также рассматривается в главе 16);
- *представления*, которые не относятся к базовым таблицам, а получаются в результате запросов к базовым таблицам, выглядят как таблицы, но не хранят данные (рассматриваются в главе 9);
- *индексированные представления* хранят данные, но определены как представления и обновляются каждый раз, когда обновляются базовые таблицы (рассматриваются в главе 15);



- *производные таблицы и табличные выражения* представляют собой встроенные запросы, на которые в запросах ссылаются как на таблицы (рассматриваются в главе 4).

При работе с таблицами необходимо знать, как создать, удалить и изменить таблицу.

Создание таблицы

В T-SQL можно создать таблицу двумя способами:

- с помощью инструкции CREATE TABLE, где явно определяются компоненты таблицы;
- с помощью инструкции SELECT INTO, которая автоматически создает таблицу, используя выходные данные запроса для основного определения таблицы.

В данном занятии рассматривается только инструкция CREATE TABLE.

Синтаксис инструкции CREATE TABLE представлен в электронной документации по SQL Server 2012 в статье "Инструкция CREATE TABLE (Transact-SQL)" по адресу <http://msdn.microsoft.com/ru-ru/library/ms174979.aspx>. Хотя полное описание является слишком сложным для понимания в данный момент, можно упростить его, рассмотрев первую часть синтаксиса.

```
CREATE TABLE [ database_name . [ schema_name ] . | schema_name . ]
table_name [ AS FileTable ]
( { <column_definition> | <computed_column_definition>
| <column_set_definition> | [ <table_constraint> ] [ ,...n ] } ) [ ON {
partition_scheme_name ( partition_column_name ) | filegroup
| "default" } ] [ { TEXTIMAGE_ON { filegroup | "default" } } [ FILESTREAM_ON
{ partition_scheme_name | filegroup
| "default" } ] [ WITH ( <table_option> [ ,...n ] ) ]
[ ; ]
```

Каждый элемент в коде можно развернуть, а некоторые из элементов могут быть развернуты и далее. В данном занятии рассмотрены следующие элементы:

- имя базы данных;
- имя схемы;
- имя таблицы;
- определение столбца;
- определение вычисляемого столбца;
- ограничение таблицы;
- параметр таблицы.

Рассмотрим пример инструкции CREATE TABLE: создание таблицы Production.Categories из базы данных TSQL2012 (ограничения таблицы рассмотрены в занятии 2).

```
CREATE TABLE Production.Categories (
categoryid INT IDENTITY(1,1) NOT NULL,
```

```
categoryname NVARCHAR(15)      NOT NULL,  
description   NVARCHAR(200)     NOT NULL)  
GO
```

Используя учебную таблицу `Production.Categories`, взгляните на основные составляющие части инструкции `CREATE TABLE`.

При создании таблицы можно указать схему базы данных; в нашем случае это `Production` (вы можете предоставить возможность SQL Server заполнить схему базы данных схемой по умолчанию вашего имени пользователя).

ПРИМЕЧАНИЕ Двухкомпонентное именование таблиц

SQL Server всегда назначает таблице ровно одну схему. Поэтому всегда следует ссылаться на таблицу с использованием двухкомпонентных имен (с указанием имени схемы и таблицы) во избежание ошибок и чтобы сделать код более надежным.

Вы должны указать:

- имя таблицы; в данном случае это `Categories`;
- столбцы таблицы, включая:
 - имена столбцов, в данном случае `categoryid`;
 - типы данных столбцов, в данном случае `INT`.

Вы также можете указать:

- для столбцов:
 - длину символьных типов данных, в данном случае 15 для столбца `categoryname`;
 - точность числовых типов данных и некоторых представлений даты;
 - необязательные специальные типы столбцов (вычисляемые, разреженные, `IDENTITY`, `ROWGUIDCOL`), у нас это `IDENTITY` в случае `categoryid`;
 - порядок сортировки столбца (как правило, используется, если нужно указать нестандартный порядок сортировки);
- ограничения, включая:
 - возможность использования значений `NULL` (столбец `categoryid` определен с ограничением `NOT NULL`);
 - стандартные (по умолчанию) и проверочные ограничения;
 - необязательные параметры сортировки столбцов;
 - первичный ключ (в данном случае `PK_Categories`);
 - ограничения внешнего ключа;
 - ограничения уникальности;
- возможные направления хранения таблиц, включая:
 - файловую группу (в данном случае `ON [PRIMARY]`, что означает файловую группу `primary`);

- схему секционирования;
- сжатие таблицы.

Принято определять ограничения таблицы позже, после создания таблицы, с помощью команды ALTER TABLE. Мы рассмотрим все их по порядку, начиная с обязательных элементов, в следующих разделах.



Определение схемы базы данных

Каждая таблица принадлежит к группировке объектов внутри базы данных, которую называют схемой базы данных. Схема базы данных — это поименованный контейнер (пространство имен), который можно использовать для того, чтобы группировать таблицы и другие объекты баз данных. Для таблицы Production.Categories базы данных TSQL2012 схемой базы данных является Production.

Главная задача схемы базы данных — сгруппировать множество объектов баз данных, таких как таблицы, вместе. Для таблиц схема базы данных также позволяет многим таблицам с одним и тем же именем таблицы принадлежать разным схемам. Такой подход срабатывает, потому что схема базы данных становится частью имени таблицы и помогает идентифицировать таблицу. Если вы не укажете имя схемы базы данных при создании таблицы, SQL Server сделает это, используя стандартную схему, соответствующую вашему имени пользователя базы данных.

ВАЖНО!

Схема базы данных и схема таблицы

Не следует путать термин "схема базы данных" с термином "схема таблицы". Схема базы данных — это контейнер объектов в пределах базы данных. Схема таблицы — это определение таблицы, которое включает инструкцию CREATE TABLE со всеми определениями столбцов.

Например, посмотрите следующий пример:

```
SELECT TOP (10) categoryname FROM Production.Categories;
```

Имя Production.Categories указывает имя таблицы внутри базы данных. Могут быть и другие объекты с тем же именем базы данных Categories, но только один объект с таким же именем может существовать в схеме Production. Поэтому для точного определения имени таблицы следует представить имя схемы базы данных.

Следующие четыре встроенных схемы баз данных не могут быть удалены:

- схема базы данных dbo — это стандартная (по умолчанию) схема базы данных для новых объектов, созданных пользователями, имеющими роли db_owner или db_ddl_admin;
- схема guest используется для объектов, которые должны быть доступны пользователю guest. Эта схема используется редко;
- схема INFORMATION_SCHEMA используется представлениями Information Schema, которая предоставляет доступ к метаданным по стандарту ANSI;
- схема базы данных sys зарезервирована SQL Server для системных объектов, таких как системные таблицы и представления.

Дополнительный набор схем базы данных называют по имени встроенных ролей базы данных, и хотя они могут быть удалены, они предназначены для объединения с ролями баз данных. Они также используются редко.

До выхода версии SQL Server 2005, когда имена пользователей, которые владели объектами, совпадали со схемами, было принято назначать объекты пользователю dbo, если доступ к ним был нужен всем пользователям. Начиная с SQL Server 2005, можно создавать схемы, которые не имеют непосредственной связи с пользователями и могут служить для предоставления более тонкой структуры разрешений для таблиц базы данных. Например, в базе данных TSQl2012 существуют 4 схемы базы данных, определенные пользователем: HR, Production, Sales и Stats.

Следует отметить, что в списке таблиц в SQL Server Management Studio (SSMS) каждая таблица имеет две части имени: *имя схемы базы данных*, стоящее перед *именем таблицы в схеме*, как например Production.Categories.

ПРИМЕЧАНИЕ Схемы баз данных не могут иметь вложенности

Возможен только один уровень схемы базы данных; одна схема не может включать в себя другую схему.

Каждой схемой базы данных может владеть только один авторизованный пользователь базы данных. Этот пользователь может затем дать разрешения на объекты этой схемы другим пользователям. Например, следующая инструкция создает схему Production.

```
CREATE SCHEMA Production AUTHORIZATION dbo;
GO
```

Именем схемы Production фактически владеет пользователь с именем dbo, а не схема базы данных dbo. Это позволяет одному пользователю (например, dbo) владеть несколькими разными схемами базы данных.

СОВЕТ Подготовка к экзамену

Можно переместить таблицу из одной схемы в другую с помощью инструкции ALTER SCHEMA TRANSFER. При условии, что объект с именем Categories не существует в схеме базы данных Sales, следующая инструкция перемещает таблицу production.Categories в схему базы данных Sales.

```
ALTER SCHEMA Sales TRANSFER Production.Categories;
```

Выполните следующую инструкцию для того, чтобы переместить ее обратно.

```
ALTER SCHEMA Production TRANSFER Sales.Categories;
```

Именование таблиц и столбцов

Вы имеете возможность выбирать самые разные имена для схем, таблиц и столбцов. Однако имеются некоторые важные ограничения и полезные рекомендации для этого, описанные в данном разделе.

Все имена схем, таблиц и столбцов должны быть допустимыми идентификаторами SQL Server. Идентификаторы должны быть длиной не менее одного символа и не более 128 символов.

Существуют два типа идентификаторов: *регулярные идентификаторы* и *идентификаторы с разделителем*.

Регулярные идентификаторы — это имена, которые следуют набору правил и не должны быть окружены разделителями, такими как квадратные скобки ([]) или кавычки (одиночный символ "'). В регулярных идентификаторах символами могут быть:

- буквы в соответствии со стандартом Unicode Standard 3.2;
- десятичные числа Basic Latin или других национальных шрифтов.

Первый символ должен быть буквой в стандарте Unicode Standard 3.2 или знак подчеркивания (_), он не может быть цифрой. Однако существуют два исключения:

- переменные должны начинаться со значка @;
- временные таблицы или процедуры должны начинаться со значка номера (#).

Последующие символы идентификатора могут включать:

- буквы в соответствии с Unicode Standard 3.2;
- числа из набора Basic Latin (от 0 до 9) или других наборов;
- значок @, знак доллара (\$), значок номера (#) и подчеркивание (_).

Регулярный идентификатор не может быть зарезервированным словом T-SQL и не может включать пробелы или не-алфавитно-цифровые символы, отличные от знаков @, \$ и _. Например, имя таблицы Production.Categories использует два правильных регулярных идентификатора: Production как имя схемы и Categories как имя таблицы.

ПРИМЕЧАНИЕ Использование регулярных идентификаторов там, где это возможно

Если используются специальные символы, отличные от @, # и \$ в идентификаторе для имени схемы, таблицы или столбца, это уже будет идентификатор с разделителями, а не регулярный. Обычно рекомендуется использовать регулярные идентификаторы, применяя только буквы, цифры и подчеркивания. Тогда пользователям для ссылки на имена объектов не нужны разделители. Некоторые разработчики T-SQL любят использовать подчеркивания между именами, чтобы сделать их более удобными для чтения. Например, столбец categoryid можно записать как category_id.

Идентификаторы с разделителем — это имена, которые не следуют правилам, определенным для регулярных идентификаторов. Не существует ограничений на символы, которые можно вставлять в такие идентификаторы, но если они не следуют правилам, написанным для регулярных идентификаторов, необходимо использовать в качестве разделителей квадратные скобки или кавычки при ссылке на них. В T-SQL квадратные скобки всегда могут использоваться для идентификаторов с разделителем.

Кавычки в качестве разделителей являются стандартом ANSI SQL. Однако для их использования требуется, чтобы параметр SET QUOTED_IDENTIFIER был установлен в значение ON, что является значением по умолчанию в SQL Server. Поскольку этот параметр можно переключить в значение OFF, использование кавычек небезопасно.

Например, можно создать таблицу следующим образом:

```
CREATE TABLE Production.[Yesterday's News]
```

Или это можно записать таким образом:

```
CREATE TABLE Production."Tomorrow's Schedule"
```

...

Поскольку имеется внутренний пробел и знак апострофа, это не регулярные идентификаторы, и использование разделителей обязательно.

ПРИМЕЧАНИЕ Регулярные идентификаторы более удобны для пользователей

Хотя квадратные скобки можно применять в качестве разделителей, более правильным считается всегда следить за тем, чтобы эти имена следовали правилам создания регулярных идентификаторов. В этом случае, если пользователь не применяет разделители в запросе, запросы все равно будут работать правильно.

При выборе имен для схем, таблиц и столбцов следуйте правилам присвоения имен, принятым в вашей организации или в проекте.

ПРИМЕЧАНИЕ Не используйте слишком длинные имена объектов

Не следует задавать имена схем, таблиц и столбцов слишком длинными. Организации часто принимают решение, как часть соглашения об именовании для имен ограничений и индексов, включать имя таблицы и имена столбцов, используемых в качестве ключей, в имя столбца или таблицы. Поскольку имена ограничений и индексов также должны быть идентификаторами, они не должны превышать максимальной длины идентификатора в 128 символов.

Обычно считается правильным делать имена схем, таблиц и столбцов короткими, но достаточно описательными. Также следует избегать аббревиатур, только если они действительно не общепонятны или необходимы. Например, имя столбца categoryid использует аббревиатуру id (от англ. *identification*), но это настолько принято, что риск неправильного истолкования минимален.

Выбор типов данных для столбцов

Тип данных, используемый для каждого столбца, имеет большое значение. Полная информация о типах данных представлена в [занятии 2 главы 2](#).

Далее приведены краткие рекомендации, которым нужно следовать при выборе типов данных для столбцов.

- Страйтесь использовать наиболее эффективный тип данных: такой, который требует меньше дискового пространства, определяет данные соответствующим образом и не потребует изменения в дальнейшем, при заполнении таблицы данными.
- Для хранения символьных строк, которые, возможно, будут иметь переменную длину, используйте тип данных NVARCHAR или VARCHAR вместо фиксированных NCHAR или CHAR. Если значение столбца может часто обновляться, и особенно если оно короткое, использование фиксированной длины поможет избежать ненужного перемещения строк.

- Типы данных DATE, TIME и DATETIME2 могут хранить данные более эффективно и с большей точностью, чем типы DATETIME и SMALLDATETIME.
- Используйте типы данных VARCHAR (MAX), NVARCHAR (MAX) и VARBINARY (MAX) вместо устаревших типов данных TEXT, NTEXT и IMAGE.
- Используйте тип данных ROWVERSION вместо устаревшего типа TIMESTAMP.
- Типы DECIMAL и NUMERIC — это один и тот же тип данных, но обычно люди предпочтют DECIMAL, поскольку это имя несколько более описательно. Используйте типы данных DECIMAL и NUMERIC вместо FLOAT или REAL, кроме случаев, когда вам действительно нужна точность с плавающей запятой и известны возможные проблемы с округлением.

Значение NULL и значения по умолчанию

Обработка неизвестных значений — это сложная проблема в теории базы данных и столь же сложная при проектировании баз данных. Когда данные невозможно вставить в определенный столбец строки, как вы можете указать это? Язык T-SQL следует стандарту ANSI SQL в вопросе разрешения одного незначимого свойства столбца с именем NULL. NULL — это не значение столбца, это лишь способ обозначить, что значение целиком и полностью неизвестно.

Можно указать, разрешены ли в столбце значения NULL, установив опцию NULL или NOT NULL сразу после типа данных столбца. Вариант NULL означает, что столбец разрешает значения NULL, а NOT NULL означает, что он не разрешает значения NULL. Используйте следующие правила.

- Если вы знаете, что значение столбца может быть необязательным, потому что иногда неизвестно, какое значение нужно указать в момент вставки строки, тогда определите этот столбец как NULL.
- Если вы не хотите разрешать значение NULL в столбце, но при этом хотите указать какие-то значения по умолчанию, чтобы указать, что столбец еще не заполнен, вы можете указать ограничение DEFAULT, добавив предложение DEFAULT сразу за выражением NOT NULL.

Например, можно указать, что значения для столбца description в таблице Production.Categories еще не введены с помощью пустой строки (две одиночных кавычки без пробела между ними: '') в качестве значения по умолчанию.

```
CREATE TABLE Production.Categories (
    categoryid INT IDENTITY(1,1) NOT NULL,
    categoryname NVARCHAR(15) NOT NULL,
    description NVARCHAR(200) NOT NULL DEFAULT ('')
) ON [PRIMARY];
GO
```

Теперь, если приложение вставляет строку с новой категорией, пользователю не нужно добавлять описание немедленно, а можно вернуться к этому позже, когда можно будет обновить реальное значение описания. Дополнительную информацию о значениях по умолчанию см. в занятии 2.

Свойство идентификатора и порядковые номера

В языке T-SQL свойство идентификатора (identity) может быть назначено столбцу для того, чтобы автоматически генерировать последовательность чисел. Его можно применять только к одному столбцу в таблице, и для генерируемой последовательности номеров нужно указать начальное число (seed) и приращение (increment).

Когда это свойство задается в инструкции `CREATE TABLE`, можно указать начальное число (т. е. значение, с которого нужно начинать) и затем величину приращения (т. е. значение, на которое будет увеличиваться каждый последующий номер). Наиболее часто используются значения `seed` и `increment`, равные `(1, 1)`, как показано в следующем примере для таблицы `Production.Categories` базы данных TSQL2012.

```
CREATE TABLE Production.Categories(categoryid INT IDENTITY(1,1) NOT NULL, ...)
```

У многих таблиц TSQL2012 имеются столбцы первичного ключа со свойством идентификатора.

В SQL Server 2012 существует дополнительный способ генерации последовательных номеров с помощью объектов последовательности. Но поскольку объекты последовательности работают иначе, чем свойство столбца `IDENTITY`, они могут оказаться как хорошей заменой свойства идентификатора, так и неудачной.

Дополнительные сведения о свойстве столбца `IDENTITY` и объектах последовательности можно найти в [занятии 1 главы 11](#).

Вычисляемые столбцы

Столбцы можно также определять как значения, которые вычисляются с помощью выражений. Эти выражения могут основываться на значении других столбцов в строке или с использованием функций T-SQL. Например, вы можете запросить данные из таблицы `Sales.OrderDetails` и увидеть, что два столбца, `unitprice` и `qty`, можно перемножить, чтобы получить начальную стоимость элемента заказа (до скидки). Можно выполнить это вычисление в инструкции `SELECT` следующим образом:

```
SELECT TOP (10) orderid, productid,  
       unitprice, qty, unitprice * qty AS initialcost    -- выражение  
FROM Sales.OrderDetails;
```

Вы можете взять это выражение, `unitprice * qty AS initialcost`, и встроить его в инструкцию `CREATE TABLE` как вычисляемый столбец следующим образом:

```
CREATE TABLE Sales.OrderDetails  
( orderid INT NOT NULL,  
  ...  
  initialcost AS unitprice * qty      -- вычисленный столбец  
);
```

Вычисляемый столбец можно также сделать материализованным (`persisted`), т. е. SQL Server будет хранить вычисляемые значения с данными таблицы и не станет вычислять эти значения "на лету". Но если вычисляемый столбец должен быть ма-

териализованным, этот столбец не может воспользоваться никакими функциями, не являющимися детерминированными, а это означает, что выражение не может ссылаться на многие динамические функции, такие как `GETDATE()` или `CURRENT_TIMESTAMP`. Дополнительные сведения о детерминированных функциях можно найти в электронной документации в разделе "Детерминированные и недетерминированные функции" по адресу <http://msdn.microsoft.com/ru-ru/library/ms178091.aspx>.

Сжатие таблиц

Если вы используете редакцию Enterprise Edition SQL Server 2012 (в дополнение к SQL Server 2008 или SQL Server 2008 R2), то можете воспользоваться сжатием данных в таблице, что, помимо использования индексов, позволяет повысить эффективность хранения данных. Существуют два уровня сжатия данных:

- `ROW` — при сжатии на уровне строк SQL Server применяет более компактный формат хранения к каждой строке в таблице;
- `PAGE` — сжатие на уровне страниц включает в себя сжатие на уровне строк плюс дополнительные алгоритмы сжатия, которые могут выполняться на уровне страницы.

Следующая команда добавляет сжатие на уровне строк для таблицы `Production.OrderDetails` как часть инструкции `CREATE TABLE`.

```
CREATE TABLE Sales.OrderDetails
( orderid INT NOT NULL,
  ...
  )
WITH (DATA_COMPRESSION = ROW);
```

Для изменения команды, позволяющей применить сжатие на уровне страниц, достаточно просто задать `DATA_COMPRESSION = PAGE`.

Для изменения таблицы с целью установки режима сжатия таблицы можно также использовать команду `ALTER`.

```
ALTER TABLE Sales.OrderDetails
REBUILD WITH (DATA_COMPRESSION = PAGE);
```

В SQL Server имеется хранимая процедура `sp_estimate_data_compression_savings`, которая поможет определить, выиграет ли таблица с данными от сжатия. Более подробную информацию о сжатии таблиц можно найти в электронной документации в разделе "Сжатие данных" по адресу <http://msdn.microsoft.com/ru-ru/library/cc280449.aspx> и в разделе "sp_estimate_data_compression_savings (Transact-SQL)" по адресу <http://msdn.microsoft.com/ru-ru/library/cc280574.aspx>.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Может ли имя таблицы содержать пробелы, апострофы и прочие нестандартные символы?
2. Какие существуют типы сжатия таблиц?

Ответы на контрольные вопросы

1. Да, имена таблиц и столбцов могут быть идентификаторами с разделителями, содержащими нестандартные символы.
2. Можно использовать сжатие страниц или строк в таблицах. Сжатие на уровне страниц включает в себя сжатие на уровне строк.

Изменение таблицы

После создания таблицы можно изменить ее структуру, а также добавить или удалить некоторые ее свойства, такие как ограничения таблицы, с помощью команды ALTER TABLE. Команду ALTER TABLE можно использовать для следующих целей:

- добавить или удалить столбец, включая вычисляемый столбец (новые столбцы помещаются в конце последовательности столбцов таблицы);
- изменить тип данных столбца;
- изменить допустимость значения NULL (с NULL на NOT NULL или наоборот);
- добавить или удалить ограничение, включая следующие:
 - ограничение первичного ключа (primary key constraint);
 - ограничение уникальности (unique constraint);
 - ограничение внешнего ключа (foreign key constraint);
 - проверочное ограничение (check constraint);
 - ограничение по умолчанию (default constraint).

Если вы хотите изменить определение ограничения или определение вычисляемого столбца, удалите ограничение или столбец со старым определением и добавьте ограничение или вычисляемый столбец снова с новым определением.

Нельзя использовать команду ALTER TABLE для:

- изменения имени столбца;
- добавления свойства идентификатора;
- удаления свойства идентификатора.

Выбор индексов таблицы

Можно выбрать индексы для таблицы при создании таблицы или добавить индексы позже, когда вам станет понятно, как в действительности пользователи запрашивают данные. Некоторые индексы создаются автоматически с ограничениями, мы рассмотрим это в следующем занятии. Общее описание индексов можно найти в главе 15.

ПРАКТИКУМ Создание и изменение таблиц

В этом практикуме вам нужно использовать команду ALTER TABLE для добавления столбцов в таблице и изменения типов данных.

Задание 1. Использование команды *ALTER TABLE* для добавления и изменения столбцов

Рассмотрите следующую инструкцию CREATE TABLE из сценария TSQL2012.sql, которая используется для создания таблицы Production.Categories.

```
/* From TSQL2012.sql:  
-- Создать таблицу Production.Categories  
CREATE TABLE Production.Categories  
( categoryid INT NOT NULL IDENTITY,  
  categoryname NVARCHAR(15) NOT NULL,  
  description NVARCHAR(200) NOT NULL,  
  CONSTRAINT PK_Categories PRIMARY KEY(categoryid) );  
*/
```

В этом задании вы создадите подобную таблицу с именем Production.CategoriesTest, по одному столбцу за один раз. Затем с помощью команды SET IDENTITY_INSERT вы вставите новую строку.

1. Откройте новое окно запроса в SSMS и убедитесь в том, что на вашем сервере имеется свежая копия базы данных TSQL2012. В этом задании вы создадите новую таблицу в базе данных TSQL2012 и затем удалите ее.
2. Создайте таблицу с одним столбцом. Выполните следующие инструкции для того, чтобы создать свою копию исходной таблицы, но для начала только с одним столбцом.

```
USE TSQL2012;  
GO  
CREATE TABLE Production.CategoriesTest  
( categoryid INT NOT NULL IDENTITY );  
GO
```

3. Добавьте столбцы categoryname и description, чтобы ваша таблица соответствовала исходной таблице.

```
ALTER TABLE Production.CategoriesTest  
  ADD categoryname NVARCHAR(15) NOT NULL;  
GO  
ALTER TABLE Production.CategoriesTest  
  ADD description NVARCHAR(200) NOT NULL;  
GO
```

4. Сейчас вы попробуете сделать вставку в копию таблицы из исходной таблицы, но операция вставки закончится неудачей. Выполните следующий код:

```
INSERT Production.CategoriesTest (categoryid, categoryname, description)
    SELECT categoryid, categoryname, description
    FROM Production.Categories;
GO
```

5. Попробуйте снова выполнить те же действия с опцией `IDENTITY_INSERT ON`, которая позволяет вставить строку с явным значением идентификатора.

```
SET IDENTITY_INSERT Production.CategoriesTest ON;
INSERT Production.CategoriesTest (categoryid, categoryname, description)
    SELECT categoryid, categoryname, description
    FROM Production.Categories;
GO
SET IDENTITY_INSERT Production.CategoriesTest OFF;
GO
```

6. Для очистки базы данных удалите таблицу. Этот шаг можно опустить, если вы намерены перейти к следующему заданию:

```
IF OBJECT_ID('Production.CategoriesTest','U') IS NOT NULL
    DROP TABLE Production.CategoriesTest;
GO
```

Задание 2. Работа с *NULL*-столбцами в таблице

Здесь вы будете использовать таблицу из предыдущего задания и изучите процедуру добавления столбца, который не позволяет, а затем разрешает использовать значение `NULL`.

1. Создайте и заполните таблицу из предыдущего задания, выполнив следующий код. Вы можете пропустить этот шаг, если у вас есть таблица в базе данных TSQL2012 из предыдущего задания.

```
-- Создать таблицу Production.CategoriesTest
CREATE TABLE Production.CategoriesTest
( categoryid      INT          NOT NULL IDENTITY,
  categoryname NVARCHAR(15)  NOT NULL,
  description   NVARCHAR(200) NOT NULL,
  CONSTRAINT  PK_CategoriesTest PRIMARY KEY(categoryid) );
-- Заполнить таблицу Production.CategoriesTest
SET IDENTITY_INSERT Production.CategoriesTest ON;
INSERT Production.CategoriesTest (categoryid, categoryname, description)
    SELECT categoryid, categoryname, description
    FROM Production.Categories;
GO
SET IDENTITY_INSERT Production.CategoriesTest OFF;
GO
```

2. Увеличьте размер столбца `description`.

```
ALTER TABLE Production.CategoriesTest
    ALTER COLUMN description NVARCHAR(500) NOT NULL;
GO
```

3. Проверьте таблицу на наличие значений NULL. Как видите, их там нет:

```
SELECT description  
FROM Production.CategoriesTest
```

4. Попробуйте изменить значение в столбце description на NULL. Попытка будет неудачной.

```
UPDATE Production.CategoriesTest  
SET description = NULL  
WHERE categoryid = 8;  
GO
```

5. Измените таблицу и сделайте значение NULL разрешенным в столбце description.

```
ALTER TABLE Production.CategoriesTest  
ALTER COLUMN description NVARCHAR(500) NULL ;  
GO
```

6. Теперь повторите попытку изменить значение. Она завершится удачно.

```
UPDATE Production.CategoriesTest  
SET description = NULL  
WHERE categoryid = 8;  
GO
```

7. Теперь попробуйте изменить этот столбец назад на NOT NULL. Попытка будет неудачной.

```
ALTER TABLE Production.CategoriesTest  
ALTER COLUMN description NVARCHAR(500) NOT NULL ;  
GO
```

8. Повторите попытку, но присвойте столбцу description исходное значение.

```
UPDATE Production.CategoriesTest  
SET description = 'Seaweed and fish'  
WHERE categoryid = 8;  
GO
```

9. Измените столбец description обратно на NOT NULL. Попытка завершится удачно.

```
ALTER TABLE Production.CategoriesTest  
ALTER COLUMN description NVARCHAR(500) NOT NULL ;  
GO
```

10. Для очистки базы данных удалите таблицу.

```
IF OBJECT_ID('Production.CategoriesTest', 'U') IS NOT NULL  
DROP TABLE Production.CategoriesTest;  
GO
```

Резюме занятия

- При создании таблицы задается схема таблицы как пространство имен или контейнер для таблицы.

- Следует соблюдать аккуратность при присвоении имен таблицам и столбцам и делать их описательными.
- Выбирайте наиболее эффективные и точные типы данных для столбцов.
- Выбирайте подходящие свойства для столбцов, такие как свойство столбца `IDENTITY`, и возможность разрешения в столбце значений `NULL`.
- Можно задать возможность сжатия таблицы при ее создании.
- Можно использовать команду `ALTER TABLE` для изменения большинства свойств столбцов уже после создания таблицы.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какие из перечисленных идентификаторов являются регулярными идентификаторами T-SQL? (Выберите все подходящие варианты.)
 - A. `categoryname`.
 - B. `category name`.
 - C. `category$name`.
 - D. `category_name`.
2. Какой тип данных следует использовать вместо `TIMESTAMP`?
 - A. `VARBINARY`.
 - B. `ROWVERSION`.
 - C. `DATETIME2`.
 - D. `TIME`.
3. Как обозначить, что столбец `categoryname` разрешает значения `NULL`?
 - A. `categoryname PERMIT NULL NVARCHAR(15)`.
 - B. `categoryname NVARCHAR(15) ALLOW NULL`.
 - C. `categoryname NVARCHAR(15) PERMIT NULL`.
 - D. `categoryname NVARCHAR(15) NULL`.

Занятие 2. Обеспечение целостности данных

Поскольку базы данных хранят данные в фиксированном виде¹, нужен какой-то способ проверки правильности данных в таблицах базы данных, независимо от того, как данные изменяются во внешних источниках. Эти проверки правильности не

¹ То есть в базе данных хранятся именно данные, а не ссылки на какие-нибудь внешние источники, в которых эти данные могут изменяться. — Ред.

ограничиваются просто типами данных; они проверяют, какие столбцы должны иметь уникальные значения, какие диапазоны допустимых значений столбец может принимать и должно ли значение столбца соответствовать некоторому столбцу в другой таблице.

Когда эти методы проверки данных встроены в определение таблицы, это называется декларативной целостностью данных и реализуется с помощью ограничений таблиц. Для создания этих ограничений вы должны использовать SQL-команды по стандарту ISO, на основании принципа "table-by-table".

В этом занятии рассматриваются типы ограничений, которые можно создавать на таблицах и которые помогут обеспечивать целостность ваших данных.

Изучив материал этого занятия, вы сможете:

- ✓ Реализовать декларативную целостность данных в ваших таблицах
- ✓ Определять и использовать ограничения первичного ключа
- ✓ Определять и использовать ограничения уникальности
- ✓ Определять и использовать ограничения внешнего ключа
- ✓ Определять и использовать проверочные ограничения
- ✓ Определять ограничения по умолчанию

Продолжительность занятия — 30 минут.

Использование ограничений

Лучший способ обеспечения целостности данных в таблицах SQL Server — это создание или объявление ограничений на базовых таблицах. Эти ограничения применяются к таблице и ее столбцам с помощью инструкций CREATE TABLE или ALTER TABLE.

ПРИМЕЧАНИЕ Не используйте устаревшие правила

В самых первых версиях SQL Server не поддерживались ограничения и вместо них использовались "правила" баз данных с помощью команды CREATE RULE. Правила не настолько хорошо подходят для обеспечения целостности данных, как декларативные ограничения. Также правила считаются устаревшими и будут удалены из последующих версий SQL Server. В любом случае следует избегать использования правил и вместо них нужно применять ограничения.

В SQL Server все ограничения на таблицы являются объектами баз данных, так же как таблицы, представления, хранимые процедуры, функции и пр. Таким образом, ограничения должны иметь уникальные имена в пределах базы данных. Но поскольку каждое ограничение таблицы имеет область действия отдельную таблицу, разумно принимать соглашение об именовании, которое устанавливает тип ограничения, имя таблицы и затем, если необходимо, ключевые столбцы, объявленные в ограничении. Например, у таблицы Production.Categories есть первичный ключ с именем PK_Categories. Когда вы принимаете соглашение об именах, подобное этому, легко сказать, что это за объект, по его имени.

Ограничения первичного ключа

Каждая таблица в реляционной базе данных должна иметь способ отличать каждую строку от всех других. Наиболее общий способ — обозначить столбец как первичный ключ, который будет иметь уникальное значение для каждой строки. Иногда может потребоваться комбинация столбцов, но наиболее принятым является использование одного столбца.

Столбец (или комбинация столбцов) в пределах данных таблицы, который уникальным образом идентифицирует каждую строку (как, например, имя категории в таблице Production.Categories базы данных TSQL2012), называется естественным (natural) ключом или бизнес-ключом таблицы. Можно использовать естественный ключ таблицы в качестве ее первичного ключа, но проектировщики баз данных, как правило, считают более подходящим создавать специальный столбец с числовым типом данных (например, integer), который будет иметь уникальное, не имеющее смысла в другом применении, значение, называемое суррогатным ключом. Суррогатный ключ служит в качестве первичного ключа, и уникальность естественного ключа обеспечивается посредством ограничения уникальности.

Например, снова рассмотрим таблицу Production.Categories базы данных TSQL2012. Следующий код показывает, как она определена в сценарии TSQL2012.sql.

```
CREATE TABLE Production.Categories (
    categoryid      INT          NOT NULL IDENTITY,
    categoryname    NVARCHAR(15)  NOT NULL,
    description     NVARCHAR(200) NOT NULL,
    CONSTRAINT PK_Categories PRIMARY KEY(categoryid)
);
```

В этой таблице categoryid является первичным ключом, что можно утверждать из-за добавленного предложения CONSTRAINT в конце инструкции CREATE TABLE. Имя представляемого ограничения — PK_Categories.

Другой способ объявления столбца первичным ключом — использование инструкции ALTER TABLE, которую можно записать следующим образом:

```
ALTER TABLE Production.Categories
    ADD CONSTRAINT PK_Categories PRIMARY KEY(categoryid);
GO
```

Важно помнить, что столбцы, которые вы выбираете в качестве первичных ключей, в конце концов, будут использоваться в других таблицах для ссылки обратно на исходную таблицу. Рекомендуется использовать одно и то же имя для столбцов в обеих таблицах, если возможно. Также можно облегчить для пользователей запрос к ссылочной таблице с помощью описательного имени столбца. Иными словами, выбирайте имя для столбца первичного ключа, которое естественным образом связано с именем таблицы. Тогда будет проще распознать, когда этот столбец является внешним ключом в других таблицах. Вы можете заметить, например, что все первичные ключи в базе данных TSQL2012 — это просто имя таблицы с "id" на конце.

Это делает несложной попытку в других таблицах узнать таблицу, на которую будет ссылаться внешний ключ.

Существует несколько требований для создания первичного ключа на столбце.

- Столбец или столбцы не могут разрешать значение NULL. Если столбец или столбцы разрешают значение NULL, команда ограничения завершится ошибкой.
- Любые данные в таблице должны иметь уникальные значения в столбце или столбцах первичного ключа. При наличии дубликатов инструкция ALTER TABLE завершится ошибкой.
- Может существовать только одно ограничение первичного ключа в таблице в данный момент времени. Если вы попытаетесь создать два ограничения первичного ключа на одной и той же таблице, команда не будет выполнена.

Когда создается первичный ключ, SQL Server обеспечивает ограничение внутренним образом, создавая уникальный индекс на этом столбце и используя столбец или столбцы первичного ключа как ключи этого индекса.

Чтобы просмотреть ограничения первичного ключа в базе данных, можно сделать запрос к таблице sys.key_constraints с фильтрацией по PK.

```
SELECT *
FROM sys.key_constraints
WHERE type = 'PK';
```

Также можно найти уникальный индекс, который использует SQL Server для обеспечения ограничения первичного ключа, запросив sys.indexes. Например, следующий запрос показывает уникальный индекс, объявленный на таблице Production.Categories для ограничения первичного ключа PK_Categories.

```
SELECT *
FROM sys.indexes
WHERE object_id = OBJECT_ID('Production.Categories') AND
      name = 'PK_Categories';
```

Дополнительную информацию об индексах можно найти в главе 15.

Ограничения уникальности

Ограничения уникальности очень похожи на ограничения первичного ключа. Часто у вас будет более одного столбца или набора столбцов, которые уникально определяют строки в таблице. Например, если у вас есть суррогатный ключ, определенный как первичный ключ, вы, скорее всего, также имеете естественный ключ, уникальность которого тоже захотите обеспечить. Вы можете использовать ограничение уникальности для естественных ключей и уникальных бизнес-ключей.

Например, в таблице Production.Categories вы можете захотеть, чтобы были уникальными все названия категорий, тогда можно было бы объявить ограничение уникальности на столбце categoryname следующим образом:

```
ALTER TABLE Production.Categories
ADD CONSTRAINT UC_Categories UNIQUE (categoryname);
GO
```

Также как ограничение первичного ключа, ограничение уникальности автоматически создает уникальный индекс с тем же именем, что и ограничение. Уникальный индекс может быть как кластеризованным, так и некластеризованным. SQL Server использует этот индекс для обеспечения уникальности столбца или комбинации столбцов.

СОВЕТ**Подготовка к экзамену**

Ограничение уникальности не требует, чтобы столбец был NOT NULL. Можно разрешить значение NULL в столбце и при этом иметь ограничение уникальности, но только одна строка может быть NULL.

И первичный ключ, и ограничения уникальности имеют те же ограничения по размеру, что и индекс: можно комбинировать не более 16 столбцов в качестве ключевых столбцов индекса, и максимальная длина должна быть 900 байт по всем этим столбцам.

ПРИМЕЧАНИЕ Ограничения и вычисляемые столбцы

Можно также создать и первичный ключ, и ограничения уникальности на вычисляемых столбцах.

Как и в случае с ограничениями первичного ключа, можно просмотреть список ограничений уникальности в базе данных с помощью запроса к таблице sys.key_constraints, применив фильтр по типу UQ.

```
SELECT *
FROM sys.key_constraints
WHERE type = 'UQ';
```

Можно найти уникальный индекс, который SQL Server использует для обеспечения ограничения первичного ключа, сделав запрос к sys.indexes и применив фильтр по имени ограничения.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как SQL Server обеспечивает уникальность по первичному ключу и ограничению уникальности?
2. Может ли первичный ключ на одной таблице иметь то же самое имя как первичный ключ в другой таблице в одной и той же базе данных и в одной и той же схеме?

Ответы на контрольные вопросы

1. SQL Server использует уникальные индексы для обеспечения уникальности и для первичного ключа, и для ограничения уникальности.
2. Нет, все ограничения таблиц должны иметь уникальные имена в пределах схемы базы данных.

Ограничения внешнего ключа

Внешний ключ — это столбец или комбинация столбцов в одной таблице, которые служат как ссылка для поиска данных в другой таблице. Во второй таблице, часто

называемой таблицей уточняющих запросов (lookup table), соответствующий столбец или комбинация столбцов имеют первичный ключ или ограничение уникальности, примененные к ним, либо уникальный индекс. Так что значение в первой таблице может иметь дубликат, но во второй таблице, где вы запрашиваете соответствующее значение, оно должно быть уникальным. Если вы знаете значение в первой таблице, то отношение внешнего ключа даст вам возможность получить связанные данные из другой таблицы. Для этого вы должны выполнить уточняющий запрос к соответствующим данным.

Например, имеется столбец с именем `categoryid` в таблице `Production.Products`. Этот столбец соответствует первичному ключу `categoryid` в таблице `Production.Categories`. Для любого конкретного продукта в таблице `Products` можно найти связанную информацию о категории с помощью уточняющего запроса к таблице `Categories`.

Можно использовать ограничение внешнего ключа для обеспечения, что каждый элемент в столбце `categoryid` таблицы `Production.Products` — это допустимое значение `categoryid` из таблицы `Production.Categories`. Вот приведен код для создания внешнего ключа:

```
USE TSQL2012
GO
ALTER TABLE Production.Products WITH CHECK
    ADD CONSTRAINT FK_Products_Categories FOREIGN KEY(categoryid)
        REFERENCES Production.Categories (categoryid)
GO
```

Далее описано, как работает эта команда.

- Объявляется ограничение внешнего ключа для таблицы, для которой этот ключ является "внешним, т. е. ключом из другой таблицы. Поэтому вы должны изменить (`ALTER`) таблицу `Production.Products`.
- Следует объявить, разрешаются ли нарушения при создании ограничения. Создание ограничения `WITH CHECK` подразумевает, что если в таблице уже существуют данные и если возможны нарушения ограничения, тогда инструкция `ALTER TABLE` завершится ошибкой.
- Добавляется ограничение и указывается имя ограничения внешнего ключа. В данном случае `TSQL2012` использует `FK_` в качестве префикса для внешних ключей.
- После ввода типа ограничения, `FOREIGN KEY`, в круглых скобках задается столбец (или комбинация столбцов) в этой таблице, на которые накладывается ограничение для проверки посредством поиска в другой таблице.
- Затем устанавливается, что такая другая таблица, т. е. таблица в текущей базе данных, на которую данное ограничение ссылается (`REFERENCES`), вместе со столбцом или комбинацией столбцов. Этот столбец (или столбцы) принадлежит ссылочной (`referenced`) таблице и должен быть первичным ключом или ограничением уникальности в таблице, или вместо этого может использоваться индекс уникальности.

При создании внешнего ключа надо помнить следующие правила:

- столбец или набор столбцов из каждой таблицы должны иметь точно такие же типы данных и параметры сортировки (если они имеют строковый тип данных);
- как уже упоминалось, столбцы из ссылочных таблиц должны иметь уникальный индекс, созданный на них, либо неявно с помощью первичного ключа или ограничения уникальности, либо явно через создание индекса;
- вы также можете создавать внешние ключи и ограничения на вычисляемых столбцах.

Таблицы часто соединяются по внешнему ключу, так что запрос может возвращать данные, связанные в рамках этих двух таблиц. Например, следующий запрос возвращает имя категории `categoryname` для набора продуктов из таблицы `Production.Products`.

```
SELECT P.productname, C.categoryname  
FROM Production.Products AS P  
JOIN Production.Categories AS C  
ON P.categoryid = C.categoryid;
```

Следует отметить, что этот запрос возвращает правильное значение `categoryname` для каждого продукта, поскольку ключевое слово `JOIN` относится к внешнему ключу `P.categoryid` и ссылочному столбцу `C.categoryid` в таблице `Production.Categories`.

СОВЕТ

Подготовка к экзамену

Поскольку соединения часто выполняются на внешних ключах, можно улучшить производительность запроса созданием некластеризованного индекса на внешнем ключе в ссылающейся таблице. Соответствующий столбец в ссылочной (на которую ссылаются) таблице уже имеет уникальный индекс, но если ссылающаяся таблица, в нашем случае `Production.Products`, содержит много строк, SQL Server может быстрее разрешить соединение, если он будет иметь возможность использовать индекс на большой таблице.

Наконец, для того чтобы найти внешний ключ базы данных, можно создать запрос к таблице `sys.foreign_keys`. Следующий запрос находит строку для таблицы `FK_Products_Categories`.

```
SELECT *  
FROM sys.foreign_keys  
WHERE name = 'FK_Products_Categories';
```

Проверочные ограничения

С помощью проверочного ограничения объявляется, что значения столбца ограничены некоторым образом. Значения уже ограничены типом данных, поэтому проверочное ограничение добавляет некоторые дополнительные ограничения по диапазонам или набору разрешенных значений. Когда вы создаете проверочное ограничение, то указываете некоторое выражение таким образом, что SQL Server может ограничить допустимые значения. Выражение может ссылаться на другие столбцы в той же строке таблицы и использовать встроенные функции T-SQL.

Например, таблица Production.Products базы данных TSQL2012 имеет проверочное ограничение с именем `CHK_Products_unitprice` на столбце `unitprice`. Далее описано, как его создать.

```
ALTER TABLE Production.Products WITH CHECK  
ADD CONSTRAINT CHK_Products_unitprice  
CHECK (unitprice>=0);  
GO
```

Столбец `unitprice` уже имеет тип данных `money`, но это не мешает ему хранить отрицательные значения. Однако отрицательная цена не имеет смысла. Можно запретить отрицательные значения, создав ограничение на таблице, ссылаясь на столбец и объявив выражение, которое должно быть истинным: значение `unitprice` больше или равно нулю. Значение меньше нуля не допускается.

Проверочные ограничения имеют несколько преимуществ.

- Их выражения подобны выражениям фильтров в предложении `WHERE` инструкции `SELECT`.
- Ограничение находится в таблице, поэтому оно действует всегда до тех пор, пока указана опция `WITH CHECK`. Если бы подобное ограничение действовало только в приложении вне базы данных, всегда был бы шанс, что данные могли бы попасть в таблицу, что нарушило бы допустимые значения.
- Они могут работать лучше, чем альтернативные способы ограничения столбцов, такие как триггеры.

При использовании проверочных ограничений надо иметь в виду следующее.

- Если в столбце разрешены значения `NULL`, убедитесь в том, что выражение учитывает потенциальные значения `NULL`. Значение `NULL`, например, не является отрицательным, но оно и не положительное. Вставка значения `NULL` проходит ограничение `unitprice >= 0`, но она также проходит и ограничение `unitprice < 0`.
- Нельзя настроить сообщение об ошибке от проверочного ограничения, как это можно сделать при реализации ограничения с помощью триггера.
- Проверочное ограничение не может протестировать факт обновления: невозможно сослаться на предыдущее значение столбца в выражении проверочного ограничения. Если в этом есть необходимость, следует использовать триггер. Например, если нужно создать ограничение, задающее, что при любом обновлении значение `unitprice` не может быть увеличено более чем на 25%, следует использовать триггер.

Можно получить список проверочных ограничений для таблицы, создав запрос к таблице `sys.check_constraints`, следующим образом:

```
SELECT *  
FROM sys.check_constraints  
WHERE parent_object_id = OBJECT_ID('Production.Products');
```

Величина `parent_object_id` — это `object_id` таблицы, которой принадлежит проверочное ограничение.

Ограничение по умолчанию

Последнее ограничение, используемое в таблицах T-SQL, — это ограничение по умолчанию. Фактически можно сказать, что ограничения по умолчанию в действительности ничего не ограничивают; они просто задают значение по умолчанию в процессе вставки (`INSERT`), если не задано никакое другое значение.

Ограничения по умолчанию наиболее полезны, когда у вас есть столбец в таблице, который не разрешает значение `NULL`, но вы не хотите препятствовать успешному выполнению вставки (`INSERT`), если значение из столбца при вставке не указано. Но можно с таким же успехом применять ограничение по умолчанию к столбцу, который не разрешает значение `NULL`, но вы при этом хотите, чтобы вместо применения `NULL` вставлялось значение по умолчанию, когда в инструкции `INSERT` не указано значение.

В качестве примера ограничения по умолчанию рассмотрим такую ситуацию. Столбец `unitprice` таблицы `Production.Products` имеет ограничение по умолчанию, определенное как 0. Хотя вы можете использовать команду `ALTER TABLE` для добавления ограничения по умолчанию, принято помещать его в инструкцию `CREATE TABLE`. Далее приведен пример из базы данных TSQL2012.

```
CREATE TABLE Production.Products
( productid INT NOT NULL IDENTITY,
  productname NVARCHAR(40) NOT NULL,
  supplierid INT NOT NULL,
  categoryid INT NOT NULL,
  unitprice MONEY NOT NULL
    CONSTRAINT DFT_Products_unitprice DEFAULT(0),
  discontinued BIT NOT NULL
    CONSTRAINT DFT_Products_discontinued DEFAULT(0),
  ... );
```

В этом случае, ограничение по умолчанию приведено сразу после типа данных столбца. Здесь используется явное имя. Если вы не укажете явное имя, SQL Server применит имя, сгенерированное машиной.

Наличие значений по умолчанию для столбцов `unitprice` и `discontinued` означает, что инструкция `INSERT` может успешно выполнить добавление новой строки, без необходимости указания значений для этих столбцов. Помните, что ограничения по умолчанию, как и все прочие ограничения, являются объектами в пределах базы данных. Их имена должны быть уникальными по всей базе данных. Никакие две таблицы, принадлежащие одной схеме, не могут иметь ограничения по умолчанию с одинаковыми именами.

Список ограничений по умолчанию можно получить с помощью запроса к таблице `sys.default_constraints`. Следующий запрос находит все ограничения по умолчанию для таблицы `Production.Products`.

```
SELECT *
FROM sys.default_constraints
WHERE parent_object_id = OBJECT_ID('Production.Products');
```

ПРАКТИКУМ Обеспечение целостности данных

В этом практикуме вы будете использовать команду ALTER TABLE для добавления и удаления ограничений на таблице, включая первичный ключ, ограничение уникальности и ограничения внешнего ключа.

Задание 1. Работа с ограничениями первичного и внешнего ключа

Далее приведена инструкция CREATE TABLE таблицы Production.Products, взятой из базы данных TSQL2012.sql.

```
/* -- Создать таблицу Production.Products
CREATE TABLE Production.Products
( productid      INT          NOT NULL IDENTITY,
productname     NVARCHAR(40)  NOT NULL,
supplierid      INT          NOT NULL,
categoryid      INT          NOT NULL,
unitprice       MONEY        NOT NULL
    CONSTRAINT DFT_Products_unitprice DEFAULT(0),
discontinued     BIT          NOT NULL
    CONSTRAINT DFT_Products_discontinued DEFAULT(0),
CONSTRAINT PK_Products PRIMARY KEY(productid),
CONSTRAINT FK_Products_Categories FOREIGN KEY(categoryid)
    REFERENCES Production.Categories(categoryid),
CONSTRAINT FK_Products_Suppliers FOREIGN KEY(supplierid)
    REFERENCES Production.Suppliers(supplierid),
CONSTRAINT CHK_Products_unitprice CHECK(unitprice >= 0) );
*/
```

В этом задании вы будете тестировать ограничения первичного ключа и ограничения внешнего ключа таблицы. Вы будете использовать инструкцию ALTER TABLE для того, чтобы удалить, протестировать и добавить ограничение внешнего ключа обратно в таблицу.

1. Протестируйте первичный ключ, используя следующий код:

```
SELECT productname FROM Production.Products
WHERE productid = 1;
SET IDENTITY_INSERT Production.Products ON;
GO
INSERT INTO Production.Products (productid, productname, supplierid,
categoryid, unitprice, discontinued)
VALUES (1, N'Product TEST', 1, 1, 18, 0);
GO
SET IDENTITY_INSERT Production.Products OFF;
```

2. Вставьте новую строку, которая позволяет свойству идентификатора назначить новый productid.

```
INSERT INTO Production.Products (productname, supplierid, categoryid,
unitprice, discontinued)
VALUES (N'Product TEST', 1, 1, 18, 0);
GO
```

3. Удалите тестовую строку.

```
DELETE FROM Production.Products WHERE productname = N'Product TEST';
GO
```

4. Попробуйте еще раз с неправильным значением categoryid = 99. Вставка потерпит неудачу из-за ограничения внешнего ключа.

```
INSERT INTO Production.Products (productname, supplierid, categoryid,
unitprice, discontinued)
VALUES (N'Product TEST', 1, 99, 18, 0);
GO
```

5. Удалите ограничение внешнего ключа.

```
ALTER TABLE Production.Products DROP CONSTRAINT FK_Products_Categories;
GO
```

6. Снова попробуйте вставить строку с неправильным значением categoryid = 99. Вставка выполнена успешно.

```
INSERT INTO Production.Products (productname, supplierid, categoryid,
unitprice, discontinued)
VALUES (N'Product TEST', 1, 99, 18, 0);
GO
```

7. Попробуйте добавить ограничение внешнего ключа обратно с помощью WITH CHECK. Команда не выполнится.

```
ALTER TABLE Production.Products WITH CHECK
ADD CONSTRAINT FK_Products_Categories FOREIGN KEY(categoryid)
    REFERENCES Production.Categories (categoryid);
GO
```

8. Обновите строку так, чтобы она имела допустимое значение categoryid.

```
UPDATE Production.Products
SET categoryid = 1
WHERE productname = N'Product TEST';
GO
```

9. Теперь попытайтесь добавить ограничение внешнего ключа снова в таблицу. Все выполняется успешно.

```
ALTER TABLE Production.Products WITH CHECK
ADD CONSTRAINT FK_Products_Categories FOREIGN KEY(categoryid)
    REFERENCES Production.Categories (categoryid);
GO
```

10. Удалите тестовую строку из таблицы.

```
DELETE FROM Production.Products WHERE productname = N'Product TEST';
GO
```

Задание 2. Использование ограничений уникальности

В этом задании вам нужно создать ограничение уникальности на столбце productname таблицы Production.Products базы данных TSQL2012. Вы должны выполнить проверку, что все имена уникальны, когда применено ограничение.

1. Проверьте, что все значения productname в таблице Production.Products уникальны.

```
USE TSQL2012;
GO
SELECT productname, COUNT(*) AS productnamecount
FROM Production.Products
GROUP BY productname
HAVING COUNT(*) > 1;
```

2. Вставьте productname для productid = 1; значение равно 'Product HHYDP'.

```
SELECT productname
FROM Production.Products
WHERE productid = 1;
```

3. Используйте инструкцию UPDATE для проверки, может ли быть дубликат имени продукта.

```
UPDATE Production.Products
SET productname = 'Product RECZE'
WHERE productid = 1;
```

4. Убедитесь в том, что имеются дубликаты.

```
SELECT productname, COUNT(*) AS productnamecount
FROM Production.Products
GROUP BY productname
HAVING COUNT(*) > 1;
```

5. Попробуйте добавить ограничение уникальности. Вы увидите, что попытка не удалась.

```
ALTER TABLE Production.Products
ADD CONSTRAINT U_Productname UNIQUE (productname);
```

6. Восстановите исходное имя продукта.

```
UPDATE Production.Products
SET productname = 'Product HHYDP'
WHERE productid = 1;
```

7. Еще раз попробуйте добавить ограничение уникальности.

```
ALTER TABLE Production.Products
ADD CONSTRAINT U_Productname UNIQUE (productname);
```

8. Удалите ограничение уникальности.

```
ALTER TABLE Production.Products
DROP CONSTRAINT U_Productname;
```

Резюме занятия

- Для поддержки целостности данных в таблицах базы данных можно объявить ограничения, которые сохраняются в базе данных.
- Ограничения обеспечивают подчинение данных, введенных в таблицы, более сложным правилам, чем определенные для типов данных и допустимости значений NULL.
- Ограничения таблиц включают в себя ограничения первичного ключа и ограничения уникальности, которые обеспечиваются в SQL Server с помощью уникального индекса. Они также включают ограничения внешнего ключа, гарантирующие, что только данные, правильность которых проверена в другой таблице уточненных запросов (lookup table), разрешены в исходной таблице. Также к ним относятся проверочные ограничения и ограничения по умолчанию, которые применяются к столбцам.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какой из перечисленных столбцов может использоваться в качестве суррогатного ключа? (Выберите все подходящие варианты.)
 - A. Время (в тысячных долях секунды), когда строка была вставлена.
 - B. Автоматически увеличивающееся целое число.
 - C. Последние 4 цифры номера социального страхования, объединенные с первыми 8 знаками фамилии пользователя.
 - D. Уникальный идентификатор (GUID), выбранный из SQL Server в момент вставки строки.
2. Вы хотите гарантировать ввод достоверного значения `supplierid` для каждого значения `productid` в таблице `Production.Products`. Какое ограничение следует использовать?
 - A. Ограничение уникальности.
 - B. Ограничение по умолчанию.
 - C. Ограничение внешнего ключа.
 - D. Ограничение первичного ключа.
3. Какие таблицы метаданных дают возможность получить список ограничений в базе данных? (Выберите все подходящие варианты.)
 - A. `sys.key_constraints`.
 - B. `sys.indexes`.
 - C. `sys.default_constraints`.
 - D. `sys.foreign_keys`.

Упражнения

В следующих упражнениях вы примените полученные знания о таблицах SQL Server и целостности данных. Ответы на вопросы можно найти в *приложении "Ответы" в конце книги*.

Упражнение 1. Работа с ограничениями таблиц

Как ведущий разработчик баз данных на новом проекте, вы заметили, что проверка правильности базы данных выполняется на клиентском приложении. В результате разработчики базы данных периодически запускают очень затратные запросы для проверки целостности данных. Вам нужно принять решение о необходимости рефакторинга базы данных командой для улучшения целостности базы данных и сокращения дорогостоящих, выполняемых каждую ночь, запросов проверки правильности. Ответьте на следующие вопросы о действиях, которые вы можете предпринять.

1. Как можно убедиться, что определенные комбинации столбцов в таблице имеют уникальное значение?
2. Как можно обеспечить ограничение значений в определенных таблицах указанными диапазонами?
3. Как можно гарантировать, что все столбцы, которые содержат значения из таблиц уточняющих запросов, правильны?
4. Как можно гарантировать, что все таблицы имеют первичный ключ, даже те таблицы, которые в данный момент не имеют объявленного первичного ключа?

Упражнение 2. Использование ограничений уникальности и ограничений по умолчанию

При более тщательном обследовании базы данных вашего текущего проекта вы обнаружили, что имеется больше проблем целостности данных, чем вы нашли вначале. Далее перечислены некоторые из обнаруженных вами проблем. Как вы предлагаете их решить?

1. Большинство таблиц имеет суррогатный ключ, который вы реализовали как первичный ключ. Но существуют и другие столбцы или комбинации столбцов, которые должны быть уникальными, и таблица может иметь только один первичный ключ. Как вы можете обеспечить уникальность других конкретных столбцов или комбинаций столбцов?
2. Несколько столбцов разрешают значения NULL, хотя предполагается, что приложение всегда их заполняет. Как можно обеспечить, что эти столбцы никогда не будут разрешать значения NULL?
3. Часто приложение должно указывать определенные значения для каждого столбца при вставке в строку. Как вы можете настроить столбцы так, что если

приложение не вставит значение, автоматически будет вставляться значение по умолчанию?

Рекомендуемые упражнения

Для успешного освоения экзаменационных тем, представленных в данной главе, выполните следующие задания.

Создание таблиц и обеспечение целостности данных

Следующие практические задания расширяют код, с которым вы работали в занятиях и заданиях данной главы. Продолжите их разработку в базе данных TSQL2012.

- **Задание 1.** Используйте команду `ALTER TABLE` для добавления нового столбца с именем `categorystatus` в тестовую таблицу `Production.CategoriesTest` из *задания 1 в занятии 1*. Определите столбец с помощью инструкции `NVARCHAR(15) NOT NULL`. Если таблица содержит данные, добавление столбца `NOT NULL` потерпит крах. Теперь попробуйте ту же команду `ALTER TABLE`, но на этот раз определите столбец с помощью `NVARCHAR(15) NOT NULL DEFAULT ''` (две одиночные кавычки). В чем состоит разница между этими определениями столбцов?
- **Задание 2.** Протестируйте проверочное ограничение `CHK_Products_unitprice` в таблице `Production.Products` с помощью той же логики, что вы использовали в *задании 1 к занятию 2*. Попробуйте вставить новую строку со всеми правильными столбцами, но используйте отрицательное значение `unitprice`, равное `-10`. Удалите проверочное ограничение. Повторите вставку. Попробуйте вернуть проверочное ограничение в таблицу. Обновите вставленную строку так, что в ней имеется положительное значение `unitprice`. Теперь вновь попробуйте добавить проверочное ограничение в таблице. Получится ли у вас добавить снова проверочное ограничение в таблицу, если там не будет строк? Почему?

ГЛАВА 9

Проектирование и создание представлений, встроенных функций и синонимов

Темы экзамена

- Создание объектов базы данных.
 - Создание и изменение представлений (простые инструкции).
 - Проектирование представлений.
- Работа с данными.
 - Запрос данных с помощью инструкции SELECT.

Microsoft SQL Server предлагает три различных способа логического представления таблиц пользовательским запросам без непосредственного обращения к физической базовой таблице. Представления ведут себя точно так же, как таблицы, но могут скрывать сложную логику; встроенные функции могут быть использованы как представления, но могут также иметь параметры; синонимы — это простой способ ссылки на объекты базы данных под другим именем.

В данной главе обсуждается проектирование, создание и изменение объектов, которые представляют базы данных коду T-SQL косвенными способами.

ПРЕЖДЕ ВСЕГО

Для полного освоения этой главы необходимо иметь:

- понимание реляционных принципов баз данных;
- опыт работы с SQL Server Management Studio (SSMS);
- некоторый опыт в написании кода на языке T-SQL;
- доступ к экземпляру SQL Server 2012 с установленной учебной базой данных TSQL2012.

Занятие 1. Проектирование и реализация представлений и встроенных функций

С помощью представлений и встроенных функций можно получить содержимое одной или более базовых таблиц данных пользователя, а также инкапсулировать сложную логику, например соединения и фильтры, таким образом, что пользователям не нужно помнить о них. В этом занятии вы узнаете, как создавать представления и встроенные функции и управлять ими.

Изучив материал этого занятия, вы сможете:

- ✓ Использовать инструкцию `CREATE VIEW` для создания представления
- ✓ Понимать, как проектировать представления
- ✓ Использовать инструкцию `ALTER VIEW` для повторного создания представления
- ✓ Проектировать и реализовывать встроенные функции

Продолжительность занятия — 20 минут.

Введение

В SQL *представления* используются для хранения и повторного использования запросов к базе данных. Представления могут использоваться практически для тех же целей, что и таблицы. Вы можете делать выборки из них и фильтровать результаты, точно так же, как это делается для таблиц. Используя представления, можно даже выставлять, обновлять или удалять строки, правда, с ограничениями.



Каждое представление определяется инструкцией `SELECT`, которая может ссылаться на несколько базовых таблиц и других представлений. Поэтому представления можно также использовать для упрощения сложных операций, необходимых для соединения нескольких таблиц вместе, облегчая пользователям и приложениям доступ к данным в базе данных. В этом занятии вы будете создавать и модифицировать представления.

Представления

Для создания представления необходимо дать ему имя и затем указать инструкцию `SELECT`, которая будет формировать представление. Например, следующая инструкция `CREATE VIEW`, создающая представление `Sales.OrderTotalsByYear`, взята из `TSQL2012.sql`.

```
USE TSQL2012;
GO
CREATE VIEW Sales.OrderTotalsByYear
    WITH SCHEMABINDING
AS
```

```
SELECT  
    YEAR(O.orderdate) AS orderyear,  
    SUM(OD.qty) AS qty  
FROM Sales.Orders AS O  
    JOIN Sales.OrderDetails AS OD  
        ON OD.orderid = O.orderid  
GROUP BY YEAR(orderdate);  
GO
```

Можно выполнять чтение из представления точно так же, как из таблицы. Поэтому выборка из него может быть сделана следующим образом:

```
SELECT orderyear, qty  
FROM Sales.OrderTotalsByYear;
```

Далее приведены некоторые замечания относительно данного представления.

- В любой инструкции CREATE, как например CREATE TABLE, можно дополнитель но указать схему базы данных как контейнер для представления. В данном случае представление создано в схеме Sales. Наиболее правильным считается при ссылке на объекты базы данных, такие как представления, использовать двух компонентные (состоящие из двух частей) имена, которые включают в себя имя схемы. (Дополнительную информацию о схемах баз данных можно найти в разд. "Определение схемы базы данных" главы 8.)
- Это представление создано с использованием параметра представления SCHEMABINDING, который гарантирует, что структуры базовых таблиц не могут быть изменены без удаления представления.
- Тело представления — это просто инструкция SELECT, с соблюдением всех обычных правил для инструкций SELECT.
- Можно добавлять новые столбцы в представлении путем создания новых столбцов в инструкции SELECT с помощью выражений.
- Можно сделать некоторые столбцы базовой таблицы невидимыми для пользователя, удалив их из инструкции SELECT, которая определяет представление.
- Можно переименовать столбцы, используя их псевдонимы, точно так же, как это делается в инструкции SELECT.

ПРИМЕЧАНИЕ Представления показывают пользователю абстрактные уровни

Основное использование представлений в реляционных базах данных, как для оперативной обработки транзакций (online transaction processing, OLTP), так и в системах хранения данных — представить уровень абстракции между конечным пользователем и базой данных. Когда базе данных требуются сложные соединения таблиц, можно упростить запросы пользователя путем внедрения этих соединений в представления. Пользователи запрашивают представления, а не таблицы, что позволяет им работать с более простым, логическим представлением базы данных, без необходимости знать их сложные физические детали.

Синтаксис представлений базы данных

Рассмотрим базовый синтаксис инструкции CREATE VIEW:

```
CREATE VIEW [ schema_name . ] view_name [ (column [ ,...n ] ) ]
[ WITH <view_attribute> [ ,...n ] ]
AS select_statement
[ WITH CHECK OPTION ] [ ; ]
```

Далее приведено детальное описание ее компонентов.

- Хотя этого нет в синтаксисе, инструкция CREATE VIEW должна быть первой инструкцией в пакете. Нельзя поставить перед ней никакие другие инструкции T-SQL, так же как нельзя сделать ее условной, поместив внутри инструкции IF.
- Представлению дается имя точно так же, как любому другому объекту базы данных (процедурам или функциям, например).
- Можно указать набор выходных столбцов после имени представления. Например, можно переписать инструкцию CREATE VIEW для представления Sales.OrderTotalsByYear и указать имена столбцов сразу после имени представления, а не в инструкции SELECT, как показано в следующем примере. Но следует отметить, что тогда при чтении инструкции SELECT труднее понять, к чему относятся столбцы в этом представлении.

```
CREATE VIEW Sales.OrderTotalsByYear(orderyear, qty)
    WITH SCHEMABINDING
AS
SELECT
    YEAR(O.orderdate),
    SUM(OD.qty)
FROM Sales.Orders AS O
    JOIN Sales.OrderDetails AS OD
        ON OD.orderid = O.orderid
GROUP BY YEAR(orderdate);
GO
```

ПРИМЕЧАНИЕ Создавайте самодокументируемые представления

Считается наиболее правильным создавать самодокументируемый код T-SQL. В общем случае, представление будет более самодокументируемым, если имена столбцов представления указаны в инструкции SELECT, а не перечислены отдельно в представлении.

Параметры представления

Можно добавить следующие параметры представления в любой комбинации.

- С помощью предложения WITH ENCRYPTION можно указать, что текст представления должен быть сохранен в неявном виде (это не строгое шифрование). Это затрудняет для пользователей раскрытие текста инструкции SELECT представления.
- Предложение WITH SCHEMABINDING, как уже говорилось ранее, привязывает представление к схемам базовых таблиц: нельзя изменить определения схем таблицы

без удаления представления. Это защищает представление от повреждения в результате изменения структур таблиц.

- Предложение `WITH VIEW_METADATA`, когда оно указано, возвращает метаданные представления, а не базовой таблицы.

Инструкции ***SELECT*** и ***UNION*** в представлении

Обратите внимание, в приведенном синтаксисе представления имеется только одна инструкция `SELECT`, и, соответственно, только одна инструкция `SELECT` разрешена в представлении. Это действительно так, поскольку главное требование — представление должно возвращать только один результирующий набор, чтобы для большинства инструкций SQL представление всегда выглядело так, как если бы это была таблица.

Однако имеется возможность объединять инструкции `SELECT`, возвращающие одинаковые результирующие наборы, с помощью предложений `UNION` или `UNION ALL` в инструкции `SELECT`. Мы рассмотрим это позже в разд. "Секционированные представления" данного занятия. Дополнительную информацию о предложении `UNION` можно найти в занятии 3 главы 4.

Предложение ***WITH CHECK OPTION***

Наконец, в представление можно добавить предложение `WITH CHECK OPTION`. Это важный параметр. Если вы определяете представление с ограничением фильтра в предложении `WHERE` инструкции `SELECT`, а затем измените строки таблицы с помощью представления, то можете изменить некоторое значение так, что задействованная строка уже не будет удовлетворять фильтру предложения `WHERE`. Возможно даже обновление строк, которые выходят за пределы области фильтра. Предложение `WITH CHECK OPTION` препятствует подобному исчезновению строк при обновлении через представление, а также ограничивает модификации только строками, которые удовлетворяют критериям фильтра. Дополнительные сведения об обновлении представлений можно найти в разд. "Модификация данных с помощью представления" далее в этом занятии.

Имена представлений

Каждое представление — это объект базы данных, и его имя действует в пределах базы данных. Поэтому для базы данных каждое имя представления в схеме базы данных должно быть уникальным. У представления не может быть одинакового сочетания имени схемы и имени объекта, как и у любых других объектов, действующих в пределах схемы в базе данных, к которым относятся следующие:

- представления;
- таблицы;
- хранимые процедуры;
- функции;
- синонимы.

Синонимы рассмотрены подробно в занятии 2 данной главы.

Имена представлений должны быть идентификаторами T-SQL, так же как имена таблиц, хранимых процедур, функций, индексов и прочих объектов базы данных SQL Server. (Идентификаторы T-SQL подробно рассмотрены в разд. "Именование таблиц и столбцов" главы 8.)

Ограничения в представлениях

Представления имеют некоторые ограничения, перечисленные далее.

- В представлении нельзя добавить предложение ORDER BY в инструкции SELECT. Представление должно выглядеть точно так же, как таблица, а таблицы в реляционных базах данных содержат наборы строк. Сами по себе наборы не упорядочены, хотя можно применить к ним сортировку в результирующем наборе с помощью предложения ORDER BY. Аналогично, строки таблиц и представлений в SQL Server не имеют логического порядка, хотя их можно упорядочить, добавив предложение ORDER BY в самой внешней инструкции SELECT при доступе к представлению.
- Представлению нельзя передавать параметры. Аналогично, представление не может ссылаться на переменную внутри инструкции SELECT. В разд. "Встроенные функции" далее в этом занятии приведены сведения о том, как использовать функции для имитации передачи параметров представлению.
- Представление не может создать таблицу, как постоянную, так и временную. Иными словами, в представлении нельзя использовать синтаксис SELECT/INTO.
- Представление может ссылаться только на постоянные таблицы; оно не может ссылаться на временную таблицу.

СОВЕТ

Подготовка к экзамену

Результаты представления никогда не упорядочены. Для выборки из представления необходимо добавить свое предложение ORDER BY. Можно включить предложение ORDER BY в представление, только добавив оператор TOP или предложение OFFSET...FETCH в предложении SELECT. Даже тогда результат представления не будет упорядочен. Поэтому присутствие предложения ORDER BY в представлении, даже если вы можете его ввести, бесполезно.

Индексированные представления

Обычно представление — это просто определение инструкцией SELECT, как должны быть построены результаты: никакие данные при этом не сохраняются. Другими словами, хранится только инструкция SELECT, и никакие другие данные.

Но можно создать уникальный кластеризованный индекс на представлении и таким образом материализовать данные. В этом случае хранится не только определение представления. На диске сохраняются фактические результаты запроса, в структуре кластеризованного индекса. Для того чтобы быть проиндексированным, представление должно отвечать некоторым важным ограничениям. Более подробно индексированные представления рассмотрены в разд. "Реализация индексированных представлений" главы 15.



Выполнение запросов из представлений

При выполнении запроса из стандартного нематериализованного представления оптимизатор запросов SQL Server объединяет ваш внешний запрос с запросом, встроенным в представление, и обрабатывает этот объединенный запрос. В результате, когда вы посмотрите на планы запросов для запросов, в которых выборка выполняется из представлений, то увидите в плане запроса ссылочные базовые таблицы представления; само представление не будет объектом в плане запроса.

Изменение представления

После создания запроса можно изменить его структуру, а также добавить или удалить свойства представления с помощью команды ALTER VIEW. Команда ALTER VIEW просто переопределяет работу представления посредством полного повторного определения представления. Например, можно переопределить представление Sales.OrderTotalsByYear с целью добавления нового столбца для региона, в который был поставлен заказ, — столбца shipregion, следующим образом:

```
ALTER VIEW Sales.OrderTotalsByYear
    WITH SCHEMABINDING
AS
SELECT O.shipregion,
    YEAR(O.orderdate) AS orderyear,
    SUM(OD.qty) AS qty
FROM Sales.Orders AS O
JOIN Sales.OrderDetails AS OD
    ON OD.orderid = O.orderid
GROUP BY YEAR(orderdate), O.shipregion;
GO
```

Теперь можно изменить способ выборки из представления, как если бы это была таблица, для включения нового столбца; при желании вы можете упорядочить результаты с помощью предложения ORDER BY следующим образом:

```
SELECT shipregion, orderyear, qty
FROM Sales.OrderTotalsByYear
ORDER BY shipregion;
```

Удаление представления

Представление удаляется так же, как таблица.

```
DROP VIEW Sales.OrderTotalsByYear;
```

Когда необходимо создать новое представление и заменить им старое, следует сначала удалить старое представление и затем создать новое. В следующем примере приведен один из способов сделать это:

```
IF OBJECT_ID('Sales.OrderTotalsByYear', 'V') IS NOT NULL
    DROP VIEW Sales.OrderTotalsByYear;
```

```
GO  
CREATE VIEW Sales.OrderTotalsByYear  
...
```

Параметр 'v' в функции OBJECT_ID() выполняет поиск представлений в текущей базе данных и затем возвращает object_id, если предложение с этим именем обнаружено.

Модификация данных с помощью представления

Можно обновлять, вставлять или удалять данные с помощью представления вместо того, чтобы напрямую ссылаться на базовые таблицы, но при этом следует учитывать множество ограничений, приведенных далее.

- DML-инструкции (INSERT, UPDATE и DELETE) должны ссылаться точно на одну таблицу за один раз, неважно, на сколько таблиц ссылается представление.
- Столбцы представления должны напрямую ссылаться на столбцы таблиц и не являться выражениями или функциями, окружающими значение столбца.

Соответственно, нельзя модифицировать столбец представления, у которого имеется статистическая функция, такая как SUM(), MAX() или MIN(), применяемая к столбцу таблицы.

- Нельзя модифицировать столбец представления, который является вычисляемым и формируется с помощью операторов UNION/UNION ALL, CROSS JOIN, EXCEPT или INTERSECT.
- Нельзя модифицировать столбец представления, значения которого получены группированием, например, с использованием предложений DISTINCT, GROUP BY и HAVING.
- Нельзя модифицировать представление, имеющее конструкции TOP или OFFSET...FETCH в инструкции SELECT, вместе с предложением WITH CHECK OPTION.

Если все-таки необходимо обновлять таблицы с помощью представления и представление не удовлетворяет всем этим требованиям, можно создать триггер INSTEAD OF на представлении и использовать этот триггер для обновления базовых таблиц. Дополнительную информацию о триггерах INSTEAD OF см. в разд. "Триггеры INSTEAD OF" главы 13.

Секционированные представления

SQL Server поддерживает использование представлений для секционирования больших таблиц на одном сервере, в одной или более таблицах, расположенныхных в нескольких базах данных или на нескольких серверах. Если у вас нет возможности использовать секционирование таблиц, вы можете разделить ваши таблицы вручную и создать представление, которое применяет инструкцию UNION к этим таблицам. Результат этого называется *секционированным представлением*. Если таблицы расположены в одной базе данных или по крайней мере на одном экземпляре SQL Server, это называется секционированным представлением или локальным секционированным представлением. Если таблицы распределены



между несколькими экземплярами SQL Server, представление называется распределенным секционированным представлением.

Что касается секционированного представления, если вы хотите, чтобы оптимизатор запросов SQL Server использовал преимущества секционирования и эффективно разрешал запросы, используя устранение секций, представление должно удовлетворять целому ряду важных условий. После того как эти условия соблюдены, можно выбирать и изменять данные с использованием представления эффективно и с поддержкой SQL-сервера. Дополнительную информацию о секционированных представлениях можно найти в электронной документации по SQL Server 2012 в разделе "Использование секционированных представлений" по адресу <http://msdn.microsoft.com/ru-ru/library/ms190019.aspx>.

Представления и метаданные

Представления были разработаны по подобию таблиц, и в действительности, когда вы используете среду SQL Server Management Studio (SSMS) и другие инструменты для детализации представления, обратите внимание, что оно представляется в виде столбцов с типами данных точно так же, как это происходит с таблицами. Чтобы обеспечить пользователям базы данных возможность видеть метаданные представления, дайте им разрешение `VIEW DEFINITION` на рассматриваемые представления.

Чтобы просмотреть метаданные представления с помощью T-SQL, можно выполнить запрос к представлению каталога `sys.views` следующим образом:

```
USE TSQL2012;
GO
SELECT name, object_id, principal_id, schema_id, type
FROM sys.views;
```

Также можно запросить системное представление `INFORMATION_SCHEMA.TABLES`, но это немного сложнее.

```
SELECT TABLE_SCHEMA, TABLE_NAME, TABLE_TYPE
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE = 'VIEW';
```

Использование представления `sys.views` более информативно, и из него можно связаться с другими представлениями каталога, такими как `sys.sql_modules`, для получения дополнительной информации.

Контрольные вопросы

1. Должно ли представление состоять только из одной инструкции `SELECT`?
2. Какие типы представлений доступны в языке T-SQL?

Ответы на контрольные вопросы

1. Технически — да, но можно обойти это ограничение с помощью объединения (используя инструкцию `UNION`) нескольких инструкций `SELECT`, которые вместе дадут один результирующий набор.

2. Можно создать стандартные представления, которые являются просто сохраненными инструкциями `SELECT`, или индексированные представления, которые фактически материализуют данные, в дополнение к секционированным представлениям.

Встроенные функции

В языке T-SQL единственный способ отфильтровать представление — добавить фильтр в предложение `WHERE`, когда выполняется выборка из представления. Не существует возможности передать представлению параметр, чтобы отфильтровать строки. Но можно использовать *встроенную функцию*, возвращающую табличное значение (табличную функцию) для имитации передачи параметра представлению, или, другими словами, имитировать параметризованное представление.

Встроенная функция, возвращающая табличное значение, возвращает набор данных с использованием инструкции `SELECT`, которую вставляют в код функции. По сути, вы используете табличную функцию как таблицу и делаете выборку из нее с помощью инструкции `SELECT FROM`. Например, можно создать встроенную функцию, которая действовала бы так же, как представление `Sales.OrderTotalsByYear`, без параметров, следующим образом:

```
USE TSQL2012;
GO
IF OBJECT_ID ('N'Sales.fn_OrderTotalsByYear', N'IF') IS NOT NULL
    DROP FUNCTION Sales.fn_OrderTotalsByYear;
GO
CREATE FUNCTION Sales.fn_OrderTotalsByYear ()
RETURNS TABLE
AS
RETURN
( SELECT
    YEAR(O.orderdate) AS orderyear,
    SUM(OD.qty) AS qty
FROM Sales.Orders AS O
JOIN Sales.OrderDetails AS OD
    ON OD.orderid = O.orderid
GROUP BY YEAR(orderdate)
);
GO
```



Для создания встроенной табличной функции необходимо выполнить следующие действия.

- Указать параметры. Параметры являются необязательными, но круглые скобки, в которые они должны быть заключены, обязательны.
- Добавить предложение `RETURNS TABLE`, чтобы проинформировать SQL Server, что это табличная функция.

- После блока AS ввести одну инструкцию RETURN. Она действует как встроенная функция для возвращения внедренной инструкции SELECT.
- Внедрить инструкцию SELECT, которая будет определять, что функция должна возвратить в виде набора строк вызывающей стороны.
- Точка с запятой, стоящая после последних круглых скобок, не обязательна, но если она есть, она должна стоять после закрывающей круглой скобки.

Во встроенной функции, возвращающей табличное значение, телом функции может быть только инструкция SELECT; вы не можете объявлять переменные и выполнять другие команды T-SQL, как это можно делать со скалярными определяемыми пользователем функциями (UDF) и многооператорными возвращающими табличное значение функциями. (Дополнительную информацию об определенных пользователем функциях T-SQL в SQL Server можно найти в *занятии 3 главы 13*.)

В предыдущем примере инструкция SELECT была такой же сложной, как исходное представление Sales.OrderTotalsByYear. Если некоторые столбцы таблицы не нужны, можно в действительности упростить функцию, выполняя выборку напрямую из представления.

```
USE TSQL2012;
GO
IF OBJECT_ID (N'Sales.fn_OrderTotalsByYear', N'IF') IS NOT NULL
    DROP FUNCTION Sales.fn_OrderTotalsByYear;
GO
CREATE FUNCTION Sales.fn_OrderTotalsByYear()
RETURNS TABLE
AS
RETURN
    (SELECT orderyear, qty FROM Sales.OrderTotalsByYear);
GO
```

Если бы вам было нужно видеть 2007 год, это просто нужно было указать это в предложении WHERE при выборке из представления.

```
SELECT orderyear, qty
FROM Sales.OrderTotalsByYear
WHERE orderyear = 2007;
```

Чтобы сделать предложение WHERE более гибким, можно объявить переменную и затем выполнить фильтрацию с использованием этой переменной.

```
DECLARE @orderyear int = 2007;
SELECT orderyear, qty
FROM Sales.OrderTotalsByYear
WHERE orderyear = @orderyear;
```

Принимая все это во внимание, теперь до встроенной функции остается только один шаг. Вместо объявления переменной @orderyear определите параметр @orderyear в функции при фильтрации инструкции SELECT таким же образом, как в предыдущем случае.

```
USE TSQSL2012;
GO
IF OBJECT_ID (N'Sales.fn_OrderTotalsByYear', N'IF') IS NOT NULL
    DROP FUNCTION Sales.fn_OrderTotalsByYear;
GO
CREATE FUNCTION Sales.fn_OrderTotalsByYear (@orderyear int)
RETURNS TABLE
AS
RETURN
( SELECT orderyear, qty FROM Sales.OrderTotalsByYear
  WHERE orderyear = @orderyear);
GO
```

Вы можете запросить функцию с передачей года, который хотите видеть, следующим образом:

```
SELECT orderyear, qty FROM Sales.fn_OrderTotalsByYear(2007);
```

Вы сейчас создали параметризованное представление с использованием встроенной функции. Встроенная функция является более гибкой, чем представление, поскольку она возвращает результаты с учетом значения переданного параметра. Нет необходимости добавлять дополнительное предложение WHERE.

Параметры встроенной функции

Встроенные функции имеют два важных параметра, таких же, как и у представления.

- Можно создать функцию с использованием предложения WITH ENCRYPTION, затруднив для пользователя раскрытие текста SELECT для функции.
- Можно добавить предложение WITH SCHEMABINDING, которое привязывает схемы таблицы базовых объектов, таких как таблицы или представления, к функции. Схемы объектов, на которые выполняется ссылка, не могут изменяться, пока функция не будет удалена или не будет удален параметр WITH SCHEMABINDING.

Контрольные вопросы

1. Какой тип данных возвращает встроенная функция?
2. Какой тип представления может имитировать встроенная функция?

Ответы на контрольные вопросы

1. Встроенные функции возвращают таблицы и, соответственно, часто называются встроенными функциями, возвращающими табличное значение.
2. Встроенная функция, возвращающая табличное значение, может имитировать параметризованное представление, т. е. представление, которое принимает параметры.

ПРАКТИКУМ Работа с представлениями и встроенными функциями

В этом практикуме вы проверите ваше понимание представлений T-SQL и используете встроенную функцию для имитации параметризованного представления.

Задание 1. Построение представления для отчета

Рассмотрим следующий сценарий: вас попросили разработать интерфейс для отчета из базы данных TSQL2012. Приложению нужно представление, которое показывает проданное количество и общий объем продаж для всех продаж, по году, по клиенту и по грузоотправителю. Пользователь также хотел бы иметь возможность фильтровать результаты по верхнему и нижнему значениям общего суммарного количества.

1. Начните с текущего представления `Sales.OrderTotalsByYear`, ранее использовавшегося в этом занятии. Введите инструкцию `SELECT` без определения представления.

```
USE TSQL2012;
GO
SELECT YEAR(O.orderdate) AS orderyear, SUM(OD.qty) AS qty
FROM Sales.Orders AS O
JOIN Sales.OrderDetails AS OD
    ON OD.orderid = O.orderid
GROUP BY YEAR(orderdate);
```

Обратите внимание, представление `Sales.OrderValues` содержит вычисленную сумму продаж.

```
CAST(SUM(OD.qty * OD.unitprice * (1 - OD.discount)) )
AS NUMERIC(12, 2)) AS val
```

2. Объедините эти два запроса.

```
SELECT YEAR(O.orderdate) AS orderyear,
       SUM(OD.qty) AS qty,
       CAST(SUM(OD.qty * OD.unitprice * (1 - OD.discount)) )
AS NUMERIC(12, 2)) AS val
FROM Sales.Orders AS O
JOIN Sales.OrderDetails AS OD
    ON OD.orderid = O.orderid
GROUP BY YEAR(orderdate);
```

3. Добавьте столбцы для значения `custid` для возвращения ID клиента и `ID грузоотправителя`. Заметьте, вы должны теперь изменить предложение `GROUP BY`, чтобы предоставить эти два идентификатора.

```
SELECT O.custid, O.shipperid,
       YEAR(O.orderdate) AS orderyear,
       SUM(OD.qty) AS qty,
       CAST(SUM(OD.qty * OD.unitprice * (1 - OD.discount)) )
```

```

AS NUMERIC(12, 2)) AS val
FROM Sales.Orders AS O
JOIN Sales.OrderDetails AS OD
    ON OD.orderid = O.orderid
GROUP BY YEAR(O.orderdate), O.custid, O.shipperid;

```

4. Пока что все идет хорошо, но вам нужно в результатах показать имена грузоотправителя и клиента для отчета. Поэтому необходимо добавить инструкции JOIN в таблицу Sales.Customers и в таблицу Sales.Shippers.

```

SELECT YEAR(O.orderdate) AS orderyear,
       SUM(OD.qty) AS qty,
       CAST(SUM(OD.qty * OD.unitprice * (1 - OD.discount)) )
AS NUMERIC(12, 2)) AS val
FROM Sales.Orders AS O
JOIN Sales.OrderDetails AS OD
    ON OD.orderid = O.orderid
JOIN Sales.Customers AS C
    ON O.custid = C.custid
JOIN Sales.Shippers AS S
    ON O.shipperid = S.shipperid
GROUP BY YEAR(O.orderdate);

```

5. Добавьте название компании клиента (`companyname`) и название компании-грузоотправителя (`companyname`). Вы должны развернуть предложение GROUP BY, чтобы представить эти столбцы.

```

SELECT C.companyname AS customercompany,
       S.companyname AS shippercompany,
       YEAR(O.orderdate) AS orderyear,
       SUM(OD.qty) AS qty,
       CAST(SUM(OD.qty * OD.unitprice * (1 - OD.discount)) )
AS NUMERIC(12, 2)) AS val
FROM Sales.Orders AS O
JOIN Sales.OrderDetails AS OD
    ON OD.orderid = O.orderid
JOIN Sales.Customers AS C
    ON O.custid = C.custid
JOIN Sales.Shippers AS S
    ON O.shipperid = S.shipperid
GROUP BY YEAR(O.orderdate), C.companyname, S.companyname;

```

6. Преобразуйте его в представление с именем `Sales.OrderTotalsByYearCustShip`.

```

IF OBJECT_ID (N'Sales.OrderTotalsByYearCustShip', N'V') IS NOT NULL
DROP VIEW Sales.OrderTotalsByYearCustShip;
GO
CREATE VIEW Sales.OrderTotalsByYearCustShip
    WITH SCHEMABINDING
AS

```

```

SELECT C.companyname AS customercompany,
       S.companyname AS shippercompany,
       YEAR(O.orderdate) AS orderyear,
       SUM(OD.qty) AS qty,
       CAST(SUM(OD.qty * OD.unitprice * (1 - OD.discount)) )
AS NUMERIC(12, 2)) AS val
FROM Sales.Orders AS O
JOIN Sales.OrderDetails AS OD
  ON OD.orderid = O.orderid
JOIN Sales.Customers AS C
  ON O.custid = C.custid
JOIN Sales.Shippers AS S
  ON O.shipperid = S.shipperid
GROUP BY YEAR(O.orderdate), C.companyname, S.companyname;
GO

```

7. Протестируйте представление, выполнив выборку из него.

```

SELECT customercompany, shippercompany, orderyear, qty, val
FROM Sales.OrderTotalsByYearCustShip
ORDER BY customercompany, shippercompany, orderyear;

```

8. Удалите представление для очистки данных.

```

IF OBJECT_ID(' Sales.OrderTotalsByYearCustShip', N'V') IS NOT NULL
DROP VIEW Sales.OrderTotalsByYearCustShip

```

Задание 2. Преобразование представления во встроенную функцию

В этом задании вы выполните преобразование представления из предыдущего задания во встроенную функцию.

1. Преобразуйте представление во встроенную функцию, которая выполняет фильтрацию по нижней и верхней величинам общего количества. Добавьте два параметра с именами @highqty и @lowqty, оба целые числа, и добавьте предложение HAVING для фильтрации результатов. Дайте функции имя Sales.fn_OrderTotalsByYearCustShip.

```

IF OBJECT_ID(N'Sales.fn_OrderTotalsByYearCustShip', N'IF') IS NOT NULL
DROP FUNCTION Sales.fn_OrderTotalsByYearCustShip;
GO
CREATE FUNCTION Sales.fn_OrderTotalsByYearCustShip (@lowqty int, @highqty
int)
RETURNS TABLE
AS
RETURN
(SELECT C.companyname AS customercompany,
       S.companyname AS shippercompany,
       YEAR(O.orderdate) AS orderyear,

```

```

        SUM(OD.qty) AS qty,
        CAST(SUM(OD.qty * OD.unitprice * (1 - OD.discount)) )
        AS NUMERIC(12, 2)) AS val
    FROM Sales.Orders AS O
    JOIN Sales.OrderDetails AS OD
        ON OD.orderid = O.orderid
    JOIN Sales.Customers AS C
        ON O.custid = C.custid
    JOIN Sales.Shippers AS S
        ON O.shipperid = S.shipperid
    GROUP BY YEAR(O.orderdate), C.companyname, S.companyname
    HAVING SUM(OD.qty) >= @lowqty AND SUM(OD.qty) <= @highqty
);
GO

```

2. Протестируйте функцию.

```

SELECT customercompany, shippercompany, orderyear, qty, val
FROM Sales.fn_OrderTotalsByYearCustShip (100, 200)
ORDER BY customercompany, shippercompany, orderyear;

```

Поэкспериментируйте с другими значениями, пока не убедитесь, что хорошо понимаете, как работает функция и ее фильтрация.

3. Удалите представление и функцию для очистки данных.

```

IF OBJECT_ID (N'Sales.OrderTotalsByYearCustShip', N'V') IS NOT NULL
    DROP VIEW Sales.OrderTotalsByYearCustShip;
GO
IF OBJECT_ID (N'Sales.fn_OrderTotalsByYearCustShip', N'IF') IS NOT NULL
    DROP FUNCTION Sales.fn_OrderTotalsByYearCustShip;
GO

```

Резюме занятия

- Представления — это сохраненные инструкции SELECT языка T-SQL, которые могут обрабатываться, как если бы это были таблицы.
- Обычно представление состоит только из одной инструкции SELECT, но можно обойти это правило, объединив инструкции SELECT со сходными результатами, используя инструкций UNION или UNION ALL.
- Представления могут ссылаться на несколько таблиц и упростить сложные объединения для пользователей.
- По умолчанию представления не содержат никаких данных. Создание уникального кластеризованного индекса на представлении дает в результате индексированное представление, которое материализует данные.
- При выполнении выборки из представления SQL Server берет самую внешнюю инструкцию SELECT и объединяет ее с инструкцией SELECT из определения пред-

ставления. Затем SQL Server выполняет эту комбинированную инструкцию SELECT.

- Можно модифицировать данные с помощью представления, но только для одной таблицы за один раз и лишь в столбцах определенных типов.
- Можно добавить к представлению предложение WITH CHECK OPTION для того, чтобы предотвратить любые изменения через представление, которые могут привести к получению в строках значений, не удовлетворяющих предложению WHERE данного представления.
- Представления могут ссылаться на таблицы или представления в других базах данных и на других серверах через связанные серверы.
- Специальные представления, называемые секционированными представлениями, могут быть созданы при условии удовлетворения определенному набору требований, и если SQL Server выполняет маршрутизацию соответствующих запросов и обновлений на нужную секцию представления.
- Встроенные функции можно использовать для имитации параметризованных представлений. Представления языка T-SQL не могут принимать параметры. Но встроенная функция, возвращающая табличное значение, может возвращать те же данные, что и представление, а также принимать параметры, которые могут фильтровать результаты.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какие из перечисленных операторов работают в представлениях T-SQL? (Выберите все подходящие варианты.)
 - A. Предложение WHERE.
 - B. Предложение ORDER BY.
 - C. Операторы UNION или UNION ALL.
 - D. Предложение GROUP BY.
2. Каков результат выражения WITH SCHEMABINDING в представлении?
 - A. Представление не может быть изменено без изменения таблицы.
 - B. Таблица, на которую делается ссылка в представлении, не может быть изменена, пока не будет изменена инструкция SELECT представления.
 - C. Таблица, на которую делается ссылка в представлении, не может быть изменена, пока не будет удалено представление.
 - D. Представление не может быть изменено, пока не будет удалена таблица, на которую оно ссылается.

3. Что является результатом предложения WITH CHECK OPTION в представлении, которое имеет предложение WHERE в своей инструкции SELECT?
- A. Данные больше не могут обновляться через представление.
 - B. Данные могут обновляться через представление, но значения первичного ключа не могут быть изменены.
 - C. Данные могут обновляться через представление, но значения изменяться не могут, что приведет к попаданию строк вне фильтра в предложении WHERE.
 - D. Данные могут обновляться через представление, но только столбцы с проверочными ограничениями могут быть изменены.

Занятие 2. Использование синонимов

Кроме представлений, которые могут предоставлять уровень абстракции для таблиц базы данных, в SQL Server имеются синонимы, которые способны обеспечить уровень абстракции для всех действующих в пределах схемы объектов базы данных. Синонимы — это имена, сохраняемые в базе данных, которые могут использоваться для замещения имен других объектов. Эти имена также действуют в пределах базы данных и добавляются к имени схемы.



Изучив материал этого занятия, вы сможете:

- ✓ Создавать и удалять синонимы
- ✓ Понимать использование синонимов как уровня абстракции
- ✓ Понимать сходство и различие между синонимами и другими объектами баз данных

Продолжительность занятия — 15 минут.

Создание синонима

Для создания синонима нужно просто дать ему имя и указать имя объекта базы данных, которому оно будет назначено. Например, можно определить синоним Categories и поместить его в схему dbo так, что пользователям не нужно будет помнить в своих запросах имя Production.Categories в формате "схема.объект". Можно выполнить следующий код:

```
USE TSQL2012;
GO
CREATE SYNONYM dbo.Categories FOR Production.Categories;
GO
```

Затем конечный пользователь может делать выборку из Categories без необходимости указывать схему.

```
SELECT categoryid, categoryname, description FROM Categories;
```

Базовый синтаксис для создания синонима весьма прост.

```
CREATE SYNONYM schema_name.synonym_name FOR object_name
```

Имя синонима — это объект базы данных, и оно должно соответствовать следующим правилам, принятым для идентификатора T-SQL.

- Синонимы не хранят никаких данных или кода T-SQL.
- Имена синонимов должны быть идентификаторами T-SQL, как и все прочие объекты баз данных. (Дополнительную информацию об идентификаторах T-SQL см. в разд. "Именование таблиц и столбцов" главы 8.)
- Если не указано имя схемы, SQL Server будет использовать схему по умолчанию, соответствующую вашему имени пользователя.
- Элемент *object_name* в действительности может не указываться, и SQL Server не тестирует его. Это происходит из-за свойства позднего связывания синонимов, рассмотренного далее в этом занятии.
- SQL Server проверит существование объекта, только когда вы начнете использовать синоним в инструкции T-SQL.

Синонимы могут применяться для следующих типов объектов:

- таблицы (включая временные таблицы);
- представления;
- пользовательские функции (скалярные, табличные, встроенные);
- хранимые процедуры (T-SQL, расширенные хранимые процедуры и процедуры фильтра репликации);
- сборки среды CLR (хранимые процедуры, табличные, скалярные и статистические функции).

Дополнительную информацию о типах объектов, для которых могут использоваться синонимы, можно найти в электронной документации по SQL Server 2012 в разделе "CREATE SYNONYM (Transact-SQL)" по адресу <http://msdn.microsoft.com/ru-ru/library/ms177544.aspx>.

COBET

Подготовка к экзамену

Синонимы не могут ссылаться на другие синонимы. Они могут ссылаться только на такие объекты базы данных, как таблицы, представления, хранимые процедуры и функции. Другими словами, создание цепочки синонимов не разрешается.

Можно использовать синонимы в инструкциях T-SQL, ссылающихся на типы объектов, которые синонимы могут обозначать. В дополнение к инструкции EXECUTE для хранимых процедур, можно использовать инструкции манипулирования данными: SELECT, INSERT, UPDATE и DELETE.

ПРИМЕЧАНИЕ Использование инструкции ALTER С СИНОНИМАМИ

Нельзя ссылаться на синоним в инструкции DDL, такой как ALTER. Эти инструкции требуют ссылки на базовый объект.

Удаление синонима

Можно удалить синоним с помощью инструкции DROP SYNONYM.

```
DROP SYNONYM dbo.Categories
```

Поскольку не существует инструкции ALTER SYNONYM, для изменения синонима надо сначала удалить его и затем создать заново.

Уровень абстракции

Синонимы могут ссылаться на объекты в других базах данных, а также на объекты, на которые ссылаются связанные серверы. Это позволяет значительно упростить запросы в базе данных и теоретически избавиться от необходимости в ссылках, состоящих из трех и четырех частей.

Например, предположим, что база данных ReportsDB имеет представление с именем Sales.Reports и находится на том же сервере, что база данных TSQL2012. Чтобы сделать запрос к ней из TSQL2012, надо написать что-то подобное следующему запросу:

```
SELECT report_id, report_name FROM ReportsDB.Sales.Reports
```

Далее предположим, что мы добавили синоним, названный просто Sales.Reports.

```
CREATE SYNONYM Sales.Reports FOR ReportsDB.Sales.Reports
```

Тогда запрос будет упрощен до следующего вида:

```
SELECT report_id, report_name FROM Sales.Reports
```

Теперь пользователю не нужно помнить имя другой базы данных. Это свойство оказывается еще более полезным, когда используются связанные серверы. Вы можете упростить четырехкомпонентные имена до двухкомпонентных и даже до состоящих всего из одной части, тем самым значительно упрощая жизнь конечному пользователю.

Синонимы и ссылки на несуществующие объекты

Можно создать синоним, даже если объект, на который выполняется ссылка, не существует. Преимущество этого состоит в том, что можно использовать один синоним для множества различных объектов, просто заново создавая синоним для каждого объекта по мере необходимости. Или вы можете создать тот же самый синоним в нескольких базах данных, которые ссылаются на один объект, и не нужно будет использовать трехкомпонентную ссылку.

Недостатком этого свойства является то, что не существует такого компонента, как WITH SCHEMABINDING: если вы удаляете объект в базе данных, он будет удален, независимо от того, ссылается на него синоним или нет. Любые синонимы, ссылающиеся на объект, фактически являются потерянными объектами; они прекращают работать, когда кто-то пытается их использовать.

Разрешения для работы с синонимами

Для создания синонима необходимо иметь разрешение CREATE SYNONYM, которое наследуется из разрешения CONTROL SERVER. После того как синоним создан, можно дать другим пользователям разрешения на этот синоним, такие как EXECUTE или SELECT, в зависимости от объекта, который обозначает синоним.

Сравнение синонимов с другими объектами баз данных

Синонимы необычны в том смысле, что хотя технически они являются объектами базы данных, принадлежащими тому же пространству имен, что и все прочие объекты базы данных, они не содержат никаких данных и никакой код. Интересно сравнить синонимы с другими объектами базы данных.

Далее приведены преимущества синонимов перед представлениями.

- В отличие от представлений, синонимы могут замещать многие другие типы объектов базы данных, не только таблицы.
- Также как и представления, синонимы могут предоставлять уровень абстракции, давая возможность создать логическое представление системы без необходимости показывать физические имена объектов базы данных конечному пользователю. Если базовый объект изменен, его синоним от этого не пострадает.

К недостаткам синонимов относятся следующие.

- В отличие от представлений, синонимы не способны упростить сложную логику, как, например, представление может упростить сложные соединения. Синонимы в действительности — это просто имена.
- Представление может ссылаться на множество таблиц, а синоним может ссылаться только на один объект.
- Представление может ссылаться на другое представление, тогда как синоним не может ссылаться на другой синоним; создание цепочки синонимов недопустимо.

Когда представление замещает таблицу, пользователь может видеть столбцы и типы данных представления. Но синоним не отображает метаданные базовой таблицы или представления, которые он замещает. Это может быть как преимуществом, так и недостатком, в зависимости от контекста.

- Если вы не хотите предоставлять данные пользователю, это можно рассматривать как преимущество. В среде SSMS, когда пользователь открывает дерево, чтобы взглянуть на синоним, этот пользователь не увидит столбцов или типов данных, если синоним ссылается на таблицу или представление, а также пользователь не увидит никаких параметров, если синоним ссылается на процедуру или функцию.
- Если вы хотите предоставлять метаданные пользователю как часть его обучения, тогда синоним может быть недостатком. Например, пользователю могла бы понадобиться внешняя документация, чтобы выяснить, какие столбцы доступны.

Во многих отношениях синонимы ведут себя точно так же, как прочие объекты базы данных, такие как таблицы, представления и объекты кода T-SQL. Например, можно использовать синоним в инструкциях SELECT вместо имен таблиц, имен представлений и имен встроенных функций, а также вы можете присваивать те же самые наборы разрешений синонимам, что и таблицам или представлениям.

КОНТРОЛЬНЫЕ ВОПРОСЫ

- Хранит ли синоним код T-SQL или какие-либо данные?
- Может ли синоним быть изменен?

Ответы на контрольные вопросы

- Нет, синоним — это всего лишь имя. Единственное, что сохраняется с синонимом, — это объект, на который нужно ссылаться.
- Нет, для изменения синонима его нужно удалить и создать заново.

ПРАКТИКУМ Использование синонимов

В этом практикуме вы будете использовать полученные знания о синонимах для создания пользовательского интерфейса и формирования отчетов.

Задание 1. Использование синонимов для создания более описательных имен отчетов

В этом задании вам нужно разработать пользовательский интерфейс для создания отчетов в базе данных TSQL.

Представьте себе такой сценарий: система TSQL2012 находится в промышленной эксплуатации уже некоторое время, и к вам обратились с просьбой предоставить доступ к базе данных новому приложению подготовки отчетов. Но существующие имена представлений не настолько описательны, как хотелось бы пользователям отчетов, поэтому вы будете применять синонимы для того, чтобы сделать их более описательными.

- Запустите базу данных TSQL.

```
USE TSQL2012;
GO
```

- Создайте специальную схему для отчетов.

```
CREATE SCHEMA Reports AUTHORIZATION dbo;
GO
```

- Создайте синоним для представления Sales.CustOrders в базе данных TSQL2012. Посмотрите на данные.

```
SELECT custid, ordermonth, qty FROM Sales.CustOrders;
```

Вы определили, что данные в действительности показывают ID клиента, затем итог по столбцу qty, по месяцам. Поэтому создайте синоним TotalCustQtyByMonth и протестируйте его.

```
CREATE SYNONYM Reports.TotalCustQtyByMonth FOR Sales.CustOrders;
SELECT custid, ordermonth, qty FROM Reports.TotalCustQtyByMonth;
```

- Создайте синоним для представления Sales.EmpOrders, сначала изучив данные.

```
SELECT empid, ordermonth, qty, val, numorders FROM Sales.EmpOrders;
```

Данные показывают ID сотрудника, затем итог по столбцам qty и val, по месяцам. Теперь создайте синоним TotalEmpQtyValOrdersByMonth и протестируйте его.

```
CREATE SYNONYM Reports.TotalEmpQtyValOrdersByMonth FOR Sales.EmpOrders;
SELECT empid, ordermonth, qty, val, numorders FROM
    Reports.TotalEmpQtyValOrdersByMonth;
```

5. Проверьте данные для Sales.OrderTotalsByYear.

```
SELECT orderyear, qty FROM Sales.OrderTotalsByYear;
```

Это представление показывает итог для qty по годам, поэтому назовите синоним TotalQtyByYear.

```
CREATE SYNONYM Reports.TotalQtyByYear FOR Sales.OrderTotalsByYear;
SELECT orderyear, qty FROM Reports.TotalQtyByYear;
```

6. Проверьте данные для Sales.OrderValues.

```
SELECT orderid, custid, empid, shipperid, orderdate, requireddate,
       shippeddate, qty, val
FROM Sales.OrderValues;
```

Это представление показывает итог по val и qty для каждого заказа, поэтому назовите синоним TotalQtyValOrders.

```
CREATE SYNONYM Reports.TotalQtyValOrders FOR Sales.OrderValues;
SELECT orderid, custid, empid, shipperid, orderdate, requireddate,
       shippeddate, qty, val
FROM Reports.TotalQtyValOrders;
```

Обратите внимание, комбинация столбцов не имеет уникального ключа, за исключением столбца orderid, в предложении GROUP BY представления Sales.OrderValues. Сейчас количество сгруппированных строк также является количеством заказов, но это не гарантировано. Вы должны ответить команде разработчиков, что если этот набор столбцов действительно определяет уникальную строку в таблице, им нужно создать ограничение уникальности (или уникальный индекс) на таблице.

7. Теперь внимательно просмотрите метаданные синонимов. Обратите внимание, вы можете использовать функцию SCHEMA_NAME(), чтобы получить имя схемы, не прибегая к необходимости соединяться с таблицей sys.schemas.

```
SELECT name, object_id, principal_id, schema_id, parent_object_id FROM
    sys.synonyms;
SELECT SCHEMA_NAME(schema_id) AS schemaname, name, object_id,
       principal_id, schema_id, parent_object_id
FROM sys.synonyms;
```

8. Теперь при желании можно очистить базу данных TSQL и удалить ваши данные.

```
DROP SYNONYM Reports.TotalCustQtyByMonth;
DROP SYNONYM Reports.TotalEmpQtyValOrdersByMonth;
DROP SYNONYM Reports.TotalQtyByYear;
DROP SYNONYM Reports.TotalQtyValOrders;
```

```
GO  
DROP SCHEMA Reports;  
GO
```

Задание 2. Использование синонимов для упрощения межбазовых запросов

В этом задании вам нужно показать, как отчеты могут запускаться из базы данных отчетов с помощью синонимов, которые ссылаются на другую базу данных.

Рассмотрите следующий сценарий. Вы хотите показать разработчикам отчетов, что они могут запускать свои отчеты из специально выделенной базы данных отчетов на сервере, не запрашивая напрямую главную базу данных TSQL2012. Вы решили использовать синонимы для прототипирования этой стратегии.

1. Создайте новую базу данных отчетов TSQL2012Reports:

```
USE master;  
GO  
CREATE DATABASE TSQL2012Reports;  
GO
```

2. В базе данных отчетов создайте схему с именем Reports.

```
USE TSQL2012Reports;  
GO  
CREATE SCHEMA Reports AUTHORIZATION dbo;  
GO
```

3. В качестве начальной проверки создайте синоним TotalCustQtyByMonth для несуществующего локального объекта Sales.CustOrders и протестируйте его.

```
CREATE SYNONYM Reports.TotalCustQtyByMonth FOR Sales.CustOrders;  
GO  
SELECT custid, ordermonth, qty FROM Reports.TotalCustQtyByMonth;  
-- Ошибка выполнения
```

```
GO  
DROP SYNONYM Reports.TotalCustQtyByMonth;  
GO
```

4. Создайте синоним TotalCustQtyByMonth, ссылающийся на представление Sales.CustOrders в базе данных TSQL2012, и протестируйте его.

```
CREATE SYNONYM Reports.TotalCustQtyByMonth  
FOR TSQL2012.Sales.CustOrders;  
GO  
SELECT custid, ordermonth, qty FROM Reports.TotalCustQtyByMonth;  
-- Выполнено успешно  
GO
```

5. После демонстрации команде разработки отчетов работающего сценария очистите и удалите базу данных.

```
DROP SYNONYM Reports.TotalCustQtyByMonth;
GO
DROP SCHEMA Reports;
GO
USE Master; GO
DROP DATABASE TSQL2012Reports;
GO
```

Резюме занятия

- Синоним — это имя, которое ссылается на другие объекты базы данных, такие как таблица, представление, функция или хранимая процедура.
- Синоним не хранит никакого кода T-SQL и никаких данных. Он хранит только объект, на который выполняется ссылка.
- Синонимы находятся в области действия базы данных и поэтому существуют в том же пространстве имен, что и объекты, на которые они ссылаются. Следовательно, синониму нельзя дать то же имя, что и другому объекту базы данных.
- Цепочки синонимов не разрешены; синоним не может ссылаться на другой синоним.
- Синонимы не предоставляют никаких метаданных объектов, на которые они ссылаются.
- Синонимы могут использоваться для предоставления уровня абстракции пользователю с помощью различных имен для объектов баз данных.
- Можно модифицировать данные с помощью синонима, но нельзя изменить базовый объект.
- Чтобы изменить синоним, его надо сначала удалить, а затем снова создать.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какие типы объектов баз данных могут иметь синонимы? (Выберите все подходящие варианты.)
 - A. Хранимые процедуры.
 - B. Индексы.
 - C. Временные таблицы.
 - D. Пользователи базы данных.
2. Какие высказывания из перечисленных ниже справедливы для синонимов? (Выберите все подходящие варианты.)
 - A. Синонимы не хранят код T-SQL или данные.
 - B. Синонимам не требуются имена схем.
 - C. Имена синонимов могут сопоставлять те объекты, на которые они ссылаются.

- D. Синонимы могут ссылаться на объекты в других базах данных или через связанные сервера.
3. Каковы зависимости синонимов и других объектов, на которые они ссылаются?
- A. Синонимы могут быть созданы с использованием предложения WITH SCHEMABINDING, чтобы предотвратить изменение базовых объектов.
- B. Синонимы могут иметь ссылки на другие синонимы.
- C. Синонимы могут быть созданы, чтобы ссылаться на объекты базы данных, которые еще не существуют.
- D. Синонимы могут быть изначально созданы без имени схемы, которое можно добавить позже.

Упражнения

В следующих упражнениях вы примените полученные знания о таблицах SQL Server и целостности данных. Ответы на вопросы можно найти в *приложении "Ответы" в конце книги*.

Упражнение 1. Сравнение представлений, встроенных функций и синонимов

Как ведущему разработчику баз данных на новом проекте, вам нужно обеспечить логическое представление базы данных приложениям, которые создают ежедневные отчеты. Ваша задача — подготовить для команды администраторов баз данных отчет, показывающий преимущества и недостатки представлений, встроенных функций и синонимов с точки зрения создания логического представления базы данных. Что бы вы порекомендовали использовать с учетом перечисленных далее условий: представления, встроенные функции или синонимы?

- Разработчики приложения не хотят работать со сложными соединениями при подготовке отчетов. Для обновления данных они предпочитают использовать хранимые процедуры.
- В некоторых случаях вам нужно иметь возможность изменять имена таблиц или представлений без перекодирования приложения.
- В других случаях приложению требуется фильтровать данные отчетов в базе данных с помощью передачи параметров, но разработчики не хотят использовать хранимые процедуры для извлечения этих данных.

Упражнение 2. Преобразование синонимов в другие объекты

Вас только что назначили разработчиком базы данных, которая широко использует синонимы вместо таблиц и представлений. На основании обратной связи от пользователей вам необходимо заменить некоторые синонимы. Какие действия вы мо-

жете предпринять, которые не заставят пользователей или приложения менять код в перечисленных далее случаях?

1. Некоторые синонимы ссылаются на таблицы. Но некоторые таблицы должны фильтроваться. Вам нужно оставить синонимы, но каким-то образом фильтровать возвращаемые таблицами данные.
2. Некоторые синонимы ссылаются на таблицы. Иногда имена столбцов в таблицах могут меняться, но синоним все равно должен возвратить старые имена столбцов.
3. Некоторые синонимы ссылаются на представления. Вам нужно организовать для пользователей возможность видеть имена и типы данных столбцов, возвращаемых представлениями, когда пользователи просматривают базу данных с помощью SSMS.

Рекомендуемые упражнения

Выполните следующие задачи, которые помогут вам успешно справиться с заданиями экзамена, представленными в данной главе.

Проектирование и создание представлений, встроенных функций и синонимов

Следующие практические задания расширяют код, с которым вы работали в занятиях и упражнениях данной главы. Продолжите их разработку в базе данных TSQL2012.

- **Задание 1.** Создайте простое представление для таблицы HR.Employees, включая фильтр WHERE country = 'USA' и предложение WITH CHECK OPTION. Вставьте новую строку с названием страны 'Canada'. Затем попытайтесь изменить значение страны для одного из сотрудников с USA на Canada. Работают эти обновления? Затем повторно создайте представление, но не включайте в него предложение WITH CHECK OPTION. Примените изменения. Теперь они работают? Можете объяснить, почему?
- **Задание 2.** Исследуйте создание синонимов для хранимых процедур и функций. Создайте хранимую процедуру, которая вставляет данные в таблицу Production.Categories с параметрами, предоставляющими значения для новой строки. Затем создайте синоним для этой хранимой процедуры. Выполните синоним с параметрами и убедитесь в успешности его работы. Теперь выполните синоним без этих параметров. Какое сообщение об ошибке вы получили? Относится ли оно к синониму или к хранимой процедуре? Вы понимаете, почему?

ГЛАВА 10

Вставка, обновление и удаление данных

Темы экзамена

- Модификация данных.
 - Модификация данных с помощью инструкций `INSERT`, `UPDATE` и `DELETE`.

Эта глава посвящена вопросам модификации данных. В ней описано, как вставлять, обновлять и удалять данные с помощью различных инструкций T-SQL. Эта тема продолжена в *главе 11*, в которой рассматриваются более специфичные аспекты модификации данных.

ПРЕЖДЕ ВСЕГО

Для полного освоения этой главы необходимо иметь:

- опыт работы с SQL Server Management Studio (SSMS);
- некоторый опыт в написании кода на языке T-SQL;
- доступ к экземпляру SQL Server 2012 с установленной учебной базой данных TSQl2012;
- понимание принципов фильтрации и сортировки данных;
- понимание процесса создания таблиц и обеспечения целостности данных.

Занятие 1. Вставка данных

Язык T-SQL поддерживает несколько различных методов, которые можно использовать для вставки данных в базу данных. К ним относятся такие инструкции, как `INSERT VALUES`, `INSERT SELECT`, `INSERT EXEC` и `SELECT INTO`. В этом занятии мы рассмотрим перечисленные инструкции и покажем на примерах, как их использовать.

Изучив материал этого занятия, вы сможете:

- ✓ Вставлять одну или несколько строк в таблицу с помощью инструкции `INSERT VALUES`
- ✓ Вставлять результат запроса в таблицу с помощью инструкции `INSERT SELECT`

- ✓ Вставлять результат хранимой процедуры или динамического пакета в таблицу с помощью инструкции `INSERT EXEC`
- ✓ Использовать результат запроса для создания и заполнения таблицы с помощью инструкции `SELECT INTO`

Продолжительность занятия — 30 минут.

Демонстрационные данные

В некоторых примерах кода в данном занятии используется таблица `Sales.MyOrders`. Для создания этой таблицы в демонстрационной базе данных TSQL2012 используйте следующий код:

```
USE TSQL2012;
IF OBJECT_ID('Sales.MyOrders') IS NOT NULL DROP TABLE Sales.MyOrders;
GO
CREATE TABLE Sales.MyOrders
( orderid INT NOT NULL IDENTITY(1, 1)
    CONSTRAINT PK_MyOrders_orderid PRIMARY KEY,
  custid     INT     NOT NULL,
  empid      INT     NOT NULL,
  orderdate  DATE    NOT NULL
    CONSTRAINT DFT_MyOrders_orderdate DEFAULT
      (CAST(SYSDATETIME() AS DATE)),
  shipcountry NVARCHAR(15) NOT NULL,
  freight    MONEY   NOT NULL );
```

Обратите внимание, столбец `orderid` имеет свойство `IDENTITY`, определенное с начальным значением 1 и приращением 1. Это свойство автоматически генерирует значения в этом столбце при вставке строк. В главе 11 свойство столбца `IDENTITY` рассматривается подробно, наряду с альтернативными методами генерации суррогатных ключей, такими как использование объекта последовательности.

Также обратите внимание, что столбец `orderdate` имеет ограничение по умолчанию с выражением, которое возвращает текущую системную дату.

СОВЕТ

Подготовка к экзамену

Вам необходимо четко понимать, как ограничения, определенные в целевой таблице, влияют на инструкции модификации данных.

Инструкция `INSERT VALUES`

С помощью инструкции `INSERT VALUES` можно вставить одну или более строк в целевую таблицу, опираясь на выражения значений. Далее приведен пример инструкции, вставляющей одну строку в таблицу `Sales.MyOrderValues`.

```
INSERT INTO Sales.MyOrders(custid, empid, orderdate, shipcountry, freight)
VALUES(2, 19, '20120620', N'USA', 30.00);
```

Указание имен целевых столбцов после имени таблицы является необязательным, но считается наиболее правильным. Причина этого в том, что таким образом можно контролировать соответствие исходного значения целевому столбцу, независимо от порядка, в котором столбцы определены в таблице.

Без списка целевых столбцов необходимо указывать значения в порядке определения столбцов. Если определение базовой таблицы изменится, но инструкции `INSERT` соответствующим образом не будут модифицированы, это может привести либо к появлению ошибок, либо, что еще хуже, значения будут записаны не в те столбцы.

В инструкции `INSERT VALUES` не указывается значение столбца, имеющего свойство `IDENTITY`, поскольку это свойство генерирует значение столбца автоматически. Заметьте, в предыдущей инструкции не указан столбец `orderid`. Если вы хотите задать собственное значение вместо того, чтобы предоставить это свойству `IDENTITY`, сначала вы должны установить параметр сеанса `IDENTITY_INSERT` следующим образом:

```
SET IDENTITY_INSERT <table> ON;
```

Если вы установили этот параметр, не забудьте отключить его.

Для того чтобы использовать этот параметр, нужно иметь специальные разрешения; вы должны быть владельцем таблицы или иметь разрешение `ALTER` на эту таблицу.

Помимо использования свойства `IDENTITY`, существуют и другие способы автоматического получения столбцом его значения в инструкции `INSERT`. Столбец может иметь связанное с ним ограничение по умолчанию, как например столбец `orderdate` в таблице `Sales.MyOrders`. Если инструкция `INSERT` не указывает значение столбца явно, SQL Server будет использовать выражение по умолчанию для генерации этого значения. Например, в следующей инструкции не указано значение `orderdate`, и поэтому SQL Server использует выражение по умолчанию.

```
INSERT INTO Sales.MyOrders(custid, empid, shipcountry, freight)
VALUES(3, 11, N'USA', 10.00);
```

Другой способ добиться такого же поведения — указать имя столбца в списке имен и ключевое слово `DEFAULT` в соответствующем элементе в списке `VALUES`. Далее приведен пример инструкции `INSERT`, демонстрирующий это:

```
INSERT INTO Sales.MyOrders(custid, empid, orderdate, shipcountry, freight)
VALUES(3, 17, DEFAULT, N'USA', 30.00);
```

Если значение для столбца не задано, SQL Server сначала проверит, получает ли столбец свое значение автоматически, например, с помощью свойства `IDENTITY` или ограничения по умолчанию. Если нет, SQL Server проверит, допускается ли в столбце использование значений `NULL`, и если так, будет присвоено значение `NULL`. Если и это невозможно, SQL Server сгенерирует ошибку:

Инструкция `INSERT VALUES` не ограничивается вставкой только одной строки; напротив, она позволяет вставлять несколько строк. Нужно просто отделить строки запятой, как в следующем примере.

```
INSERT INTO Sales.MyOrders(custid, empid, orderdate, shipcountry, freight)
VALUES (2, 11, '20120620', N'USA', 50.00),
       (5, 13, '20120620', N'USA', 40.00),
       (7, 17, '20120620', N'USA', 45.00);
```

Обратите внимание, вся инструкция рассматривается как одна транзакция. Это означает, что если одна строка не сможет быть вставлена в таблицу, вся инструкция не будет выполнена и не будет вставлена ни одна строка.

Чтобы просмотреть результат запуска всех примеров INSERT в данном занятии, выполните запрос к таблице с помощью следующего кода:

```
SELECT *
FROM Sales.MyOrders;
```

ВАЖНО!**Использование SELECT ***

Как уже упоминалось в главе 2, использование `SELECT *` в промышленной эксплуатации считается плохой практикой. В данной главе `SELECT *` применяется только в запросах, предназначенных для изучения содержимого таблиц после применения изменений.

После запуска этот код возвращает следующий результат:

orderid	custid	empid	orderdate	shipcountry	freight
1	2	19	2012-06-20	USA	30.00
2	3	11	2012-04-19	USA	10.00
3	3	17	2012-04-19	USA	30.00
4	2	11	2012-06-20	USA	50.00
5	5	13	2012-06-20	USA	40.00
6	7	17	2012-06-20	USA	45.00

Помните, что некоторые примеры инструкции `INSERT` основываются на выражении по умолчанию, связанном со столбцом `orderdate`, поэтому естественно, что полученные даты будут соответствовать дате выполнения этих примеров.

Инструкция `INSERT SELECT`

Инструкция `INSERT SELECT` вставляет результирующий набор, возвращенный запросом, в указанную целевую таблицу. Как и в случае с инструкцией `INSERT VALUES`, инструкция `INSERT SELECT` поддерживает необязательное указание имен целевых столбцов. Также можно опустить столбцы, которые получают свои значения автоматически из свойства `IDENTITY`, из ограничения по умолчанию или если разрешены значения `NULL`.

В качестве примера, следующий код вставляет в таблицу `Sales.MyOrders` результат запроса к таблице `Sales.Orders`, возвращающего заказы, отгруженные клиентам из Норвегии.

```
SET IDENTITY_INSERT Sales.MyOrders ON;
INSERT INTO Sales.MyOrders(orderid, custid, empid, orderdate, shipcountry,
                           freight)
```

```
SELECT orderid, custid, empid, orderdate, shipcountry, freight
FROM Sales.Orders
WHERE shipcountry = N'Norway';
SET IDENTITY_INSERT Sales.MyOrders OFF;
```

Этот код включает параметр `IDENTITY_INSERT` применительно к таблице `Sales.MyOrders`, чтобы использовать исходный порядок идентификаторов и не позволить генерировать их свойству `IDENTITY`.

После выполнения этого кода выполните следующий запрос к таблице:

```
SELECT *
FROM Sales.MyOrders;
```

Он возвращает следующий результат:

orderid	custid	empid	orderdate	shipcountry	freight
1	2	19	2012-06-20	USA	30.00
2	3	11	2012-04-19	USA	10.00
3	3	17	2012-04-19	USA	30.00
4	2	11	2012-06-20	USA	50.00
5	5	13	2012-06-20	USA	40.00
6	7	17	2012-06-20	USA	45.00
10387	70	1	2006-12-18	Norway	93.63
10520	70	7	2007-04-29	Norway	13.37
10639	70	7	2007-08-20	Norway	38.64
10831	70	3	2008-01-14	Norway	72.19
10909	70	1	2008-02-26	Norway	53.05
11015	70	2	2008-04-10	Norway	4.62

Последние 6 строк в выходном наборе (со значением `shipcountry`, равным `Norway`) были добавлены последней инструкцией `INSERT SELECT`.

В определенных условиях инструкция `INSERT SELECT` может выигрывать от минимизации записей в журнал, что в свою очередь может привести к повышению производительности по сравнению с полным протоколированием. К таким условиям относится использование простой модели восстановления или модели восстановления с неполным протоколированием, подсказка `TABLOCK` и др. Для получения дополнительных сведений обратитесь к статье "The Data Loading Performance Guide" (Руководство по производительности загрузки данных) по адресу <http://msdn.microsoft.com/en-us/library/dd425070.aspx>.

Инструкция `INSERT EXEC`

С помощью инструкции `INSERT EXEC` можно вставлять результирующий набор (или наборы), возвращенный динамическим пакетом или хранимой процедурой в указанную целевую таблицу. Так же как инструкции `INSERT VALUES` и `INSERT SELECT`, инструкция `INSERT EXEC` поддерживает задание необязательного списка целевых столбцов и позволяет опускать столбцы, которые получают свои значения автоматически.

Чтобы продемонстрировать работу инструкции `INSERT EXEC`, рассмотрим следующий пример, который использует процедуру `Sales.OrdersForCountry`, принимающую страну отгрузки на вход и возвращающую заказы, отгруженные в страну, указанную на входе. Запустите следующий код для создания процедуры `Sales.OrdersForCountry`:

```
IF OBJECT_ID('Sales.OrdersForCountry', 'P') IS NOT NULL
    DROP PROC Sales.OrdersForCountry;
GO
CREATE PROC Sales.OrdersForCountry @country AS NVARCHAR(15)
AS
SELECT orderid, custid, empid, orderdate, shipcountry, freight
FROM Sales.Orders
WHERE shipcountry = @country;
GO
```

Запустите следующий код для вызова этой хранимой процедуры, указав в качестве страны на входе `Portugal` (Португалия), и вставьте результат процедуры в таблицу `Sales.MyOrders`.

```
SET IDENTITY_INSERT Sales.MyOrders ON;
INSERT INTO Sales.MyOrders(orderid, custid, empid, orderdate, shipcountry,
                           freight)
EXEC Sales.OrdersForCountry
@country = N'Portugal';
SET IDENTITY_INSERT Sales.MyOrders OFF;
```

Здесь код также включает параметр `IDENTITY_INSERT` для целевой таблицы, чтобы разрешить инструкции `INSERT` указывать значения для столбца `IDENTITY`, вместо того, чтобы позволить свойству назначать их автоматически.

Выполните запрос к таблице после запуска инструкции `INSERT`.

```
SELECT *
FROM Sales.MyOrders;
```

Далее приведены выходные данные этого кода.

orderid	custid	empid	orderdate	shipcountry	freight
1	2	19	2012-06-20	USA	30.00
2	3	11	2012-04-19	USA	10.00
3	3	17	2012-04-19	USA	30.00
4	2	11	2012-06-20	USA	50.00
5	5	13	2012-06-20	USA	40.00
6	7	17	2012-06-20	USA	45.00
10328	28	4	2006-10-14	Portugal	87.03
10336	60	7	2006-10-23	Portugal	15.51
10352	28	3	2006-11-12	Portugal	1.30
10387	70	1	2006-12-18	Norway	93.63
10397	60	5	2006-12-27	Portugal	60.26
10433	60	3	2007-02-03	Portugal	73.83

10464	28	4	2007-03-04	Portugal	89.00
10477	60	5	2007-03-17	Portugal	13.02
10491	28	8	2007-03-31	Portugal	16.96
10520	70	7	2007-04-29	Norway	13.37
10551	28	4	2007-05-28	Portugal	72.95
10604	28	1	2007-07-18	Portugal	7.46
10639	70	7	2007-08-20	Norway	38.64
10664	28	1	2007-09-10	Portugal	1.27
10831	70	3	2008-01-14	Norway	72.19
10909	70	1	2008-02-26	Norway	53.05
10963	28	9	2008-03-19	Portugal	2.70
11007	60	8	2008-04-08	Portugal	202.24
11015	70	2	2008-04-10	Norway	4.62

СОВЕТ**Инструкция INSERT EXEC с несколькими запросами**

Инструкция `INSERT EXEC` работает, даже если исходный динамический пакет или хранимая процедура содержат более одного запроса. Но это возможно только в том случае, если все запросы возвращают результирующие наборы, совместимые с определением целевой таблицы.

Инструкция `SELECT INTO`

Инструкция `SELECT INTO` состоит из двух частей — запроса (часть `SELECT`) и целевой таблицы (часть `INTO`). Эта инструкция создает целевую таблицу, опираясь на определение источника, и вставляет результирующие строки из запроса в эту таблицу. Кроме самих данных, инструкция копирует из источника некоторые компоненты определения данных, такие как имена столбцов, типы данных, возможность использования значения `NULL` и свойство `IDENTITY`. Часть атрибутов определения данных — индексы, ограничения, триггеры, разрешения и пр., — не копируется. Если нужно включить их в число копируемых, следует записать их из источника и применить к целевой таблице.

В следующем коде приведен пример инструкции `SELECT INTO`, которая запрашивает таблицу `Sales.Orders`, возвращая заказы, отгруженные в Норвегию (`Norway`), создает целевую таблицу с именем `Sales.MyOrders` и сохраняет результат запроса в целевой таблице.

```
IF OBJECT_ID('Sales.MyOrders', 'U') IS NOT NULL DROP TABLE Sales.MyOrders;
SELECT orderid, custid, orderdate, shipcountry, freight
INTO Sales.MyOrders
FROM Sales.Orders
WHERE shipcountry = N'Norway';
```

Как уже упоминалось, инструкция `SELECT INTO` создает целевую таблицу, опираясь на определение источника. У вас нет прямого контроля над определением целевой таблицы. Если необходимо, чтобы целевые столбцы были определены иначе, чем исходные, необходимо выполнить некоторые манипуляции.

Например, исходный столбец `orderid` имеет свойство `IDENTITY`, и, соответственно, целевой столбец также определен со свойством `IDENTITY`. Если вы хотите, чтобы

у целевого столбца не было этого свойства, вам следует выполнить некоторую обработку, например, `orderid + 0 AS orderid`. Обратите внимание, после применения этой обработки целевой столбец будет определен как допускающий значения NULL. Если необходимо, чтобы целевой столбец был определен как недопускающий значения NULL, требуется использовать функцию `ISNULL`, возвращающую не-NULL-значение в случае, если источник содержит NULL. Это не более чем искусственное выражение, которое сообщает SQL Server, что выходное значение не может быть NULL, следовательно, столбец может быть определен как неразрешающий значения NULL. Например, можно использовать такое выражение: `ISNULL(orderid + 0, -1) AS orderid`.

Аналогично, исходный столбец `custid` определен в источнике как разрешающий значения NULL. Чтобы целевой столбец был определен как NOT NULL, следует использовать выражение `ISNULL(custid, -1) AS custid`.

Если необходимо, чтобы тип целевого столбца отличался от типа источника, можно использовать функции `CAST` или `CONVERT`. Но следует помнить, что в таком случае целевой столбец будет определен как допускающий значения NULL, даже если исходный столбец не разрешает их использование, поскольку вы обработали выходной столбец. Как и в предыдущих примерах, можно использовать функцию `ISNULL` для того, чтобы SQL Server определял целевой столбец как недопускающий значения NULL. Например, чтобы преобразовать исходный тип данных столбца `orderdate` `DATETIME` в тип данных `DATE` в целевом столбце и отменить разрешение значений NULL, используйте выражение `ISNULL(CAST(orderdate AS DATE), '19000101') AS orderdate`.

Чтобы обобщить все сказанное, следующий код использует запрос, подобный предыдущему примеру, но только определяя столбец `orderid` без свойства `IDENTITY` и как NOT NULL, столбец `custid` как NOT NULL и столбец `orderdate` как DATE NOT NULL.

```
IF OBJECT_ID('Sales.MyOrders', 'U') IS NOT NULL DROP TABLE Sales.MyOrders;
```

```
SELECT
    ISNULL(orderid + 0, -1) AS orderid, -- избавляемся от свойства IDENTITY,
                                         -- делаем столбец NOT NULL
    ISNULL(custid, -1) AS custid,      -- делаем столбец NOT NULL
    empid,
    ISNULL(CAST(orderdate AS DATE), '19000101') AS orderdate,
    shipcountry, freight
INTO Sales.MyOrders
FROM Sales.Orders
WHERE shipcountry = N'Norway';
```

Помните, что инструкция `SELECT INTO` не копирует ограничения из исходной таблицы, поэтому, если они нужны, вы сами должны определить их в целевой таблице. Например, следующий код определяет ограничение первичного ключа в целевой таблице.

```
ALTER TABLE Sales.MyOrders
ADD CONSTRAINT PK_MyOrders PRIMARY KEY (orderid);
```

Выполните запрос к таблице, чтобы увидеть результат инструкции SELECT INTO.

```
SELECT *
FROM Sales.MyOrders;
```

Вы получите следующий результат:

orderid	custid	empid	orderdate	shipcountry	freight
10387	70	1	2006-12-18	Norway	93.63
10520	70	7	2007-04-29	Norway	13.37
10639	70	7	2007-08-20	Norway	38.64
10831	70	3	2008-01-14	Norway	72.19
10909	70	1	2008-02-26	Norway	53.05
11015	70	2	2008-04-10	Norway	4.62

Одним из преимуществ использования инструкции SELECT INTO является то, что когда для базы данных задана не модель полного протоколирования данных, а простая модель или модель с неполным протоколированием, эта инструкция использует минимальное протоколирование. Это может привести к более быстрому выполнению вставки по сравнению с режимом полного протоколирования.

У инструкции SELECT INTO имеются и недостатки. Один из них — ограниченный контроль над определением целевой таблицы. Ранее в этом занятии вы изучали косвенное управление определением целевых столбцов. Но некоторые вещи просто невозможно контролировать — например, файловую группу целевой таблицы.

Также следует помнить, что инструкция SELECT INTO затрагивает как создание таблицы, так и заполнение ее данными. Это означает, что и относящиеся к целевой таблице метаданные, и данные монопольно блокируются до тех пор, пока не будет закончена транзакция SELECT INTO. В результате можно столкнуться с блокировками, вызванными конфликтами, связанными с доступом как к данным, так и к метаданным.

По окончании запустите следующий код для очистки данных:

```
IF OBJECT_ID('Sales.MyOrders', 'U') IS NOT NULL
DROP TABLE Sales.MyOrders;
```

КОНТРОЛЬНЫЕ ВОПРОСЫ

- Почему рекомендуется указывать имена целевых столбцов в инструкциях INSERT?
- В чем заключается разница между инструкциями SELECT INTO и INSERT SELECT?

Ответы на контрольные вопросы

- Потому что тогда не нужно заботиться о порядке, в котором столбцы определены в таблице. Вы также не пострадаете при изменении порядка столбцов из-за будущего изменения определения, а также при добавлении столбцов, получающих свои значения автоматически.
- Инструкция SELECT INTO создает целевую таблицу и вставляет в нее результат запроса. Инструкция INSERT SELECT вставляет результат запроса в уже существующую таблицу.

ПРАКТИКУМ Вставка данных

В этом практикуме вы проверите ваши знания в области вставки данных.

Задание 1. Вставка данных о клиентах без заказов

В этом задании вам нужно определить клиентов, не разместивших заказ, и вставить данные этих клиентов в целевую таблицу. Вам следует использовать инструкции `INSERT` и `SELECT INTO`.

1. Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.

2. Проанализируйте структуру таблицы `Sales.Customers`, запустив следующий код:

```
EXEC sp_describe_first_result_set N'SELECT * FROM Sales.Customers;';
```

Обратите внимание на следующие атрибуты на выходе `sp_describe_first_result_set`: `name`, `system_type_name` и `is_nullable`.

3. Создайте таблицу `Sales.MyCustomers`, используя определение таблицы `Sales.Customers`. Задайте первичный ключ на столбце `custid`. Не задавайте свойство `IDENTITY` на столбце `custid`. Ваша инструкция для создания таблицы должна выглядеть следующим образом:

```
IF OBJECT_ID('Sales.MyCustomers') IS NOT NULL DROP TABLE Sales.MyCustomers;
```

```
CREATE TABLE Sales.MyCustomers
( custid      INT NOT NULL
    CONSTRAINT PK_MyCustomers PRIMARY KEY,
  companyname NVARCHAR(40) NOT NULL,
  contactname NVARCHAR(30) NOT NULL,
  contacttitle NVARCHAR(30) NOT NULL,
  address     NVARCHAR(60) NOT NULL,
  city        NVARCHAR(15) NOT NULL,
  region      NVARCHAR(15) NULL,
  postalcode  NVARCHAR(10) NULL,
  country     NVARCHAR(15) NOT NULL,
  phone       NVARCHAR(24) NOT NULL,
  fax         NVARCHAR(24) NULL );
```

4. Напишите инструкцию `INSERT`, которая вставляет в таблицу `Sales.MyCustomers` данные из таблицы `Sales.Customers` для клиентов, не разместивших заказы. Ваша инструкция `INSERT` должна выглядеть следующим образом:

```
INSERT INTO Sales.MyCustomers
(custid, companyname, contactname, contacttitle, address,
 city, region, postalcode, country, phone, fax)
SELECT custid, companyname, contactname, contacttitle, address,
       city, region, postalcode, country, phone, fax
  FROM Sales.Customers AS C
```

```
WHERE NOT EXISTS
  (SELECT * FROM Sales.Orders AS O
   WHERE O.custid = C.custid);
```

5. После выполнения предыдущей инструкции `INSERT` сделайте запрос к таблице `Sales.MyCustomers` для возвращения идентификаторов вставленных клиентов.

```
SELECT custid FROM Sales.MyCustomers;
```

Вы получите следующий результат:

```
custid
-----
22
57
```

Задание 2. Использование инструкции `SELECT INTO`

В этом задании вы будете использовать инструкцию `SELECT INTO` для создания таблицы и заполнения ее данными для клиентов, не разместивших заказы.

1. Получите тот же результат, что и в *задании 1*, но на этот раз с помощью команды `SELECT INTO` вместо инструкций `CREATE TABLE` и `INSERT SELECT`. Ваше решение должно выглядеть следующим образом:

```
IF OBJECT_ID('Sales.MyCustomers') IS NOT NULL DROP TABLE Sales.MyCustomers;

SELECT ISNULL(custid, -1) AS custid,
       companyname, contactname, contacttitle, address,
       city, region, postalcode, country, phone, fax
  INTO Sales.MyCustomers
   FROM Sales.Customers AS C
  WHERE NOT EXISTS
    (SELECT * FROM Sales.Orders AS O
     WHERE O.custid = C.custid);

ALTER TABLE Sales.MyCustomers
  ADD CONSTRAINT PK_MyCustomers PRIMARY KEY(custid);

SELECT custid FROM Sales.MyCustomers;
```

На выходе последнего запроса вы получите следующий результат:

```
custid
-----
22
57
```

Резюме занятия

- Язык T-SQL поддерживает различные инструкции, выполняющие вставку данных в таблицы в базе данных. К ним относятся инструкции `INSERT VALUES`, `INSERT SELECT`, `INSERT EXEC`, `SELECT INTO` и др.

- С помощью инструкции `INSERT VALUES` можно вставлять одну или более строк в целевую таблицу, опираясь на выражения значений.
- С помощью инструкции `INSERT SELECT` можно вставлять в целевую таблицу результат запроса.
- Можно использовать инструкцию `INSERT EXEC` для вставки в целевую таблицу результата запросов в динамическом пакете или хранимой процедуре.
- Используя инструкции `INSERT VALUES`, `INSERT SELECT` и `INSERT EXEC`, можно опустить столбцы, получающие свои значения автоматически. Столбец может получить свое значение автоматически, если он имеет связанное с ним ограничение по умолчанию, или свойство `IDENTITY`, или если он допускает использование значений `NULL`.
- Инструкция `SELECT INTO` создает целевую таблицу на основании определения данных в исходном запросе и вставляет результат запроса в целевую таблицу.
- Считается наиболее правильным указывать в инструкции `INSERT` имена целевых столбцов, чтобы удалить зависимость от порядка следования столбцов в определении целевой таблицы.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. В каком из перечисленных ниже случаев, как правило, нельзя указать целевой столбец в инструкции `INSERT`?
 - A. Если столбец имеет связанное с ним ограничение по умолчанию.
 - B. Если столбец разрешает значения `NULL`.
 - C. Если столбец не разрешает значения `NULL`.
 - D. Если столбец имеет свойство `IDENTITY`.
2. Что именно инструкция `SELECT INTO` не копирует из источника? (Выберите все подходящие варианты.)
 - A. Индексы.
 - B. Ограничения.
 - C. Свойство `IDENTITY`.
 - D. Тrigгеры.
3. В чем заключаются преимущества использования комбинации инструкций `CREATE TABLE` и `INSERT SELECT` перед инструкцией `SELECT INTO`? (Выберите все подходящие варианты.)
 - A. С помощью инструкции `CREATE TABLE` можно контролировать все атрибуты целевой таблицы. Используя инструкцию `SELECT INTO`, невозможно контролировать некоторые атрибуты, такие как целевая файловая группа.

- В. Инструкция `INSERT SELECT` более быстрая по сравнению с инструкцией `SELECT INTO`.
- С. Инструкция `SELECT INTO` блокирует как данные, так и метаданные на весь период транзакции. Это означает, что до окончания транзакции можно столкнуться с блокировкой, связанной и с данными, и с метаданными. Если инструкции `CREATE TABLE` и `INSERT SELECT` запускаются из разных транзакций, блокировки метаданных будут сняты довольно быстро, снижая возможность и продолжительность блокирования метаданных.
- Д. Использование инструкции `CREATE TABLE` вместе с `INSERT SELECT` требует написания меньшего количества кода, чем использование инструкции `SELECT INTO`.

Занятие 2. Обновление данных

Язык T-SQL поддерживает инструкцию `UPDATE`, позволяющую обновлять существующие данные в таблицах. В этом занятии вы познакомитесь со стандартной инструкцией `UPDATE`, а также с несколькими дополнениями в T-SQL к стандарту. Также вы узнаете о модификации данных с использованием соединений и изучите недетерминированные обновления. Наконец, мы расскажем о модификации данных посредством табличных выражений, обновлении данных с помощью переменных и о том, как обновления влияют операции, основанные на принципе единовременности.

Изучив материал этого занятия, вы сможете:

- ✓ Использовать инструкцию `UPDATE` для модификации строк
- ✓ Обновлять данные с помощью соединений
- ✓ Описывать обстоятельства, при которых получаются недетерминированные обновления
- ✓ Обновлять данные с помощью табличных выражений
- ✓ Обновлять данные с помощью переменных
- ✓ Объяснить влияние принципа единовременности в SQL на обновления

Продолжительность занятия — 30 минут.

Демонстрационные данные

Этот раздел, в котором рассматривается обновление данных, так же как и следующий, описывающий удаление данных, использует демонстрационные данные, представленные в таблицах `Sales.MyCustomers` с данными о клиентах, `Sales.MyOrders` с данными о заказах и `Sales.MyOrderDetails`, содержащей подробные данные о заказах. Эти таблицы являются копиями таблиц `Sales.Customers`, `Sales.Orders` и `Sales.OrderDetails` из учебной базы данных TSQL2012. Работая с копиями исходных таблиц, вы можете запускать примеры кода, которые обновляют и удаляют строки, не боясь изменить исходные таблицы. Для создания и заполнения этих учебных таблиц используйте следующий код:

```
IF OBJECT_ID('Sales.MyOrderDetails', 'U') IS NOT NULL
    DROP TABLE Sales.MyOrderDetails;
IF OBJECT_ID('Sales.MyOrders', 'U') IS NOT NULL
    DROP TABLE Sales.MyOrders;
IF OBJECT_ID('Sales.MyCustomers', 'U') IS NOT NULL
    DROP TABLE Sales.MyCustomers;

SELECT * INTO Sales.MyCustomers FROM Sales.Customers;
ALTER TABLE Sales.MyCustomers
    ADD CONSTRAINT PK_MyCustomers PRIMARY KEY(custid);

SELECT * INTO Sales.MyOrders FROM Sales.Orders;
ALTER TABLE Sales.MyOrders
    ADD CONSTRAINT PK_MyOrders PRIMARY KEY(orderid);

SELECT * INTO Sales.MyOrderDetails FROM Sales.OrderDetails;
ALTER TABLE Sales.MyOrderDetails
    ADD CONSTRAINT PK_MyOrderDetails PRIMARY KEY(orderid, productid);
```

Инструкция *UPDATE*

T-SQL поддерживает стандартную инструкцию *UPDATE*, которая позволяет обновлять существующие строки в таблице. Стандартная инструкция *UPDATE* имеет следующий формат:

```
UPDATE <target table>
    SET <col 1> = <expression 1>,
        <col 2> = <expression 2>,
        ... , <col n> = <expression n>
    WHERE <predicate>;
```

Имя целевой таблицы указывается в предложении *UPDATE*. Если вам нужно отфильтровать подмножество строк, следует указать предложение *WHERE* с предикатом. Только строки, для которых предикат принимает значение "истина", будут обновлены. Строки, для которых предикат принимает значение "ложь" или "неизвестное значение", не будут обработаны. Инструкция *UPDATE* без предложения *WHERE* воздействует на все строки. Значения присваиваются целевым столбцам в предложении *SET*. В исходных выражениях могут присутствовать столбцы из таблицы, в таком случае используются их значения до обновления.

ВАЖНО!

Остерегайтесь неуточненных обновлений

Как уже упоминалось, неполная инструкция *UPDATE* воздействует на все строки в целевой таблице. Следует соблюдать особенную осторожность при непреднамеренном использовании только предложений *UPDATE* и *SET* этой инструкции без предложения *WHERE*.

В качестве примера измените строки в таблице *Sales.MyOrderDetails*, представляющей строки заказа, относящиеся к заказу 10251. Перед обновлением запросите эти строки, чтобы проверить их состояние.

Вы получите такой результат:

orderid	productid	unitprice	qty	discount
10251	22	16.80	6	0.050
10251	57	15.60	15	0.050
10251	65	16.80	20	0.000

Следующий код демонстрирует инструкцию UPDATE, которая добавляет скидку 5% к этим строкам заказа.

```
UPDATE Sales.MyOrderDetails
    SET discount += 0.05
WHERE orderid = 10251;
```

Обратите внимание на использование оператора сложного присвоения `discount += 0.05`. Это присвоение эквивалентно выражению `discount = discount + 0.05`. T-SQL поддерживает такие расширенные операторы для всех бинарных операторов присвоения: `+=` (добавление), `-=` (вычитание), `*=` (умножение), `/=` (деление), `%=` (деление по модулю), `&=` (побитовая операция "И"), `|=` (побитовая операция "ИЛИ"), `^=` (побитовая операция "исключающее ИЛИ"), `+=` (конкатенация).

Снова выполните запрос строк заказов, связанных с заказом 10251, чтобы увидеть их состояние после обновления.

```
SELECT *
FROM Sales.MyOrderDetails
WHERE orderid = 10251;
```

Вы получите следующий результат, показывающий увеличение скидки (`discount`) на 5%.

orderid	productid	unitprice	qty	discount
10251	22	16.80	6	0.100
10251	57	15.60	15	0.100
10251	65	16.80	20	0.050

Используйте следующий код для снижения скидки в указанных выше строках на 5%.

```
UPDATE Sales.MyOrderDetails
    SET discount -= 0.05
WHERE orderid = 10251;
```

Теперь эти строки должны вернуться в свое начальное состояние, в котором они были до первого обновления.

Обновление с использованием объединения

Стандартный язык SQL не поддерживает объединения в инструкции UPDATE, но язык T-SQL поддерживает. Суть состоит в том, что вы можете захотеть обновить строки в таблице и сослаться на строки в других таблицах для выполнения операций фильтрации или присвоения.

В качестве примера представьте, что вам необходимо добавить скидку в 5% в строках заказов, связанных с заказами, размещенными клиентами из Норвегии (*Norway*). Строки, которые вы будете модифицировать, находятся в таблице *Sales.MyOrderDetails*. Но информация, которую надо исследовать для выполнения фильтрации, находится в строках, расположенных в таблице *Sales.MyCustomers*. Чтобы связать клиента с соответствующими ему строками заказов, надо объединить *Sales.MyCustomers* с таблицей *Sales.MyOrders* и затем объединить результат с таблицей *Sales.MyOrderDetails*. Обратите внимание, недостаточно исследовать столбец *shipcountry* в таблице *Sales.MyOrders*; вы должны проверять столбец *country* в таблице *Sales.MyCustomers*.

В соответствии с полученными в предыдущих главах знаниями об объединениях, если бы вы захотели написать инструкцию *SELECT*, возвращающую строки заказов, являющиеся целевыми для операции обновления, то вы написали бы запрос, подобный следующему:

```
SELECT OD.*  
FROM Sales.MyCustomers AS C  
INNER JOIN Sales.MyOrders AS O  
    ON C.custid = O.custid  
Sales.MyOrderDetails AS OD  
    ON O.orderid = OD.orderid  
WHERE C.country = N'Norway';
```

Этот запрос генерирует следующие выходные данные:

ordered	productid	unitprice	qty	discount
10387	24	3.60	15	0.000
10387	28	36.40	6	0.000
10387	59	44.00	12	0.000
10387	71	17.20	15	0.000
10520	24	4.50	8	0.000
10520	53	32.80	5	0.000
10639	18	62.50	8	0.000
10831	19	9.20	2	0.000
10831	35	18.00	8	0.000
10831	38	263.50	8	0.000
10831	43	46.00	9	0.000
10909	7	30.00	12	0.000
10909	16	17.45	15	0.000
10909	41	9.65	5	0.000
11015	30	25.89	15	0.000
11015	77	13.00	18	0.000

Чтобы выполнить требуемое обновление, просто замените предложение *SELECT* в последнем запросе предложением *UPDATE*, указав псевдоним таблицы, которая является целевой для *UPDATE* (в данном случае это *OD*), и назначение в предложении *SET* следующим образом:

```

UPDATE OD
    SET OD.discount += 0.05
  FROM Sales.MyCustomers AS C
    INNER JOIN Sales.MyOrders AS O
      ON C.custid = O.custid
    INNER JOIN Sales.MyOrderDetails AS OD
      ON O.orderid = OD.orderid
 WHERE C.country = N'Norway';

```

Вы получите следующий результат:

ordered	productid	unitprice	qty	discount
10387	24	3.60	15	0.050
10387	28	36.40	6	0.050
10387	59	44.00	12	0.050
10387	71	17.20	15	0.050
10520	24	4.50	8	0.050
10520	53	32.80	5	0.050
10639	18	62.50	8	0.050
10831	19	9.20	2	0.050
10831	35	18.00	8	0.050
10831	38	263.50	8	0.050
10831	43	46.00	9	0.050
10909	7	30.00	12	0.050
10909	16	17.45	15	0.050
10909	41	9.65	5	0.050
11015	30	25.89	15	0.050
11015	77	13.00	18	0.050

Обратите внимание на 5-процентное увеличение скидки в обработанных строках заказа.

Чтобы вернуть строки заказа в их исходное состояние, запустите инструкцию UPDATE, которая снижает скидку на 5%.

```

UPDATE OD
    SET OD.discount -= 0.05
  FROM Sales.MyCustomers AS C
    INNER JOIN Sales.MyOrders AS O
      ON C.custid = O.custid
    INNER JOIN Sales.MyOrderDetails AS OD
      ON O.orderid = OD.orderid
 WHERE C.country = N'Norway';

```

Недетерминированная инструкция *UPDATE*

Вы должны знать, что синтаксис собственной инструкции UPDATE языка T-SQL, использующей объединения, может быть недетерминированным. Инструкция является недетерминированной, когда несколько исходных строк соответствуют одной

целевой строке. К сожалению, в таком случае SQL Server не генерирует ошибку или хотя бы предупреждение. Вместо этого SQL Server молча выполняет недетерминированную инструкцию UPDATE, в которой одна из исходных строк "выигрывает" произвольно.

СОВЕТ**Использование инструкции MERGE вместо UPDATE**

Вместо того чтобы использовать нестандартную инструкцию UPDATE, использующую объединения, вы можете применить стандартную инструкцию MERGE. Она генерирует сообщение об ошибке, если несколько исходных строк связаны с одной целевой строкой, требуя от вас исправить ваш код так, чтобы он стал детерминированным. Инструкция MERGE рассматривается в главе 11.

Возьмем для примера следующий запрос, который связывает клиентов с соответствующими им заказами, возвращая почтовые индексы клиентов, а также почтовые индексы мест отгрузки для соответствующих заказов.

```
SELECT C.custid, C.postalcode, O.shippostalcode
FROM Sales.MyCustomers AS C
INNER JOIN Sales.MyOrders AS O
    ON C.custid = O.custid
ORDER BY C.custid;
```

Этот запрос генерирует следующие выходные данные:

custid	postalcode	shippostalcode
1	10092	10154
1	10092	10156
1	10092	10155
1	10092	10154
1	10092	10154
1	10092	10154
2	10077	10182
2	10077	10181
2	10077	10181
2	10077	10180
...		

Каждая строка клиента повторяется в выходных данных для каждого соответствующего заказа. Это означает, что единственный почтовый индекс каждого клиента столько раз повторяется в выходных данных, сколько имеется соответствующих заказов. Для данного примера важно помнить, что существует только один почтовый индекс для каждого клиента. Почтовый индекс отгрузки связан с заказом, поэтому вы можете заметить, что может быть несколько различных почтовых индексов отгрузки для каждого клиента.

Помня об этом, рассмотрим следующую инструкцию UPDATE:

```
UPDATE C
SET C.postalcode = O.shippostalcode
```

```
FROM Sales.MyCustomers AS C
INNER JOIN Sales.MyOrders AS O
ON C.custid = O.custid;
```

89 клиентов имеют связанные с ними заказы — некоторые с несколькими соответствиями. SQL Server не генерирует сообщение об ошибке; вместо этого он произвольно выбирает целевую строку, для которой исходная строка будет выбрана для обновления, возвращая следующее сообщение:

```
(89 row(s) affected)
```

Запросите строки из таблицы Sales.Customers после обновления.

```
SELECT custid, postalcode
FROM Sales.MyCustomers
ORDER BY custid;
```

Этот запрос генерирует следующие выходные данные, но у вас может получиться другой результат.

```
custid  postalcode
-----  -----
1        10154
2        10182
...
(91 row(s) affected)
```

Заметьте, в таблице 91 строка, но поскольку только 89 из этих клиентов имеют соответствующие заказы, предыдущая инструкция UPDATE обработала 89 строк.

Что касается того, какая исходная строка выбирается как соответствующая каждой целевой строке, этот выбор является произвольным, но не случайным; иными словами, он зависит от оптимизации. Так или иначе, язык не предлагает никаких логических элементов, позволяющих контролировать этот выбор. Рекомендуется просто не использовать такую недетерминированную инструкцию UPDATE. Сначала надо логически выяснить, как разорвать связи, и после того, как это сделано, вы можете написать детерминированную инструкцию UPDATE, которая включает в себя логику разрыва связей.

Например, пусть вы хотите обновить почтовый индекс клиента почтовым индексом отгрузки из первого заказа этого клиента (используя порядок сортировки orderdate, orderid). Этого можно достигнуть с помощью оператора APPLY следующим образом:

```
UPDATE C
SET C.postalcode = A.shippostalcode
FROM Sales.MyCustomers AS C
CROSS APPLY (SELECT TOP (1) O.shippostalcode
              FROM Sales.MyOrders AS O
             WHERE O.custid = C.custid
             ORDER BY orderdate, orderid) AS A;
```

SQL Server сгенерирует следующее сообщение:

```
(89 row(s) affected)
```

Выполните запрос к таблице Sales.MyCustomers после обновления.

```
SELECT custid, postalcode
FROM Sales.MyCustomers
ORDER BY custid;
```

Вы получите следующий результат:

```
custid      postalcode
-----  -----
1          10154
2          10180
...
(91 row(s) affected)
```

Если в качестве источника обновления вы хотите использовать самый последний порядок сортировки, просто используйте сортировку по убыванию в обоих столбцах: ORDER BY orderdate DESC, orderid DESC.

Инструкция *UPDATE* и табличные выражения

Используя язык T-SQL, можно модифицировать данные с помощью табличных выражений, таких как обобщенные табличные выражения и производные таблицы. Это может быть полезно, например, когда вам понадобится возможность посмотреть, какие строки будут обновлены и какими данными, прежде чем вы действительно будете выполнять обновление.

Предположим, вам необходимо модифицировать столбцы country и postalcode таблицы Sales.MyCustomers, используя данные из соответствующих строк таблицы Sales.Customers. Но вы хотите иметь возможность сначала выполнить код в виде предложения SELECT, чтобы просмотреть данные, которые собираетесь обновить. Сначала можно написать запрос SELECT, как в следующем примере:

```
SELECT TGT.custid,
       TGT.country AS tgt_country, SRC.country AS src_country,
       TGT.postalcode AS tgt_postalcode, SRC.postalcode AS src_postalcode
  FROM Sales.MyCustomers AS TGT
    INNER JOIN Sales.Customers AS SRC
      ON TGT.custid = SRC.custid;
```

Этот запрос генерирует следующие выходные данные:

custid	tgt_country	src_country	tgt_postalcode	src_postalcode
1	Germany	Germany	10154	10092
2	Mexico	Mexico	10180	10077
3	Mexico	Mexico	10211	10097
4	UK	UK	10238	10046

5	Sweden	Sweden	10269	10112
6	Germany	Germany	10302	10117
7	France	France	10329	10089
8	Spain	Spain	10359	10104
9	France	France	10369	10105
10	Canada	Canada	10130	10111

Но для того, чтобы действительно выполнить обновление, нужно заменить предложение `SELECT` предложением `UPDATE`, как показано далее.

```
UPDATE TGT
    SET TGT.country = SRC.country,
        TGT.postalcode = SRC.postalcode
    FROM Sales.MyCustomers AS TGT
    INNER JOIN Sales.Customers AS SRC
        ON TGT.custid = SRC.custid;
```

В качестве альтернативы, более простым вам может показаться определение табличного выражения на основании последнего запроса и выполнение модификации посредством табличного выражения. Следующий код показывает, как этого можно достичнуть, используя обобщенное табличное выражение.

```
WITH C AS
    (SELECT TGT.custid,
        TGT.country AS tgt_country, SRC.country AS src_country,
        TGT.postalcode AS tgt_postalcode, SRC.postalcode AS src_postalcode
    FROM Sales.MyCustomers AS TGT
    INNER JOIN Sales.Customers AS SRC
        ON TGT.custid = SRC.custid )
UPDATE C
    SET tgt_country = src_country,
        tgt_postalcode = src_postalcode;
```

В конечном итоге таблица `Sales.MyCustomers` модифицирована. Используя такое решение, вы всегда можете выделить внутренний запрос `SELECT` и запустить его независимо, чтобы увидеть данные, участвующие в обновлении, не выполняя собственно обновление.

Ту же цель можно достичь, используя производную таблицу, как показано далее.

```
UPDATE D
    SET tgt_country = src_country,
        tgt_postalcode = src_postalcode
    FROM (SELECT TGT.custid, TGT.country AS tgt_country,
                SRC.country AS src_country, TGT.postalcode AS tgt_postalcode,
                SRC.postalcode AS src_postalcode
            FROM Sales.MyCustomers AS TGT
            INNER JOIN Sales.Customers AS SRC
                ON TGT.custid = SRC.custid
        ) AS D;
```

Обратите внимание, необходимо использовать предложение `FROM` для того, чтобы определить производную таблицу и затем задать имя производной таблицы в предложении `UPDATE`.

Возвращаясь к инструкциям `UPDATE` с использованием объединений, ранее в этом занятии вы видели следующий код:

```
UPDATE TGT
    SET TGT.country = SRC.country,
        TGT.postalcode = SRC.postalcode
  FROM Sales.MyCustomers AS TGT
    INNER JOIN Sales.Customers AS SRC
      ON TGT.custid = SRC.custid;
```

Интересно, что если вы напишете инструкцию `UPDATE` с таблицей А в предложении `UPDATE` и таблицей В (не А) в предложении `FROM`, то получите неявное перекрестное соединение между А и В. Если вы еще добавите фильтр с предикатом, содержащим элементы из обеих таблиц, то получите логический эквивалент внутреннего соединения. Опираясь на эту логику, следующая инструкция достигает того же результата, что и предыдущая.

```
UPDATE Sales.MyCustomers
    SET MyCustomers.country = SRC.country,
        MyCustomers.postalcode = SRC.postalcode
  FROM Sales.Customers AS SRC
 WHERE MyCustomers.custid = SRC.custid;
```

Этот код эквивалентен следующему использованию явного перекрестного соединения с фильтром:

```
UPDATE TGT
    SET TGT.country = SRC.country,
        TGT.postalcode = SRC.postalcode
  FROM Sales.MyCustomers AS TGT
    CROSS JOIN Sales.Customers AS SRC
 WHERE TGT.custid = SRC.custid;
```

И этот код является логическим эквивалентом, упоминавшимся ранее в инструкции `UPDATE` с явным внутренним соединением.

Возможность обновления данных с помощью табличных выражений также удобна, когда вы хотите изменить строки с помощью выражений, которые обычно запрещены в предложении `SET`. Например, оконные функции не поддерживаются в предложении `SET`. Чтобы обойти это, можно вызвать оконную функцию в списке `SELECT` внутреннего запроса и назначить псевдоним столбца результирующему столбцу. Затем во внешней инструкции `UPDATE` можно сослаться на псевдоним столбца как на исходное выражение в предложении `SET`.

Инструкция *UPDATE* с использованием переменной

Иногда необходимо изменить строку и также занести результат модифицированных столбцов в переменные. Для достижения этого можно использовать комбинацию инструкций *UPDATE* и *SELECT*, но для этого потребуются два обращения к строке. Язык T-SQL поддерживает специализированный синтаксис *UPDATE*, который позволяет решить такую задачу с помощью одной инструкции и одного обращения к строке.

В качестве примера выполните следующий запрос для просмотра текущего состояния строки заказа, связанной с заказом 10250 и продуктом 51.

```
SELECT *
FROM Sales.MyOrderDetails
WHERE orderid = 10250 AND productid = 51;
```

Этот код генерирует следующие выходные данные:

orderid	productid	unitprice	qty	discount
10250	51	42.40	35	0.150

Предположим, вам нужно модифицировать строку, увеличив скидку (*discount*) на 5% и поместить новое значение скидки в переменную с именем *@newdiscount*. Это можно сделать, используя единственную инструкцию *UPDATE* следующим образом:

```
DECLARE @newdiscount AS NUMERIC(4, 3) = NULL;

UPDATE Sales.MyOrderDetails
    SET @newdiscount = discount += 0.05
WHERE orderid = 10250
    AND productid = 51;
```

```
SELECT @newdiscount;
```

Как видите, предложения *UPDATE* и *WHERE* подобны тем, которые вы используете в обычной инструкции *UPDATE*. Но предложение *SET* использует присвоение *@newdiscount = discount += 0.05*, которое эквивалентно использованию выражения *@newdiscount = discount = discount + 0.05*. Эта инструкция присваивает результат *discount + 0.05* величине *discount* и затем присваивает результат переменной *@newdiscount*.

Последняя инструкция *SELECT* в коде генерирует следующий результат:

```
-----
0.200
```

По окончании запустите следующий код, чтобы отменить последнее изменение:

```
UPDATE Sales.MyOrderDetails
    SET discount -= 0.05
WHERE orderid = 10250
    AND productid = 51;
```

UPDATE и принцип единовременности

Ранее в этом учебном курсе в главах 1 и 3 мы обсуждали концепцию, называемую принципом единовременности (all-at-once). В этих главах объяснялось, что данный принцип означает, что все выражения, которые появляются в той же фазе логической обработки запроса, концептуально оцениваются в один и тот же момент времени.

Принцип единовременности также оказывает влияние на инструкцию UPDATE. Чтобы продемонстрировать такие воздействия, в данном разделе нам понадобится таблица с именем T1. Используйте следующий код для создания этой таблицы T1 и вставки в нее строки:

```
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;

CREATE TABLE dbo.T1
( keycol INT NOT NULL CONSTRAINT PK_T1 PRIMARY KEY,
  col1 INT NOT NULL,
  col2 INT NOT NULL );
INSERT INTO dbo.T1(keycol, col1, col2) VALUES(1, 100, 0);
```

Далее рассмотрите следующий код, но не запускайте его пока:

```
DECLARE @add AS INT = 10;

UPDATE dbo.T1
    SET col1 += @add, col2 = col1
WHERE keycol = 1;
SELECT * FROM dbo.T1;
```

Как вы предполагаете, что за величина должна находиться в столбце col2 в модифицированной строке после обновления? Если вы думаете, что это 110, вы просто не подумали о свойстве SQL, называемом принципом единовременности. На основании этого свойства все присвоения используют оригинальные значения строки в качестве исходных значений, независимо от последовательности их появления. Поэтому присвоение col2 = col1 получает не значение col1 после изменения, а значение до изменения, т. е. 100. Чтобы это проверить, запустите следующий код:

Вы получите такой результат:

keycol	col1	col2
1	110	100

По окончании запустите следующий код для очистки данных:

```
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;
```

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие строки таблицы обновляются в инструкции UPDATE без предложения WHERE?
2. Можно ли обновить строки более чем в одной таблице, используя инструкцию UPDATE?

Ответы на контрольные вопросы

1. Все строки таблицы.
2. Нет, вы можете использовать в качестве исходных столбцы из нескольких таблиц, но обновить можно только одну таблицу за один раз.

ПРАКТИКУМ Обновление данных

В этом практикуме вы будете использовать полученные знания об обновлении данных.

Задание 1. Обновление данных с использованием соединений

В этом задании вам нужно выполнить обновление данных с помощью объединений.

1. Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
2. Используйте следующий код для создания таблицы Sales.MyCustomers и заполнения ее парой строк, представляющих клиентов с идентификаторами 22 и 57.

```
IF OBJECT_ID('Sales.MyCustomers') IS NOT NULL DROP TABLE Sales.MyCustomers;
CREATE TABLE Sales.MyCustomers
( custid      INT NOT NULL
  CONSTRAINT PK_MyCustomers PRIMARY KEY,
  companyname  NVARCHAR(40) NOT NULL,
  contactname  NVARCHAR(30) NOT NULL,
  contacttitle NVARCHAR(30) NOT NULL,
  address      NVARCHAR(60) NOT NULL,
  city         NVARCHAR(15) NOT NULL,
  region       NVARCHAR(15) NULL,
  postalcode   NVARCHAR(10) NULL,
  country      NVARCHAR(15) NOT NULL,
  phone        NVARCHAR(24) NOT NULL,
  fax          NVARCHAR(24) NULL );
```

3. Напишите инструкцию UPDATE, которая перезаписывает значения неключевых столбцов в таблице Sales.MyCustomers значениями из соответствующих строк в таблице Sales.Customers. Ваш код должен выглядеть следующим образом:

```
UPDATE TGT
  SET TGT.companyname = SRC.companyname,
      TGT.contactname = SRC.contactname,
      TGT.contacttitle = SRC.SRC.contacttitle,
      TGT.address = SRC.SRC.address,
      TGT.city = SRC.city,
      TGT.region = SRC.region,
      TGT.postalcode = SRC.postalcode,
```

```
TGT.country      = SRC.country,
TGT.phone        = SRC.phone,
TGT.fax          = SRC.fax,
FROM Sales.MyCustomers AS TGT
INNER JOIN Sales.Customers AS SRC
ON TGT.custid = SRC.custid;
```

Задание 2. Обновление данных с помощью обобщенного табличного выражения

В этом задании вам нужно выполнить обновление данных с помощью обобщенного табличного выражения (CTE).

1. Вам нужно выполнить ту же задачу, что и в п. 3 *задания 1*, а именно обновить значения неключевых столбцов в таблице `Sales.MyCustomers` значениями из соответствующих строк в таблице `Sales.Customers`. Но на этот раз вы хотите изучить данные, подлежащие модификации, до того, как в действительности выполните обновление. Выполните задачу, используя обобщенное табличное выражение. Ваш код должен выглядеть следующим образом:

```
WITH C AS
(SELECT
    TGT.companyname AS tgt_companyname, SRC.companyname AS src_companyname,
    TGT.contactname AS tgt_contactname, SRC.contactname AS src_contactname,
    TGT.contacttitle AS tgt_contacttitle, SRC.contacttitle AS
src_contacttitle,
    TGT.address      AS tgt_address,     SRC.address      AS src_address,
    TGT.city         AS tgt_city,        SRC.city         AS src_city,
    TGT.region       AS tgt_region,      SRC.region       AS src_region,
    TGT.postalcode   AS tgt_postalcode,  SRC.postalcode   AS src_postalcode,
    TGT.country      AS tgt_country,    SRC.country      AS src_country,
    TGT.phone        AS tgt_phone,      SRC.phone        AS src_phone,
    TGT.fax          AS tgt_fax,        SRC.fax          AS src_fax,
FROM Sales.MyCustomers AS TGT
INNER JOIN Sales.Customers AS SRC
ON TGT.custid = SRC.custid
)
UPDATE C
SET tgt_companyname = src_companyname,
    tgt_contactname = src_contactname,
    tgt_contacttitle = src_contacttitle,
    tgt_address      = src_address,
    tgt_city         = src_city,
    tgt_region       = src_region,
    tgt_postalcode   = src_postalcode,
    tgt_country      = src_country,
    tgt_phone        = src_phone,
    tgt_fax          = src_fax;
```

Вы можете использовать внутренний запрос `SELECT` с объединением как до выполнения обновления, так и после него, чтобы убедиться, что получили желаемый результат.

Резюме занятия

- Язык T-SQL поддерживает стандартную инструкцию `UPDATE`, а также несколько расширений к стандарту.
- Вы можете изменить данные в одной таблице, используя данные другой таблицы, с помощью инструкции `UPDATE` на основе объединений. Но следует помнить, что если несколько исходных строк соответствуют одной целевой строке, обновление не приведет к ошибке; оно будет недетерминированным. В общем случае следует избегать таких обновлений.
- T-SQL поддерживает обновление данных с помощью табличных выражений. Эта возможность удобна, когда вы хотите иметь возможность увидеть результат запроса до фактического обновления данных. Также это удобно, когда нужно изменить строки с помощью выражений, которые обычно запрещены в предложении `SET`, например оконных функций.
- Если вам нужно модифицировать строку и запросить результат этого изменения, можно использовать специализированную инструкцию `UPDATE` с переменной, которая может выполнить это за одно обращение к строке.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Как вы можете модифицировать значение столбца в целевой строке и получить результат этой модификации за одно обращение к строке?
 - A. Используя инструкцию `UPDATE` на основе соединения.
 - B. Используя инструкцию `UPDATE` на основе табличного выражения.
 - C. Используя инструкцию `UPDATE` с переменными.
 - D. Результат невозможно получить с помощью только одного обращения к строке.
2. Какие преимущества имеет использование инструкции `UPDATE` с помощью соединений? (Выберите все подходящие варианты.)
 - A. Можно фильтровать строки, чтобы выполнить обновление с использованием информации в связанных строках другой таблицы.
 - B. Можно обновить несколько таблиц с помощью одной инструкции.
 - C. Вы можете получить информацию из соответствующих строк одной таблицы, чтобы использовать в исходных выражениях в предложении `SET`.

- D. Вы можете использовать данные из нескольких исходных строк, которые соответствуют одной целевой строке, для обновления данных в целевой строке.
3. Как можно обновить таблицу, задавая столбцу результат оконной функции?
- A. Используя инструкцию UPDATE на основе соединения.
- B. Используя инструкцию UPDATE на основе табличного выражения.
- C. Используя инструкцию UPDATE с переменными.
- D. Цель не может быть достигнута.

Занятие 3. Удаление данных

Язык T-SQL поддерживает две инструкции, которые можно использовать для удаления строк из таблицы: `DELETE` и `TRUNCATE`. В этом занятии описаны эти инструкции, разница между ними и различные аспекты работы с ними.

Изучив материал этого занятия, вы сможете:

- ✓ Использовать инструкции `DELETE` и `TRUNCATE` для удаления строк из таблицы
- ✓ Использовать инструкцию `DELETE` на основе соединений
- ✓ Использовать инструкцию `DELETE` на основе табличных выражений

Продолжительность занятия — 30 минут.

Демонстрационные данные

В этом разделе применяются те же демонстрационные данные, которые использовались в занятии 2. Напоминаем, что применяются таблицы `Sales.MyCustomers`, `Sales.MyOrders` и `Sales.MyOrderDetails`, изначально созданные как копии таблиц `Sales.Customers`, `Sales.Orders` и `Sales.OrderDetails` соответственно. Используйте следующий код для создания этих таблиц и заполнения их учебными данными:

```
IF OBJECT_ID('Sales.MyOrderDetails', 'U') IS NOT NULL
    DROP TABLE Sales.MyOrderDetails;
IF OBJECT_ID('Sales.MyOrders', 'U') IS NOT NULL
    DROP TABLE Sales.MyOrders;
IF OBJECT_ID('Sales.MyCustomers', 'U') IS NOT NULL
    DROP TABLE Sales.MyCustomers;

SELECT * INTO Sales.MyCustomers FROM Sales.Customers;
ALTER TABLE Sales.MyCustomers
    ADD CONSTRAINT PK_MyCustomers PRIMARY KEY(custid);

SELECT * INTO Sales.MyOrders FROM Sales.Orders;
ALTER TABLE Sales.MyOrders
    ADD CONSTRAINT PK_MyOrders PRIMARY KEY(orderid);
SELECT * INTO Sales.MyOrderDetails FROM Sales.OrderDetails;
ALTER TABLE Sales.MyOrderDetails
    ADD CONSTRAINT PK_MyOrderDetails PRIMARY KEY(orderid, productid);
```

Инструкция *DELETE*

С помощью инструкции *DELETE* можно удалять строки из таблицы. При желании можно указать предикат для того, чтобы ограничить число удаляемых строк. В общем случае инструкции *DELETE* имеет следующий вид:

```
DELETE FROM <table>
WHERE <predicate>;
```

Если вы не укажете предикат, будут удалены все строки из целевой таблицы. Как и в случае неуточненных обновлений, следует соблюдать особую осторожность относительно случайного удаления всех строк, используя только часть *DELETE* инструкции и опуская часть *WHERE*.

Следующий пример удаляет все строки заказов, содержащие идентификатор продукта (*productid*), равный 11, из таблицы *Sales.MyOrderDetails*.

```
DELETE FROM Sales.MyOrderDetails
WHERE productid = 11;
```

При выполнении этого кода SQL Server возвращает следующее сообщение, указывая, что удалено 38 строк.

```
(38 row(s) affected)
```

Таблицы, используемые в данной главе в качестве примеров, небольшие, но в реальной среде объемы данных, скорее всего, будут значительно больше. Инструкция *DELETE* выполняется с полным протоколированием, и, следовательно, удаление большого количества данных может требовать много времени для своего завершения. Такие объемные удаления могут приводить к значительному увеличению журнала транзакций. Они также могут приводить к укрупнению блокировок, а это означает, что SQL Server укрупняет блокировки мелких фрагментов данных, такие как блокировки строк, до полной блокировки таблицы. Такое укрупнение блокировок может приводить к блокированию доступа ко всем данным таблицы для других процессов.

Для предотвращения названных выше проблем следует разбить большой процесс удаления на меньшие блоки. Этого можно достичь с помощью инструкции *DELETE*, использующей параметр *TOP*, который ограничивает количество задействованных строк в цикле. Далее приведен пример реализации такого решения.

```
WHILE 1 = 1
BEGIN
    DELETE TOP (1000) FROM Sales.MyOrderDetails
    WHERE productid = 12;

    IF @@rowcount < 1000 BREAK;
END
```

Как видите, в коде используется бесконечный цикл (условие *WHILE 1 = 1* всегда имеет значение "истина"). В каждой итерации инструкция *DELETE* с параметром *TOP* ограничивает число затрагиваемых строк до значения, не превышающего 1000, за

один раз. Затем инструкция `IF` проверяет, действительно ли количество обрабатываемых строк менее 1000; в таком случае последняя итерация удаляет последний блок выбранных строк. После удаления последнего блока строк цикл прекращается. В наших учебных данных выбрано всего 14 строк. Поэтому если вы запустите этот код, он выполнится за один проход; затем произойдет выход из цикла и будет возвращено следующее сообщение:

```
(14 row(s) affected)
```

Но при очень большом количестве выбранных строк, скажем, в много миллионов, такого решения лучше избегать.

Инструкция `TRUNCATE`

Инструкция `TRUNCATE` удаляет все строки в целевой таблице. В отличие от инструкции `DELETE`, она не имеет дополнительного фильтра, поэтому удаляется либо все, либо ничего. В качестве примера следующая инструкция усекает таблицу `Sales.MyOrderDetails`:

```
TRUNCATE TABLE Sales.MyOrderDetails;
```

После выполнения этой инструкции целевая таблица будет пустой.

Существует несколько важных различий между инструкцией `DELETE` и инструкцией `TRUNCATE`.

- Инструкция `DELETE` записывает в журнал транзакций значительно больше информации, чем инструкция `TRUNCATE`. В случае инструкции `DELETE`, SQL Server записывает в журнал реальные данные, которые были удалены. Для инструкции `TRUNCATE` SQL Server записывает только информацию о том, какие страницы были освобождены. В результате инструкция `TRUNCATE` выполняется существенно быстрее.
- Инструкция `DELETE` не пытается сбросить свойство идентификатора, если оно установлено для столбца в целевой таблице. Инструкция `TRUNCATE` делает это. Если вы используете инструкцию `TRUNCATE` и хотели бы не сбрасывать это свойство, вам необходимо сохранить текущее значение идентификатора в переменной (используя функцию `IDENT_CURRENT`) и восстановить сохраненное значение свойства после усечения таблицы.
- Инструкция `DELETE` поддерживается, если имеется внешний ключ, указывающий на запрашиваемую таблицу, при условии, что в ссылающейся таблице нет связанных строк. Инструкция `TRUNCATE` не разрешена, если внешний ключ указывает на таблицу — даже если в ссылающейся таблице нет связанных строк и даже если внешний ключ запрещен.
- Инструкция `DELETE` применяется к таблице, являющейся частью индексированного представления. Инструкция `TRUNCATE` в таком случае является недопустимой.
- Инструкция `DELETE` требует разрешений `DELETE` на целевую таблицу. Инструкция `TRUNCATE` требует разрешений `ALTER` на целевую таблицу.

Если необходимо удалить все строки таблицы, обычно следует использовать инструкцию TRUNCATE, потому что она значительно быстрее инструкции DELETE. Однако она требует больших разрешений и имеет больше ограничений.

Инструкция *DELETE* на основе объединений

Язык T-SQL поддерживает собственный синтаксис инструкции *DELETE* на основе объединений, подобный синтаксису *UPDATE*, описанному в занятии 2. Суть заключается в предоставлении возможности удалять строки из одной таблицы, используя информацию из соответствующих строк другой таблицы.

В качестве примера предположим, что вам нужно удалить все заказы, размещенные клиентами из США (*USA*). Страна является свойством клиента, а не заказа. Поэтому хотя целевой таблицей для инструкции *DELETE* является таблица *Sales.MyOrders*, вам необходимо изучить столбец *country* в соответствующей строке клиента в таблице *Sales.MyCustomers*. Это можно сделать с помощью инструкции *DELETE* на основе объединения следующим образом:

```
DELETE FROM O
FROM Sales.MyOrders AS O
INNER JOIN Sales.MyCustomers AS C
    ON O.custid = C.custid
WHERE C.country = N'USA';
```

Предложение *FROM*, определяющее оператор таблицы *JOIN*, логически оценивается первым. Объединение связывает заказы с соответствующими им клиентами. Затем предложение *WHERE* фильтрует только строки, в которых страна клиента — *USA*. Этот фильтр приводит к сохранению только заказов, размещенных клиентами из США. Затем предложение *DELETE* задает псевдоним той стороны объединения, которая действительно является целью для удаления — *O* для *Orders* в данном случае. Эта инструкция генерирует следующие выходные данные, указывая, что были удалены 122 строки.

```
(122 row(s) affected)
```

Можно реализовать эту же задачу, используя вложенный запрос объединения, как показано в следующем примере:

```
DELETE FROM Sales.MyOrders
WHERE EXISTS
( SELECT *
  FROM Sales.MyCustomers
 WHERE MyCustomers.custid = MyOrders.custid
   AND MyCustomers.country = N'USA');
```

Эта инструкция оптимизируется так же, как инструкция, использующая объединение, поэтому, с точки зрения производительности, не существует предпочтения одной версии по сравнению с другой. Но следует отметить, что вариант с вложенным запросом считается стандартным, тогда как версия с объединением — нет. Исходя из этого, если приоритетным считается следование стандарту, надо придерживаться

версии со встроенным запросом. Некоторым людям удобнее использовать для решения подобных задач объединение, другим — вложенный запрос; это дело вкуса.

Инструкция *DELETE* с табличными выражениями

Как и в случае с обновлением данных, язык T-SQL поддерживает удаление строк с помощью табличных выражений. Идея в том, чтобы использовать табличное выражение, такое как обобщенное табличное выражение или производная таблица, для определения строк, которые вы хотите удалить, и затем выполнить инструкцию *DELETE* к табличному выражению. Разумеется, строки удаляются из базовой таблицы.

В качестве примера предположим, что вы хотите удалить 100 самых старых заказов (основываясь на сортировке по *orderdate*, *orderid*). Инструкция *DELETE* поддерживает использование параметра *TOP* напрямую, но она не поддерживает предложение *ORDER BY*. Так что вы не можете контролировать, какие строки захватит фильтр *TOP*. Чтобы обойти это, можно определить табличное выражение на основе запроса *SELECT* с параметром *TOP* и предложение *ORDER BY*, контролирующее, какие строки отфильтровываются. Затем можно применить инструкцию *DELETE* к этому табличному выражению. Вот как выглядит полный код для выполнения этой задачи:

```
WITH OldestOrders AS
( SELECT TOP (100) *
  FROM Sales.MyOrders
 ORDER BY orderdate, ordered )
DELETE FROM OldestOrders;
```

Этот код генерирует следующий результат, означающий, что было удалено 100 строк.

```
(100 row(s) affected)
```

По окончании запустите следующий код для очистки данных.

```
IF OBJECT_ID('Sales.MyOrderDetails', 'U') IS NOT NULL
    DROP TABLE Sales.MyOrderDetails;
IF OBJECT_ID('Sales.MyOrders', 'U') IS NOT NULL
    DROP TABLE Sales.MyOrders;
IF OBJECT_ID('Sales.MyCustomers', 'U') IS NOT NULL
    DROP TABLE Sales.MyCustomers;
```

Контрольные вопросы

1. Какие строки будут удалены из целевой таблицы с помощью инструкции *DELETE*, если не используется предложение *WHERE*?
2. Что может служить альтернативой инструкции *DELETE* без предложения *WHERE*?

Ответы на контрольные вопросы

1. Все строки целевой таблицы.
2. Инструкция TRUNCATE. Но между ними существуют различия, которые следует принимать во внимание.

ПРАКТИКУМ Удаление и усечение данных

В этом практикуме вы будете проверять ваши знания об удалении данных с помощью инструкций DELETE и TRUNCATE.

Задание 1. Удаление данных с помощью соединений

В этом задании вы будете удалять данные с помощью соединений.

1. Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
2. Запустите следующий код для создания таблиц Sales.MyCustomers и Sales.MyOrders, как исходных копий таблиц Sales.Customers и Sales.MyOrders соответственно.

```
IF OBJECT_ID('Sales.MyOrders', 'U') IS NOT NULL
    DROP TABLE Sales.MyOrders;
IF OBJECT_ID('Sales.MyCustomers', 'U') IS NOT NULL
    DROP TABLE Sales.MyCustomers;

SELECT * INTO Sales.MyCustomers FROM Sales.Customers;
ALTER TABLE Sales.MyCustomers
    ADD CONSTRAINT PK_MyCustomers PRIMARY KEY(custid);
SELECT * INTO Sales.MyOrders FROM Sales.Orders;
ALTER TABLE Sales.MyOrders
    ADD CONSTRAINT PK_MyOrders PRIMARY KEY(orderid);

ALTER TABLE Sales.MyOrders
    ADD CONSTRAINT FK_MyOrders_MyCustomers
        FOREIGN KEY(custid) REFERENCES Sales.MyCustomers(custid);
```

3. Напишите инструкцию DELETE, которая удаляет строки из таблицы Sales.MyCustomers, если у клиента нет связанных с ним заказов в таблице Sales.MyOrders. Для реализации этой задачи используйте инструкцию DELETE на основе объединения. Ваше решение должно выглядеть следующим образом:

```
DELETE FROM TGT
FROM Sales.MyCustomers AS TGT
    LEFT OUTER JOIN Sales.MyOrders AS SRC
        ON TGT.custid = SRC.custid
    WHERE SRC.orderid IS NULL;
```

4. Используйте следующий запрос, чтобы подсчитать клиентов, оставшихся в таблице.

```
SELECT COUNT(*) AS cnt FROM Sales.MyCustomers;
```

Вы получите 89.

Задание 2. Усечение данных

В этом задании вы будете выполнять усечение данных.

1. Используйте инструкцию TRUNCATE для очистки сначала таблицы Sales.MyOrders, а затем таблицы Sales.MyCustomers. Ваш код должен выглядеть следующим образом:

```
TRUNCATE TABLE Sales.MyOrders;
TRUNCATE TABLE Sales.MyCustomers;
```

Вторая инструкция завершится ошибкой:

```
Msg 4712, Level 16, State 1, Line 1
```

```
Cannot truncate table 'Sales.MyCustomers' because it is being referenced by
a FOREIGN KEY constraint.
```

2. Объясните причину ошибки и предложите решение для ее устранения.

Ошибка произошла, т. к. инструкция TRUNCATE запрещена, когда на целевую таблицу ссылаются посредством ограничения внешнего ключа, даже если в ссылающейся таблице нет связанных строк. Чтобы решить проблему, надо удалить внешний ключ, выполнить усечение целевой таблицы и затем создать внешний ключ снова.

```
ALTER TABLE Sales.MyOrders
DROP CONSTRAINT FK_MyOrders_MyCustomers;

TRUNCATE TABLE Sales.MyCustomers;

ALTER TABLE Sales.MyOrders
ADD CONSTRAINT FK_MyOrders_MyCustomers
FOREIGN KEY(custid) REFERENCES Sales.MyCustomers(custid);
```

3. По окончании запустите следующий код для очистки данных:

```
IF OBJECT_ID('Sales.MyOrders', 'U') IS NOT NULL
DROP TABLE Sales.MyOrders;
IF OBJECT_ID('Sales.MyCustomers', 'U') IS NOT NULL
DROP TABLE Sales.MyCustomers;
```

Резюме занятия

- С помощью инструкции DELETE можно удалять строки из таблицы и при желании ограничивать удаляемые строки, используя фильтр на основе предиката. Также можно ограничить число удаляемых строк с помощью фильтра TOP, но тогда не будет возможности контролировать, какие строки выбраны для удаления.

- С помощью инструкции TRUNCATE можно удалять все строки в целевой таблице. Эта инструкция не поддерживает фильтры. Преимущество инструкции TRUNCATE перед инструкцией DELETE заключается в том, что первая использует оптимизированное ведение журнала и поэтому работает быстрее, по сравнению с последней. Однако инструкция TRUNCATE имеет больше ограничений, чем инструкция DELETE, и требует более строгих разрешений.
- T-SQL поддерживает синтаксис DELETE на основе объединений, давая возможность удалять строки из одной таблицы, используя информацию из связанных строк в других таблицах.
- Также язык T-SQL поддерживает удаление строк с помощью табличных выражений, таких как обобщенные табличные выражения (CTE) и производные таблицы.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Как удалить строки из таблицы, для которой вычисление ROW_NUMBER равно 1?
 - A. Вы сошлетесь на функцию ROW_NUMBER в предложении WHERE инструкции DELETE.
 - B. Вы будете использовать табличное выражение, такое как обобщенное табличное выражение или производная таблица, вычисляющее столбец на основе функции ROW_NUMBER, и затем примените инструкцию DELETE с фильтром к табличному выражению.
 - C. Вы будете использовать табличное выражение, такое как обобщенное табличное выражение или производная таблица, вычисляющее столбец на основе функции ROW_NUMBER, и затем примените инструкцию TRUNCATE с фильтром к табличному выражению.
 - D. Задача не имеет решения.
2. Что из нижеперечисленного применимо к инструкции DELETE? (Выберите все подходящие варианты.)
 - A. Эта инструкция записывает больше данных в журнал транзакций, чем инструкция TRUNCATE.
 - B. Эта инструкция сбрасывает свойство IDENTITY.
 - C. Эта инструкция запрещена, если внешний ключ указывает на целевую таблицу.
 - D. Эта инструкция запрещена, если существует индексированное представление на основе целевой таблицы.
3. Что из нижеперечисленного применимо к инструкции TRUNCATE? (Выберите все подходящие варианты.)
 - A. Эта инструкция записывает больше данных в журнал транзакций, чем инструкция DELETE.

- В. Эта инструкция сбрасывает свойство IDENTITY.
- С. Эта инструкция запрещена, если внешний ключ указывает на целевую таблицу.
- Д. Эта инструкция запрещена, если существует индексированное представление на основе целевой таблицы.

Упражнения

В следующих упражнениях вы примените полученные знания о вставке, обновлении и удалении данных. Ответы на вопросы можно найти в *приложении "Ответы" в конце книги*.

Упражнение 1. Использование модификаций, поддерживающих оптимизированное ведение журнала

Вы являетесь консультантом в отделе информационных технологий в большой компании розничной торговли. В компании каждую ночь запускается процесс, который сначала удаляет все строки в таблице, используя инструкцию `DELETE`, и затем заполняет эту таблицу данными результата запроса к другим таблицам. Результат содержит несколько десятков миллионов строк. Процесс работает очень медленно. Вас просят дать рекомендации для улучшения ситуации.

1. Дайте рекомендации по усовершенствованию части процесса, выполняющей удаление данных.
2. Дайте рекомендации по усовершенствованию части процесса, выполняющей вставку данных.

Упражнение 2. Усовершенствование процесса обновления данных

Та же компания, которая наняла вас для консультации по поводу неэффективности ночного процесса обработки данных в первом упражнении, снова пригласила вас на работу. Ее сотрудники просят вашего совета в отношении следующего процесса обновления данных.

1. В базе данных имеется таблица, содержащая около 100 млн строк. Около трети существующих строк должно быть обновлено. Можете ли вы дать рекомендации по поводу того, как организовать обновление данных, не вызывая при этом ненужных проблем с производительностью в системе?
2. Имеется инструкция `UPDATE`, которая модифицирует строки в одной таблице, на основе информации из связанных строк в другой таблице. Инструкция `UPDATE` в настоящее время использует отдельный вложенный запрос для каждого столбца, который должен быть модифицирован, получая значение соответствующего столбца из связанной с ним строки в исходной таблице. Инструкция также использует вложенный запрос для фильтрации только строк, которые имеют соот-

ветствия в исходной таблице. Процесс происходит очень медленно. Можете ли вы предложить способы его улучшения?

Рекомендуемые упражнения

Для успешного освоения экзаменационных тем, представленных в данной главе, выполните следующие задания.

Сравнение инструкций *DELETE* и *TRUNCATE*

Это практическое задание поможет вам понять существенную разницу в производительности между инструкциями *DELETE* и *TRUNCATE*. Используйте ваши знания о перекрестных соединениях, об инструкциях *SELECT INTO*, *DELETE* и *TRUNCATE* для того, чтобы увидеть разницу в производительности между удалением с полным протоколированием и простым протоколированием.

- **Задание 1.** Первая задача в проверке производительности, которую вы собираетесь запустить, — подготовить тестовые данные. Для этого вы используете инструкцию *SELECT INTO*. Помните, чтобы инструкция *SELECT INTO* выигрывала от минимального протоколирования, необходимо настроить модель восстановления базы данных на простое или неполное протоколирование. Вам нужно создать тестовую таблицу и заполнить ее достаточным количеством данных для тестирования производительности. Нескольких миллионов строк должно быть достаточно. Для этого вы можете выполнить перекрестное соединение между одной из таблиц в учебной базе данных TSQL2012 (например, *Sales.Orders*) и таблицей *dbo.Nums*. Вы можете применить фильтр к столбцу *Nums.n*, чтобы контролировать число строк, которое должно получиться в результате. Если установить фильтр *n <= 2000*, вы получите 2000 копий другой таблицы в соединении. Используйте инструкцию *SELECT INTO* для создания целевой таблицы и заполните ее результатами запроса.
- **Задание 2.** Удалите все строки из целевой таблицы с помощью инструкции *DELETE* и обратите внимание на то, сколько времени потребовалось на выполнение инструкции.
- **Задание 3.** Создайте снова те же тестовые данные. Затем используйте инструкцию *TRUNCATE* для удаления всех строк из целевой таблицы. Сравните производительность обоих методов.

ГЛАВА 11

Другие виды модификации данных

Темы экзамена

- Модификация данных.
 - Модификация данных с помощью инструкций `INSERT`, `UPDATE` и `DELETE`.
 - Комбинирование наборов данных.

В главе 10 рассматривались три основные инструкции модификации данных: `INSERT`, `UPDATE` и `DELETE`. Данная глава посвящена дополнительным возможностям модификации данных, таким как объект последовательности и свойство столбца `IDENTITY`, инструкция `MERGE` и параметр `OUTPUT`.

ПРЕЖДЕ ВСЕГО

Для полного освоения этой главы необходимо иметь:

- опыт работы с SQL Server Management Studio (SSMS);
- некоторый опыт в написании кода на языке T-SQL;
- понимание типов данных;
- понимание принципов комбинирования наборов данных;
- понимание инструкций модификации данных;
- доступ к экземпляру SQL Server 2012 с установленной учебной базой данных `TSQL2012`.

Занятие 1. Использование объекта последовательности и свойства столбца `IDENTITY`

Свойство столбца `IDENTITY` и объект последовательности — это свойства, которые можно использовать, чтобы автоматически генерировать последо-



вательность чисел. Эти числа, как правило, используются как суррогатные ключи в таблицах для таких сущностей, как заказы, продукты, сотрудники и клиенты. Свойство `IDENTITY` — это очень старая функциональность SQL Server, имеющая ряд недостатков и ограничений. Объект последовательности был введен в SQL Server 2012 и преодолевает многие недостатки свойства `IDENTITY`. В начале этого занятия мы рассмотрим свойство `IDENTITY` и затем опишем объект последовательности.

Изучив материал этого занятия, вы сможете:

- ✓ Использовать свойство столбца `IDENTITY` и объект последовательности
- ✓ Описать преимущества объекта последовательности

Продолжительность занятия — 40 минут.

Использование свойства столбца `IDENTITY`

Свойство `IDENTITY` — это свойство столбца в таблице. Оно автоматически присваивает значение столбцу в процессе вставки данных. Его можно определить для столбцов с любым числовым типом данных с масштабом 0, т. е. все целочисленные типы, а также `NUMERIC/DECIMAL` с масштабом 0. При определении этого свойства можно в качестве необязательных параметров задать начальное значение и приращение. Если это не сделано, принимаются значения по умолчанию 1 и 1. Только один столбец в таблице может иметь свойство `IDENTITY`.

COBET Подготовка к экзамену

На экзамене, когда нужно выбрать ответ из нескольких предложенных вариантов, сначала важно отбросить заведомо неправильные ответы. Например, пусть вы получили вопрос и несколько вариантов ответа, содержащих примеры кода. Предположим, один из ответов содержит код, в котором определена таблица, имеющая два столбца со свойством `IDENTITY` или два ограничения `PRIMARY KEY`. Эти ответы можно быстро исключить из числа правильных. Таким образом, вы сможете потратить больше времени на оставшиеся ответы.

Далее приведен пример кода, который создает таблицу с именем `Sales.MyOrders`, имеющую столбец `orderid`, который имеет свойство `IDENTITY` с начальным значением, равным 1, и приращением, также равным 1.

```
USE TSQSL2012;
IF OBJECT_ID('Sales.MyOrders') IS NOT NULL DROP TABLE Sales.MyOrders;
GO
CREATE TABLE Sales.MyOrders
( orderid  INT NOT NULL IDENTITY(1, 1)
  CONSTRAINT PK_MyOrders_orderid PRIMARY KEY,
  custid   INT NOT NULL
  CONSTRAINT CHK_MyOrders_custid CHECK(custid > 0),
  empid    INT NOT NULL
  CONSTRAINT CHK_MyOrders_empid CHECK(empid > 0),
  orderdate DATE NOT NULL );
```

При вставке строки в таблицу не нужно указывать значение для столбца `IDENTITY`, поскольку он получает свое значение автоматически. Например, следующий код выполняет вставку трех строк и не указывает значения для столбца `orderid`.

```
INSERT INTO Sales.MyOrders(custid, empid, orderdate) VALUES
```

```
(1, 2, '20120620'),  
(1, 3, '20120620'),  
(2, 2, '20120620');
```

```
SELECT * FROM Sales.MyOrders;
```

После вставки запрос возвратит следующий результат:

orderid	custid	empid	orderdate
1	1	2	2012-06-20
2	1	3	2012-06-20
3	2	2	2012-06-20

В случаях, когда при вставке строк в таблицу вы хотите указать собственные значения для столбца `orderid`, необходимо установить параметр сеанса, который называется `SET IDENTITY_INSERT <table> to ON`. Обратите внимание, не существует параметра, который можно было бы установить для обновления столбца `IDENTITY`.

В языке T-SQL имеется несколько функций, которые можно использовать, чтобы запросить последнее сгенерированное значение идентификатора, например, если оно вам необходимо, когда вы выполняете вставку связанных строк в другую таблицу:

- функция `SCOPE_IDENTITY` возвращает последнее значение идентификатора, сгенерированное в вашем сеансе в данной области;
- функция `@@IDENTITY` возвращает последнее значение идентификатора, сгенерированное в вашем сеансе независимо от области;
- функция `IDENT_CURRENT` принимает в качестве входа таблицу и возвращает последнее значение идентификатора, сгенерированное во входной таблице независимо от сессии.

В качестве примера, следующий код запрашивает все три функции в той же сессии, в которой выполнялась инструкция `INSERT`.

```
SELECT  
    SCOPE_IDENTITY() AS SCOPE_IDENTITY,  
    @@IDENTITY AS @@IDENTITY,  
    IDENT_CURRENT('Sales.MyOrders') AS IDENT_CURRENT;
```

Поскольку никаких действий после последней инструкции `INSERT` не выполнялось ни в данном сеансе, ни в каком-либо другом, все три функции возвращают одинаковые значения.

```
SCOPE_IDENTITY  @@IDENTITY  IDENT_CURRENT  
-----  
3             3            3
```

А теперь откройте новое окно запроса и выполните запрос снова. На этот раз вы получите следующий результат:

```
SCOPE_IDENTITY() @IDENTITY IDENT_CURRENT
-----
NULL           NULL        3
```

Так как вы запускаете запрос в другом сеансе, чем тот, в котором было сгенерировано значение идентификатора, функции SCOPE_IDENTITY и @@IDENTITY возвращают значения NULL. Функция IDENT_CURRENT возвращает последнее значение, сгенерированное во входной таблице, независимо от сеанса.

Что касается разницы между функциями SCOPE_IDENTITY и @@IDENTITY, предположим, что у вас есть хранимая процедура P1 с тремя инструкциями:

- с инструкцией INSERT, которая генерирует новое значение идентификатора;
- с вызовом хранимой процедуры P2, в которой также есть инструкция INSERT, генерирующая новое значение идентификатора;
- с инструкцией, которая запрашивает функции SCOPE_IDENTITY и @@IDENTITY.

Функция SCOPE_IDENTITY возвратит значение, сгенерированное процедурой P1 (тот же самый сеанс и область действия). Функция @@IDENTITY возвратит значение, сгенерированное процедурой P2 (тот же сеанс независимо от scope).

Если вам требуется удалить все строки из таблицы, вы должны знать о существенной разнице между выполнением такого удаления с помощью инструкции DELETE без предложения WHERE и использования для этой же цели инструкции TRUNCATE. Первая не влияет на текущее значение идентификатора, тогда как последняя сбрасывает его до исходного базового значения. Например, выполните следующий код, чтобы очистить таблицу Sales.MyOrders с помощью инструкции TRUNCATE.

```
TRUNCATE TABLE Sales.MyOrders;
```

Теперь запросите текущее значение идентификатора в таблице.

```
SELECT IDENT_CURRENT('Sales.MyOrders') AS [IDENT_CURRENT];
```

Вы получите в результате 1. Чтобы повторно указать текущее значение идентификатора, используйте команду DBCC CHECKIDENT следующим образом:

```
DBCC CHECKIDENT('Sales.MyOrders', RESEED, 4);
```

Чтобы убедиться, что значение было переустановлено, запустите инструкцию INSERT и выполните запрос к таблице.

```
INSERT INTO Sales.MyOrders(custid, empid, orderdate)
VALUES(2, 2, '20120620');
SELECT * FROM Sales.MyOrders;
```

Вы получите следующий результат:

```
orderid      custid      empid      orderdat
-----      -----      -----      -----
4            2            2          2012-06-20
```

Важно понимать, что свойство `IDENTITY` может гарантировать не все. Оно не может гарантировать уникальность. Помните, что вы можете ввести явные значения, если включить параметр `IDENTITY_INSERT`. Также вы можете переустановить значение свойства. Для обеспечения уникальности следует использовать такие ограничения, как `PRIMARY KEY` или `UNIQUE`.

Также свойство `IDENTITY` не гарантирует, что не будет разрывов между значениями. Если инструкция `INSERT` закончится неудачно, текущее значение идентификатора не возвратится к оригинальному значению, поэтому неиспользованное значение будет потеряно. Очередная вставка приведет к увеличению значения, следующему за неиспользованным. Чтобы убедиться в этом, выполните такую инструкцию `INSERT`:

```
INSERT INTO Sales.MyOrders(custid, empid, orderdate)
VALUES(3, -1, '20120620');
```

Эта инструкция нарушает ограничение `CHECK`, определенное для таблицы с помощью предиката `empid > 0`, и код генерирует следующую ошибку.

```
Msg 547, Level 16, State 0, Line 1
The INSERT statement conflicted with the CHECK constraint "CHK_MyOrders_empid".
The conflict occurred in database "TSQL2012", table "Sales.MyOrders", column
'empid'. The statement has been terminated.
```

Свойство `IDENTITY` генерирует новое значение идентификатора, равное 5, для инструкции `INSERT`; однако SQL Server не отменил изменение текущего значения идентификатора из-за сбоя. Теперь выполните другую инструкцию `INSERT`.

```
INSERT INTO Sales.MyOrders(custid, empid, orderdate)
VALUES(3, 1, '20120620');
```

На этот раз вставка выполнится успешно. Выполните запрос к таблице.

```
SELECT * FROM Sales.MyOrders;
```

Вы получите следующие выходные данные:

orderid	custid	empid	orderdate
4	2	2	2012-06-20
6	3	1	2012-06-20

Обратите внимание, значение 5, сгенерированное для закончившейся сбоем инструкции `INSERT`, не было использовано, и теперь вы имеете разрыв в значениях. Поэтому свойство столбца `IDENTITY` не может считаться подходящим решением для упорядочивания данных, когда разрывы недопустимы. Примером могут служить системы выставления счетов, т. к. разрывы между номерами счетов недопустимы. В подобных случаях необходимо использовать альтернативное решение, такое как сохранение последнего использованного в таблице значения.

Свойство `IDENTITY` не поддерживает циклическое повторение. Это означает, что после того, как вы достигнете максимального для этого типа данных значения, следующая вставка потерпит неудачу из-за ошибки переполнения. Чтобы решить эту проблему, нужно повторно установить текущее значение идентификатора до выполнения такой попытки.

Использование объекта последовательности

В SQL Server 2012 был введен объект последовательности. В отличие от свойства столбца `IDENTITY`, последовательность — это независимый объект базы данных. Объект последовательности лишен многих ограничений, присущих свойству `IDENTITY`, к которым относятся следующие.

- Свойство `IDENTITY` привязано к конкретному столбцу в определенной таблице. Вы не можете удалить свойство, уже существующее у столбца, а также добавить это свойство существующему столбцу. Столбец должен быть определен с этим свойством.
- Иногда нужно, чтобы ключи не конфликтовали в разных таблицах, но свойство `IDENTITY` определяется на уровне таблицы.
- Иногда требуется сгенерировать значение до его использования. Это невозможно при использовании свойства `IDENTITY`. Необходимо вставить строку и только потом получить новое значение с помощью функции.
- Нельзя обновить столбец `IDENTITY`.
- Свойство `IDENTITY` не поддерживает циклическое повторение.
- Инструкция `TRUNCATE` сбрасывает свойство идентификатора.

Объект последовательности не страдает от этих ограничений. Данный раздел объясняет, как работать с объектом последовательности и показывает, каким образом он избавлен от ограничений, присущих свойству `IDENTITY`.

Объект последовательности создается как независимый объект базы. Он не привязан к определенному столбцу в определенной таблице. Для создания последовательности можно использовать команду `CREATE SEQUENCE`. Как минимум, следует указать имя объекта:

```
CREATE SEQUENCE <schema>.<object>;
```

Как и для свойства `IDENTITY`, поддерживаются все числовые типы данных с масштабом 0. Но если тип данных не указан явно, SQL Server по умолчанию установит значение `BIGINT`. Если нужен другой тип данных, его надо запросить явно, добавив после имени последовательности `AS <type>`.

Вы можете установить ряд свойств, каждое из которых имеет значение по умолчанию на случай, если не представлено другое. Далее приведены некоторые из этих свойств с их значениями по умолчанию:

- `INCREMENT BY` — значение приращения, по умолчанию равно 1;
- `MINVALUE` — минимальное поддерживаемое значение. Значением по умолчанию является минимальное значение, установленное для указанного типа данных. Например, для типа `INT` это будет значение `-2 147 483 648`;
- `MAXVALUE` — максимальное поддерживаемое значение; значением по умолчанию является максимальное значение, установленное для указанного типа данных;

- CYCLE | NO CYCLE — определяет, разрешено ли для последовательности циклическое повторение; значение по умолчанию — NO CYCLE;
- START WITH — начальное значение последовательности; по умолчанию устанавливается в MINVALUE для возрастающей последовательности (положительное приращение) и MAXVALUE для убывающей последовательности.

Далее приведен пример для определения последовательности, которую можно использовать для генерации идентификаторов заказов.

```
CREATE SEQUENCE Sales.SeqOrderIDs AS INT  
    MINVALUE 1  
    CYCLE;
```

Обратите внимание, это определение устанавливает минимальное значение равным 1 (и, следовательно, начальное значение тоже равно 1) и указывает, что последовательность должна разрешать циклическое повторение. Как правило, если нужно, чтобы последовательность начиналась с 1, используется установка свойства START WITH в 1. Однако это не изменит величину минимального значения –2 147 483 648. Это может стать проблемой, если последовательность разрешает циклическое повторение. Как только будет достигнуто последнее значение для указанного типа данных, следующим сгенерированным значением будет не 1, а –2 147 483 648. Поэтому, если нужно, чтобы последовательность генерировала только положительные значения, наиболее разумным решением будет установить параметр MINVALUE в 1. Это также неявно установит значение START WITH в 1.

Следует отметить, что в реальной жизни, как правило, вы не будете разрешать последовательности генерировать идентификаторы заказов с циклическим повторением, но в данном случае мы определили последовательность с циклическим повторением исключительно с демонстрационной целью.

Чтобы запросить новое значение из последовательности, следует использовать функцию NEXT VALUE FOR <sequence name>. Например, выполните следующий код три раза:

```
SELECT NEXT VALUE FOR Sales.SeqOrderIDs;
```

Вы получите значения 1, 2 и 3. Эту функцию можно вызывать из инструкций INSERT VALUES и INSERT SELECT, из предложения SET инструкции UPDATE, из присвоения переменной, из выражения ограничения DEFAULT и других мест. Далее в этом занятии приведены примеры ее использования.

Невозможно изменить тип данных существующей последовательности, но с помощью команды ALTER SEQUENCE можно изменить все ее свойства. Например, если вам необходимо изменить текущее значение, сделать это можно, используя следующий код:

```
ALTER SEQUENCE Sales.SeqOrderIDs RESTART WITH 1;
```

Чтобы увидеть собственными глазами, как используются значения последовательности при вставке строк в таблицу, повторно создайте таблицу Sales.MyOrders, выполнив следующий код:

```

IF OBJECT_ID('Sales.MyOrders') IS NOT NULL DROP TABLE Sales.MyOrders;
GO

CREATE TABLE Sales.MyOrders
( orderid INT NOT NULL
    CONSTRAINT PK_MyOrders_orderid PRIMARY KEY,
  custid INT NOT NULL
    CONSTRAINT CHK_MyOrders_custid CHECK(custid > 0),
  empid INT NOT NULL
    CONSTRAINT CHK_MyOrders_empid CHECK(empid > 0),
  orderdate DATE NOT NULL );

```

Обратите внимание, на этот раз столбец `orderid` не имеет свойства `IDENTITY`.

Далее приведен пример использования функции `NEXT VALUE FOR` в предложении `INSERT VALUES`, которое вставляет три строки в таблицу.

```

INSERT INTO Sales.MyOrders(orderid, custid, empid, orderdate) VALUES
(NEXT VALUE FOR Sales.SeqOrderIDs, 1, 2, '20120620'),
(NEXT VALUE FOR Sales.SeqOrderIDs, 1, 3, '20120620'),
(NEXT VALUE FOR Sales.SeqOrderIDs, 2, 2, '20120620');

```

Как уже упоминалось, можно использовать эту функцию в инструкции `INSERT SELECT`. В таком случае, при желании можно добавить предложение `OVER` со списком `ORDER BY` для того, чтобы контролировать порядок, в котором значения последовательности присваиваются результирующим строкам.

```

INSERT INTO Sales.MyOrders(orderid, custid, empid, orderdate)
SELECT
    NEXT VALUE FOR Sales.SeqOrderIDs OVER(ORDER BY orderid),
    custid, empid, orderdate
FROM Sales.Orders
WHERE custid = 1;

```

Это является расширением языка T-SQL к стандарту.

Выполните запрос к таблице, чтобы посмотреть значения, сгенерированные обеими инструкциями.

```
SELECT * FROM Sales.MyOrders;
```

Вы получите следующий результат:

orderid	custid	empid	orderdate
1	1	2	2012-06-20
2	2	3	2012-06-20
3	1	2	2012-06-20
4	1	6	2007-08-25
5	1	4	2007-10-03
6	1	4	2007-10-13
7	1	1	2008-01-15
8	1	1	2008-03-16
9	1	3	2008-04-09

Также можно использовать функцию NEXT VALUE FOR в ограничении DEFAULT, что позволяет ограничению автоматически генерировать значения в процессе вставки строк. Чтобы определить такое ограничение DEFAULT для столбца `orderid`, выполните следующий код:

```
ALTER TABLE Sales.MyOrders  
    ADD CONSTRAINT DFT_MyOrders_orderid  
        DEFAULT(NEXT VALUE FOR Sales.SeqOrderIDs) FOR orderid;
```

Затем выполните следующую инструкцию `INSERT`, пропуская столбец `orderid`.

```
INSERT INTO Sales.MyOrders(custid, empid, orderdate)  
    SELECT custid, empid, orderdate  
    FROM   Sales.Orders  
    WHERE  custid = 2;
```

На этот раз значения идентификаторов заказов (order ID) были сгенерированы автоматически. Это свойство является расширением к стандарту, которое упрощает возможность выбора альтернативы свойства `IDENTITY`. В действительности та альтернатива является более гибкой, чем свойство `IDENTITY`, потому что выполняет только присвоение значения по умолчанию, если оно не было указано явно в инструкции `INSERT`.

Объект последовательности также поддерживает параметр кэширования, который проверяет, как часто текущее значение последовательности записано на диск по сравнению с записью в память. Например, последовательность с определенным для нее параметром `CACHE 100` будет выполнять запись на диск один раз на каждые 100 изменений. SQL Server сохраняет в памяти два элемента, содержащие текущее значение последовательности, а также количество оставшихся значений. Поэтому он будет записывать 100 раз только в память, и только когда эти 100 значений будут исчерпаны, он запишет на диск текущее значение плюс 100. Преимуществом является повышение производительности при выделении порядковых номеров. Но при непредвиденном завершении работы возникает риск потери порядковых номеров вплоть до размера кэша.

Следует помнить, что так же, как и свойство `IDENTITY`, объект последовательности не гарантирует отсутствие разрывов в данных. Если SQL Server создает новое значение последовательности в транзакции, и происходит сбой этой транзакции, изменение значения последовательности не отменяется. Поэтому если вы работаете с объектом последовательности, то должны согласиться с возможностью наличия разрывов.

Существует очень большая разница в производительности между использованием `NO CACHE` и `CACHE <some value>`. Если установлен параметр `NO CACHE`, SQL Server должен делать запись на диск для каждого запроса нового значения последовательности. Но с кэшированием производительность намного выше. По умолчанию в момент написания этого курса значение кэша устанавливалось равным 50, но Microsoft не публикует значение по умолчанию, оставляя за собой право менять его в будущем.

Далее приведен пример установки значения кэша равным 100.

```
ALTER SEQUENCE Sales.SeqOrderIDs  
    CACHE 100;
```

T-SQL также поддерживает хранимую процедуру `sp_sequence_get_range`, которую можно использовать для выделения целого диапазона значений последовательности требуемого размера. Нужный размер диапазона задается с помощью входного параметра `@range_size` и получает первое значение в выделенном диапазоне с помощью выходного параметра `@range_first_value`. Затем можно присваивать значения в выделенном диапазоне по своему усмотрению. Сама последовательность изменяется только один раз, когда она расширяется от своего текущего значения до текущего значения плюс `@range_size`. Подробности о прочих поддерживаемых параметрах этой хранимой процедуры можно найти в электронной документации по SQL Server 2012 в разделе "sp_sequence_get_range (Transact-SQL)" по адресу <http://msdn.microsoft.com/ru-ru/library/ff878352.aspx>.

В SQL Server имеется представление с именем `sys.sequences`, которое можно использовать для запроса свойств последовательностей, определенных в текущей базе данных.

Возвращаясь к списку ограничений свойства `IDENTITY`, о которых говорилось ранее, перечислим преимущества объекта последовательности.

- Объект последовательности не привязан к конкретному столбцу в определенной таблице. При желании можно указать новое значение с помощью ограничения `DEFAULT`. Это ограничение можно добавить к существующему столбцу или удалить у него.
- Поскольку последовательность — это независимый объект базы данных, можно применять ту же последовательность для генерации ключей, которые используются в разных таблицах. Таким образом, не будет конфликта ключей в разных таблицах.
- Можно сгенерировать значение последовательности до его использования, сохранив результат функции `NEXT VALUE FOR` в переменной.
- Можно обновлять столбцы с помощью инструкции `UPDATE`, используя результаты функции `NEXT VALUE FOR`.
- Объект последовательности поддерживает циклическое повторение.
- Инструкция `TRUNCATE` не сбрасывает текущее значение объекта последовательности, поскольку последовательность не зависит от использующих ее таблиц.

Убедитесь в том, что у вас есть таблица `Sales.MyOrders`, потому что она используется в последующих занятиях данной главы.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Сколько столбцов, имеющих свойство `IDENTITY`, может быть в таблице?
2. Как можно получить новое значение из последовательности?

Ответы на контрольные вопросы

1. Один.
2. С помощью функции NEXT VALUE FOR.

ПРАКТИКУМ Использование объекта последовательности

В этом практикуме вы будете генерировать ключи с помощью объекта последовательности.

Задание 1. Создание последовательности с параметрами по умолчанию

В этом задании вам нужно создать объект последовательности с параметрами по умолчанию и запросить его свойства.

1. Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
2. Выполните следующий код для создания последовательности dbo.Seq1. Вам нужно указать только схему и имена объектов, а все свойства последовательности будут определены по умолчанию.

```
CREATE SEQUENCE dbo.Seq1;
```

3. Чтобы запросить свойства последовательности, выполните следующий запрос к представлению sys.sequences:

```
SELECT
    TYPE_NAME(system_type_id) AS type,
    start_value, minimum_value, current_value, increment, is_cycling
FROM sys.sequences
WHERE object_id = OBJECT_ID('dbo.Seq1');
```

Этот код генерирует следующие выходные данные:

type	start_value	minimum_value
bigint	-9223372036854775808	-9223372036854775808

current_value	increment	is_cycling
-9223372036854775808	1	0

Обратите внимание, SQL Server использовал по умолчанию тип данных BIGINT. При этом наименьшее поддерживаемое этим типом данных значение ($-9\ 223\ 372\ 036\ 854\ 775\ 808$) используется как минимальное и текущее значения, а наибольшее поддерживаемое этим типом значение — как максимальное значение. Приращение равно 1, циклическое повторение отсутствует.

Задание 2. Создание последовательности со значениями, отличными от значений по умолчанию

В этом задании вы будете создавать последовательность с параметрами, отличными от значений по умолчанию.

1. Начните с последовательности `dbo.Seq1`, которую вы создали в предыдущем задании. Здесь вам нужно изменить тип данных последовательности со значениями по умолчанию `BIGINT` на `INT`, а также с начальным значением 1 и поддержкой циклического повторения. Однако, в отличие от всех прочих свойств, тип данных существующей последовательности изменить нельзя. Вам необходимо заново создать последовательность. Выполните следующий код для того, чтобы удалить и заново создать данную последовательность:

```
IF OBJECT_ID('dbo.Seq1') IS NOT NULL DROP SEQUENCE dbo.Seq1;
CREATE SEQUENCE dbo.Seq1 AS INT
    START WITH 1 CYCLE;
```

Этот код создает последовательность с типом данных `INT`, указывает 1 в качестве начального значения и задает поддержку циклического повторения. Запросите свойства этой последовательности.

```
SELECT
    TYPE_NAME(system_type_id) AS type,
    start_value, minimum_value, current_value, increment, is_cycling
FROM sys.sequences
WHERE object_id = OBJECT_ID('dbo.Seq1');
```

Вы получите на выходе следующий результат:

```
type start_value minimum_value
-----
int      1          -2147483648

current_value increment is_cycling
----- ----- -----
1           1          1
```

Обратите внимание, хотя начальное значение для последовательности было определено равным 1, минимальное значение равно наименьшему значению для указанного типа данных (`-2 147 483 648` для типа данных `INT`) по умолчанию.

2. Увидеть, что происходит после того, как вы доберетесь до максимального значения, можно, сначала изменив текущее значение последовательности на максимальное поддерживаемое данным типом данных с помощью следующего кода:

```
ALTER SEQUENCE dbo.Seq1 RESTART WITH 2147483647;
```

Затем дважды выполните следующий код:

```
SELECT NEXT VALUE FOR dbo.Seq1;
```

Сначала вы получите величину 2 147 483 647, а затем –2 147 483 648 — не 1, поскольку минимальное значение последовательности определено как –2 147 483 648.

3. Если вы хотите создать последовательность, которая выполняет циклическое повторение и поддерживает только положительные значения, вам необходимо установить свойство MINVALUE равным 1. Для этого выполните следующий код:

```
IF OBJECT_ID('dbo.Seq1') IS NOT NULL DROP SEQUENCE dbo.Seq1;
CREATE SEQUENCE dbo.Seq1 AS INT
    MINVALUE 1 CYCLE;
```

4. Запросите свойства последовательности, выполнив следующий код:

```
SELECT
    TYPE_NAME(system_type_id) AS type,
    start_value, minimum_value, current_value, increment, is_cycling
FROM sys.sequences
WHERE object_id = OBJECT_ID('dbo.Seq1');
```

Вы получите следующий результат:

```
type start_value minimum_value
-----
int      1          1

current_value increment is_cycling
----- ----- -----
1           1          1
```

Обратите внимание: и минимальное значение, и начальное значение установлены в 1.

5. Увидеть, что произойдет, когда вы достигнете максимального значения, можно, выполнив следующий код:

```
ALTER SEQUENCE dbo.Seq1 RESTART WITH 2147483647;
```

Затем дважды выполните следующий код, чтобы запросить два новых значения последовательности:

```
SELECT NEXT VALUE FOR dbo.Seq1;
```

Сначала вы получите 2 147 483 647, а затем 1.

Резюме занятия

- SQL Server поддерживает две возможности, позволяющие генерировать последовательность ключей: свойство столбца IDENTITY и объект последовательности.
- Свойство столбца IDENTITY определяется с начальным значением и приращением. При вставке новой строки в целевую таблицу вам не нужно указывать значение для столбца IDENTITY, вместо этого SQL Server генерирует его автоматически.

- Чтобы получить вновь сгенерированное свойство идентификатора, можно выполнить запрос с помощью функций `SCOPE_IDENTITY`, `@@IDENTITY` и `IDENT_CURRENT`. Первая возвращает последнее значение идентификатора, сгенерированное в данном сеансе и диапазоне. Вторая возвращает последний идентификатор, сгенерированный в данном сеансе. Третья возвращает последний идентификатор, сгенерированный во входной таблице.
- Объект последовательности — это независимый объект в базе данных. Он не привязан к определенному столбцу в конкретной таблице.
- Объект последовательности поддерживает определение начального значения, значения приращения, минимального и максимального поддерживаемых значений, циклического повторения и кэширования.
- Функция `NEXT VALUE FOR` используется для запроса нового значения из последовательности. Эту функцию можно применять в инструкциях `INSERT` и `UPDATE`, ограничениях `DEFAULT` и присвоениях значений переменным.
- Объект последовательности обходит многие ограничения свойства `IDENTITY`.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какую функцию вы будете использовать для возвращения последнего значения идентификатора, сгенерированного в определенной таблице?
 - A. `MAX`.
 - B. `SCOPE_IDENTITY`.
 - C. `@@IDENTITY`.
 - D. `IDENT_CURRENT`.
2. В чем заключаются преимущества использования объекта последовательности вместо свойства `IDENTITY`? (Выберите все подходящие варианты.)
 - A. Свойство `IDENTITY` не гарантирует отсутствие разрывов, а объект последовательности гарантирует.
 - B. Свойство `IDENTITY` не может быть добавлено или удалено для существующего столбца; ограничение `DEFAULT` с функцией `NEXT VALUE FOR` может быть добавлено или удалено для существующего столбца.
 - C. Новое значение идентификатора не может быть сгенерировано перед использованием инструкции `INSERT`, тогда как значение последовательности может.
 - D. Вы не можете задать собственное значение, когда выполняете вставку строки в таблицу со столбцом `IDENTITY`, без специальных разрешений. Но вы можете указать собственное значение для столбца, который обычно получает свое значение из объекта последовательности.

3. Как вы сгенерируете значения последовательности в определенном порядке в инструкции `INSERT SELECT`?
- Используйте предложение `OVER` в функции `NEXT VALUE FOR`.
 - Укажите предложение `ORDER BY` в конце запроса.
 - Используйте в запросе параметр `TOP (100) PERCENT` и предложение `ORDER BY`.
 - Используйте в запросе параметр `TOP (9223372036854775807) PERCENT` и предложение `ORDER BY`.

Занятие 2. Слияние данных

С помощью инструкции `MERGE` выполняется слияние данных из исходной таблицы или табличного выражения и целевой таблицы с помещением их в целевую таблицу. Эта инструкция имеет множество практических использований как в сценариях оперативной обработки транзакций (online transaction processing, OLTP), так и при работе с хранилищами данных. В качестве примера варианта использования OLTP предположим, что у вас есть таблица, которая не обновляется напрямую вашим приложением; вместо этого, вы периодически получаете диапазон изменений от внешней системы. Сначала вы загружаете этот диапазон изменений в промежуточную таблицу и затем используете эту промежуточную таблицу в качестве источника для операции слияния с целевой таблицей.

Как пример сценария работы с хранилищем данных, предположим, что вы поддерживаете агрегированные представления данных в своем хранилище данных. Используя инструкцию `MERGE`, вы можете применять изменения, которые происходят в строках детализации, в агрегированный формат.

Мы привели только два типичных примера вариантов использования, хотя в действительности их значительно больше. В данном занятии описана инструкция `MERGE` и ее различные параметры и приведены примеры ее использования. Эти примеры используют таблицу `Sales.MyOrders` и последовательность `Sales.SeqOrderIDs`, которые вы создали в предыдущем занятии.

Изучив материал этого занятия, вы сможете:

- ✓ Использовать инструкцию `MERGE` для выполнения слияния данных и их помещения в целевую таблицу
- ✓ Определять предикат, указывающий, имеется ли соответствие между строкой источника и целевой строкой
- ✓ Определять действие, которое должно быть выполнено по отношению к целевой таблице, если строке источника сопоставлена целевая строка
- ✓ Определять действие, которое должно быть выполнено по отношению к целевой таблице, когда строке источника не сопоставлена целевая строка
- ✓ Определять действие, которое должно быть выполнено по отношению к целевой таблице, когда целевой строке не сопоставлена строка источника
- ✓ Описать разницу между ролью предложения `ON`, используемого инструкцией `MERGE`, и предложения `ON`, используемого соединением

Продолжительность занятия — 40 минут.

Использование инструкции *MERGE*

С помощью инструкции *MERGE* выполняется слияние данных из исходной таблицы или табличного выражения и целевой таблицы с помещением их в целевую таблицу. Общий формат инструкции *MERGE* выглядит следующим образом:

```

MERGE INTO <target table> AS TGT
  USING <SOURCE TABLE> AS SRC
    ON <merge predicate>
WHEN MATCHED [AND <predicate>]      -- допускаются два предложения:
    THEN <action>                   -- одно с UPDATE и одно с DELETE
WHEN NOT MATCHED [BY TARGET] [AND <predicate>]  -- допускается одно
                                                -- предложение:
    THEN INSERT...                  -- если указано, должно быть действие INSERT
WHEN NOT MATCHED BY SOURCE [AND <predicate>] -- допускаются два
                                                -- предложения:
    THEN <action>;                -- одно с UPDATE и одно с DELETE

```

Далее перечислены предложения этой инструкции и их роли.

- **MERGE INTO <target table>.** Это предложение определяет целевую таблицу (*target table*) данной операции. При желании можно задать псевдоним таблицы в этой операции.
- **USING <source table>.** Это предложение определяет исходную таблицу (*source table*) для данной операции. При желании можно задать псевдоним таблицы в этой операции. Заметьте, что предложение **USING** подобно предложению **FROM** в запросе **SELECT**. Это означает, что в данном предложении можно определять табличные операторы, такие как соединения, ссылающиеся на табличное выражение, такое как производная таблица или обобщенное табличное выражение, или даже ссылающиеся на табличную функцию, такую как **OPENROWSET**. Выход предложения **USING** в конечном итоге представляет собой табличный результат, и эта таблица будет рассматриваться в качестве источника операции слияния.
- **ON <merge predicate>.** В этом предложении вы указываете предикат (*merge predicate*), который выполняет сопоставление строк между источником и целевой таблицей и определяет, имеется или нет целевая строка, соответствующая исходной строке. Обратите внимание, это предложение не является фильтром, как, например, предложение **ON** в соединении. Вы убедитесь в важности этого в примере к данному занятию.
- **WHEN MATCHED [AND <predicate>] THEN <action>.** Это предложение определяет действие (*action*), которое следует предпринять, если строке источника соответствует целевая строка. Поскольку целевая строка существует, операция **INSERT** не разрешена в этом предложении; разрешенными являются операции **UPDATE** и **DELETE**. Если требуется применить другие действия при других условиях, можно указать предложения **WHEN MATCHED**, каждое со своим дополнительным предикатом (*predicate*), для определения, когда применять операцию **UPDATE**, а когда — **DELETE**.

- WHEN NOT MATCHED [BY TARGET] [AND <predicate>] THEN <action>. Это предложение определяет, какое действие (*action*) следует предпринять, если строке источника не соответствует целевая строка. Поскольку целевой строки не существует, единственная операция, которая разрешена в этом предложении (если вы решили включить это предложение в инструкцию), является операция INSERT. Использование операций UPDATE или DELETE просто не имеет смысла, если целевой строки не существует. По-прежнему можно добавить дополнительный предикат (*predicate*), который должен принимать значение "истина" для выполнения действия.
- WHEN NOT MATCHED BY SOURCE [AND <predicate>] THEN <action>. Это предложение определяет действие (*action*), которое следует предпринять, если целевая строка существует, но ей не соответствует строка источника. Поскольку целевая строка существует, можно применить либо операцию UPDATE, либо и DELETE, но не INSERT. При желании вы можете иметь два таких предложения с разными дополнительными предикатами (*predicate*), которые определяют, когда нужно использовать UPDATE, а когда — DELETE.

Как уже упоминалось, для демонстрации примеров инструкции MERGE в данном занятии используются таблица Sales.MyOrders и последовательность Sales.SeqOrderIDs из предыдущего занятия. Если они все еще присутствуют в вашей базе данных, используйте следующий код, чтобы очистить таблицу и сбросить последовательность:

```
TRUNCATE TABLE Sales.MyOrders;
ALTER SEQUENCE Sales.SeqOrderIDs RESTART WITH 1;
```

Если в вашей базе данных отсутствуют эта таблица и последовательность, создайте их с помощью следующего кода:

```
IF OBJECT_ID('Sales.MyOrders') IS NOT NULL DROP TABLE Sales.MyOrders;
IF OBJECT_ID('Sales.SeqOrderIDs') IS NOT NULL DROP SEQUENCE Sales.SeqOrderIDs;
```

```
CREATE SEQUENCE Sales.SeqOrderIDs AS INT
    MINVALUE 1
    CYCLE;
```

```
CREATE TABLE Sales.MyOrders
( orderid INT NOT NULL
    CONSTRAINT PK_MyOrders_orderid PRIMARY KEY
    CONSTRAINT DFT_MyOrders_orderid
        DEFAULT(NEXT VALUE FOR Sales.SeqOrderIDs),
    custid INT NOT NULL
    CONSTRAINT CHK_MyOrders_custid CHECK(custid > 0),
    empid INT NOT NULL
    CONSTRAINT CHK_MyOrders_empid CHECK(empid > 0),
    orderdate DATE NOT NULL );
```

Предположим, вам нужно определить хранимую процедуру, которая принимает в качестве входных параметров атрибуты заказа. Если заказ с исходным идентифи-

катором заказа уже существует в таблице Sales.MyOrders, вам нужно обновить строку, установив новые значения неключевых столбцов. Если идентификатор заказа не существует в целевой таблице, надо вставить новую строку. Поскольку в данном учебном курсе хранимые процедуры рассматриваются только в одной из следующих глав, примеры в данном занятии используют локальные переменные. Инструкция MERGE в хранимой процедуре просто ссылается на входные параметры процедуры, а не на локальные переменные.

Первое, что нужно указать в инструкции MERGE, — это целевая и исходная таблицы. Указать целевую таблицу просто — это таблица Sales.MyOrders. Предполагается, что источником будет таблица или табличное выражение, но в данном случае это просто набор входных параметров, представляющих заказ. Чтобы превратить входные данные в табличное выражение, можно использовать один из следующих вариантов: инструкцию SELECT без предложения FROM или конструктор табличных значений VALUES.

Далее приведен пример запроса к табличному выражению, определенному из входных переменных на основании инструкции SELECT без предложения FROM.

```
DECLARE
    @orderid    AS INT    = 1,
    @custid     AS INT    = 1,
    @empid      AS INT    = 2,
    @orderdate  AS DATE   = '20120620';
```

```
SELECT *
FROM (SELECT @orderid, @custid, @empid, @orderdate )
     AS SRC( orderid, custid, empid, orderdate );
```

Далее представлен пример выполнения того же самого запроса с помощью конструктора табличных значений VALUES.

```
DECLARE
    @orderid    AS INT    = 1,
    @custid     AS INT    = 1,
    @empid      AS INT    = 2,
    @orderdate  AS DATE   = '20120620';
```

```
SELECT *
FROM (VALUES (@orderid, @custid, @empid, @orderdate))
     AS SRC( orderid, custid, empid, orderdate);
```

В каждом примере вы определили табличное выражение на основе входных переменных, которые все вместе составляют один заказ. Инструкция SELECT к табличному выражению возвращает следующий результат:

orderid	custid	empid	orderdate
1	1	2	2012-06-20

Инструкция MERGE ожидает в качестве входных данных таблицу или табличное выражение, и такая входная таблица может быть основана на одном из рассматриваемых

мых здесь способов. Например, следующий код реализует принцип, называемый некоторыми "upsert logic" (update where exists, insert where not exists — обновить существующее, вставить несуществующее). Код использует таблицу Sales.MyOrders в качестве целевой и конструктор табличных значений из предыдущего примера в качестве исходной таблицы.

```
DECLARE  
    @orderid    AS INT    = 1,  
    @custid     AS INT    = 1,  
    @empid      AS INT    = 2,  
    @orderdate  AS DATE   = '20120620';  
MERGE INTO Sales.MyOrders WITH (HOLDLOCK) AS TGT  
USING (VALUES(@orderid, @custid, @empid, @orderdate))  
    AS SRC( orderid, custid, empid, orderdate)  
ON SRC.orderid = TGT.orderid  
WHEN MATCHED THEN UPDATE  
    SET TGT.custid      = SRC.custid,  
        TGT.empid       = SRC.empid,  
        TGT.orderdate   = SRC.orderdate  
WHEN NOT MATCHED THEN INSERT  
    VALUES (SRC.orderid, SRC.custid, SRC.empid, SRC.orderdate);
```

Вы видите, что предикат MERGE сравнивает исходный идентификатор заказа (orderid) с целевым идентификатором заказа. Когда обнаружено соответствие (значение идентификатора заказа источника совпадает со значением идентификатора заказа целевой таблицы), инструкция MERGE выполняет операцию UPDATE, которая обновляет значения неключевых столбцов в целевой таблице значениями из соответствующих строк исходной таблицы.

Если соответствия не найдены (значение идентификатора заказа источника не совпадает со значением идентификатора заказа целевой таблицы), инструкция MERGE вставляет новую строку с информацией о заказе из исходной таблицы в целевую таблицу.

ВАЖНО! Избегайте конфликтов слияния

Предположим, что некоторый ключ K пока что не существует в целевой таблице. Два процесса, P1 и P2, выполняют инструкцию MERGE, подобную предыдущей, в одно и то же время и с одним и тем же исходным ключом K. Инструкция MERGE, принадлежащая процессу P1, вполне может вставить новую строку с ключом K между моментами времени, когда инструкция MERGE, принадлежащая P2, проверяет, имеет ли целевая строка этот ключ, и вставляет строку. В таком случае инструкция MERGE, принадлежащая P2, завершится сбоем из-за нарушения ограничения первичного ключа. Чтобы предотвратить такую ситуацию, используйте подсказку SERIALIZABLE или HOLDLOCK (они имеют эквивалентное значение) к целевой таблице, как показано в предыдущей инструкции. Подробнее уровень изоляции SERIALIZABLE рассмотрен в главе 12.

Напоминаем, что вы очистили таблицу Sales.MyOrders в начале этого раздела. Поэтому, если вы запустите на выполнение предыдущий код в первый раз, он выполнит действие INSERT применительно к целевой таблице. Если вы запустите его во второй раз, он выполнит операцию UPDATE.

COBET**Инструкции MERGE****минимально требуется только одно предложение**

Инструкция MERGE не всегда требует присутствия предложений WHEN MATCHED и WHEN NOT MATCHED; в минимальном варианте вы должны указать только одно предложение, и это может быть любое из трех предложений WHEN. Например, инструкция MERGE, которая указывает только предложение WHEN MATCHED, является стандартной альтернативой инструкции UPDATE, основанной на соединении, которая не является стандартной.

Что касается выполнения кода во второй раз, обратите внимание, выполнение операции UPDATE при совершенно идентичных исходных и целевых строках является излишним. Обновление требует ресурсов и времени, и более того, если имеются триггеры или имеет место какой-либо аудит, целевые строки будут рассматриваться как обновленные. Можно избежать обновления, когда никакие значения в действительности не изменялись. Помните, что каждое предложение WHEN в инструкции MERGE разрешает дополнительный предикат, который должен принимать значение "истина" для применения соответствующего действия. Можно добавить предикат, который говорит, что хотя бы одно из значений неключевых столбцов в исходной и целевой таблицах должно различаться, чтобы действие UPDATE было применено. Ваш код будет выглядеть следующим образом:

```
DECLARE
    @orderid    AS INT    = 1,
    @custid     AS INT    = 1,
    @empid      AS INT    = 2,
    @orderdate  AS DATE   = '20120620';

MERGE INTO Sales.MyOrders WITH (HOLDLOCK) AS TGT
USING (VALUES(@orderid, @custid, @empid, @orderdate))
    AS SRC( orderid, custid, empid, orderdate)
ON SRC.orderid = TGT.orderid
WHEN MATCHED AND (TGT.custid      <> SRC.custid
                  OR TGT.empid      <> SRC.empid
                  OR TGT.orderdate <> SRC.orderdate) THEN UPDATE
    SET TGT.custid      = SRC.custid,
        TGT.empid      = SRC.empid,
        TGT.orderdate  = SRC.orderdate
WHEN NOT MATCHED THEN INSERT
    VALUES (SRC.orderid, SRC.custid, SRC.empid, SRC.orderdate);
```

Теперь код обновляет целевую строку только тогда, когда исходный идентификатор заказа равен целевому идентификатору заказа и по крайней мере один из оставшихся столбцов имеет разные значения в целевой и исходной таблицах. Если исходный идентификатор заказа не найден в целевой таблице, инструкция вставит новую строку так же, как и до этого.

ВАЖНО!**Предикат MERGE и значения NULL**

При проверке, отличается ли значение целевого столбца от значения исходного столбца, предикат инструкции MERGE использует оператор неравенства ($<\gt;$). В данном примере ни целевой, ни исходный столбцы не могут иметь значение NULL. Но если использование зна-

чений NULL разрешено в данных, необходимо добавить логику для их обработки и рассмотреть случай, когда одна сторона имеет значение NULL, а другая — нет. Например, пусть столбец custid разрешает значения NULL. Вам нужно использовать следующий предикат:

```
TGT.custid <> SRC.custid OR (TGT.custid IS NULL AND SRC.custid IS NOT NULL) OR (TGT.custid IS NOT NULL AND SRC.custid IS NULL)
```

Если вы сейчас запустите предыдущий пример, он должен показать, что обработано 0 строк.

Касательно предложения `USING`, в котором вы определяете источник для операции `MERGE`, особенно интересно то, что оно подобно предложению `FROM` в инструкции `SELECT`. Это означает, что вы можете определить табличные операторы, такие как `JOIN`, `APPLY`, `PIVOT` и `UNPIVOT`; а также использовать табличные выражения, такие как производные таблицы, обобщенные табличные выражения, представления, встроенные табличные функции и даже такие табличные функции, как `OPENROWSET` и `OPENXML`. Вы можете ссылаться на обычные таблицы, временные таблицы или табличные переменные как на источник. В конечном итоге, предложение `USING` возвращает табличный результат, и этот результат используется как источник для инструкции `MERGE`.

Язык T-SQL расширяет стандартный язык SQL, поддерживая третье предложение, которое называется `WHEN NOT MATCHED BY SOURCE`. С его помощью можно определить действие, которое следует применить к целевой строке, когда целевая строка существует, но не имеет соответствующей ей исходной строки. Допустимыми являются операции `UPDATE` и `DELETE`. Предположим, что вам нужно добавить такое предложение в последний пример, чтобы указать, что если целевая строка существует и ей не соответствует ни одна исходная строка, вы хотите удалить эту целевую строку. Далее показано, как должна выглядеть инструкция `MERGE` для такого случая (здесь в качестве источника используется табличная переменная с несколькими заказами).

```
DECLARE @Orders AS TABLE  
( orderid      INT      NOT NULL PRIMARY KEY,  
  custid       INT      NOT NULL,  
  empid        INT      NOT NULL,  
  orderdate    DATE     NOT NULL );  
  
INSERT INTO @Orders(orderid, custid, empid, orderdate) VALUES  
        (2, 1, 3, '20120612'),  
        (3, 2, 2, '20120612'),  
        (4, 3, 5, '20120612');  
  
MERGE INTO Sales.MyOrders AS TGT  
USING @Orders AS SRC  
    ON SRC.orderid = TGT.orderid  
WHEN MATCHED AND (TGT.custid      <> SRC.custid  
                  OR TGT.empid      <> SRC.empid  
                  OR TGT.orderdate <> SRC.orderdate) THEN UPDATE
```

```

SET TGT.custid      = SRC.custid,
    TGT.empid       = SRC.empid,
    TGT.orderdate   = SRC.orderdate
WHEN NOT MATCHED THEN INSERT
    VALUES(SRC.orderid, SRC.custid, SRC.empid, SRC.orderdate)
WHEN NOT MATCHED BY SOURCE THEN
    DELETE;

```

До того как вы запустили эту инструкцию на выполнение, только одна строка в таблице имела идентификатор заказа, равный 1. Инструкция вставила три строки с идентификаторами заказа 2, 3 и 4 и удалила строку, у которой идентификатор заказа был равен 1. Запросите текущее состояние таблицы.

```

SELECT *
FROM Sales.MyOrders;

```

Вы получите следующий результат с тремя оставшимися строками:

orderid	custid	empid	orderdate
2	1	3	2012-06-12
3	2	2	2012-06-12
4	3	5	2012-06-12

Контрольные вопросы

1. Какова цель предложения ON в инструкции MERGE?
2. Каковы возможные действия в предложении WHEN MATCHED?
3. Сколько предложений WHEN MATCHED может иметь одна инструкция MERGE?

Ответы на контрольные вопросы

1. Предложение ON определяет, соответствует ли строке источника целевая строка и соответствует ли целевой строке строка источника. На основании результата предиката инструкция MERGE знает, какое предложение WHEN активизировать в качестве результата и какое действие применить к целевой таблице.
2. UPDATE и DELETE.
2. Два — одно с действием UPDATE и одно с действием DELETE.

ПРАКТИКУМ Использование инструкции *MERGE*

В этом практикуме вы будете использовать полученные знания об инструкции MERGE.

Задание 1. Использование инструкции *MERGE*

В этом задании вам нужно выполнить слияние строк исходной и целевой таблиц и поместить их в целевую таблицу.

1. Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
2. Выполните следующий код для создания таблицы Sales.MyOrders:

```
IF OBJECT_ID('Sales.MyOrders') IS NOT NULL DROP TABLE Sales.MyOrders;
IF OBJECT_ID('Sales.SeqOrderIDs') IS NOT NULL DROP SEQUENCE
Sales.SeqOrderIDs;

CREATE SEQUENCE Sales.SeqOrderIDs AS INT
    MINVALUE 1
CYCLE;
CREATE TABLE Sales.MyOrders
( orderid INT NOT NULL
    CONSTRAINT PK_MyOrders_orderid PRIMARY KEY
    CONSTRAINT DFT_MyOrders_orderid
        DEFAULT(NEXT VALUE FOR Sales.SeqOrderIDs),
custid INT NOT NULL
    CONSTRAINT CHK_MyOrders_custid CHECK(custid > 0),
empid INT NOT NULL
    CONSTRAINT CHK_MyOrders_empid CHECK(empid > 0),
orderdate DATE NOT NULL );
```

3. Напишите инструкцию MERGE, которая выполняет слияние данных из таблицы Sales.Orders в таблицу Sales.MyOrders. Чтобы проверить, совпадает ли исходная строка целевой строке, сравните исходный столбец orderid с целевым столбцом orderid. Если исходная строка соответствует целевой строке, обновите неключевые столбцы в целевой таблице данными из исходной таблицы. Если исходной строке не соответствует целевая строка, вставьте новую строку с информацией из строки источника. Не обновляйте целевые строки, полностью идентичные соответствующим исходным строкам. Реализуйте и выполните следующую инструкцию MERGE:

```
MERGE INTO Sales.MyOrders AS TGT
USING Sales.Orders AS SRC
    ON SRC.orderid = TGT.orderid
WHEN MATCHED AND (TGT.custid      <> SRC.custid
                  OR TGT.empid      <> SRC.empid
                  OR TGT.orderdate <> SRC.orderdate) THEN UPDATE
SET TGT.custid      = SRC.custid,
    TGT.empid      = SRC.empid,
    TGT.orderdate = SRC.orderdate
WHEN NOT MATCHED THEN INSERT
VALUES(SRC.orderid, SRC.custid, SRC.empid, SRC.orderdate);
```

Не удаляйте таблицу Sales.MyOrders, поскольку вы будете ее использовать далее.

Задание 2. Выяснение роли предложения *ON* в инструкции *MERGE*

В этом задании вы будете использовать инструкцию *MERGE* и изучать особую роль, которую играет предложение *ON*. Вы увидите, что в отличие от операции объединения таблиц, в инструкции *MERGE* предложение *ON* не выполняет фильтрацию строк; оно только определяет наличие или отсутствие соответствия исходной и целевой строк, а также действие, которое нужно применить к целевой таблице.

- Выполните следующий код, чтобы заполнить таблицу *Sales.MyOrders* заказами из таблицы *Sales.Orders*, которые были отгружены в любую страну, кроме Норвегии (*Norway*).

```
TRUNCATE TABLE Sales.MyOrders;
INSERT INTO Sales.MyOrders(orderid, custid, empid, orderdate)
    SELECT orderid, custid, empid, orderdate
        FROM Sales.Orders
        WHERE shipcountry <> N'Norway';
```

- Перенесите заказы, которые были отгружены в Норвегию, из таблицы *Sales.Orders* в таблицу *Sales.MyOrders*. Если заказу в источнике соответствует заказ в целевой таблице и хотя бы один неключевой атрибут отличается, необходимо обновить целевую строку. Если исходной строке не соответствует ни одна строка в целевой таблице, нужно вставить такой заказ в целевую таблицу. Помните, нужно перенести только заказы, отгруженные в Норвегию. Если вы по ошибке решите, что предложение *ON* выполняет функцию фильтра, то можете попытаться реализовать инструкцию *MERGE* с предикатом *shipcountry = N'Norway'* как часть предложения *ON* следующим образом:

```
MERGE INTO Sales.MyOrders AS TGT
USING Sales.Orders AS SRC
    ON SRC.orderid = TGT.orderid
        AND shipcountry = N'Norway'
WHEN MATCHED AND (TGT.custid <> SRC.custid
                    OR TGT.empid <> SRC.empid
                    OR TGT.orderdate <> SRC.orderdate) THEN UPDATE
SET TGT.custid      = SRC.custid,
    TGT.empid       = SRC.empid,
    TGT.orderdate   = SRC.orderdate
WHEN NOT MATCHED THEN INSERT
    VALUES(SRC.orderid, SRC.custid, SRC.empid, SRC.orderdate);
```

Если вы попытаетесь выполнить этот код, то получите следующее сообщение об ошибке:

```
Msg 2627, Level 14, State 1, Line 1
Violation of PRIMARY KEY constraint 'PK_MyOrders_orderid'. Cannot insert
duplicate key in object 'Sales.MyOrders'. The duplicate key value is
(10248). The statement has been terminated.
```

Объясните причину ошибки.

Причина заключается в том, что предложение `ON` не выполняет фильтрацию строк; оно только определяет, имеет ли исходная строка соответствующую целевую строку и имеет ли целевая строка соответствующую исходную строку. На основании выходных данных этого предиката активируется нужное предложение `WHEN` и выполняется соответствующее действие. Поэтому в данном случае заказы, отгруженные во все страны, кроме Норвегии, просто рассматриваются, как не имеющие соответствия; затем активизируется предложение `WHEN NOT MATCHED` и применяется операция `INSERT`. Поскольку целевая таблица уже имеет заказы, которые были отгружены в страны, отличные от Норвегии, попытка вставить строки терпит неудачу из-за нарушения ограничения первичного ключа.

- В качестве возможного решения выполните фильтрацию релевантных строк в табличном выражении, таком как обобщенное табличное выражение (CTE) или производная таблица. В этом случае исходные данные состоят только из заказов, отгруженных в Норвегию. Напишите следующий код, содержащий обобщенное табличное выражение:

```
WITH SRC AS
( SELECT *
  FROM Sales.Orders
  WHERE shipcountry = N'Norway' )
MERGE INTO Sales.MyOrders AS TGT
USING SRC
  ON SRC.orderid = TGT.orderid
WHEN MATCHED AND (TGT.custid      <> SRC.custid
                  OR TGT.empid     <> SRC.empid
                  OR TGT.orderdate <> SRC.orderdate) THEN UPDATE
    SET TGT.custid      = SRC.custid,
        TGT.empid      = SRC.empid,
        TGT.orderdate  = SRC.orderdate
WHEN NOT MATCHED THEN INSERT
  VALUES (SRC.orderid, SRC.custid, SRC.empid, SRC.orderdate);
```

Он должен выполниться правильно и показать, что обработано 6 строк.

Теперь реализуйте тот же подход с помощью производной таблицы в качестве источника, используя следующий код:

```
MERGE INTO Sales.MyOrders AS TGT
USING (SELECT *
        FROM Sales.Orders
        WHERE shipcountry = N'Norway') AS SRC
  ON SRC.orderid = TGT.orderid
WHEN MATCHED AND (TGT.custid      <> SRC.custid
                  OR TGT.empid     <> SRC.empid
                  OR TGT.orderdate <> SRC.orderdate) THEN UPDATE
    SET TGT.custid      = SRC.custid,
        TGT.empid      = SRC.empid,
        TGT.orderdate  = SRC.orderdate
WHEN NOT MATCHED THEN INSERT
  VALUES (SRC.orderid, SRC.custid, SRC.empid, SRC.orderdate);
```

Резюме занятия

- С помощью инструкции `MERGE` можно выполнить слияние данных из исходной таблицы или табличного выражения в целевую таблицу.
- Целевую таблицу указывают в предложении `MERGE INTO`, а исходную таблицу — в предложении `USING`. Предложение `USING` похоже на предложение `FROM` инструкции `SELECT`, что означает возможность использования табличных операторов, табличных выражений, табличных функций и пр.
- Предикат слияния указывается в предложении `ON`, которое определяет, сопоставляется ли строке источника целевая строка и сопоставляется ли целевой строке строка источника. Помните, что предложение `ON` не используется для фильтрации данных; напротив, оно служит только для определения соответствия или несоответствия строк и для определения действий, которые следует применить к целевой таблице.
- Должны быть определены разные предложения `WHEN`, которые указывают, какое действие применить к целевой таблице в зависимости от выходных данных предиката. Можно указать действия, которые следует выполнить, когда исходной строке соответствует целевая строка, когда исходной строке не соответствует ни одна целевая строка и когда целевой строке не соответствует исходная строка.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какие предложения `WHEN` минимально требуются в инструкции `MERGE`?
 - A. Как минимум, требуются предложения `WHEN MATCHED` и `WHEN NOT MATCHED`.
 - B. Как минимум, требуется одно предложение, и это может быть любое предложение `WHEN`.
 - C. Как минимум, требуется предложение `WHEN MATCHED`.
 - D. Как минимум, требуется предложение `WHEN NOT MATCHED`.
2. Что может быть указано в качестве исходных данных в предложении `USING`? (Выберите все подходящие варианты.)
 - A. Обычная таблица, табличная переменная или временная таблица.
 - B. Табличное выражение, такое как производная таблица или обобщенное табличное выражение.
 - C. Хранимая процедура.
 - D. Табличная функция, такая как `OPENROWSET` или `OPENXML`.
3. Какое предложение инструкции `MERGE` не является стандартным?
 - A. Предложение `WHEN MATCHED`.
 - B. Предложение `WHEN NOT MATCHED`.

- C. Предложение WHEN NOT MATCHED BY SOURCE.
- D. Все предложения MERGE являются стандартными.

Занятие 3. Использование предложения *OUTPUT*

Язык T-SQL поддерживает предложение `OUTPUT` в инструкциях модификации данных, которое можно использовать для возвращения информации из модифицированных строк. Выходные данные можно использовать для таких целей, как аудит или архивация данных. В этом занятии рассмотрено предложение `OUTPUT` с разными видами инструкций модификации данных и показано использование данного предложения на примерах. В занятии используются таблица `Sales.MyOrders` и последовательность `Sales.SeqOrderIDs` из предыдущего занятия, поэтому убедитесь в том, что они у вас есть.

Изучив материал этого занятия, вы сможете:

- ✓ Использовать предложение `OUTPUT` в инструкциях модификации данных
- ✓ Возвращать вызывающей стороне результат предложения `OUTPUT`
- ✓ Использовать предложение `INTO` для сохранения результата в таблице
- ✓ Объяснять особенности использования предложения `OUTPUT` в инструкции `MERGE`
- ✓ Фильтровать выходные строки с помощью компонуемого DML

Продолжительность занятия — 30 минут.

Работа с предложением *OUTPUT*

Конструкция предложения `OUTPUT` очень похожа на структуру предложения `SELECT` в том смысле, что вы можете указать выражения и назначить им псевдонимы результирующих столбцов. Разница по сравнению с предложением `SELECT` заключается в том, что в предложении `OUTPUT`, когда вы ссылаетесь на столбцы из модифицированных строк, имена столбцов должны предваряться ключевыми словами `inserted` или `deleted`. Слово `inserted` используется для вставляемых строк, а слово `deleted` — для удаленных строк. В предложении `UPDATE` ключевое слово `inserted` представляет статус строк после обновления, а ключевое слово `deleted` — статус строк до обновления.

Предложение `OUTPUT` может возвращать результирующий набор вызывающей стороне так же, как это делает инструкция `SELECT`. Также можно добавить предложение `INTO` для направления выходных строк в целевую таблицу. В действительности, при желании можно иметь два предложения `OUTPUT` — одно с инструкцией `INTO`, направляющей строки в таблицу, и второе — с инструкцией `INTO`, возвращающей результирующий набор из запроса. Если вы используете предложение `INTO`, целевая таблица не может участвовать ни в одной стороне отношения по внешнему ключу и не может иметь определенных на ней триггеров.

Как уже упоминалось, в данном занятии используются таблица Sales.MyOrders и последовательность Sales.SeqOrderIDs из предыдущего занятия. Выполните следующий код для того, чтобы очистить таблицу и сбросить начальное значение последовательности в 1.

```
TRUNCATE TABLE Sales.MyOrders;
ALTER SEQUENCE Sales.SeqOrderIDs RESTART WITH 1;
```

Если в вашей базе данных отсутствуют эта таблица и последовательность, создайте их с помощью следующего кода:

```
IF OBJECT_ID('Sales.MyOrders') IS NOT NULL DROP TABLE Sales.MyOrders;
IF OBJECT_ID('Sales.SeqOrderIDs') IS NOT NULL DROP SEQUENCE Sales.SeqOrderIDs;

CREATE SEQUENCE Sales.SeqOrderIDs AS INT
    MINVALUE 1
    CYCLE;

CREATE TABLE Sales.MyOrders
( orderid INT NOT NULL
    CONSTRAINT PK_MyOrders_orderid PRIMARY KEY
    CONSTRAINT DFT_MyOrders_orderid
        DEFAULT(NEXT VALUE FOR Sales.SeqOrderIDs),
    custid INT NOT NULL
    CONSTRAINT CHK_MyOrders_custid CHECK(custid > 0),
    empid INT NOT NULL
    CONSTRAINT CHK_MyOrders_empid CHECK(empid > 0),
    orderdate DATE NOT NULL);
```

Инструкция *INSERT* с предложением *OUTPUT*

Предложение *OUTPUT* может использоваться в инструкции *INSERT* для того, чтобы возвращать информацию из вставленных строк. Примером его практического использования может служить многострочная инструкция *INSERT*, которая генерирует новые ключи с помощью свойства *IDENTITY* или последовательности, а вам нужно знать, какие новые ключи были сгенерированы.

Например, предположим, что вам нужно запросить таблицу Sales.Orders и вставить заказы, отгруженные в Норвегию (*Norway*), в таблицу Sales.MyOrders. Вы не планируете использовать в целевых строках исходные идентификаторы заказов; вместо этого вы предоставите возможность сгенерировать их объекту последовательности. Но вы хотите получить обратно информацию из инструкции *INSERT* о том, какие идентификаторы заказов были сгенерированы, а также сведения о дополнительных столбцах из вставленных строк. Для этого просто добавьте предложение *OUTPUT* в инструкцию *INSERT* прямо перед запросом. Перечислите столбцы, которые нужно возвратить из вставленных строк, предварив их имена ключевым словом *inserted*, как показано в следующем примере:

```
INSERT INTO Sales.MyOrders(custid, empid, orderdate)
OUTPUT
    inserted.orderid, inserted.custid, inserted.empid, inserted.orderdate
SELECT custid, empid, orderdate
FROM Sales.Orders
WHERE shipcountry = N'Norway';
```

Этот код генерирует следующие выходные данные:

ordered	custid	empid	orderdate
1	70	1	2006-12-18
2	70	7	2007-04-29
3	70	7	2007-08-20
4	70	3	2008-01-14
5	70	1	2008-02-26
6	70	2	2008-04-10

Вы видите, что объект последовательности сгенерировал идентификаторы заказов от 1 до 6 для новых строк. Если вам нужно сохранить результат в таблице, а не возвращать его назад вызывающей стороне, добавьте предложение `INTO` с именем целевой таблицы следующим образом:

```
INSERT INTO Sales.MyOrders(custid, empid, orderdate)
OUTPUT
    inserted.orderid, inserted.custid, inserted.empid, inserted.orderdate
    INTO SomeTable(orderid, custid, empid, orderdate)
SELECT custid, empid, orderdate
FROM Sales.Orders
WHERE shipcountry = N'Norway';
```

Не выполняйте этот код, т. к. таблица `SomeTable` не существует в базе данных — это просто пример.

Как уже упоминалось, если используется предложение `INTO`, целевая таблица не может участвовать ни в одной стороне отношения по внешнему ключу и не может иметь определенных на ней триггеров.

Инструкция `DELETE` с предложением `OUTPUT`

Можно использовать предложение `OUTPUT` в инструкции `DELETE` для возвращения информации из удаленных строк. Столбцы, на которые вы ссылаетесь, должны иметь в качестве префикса ключевое слово `deleted`.

Следующий пример удаляет строки из таблицы `Sales.MyOrders`, в которых идентификатор сотрудника (employee ID) равен 1. Используя предложение `OUTPUT`, код возвращает идентификаторы заказов для удаленных заказов.

```
DELETE FROM Sales.MyOrders
OUTPUT deleted.orderid
WHERE empid = 1;
```

Этот код генерирует следующие выходные данные:

```
orderid
-----
1
5
```

Помните, что если вам нужно сохранять выходные строки в таблице — например, для архивирования — вы можете добавить предложение `INTO` с именем целевой таблицы.

Инструкция *UPDATE* с предложением *OUTPUT*

Можно использовать предложение `OUTPUT` в предложении `UPDATE` для возвращения информации их модифицированных строк. С помощью обновленных строк имеется доступ как к старому, так и к новому содержимому изменяемых строк. Для ссылки на столбцы строк в ее исходном состоянии до обновления добавьте к именам столбцов префикс `deleted`. Для ссылки на столбцы строки в ее новом состоянии после обновления добавьте к именам столбцов префикс `inserted`.

В качестве примера следующая инструкция `UPDATE` добавляет день к дате заказа всех заказов, которые обрабатывались сотрудником с номером 7.

```
UPDATE Sales.MyOrders
    SET orderdate = DATEADD(day, 1, orderdate)
    OUTPUT inserted.orderid,
          deleted.orderdate AS old_orderdate,
          inserted.orderdate AS neworderdate
    WHERE empid = 7;
```

Этот код использует предложение `OUTPUT` для того, чтобы возвратить идентификаторы заказов из измененных строк, а также даты заказов — как до, так и после обновления. Этот код генерирует следующие выходные данные:

orderid	old_orderdate	neworderdate
2	2007-04-29	2007-04-30
3	2007-08-20	2007-08-21

Инструкция *MERGE* с предложением *OUTPUT*

Вы можете использовать предложение `OUTPUT` с инструкцией `MERGE`, но тогда следует принимать во внимание некоторые особенности этой инструкции. Помните, что одна инструкция `MERGE` может применяться к целевой таблице разные действия. Предположим, что при возвращении выходных строк вам нужно знать, какое именно действие (`INSERT`, `UPDATE` или `DELETE`) было применено к выходной строке. Для этого SQL Server предоставляет функцию `$action`. Эта функция возвращает строку ('`INSERT`', '`UPDATE`' или '`DELETE`'), обозначающую выполненную операцию.

Как мы уже говорили, можно сослаться на столбцы из удаленных строк с префиксом `deleted` и на столбцы из вставленных строк с префиксом `inserted`. Строки, об-

работанные операцией `INSERT`, имеют значения во вставленной строке и значения `NULL` в удаленной строке. Строки, обработанные операцией `DELETE`, имеют значения `NULL` во вставленной строке и значения в удаленной строке. Строки, обработанные операцией `UPDATE`, имеют значения в обеих строках. Например, если вы хотите возвратить ключ обработанной строки (при условии, что сам ключ не был изменен), можно использовать выражение `COALESCE (inserted.orderid, deleted.orderid)`.

Следующий пример демонстрирует использование инструкции `MERGE` с предложением `OUTPUT`, возвращающим выходные данные функции `$action`, чтобы указать, какое действие было применено к строке и ключ измененной строки.

```

MERGE INTO Sales.MyOrders AS TGT
USING (VALUES(1,    70,    1,    '20061218')
        (2,    70,    7,    '20070429')
        (3,    70,    7,    '20070820')
        (4,    70,    3,    '20081014')
        (5,    70,    1,    '20080226')
        (6,    70,    2,    '20080410'))
AS SRC(orderid, custid, empid, orderdate)
ON SRC.orderid = TGT.orderid
WHEN MATCHED AND (TGT.custid      <> SRC.custid
                   OR   TGT.empid      <> SRC.empid
                   OR   TGT.orderdate <> SRC.orderdate) THEN UPDATE
SET TGT.custid      = SRC.custid,
    TGT.empid      = SRC.empid,
    TGT.orderdate = SRC.orderdate
WHEN NOT MATCHED THEN INSERT
    VALUES(SRC.orderid, SRC.custid, SRC.empid, SRC.orderdate)
WHEN NOT MATCHED BY SOURCE THEN
    DELETE
OUTPUT
    $action AS the_action,
    COALESCE(inserted.orderid, deleted.orderid) AS orderid;

```

Этот код генерирует следующие выходные данные:

the_action	orderid
INSERT	1
INSERT	5
UPDATE	2
UPDATE	3

Эти выходные данные показывают, что две строки были вставлены и две обновлены.

COBET

MERGE И OUTPUT

В инструкциях `INSERT`, `UPDATE` и `DELETE` можно ссылаться только на столбцы из целевой таблицы в предложении `OUTPUT`. В инструкции `MERGE` можно ссылаться на столбцы как целевой таблицы, так и исходной таблицы.

Выполните следующий код для очистки таблицы Sales.MyOrders:

```
TRUNCATE TABLE Sales.MyOrders;
```

Компонуемый DML

Предположим, вам необходимо получить выходные данные из модифицированной инструкции, но при этом интерес для вас представляют не все выходные строки, а только их часть. Язык T-SQL предлагает решение этой проблемы в виде так называемого компонуемого DML (data manipulation language, язык обработки данных).

Используя язык T-SQL, можно определить некое подобие производной таблицы на основе модификации с предложением `OUTPUT`. Затем можно применить к целевой таблице внешнюю инструкцию `INSERT SELECT`, где исходной таблицей является эта особая производная таблица. Внешняя инструкция `INSERT SELECT` может иметь предложение `WHERE`, которое отфильтровывает выходные строки из производной таблицы, вставляя только строки, которые соответствуют условиям поиска, в целевую таблицу. Внешняя инструкция `INSERT SELECT` не может иметь иных элементов, кроме `WHERE` в качестве табличных операторов, `GROUP BY`, `HAVING` и т. д.

В качестве примера *компонентного DML* рассмотрим предыдущую инструкцию `MERGE`. Предположим, что вам нужно получить лишь строки, к которым применялась только операция `INSERT`, и передать их в табличную переменную для последующей обработки. Этого можно достигнуть с помощью следующего кода:

```
DECLARE @InsertedOrders AS TABLE
( orderid      INT      NOT NULL PRIMARY KEY,
  custid       INT      NOT NULL,
  empid        INT      NOT NULL,
  orderdate    DATE     NOT NULL );
INSERT INTO @InsertedOrders(orderid, custid, empid, orderdate)
SELECT orderid, custid, empid, orderdate
FROM (MERGE INTO Sales.MyOrders AS TGT
      USING (VALUES(1,    70,    1,    '20061218'),
              (2,    70,    7,    '20070429'),
              (3,    70,    7,    '20070820'),
              (4,    70,    3,    '20080114'),
              (5,    70,    1,    '20080226'),
              (6,    70,    2,    '20080410'),
              AS SRC(orderid, custid, empid, orderdate )
      ON SRC.orderid = TGT.orderid
      WHEN MATCHED AND (TGT.custid      <> SRC.custid
                         OR   TGT.empid      <> SRC.empid
                         OR   TGT.orderdate <> SRC.orderdate) THEN UPDATE
          SET TGT.custid      = SRC.custid,
              TGT.empid      = SRC.empid,
              TGT.orderdate = SRC.orderdate)
```



```

WHEN NOT MATCHED THEN INSERT
    VALUES(SRC.orderid, SRC.custid, SRC.empid, SRC.orderdate)
WHEN NOT MATCHED BY SOURCE THEN
    DELETE
OUTPUT
    $action AS the_action, inserted.*)
AS D
WHERE the_action = 'INSERT';

SELECT *
FROM @InsertedOrders;

```

Обратите внимание, производная таблица D определена на основе инструкции MERGE с предложением OUTPUT. Предложение OUTPUT возвращает, кроме прочего, результат функции \$action, поименовав целевой столбец как the_action. Код использует инструкцию INSERT SELECT с источником, являющимся производной таблицей D, и целевой таблицей, являющейся табличной переменной @InsertedOrders. Предложение WHERE во внешнем запросе фильтрует только строки, содержащие операцию INSERT.

При запуске в первый раз предыдущего кода выводится следующий результат:

orderid	custid	empid	orderdate
1	70	1	2006-12-18
2	70	7	2007-04-29
3	70	7	2007-08-20
4	70	3	2008-01-14
5	70	1	2008-02-26
6	70	2	2008-04-10

Запустите его второй раз. Теперь он должен возвратить пустой набор.

orderid	custid	empid	orderdate

По окончании запустите следующий код для очистки данных:

```

IF OBJECT_ID('Sales.MyOrders') IS NOT NULL DROP TABLE Sales.MyOrders;
IF OBJECT_ID('Sales.SeqOrderIDs') IS NOT NULL DROP SEQUENCE Sales.SeqOrderIDs;

```

Контрольные вопросы

- Сколько предложений OUTPUT может содержать одна инструкция?
- Как можно определить, какое действие было выполнено применительно к строке OUTPUT в инструкции MERGE?

Ответы на контрольные вопросы

- Два — одно с INTO и одно без INTO.
- Используйте функцию \$action.

ПРАКТИКУМ Использование предложения *OUTPUT*

В этом практикуме вы будете проверять полученные вами знания о предложении *OUTPUT*.

Задание 1. Использование предложения *OUTPUT* в инструкции *UPDATE*

В этом задании вы будете использовать предложение *OUTPUT* в инструкции *UPDATE* и сравнивать столбцы до и после изменения.

1. Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
2. Вам нужно применить обновления к продуктам в категории 1, которые поставлены поставщиком с номером 16. Сначала вы должны выполнить следующий запрос к таблице *Production.Products* для того, чтобы просмотреть продукты, которые вы собираетесь обновить.

```
SELECT productid, productname, unitprice  
FROM Production.Products WHERE categoryid = 1  
    AND supplierid = 16;
```

Вы получите следующий результат:

productid	productname	unitprice
34	Product SWNJP	14.00
35	Product NEVTJ	18.00
67	Product XLXQF	14.00

3. Напишите инструкцию *UPDATE*, модифицирующую данные о продуктах в категории 1, которые поставлены поставщиком с номером 16, увеличив их цену на 2,5. Включите предложение *OUTPUT*, которое возвращает идентификатор продукта, имя продукта, старую цену, новую цену и разницу в процентах между новой и старой ценой. Ваша инструкция *UPDATE* должна выглядеть следующим образом:

```
UPDATE Production.Products  
    SET unitprice += 2.5  
    OUTPUT inserted.productid,  
          inserted.productname,  
          deleted.unitprice AS oldprice,  
          inserted.unitprice AS newprice,  
          CAST(100.0 * (inserted.unitprice - deleted.unitprice)  
              / deleted.unitprice AS NUMERIC(5, 2)) AS pct  
    WHERE categoryid = 1  
        AND supplierid = 16;
```

Эта инструкция генерирует следующие выходные данные, содержащие запрашиваемую информацию.

productid	productname	oldprice	newprice	pct
34	Product SWNJY	14.00	16.50	17.86
35	Product NEVTJ	18.00	20.50	13.89
67	Product XLXQF	14.00	16.50	17.86

4. Чтобы вернуться к исходным значениям, напишите обратное предложение UPDATE, снижающее цены продуктов на 2,5. Включите те же выходные данные, что и в предыдущей инструкции.

Ваш код должен выглядеть следующим образом:

```
UPDATE Production.Products
    SET unitprice == 2.5
    OUTPUT inserted.productid,
          inserted.productname,
          deleted.unitprice AS oldprice,
          inserted.unitprice AS newprice,
    CAST(100.0 * (inserted.unitprice - deleted.unitprice)
        / deleted.unitprice AS NUMERIC(5, 2)) AS pct
    WHERE categoryid = 1 AND supplierid = 16;
```

Эта инструкция генерирует следующие выходные данные:

productid	productname	oldprice	newprice	pct
34	Product SWNJY	16.50	14.00	-15.15
35	Product NEVTJ	20.50	1800	-12.20
67	Product XLXQF	16.50	14.00	-15.15

Задание 2. Использование компонуемого DML

В этом задании вы будете использовать компонуемый DML. Вам нужно удалить строки из таблицы и заархивировать в другой таблице подмножество удаленных строк.

1. Создайте таблицу и подготовьте данные для этого задания, выполнив следующий код:

```
IF OBJECT_ID('Sales.MyOrdersArchive') IS NOT NULL
    DROP TABLE Sales.MyOrdersArchive;
IF OBJECT_ID('Sales.MyOrders') IS NOT NULL
    DROP TABLE Sales.MyOrders;

CREATE TABLE Sales.MyOrders
( orderid      INT NOT NULL
    CONSTRAINT PK_MyOrders PRIMARY KEY,
  custid       INT NOT NULL,
  empid        INT NOT NULL,
  orderdate    DATE NOT NULL );
INSERT INTO Sales.MyOrders(orderid, custid, empid, orderdate)
    SELECT orderid, custid, empid, orderdate
    FROM Sales.Orders;
```

```
CREATE TABLE Sales.MyOrdersArchive
( orderid      INT      NOT NULL
    CONSTRAINT PK_MyOrdersArchive PRIMARY KEY,
  custid       INT      NOT NULL,
  empid        INT      NOT NULL,
  orderdate    DATE     NOT NULL );
```

2. Напишите запрос с инструкцией к таблице Sales.MyOrders, которая удаляет заказы, размещенные до 2007 г. Используйте компонуемый DML для архивирования удаленных заказов, которые были размещены клиентами, имеющими идентификаторы 17 и 19. Реализуйте и выполните следующую инструкцию:

```
INSERT INTO Sales.MyOrdersArchive(orderid, custid, empid, orderdate)
  SELECT orderid, custid, empid, orderdate
    FROM (DELETE FROM Sales.MyOrders
  OUTPUT deleted.*
    WHERE orderdate < '20070101') AS D
  WHERE custid IN (17, 19);
```

3. Выполните запрос к таблице Sales.MyOrdersArchive, чтобы посмотреть, какие строки заархивированы.

```
SELECT *
  FROM Sales.MyOrdersArchive;
```

Вы получите следующий результат:

orderid	custid	empid	orderdate
10363	17	4	2006-11-26
10364	19	1	2006-11-26
10391	17	3	2006-12-23

4. По окончании запустите следующий код для очистки данных:

```
IF OBJECT_ID('Sales.MyOrdersArchive') IS NOT NULL
  DROP TABLE Sales.MyOrdersArchive';
IF OBJECT_ID('Sales.MyOrders') IS NOT NULL
  DROP TABLE 'Sales.MyOrders;
```

Резюме занятия

- С помощью предложения OUTPUT можно возвращать информацию из измененных строк в инструкциях модификации.
- Предложение OUTPUT построено как предложение SELECT, что позволяет формировать выражения и присваивать результирующим столбцам псевдонимы.
- Результат предложения OUTPUT можно отправить обратно вызывающей стороне в виде результирующего набора из запроса или сохранить в целевой таблице с помощью предложения INTO.

- При ссылке на столбцы из модифицированных строк следует присваивать в качестве префикса именам столбцов ключевое слово `inserted` для вставленных строк и `deleted` для удаленных строк.
- В инструкции `MERGE` можно использовать функцию `$action` для возвращения строки, которая представляет действие, примененное к целевой строке.
- Компонуемый DML следует использовать для фильтрации выходных строк, которые необходимо сохранить в целевой таблице.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Когда следует предварять имена столбцов префиксом в виде ключевого слова `inserted` при ссылке в предложении `OUTPUT` на столбцы из вставленных строк?
 - A. Всегда.
 - B. Никогда.
 - C. Только когда используется инструкция `UPDATE`.
 - D. Только когда используется инструкция `MERGE`.
2. Что является ограничением для таблицы, указанной в качестве целевой в предложении `OUTPUT INTO`? (Выберите все подходящие варианты.)
 - A. Таблица может быть только табличной переменной.
 - B. Таблица может быть только временной таблицей.
 - C. Таблица не может быть задействована ни в одной части отношения по внешнему ключу.
 - D. Таблица не может иметь триггеров, определенных на ней.
3. Что из нижеперечисленного возможно только при использовании инструкции `MERGE` с точки зрения применения предложения `OUTPUT`?
 - A. Ссылка на столбцы из исходной таблицы.
 - B. Ссылка и на ключевое слово `deleted`, и на ключевое слово `inserted`.
 - C. Назначение псевдонимов выходным столбцам.
 - D. Использование компонуемого DML.

Упражнения

В следующих упражнениях вы примените полученные знания о вставке, обновлении и удалении данных. Ответы на вопросы можно найти в *приложении "Ответы" в конце книги*.

Упражнение 1. Лучшее решение для генерации ключей

Вы являетесь членом группы администраторов баз данных в компании, занимающейся производством туристского снаряжения. Большинство таблиц в базах данных OLTP компании в настоящий момент использует свойство `IDENTITY`, но требует большей гибкости. Например, часто приложению нужно генерировать новый ключ до его использования. Иногда приложению требуется обновить ключевой столбец, перезаписав в него новые значения. Кроме того, предложению требуется генерировать ключи, которые не конфликтуют при использовании в разных таблицах.

1. Предложите альтернативу использованию свойства столбца `IDENTITY`.
2. Объясните, как ваше альтернативное предложение может решить существующие проблемы.

Упражнение 2. Усовершенствование модификаций

Вы работаете в группе поддержки баз данных в компании, которая недавно перешла с версии SQL Server 2000 на версию SQL Server 2005 и затем на версию SQL Server 2012. До сих пор используется код, совместимый с SQL Server 2000. Существуют проблемы с модификациями, внесеннымными приложением в базу данных.

Приложение использует процедуру, которая принимает в качестве входных данных атрибуты строки. Затем процедура использует логику, которая проверяет, существует ли ключ в целевой таблице, и при его наличии обновляет целевую строку. В противном случае процедура вставляет новую строку в целевую таблицу. Проблема заключается в том, что периодически процедура дает сбой из-за нарушения ограничения первичного ключа. Это происходит, когда проверка на наличие строки не находит ее, но между этой проверкой и вставкой кто-то еще успевает вставить новую строку с тем же ключом.

В приложении имеется ежемесячный процесс, который архивирует данные, подлежащие удалению. В настоящий момент приложение сначала копирует данные, которые должны быть удалены, в архивную таблицу в одной инструкции и затем удаляет эти строки в другой инструкции. Обе инструкции используют фильтр на основе столбца с датой, который имеет имя `dt`. Требуется отфильтровывать строки, в которых `dt` меньше определенной даты. Проблема заключается в том, что иногда строки, представляющие последние прибытия, вставляются в таблицу между копированием и удалением строк, и процесс удаления заканчивается удалением строк, которые не были заархивированы.

Перед вами поставлена задача найти решения существующих проблем.

1. Можете ли вы предложить решение существующей проблемы с процедурой, которая обновляет строку, когда исходный ключ существует в целевой таблице, и вставляет строку, если его нет?
2. Можете ли вы предложить решение проблемы с процессом архивирования, которое предотвратит удаление строк, которые еще не заархивированы?

Рекомендуемые упражнения

Для успешного освоения экзаменационных тем, представленных в данной главе, выполните следующие задания.

Сравнение старых и новых свойств

В следующих практических заданиях вы проверите, как много вам удалось запомнить из прочитанного в данной главе. Постарайтесь использовать лишь свою память. Только по завершении задачи просмотрите текст главы, чтобы проверить, не пропустили ли вы что-нибудь.

- Задание 1.** Напишите список преимуществ объекта последовательности по сравнению со свойством столбца IDENTITY.
- Задание 2.** Напишите список преимуществ инструкции MERGE по сравнению с использованием отдельных инструкций для различных случаев (WHEN MATCHED, WHEN NOT MATCHED, WHEN NOT MATCHED BY SOURCE).
- Задание 3.** Напишите список преимуществ предложения OUTPUT перед способами, которые не используют предложение OUTPUT для достижения таких же результатов. В качестве примеров возьмите аудит и архивирование.

ГЛАВА 12

Реализация транзакций, обработка ошибок и динамический SQL

Темы экзамена

- Работа с данными.
 - Запрос данных с помощью инструкций SELECT.
- Устранение неполадок и оптимизация.
 - Оптимизация запросов.
 - Управление транзакциями.
 - Реализация обработки ошибок.

Microsoft SQL Server — это реляционная база данных, в которой строго выполняется транзакционный способ изменений в базах данных с целью защиты целостности данных. В данной главе код T-SQL рассматривается с точки зрения транзакций, а затем обсуждается использование этого кода при обработке ошибок и в динамическом SQL.

ПРЕЖДЕ ВСЕГО

Для полного освоения этой главы необходимо иметь:

- понимание основных принципов баз данных;
- опыт работы с SQL Server Management Studio (SSMS);
- некоторый опыт в написании кода на языке T-SQL;
- доступ к экземпляру SQL Server 2012 с установленной учебной базой данных TSQL2012.

Занятие 1. Управление транзакциями и параллелизм

Система управления реляционными базами данных (relational database management system, RDBMS) SQL Server 2012 поддерживает транзакционный контроль над

всеми изменениями данных в базе данных. Строгое следование транзакционному контролю в SQL Server гарантирует, что целостность данных в базе данных никогда не будет нарушена из-за частично завершенных транзакций, нарушенных ограничений, вмешательства других транзакций или прерывания обслуживания.

Изучив материал этого занятия, вы сможете:

- ✓ Дать определение ACID-свойств транзакций
- ✓ Описать и установить режимы и типы транзакций
- ✓ Описать режимы блокировки, блокирование и взаимоблокирование
- ✓ Описать и установить уровни изоляции транзакций
- ✓ Описать правила эффективного кодирования транзакций

Продолжительность занятия — 50 минут.



Основные понятия транзакций

Транзакция — это логическая единица работы. Либо вся работа выполняется как единое целое, либо она не выполняется вовсе. Транзакции — это весьма обычное явление в нашей повседневной жизни. Например, любую покупку можно рассматривать как транзакцию. Когда вы платите деньги за покупку, но не получаете ее, транзакция останавливается, и вы ожидаете возвращения ваших денег. Оплата деньгами за покупку и получение того, что вы купили, составляет логическую единицу работы. Либо оба шага завершаются успешно, либо оба они должны быть не выполнены.

В SQL Server все изменения данных в базе данных происходят в виде транзакций. Другими словами, все операции, которые каким-либо образом делают запись в базу данных, воспринимаются в SQL Server как транзакции. К ним относятся:

- все инструкции языка обработки данных (data manipulation language, DML), такие как `INSERT`, `UPDATE` и `DELETE`;
- все инструкции языка описания данных (data definition language, DDL), такие как `CREATE TABLE` и `CREATE INDEX`.



Строго говоря, даже инструкции `SELECT` представляют собой тип транзакции в SQL Server; их называют транзакциями только для чтения. Поскольку они не относятся ни к DML-транзакциям, ни к DDL-транзакциям, они не рассматриваются в данной книге. Подробную информацию о транзакциях только для чтения можно найти в электронной документации по SQL Server 2012 в разделе "sys.dm_tran_database_transactions" по адресу [http://msdn.microsoft.com/ru-ru/library/ms186957\(v=sql.90\).aspx](http://msdn.microsoft.com/ru-ru/library/ms186957(v=sql.90).aspx).

Термины "фиксация" (`commit`) и "откат" (`rollback`) используются в этой книге для обозначения действия по управлению результатом транзакций в SQL Server. Когда работа транзакции утверждена пользователем, SQL Server завершает изменения транзакции путем их фиксации. Если возникает фатальная ошибка или пользователь решает не выполнять фиксацию, будет выполнен откат транзакции.

Свойства транзакций ACID

В реляционных базах данных аббревиатура ACID используется для описания свойств транзакций. К ACID свойствам относятся следующие.

- **Атомарность (atomicity).** Каждая транзакция является атомарной единицей работы. Это означает, что в транзакции либо выполняются все изменения базы данных, либо ни одно из них.
- **Согласованность (consistency).** Каждая транзакция, как завершившаяся, так и прерванная, оставляет базу данных в согласованном состоянии, как определяется всеми ограничениями объектов и базы данных. При возникновении несогласованного состояния SQL Server выполнит откат транзакции, чтобы поддерживать согласованное состояние.
- **Изоляция (isolation).** Каждая транзакция выполняется так, как будто она существует в изоляции от всех остальных транзакций по отношению к изменениям в базе данных. Степень изолированности может меняться в зависимости от уровня изоляции.
- **Устойчивость (durability).** Каждая транзакция претерпевает прерывание сервиса. Когда сервис восстанавливается, все зафиксированные (committed) транзакции накатываются (зафиксированные изменения в базе данных завершены), а все нефиксированные транзакции откатываются (нефиксированные изменения удаляются).

SQL Server обеспечивает все эти ACID-свойства посредством различных механизмов. Для поддержки атомарности SQL Server обрабатывает каждую DML- или DDL-команду индивидуально и не допустит только частичного выполнения ни одной команды. Например, рассмотрим инструкцию `UPDATE`, которая должна обновить 500 строк в таблице в момент времени, когда начинается транзакция. Эта команда не будет завершена, пока точно все эти 500 строк не будут обновлены. Если что-либо помешает команде обновить все 500 строк, SQL Server прервет команду и откатит транзакцию.

Если в транзакции имеется более одной команды, SQL Server обычно не разрешает принять всю транзакцию, пока обе инструкции не будут выполнены. (Если параметр `XACT_ABORT` установлен в состояние `OFF`, которое является состоянием по умолчанию, можно вставить код для принятия решения, откатить транзакцию или зафиксировать ее. Дополнительную информацию можно найти в разд. "Использование параметра `XACT_ABORT` с блоками `TRY/CATCH`" далее в этой главе.)

Для обеспечения согласованности SQL Server гарантирует, что все ограничения в базе данных соблюdenы. Если транзакция пытается вставить строку, которая имеет, например, индивидуальный внешний ключ, тогда SQL Server определит, что ограничение будет нарушено, и сгенерирует сообщение об ошибке. Вы можете добавить логику, которая будет принимать решение, откатывать такую транзакцию или нет.

Для обеспечения изолированности транзакции SQL Server гарантирует, что когда транзакция выполняет несколько изменений в базе данных, ни один из объектов,

изменяемых этой транзакцией, не может быть изменен никакой другой транзакцией. Иными словами, изменения одной транзакции изолированы от любых действий других транзакций. Если двум транзакциям нужно изменить одни и те же данные, одна из них должна подождать, пока другая транзакция не закончится.

SQL Server обеспечивает изоляцию транзакций посредством механизмов блокирования и версионирования строк. SQL Server блокирует объекты (например, строки и таблицы), чтобы предотвратить вмешательство других транзакций в действия данной транзакции (блокирование рассмотрено далее в этом занятии).

Изоляция может значительно отличаться в зависимости от того, какая степень изолированности существует у транзакции, когда она читает данные. Это различие обеспечивается настройкой уровней изоляции (которые рассматриваются далее в этом занятии).

SQL Server поддерживает транзакционную устойчивость с помощью журнала транзакций базы данных. Любое изменение базы данных (инструкция модификации данных или инструкция DDL) сначала записывается в журнал транзакций, с исходной версией данных (в случае обновлений и удалений). Когда транзакция зафиксирована и прошли все проверки согласованности, факт успешной фиксации транзакции записывается в журнал транзакций. Например, если сервер базы данных неожиданно завершит работу сразу после того, как факт успешной фиксации был записан в журнал транзакций, когда SQL Server запустит базу данных, будет выполнен накат транзакции и любые незаписанные изменения в базе данных будут завершены.

С другой стороны, если сервер базы данных неожиданно завершит работу перед тем, как факт успешной фиксации может быть записан в журнал транзакций, когда SQL Server запустит базу данных, будет выполнен откат транзакции и любые изменения в базе данных будут отменены.

В SQL Server каждая база данных, включая каждую системную базу данных, имеет журнал транзакций, чтобы обеспечить устойчивость транзакций. Невозможно отключить журнал транзакций базы данных или удалить его. Хотя некоторые операции могут каким-либо образом уменьшить запись в журнал транзакций, все изменения базы данных всегда сначала записываются в журнал транзакций.

К СВЕДЕНИЮ SQL Server и устойчивость транзакций

Дополнительную информацию о том, как журнал транзакций реализует устойчивость, можно найти в электронной документации по SQL Server 2008 R2 в разделе "Журнал транзакций с упреждающей записью" по адресу:

[http://msdn.microsoft.com/ru-ru/library/ms186259\(SQL.105\).aspx](http://msdn.microsoft.com/ru-ru/library/ms186259(SQL.105).aspx).

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Почему для SQL Server важно поддерживать качество транзакций, соответствующее свойствам ACID?
2. Как SQL Server реализует устойчивость транзакций?

Ответы на контрольные вопросы

- Чтобы гарантировать, что целостность данных в базе данных не будет нарушена.
- Записывая все изменения в журнал транзакций базы данных до внесения изменений в данные базы данных.

Типы транзакций

В SQL Server существуют два основных типа транзакций.

- **Системные транзакции.** SQL Server использует *системные транзакции* для поддержки всех своих внутренних постоянных системных таблиц. Пользователи не могут управлять этими транзакциями.
- **Пользовательские транзакции.** Транзакции, создаваемые пользователями в процессе изменения или даже чтения данных, либо автоматически, т. е. неявно, либо явно, называются *пользовательскими транзакциями*. Имена транзакций можно просмотреть в столбце `name` динамического административного представления (dynamic management view, DMV) `sys.dm_tran_active_transactions`. По умолчанию имя пользовательских транзакций — `user_transaction`. Можно также присвоить транзакции собственное имя с помощью явных транзакций, описанных в следующем разделе.

Оставшаяся часть этого занятия посвящена только пользовательским транзакциям.

Команды транзакций

Команды, описанные в этом разделе, управляют тем, какой код включен в явную транзакцию и поведение этой транзакции.

Каждая транзакция состоит из инструкции языка T-SQL `BEGIN TRANSACTION`, которая обозначает начало транзакции в коде. Можно так же записать эту команду, как `BEGIN TRAN`. Транзакции может быть присвоено имя.

Транзакция должна быть завершена в некоторой точке путем ее фиксации или отката. Чтобы зафиксировать транзакцию, используйте команду `COMMIT TRANSACTION`, которую можно записать так же, как `COMMIT TRAN`, `COMMIT WORK` или просто `COMMIT`. Чтобы откатить транзакцию, необходимо выполнить команду `ROLLBACK TRANSACTION`, обозначающуюся как `ROLLBACK TRAN`, `ROLLBACK WORK` или просто `ROLLBACK`.

Транзакции могут быть вложенными, т. е. можно помещать транзакции внутрь других транзакций, а также они могут входить в состав пакетов в коде T-SQL.

Уровни и состояния транзакций

Уровень транзакции и ее состояние можно определить с помощью двух системных функций.

- Функцию `@@TRANCOUNT` можно запросить для того, чтобы узнать уровень вложенности транзакций.

- Уровень, равный 0, указывает, что в данный момент код находится не внутри транзакции.
 - Уровень больше 0 указывает, что транзакция активна, при этом если значение больше 1, то оно указывает уровень вложенности для вложенных транзакций.
- Функцию XACT_STATE() можно запросить для определения состояния транзакции.
- Состояние, равное 0, указывает на то, что это неактивная транзакция.
 - Состояние, равное 1, указывает, что это незафиксированная транзакция, которая может быть зафиксирована, но не известен уровень ее вложенности.
 - Состояние, равное -1, указывает, что имеется незафиксированная транзакция, но она не может быть зафиксирована из-за предшествующей фатальной ошибки.

Эти две функции дополняют друг друга: функция @@TRANCOUNT не показывает транзакции, которые не могут быть зафиксированы, а функция XACT_STATE() не показывает уровень вложенности транзакции.

Режимы транзакций

В SQL Server существуют три режима пользовательских транзакций, т. е. три способа работы с транзакциями:

- автофиксация (autocommit);
- неявная транзакция (implicit transaction);
- явная транзакция (explicit transaction).

Это режимы, в которых работает код при использовании транзакций в SQL Server. Они не изменяют поведение транзакций.

Режим автоматической фиксации

В режиме автоматической фиксации модификация данных и инструкции DDL языка T-SQL выполняются как транзакция, которая будет автоматически зафиксирована, если инструкция выполнена без ошибок, в противном случае будет выполнен откат транзакции.

Режим автоматической фиксации является режимом управления транзакциями по умолчанию. На простой диаграмме состояний на рис. 12.1 показан режим автоматической фиксации.

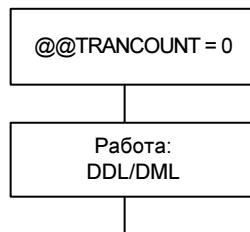


Рис. 12.1. Транзакция в режиме автоматической фиксации, не требующая инструкции COMMIT

В режиме автоматической фиксации не требуется никаких других команд управления транзакциями, таких как BEGIN TRAN, ROLLBACK TRAN или COMMIT TRAN. Более того, значение @@TRANCOUNT (для сеанса пользователя) обычно не может быть определено для этой команды, хотя оно может быть в триггере инструкции модификации данных. Любые изменения в базе данных обрабатываются автоматически, инструкция за инструкцией, как транзакции. Помните, автоматическая фиксация — это режим по умолчанию в SQL Server.

Режим неявных транзакций

В режиме неявных транзакций, когда выполняется одна или более инструкция DML или DDL либо инструкция SELECT, SQL Server запускает транзакцию, увеличивает @@TRANCOUNT, но не выполняет автоматической фиксации или отката инструкции. Для завершения транзакции необходимо интерактивно выполнять инструкции COMMIT или ROLLBACK, даже если единственной инструкцией будет инструкция SELECT.

Режим неявных транзакций не является режимом по умолчанию в SQL Server. Его необходимо задать с помощью такой команды:

```
SET IMPLICIT_TRANSACTIONS ON;
```

Так же можно использовать следующую команду. Но эта команда только запустит первую команду:

```
SET ANSI_DEFAULTS ON;
```

Как только вы начинаете выполнять какие-то действия, т. е. вносить изменения в базу данных, транзакция начинается автоматически. На рис. 12.2 показано, как это происходит.

Как только вы вводите какую-либо команду для изменения данных, значение @@TRANCOUNT становится равным 1, указывая, что вы находитесь на первом уровне

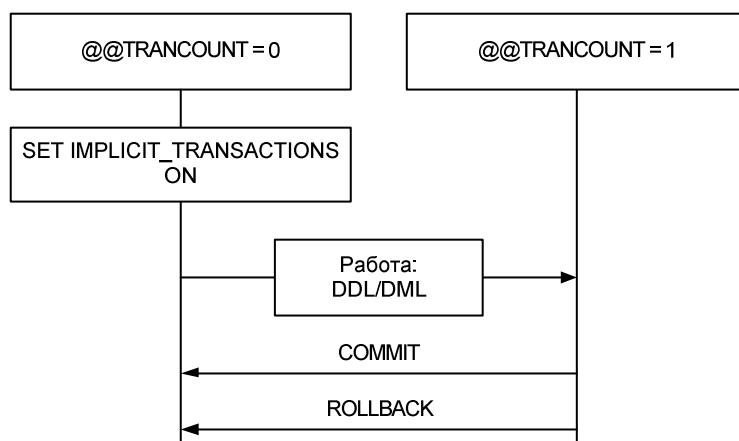


Рис. 12.2. Неявная транзакция с использованием инструкций COMMIT или ROLLBACK

вложенности транзакции. Затем необходимо вручную запустить команду `COMMIT` или `ROLLBACK` для завершения транзакции. Если вы запустите дополнительные инструкции DML или DDL, они также станут частью этой транзакции.

К преимуществам использования неявных транзакций относятся следующие:

- можно откатить неявную транзакцию после того, как была выполнена команда;
- поскольку инструкцию `COMMIT` нужно запускать автоматически, у вас есть возможность обнаружить ошибки после окончания выполнения команды.

К недостаткам неявных транзакций относятся следующие.

- Любые блокировки, вызванные командой, сохраняются до завершения транзакции. Поэтому вы можете заблокировать других пользователей при выполнении их задач.
- Поскольку это нестандартный метод использования SQL Server, вы должны постоянно помнить о необходимости его установки для вашего сеанса.
- Режим неявных транзакций плохо работает с явными транзакциями, поскольку приводит к неожиданному увеличению значения `@@TRANCOUNT` до 2.
- Если вы забудете зафиксировать неявную транзакцию, то можете оставить блокировки открытыми. Помните, что неявные транзакции могут входить в пакеты.

К СВЕДЕНИЮ Неявные транзакции

Дополнительную информацию о неявных транзакциях можно найти в электронной документации к SQL Server 2012 в разделе "SET IMPLICIT_TRANSACTIONS (Transact-SQL)" по адресу <http://msdn.microsoft.com/ru-ru/library/ms187807.aspx>.

Режим явных транзакций

Явная транзакция выполняется, если для запуска транзакции явно указана команда `BEGIN TRANSACTION` или `BEGIN TRAN`. На рис. 12.3 показана диаграмма состояний для явной транзакции.

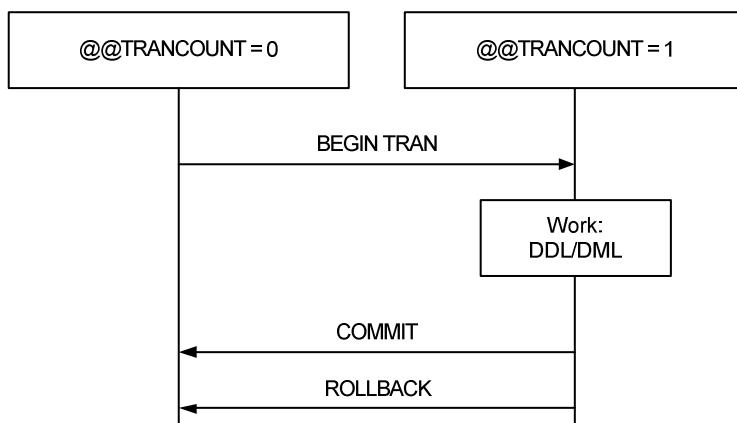


Рис. 12.3. Явная транзакция с инструкцией `COMMIT` или `ROLLBACK`

При использовании явной транзакции, как только вы запускаете команду BEGIN TRAN, значение @@TRANCOUNT увеличивается на 1. Затем вы запускаете команды DML или DDL и, когда они выполнены, запускаете инструкции COMMIT или ROLLBACK.

Явные транзакции можно выполнять интерактивно в диалоге или в коде, как, например, хранимые процедуры.

Явные транзакции могут использоваться в режиме неявных транзакций, но если вы запускаете явную транзакцию для сеанса, работающего в режиме неявных транзакций, значение @@TRANCOUNT будет увеличено с 0 до 2 сразу после команды BEGIN TRAN. В действительности вы получаете вложенную транзакцию. Поэтому считается не лучшим решением позволять значению @@TRANCOUNT увеличиваться не на 1 при использовании неявных транзакций.

Что происходит, когда какие-либо изменения данных или инструкции DDL вызывают ошибку? Некоторые ошибки приводят к откату всей транзакции, тогда как другие, такие как неправильный внешний ключ, не вызывают отката транзакции целиком. Для обеспечения правильного выполнения транзакции необходимо добавить в код обработку ошибок.

СОВЕТ Подготовка к экзамену

Обратите внимание, транзакции могут охватывать пакеты. Это относится и к неявным, и к явным транзакциям, т. е. инструкциям GO. Но часто считается более правильным убедиться в том, что каждая транзакция существует в одном пакете.

Вложенные транзакции

Когда явные транзакции вложены, т. е. расположены одна в другой, они называются *вложенными транзакциями*. Поведение инструкций COMMIT и ROLLBACK изменяется для вложенных транзакций.

На рис. 12.4 показана двухуровневая вложенная транзакция, в которой выполнена только инструкция COMMIT.



СОВЕТ Подготовка к экзамену

Внутренняя инструкция COMMIT не имеет фактического влияния на транзакцию, только увеличивает @@TRANCOUNT на 1. Только самая внешняя инструкция COMMIT, та, которая выполняется при @@TRANCOUNT = 1, действительно фиксирует транзакцию.

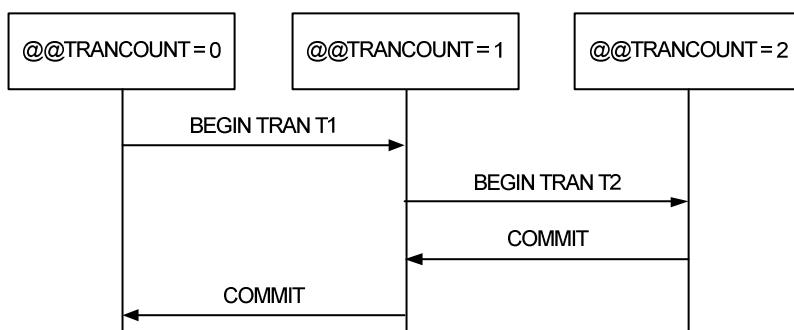


Рис. 12.4. Финальная самая внешняя инструкция COMMIT во вложенной транзакции

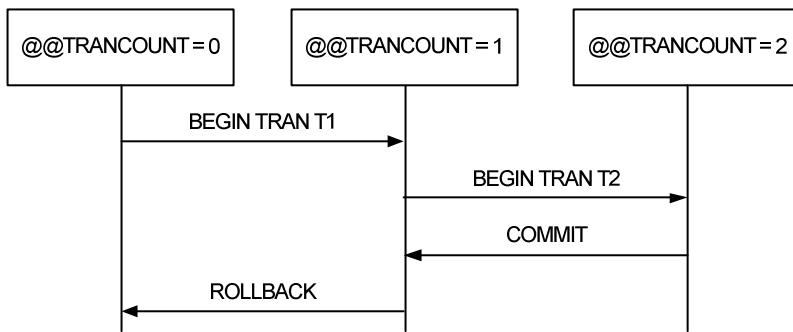


Рис. 12.5. Конечная инструкция ROLLBACK, выполняющая откат всей транзакции

На рис. 12.5 показана двухуровневая вложенная транзакция с командой ROLLBACK, запущенной на самом внешнем уровне.

Как уже говорилось ранее, внутренняя инструкция COMMIT не оказывает другого влияния, чем уменьшение значения @@TRANCOUNT.

Наконец, на рис. 12.6 показана вложенная транзакция, в которой команда ROLLBACK запущена раньше какой-либо инструкции COMMIT.

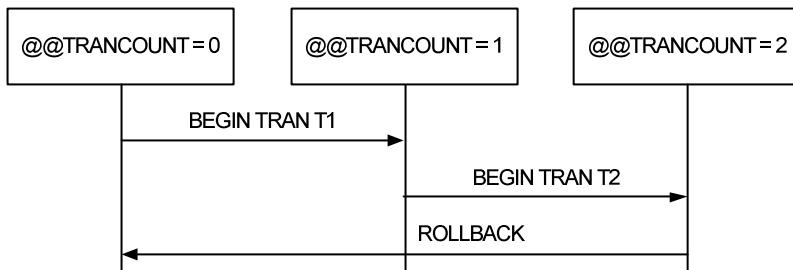


Рис. 12.6. Вложенная транзакция с инструкцией ROLLBACK, выполняющей откат всей транзакции

СОВЕТ

Подготовка к экзамену

Обратите внимание, не имеет значения, на каком уровне запущена команда ROLLBACK. Транзакция может содержать только одну команду ROLLBACK, которая выполнит откат всей транзакции и сбросит счетчик @@TRANCOUNT в 0.

Разметка транзакции

Можно дать имя явной транзакции, поместив его после инструкции BEGIN TRAN. Имена транзакций должны следовать правилам, установленным для идентификаторов SQL Server; однако SQL Server распознает только первые 32 символа как уникальное имя и игнорирует все оставшиеся символы, поэтому все имена транзакций должны быть длиной 32 символа или меньше. Имя транзакции выводится в столбце name функции sys.dm_tran_active_transactions, как показано в следующем примере:

```

USE TSQl2012;
BEGIN TRANSACTION Tran1;
  
```

Обратите внимание, SQL Server записывает имена транзакций только для самых внешних транзакций. Если имеются вложенные транзакции, имена всех вложенных транзакций игнорируются.

Именованные транзакции используются для помещения метки в журнал транзакций, чтобы указать точку, в которую могут быть восстановлены одна или более баз данных. Когда транзакция записывается в журнал транзакций базы данных, метка транзакции так же записывается, как показано в следующем примере:

```
USE TSQL2012;
BEGIN TRAN Tran1 WITH MARK;
-- <работа транзакции>
COMMIT TRAN; -- или ROLLBACK TRAN
-- <прочая работа>
```

Если позже потребуется восстановить базу данных в метке транзакции, можно выполнить следующий код:

```
RESTORE DATABASE TSQL2012 FROM DISK = 'C:\SQLBackups\TSQL2012.bak'
WITH NORECOVERY;
GO
RESTORE LOG TSQL2012 FROM DISK = 'C:\SQLBackups\TSQL2012.trn'
WITH STOPATMARK = 'Tran1';
GO
```

При использовании предложения `WITH MARK` необходимо помнить следующее.

- Вы должны использовать имя транзакции с ключевым словом `WITH STOPATMARK`.
- Вы можете поместить описание после предложения `WITH MARK`, но SQL Server будет его игнорировать.
- Вы можете выполнить восстановление на момент непосредственно перед транзакцией с помощью предложения `STOPBEFOREMARK`.
- Можно восстановить базу данных из копии с использованием ключевых слов `WITH STOPATMARK` или `STOPBEFOREMARK`.
- Вы можете добавить параметр `RECOVERY` в списке `WITH`, но это не будет иметь никакого эффекта.

Дополнительные параметры транзакции

Существует множество дополнительных более специфичных параметров транзакций. К ним относятся следующие.

- **Точки сохранения.** Это точки внутри транзакций, которые можно использовать для отката выборочной части работы.
 - Можно определить точку сохранения с помощью команды `SAVE TRANSACTION <savepoint name>`.
 - Инструкция `ROLLBACK` должна ссылаться на точку сохранения. В противном случае, если эта инструкция является неточной, она приведет к откату всей транзакции.

- **Межбазовые транзакции.** Транзакция может охватывать одну или более баз данных на одном экземпляре SQL Server без каких-либо дополнительных действий на стороне пользователя.
 - SQL Server сохраняет ACID-свойства межбазовых транзакций без каких-либо дополнительных условий.
 - Однако при использовании межбазовых транзакций существуют ограничения на зеркалирование базы данных. Межбазовая транзакция может быть не сохранена после перехода на другой ресурс одной из баз данных.
- **Распределенные транзакции.** Транзакция может распространяться на несколько серверов с помощью связанного сервера. Такая транзакция называется распределенной (в противоположность локальной) транзакцией.
 - Если транзакция распространяется на несколько серверов с помощью связанного сервера, эта транзакция считается распределенной транзакцией, и SQL Server вызывает координатор распределенных транзакций (Distributed Transaction Coordinator, MSDTC).
 - Транзакция, ограниченная одной базой данных или межбазовой транзакцией, считается локальной транзакцией в противоположность распределенной транзакции, которая пересекает границы экземпляра SQL Server.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Сколько команд ROLLBACK должно быть выполнено во вложенной транзакции для того, чтобы выполнить ее откат?
2. Сколько инструкций COMMIT должно быть выполнено во вложенной транзакции, чтобы гарантировать фиксацию всей транзакции полностью?

Ответы на контрольные вопросы

1. Только одна команда ROLLBACK. Команда ROLLBACK всегда делает откат всей транзакции, независимо от того, сколько уровней вложенности имеет транзакция.
2. Одна инструкция COMMIT на каждый уровень вложенной транзакции. Только последняя инструкция COMMIT действительно фиксирует всю транзакцию.

К СВЕДЕНИЮ Неявные и явные транзакции

Дополнительную информацию о неявных транзакциях можно найти в электронной документации к SQL Server 2012 в разделе "SET IMPLICIT_TRANSACTIONS (Transact-SQL)" по адресу <http://msdn.microsoft.com/ru-ru/library/ms187807.aspx>.

Основные блокировки

Чтобы сохранить изоляцию транзакций, SQL Server реализует набор протоколов блокирования. На базовом уровне существуют два основных режима блокировок:

- совместимые блокировки (shared locks) используются для сеансов, выполняющих чтение данных, т. е. для операций считывания;

- **монопольные блокировки** (exclusive locks) используются для изменений данных, т. е. для операций записи.

ПРИМЕЧАНИЕ Расширенные режимы блокировки

Существуют и другие дополнительные режимы блокировки, которые называются блокировками обновления, блокировками с намерением и блокировками схемы и используются для специальных целей.

Когда сеанс попытается изменить данные, SQL Server будет стараться обеспечить защиту монопольной блокировки на обрабатываемых объектах. Эти монопольные блокировки всегда возникают в случае транзакции, даже если это режим автоматической фиксации, и сеанс не запускает явную транзакцию. Когда в сеансе имеется монопольная блокировка на объекте, таком как строка, таблица или некий системный объект, никакая другая транзакция не может изменить данные, пока эта транзакция не будет либо зафиксирована, либо откочена. За исключением особых уровней изоляции, другие сеансы не могут даже читать в монопольном режиме заблокированные объекты.

Совместимость блокировок

Если сеанс выполняет только чтение данных, по умолчанию SQL Server запускает только очень короткие совмещаемые блокировки на ресурсе, таком как строка или таблица. Два или более сеансов могут читать одни и те же объекты, поскольку совмещаемые блокировки совместимы с другими совмещаемыми блокировками. Но когда сеанс имеет ресурс с монопольной блокировкой, другие сеансы не могут выполнять чтение этого ресурса, а также не могут выполнять запись в него.

В табл. 12.1 приведены сведения о совместимости базовых блокировок между совмещаемыми и монопольными блокировками.

Таблица 12.1. Совместимость между совмещаемой и монопольной блокировками

Запрашиваемая	Предоставляемая	
	Exclusive (X)	Shared (S)
Монопольная	Нет	Нет
Совмещаемая	Нет	Да

ПРИМЕЧАНИЕ Совместимость блокировок

Только совмещаемые блокировки совместимы друг с другом. Монопольная блокировка несовместима ни с каким типом блокировок.

Какие же существуют типы ресурсов? Виды ресурсов многочисленны и разнообразны, но наиболее детализированным ресурсом для SQL Server является строка таблицы. Однако в SQL Server может понадобиться наложить блокировку на целую страницу или даже целую таблицу.

Блокирование

Блокирование происходит в случае, когда один сеанс выполняет монопольную блокировку ресурса, не позволяя другому сеансу никаким образом получить возможность блокировки этого ресурса. В транзакции монопольные блокировки удерживаются до конца транзакции, поэтому если первый сеанс выполняет транзакцию, второй сеанс должен ждать, пока первый либо не зафиксирует, либо не откатит транзакцию. Два сеанса не могут выполнять запись в один и тот же ресурс (такой как таблица или строка) одновременно, поэтому модули записи могут блокировать другие модули записи.

Не только два запроса монопольной блокировки на одном и том же ресурсе могут привести к блокированию. Монопольная блокировка также может блокировать запрос на чтение одних и тех данных, если модуль чтения запрашивает совмещаемую блокировку, потому что монопольная блокировка также несовместима с совмещаемой блокировкой. В транзакции, работающей на уровне изоляции по умолчанию READ COMMITTED, совмещаемые блокировки освобождаются, как только данные прочитаны, и они не удерживаются до конца транзакции, за исключением более высоких уровней изоляции.

Взаимоблокировка

Взаимоблокировка вызывается при взаимном блокировании между двумя или более сеансами. Иногда последовательности блокировок между сеансами не могут быть разрешены просто ожиданием завершения одной из транзакций. Это происходит из-за циклических связей между двумя или более командами. SQL Server определяет такой цикл как взаимоблокировку между двумя сеансами, прерывает одну из транзакций и возвращает клиенту сообщение об ошибке 1205.



Контрольные вопросы

1. Могут ли модули чтения (совмещаемые блокировки) блокировать модули чтения?
2. Могут ли модули чтения блокировать модули записи (монопольные блокировки)?

Ответы на контрольные вопросы

1. Нет, поскольку совмещаемые блокировки совместимы с другими совмещаемыми блокировками.
2. Да, даже кратковременный запрос, поскольку любой запрос монопольной блокировки должен ждать, пока совмещаемая блокировка не будет освобождена.

Можно создать взаимоблокировку и затем разрешить ее с помощью шагов, приведенных в табл. 12.2. Сначала нужно запустить SSMS и открыть два пустых окна запроса. Поместите код сеанса 1 в одно окно запроса, а код для сеанса 2 — в другое окно запроса. Затем выполните оба сеанса шаг за шагом, переходя от одного окна к другому, как этого требует код. Обратите внимание, что у каждой транзакции есть блокировка на ресурсе, блокировку на который запрашивает и другая транзакция.

Таблица 12.2. Две транзакции, выполняющие одни и те же изменения в таблице в противоположной последовательности, приводят к взаимной блокировке

Сеанс 1	Сеанс 2
USE TSQL2012; BEGIN TRAN;	USE TSQL2012; BEGIN TRAN;
UPDATE HR.Employees SET Region = N'10004' WHERE empid = 1	
	UPDATE Production.Suppliers SET Fax = N'555-1212' WHERE supplierid = 1
UPDATE Production.Suppliers SET Fax = N'555-1212' WHERE supplierid = 1	
<заблокировано>	UPDATE HR.Employees SET phone = N'555-9999' WHERE empid = 1
	<заблокировано>

Возникает взаимоблокировка; одна транзакция завершается, а другая прерывается, сообщение об ошибке 1205 отправляется клиенту.

Не забудьте выполнить команду ROLLBACK для транзакции, завершившейся успешно, как показано в следующей строке:

IF @@TRANCOUNT > 0 ROLLBACK	IF @@TRANCOUNT > 0 ROLLBACK
-----------------------------	-----------------------------

В табл. 12.3 показано, что если бы обе транзакции выполняли обновления в таблицах в одной и той же последовательности, взаимоблокирование не произошло бы. Чтобы убедиться в этом, откройте два новых пустых окна запроса в SSMS и выполните код из табл. 12.3 шаг за шагом. Поместите код сеанса 1 в одно окно запроса, а код для сеанса 2 — в другое окно запроса. Затем выполните оба сеанса шаг за шагом, переходя от одного окна к другому, как этого требует код. Транзакция сеанса 1 завершается, поскольку она никогда не блокируется. Транзакция сеанса 2 блокируется до окончания сеанса 1. Но транзакция сеанса 2 никогда не блокирует ресурс, который нужен сеансу 1.

Таблица 12.3. Когда транзакции двух сеансов выполняют те же изменения, но теперь в одинаковой последовательности, взаимоблокировка не возникает

Сеанс 1	Сеанс 2
USE TSQL2012; BEGIN TRAN;	USE TSQL2012; BEGIN TRAN;

Таблица 12.3 (окончание)

Сеанс 1	Сеанс 2
UPDATE HR.Employees SET Region = N'10004' WHERE empid = 1	
	UPDATE HR.Employees SET phone = N'555-9999' WHERE empid = 1
	<заблокировано>
UPDATE Production.Suppliers SET Fax = N'555-1212' WHERE supplierid = 1	
COMMIT TRAN;	
	<заблокировано>
	UPDATE Production.Suppliers SET Fax = N'555-1212' WHERE supplierid = 1
	COMMIT TRAN;

КОНТРОЛЬНЫЕ ВОПРОСЫ

- Если две транзакции никогда не блокируют друг друга, может ли возникнуть между ними взаимоблокировка?
- Может ли инструкция SELECT участвовать во взаимоблокировке?

Ответы на контрольные вопросы

- Нет. Чтобы возникла взаимоблокировка, каждая транзакция должна иметь заблокированный ресурс, который нужен другой транзакции, что приводит к взаимному блокированию.
- Да. Если инструкция SELECT блокирует некоторый ресурс, не позволяя завершиться другой транзакции, и сама инструкция SELECT не может завершиться, т. к. она заблокирована какой-то транзакцией, возникает цикл взаимной блокировки.

К СВЕДЕНИЮ Устранение проблем с взаимоблокировками

Существует огромное количество информации о проблемах взаимоблокировок и их устранении. Начните с электронной документации по SQL Server 2008 R2 раздела "Обнаружение и устранение взаимоблокировок" по адресу

[http://msdn.microsoft.com/ru-ru/library/ms178104\(SQL.105\).aspx](http://msdn.microsoft.com/ru-ru/library/ms178104(SQL.105).aspx).

Уровни изоляции транзакций

Среди ACID-свойств транзакций SQL Server никогда не нарушает требований атомарности, согласованности и устойчивости транзакции базы данных. Однако сте-

пень изоляции может меняться для модулей чтения в зависимости от настроек, применяемых в их сеансах.

В течение времени, когда транзакция изменяет какие-либо данные, SQL Server никогда не допускает изменения этих данных никакой другой транзакцией, пока первая транзакция не завершится, и при этом первая транзакция не может изменить данные, изменяемые другими транзакциями, пока они не завершатся. Поэтому блокирование и взаимоблокирование всегда возможно, когда транзакции изменяют данные. Модули записи всегда блокируют модули записи, и монопольные блокировки в одной транзакции не могут быть совместимы с монопольными блокировками в другой.

Но блокирование и взаимоблокирование можно увеличить или уменьшить путем изменения степени изолированности ACID-свойств транзакции. SQL Server позволяет одной транзакции читать данные другой транзакции или разрешает изменение данных другими транзакциями, если данная транзакция выполняет только чтение этих данных, с помощью *настройки уровня изоляции транзакции*.



К наиболее часто используемым уровням изоляции относятся следующие.

- READ COMMITTED.** Это уровень изоляции по умолчанию. Все модули чтения в данном сеансе будут только выполнять чтение изменений данных, которые были зафиксированы. Поэтому все инструкции `SELECT` будут пытаться получить совмещаемые блокировки, и любые базовые ресурсы данных, которые изменяются в разных сеансах и, следовательно, имеют монопольные блокировки, будут блокировать сеанс `READ COMMITTED`.
- READ UNCOMMITTED.** Этот уровень изоляции позволяет модулям чтения читать незафиксированные данные. Эта настройка удаляет совмещаемые блокировки, полученные инструкциями `SELECT`, так что модули чтения больше уже не блокируются модулями записи. Но результаты инструкции `SELECT` могут считывать незафиксированные данные, которые были изменены в течение транзакции и позже откатывались к их первоначальному состоянию. Это называется *"грязным"* чтением данных.
- READ COMMITTED SNAPSHOT.** Фактически это не новый уровень изоляции; это дополнительный способ использования уровня изоляции по умолчанию `READ COMMITTED`, уровень изоляции по умолчанию в базе данных Windows Azure SQL. Этот уровень изоляции имеет следующие особенности.
 - Часто называемый сокращенно RCSI, он использует базу данных `tempdb` для сохранения исходных версий измененных данных. Эти версии хранятся ровно столько, сколько они необходимы, чтобы позволить модулям чтения (т. е. инструкциям `SELECT`) читать базовые данные в их исходном состоянии. В результате инструкциям `SELECT` больше не нужны совмещаемые блокировки на базовом ресурсе при выполнении чтения зафиксированных данных.
 - Настройка `READ COMMITTED SNAPSHOT` устанавливается на уровне базы данных и является постоянным свойством базы данных.

- RCSI — это не отдельный уровень изоляции; это только другой способ реализации уровня READ COMMITTED, предотвращающий блокирование модулей чтения модулями записи.
- RCSI — это уровень изоляции по умолчанию для базы данных Windows Azure SQL.

Иногда вам может понадобиться более строгая изоляция транзакций, помимо уровня изоляции по умолчанию READ COMMITTED. Следующие уровни изоляции обеспечивают более строгий контроль над тем, какие данные могут быть прочитаны между транзакциями. Поскольку они могут приводить к еще большему блокированию или большим затратам, они не используются так часто, как более слабые уровни изоляции.

- REPEATABLE READ. Этот уровень изоляции, также устанавливаемый на уровне сеанса, гарантирует, что данные, считанные в транзакции, могут быть позднее снова прочитаны в транзакции. Не допускаются операции обновления и удаления уже выбранных строк. В результате совмещаемые блокировкидерживаются до конца транзакции. Однако транзакция может видеть новые строки, добавленные после первой операции чтения; это называется *фантомным чтением*.
- SNAPSHOT. Этот уровень изоляции также использует управление версиями строк в базе данных tempdb (как это делает RCSI). Он разрешен как постоянное свойство базы данных и поэтому устанавливается на отдельную транзакцию. Транзакция, использующая уровень изоляции SNAPSHOT, сможет повторить любую операцию чтения, и при этом она не будет видеть никаких фантомных чтений. Новые строки могут быть добавлены в таблицу, но транзакция не будет их видеть. Поскольку она использует управление версиями строк, уровень изоляции SNAPSHOT не требует совмещаемых блокировок на базовых данных.
- SERIALIZABLE. Этот уровень изоляции является наиболее строгим и устанавливается на сеанс. На этом уровне все операции чтения являются повторяемыми, и новые строки не разрешены в базовых таблицах, что может удовлетворять условиям инструкций SELECT в транзакции.

COBET

Подготовка к экзамену

Уровни изоляции устанавливаются на сеанс. Если иной уровень изоляции не установлен на сеанс, все транзакции будут выполняться с использованием уровня изоляции по умолчанию, READ COMMITTED, для локальных экземпляров SQL Server — тоже с уровнем READ COMMITTED. В базе данных Windows Azure SQL уровень изоляции по умолчанию — READ COMMITTED SNAPSHOT.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Если ваш сеанс использует уровень изоляции READ COMMITTED, может ли один из ваших запросов выполнять чтение незафиксированных данных?
2. Существует ли способ предотвратить блокирование модулей записи модулями чтения и при этом гарантировать, что модули чтения видят только зафиксированные данные?

Ответы на контрольные вопросы

- Да, если запрос использует табличные подсказки WITH (NOLOCK) или WITH (READUNCOMMITTED). Значение сеанса для уровня изоляции не изменяется, меняются только характеристики чтения таблицы.
- Да, это задача параметра в уровне изоляции READ COMMITTED. Модули чтения видят более ранние версии изменений данных для текущих транзакций, но не незафиксированные в данный момент данные.

ПРАКТИКУМ Реализация транзакций

В этом практикуме вам предстоит реализовать ряд транзакций, используя среду SQL Server 2012 Management Studio и базу данных TSQL2012. Для некоторых транзакций вы будете запускать инструкции T-SQL в параллельных сеансах, переходя от одного сеанса к другому. Вы будете выполнять сценарии таким образом, чтобы видеть, как они влияют друг на друга.

Задание 1. Работа с режимами транзакций

В этом задании вам предстоит работать с основными режимами транзакций.

- Начните работу с неявной транзакции — откройте SSMS, а затем пустое окно запроса. Выполните следующий код. Выполняйте каждую команду последовательно, шаг за шагом. Обратите внимание на значение @@TRANCOUNT.

```
USE TSQL2012;
SET IMPLICIT_TRANSACTIONS ON;
SELECT @@TRANCOUNT; -- 0
SET IDENTITY_INSERT Production.Products ON;
-- Запустите команду DML или DDL
INSERT INTO Production.Products(productid, productname, supplierid,
categoryid, unitprice, discontinued)
VALUES(101, N'Test2: New productid', 1, 1, 18.00, 0);
SELECT @@TRANCOUNT; -- 1
COMMIT TRAN;
SET IDENTITY_INSERT Production.Products OFF;
SET IMPLICIT_TRANSACTIONS OFF;
-- Удалите введенную строку
DELETE FROM Production.Products WHERE productid = 101; -- Стока удалена
```

- Далее вы будете работать с явной транзакцией. Выполните следующий код. Обратите внимание на значение @@TRANCOUNT.

```
USE TSQL2012;
SELECT @@TRANCOUNT; -- 0
BEGIN TRAN;
SELECT @@TRANCOUNT; -- 1
SET IDENTITY_INSERT Production.Products ON;
INSERT INTO Production.Products(productid, productname, supplierid,
categoryid, unitprice, discontinued)
VALUES(101, N'Test2: New productid', 1, 1, 18.00, 0);
```

```

SELECT @@TRANCOUNT; -- 1
SET IDENTITY_INSERT Production.Products OFF;
COMMIT TRAN;
-- Удаление вставленной строки
DELETE FROM Production.Products WHERE productid = 101; -- Стока удалена

```

3. Для работы с вложенной транзакцией с помощью COMMIT TRAN выполните следующий код. Обратите внимание, что значение @@TRANCOUNT увеличивается до 2.

```

USE TSQL2012; SELECT @@TRANCOUNT; -- = 0
BEGIN TRAN;
SELECT @@TRANCOUNT; -- = 1
BEGIN TRAN;
SELECT @@TRANCOUNT; -- = 2
-- Модификация данных события или команда DDL
COMMIT
SELECT @@TRANCOUNT; -- = 1
COMMIT TRAN;
SELECT @@TRANCOUNT; -- = 0

```

4. Для работы с вложенной транзакцией с помощью ROLLBACK TRAN выполните следующий код. Обратите внимание, что значение @@TRANCOUNT увеличивается до 2, но требуется только одна операция ROLLBACK.

```

USE TSQL2012;
SELECT @@TRANCOUNT; -- = 0
BEGIN TRAN;
SELECT @@TRANCOUNT; -- = 1
BEGIN TRAN;
SELECT @@TRANCOUNT; -- = 2
-- Модификация данных события или команда DDL
ROLLBACK; -- откат всей транзакции в этой точке
SELECT @@TRANCOUNT; -- = 0

```

Задание 2. Работа с блокированием и взаимоблокированием

В этом задании вам предстоит работать с двумя сценариями: блокированием и взаимоблокированием.

- На этом шаге вы будете работать с модулями записи, блокирующими другие модули записи. Откройте SSMS и два пустых окна запросов. Выполните код параллельно, как показано в табл. 12.4. Выполняйте действия шаг за шагом. Когда блокировки станут несовместимыми, сеанс, запрашивающий несовместимую блокировку, должен подождать и считается заблокированным. Сеанс 1 получает монопольную блокировку на строку, которая изменяется. Почти в то же время или вскоре после этого сеанс 2 пытается обновить ту же строку. Сеанс 1 не освободил свою монопольную блокировку на строке, поскольку в транзакции все монопольные блокировки удерживаются до конца транзакции. Таким образом, сеанс 2 должен ждать, пока сеанс 1 будет зафиксирован либо откачен, и блокировка на строке будет снята, чтобы закончить ее обновление.

ПРИМЕЧАНИЕ Модули записи блокируют другие модули записи

Монопольная блокировка несовместима с аналогичным запросом монопольной блокировки.

Таблица 12.4. Два сеанса с несовместимыми монопольными блокировками

Сеанс 1	Сеанс 2
USE TSQL2012; BEGIN TRAN;	USE TSQL2012;
UPDATE HR.Employees SET postalcode = N'10004' WHERE empid = 1;	UPDATE HR.Employees SET phone = N'555-9999' WHERE empid = 1;
<другая работа>	<заблокировано>
COMMIT TRAN;	
	<возвращенные результаты>
-- Очистка: UPDATE HR.Employees SET postalcode = N'10003' WHERE empid = 1;	

2. На этом шаге вы будете работать с модулями записи, блокирующими модули чтения. Откройте SSMS и два пустых окна запросов. Выполните код параллельно, как показано в табл. 12.5. Выполняйте действия шаг за шагом. В данном случае сеанс 2 должен получить совмещаемую блокировку на строку в таблице HR.Employees, которую сеанс 1 монопольно заблокировал. Поскольку совмещаемые блокировки несовместимы с монопольными блокировками на одном и том же ресурсе, сеанс 2 должен ждать, пока сеанс 1 освободит свою блокировку, закончив транзакцию.

ПРИМЕЧАНИЕ Модули записи блокируют модули чтения

Монопольная блокировка также несовместима с запросом совмещаемой блокировки.

Таблица 12.5. Два сеанса, иллюстрирующие несовместимость между монопольной блокировкой и запросом совмещаемой блокировки

Сеанс 1	Сеанс 2
USE TSQL2012; BEGIN TRAN;	USE TSQL2012;
UPDATE HR.Employees SET postalcode = N'10005' WHERE empid = 1	SELECT lastname, firstname FROM HR.Employees
	<заблокировано>
COMMIT TRAN;	
	<возвращенные результаты>

Таблица 12.5 (окончание)

Сеанс 1	Сеанс 2
-- Очистка: UPDATE HR.Employees SET postalcode = N'10003' WHERE empid = 1;	

Задание 3. Работа с уровнями изоляции транзакций

В данном задании вы будете работать с наиболее распространенными уровнями изоляции с использованием базы данных TSQL2012, проверяя влияние уровней изоляции на блокирование между двумя сеансами.

- На этом шаге вы будете работать с уровнем изоляции `READ COMMITTED`. Откройте SSMS и два пустых окна запросов. Выполните код параллельно, как показано в табл. 12.6. Выполняйте действия шаг за шагом. Обратите внимание, что инструкция `SELECT` в сеансе 2 заблокирована, но освободится, как только сеанс 1 завершит свою транзакцию.

Таблица 12.6. Уровень изоляции `READ COMMITTED`, приводящий к блокированию модулей чтения модулями записи

Сеанс 1	Сеанс 2
USE TSQL2012; BEGIN TRAN;	USE TSQL2012; SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE HR.Employees SET postalcode = N'10006' WHERE empid = 1;	
	SELECT lastname, firstname FROM HR.Employees;
	<заблокировано>
COMMIT TRAN;	
	<возвращенные результаты>
-- Очистка: UPDATE HR.Employees SET postalcode = N'10003' WHERE empid = 1;	

- На этом шаге вы будете работать с уровнем блокировки `READ UNCOMMITTED`. На уровне изоляции `READ COMMITTED` даже модулю чтения, возможно, придется ждать завершения транзакции, приводя к блокированию и даже взаимоблокированию в занятых системах. Единственным способом снижения такого блокиро-

вания является разрешение модулям чтения выполнять чтение незафиксированных данных с помощью уровня изоляции READ UNCOMMITTED. Откройте SSMS и два пустых окна запросов. Выполните код параллельно, как показано в табл. 12.7. Выполняйте действия шаг за шагом. Обратите внимание, что инструкция SELECT в сеансе 2 теперь выполняет чтение незафиксированных данных.

Таблица 12.7. Уровень изоляции READ UNCOMMITTED, приводящий к чтению незафиксированных данных, которые позже могут быть откачены

Сеанс 1	Сеанс 2
USE TSQL2012; BEGIN TRAN; UPDATE HR.Employees SET region = N'1004' WHERE empid = 1;	USE TSQL2012; SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED; SELECT lastname, firstname, region FROM HR.Employees
	<results returned: region = 1004 for empid = 1>
ROLLBACK TRAN;	
<region для empid = 1 откачен в исходное значение>	SELECT lastname, firstname, region FROM HR.Employees;
	<Возвращенные результаты показывают region в исходном состоянии для empid = 1>
-- Очистка: UPDATE HR.Employees SET region = N'1003' WHERE empid = 1;	

3. На этом шаге используется табличная подсказка для реализации уровня изоляции READ UNCOMMITTED в одной команде. Вместо применения этого уровня изоляции к целому сеансу можно использовать его к отдельной команде для определенной таблицы с помощью табличной подсказки READUNCOMMITTED. Замените команду SELECT в коде для предыдущего шага на следующую, которая содержит табличную подсказку WITH (READUNCOMMITTED). Затем измените инструкцию, чтобы использовать табличную подсказку WITH (NOLOCK).

```
SELECT lastname, firstname
FROM HR.Employees WITH (READUNCOMMITTED);
```

ПРИМЕЧАНИЕ Табличная подсказка NOLOCK является устаревшей

Табличная подсказка NOLOCK считается устаревшей и не будет разрешена в инструкциях UPDATE и DELETE в последующих версиях SQL Server. Везде, кроме этого задания, используйте вместо нее табличную подсказку READUNCOMMITTED.

4. В этом шаге вы будете использовать READ COMMITTED SNAPSHOT. В SQL Server 2005 предложен более элегантный способ снижения блокирования, так чтобы модули записи могли более не блокировать модули чтения. Откройте SSMS и два пустых окна запросов. Выполните код параллельно, как показано в табл. 12.8. Выполняйте действия шаг за шагом. Обратите внимание, что инструкция SELECT в сеансе 2 более не блокируется и теперь выполняет чтение зафиксированных данных.

Таблица 12.8. Уровень изоляции RCSI, приводящий к выполнению чтения ранее зафиксированных данных без совмещаемых блокировок

Сеанс 1	Сеанс 2
USE TSQL2012; ALTER DATABASE TSQL2012 SET READ_COMMITTED_SNAPSHOT ON;	Use master
BEGIN TRAN;	USE TSQL2012;
UPDATE HR.Employees SET postalcode = N'10007' WHERE empid = 1;	SELECT lastname, firstname, postalcode FROM HR.Employees WHERE empid=1;
	<Возвращенные результаты показывают postalcode в исходном состоянии для empid = 1>
ROLLBACK TRAN;	
<postalcode для empid = 1 откачен в исходное значение>	
-- Очистка: UPDATE HR.Employees SET postalcode = N'10003' WHERE empid = 1;	

К СВЕДЕНИЮ Управление версиями строк в SQL Server и уровни изоляции

Дополнительную информацию о том, как в SQL Server реализовано управление версиями строк для параметра READ COMMITTED SNAPSHOT уровня READ COMMITTED и для уровня изоляции SNAPSHOT, можно найти в электронной документации по SQL Server 2008 R2 в разделе "Основные сведения об уровнях изоляции на основе управления версиями строк" по адресу [http://msdn.microsoft.com/ru-ru/library/ms189050\(SQL.105\).aspx](http://msdn.microsoft.com/ru-ru/library/ms189050(SQL.105).aspx).

Резюме занятия

- Все изменения данных в SQL Server происходят в контексте транзакции. Выполнение команды ROLLBACK на любом уровне вложенности транзакции немедленно откатывает всю транзакцию полностью.

- Каждая инструкция `COMMIT` уменьшает значение `@@TRANCOUNT` на 1, и только самая внешняя инструкция `COMMIT` фиксирует всю вложенную транзакцию.
- SQL Server использует блокирование для обеспечения изоляции транзакций.
- Взаимоблокировка может возникнуть между двумя или более сеансами, если каждый сеанс получил несовместимые блокировки, которые нужны другому сеансу для завершения его инструкции. Когда SQL Server видит взаимоблокировку, он выбирает один из сеансов и прерывает пакет.
- SQL Server обеспечивает ACID-свойство, называемое уровнем изоляции, с разной степенью строгости.
- Уровень изоляции `READ COMMITTED` — это уровень изоляции по умолчанию для локального SQL Server.
- Параметр изоляции `READ COMMITTED SNAPSHOT (RCSI)` уровня изоляции по умолчанию позволяет запросам на чтение получать доступ к ранее зафиксированным версиям монопольно заблокированных данных. Это может значительно снизить блокирование и взаимоблокирование. RCSI — это уровень изоляции по умолчанию в базе данных Windows Azure SQL.
- Уровень изоляции `READ UNCOMMITTED` разрешает сеансу считывать незафиксированные данные, известные как "грязные чтения".

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какие из следующих инструкций T-SQL выполняются автоматически в контексте транзакции? (Выберите все подходящие варианты.)
 - A. Команда `ALTER TABLE`.
 - B. Команда `PRINT`.
 - C. Команда `UPDATE`.
 - D. Команда `SET`.
2. Как команды `COMMIT` и `ROLLBACK` работают с вложенными транзакциями в языке T-SQL? (Выберите все подходящие варианты.)
 - A. Одна команда `COMMIT` фиксирует всю вложенную транзакцию.
 - B. Одна команда `ROLLBACK` выполняет откат всей вложенной транзакции.
 - C. Одна команда `COMMIT` фиксирует только один уровень вложенной транзакции.
 - D. Одна команда `ROLLBACK` выполняет откат только одного уровня вложенной транзакции.
3. Какая из следующих стратегий может помочь снизить блокирование и взаимоблокирование, уменьшив совмещаемые блокировки? (Выберите все подходящие варианты.)

- А. Добавьте в запросы табличную подсказку READUNCOMMITTED.
- В. Используйте параметр READ COMMITTED SNAPSHOT.
- С. Используйте уровень изоляции REPEATABLE READ.
- Д. Используйте уровень изоляции SNAPSHOT.

Занятие 2. Реализация обработки ошибок

При написании кода T-SQL, который выполняет изменение данных, как с помощью команд модификации данных, так и используя DDL-команды, особенно если они входят в явные транзакции или/и хранимые процедуры, обязательно следует включать в код обработки ошибок. SQL Server 2012 предоставляет практически полный набор команд структурированной обработки ошибок, которые могут предусмотреть практически любые сбойные ситуации. В этом занятии мы начнем с описания существующих в языке T-SQL сообщений об ошибках, а затем перейдем к методам обработки ошибок.

Изучив материал этого занятия, вы сможете:

- ✓ Описать части сообщения об ошибках языка T-SQL
- ✓ Объяснить, как реализуется неструктурная обработка ошибок
- ✓ Реализовать блок TRY/CATCH и инструкцию THROW
- ✓ Реализовать обработку ошибок в транзакциях

Продолжительность занятия — 40 минут.

Обнаружение и инициализация ошибок

Когда в SQL Server возникает ошибка при выполнении кода T-SQL, SQL Server генерирует условия возникновения ошибки и принимает определенные меры. Вы должны быть готовы к появлению возможных ошибок в вашем коде T-SQL и их обработке, чтобы не потерять контроль над вашим кодом.

Кроме того, в ваших программах вам может понадобиться тестировать ситуации, которые SQL Server не должен рассматривать как ошибочные, но они явно являются ошибками с точки зрения выполнения вашего кода. Например, если определенная таблица не содержит строк, вы можете не захотеть дальнейшего выполнения кода. Тогда такие ситуации нужно рассматривать как ошибочные и обработать эти ошибки.

Язык T-SQL предоставляет способы обнаружения ошибок SQL Server и инициализации пользовательских ошибок. Когда SQL Server генерирует условия возникновения ошибки, системная функция @@ERROR будет иметь положительное целочисленное значение, указывающее номер ошибки.

Если код T-SQL не входит в блок TRY/CATCH, сообщение об ошибке будет передано клиенту и не может быть перехвачена в коде T-SQL.

В дополнение к сообщениям об ошибках, которые генерирует SQL Server при возникновении ошибки, вы можете инициировать собственные ошибки с помощью двух команд:

- более старой команды RAISERROR;
- команды SQL Server 2012 THROW.

Любая из этих команд может использоваться, чтобы генерировать ваши собственные ошибки в коде T-SQL.

Анализ сообщений об ошибке

Далее приведен пример сообщения об ошибке от SQL Server 2012.

```
Msg 547, Level 16, State 0, Line 11
The INSERT statement conflicted with the FOREIGN KEY constraint
"FK_Products_Categories".
The conflict occurred in database "TSQL2012", table "Production.Categories",
column 'categoryid'.
```

Сообщения об ошибке в SQL Server состоят из четырех частей.

- Номер ошибки* (error number) — это целочисленное значение.
 - Сообщения об ошибках SQL Server нумеруются от 1 до 49 999.
 - Пользовательские сообщения об ошибках нумеруются с 50 001 и выше.
 - Ошибка с номером 50 000 зарезервирована для пользовательского сообщения, которое не имеет номера пользовательской ошибки.
- Уровень серьезности* (Severity level). В SQL Server определено 26 уровней серьезности с номерами от 0 до 25.
 - По общему правилу, ошибки с уровнем серьезности от 16 и выше автоматически записываются в журнал SQL Server и в журнал приложений Windows.
 - Ошибки с уровнем серьезности от 19 до 25 могут быть указаны только членами предопределенной роли сервера sysadmin.
 - Ошибки с уровнем серьезности от 20 до 25 считаются фатальными (неустранимыми) и приводят к разрыву соединений и откату всех открытых транзакций.
 - Ошибки с уровнем серьезности 0 до 10 являются информационными.
- Состояние* (state) — целое число с максимальным значением 127, используемое компанией Microsoft для внутренних целей.
- Сообщение об ошибке* (error message) может иметь длину до 255 символов в кодировке Unicode.
 - Сообщения об ошибках SQL Server перечислены в представлении каталога sys.messages.
 - Можно добавлять пользовательские сообщения об ошибках с помощью процедуры sp_addmessage.

К СВЕДЕНИЮ Уровни серьезности ошибок

Дополнительную информацию об уровне серьезности ошибок можно найти в электронной документации к SQL Server 2012 в разделе "Степени серьезности ошибок компонента Database Engine" по адресу <http://msdn.microsoft.com/ru-ru/library/ms164086.aspx>.

Команда RAISERROR

Команда RAISERROR использует следующий синтаксис:

```
RAISERROR ( { msg_id | msg_str | @local_variable }
{ ,severity ,state }
[ ,argument [ ,...n ] ] )
[ WITH option [ ,...n ] ]
```

Сообщение (идентификатор сообщения (ID), строка или строковая переменная), а также уровень серьезности и состояние должны присутствовать обязательно. Сообщение может быть простой строкой, как показано в следующем примере:

```
RAISERROR ('Error in usp_InsertCategories stored procedure', 16, 0);
```

Можно использовать строку с форматированием:

```
RAISERROR ('Error in %s stored procedure', 16, 0, N'usp_InsertCategories');
```

Также можно использовать переменную, как в следующем примере:

```
GO
DECLARE @message AS NVARCHAR(1000) = 'Error in %s stored procedure';
RAISERROR (@message, 16, 0, N'usp_InsertCategories');
```

Кроме того, можно задать форматирование вне команды RAISERROR с помощью функции FORMATMESSAGE.

```
GO
DECLARE @message AS NVARCHAR(1000) = 'Error in %s stored procedure';
SELECT @message = FORMATMESSAGE (@message, N'usp_InsertCategories');
RAISERROR (@message, 16, 0);
```

ПРИМЕЧАНИЕ Простая форма команды RAISERROR больше не поддерживается

Использование очень простой команды RAISERROR *int, 'string'* было разрешено в более ранних версиях SQL Server, но в версии SQL Server 2012 уже не поддерживается.

К некоторым дополнительным свойствам команды RAISERROR относятся следующие.

- Можно создавать исключительно информационные сообщения (такие как PRINT) с помощью уровня серьезности от 0 до 9.
- Можно задавать в команде RAISERROR уровень серьезности более 18, если используется параметр WITH LOG и если у вас есть роль sysadmin в SQL Server. При возникновении такой ошибки SQL Server прервет соединение.
- Можно использовать команду RAISERROR с параметром NOWAIT, чтобы отправлять сообщения клиенту немедленно. Сообщение не ждет в выходном буфере, прежде чем быть отправленным.

Команда *THROW*

Команда *THROW* во многом ведет себя так же, как команда *RAISERROR*, за некоторыми важными исключениями. Базовый синтаксис команды *THROW* выглядит следующим образом:

```
THROW [ { error_number | @local_variable },
{ message | @local_variable },
{ state | @local_variable }
] [; ]
```

Команда *THROW* имеет множество таких же компонентов, что и команда *RAISERROR*, но со следующими значительными отличиями:

- команда *THROW* не использует для разделения параметров круглые скобки;
- команда *THROW* может использоваться без параметров, но только в блоке *CATCH* конструкции *TRY/CATCH*;
- если имеются параметры, то номер ошибки, сообщение и состояние являются обязательными;
- номер ошибки *error_number* не требует соответствующего сообщения, определенного в представлении каталога *sys.messages*;
- параметр сообщения не допускает форматирование, но можно использовать функцию *FORMATMESSAGE()* с переменной для получения того же результата;
- параметр состояния (*state*) должен быть целым числом в диапазоне от 0 до 255;
- любой параметр может быть переменной;
- параметр серьезности ошибки не используется, уровень серьезности ошибки всегда устанавливается равным 16;
- команда *THROW* всегда прерывает выполнение пакета, за исключением случаев, когда она используется в блоке *TRY*.

СОВЕТ

Подготовка к экзамену

Инструкция, выполняющаяся перед инструкцией *THROW*, должна завершаться точкой с запятой (;). Считается наиболее правильным завершать все инструкции T-SQL точкой с запятой.

В качестве примера можно вызвать простую инструкцию *THROW* следующим образом:

```
THROW 50000, 'Error in usp_InsertCategories stored procedure', 0;
```

Поскольку инструкция *THROW* не позволяет форматировать параметр сообщения, можно использовать функцию *FORMATMESSAGE()*, как в следующем примере:

```
GO
DECLARE @message AS NVARCHAR(1000) = 'Error in %s stored procedure';
SELECT @message = FORMATMESSAGE(@message, N'usp_InsertCategories');
THROW 50000, @message, 0;
```

Существуют дополнительные важные различия между командами `THROW` и `RAISERROR`. Например, команда `RAISERROR`, как правило, не завершает работу пакета.

```
RAISERROR('Hi there', 16, 0);
PRINT 'RAISERROR error'; -- печать
GO
```

Тогда как команда `THROW` завершает пакет.

```
THROW 50000, 'Hi there', 0;
PRINT 'THROW error'; -- нет печати
GO
```

Существует еще несколько важных различий между командами `THROW` и `RAISERROR`:

- нельзя использовать команду `THROW` с опцией `NOWAIT`, чтобы немедленно вызвать вывод сообщения из буфера;
- нельзя использовать команду `THROW` с уровнем серьезности больше чем 16 с помощью предложения `WITH LOG`, как в случае с командой `RAISERROR`.

Функции `TRY_CONVERT` и `TRY_PARSE`

Можно использовать две функции для определения потенциальных ошибок, так чтобы ваш код мог избежать неожиданных ошибок. Функция `TRY_CONVERT` пытается привести значение к выходному типу данных и в случае успеха возвращает это значение, в противном случае возвращается значение `NULL`. В следующем примере проверяются два значения типа данных `datetime`, не принимающего даты ранее 1753-01-01, как допустимые.

```
SELECT TRY_CONVERT(DATETIME, '1752-12-31');
SELECT TRY_CONVERT(DATETIME, '1753-01-01');
```

Первая инструкция возвращает `NULL`, указывая на то, что преобразование не работает. Вторая инструкция возвращает конвертированное значение даты и времени в формате `datetime`. Функция `TRY_CONVERT` имеет формат, сходный с форматом существующей функции `CONVERT`, поэтому при преобразовании строк также можно передавать стиль.

Функция `TRY_PARSE` позволяет принимать входную строку, содержащую данные с неопределенным типом данных и преобразовывать их в конкретный тип данных при возможности, или возвращать `NULL` в противном случае. В следующем примере выполняет синтаксический анализ двух строк.

```
SELECT TRY_PARSE('1' AS INTEGER);
SELECT TRY_PARSE('B' AS INTEGER);
```

Первая строка конвертируется в целочисленный тип (`integer`), поэтому функция `TRY_PARSE` возвращает это значение как `integer`. Вторая строка, '`B`', не может быть конвертирована в тип данных `integer`, поэтому функция возвращает `NULL`. Функция `TRY_PARSE` придерживается синтаксиса функций `PARSE` и `CAST`.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как можно добавлять пользовательские сообщения об ошибках?
2. Для чего используется уровень серьезности, равный 0?

Ответы на контрольные вопросы

1. Чтобы добавить собственные пользовательские сообщения об ошибках, вы можете применять системную хранимую процедуру `sp_addmessage`.
2. При использовании команды `RAISERROR` с уровнем серьезности 0 отправляется только информационное сообщение. Если вы добавите параметр `WITH NOWAIT`, сообщение будет отправляться без ожидания во внешнем буфере.

Обработка ошибок после их обнаружения

По сути, существуют два метода обработки ошибок: неструктурированный и структурированный. При неструктурной обработке ошибок необходимо обрабатывать каждую ошибку при ее возникновении с помощью функции `@@ERROR`. При структурированной обработке ошибок можно обозначить централизованное местоположение (блок `CATCH`) для обработки ошибок.

Неструктурированная обработка ошибок с помощью функции `@@ERROR`

Неструктурированная обработка ошибок заключается в проверке отдельных инструкций на их состояние ошибки непосредственно после их выполнения. Для этого необходимо запросить системную функцию `@@ERROR`. Когда SQL Server выполняет какую-либо инструкцию T-SQL, он записывает состояние ошибки результата команды в `@@ERROR`. При возникновении ошибки вы можете запросить `@@ERROR`, чтобы получить номер ошибки. При успешном завершении инструкции значение `@@ERROR` будет равно 0, в противном случае `@@ERROR` будет содержать номер ошибки.

К сожалению, любой запрос функции `@@ERROR`, даже если это делается в предложении `IF`, приводит к ее сбросу и присвоению нового номера, потому что `@@ERROR` всегда выводит состояние ошибки последней выполненной команды. Поэтому проверить значение `@@ERROR` внутри кода управления ошибкой невозможно. Лучше добавить код, который сохраняет значение `@@ERROR` в переменной, и затем проверять эту переменную.

Основная проблема неструктурной обработки ошибок — необходимость проверять `@@ERROR` после каждой модификации данных или инструкции DML. Поскольку центральное местоположение для обработки ошибок не предоставляется, обработка ошибок должна выполняться в пользовательском коде.

Чтобы добавить некоторую структурированность в обработку ошибок, можно написать собственный код. Но пользовательский код для неструктурной обработки ошибок может быстро усложняться. Поэтому встроенное централизованное пространство для обработки ошибок действительно необходимо.

Использование параметра XACT_ABORT с транзакциями

Существует другая возможность для перехвата ошибок, на шаг приближающаяся к структурированной обработке ошибок: параметр SET XACT_ABORT (где XACT означает "transaction"). XACT_ABORT работает со всеми типами кода и воздействует на целый пакет. Поставив в начало пакета SET XACT_ABORT ON, можно вызвать сбой всего этого пакета в случае возникновения ошибки. XACT_ABORT устанавливается на отдельный сеанс. После его установки в ON, ему подчиняются все остальные транзакции в этом сеансе до того, как он будет установлен в OFF.

SET XACT_ABORT имеет некоторые преимущества. Он вызывает откат транзакции из-за любой ошибки с уровнем строгости больше 10. Однако XACT_ABORT присуще множество ограничений, к которым относятся следующие:

- нельзя перехватить ошибку или получить номер ошибки;
- любая ошибка с уровнем серьезности более 10 приводит к откату транзакции;
- никакой оставшийся код в транзакции не выполняется. Даже последняя инструкция PRINT в транзакции не будет выполнена;
- после прерывания транзакции можно понять, в какой инструкции произошел сбой, только на основании сообщения об ошибке, возвращенного клиенту от SQL Server.

Таким образом, XACT_ABORT не обеспечивает возможность обработки ошибок. Вам нужна конструкция TRY/CATCH.

Структурированная обработка ошибок с помощью конструкции TRY/CATCH

В SQL Server 2005 была добавлена конструкция TRY/CATCH, позволяющая выполнять структурированную обработку ошибок в SQL Server. Конструкции TRY/CATCH работает следующим образом.

- Вы помещаете код, который хотите проверять на наличие ошибок, в блок TRY.
 - После каждого блока TRY должен стоять блок CATCH, где выполняется обработка ошибок.
 - Блоки должны соединяться между собой и находиться в одном пакете T-SQL.
- Если условие ошибки определено в инструкции T-SQL внутри блока TRY, контроль передается соответствующему блоку CATCH для обработки ошибки.
 - Оставшиеся инструкции T-SQL в блоке TRY не выполняются.
 - После обработки ошибки в блоке CATCH управление передается первой инструкции T-SQL, стоящей после инструкции END CATCH.
 - Если в блоке TRY не обнаружена ни одна ошибка, управление передается в обход блока CATCH первой инструкции T-SQL, стоящей после END CATCH.
- Когда SQL Server обнаруживает ошибку в блоке TRY, никакое сообщение не отправляется клиенту.

- Это противоречит неструктурированной обработки ошибок, где сообщение об ошибке всегда отправляется клиенту и не может быть перехвачено.
- Даже команда `RAISERROR` в блоке `TRY` с уровнем серьезности от 11 до 19 не генерирует сообщение для клиента, а вместо этого передает управление блоку `CATCH`.

Используя блок `TRY/CATCH`, уже нет необходимости перехватывать отдельные инструкции на предмет обнаружения ошибок. Почти все ошибки вызывают попадание кода в блок `CATCH`.

Далее приведены некоторые правила использования конструкции `TRY/CATCH`.

- Ошибки с уровнем серьезности больше 10 и меньше 20 в блоке `TRY` приводят к передаче управления блоку `CATCH`.
- Ошибки с уровнем серьезности 20 и больше, которые не закрывают соединения, также обрабатываются блоком `CATCH`.
- Ошибки компиляции и некоторые ошибки выполнения программы, действующие компиляцию уровня инструкции, прерывают выполнение пакета немедленно и не передают управление `CATCH`.
- Если ошибка произошла в блоке `CATCH`, транзакция прерывается и ошибка возвращается вызывающему приложению, если блок `CATCH` не вложен в блок `TRY`.
- В блоке `CATCH` можно выполнить фиксацию или откат текущей транзакции, если транзакция не может быть зафиксирована и ее необходимо откатить. Для проверки состояния транзакции можно запросить функцию `XACT_STATE`.
- Блок `TRY/CATCH` не перехватывает ошибки, приводящие к прерыванию соединения, такие как неустранимая ошибка или выполнение ролью `sysadmin` команды `KILL`.
- Также невозможно перехватить ошибки, которые возникают из-за ошибок компиляции, синтаксических ошибок или несуществующих объектов. Поэтому вы не можете использовать конструкцию `TRY/CATCH` для проверки существования объекта.
- Блоки `TRY/CATCH` могут быть вложенными; другими словами, можно поместить внутренний блок `TRY/CATCH` во внешний блок `TRY`. Ошибка внутри вложенного блока `TRY` передает выполнение соответствующему вложенному блоку `CATCH`.

Для того чтобы создать сообщение об ошибке, можно использовать следующий набор функций внутри блока `CATCH`:

- `ERROR_NUMBER` — возвращает номер ошибки;
- `ERROR_MESSAGE` — возвращает сообщение об ошибке;
- `ERROR_SEVERITY` — возвращает уровень серьезности ошибки;
- `ERROR_LINE` — возвращает номер строки в пакете, где произошла ошибка;
- `ERROR_PROCEDURE` — имя функции, триггера или процедуры, которые выполнялись в момент возникновения ошибки;
- `ERROR_STATE` — состояние ошибки.

Можно инкапсулировать вызовы этих функций в хранимую процедуру, наряду с дополнительной информацией (как например имя базы данных и сервера и, возможно, время и дату), а затем вызывать хранимую процедуру из различных блоков CATCH.

```
BEGIN CATCH
    -- Обработка ошибок
    SELECT ERROR_NUMBER() AS errornumber
        , ERROR_MESSAGE() AS errormessage
        , ERROR_LINE() AS errorline
        , ERROR_SEVERITY() AS errorseverity
        , ERROR_STATE() AS errorstate;
END CATCH;
```

РЕАЛЬНЫЙ МИР

Предупреждение ошибок

При обработке ошибок в блоке CATCH номера ошибок, которые могут произойти, являются довольно большими, поэтому трудно их все предусмотреть. Также возможны специализированные типы транзакций или процедур. Некоторые разработчики T-SQL предпочитают просто возвращать значения функций ошибок, как в предыдущей инструкции SELECT. Это может быть наиболее полезным для утилитных хранимых процедур. В других случаях некоторые разработчики T-SQL используют хранимую процедуру, которая может быть вызвана из блока CATCH, и это обеспечит общую реакцию на определенные наиболее часто встречающиеся ошибки.

Выбор между *THROW* и *RAISERROR* в блоках *TRY/CATCH*

В блоке TRY можно использовать либо команду RAISERROR, либо команду THROW (с параметрами) для генерации условий ошибки и передачи управления блоку CATCH. Команда RAISERROR в блоке TRY должна иметь уровень серьезности от 11 до 19 для передачи управления блоку CATCH.

Используете ли вы в блоке TRY команду RAISERROR или THROW, SQL Server сообщение об ошибке клиенту.

В блоке CATCH у вас есть три возможности: RAISERROR, THROW с параметрами или THROW без параметров.

Можно использовать RAISERROR в блоке CATCH для предоставления исходной ошибки обратно клиенту или инициации дополнительной ошибки, которую вы хотите выводить. Исходный номер ошибки не может быть повторно иницирован. Это должен быть пользовательский номер сообщения об ошибке или, в данном случае, номер ошибки по умолчанию 50000. Для возвращения исходного номера ошибки можно добавить в параметр строку msg_str инструкции RAISERROR.

Выполнение блока CATCH продолжается после инструкции RAISERROR.

Можно использовать инструкцию THROW с параметрами, такими как RAISERROR, для повторной инициации ошибки в блоке CATCH. Но THROW с параметрами всегда создает ошибки с пользовательским номером ошибки и уровнем серьезности 16, поэтому вы не сможете получить точную информацию. Инструкция THROW с параметрами прерывает пакет, поэтому следующие за ней команды не будут выполнены.

Инструкция `THROW` без параметров может использоваться для повторной инициации исходного сообщения об ошибке и отправки его обратно клиенту. Безусловно, это наилучший способ предоставления ошибки обратно вызывающей стороне. Теперь исходное сообщение отправлено обратно клиенту и, под вашим контролем, немедленно прервет выполнение пакета.

СОВЕТ**Подготовка к экзамену**

Вы должны проследить, чтобы инструкция `THROW` с параметрами или без них была последней инструкцией, которая должна быть выполнена в блоке `CATCH`, поскольку она прерывает выполнение пакета, и оставшиеся команды в блоке `CATCH` не выполняются.

Использование параметра `XACT_ABORT` с блоками `TRY/CATCH`

Параметр `XACT_ABORT` ведет себя иначе, когда используется в блоке `TRY`. Вместо прерывания транзакции, как это происходит в случае неструктурированной обработки ошибок, `XACT_ABORT` передает управление блоку `CATCH`, и, как того следует ожидать, любая ошибка становится неустранимой. Транзакции остаются в нефиксированном состоянии (и `XACT_STATE()` возвращает `-1`). Поэтому невозможно зафиксировать транзакцию внутри блока `CATCH`, если опция `XACT_ABORT` включена; ее необходимо откатить.

Возможные значения `XACT_STATE()`:

- 1 — существует открытая транзакция, которую можно зафиксировать или откатить;
- 0 — открытых транзакций нет; это эквивалентно `@@TRANCOUNT = 0`;
- 1 — существует открытая транзакция, но она не находится в фиксируемом состоянии. Можно выполнить только откат транзакции.

В блоке `CATCH` можно определить уровень вложенности текущей транзакции с помощью системной функции `@@TRANCOUNT`.

Если у вас есть вложенные транзакции, можно извлечь состояние самой вложенной транзакции с помощью функции `XACT_STATE`.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем заключаются главные преимущества использования блока `TRY/CATCH` по сравнению с традиционным захватом ошибок функцией `@@ERROR`?
2. Может ли блок `TRY/CATCH` охватывать пакеты?

Ответы на контрольные вопросы

1. Главное преимущество заключается в том, что у вас есть единое пространство в вашем коде, в которое будут захватываться ошибки, так что вам нужно всего лишь поместить обработку ошибок в одно место.
2. Нет, необходимо иметь один набор блоков `TRY/CATCH` для каждого пакета кода.

ПРАКТИКУМ Использование обработки ошибок

В данном практикуме вы реализуете код, который выполняет запись в базу данных и проверку на наличие ошибок, и будете исследовать неструктурированную и структурированную обработку ошибок. Вы будете использовать среду SSMS и базу данных TSQL2012.

Задание 1. Работа с неструктурированной обработкой ошибок

В этом задании вы будете работать с неструктурированной обработкой ошибок, используя функцию `@@ERROR`.

1. Этот шаг использует функцию `@@ERROR`. В следующем коде вы будете проверять значение инструкции `@@ERROR` немедленно после того, как выполнилась инструкция модификации данных. Откройте SSMS, а затем пустое окно запроса. Выполните весь пакет кода T-SQL. Обратите внимание на сообщение об ошибке, которое SQL Server отправляет обратно клиентскому приложению, SQL Server Management Studio.

```
USE TSQL2012;
GO
DECLARE @errnum AS int;
BEGIN TRAN;
    SET IDENTITY_INSERT Production.Products ON;
    INSERT INTO Production.Products(productid, productname, supplierid,
                                    categoryid, unitprice, discontinued)
        VALUES(1, N'Test1: Duplicate productid', 1, 1, 18.00, 0);
    SET @errnum = @@ERROR;
    IF @errnum <> 0 -- Обработка ошибки
    BEGIN
        PRINT 'Insert into Production.Products failed with error ' +
              CAST(@errnum AS VARCHAR);
    END;
    GO
    IF @@TRANCOUNT <> 0 ROLLBACK TRANSACTION
```

2. На этом шаге вы будете работать с неструктурированной обработкой ошибок в транзакции. В следующем коде есть две инструкции `INSERT` в пакете, которые вы поместите в транзакцию для того, чтобы выполнить откат транзакции в случае сбоя любой из инструкций. Первая инструкция `INSERT` завершается сбоем, но вторая выполняется успешно, поскольку SQL Server по умолчанию не выполняет откат транзакции с ошибкой повторяющегося первичного ключа. Заметьте, при выполнении кода первая инструкция `INSERT` завершается ошибкой из-за нарушения ограничения первичного ключа, и транзакция откачена. Но вторая инструкция `INSERT` заканчивается успешно, потому что неструктурированная обработка ошибок не передает управление программой таким образом, чтобы не выполнять вторую инструкцию `INSERT`. Для получения лучшего контроля вы должны добавить много команд для решения проблемы. Откройте SSMS, а в ней пустое окно запроса. Выполните весь пакет кода T-SQL.

```

USE TSQL2012;
GO
DECLARE @errnum AS int;
BEGIN TRAN;
    SET IDENTITY_INSERT Production.Products ON;
    -- Вставка #1 не выполнится из-за повторяющегося первичного ключа
    INSERT INTO Production.Products(productid, productname, supplierid,
                                    categoryid, unitprice, discontinued)
        VALUES(1, N'Test1: Duplicate productid', 1, 1, 18.00, 0);
    SET @errnum = @@ERROR;
    IF @errnum <> 0
        BEGIN
            IF @@TRANCOUNT > 0 ROLLBACK TRAN;
            PRINT 'Insert #1 into Production.Products failed with error ' +
                  CAST(@errnum AS VARCHAR);
        END;
    -- Вставка #2 будет успешной
    INSERT INTO Production.Products(productid, productname, supplierid,
                                    categoryid, unitprice, discontinued)
        VALUES(101, N'Test2: New productid', 1, 1, 18.00, 0);
    SET @errnum = @@ERROR;
    IF @errnum <> 0
        BEGIN
            IF @@TRANCOUNT > 0 ROLLBACK TRAN;
            PRINT 'Insert #2 into Production.Products failed with error ' +
                  CAST(@errnum AS VARCHAR);
        END;
    SET IDENTITY_INSERT Production.Products OFF;
    IF @@TRANCOUNT > 0 COMMIT TRAN;
    -- Удаление вставленной строки
    DELETE FROM Production.Products WHERE productid = 101;
    PRINT 'Deleted ' + CAST(@@ROWCOUNT AS VARCHAR) + ' rows';

```

Задание 2. Использование параметра XACT_ABORT для обработки ошибок

В этом задании вы будете работать с параметром XACT_ABORT, как методом обработки ошибок.

1. На данном шаге вы будете использовать XACT_ABORT и получите ошибку. В следующем коде проверите, что XACT_ABORT прервет пакет, если SQL Server обнаружит ошибку в инструкции модификации данных. Откройте SSMS, а в ней пустое окно запроса. Выполните оба пакета кода T-SQL. Обратите внимание на сообщение об ошибке, которое SQL Server отправляет обратно клиентскому приложению, SQL Server Management Studio.

```

USE TSQL2012;
GO
SET XACT_ABORT ON;

```

```

PRINT 'Before error';
SET IDENTITY_INSERT Production.Products ON;
INSERT INTO Production.Products(productid, productname, supplierid,
categoryid, unitprice, discontinued)
    VALUES(1, N'Test1: Duplicate productid', 1, 1, 18.00, 0);
SET IDENTITY_INSERT Production.Products OFF;
PRINT 'After error';
GO
PRINT 'New batch';
SET XACT_ABORT OFF;

```

2. На данном шаге используется `THROW` с `XACT_ABORT`. В следующем коде вы должны проверить, что `XACT_ABORT` прервёт пакет, если встретится ошибка. Откройте SSMS, а в ней пустое окно запроса. Обратите внимание, выполнение `THROW` с `XACT_ABORT ON` вызывает прерывание пакета. Выполните оба пакета кода T-SQL.

```

USE TSQL2012;
GO
SET XACT_ABORT ON;
PRINT 'Before error';
THROW 50000, 'Error in usp_InsertCategories stored procedure', 0;
PRINT 'After error';
GO
PRINT 'New batch';
SET XACT_ABORT OFF;

```

3. На данном шаге параметр `XACT_ABORT` используется в транзакции. Выполните следующий код как единый пакет. Обратите внимание, `XACT_ABORT` в транзакции не позволяет второй инструкции `INSERT` быть выполненной и во втором пакете ни одной строки не удалено. Заметьте, что предложение `IF @errnum` не будет выполнено из-за параметров `XACT_ABORT`. Откройте SSMS, а в ней пустое окно запроса. Выполните последовательно каждый пакет кода T-SQL.

```

USE TSQL2012;
GO
DECLARE @errnum AS int;
SET XACT_ABORT ON;
BEGIN TRAN;
    SET IDENTITY_INSERT Production.Products ON;
    -- Вставка #1 не выполнится из-за повторяющегося первичного ключа
    INSERT INTO Production.Products(productid, productname, supplierid,
categoryid, unitprice, discontinued)
        VALUES(1, N'Test1: Duplicate productid', 1, 1, 18.00, 0);
    SET @errnum = @@ERROR;
    IF @errnum <> 0
        BEGIN
            IF @@TRANCOUNT > 0 ROLLBACK TRAN;
            PRINT 'Error in first INSERT';
        END;

```

```
-- Вставка #2 больше не выполняется
INSERT INTO Production.Products (productid, productname, supplierid,
                                 categoryid, unitprice, discontinued)
VALUES(101, N'Test2: New productid', 1, 1, 18.00, 0);
SET @errnum = @@ERROR;
IF @errnum <> 0
BEGIN
    -- Принятие мер с учетом ошибок
    IF @@TRANCOUNT > 0 ROLLBACK TRAN;
    PRINT 'Error in second INSERT';
END;
SET IDENTITY_INSERT Production.Products OFF;
IF @@TRANCOUNT > 0 COMMIT TRAN;
GO

DELETE FROM Production.Products WHERE productid = 101;
PRINT 'Deleted ' + CAST(@@ROWCOUNT AS VARCHAR) + ' rows';
SET XACT_ABORT OFF;
GO
SELECT XACT_STATE(), @@TRANCOUNT;
```

Задание 3. Структурированная обработка ошибок с помощью блоков TRY/CATCH

- На этом шаге начните с конструкции TRY/CATCH. В следующем коде имеются две инструкции INSERT в одном пакете, помещенном в транзакцию. Первая инструкция INSERT завершается сбоем, но вторая выполняется успешно, поскольку SQL Server по умолчанию не прервет выполнение транзакции с ошибкой повторяющегося первичного ключа. Обратите внимание, когда выполняется код, первая инструкция INSERT завершается ошибкой из-за нарушения ограничений повторяющегося ключа, и транзакция откатывается. Но клиенту не отправляются никакие ошибки, и выполнение переносится в блок CATCH. Ошибка обрабатывается, и транзакция откатывается. Откройте SSMS, а в ней пустое окно запроса. Выполните весь пакет кода T-SQL.

```
USE TSQL2012;
GO
BEGIN TRY
BEGIN TRAN;
    SET IDENTITY_INSERT Production.Products ON;
    INSERT INTO Production.Products (productid, productname, supplierid,
                                     categoryid, unitprice, discontinued)
    VALUES(1, N'Test1: Duplicate productid', 1, 1, 18.00, 0);
    INSERT INTO Production.Products (productid, productname, supplierid,
                                     categoryid, unitprice, discontinued)
    VALUES(101, N'Test2: New productid', 1, 10, 18.00, 0);
    SET IDENTITY_INSERT Production.Products OFF;
```

```

COMMIT TRAN;
END TRY
BEGIN CATCH
    IF ERROR_NUMBER() = 2627 -- Duplicate key violation
        BEGIN
            PRINT 'Primary Key violation';
        END
    ELSE IF ERROR_NUMBER() = 547 -- Constraint violations
        BEGIN
            PRINT 'Constraint violation';
        END
    ELSE
        BEGIN
            PRINT 'Unhandled error';
        END;
    IF @@TRANCOUNT > 0 ROLLBACK TRANSACTION;
END CATCH;

```

2. Перепишите блок CATCH с помощью переменных для сбора информации об ошибке и повторно инициируйте ошибку с помощью RAISERROR.

```

USE TSQL2012;
GO
SET NOCOUNT ON;
DECLARE @error_number AS INT, @error_message AS NVARCHAR(1000),
@error_severity AS INT;
BEGIN TRY
BEGIN TRAN;
    SET IDENTITY_INSERT Production.Products ON;
    INSERT INTO Production.Products(productid, productname, supplierid,
    categoryid, unitprice, discontinued)
        VALUES(1, N'Test1: Duplicate productid', 1, 1, 18.00, 0);
    INSERT INTO Production.Products(productid, productname, supplierid,
    categoryid, unitprice, discontinued)
        VALUES(101, N'Test2: New productid', 1, 10, 18.00, 0);
    SET IDENTITY_INSERT Production.Products OFF;
COMMIT TRAN;
END TRY
BEGIN CATCH
    SELECT XACT_STATE() as 'XACT_STATE', @@TRANCOUNT as '@@TRANCOUNT';
    SELECT @error_number = ERROR_NUMBER(),
        @error_message = ERROR_MESSAGE(),
        @error_severity = ERROR_SEVERITY();
    RAISERROR (@error_message, @error_severity, 1);
    IF @@TRANCOUNT > 0 ROLLBACK TRANSACTION;
END CATCH;

```

3. Затем используйте инструкцию THROW без параметров для повторной инициации (повторной выдачи) исходного сообщения об ошибке и отправки его обратно

клиенту. Безусловно, это наилучший способ предоставления ошибки обратно клиенту.

```
USE TSQL2012;
GO
BEGIN TRY
BEGIN TRAN;
    SET IDENTITY_INSERT Production.Products ON;
    INSERT INTO Production.Products(productid, productname, supplierid,
        categoryid, unitprice, discontinued)
        VALUES(1, N'Test1: Duplicate productid', 1, 1, 18.00, 0);
    INSERT INTO Production.Products(productid, productname,
        categoryid, unitprice, discontinued)
        VALUES(101, N'Test2: New productid', 1, 10, 18.00,
    SET IDENTITY_INSERT Production.Products OFF;
COMMIT TRAN;
END TRY
BEGIN CATCH
    SELECT XACT_STATE() as 'XACT_STATE', @@TRANCOUNT as '@@TRANCOUNT';
    IF @@TRANCOUNT > 0 ROLLBACK TRANSACTION;
    THROW;
END CATCH;
GO
SELECT XACT_STATE() as 'XACT_STATE', @@TRANCOUNT as '@@TRANCOUNT';
```

Резюме занятия

- SQL Server 2012 использует команды RAISERROR и THROW для генерации ошибок.
- Можно запросить системную функцию @@ERROR для определения, произошла ли ошибка и какой у нее номер.
- Можно использовать команду SET XACT_ABORT ON, чтобы вызвать сбой транзакции и прерывание пакета, когда возникнет ошибка.
- Неструктурированная обработка ошибок не предоставляет единого места в коде для обработки ошибок.
- Блок TRY/CATCH предоставляет каждому блоку кода T-SQL блок CATCH, в котором выполняется обработка ошибок.
- Команда THROW может быть использована для повторной выдачи ошибок.
- Существует полный набор функций обработки ошибок для сбора информации об ошибках.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в приложении "Ответы" в конце книги.

1. В чем заключается преимущество использования команды `THROW` в блоке `CATCH`?
 - A. Команда `THROW` в блоке `CATCH` не требует параметров и поэтому более проста в написании.
 - B. Команда `THROW` повторно вызывает исходную ошибку, так что эта ошибка может быть обработана.
 - C. Команда `THROW` автоматически использует уровень серьезности 16.
 - D. Инструкция перед командой `THROW` требует точки с запятой.
2. Какие из перечисленных функций могут использоваться в блоке `CATCH` для возвращения информации об ошибке? (Выберите все подходящие варианты.)
 - A. `@@ERROR`.
 - B. `ERROR_NUMBER()`.
 - C. `ERROR_MESSAGE()`.
 - D. `XACT_STATE()`.
3. Как инструкция `SET XACT_ABORT ON` влияет на транзакцию?
 - A. Если возникает ошибка T-SQL с уровнем серьезности более 16, транзакция будет прервана.
 - B. Если возникает ошибка T-SQL с уровнем серьезности более 10, транзакция будет прервана.
 - C. Если возникает ошибка T-SQL с уровнем серьезности более 16, некоторые инструкции транзакции тем не менее могут быть выполнены.
 - D. Если возникает ошибка T-SQL с уровнем серьезности более 10, некоторые инструкции транзакции тем не менее могут быть выполнены.

Занятие 3. Использование динамического SQL

Динамическим *SQL* называется методика использования кода T-SQL для генерации и потенциального выполнения другого кода T-SQL. Используя динамический SQL, разработчик пишет код T-SQL, который будет динамически создавать другой код T-SQL, и затем, как правило, пакет за пакетом отправлять этот код на SQL Server для выполнения. В итоге вы используете T-SQL, чтобы генерировать и выполнять код T-SQL.



Изучив материал этого занятия, вы сможете:

- ✓ Объяснить использование T-SQL для генерации T-SQL
- ✓ Применять команду `EXECUTE` для выполнения динамического SQL
- ✓ Проследить за внедрением SQL-кода, добавляющего нежелательные команды в динамический SQL
- ✓ Использовать хранимую процедуру `sp_executesql` для параметризации динамического SQL и снизить риск внедрения SQL-кода

Продолжительность занятия — 40 минут.

Обзор динамического SQL

Часто в коде T-SQL встречаются задачи, которые требуют динамического представления значений во время выполнения кода, а не заранее, так что значения приходится предоставлять в переменных. Но во многих случаях нельзя заменить переменными литералы в коде T-SQL.

Например, представьте, что вам нужно посчитать строки в таблице Production.Products базы данных TS2012.

```
USE TSQ2012;
GO
SELECT COUNT(*) AS ProductRowCount FROM [Production].[Products];
```

Теперь представьте, что вы хотите заменить переменной имя таблицы и схемы, чтобы затем можно было применять эту инструкцию к любому количеству таблиц. Простая замена переменной не даст положительного результата.

```
USE TSQ2012;
GO
DECLARE @tablename AS NVARCHAR(261) = N'[Production].[Products]';
SELECT COUNT(*) FROM @tablename;
```

ПРИМЕЧАНИЕ Планирование длины идентификатора SQL

Этот код использует тип данных NVARCHAR(261), чтобы строковая переменная была достаточно длинной для обработки двух идентификаторов SQL Server (128 символов) плюс точка-разделитель и скобки.

Но выполнив объединение этой переменной со строковым литералом, вы сможете вывести команду на печать.

```
USE TSQ2012;
GO
DECLARE @tablename AS NVARCHAR(261) = N'[Production].[Products]';
PRINT N'SELECT COUNT(*) FROM ' + @tablename;
```

Или можно использовать инструкцию SELECT для получения того же результата, но в итоговом наборе.

```
DECLARE @tablename AS NVARCHAR(261) = N'[Production].[Products]';
SELECT N'SELECT COUNT(*) FROM ' + @tablename;
```

В каждом случае результатом является правильный код T-SQL.

```
SELECT COUNT(*) AS ProductRowCount FROM [Production].[Products];
```

Можно скопировать код в окно запроса и выполнить его вручную либо внедрить его и выполнить немедленно, используя команду EXECUTE, или хранимую процедуру sp_executesql для прямого выполнения.

```
DECLARE @tablename AS NVARCHAR(261) = N'[Production].[Products]';
EXECUTE (N'SELECT COUNT(*) AS TableRowCount FROM ' + @tablename);
```

Динамический SQL заключает в себе и генерацию нового кода T-SQL, и немедленное выполнение сгенерированного кода.

Использование динамического SQL

Динамический SQL является полезной возможностью, т. к. T-SQL не разрешит прямую замену многих частей команд переменными, включая следующие:

- имя базы данных в инструкции `USE`;
- имена таблиц в предложении `FROM`;
- имена столбцов в предложениях `SELECT`, `WHERE`, `GROUP BY` и `HAVING`, а также в предложении `ORDER BY`;
- содержимое списков, например, в предложениях `IN` и `PIVOT`.

Если вам необходимо использовать для этих частей команд переменные, вы должны будете использовать динамический SQL.

К общим сценариям использования динамического SQL относятся следующие.

- Генерация кода для автоматизации административных задач.
- Выполнение итераций во всех базах данных на сервере, во всех типах объектов в базе данных и в метаданных объектов, таких как имена столбцов или индексы.
- Построение хранимых процедур с множеством необязательных параметров, составляющих результирующие запросы, на основании которых параметры получают значения. Например, процедура поиска может принимать имя, адрес, город и штат, но пользователь передает только значение имени. Тогда процедура должна создать команду `SELECT`, которая выполняет фильтрацию только по имени и ни по каким другим параметрам.
- Конструирование параметризованных оперативных запросов, которые могут повторно использовать предварительно помещенные в кэш планы выполнения (см. разд. "Использование хранимой процедуры `sp_executesql`" далее в этом занятии).
- Конструирование команд, требующих элементы кода на основе запроса базовых данных; например, динамическое построение запроса `PIVOT`, когда вам заранее не известно, какие лiteralные значения должны появиться в предложении `IN` оператора `PIVOT`.

Генерация строк T-SQL

При генерации инструкций T-SQL вы работаете со строками и должны обращать особое внимание на способ определения этих строк. По умолчанию в SQL Server 2012 вы должны использовать одиночную кавычку (т. е. апостроф) для разделения строк. Причина этого — в параметре `QUOTED_IDENTIFIER`.

Когда параметр `QUOTED_IDENTIFIER` установлен в `ON`, что является настройкой по умолчанию, необходимо разделять строковые литералы с помощью одиночных кавычек, а двойные кавычки использовать только для определения идентификаторов T-SQL (в дополнение к квадратным скобкам).

Если параметр `QUOTED_IDENTIFIER` установлен в `OFF`, то, наряду с одиночными кавычками, для разделения строк можно также использовать двойные кавычки. Но

тогда вы обязаны использовать квадратные скобки для разделения идентификаторов T-SQL.

СОВЕТ**Подготовка к экзамену**

Следует оставить параметр QUOTED_IDENTIFIER установленным в ON, поскольку это соответствует стандарту ANSI и настройке SQL Server по умолчанию.

Но использование в качестве разделителей строк только одиночных кавычек приводит к возникновению проблем: что вы будете делать с внедренными одиночными кавычками? Например, как выполнить поиск адреса "5678 rue de l'Abbaye" в таблице Sales.Customers базы данных TSQL2012? Можно попробовать следующий вариант:

```
USE TSQL2012;
GO
SELECT custid, companyname, contactname, contacttitle, address FROM
[Sales].[Customers] WHERE address = N'5678 rue de l'Abbaye';
```

Обратите внимание на сообщение об ошибке.

```
Msg 156, Level 15, State 1, Line 1
Incorrect syntax near the keyword 'AS'.
Msg 105, Level 15, State 1, Line 3
Unclosed quotation mark after the character string '
'.
```

SQL Server интерпретировал строку поиска как 5678 rue de 1, поскольку он воспринимает вторую одиночную кавычку как разделитель строки. Оставшаяся часть строки Abbaye' обрабатывается как синтаксическая ошибка. Для обработки внедренных одиночных кавычек необходимо заменить нужную одиночную кавычку на две одиночных встроенных кавычки, чтобы на выходе получить одну одиночную кавычку.

```
SELECT custid, companyname, contactname, contacttitle, address FROM
[Sales].[Customers] WHERE address = N'5678 rue de l'''Abbaye';
```

Таким образом, мы получим желаемый результат. Когда Server видит две одиночные кавычки внутри строки и оценивает их, он транслирует эти две одиночные кавычки в одну одиночную кавычку.

К сожалению, внедрение двух одиночных кавычек для получения на выходе каждой одиночной кавычки затрудняет чтение и понимание динамического SQL при работе со строками, содержащими разделители. Например, для печати предыдущей команды с помощью инструкции PRINT вам понадобилось бы использовать код, подобный следующему:

```
PRINT N'SELECT custid, companyname, contactname, contacttitle, address
FROM [Sales].[Customers]
WHERE address = N'''5678 rue de l'''''Abbaye'''';'
```

В качестве альтернативы можно использовать функцию QUOTENAME, которая может скрывать сложность внедренных кавычек. Можно использовать QUOTENAME для

автоматического удваивания количества внедренных кавычек. Например, следующее выражение:

```
PRINT QUOTENAME(N'5678 rue de l''Abbaye', ''');
```

дает результат:

```
'5678 rue de l''Abbaye'
```

ПРИМЕЧАНИЕ Ограничение выходных данных Management Studio

SQL Server может генерировать очень большие строки динамического SQL, но утилита Server Management Studio не показывает более 8000 байт при выводе в окно **Text**, независимо от того, генерируются выходные данные с помощью инструкции **PRINT** или инструкции **SELECT**. Чтобы выводить более 8000 байт, необходимо разбить длинную строку на подстроки с размером, меньше или равным 8000 байт, и генерировать их по отдельности.

После того как будут сгенерированы строки, которые необходимо выполнить, можно либо сгенерировать результаты и выполнить их самостоятельно, либо отправить строку непосредственно на SQL Server. Это называется выполнением динамического SQL. Для этого существуют два способа: инструкция **EXECUTE** и хранимая процедура **sp_executesql**.

Инструкция *EXECUTE*

Простейший способ, предоставляемый SQL Server для выполнения динамического SQL, — это инструкция **EXECUTE**, которую можно записывать как **EXECUTE** или сокращенно **EXEC**. В данном занятии мы будем далее использовать краткую форму **EXEC**.

Инструкция **EXEC** используется в разных случаях, и только один из них работает с динамическим SQL:

- выполнение хранимых процедур;
- олицетворение пользователей или имени входа в систему;
- запрос связанного сервера;
- выполнение строк, сгенерированных динамическим SQL.

Данное занятие посвящено последнему пункту — исполнению строк динамического SQL.

Инструкция **EXEC** для динамического SQL принимает символьную строку в круглых скобках в качестве входных данных. Далее перечислены особенности использования команды **EXEC**, которые следует знать.

- Входная строка должна быть одним пакетом T-SQL. Стока может содержать множество команд T-SQL, но она не может содержать разделители **GO**.
- Можно использовать строковые литералы, строковые переменные или их объединение.
 - Строковые переменные могут иметь любой строковый тип данных, обычные символы или символы в формате Unicode.
 - Строковые переменные могут иметь определения длины (**MAX**).

Следующий пример команды EXEC иллюстрирует потенциальное значение использования динамического SQL. Приведенный далее код возвращает нас к исходному примеру в начале занятия, но на этот раз мы расширим его использованием строковых переменных.

```
USE TSQL2012;
GO
DECLARE @SQLString AS NVARCHAR(4000)
    , @tablename AS NVARCHAR(261) = '[Production].[Products]';
SET @SQLString = 'SELECT COUNT(*) AS TableRowCount FROM ' + @tablename;
EXEC(@SQLString);
```

Хотя команда EXEC принимает на вход строковую переменную, она также принимает объединение двух или более строковых переменных.

```
USE TSQL2012;
GO
DECLARE @SQLString AS NVARCHAR(MAX)
    , @tablename AS NVARCHAR(261) = '[Production].[Products]';
SET @SQLString = 'SELECT COUNT(*) AS TableRowCount FROM '
EXEC(@SQLString + @tablename);
```

КОНТРОЛЬНЫЕ ВОПРОСЫ

- Можно ли сгенерировать и выполнить динамический SQL в другой базе данных по отношению к той, в которой находится код?
- На какие объекты нельзя ссылаться в T-SQL с помощью переменных?

Ответы на контрольные вопросы

- Да, поскольку команда USE <database> может быть вставлена в динамический пакет SQL.
- К объектам, для которых нельзя использовать переменные в командах T-SQL, относятся имя базы данных в инструкции USE, имя таблицы в предложении FROM, имена столбцов в предложениях SELECT и WHERE и списки литеральных значений в функциях IN() и PIVOT().

Внедрение кода SQL

Использование динамического SQL в приложениях, которые отправляют пользовательские входные данные базе данных, может послужить поводом для внедрения SQL-кода, где пользователь вводит что-то, что не предназначено для выполнения. Внедрение SQL-кода — это большая тема; существует множество видов внедрения SQL-кода, которое может произойти и на уровне клиента, и на уровне сервера. Задача разработчика T-SQL — защитить код от любого внедрения SQL-кода.

Злоумышленники поняли, что просто вставив одиночную кавычку, иногда можно заставить приложения отправлять пользователю сообщения об ошибке, указывающие на то, что команда была собрана с помощью динамического SQL и может быть повреждена. Все, что делает хакер — вводит одиночную кавычку (').

В результате SQL Server выдает следующее сообщение:

```
Msg 105, Level 15, State 1, Line 1
Unclosed quotation mark after the character string '''.
Msg 102, Level 15, State 1, Line 1
Incorrect syntax near '''.
```

Контрольное сообщение — "Unclosed quotation mark after the character string '''." ("Открытая кавычка после символьной строки"). Это сигнал хакеру, что он может добавить разделитель и дополнительный код, если строка примет более длинные входные данные. Теперь хакер понимает, что вставка одиночной кавычки может прервать строку раньше, поскольку "повисший" разделитель строки — скрытая одиночная кавычка — был обнаружен SQL-сервером. Единственное, что должен ввести хакер, — это односторонний комментарий после одиночной кавычки, чтобы заставить SQL Server игнорировать запаздывающую одиночную кавычку. Хакер вводит ' --.

Если это приводит к удачному удалению сообщения об ошибке, хакер знает, что другая команда T-SQL может быть внедрена в строку, как в следующем примере:

```
' SELECT TOP 1 name FROM sys.tables --
```

Разумеется, входная строка должна вмещать достаточное число символов, чтобы разрешить внедренную команду, но с этим хакер в конце концов справится.

Более опасными, чем инструкция `SELECT`, являются инструкции `DELETE` или `DROP`, которые могут быть использованы для повреждения данных.

Существует множество методов предупреждения внедрения SQL-кода, и это тоже обширная тема. С точки зрения разработчика T-SQL, одним из наиболее предпочтительных способов является параметризация генерации и исполнения динамического SQL с использованием хранимой процедуры `sp_executesql`.

К СВЕДЕНИЮ Внедрение кода SQL

Дополнительную информацию о внедрении SQL-кода можно найти в электронной документации к SQL Server 2012 в разделе "Атака SQL Injection" по адресу

[http://msdn.microsoft.com/ru-ru/library/ms161953\(SQL.105\).aspx](http://msdn.microsoft.com/ru-ru/library/ms161953(SQL.105).aspx).

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как может определить хакер, что существует возможность внедрения SQL-кода?
2. Где вставляется внедренный код?

Ответы на контрольные вопросы

1. Вставив одиночную кавычку и проверив сообщения об ошибке.
2. Между начальной одиночной кавычкой, которая прерывает строку входных данных, и конечным знаком комментария, который запрещает внутреннюю прерывающую одиночную кавычку.

Использование хранимой процедуры sp_executesql

Системная хранимая процедура sp_executesql появилась как альтернатива использованию команды EXEC для исполнения динамического SQL. Она генерирует, и исполняет строку динамического SQL.

Системная хранимая процедура sp_executesql поддерживает параметры, которые должны передаваться в формате Unicode.

Поддерживаются выходные параметры. Благодаря параметрам, системная хранимая процедура sp_executesql является более безопасной и может помочь предотвратить некоторые виды внедрения SQL-кода. Параметры sp_executesql не могут быть использованы для замещения необходимых строковых литералов, таких как имена таблиц и столбцов.

Далее приведен синтаксис хранимой процедуры sp_executesql.

```
sp_executesql [ @statement = ] statement  
[ {, [ @params = ] N'@parameter_name data_type [ OUT | OUTPUT ][,...n ]' }  
{, [ @param1 = ] 'value1' [ ,...n ] }]
```

Входной параметр `@statement` имеет тип данных NVARCHAR(MAX). Вам нужно представить инструкцию в виде Unicode-строки в параметр `@statement` и встроить в эту инструкцию параметры, которые должны быть заменены в финальной строке. Вы должны перечислить имена этих параметров вместе с их типами данных в строке `@params`, а затем поместить значения в списки `@param1`, `@param2` и т. д.

Таким образом, можно переписать команду, которая подсчитывает строки в динамически генерируемой таблице, следующим образом, используя параметр `@tablename`.

```
USE TSQL2012;  
GO  
DECLARE @SQLString AS NVARCHAR(4000), @address AS NVARCHAR(60);  
SET @SQLString = N'  
SELECT custid, companyname, contactname, contacttitle, address  
FROM [Sales].[Customers]  
WHERE address = @address';  
SET @address = N'5678 rue de l''Abbaye';  
EXEC sp_executesql  
    @statement = @SQLString,  
    @params = N'@address NVARCHAR(60)',  
    @address = @address;
```

COBET

Подготовка к экзамену

Способность выполнять параметризацию означает, что процедура sp_executesql избегает простых объединений, подобных используемым инструкцией EXEC. В результате она может быть применена для предотвращения внедрения SQL-кода.

Хранимая процедура sp_executesql иногда обеспечивает лучшую производительность запроса, чем команда EXEC, т. к. параметризация способствует повторенному

использованию кэшированных планов выполнения. Поскольку хранимая процедура `sp_executesql` принуждает к параметризации, часто строка запроса остается той же самой, а изменяются только значения параметров. Тогда SQL Server может сохранять всю строку постоянной и повторно использовать план запроса, созданный для исходного вызова этой отдельной строки к процедуре `sp_executesql`. Повторное использование плана не гарантируется, поскольку существуют другие факторы, которые SQL Server принимает во внимание, но шансы на повторное использование увеличиваются.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как можно передать информацию из хранимой процедуры `sp_executesql` вызывающей стороне?
2. Как хранимая процедура `sp_executesql` может остановить внедрение SQL-кода?

Ответы на контрольные вопросы

1. Необходимо использовать один или более параметров `OUTPUT`. Также можно сохранить данные во временной или постоянной таблице, но использование параметра `OUTPUT` является наиболее очевидным способом.
2. Можно использовать хранимую процедуру `sp_executesql` для параметризации входных данных пользователя, что может предотвратить выполнение любого внедренного кода.

ПРАКТИКУМ Написание и тестирование динамического SQL

В данном практикуме вам предстоит написать и протестировать динамический SQL-код. Вы будете использовать среду SSMS и базу данных TSQL2012.

Задание 1. Генерация строки T-SQL и использование функции QUOTENAME

В этом задании вы будете использовать функцию `QUOTENAME` для упрощения процесса генерации строк T-SQL. Преимущества функции `QUOTENAME` проще всего показать на примере использования переменной.

1. На этом шаге вы будете использовать переменную для генерации строк T-SQL. Откройте SSMS, а в ней пустое окно запроса. Выполните следующий пакет кода T-SQL. Обратите внимание, что результирующая строка, которая выводится на печать, не имеет правильных разделителей.

```
USE TSQL2012;
GO
DECLARE @address AS NVARCHAR(60) = '5678 rue de l''Abbaye';
PRINT N'SELECT *
FROM [Sales].[Customers]
WHERE address = '+ @address;
```

Теперь внедрите переменную с QUOTENAME до объединения ее с инструкцией PRINT. Обратите внимание, что сейчас результирующая строка получена успешно.

```
USE TSQL2012;
GO
DECLARE @address AS NVARCHAR(60) = '5678 rue de l''Abbaye';
PRINT N'SELECT *
FROM [Sales].[Customers]
WHERE address = '+ QUOTENAME(@address, '') + ';;'
```

Задание 2. Предупреждение внедрения SQL-кода

В этом задании вам нужно имитировать внедрение SQL-кода с помощью T-SQL и потренироваться в его предотвращении с помощью хранимой процедуры sp_executesql. Вам нужно передать параметр хранимой процедуре, чтобы сымитировать передачу хакером данных с помощью ввода с экрана.

1. Откройте SSMS и загрузите следующий сценарий хранимой процедуры в окно запроса. Процедура использует динамический SQL для возвращения списка клиентов с учетом их адресов. В задании используется адрес, поскольку это более длинная символьная строка, которая может разрешить добавление к ней дополнительных команд SQL.

```
USE TSQL2012;
GO
IF OBJECT_ID('Sales.ListCustomersByAddress') IS NOT NULL
    DROP PROCEDURE Sales.ListCustomersByAddress;
GO
CREATE PROCEDURE Sales.ListCustomersByAddress
    @address NVARCHAR(60)
AS
    DECLARE @SQLString AS NVARCHAR(4000);
    SET @SQLString = N'
SELECT companyname, contactname
FROM Sales.Customers WHERE address = ''' + @address + ''';
-- PRINT @SQLString;
EXEC(@SQLString);
RETURN;
GO
```

2. Хранимая процедура работает, как и предполагается, при обычном входном параметре @address. Выполните следующий код в отдельном окне запроса:

```
USE TSQL2012;
GO
EXEC Sales.ListCustomersByAddress @address = N'8901 Tsawassen Blvd.';
```

3. Чтобы сымитировать отправку хакером одиночной кавычки, вызовите хранимую процедуру с двумя одиночными кавычками в качестве разделителей строки. Запомните сообщение об ошибке от SQL Server.

```
USE TSQL2012;
GO
EXEC Sales.ListCustomersByAddress @address = N'''';
```

4. Теперь вставьте знак комментария после одиночной кавычки, чтобы конечный разделитель строки игнорировался.

```
USE TSQL2012;
GO
EXEC Sales.ListCustomersByAddress @address = N''' -- ';
```

5. Теперь остается только внедрить вредоносный код. Пользователь вводит `SELECT 1 -- '`, что можно имитировать следующим образом. Команда `SELECT 1` выполняется в SQL Server после выполнения первой команды `SELECT`. Теперь злоумышленник может вставить любую команду, если она умещается в длину принимаемой строки.

```
USE TSQL2012;
GO
EXEC Sales.ListCustomersByAddress @address = N''' SELECT 1 -- ';
```

6. Теперь перепишите хранимую процедуру с использованием `sp_executesql` и введите адрес в качестве параметра хранимой процедуры.

```
USE TSQL2012;
GO
IF OBJECT_ID('Sales.ListCustomersByAddress') IS NOT NULL
    DROP PROCEDURE Sales.ListCustomersByAddress;
GO
CREATE PROCEDURE Sales.ListCustomersByAddress
    @address AS NVARCHAR(60)
AS
    DECLARE @SQLString AS NVARCHAR(4000);
    SET @SQLString = N'
        SELECT companyname, contactname
        FROM Sales.Customers WHERE address = @address';
    EXEC sp_executesql
        @statement = @SQLString,
        @params = N'@address NVARCHAR(60)',
        @address = @address;
RETURN;
GO
```

7. Теперь введите правильный адрес с помощью исправленной хранимой процедуры. Обратите внимание, что сообщение, указывающее на присутствие незакрытой кавычки, отсутствует. Одиночная кавычка в качестве параметра к хранимой процедуре и к `sp_executesql` гарантирует, что он будет обрабатываться только как одна строка.

```
USE TSQL2012;
GO
EXEC Sales.ListCustomersByAddress @address = N'8901 Tsawassen Blvd.';
```

8. Выполните остальные шаги, чтобы убедиться, что не будут возвращены нежелательные данные.

```
USE TSQL2012;
GO
EXEC Sales.ListCustomersByAddress @address = N'';
EXEC Sales.ListCustomersByAddress @address = N''' -- ';
EXEC Sales.ListCustomersByAddress @address = N''' SELECT 1 -- ';
```

Задание 3. Использование выходных параметров с процедурой sp_executesql

В этом задании вы будете использовать хранимую процедуру `sp_executesql` для возвращения значения с помощью выходного параметра. Иногда удобно возвращать результаты динамического SQL в переменную. Это нельзя выполнить посредством инструкции `EXEC`, но хранимая процедура `sp_executesql` может это сделать с помощью выходных параметров.

1. Откройте SSMS и загрузите следующий сценарий хранимой процедуры в окно запроса. Сценарий использует команду `EXEC` для подсчета количества строк. Обратите внимание, что невозможно возвратить это значение счета напрямую. Команда `EXEC` просто не может осуществить обратную связь с вызывающей стороной, разве что сохранить значение во временной таблице или использовать еще какой-то механизм сохранения. Можно увидеть значение счета в выходных данных SSMS, но поместить его в переменную нельзя.

```
USE TSQL2012;
GO
DECLARE @SQLString AS NVARCHAR(4000);
SET @SQLString = N'SELECT COUNT(*) FROM Production.Products';
EXEC(@SQLString);
```

2. Хранимую процедуру `sp_executesql` можно использовать для сбора и возвращения значений вызывающей стороне с помощью выходных параметров. В следующем коде указывается ключевое слово `OUTPUT` в объявлении параметра и в назначении параметра.

```
USE TSQL2012;
GO
DECLARE @SQLString AS NVARCHAR(4000)
      , @outercount AS int;
SET @SQLString = N'SET @innercount = (SELECT COUNT(*) FROM
Production.Products)';
EXEC sp_executesql
      @statement = @SQLString,
      @params = N'@innercount AS int OUTPUT',
      @innercount = @outercount OUTPUT;
SELECT @outercount AS 'RowCount';
```

Резюме занятия

- Динамический SQL можно использовать для генерации и выполнения кода T-SQL в случаях, когда инструкции T-SQL должны строиться во время выполнения.
- Внедрение SQL-кода означает потенциальную возможность для приложений принимать входные данные, которые внедряют нежелательный код, выполняемый динамическим SQL.
- Хранимая процедура `sp_executesql` может использоваться, чтобы помочь предотвратить внедрение SQL-кода, путем параметризации соответствующих частей динамического SQL.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какие из следующих способов можно использовать для внедрения нежелательного кода в динамический SQL, когда пользовательские входные данные объединены с допустимыми командами SQL?
 - A. Вставьте строку комментария из двух тире, потом вредоносный код и затем одиночную кавычку.
 - B. Вставьте одиночную кавычку, потом вредоносный код и затем строку комментария из двух тире.
 - C. Вставьте вредоносный код перед одиночной кавычкой и строку комментария из двух тире.
2. В чем заключаются преимущества хранимой процедуры `sp_executesql` над командой `EXECUTE ()`? (Выберите все подходящие варианты.)
 - A. Хранимая процедура `sp_executesql` может выполнять параметризацию аргументов поиска и помочь предотвратить внедрение SQL-кода.
 - B. Хранимая процедура `sp_executesql` использует строки в формате Unicode.
 - C. Хранимая процедура `sp_executesql` может возвращать данные посредством выходных параметров.
3. Какие из следующих высказываний справедливы по отношению к инструкции `SET QUOTED_IDENTIFIER?` (Выберите все подходящие варианты.)
 - A. Когда параметр `QUOTED_IDENTIFIER` установлен в `ON`, инструкция позволяет использовать двойные кавычки для разделения идентификаторов T-SQL, таких как имена таблиц и столбцов.
 - B. Когда параметр `QUOTED_IDENTIFIER` установлен в `OFF`, инструкция позволяет использовать двойные кавычки для разделения идентификаторов T-SQL, таких как имена таблиц и столбцов.

- C. Когда параметр QUOTED_IDENTIFIER установлен в ON, инструкция позволяет использовать двойные кавычки для разделения строк.
- D. Когда параметр QUOTED_IDENTIFIER установлен в OFF, инструкция позволяет использовать двойные кавычки для разделения строк.

Упражнения

В следующих упражнениях вы примените полученные знания о транзакциях, обработке ошибок и динамическом SQL. Ответы на вопросы можно найти в *приложении "Ответы" в конце книги*.

Упражнение 1. Реализация обработки ошибок

Как разработчика баз данных на ключевом проекте в компании, вас попросили выполнить рефакторинг набора хранимых процедур на промышленном сервере баз данных. Вы обнаружили, что в хранимых процедурах практически отсутствует обработка ошибок, а когда она выполняется, она является ситуативной и неструктурированной. Хранимые процедуры не используют транзакции. Вам необходимо предложить план ваших действий.

1. Когда вы должны рекомендовать использование явных транзакций?
2. Когда вы должны рекомендовать использование другого уровня изоляции?
3. Какой тип обработки ошибок вы должны рекомендовать?
4. Какие планы вы должны включить в рефакторинг динамического SQL?

Упражнение 2. Реализация транзакций

Вас только что назначили в команду на новый проект в качестве разработчика баз данных. Приложение будет использовать хранимые процедуры для выполнения некоторых финансовых операций. Вы решили использовать транзакции T-SQL. Ответьте на следующие вопросы о том, что вы должны порекомендовать в указанных ситуациях.

1. В некоторых транзакциях, обновляющих таблицы, после того как сеанс выполняет чтение определенного значения из другой таблицы, важно, чтобы значение из другой таблицы не изменялось до окончания транзакции. Какой уровень изоляции транзакции подходит для этого?
2. Вы будете использовать сценарии T-SQL для развертывания новых объектов, таких как таблицы, представления или код T-SQL в базе данных. Если возникает любой тип ошибки T-SQL, весь сценарий развертывания должен прекратить выполняться. Как этого достигнуть, не добавляя сложной обработки ошибок?
3. Одна из хранимых процедур будет передавать деньги с одного счета на другой. В течение этого периода передачи никакие данные в обоих счетах не должны быть изменены, вставлены или удалены для диапазона значений, считываемых транзакцией. Какой уровень изоляции транзакции подходит для этого?

Рекомендуемые упражнения

Для успешного освоения экзаменационных тем, представленных в данной главе, выполните следующие задания.

Реализация обработки ошибок

Для выполнения следующих заданий используйте базу данных TS2012.

- **Задание 1.** Создайте транзакцию для обновления таблицы Production.Products базы данных TSQL2012 посредством приращения столбца unitprice на 5 для productid = 100. Используйте команду THROW с параметрами для создания сообщения об ошибке, если ни одна строка не обновлена.
- **Задание 2.** Добавьте в транзакцию блок TRY/CATCH.
- **Задание 3.** В блоке CATCH возвратите клиенту всю информацию об ошибках с помощью функций обработки ошибок, откатите транзакцию и повторно создайте сообщение об ошибке с помощью команды THROW.

ГЛАВА 13

Разработка и реализация процедур T-SQL

Темы экзамена

- Создание объектов баз данных.
 - Создание и изменение триггеров DML.
- Модификация данных.
 - Создание и изменение хранимых процедур (простые инструкции).
 - Работа с функциями.

Код T-SQL может быть сохранен в базах данных SQL Server с помощью процедур T-SQL, таких как хранимые процедуры, триггеры и функции. Благодаря этому код T-SQL может быть более переносимым, поскольку он остается в базе данных, и эти процедуры будут восстановлены из резервной копии вместе с данными базы данных. Эти процедуры затем могут быть выполнены из базы данных. В этой главе вы познакомитесь с созданием многоократно используемых процедур T-SQL в хранимых процедурах, триггерах и определяемых пользователем (пользовательских) функциях.

ПРЕЖДЕ ВСЕГО

Для полного освоения этой главы необходимо иметь:

- понимание основных концепций баз данных;
- опыт работы с SQL Server Management Studio (SSMS);
- некоторый опыт в написании кода на языке T-SQL;
- доступ к экземпляру SQL Server 2012 с установленной учебной базой данных TSQL2012.

Занятие 1. Разработка и реализация хранимых процедур

Хранимые процедуры — это процедуры, которые находятся в базе данных и инкапсулируют код. SQL Server допускает использование нескольких типов хранимых процедур, таких как следующие:



- хранимые процедуры T-SQL, написанные на языке T-SQL;
- хранимые процедуры CLR, сохраняемые в базе данных как сборки Microsoft .NET;
- расширенные хранимые процедуры, которые выполняют вызовы к внешним библиотекам (data definition language, DLL).

Занятия в данной главе познакомят вас с хранимыми процедурами T-SQL. Все упоминания хранимых процедур в данной главе подразумевают хранимые процедуры T-SQL. Любые ссылки на процедуры CLR или расширенные хранимые процедуры указываются явно. Это занятие посвящено разработке и реализации хранимых процедур.

Изучив материал этого занятия, вы сможете:

- ✓ Создавать основные хранимые процедуры T-SQL
- ✓ Написать хранимую процедуру, удовлетворяющую определенным требованиям
- ✓ Применять логику ветвления в хранимой процедуре
- ✓ Определять различные типы результатов хранимой процедуры
- ✓ Описать, как могут использоваться хранимые процедуры для уровня доступа к данным приложениям

Продолжительность занятия — 50 минут.

Основные сведения о хранимых процедурах

Хранимая процедура T-SQL состоит из одного пакета кода T-SQL. У хранимых процедур имеется множество важных свойств, к которым относятся следующие:

- их можно вызывать из кода T-SQL с помощью команды EXECUTE;
- им можно передавать данные через входные параметры и получать от них данные обратно через выходные параметры;
- они могут возвращать результирующие наборы запросов клиентскому приложению;
- они могут изменять данные в таблицах;
- они могут создавать, изменять и удалять таблицы и индексы.

Использование хранимых процедур T-SQL в базе данных SQL Server имеет множество преимуществ, таких как:

- инкапсуляция кода T-SQL. Одна хранимая процедура может вызываться из множества мест с параметрами, которые адаптируют код к разным начальным условиям;

- обеспечение большей безопасности для базы данных:
 - вместо того чтобы предоставить пользователю доступ к таблице базы данных напрямую, можно дать права хранимой процедуре;
 - хранимые процедуры могут помочь предотвратить внедрение SQL с помощью параметризации динамического SQL;
- представление пользователям и приложениям более универсального уровня доступа к данным:
 - хранимая процедура позволяет пользователю обходить сложную логику, чтобы получить требуемые результаты;
 - основные физические структуры таблиц базы данных могут изменяться, хранимая процедура может также быть изменена, но поскольку пользователь видит те же процедуру и параметры, ему не нужно знать об этих изменениях;
- повышение производительности посредством создания планов выполнения, которые могут повторно использоваться:
 - передавая параметры, можно повторно использовать кэшируемый план хранимой процедуры для различных значений параметров, предотвращая потребность перекомпиляции кода T-SQL;
 - хранимые процедуры могут уменьшить сетевой трафик. Если приложение должно выполнять всю работу, промежуточные результаты должны передаваться обратно приложению по сети. Аналогично, если приложение выполняет всю работу, оно должно отправить каждую команду T-SQL в SQL Server по сети.

Хранимые процедуры могут использоваться для инкапсуляции логики приложения, относящейся к данным и к административным функциям, таким как создание резервной копии и восстановление из нее. В действительности почти все инструкции T-SQL могут быть включены в хранимую процедуру. Однако надо учесть следующее:

- нельзя использовать команду `USE <database name>`;
- нельзя использовать инструкцию `CREATE` с любым из следующих типов объектов: `AGGREGATE`, `RULE`, `DEFAULT`, `CREATE`, `FUNCTION`, `TRIGGER`, `PROCEDURE` или `VIEW`;
- можно создать, изменить и удалить таблицу и индекс с помощью инструкций `CREATE`, `ALTER` и `DROP`.

Рассмотрим следующий код T-SQL, который посыпает запрос базе данных TSQL2012 найти все заказы, размещенные клиентом со значением идентификатора ID, равным 37, во втором квартале 2007 г.

```
USE TSQL2012;
GO
SELECT orderid, custid, shipperid, orderdate, requireddate, shippeddate
FROM Sales.Orders
WHERE custid = 37
AND orderdate >= '2007-04-01'
AND orderdate < '2007-07-01';
```

Этот запрос имеет ограничения, поскольку в нем есть литеральные значения в предложении WHERE. Чтобы сделать код более общим, можно вместо литеральных значений использовать переменные, как в следующем примере:

```
USE TSQ2012;
GO
DECLARE @custid AS INT,
        @orderdatefrom AS DATETIME,
        @orderdateto AS DATETIME,
SET @custid = 37;
SET @orderdatefrom = '2007-04-01';
SET @orderdateto = '2007-04-01';
SELECT orderid, custid, shipperid, orderdate, requireddate, shippeddate
FROM Sales.Orders
WHERE custid = @custid
      AND orderdate >= @orderdatefrom
      AND orderdate < @orderdateto;
GO
```

Все, что было нужно сделать, — это объявить три переменные для литеральных значений и присвоить этим переменным значения перед выполнением запроса. Когда пакет кода T-SQL будет выполнен применительно к исходной таблице Sales.Orders базы данных TSQ2012, вы увидите три возвращенных строки. Теперь проверьте, как может выглядеть этот код, если его затем инкапсулировать в хранимую процедуру.

```
IF OBJECT_ID('Sales.GetCustomerOrders', 'P') IS NOT NULL
    DROP PROC Sales.GetCustomerOrders;
GO
CREATE PROC Sales.GetCustomerOrders
    @custid      AS INT,
    @orderdatefrom AS DATETIME = '19000101',
    @orderdateto   AS DATETIME = '99991231',
    @numrows      AS INT = 0 OUTPUT
AS
BEGIN
    SET NOCOUNT ON;
    SELECT orderid, custid, shipperid, orderdate, requireddate, shippeddate
    FROM Sales.Orders
    WHERE custid = @custid
          AND orderdate >= @orderdatefrom
          AND orderdate < @orderdateto;
    SET @numrows = @@ROWCOUNT; RETURN;
END
GO
```

После выполнения предыдущего кода и создания хранимой процедуры вы можете вызвать эту процедуру следующим образом:

```

DECLARE @rowsreturned AS INT;
EXEC Sales.GetCustomerOrders
    @custid      = 37,
    @orderdatefrom = '20070401',
    @orderdateto = '20070701',
    @numrows = @rowsreturned OUTPUT;
SELECT @rowsreturned AS "Rows Returned";
GO

```

В результате мы получаем тот же самый набор строк с данными, но кроме этого, мы также получаем число строк в специальном параметре `OUTPUT`. Изучите эту процедуру, а также способ, которым она была вызвана, шаг за шагом, чтобы понять основные свойства хранимой процедуры T-SQL.

Проверка существования хранимой процедуры

При попытке создать хранимую процедуру, которая уже существует, команда `CREATE` выдаст ошибку. Если хранимая процедура уже существует, ее можно изменить, но если попытаться изменить несуществующую хранимую процедуру, команда `ALTER` завершится ошибкой. Первая часть сценария решает эту проблему с помощью условной команды удаления хранимой процедуры `DROP` перед попыткой создать процедуру.

```

IF OBJECT_ID('Sales.GetCustomerOrders', 'P') IS NOT NULL
    DROP PROC Sales.GetCustomerOrders;
GO

```

Есть множество способов проверить существование объекта базы данных, такого как хранимая процедура. Например, можно проверить метаданные в `sys.objects`. Но наименее подробные сведения, возможно, выдает функция `OBJECT_ID()`.

Параметры хранимой процедуры

Хранимые процедуры принимают параметры, которые имеют синтаксис, очень похожий на синтаксис `DECLARE` для переменных. Взгляните на инструкцию `CREATE PROCEDURE`.

```

CREATE PROC Sales.GetCustomerOrders
    @custid      AS INT,
    @orderdatefrom AS DATETIME = '19000101',
    @orderdateto   AS DATETIME = '99991231',
    @numrows       AS INT = 0 OUTPUT
AS
BEGIN
    SET NOCOUNT ON;
    SELECT orderid, custid, shipperid, orderdate, requireddate, shippeddate
    FROM [Sales].[Orders]
    WHERE custid = @custid
        AND orderdate >= @orderdatefrom
        AND orderdate < @orderdateto;

```

```
SET @numrows = @@ROWCOUNT;
RETURN;
END
```

При создании хранимой процедуры можно записывать инструкцию полностью — CREATE PROCEDURE или использовать сокращенную форму CREATE PROC. Затем должно идти имя хранимой процедуры.

Относительно параметров хранимой процедуры необходимо помнить следующее.

- Не обязательно указывать параметры в хранимой процедуре, но если вы решили их использовать, они должны быть перечислены сразу после начала процедуры.
- Параметры могут быть обязательными или необязательными.
 - При отсутствии инициализации по умолчанию параметр является обязательным. В предыдущем коде @custid — это обязательный параметр.
 - Если имеется инициализация по умолчанию, параметр является необязательным. В предыдущем коде параметры @orderdatefrom и @orderdateto — это необязательные параметры. Если необязательному параметру при вызове процедуры не присваивается значение, в оставшейся части процедуры будет использоваться значение по умолчанию.
- Параметры хранимой процедуры интерпретируются как переменные для оставшейся части процедуры.
- Можно инициализировать значения параметров так же, как значения переменных.
- Ключевое слово OUTPUT определяет специальный параметр, который возвращает значения вызывающей стороне. Выходные параметры всегда являются необязательными параметрами.
- Команда AS обязательно должна стоять после списка параметров.

Блок BEGIN/END

Можно заключить код в хранимой процедуре в блок BEGIN/END. Хотя это не является обязательным требованием, использование блока BEGIN/END помогает сделать код более понятным.

Инструкция SET NOCOUNT ON

Можно встроить инструкцию NOCOUNT со значением ON внутрь хранимой процедуры, чтобы запретить вывод сообщений, подобный (3 row(s) affected), при каждом выполнении процедуры.

СОВЕТ

Подготовка к экзамену

Параметр NOCOUNT устанавливается в значение ON или OFF для хранимой процедуры при ее создании. Помещение SET NOCOUNT ON в начало каждой хранимой процедуры предотвращает возвращение этой процедурой сообщения клиенту. Кроме того, SET NOCOUNT ON может повысить производительность часто выполняемых хранимых процедур, поскольку требуется меньше сетевых взаимодействий, если сообщение "row(s) affected" не возвращается клиенту.

Команда *RETURN* и коды возврата

Хранимая процедура заканчивается тогда, когда заканчивается пакет, но с помощью команды `RETURN` можно заставить процедуру завершиться в любой точке. В одной процедуре можно использовать более одной команды `RETURN`. Эта команда останавливает выполнение процедуры и возвращает управление обратно вызывающей стороне. Инструкции, стоящие после команды `RETURN`, не выполняются.

Инструкция `RETURN` сама вынуждает SQL Server посыпать код состояния обратно вызывающей стороне. При успешном выполнении статус равен 0, при наличии ошибки статус равен отрицательному числу. Но не следует полагаться на номера ошибок, т. к. они не надежны. Вместо этого следует использовать номера ошибок SQL Server, возвращаемые функцией `@ERROR` или функцией `ERROR_NUMBER()`, вызываемой в блоке `CATCH`.

Можно отправить собственные коды возврата назад вызывающей стороне, вставив целочисленные значения после инструкции `RETURN`. Но если необходимо отправить информацию обратно вызывающей стороне, считается более правильным использовать параметр `OUTPUT`.

Выполнение хранимых процедур

Для выполнения хранимой процедуры в T-SQL используется инструкция `EXECUTE` или сокращенно `EXEC`. Если хранимая процедура не имеет параметров, инструкция `EXEC` должна стоять перед именем хранимой процедуры, как в следующем примере:

```
EXEC sp_configure;
```

Этот код выполняет системную хранимую процедуру `sp_configure`. Поскольку это системная хранимая процедура из главной базы данных, она может быть выполнена из любой базы данных.

Если выполнение хранимой процедуры — первая инструкция пакета кода T-SQL или единственная инструкция, выбранная в окне запроса, инструкция `EXEC` не нужна. Если же хранимая процедура является второй или более поздней инструкцией, ей должна предшествовать инструкция `EXEC` или `EXECUTE`.

СОВЕТ

Подготовка к экзамену

При вызове хранимой процедуры всегда используйте команду `EXEC`. Это поможет избежать неожиданных и сбивающих с толку ошибок. Даже если инструкция уже не является первой в пакете, она все равно будет выполнена.

Входные параметры

Если хранимая процедура имеет входные параметры, можно передать значение параметра, либо поместив его в соответствующую позицию, либо связав значение с именем параметра. Например, в предыдущем разделе мы создали хранимую процедуру в базе данных `TSQL2012`, которая была названа `Sales.GetCustomerOrders`. Чтобы вызвать эту процедуру с передачей параметра по позиции, наберите следующий код:

```
EXEC Sales.GetCustomerOrders 37, '20070401', '20070701';
```

Заметьте, вы можете проигнорировать необязательный параметр OUTPUT.

Теперь вызовите эту процедуру, перечислив параметры по имени как в следующем примере:

```
EXEC Sales.GetCustomerOrders @custid = 37, @orderdatefrom = '20070401',
@orderdateto = '20070701';
```

Когда значения параметров передаются с использованием имен параметров, имена параметров можно указывать в любом порядке. Например, при вызове хранимой процедуры Sales.GetCustomerOrders можно поместить @custid после параметров даты. (Пока что будем игнорировать необязательный параметр OUTPUT. Выходные параметры рассматриваются в следующем разделе.)

```
EXEC Sales.GetCustomerOrders
    @orderdatefrom = '20070401',
    @orderdateto   = '20070701',
    @custid        = 37;
GO
```

Однако когда вы передаете значения параметров по позиции, необходимо использовать точное положение параметров, определенное в инструкции CREATE PROCEDURE.

```
EXEC Sales.GetCustomerOrders 37, '20070401', '20070701';
GO
```

СОВЕТ

Подготовка к экзамену

Наиболее правильным считается указывать имена параметров при вызове хранимых процедур. Хотя передача параметров по позиции может быть более компактной, она при этом является наиболее ненадежной. Если параметры передаются по имени и порядок следования параметров изменится в хранимой процедуре, вызов этой процедуры все равно будет работать.

Поскольку параметры даты являются необязательными, их можно удалить полностью из вызова процедуры, как показано в следующем примере:

```
EXEC Sales.GetCustomerOrders
    @custid = 37;
GO
```

Выходные параметры

Для использования выходных параметров надо добавить ключевое слово OUTPUT (в сокращенном варианте OUT) после параметра при объявлении его в инструкции CREATE PROC.

```
CREATE PROC Sales.GetCustomerOrders
    @custid      AS INT,
    @orderdatefrom AS DATETIME = '19000101',
    @orderdateto   AS DATETIME = '99991231',
    @numrows      AS INT = 0 OUTPUT
AS <rest of procedure>
```

Для извлечения данных из выходного параметра ключевое слово OUTPUT также должно использоваться при вызове хранимой процедуры, вдобавок должна быть предоставлена переменная для помещения в нее значения при его возвращении. Если в значении процедуры ключевое слово OUTPUT отсутствует, никакое значение не будет возвращено в переменной. Например, в следующем коде параметр @rowsreturned принимает значение NULL, поскольку ключевое слово OUTPUT не указано после строки параметра @numrows.

```
DECLARE @rowsreturned AS INT;
EXEC Sales.GetCustomerOrders
@custid      = 37,
@orderdatefrom = '20070401',
@orderdateto  = '20070701',
@numrows      = @rowsreturned;
SELECT @rowsreturned AS 'Rows Returned';
GO
```

После добавления ключевого слова OUTPUT процедура работает как положено.

```
DECLARE @rowsreturned AS INT;
EXEC Sales.GetCustomerOrders
@custid      = 37,
@orderdatefrom = '20070401',
@orderdateto  = '20070701',
@numrows      = @rowsreturned OUTPUT;
SELECT @rowsreturned AS 'Rows Returned';
GO
```

Логика ветвления

В языке T-SQL имеется несколько инструкций, которые можно использовать для управления потоком выполнения кода. Эти конструкции можно использовать в сценариях T-SQL, а также в хранимых процедурах. При использовании логики ветвления вы даете возможность вашему коду обрабатывать сложные ситуации, которые требуют различных действий в зависимости от входных данных. К инструкциям управления потоком выполнения относятся следующие:

- IF/ELSE;
- WHILE (с BREAK и CONTINUE);
- WAITFOR;
- GOTO;
- RETURN (обычно внутри процедур T-SQL).

В последнем разделе уже рассматривалась инструкция RETURN, которая является обычным компонентом хранимой процедуры. Оставшаяся часть занятия 1 посвящена остальным инструкциям управления потоком выполнения.

Конструкция IF/ELSE

Конструкция IF/ELSE предоставляет возможность выполнять код в соответствии с условиями. Выражение вводится после ключевого слова IF, и если оно дает значение "истина", инструкция (или блок инструкций) после ключевого слова IF будет выполнена. Можно использовать необязательное ключевое слово ELSE, чтобы добавить другую инструкцию или блок инструкций, которые будут выполнены, если выражение в инструкции IF даст значение "ложь". Далее приведен пример использования этой конструкции.

```
DECLARE @var1 AS INT, @var2 AS INT;
SET @var1 = 1;
SET @var2 = 2;
IF @var1 = @var2
    PRINT 'The variables are equal';
ELSE
    PRINT '@var1 equals @var2';
GO
```

Если инструкции IF или ELSE используются без блока BEGIN/END, каждая из них обрабатывает только одну инструкцию. Посмотрите на следующий код:

```
DECLARE @var1 AS INT, @var2 AS INT;
SET @var1 = 1;
SET @var2 = 1;
IF @var1 = @var2
    PRINT 'The variables are equal';
ELSE
    PRINT 'The variables are not equal';
    PRINT '@var1 does not equal @var2';
GO
```

Хотя вторая инструкция PRINT расположена с отступом, чтобы казалось, что она будет выполнена только при условии равенства переменных, фактически она находится вне области действия условия IF. Чтобы она попала в эту область действия, следует заключить обе инструкции PRINT внутрь блока BEGIN/END, как показано в следующем примере:

```
DECLARE @var1 AS INT, @var2 AS INT;
SET @var1 = 1;
SET @var2 = 1;
IF @var1 = @var2
    BEGIN
        PRINT 'The variables are equal';
        PRINT '@var1 equals @var2';
    END
ELSE
    BEGIN
        PRINT 'The variables are not equal';
```

```

PRINT '@var1 does not equal @var2';
END
GO

```

Конструкция WHILE

С помощью конструкции `WHILE` можно создавать циклы внутри T-SQL, чтобы выполнять блок инструкций так долго, сколько условие продолжает принимать значение "истина". Конструкцию `WHILE` можно использовать в курсорах или саму по себе. Ключевое слово `WHILE` ставится перед условием, которое может принимать значение "истина" или "ложь". Если условие принимает значение "истина" при первой проверке, управление выполнением передается в цикл, выполняются все команды в цикле в первый раз и затем проверяется условие. При каждом повторении цикла условие `WHILE` снова проверяется. Как только условие дает значение "ложь", цикл завершается, и управление выполнением передается инструкции, следующей после цикла `WHILE`.

К СВЕДЕНИЮ Курсоры и инструкция WHILE

Дополнительную информацию об использовании цикла `WHILE` внутри T-SQL можно найти в главе 16.

Следующий пример цикла `WHILE` просто выполняет счет от 1 до 10 и выходит из цикла.

```

SET NOCOUNT ON;
DECLARE @count AS INT = 1;
WHILE @count <= 10
BEGIN
    PRINT CAST(@count AS @count);
    SET @count += 1;
END;

```

В предыдущем коде целочисленное `@count` приводится к типу `NVARCHAR`, чтобы выходные данные было удобнее читать. Инструкция `SET` в теле цикла увеличивает значение `@count` на 1, так что условие после ключевого слова `WHILE` в конце концов примет значение "ложь". Если условие никогда не примет значение "ложь", цикл `WHILE` может выполняться вечно, превращаясь в так называемый бесконечный цикл.

СОВЕТ

Подготовка к экзамену

Когда вы пишете цикл `WHILE`, исключительно важно убедиться в том, что в нем будет событие, которое в конечном итоге приведет к получению циклом `WHILE` значения "ложь" и прервёт цикл. Всегда проверяйте тело цикла `WHILE`, чтобы убедиться, что счетчик увеличивается или значение меняется таким образом, что цикл сможет завершиться при всех условиях.

Внутри цикла `WHILE` можно использовать инструкцию `BREAK` для немедленного завершения цикла и инструкцию `CONTINUE`, чтобы дать возможность перейти к началу цикла. В качестве примера рассмотрите следующий код:

```
GO
SET NOCOUNT ON;
DECLARE @count AS INT = 1;
WHILE @count <= 100
BEGIN
    IF @count = 10
        BREAK;
    IF @count = 5
        BEGIN
            SET @count += 2;
            CONTINUE;
        END
    PRINT CAST(@count AS NVARCHAR);
    SET @count += 1;
END;
```

После выполнения этого кода вы получите следующий результат:

```
1
2
3
4
7
8
9
```

Номера 5 и 6 пропущены из-за ветви `CONTINUE`; выходной набор заканчивается цифрой 9 и не доходит до 10 из-за инструкции `BREAK`. Инструкции `BREAK` и `CONTINUE`, как правило, не нужны, поскольку можно достичь того же результата, добавив условную логику в сам цикл.

СОВЕТ Подготовка к экзамену

Хотя блок `BEGIN/END` является необязательным в цикле `WHILE`, если у вас есть только одна инструкция, считается наиболее правильным использовать его. Блок `BEGIN/END` помогает правильно организовать ваш код, делает его более удобным для чтения и будущего изменения. Любой блок инструкций в цикле `WHILE`, содержащий более одной инструкции, требует включения конструкции `BEGIN/END`.

Что особенно важно при итерации значений в цикле `WHILE`, это итерация уникального значения. Например, в приведенном далее коде цикл `WHILE` выводит каждое значение `categoryid` из всей таблицы `Production.Categories`.

```
DECLARE @categoryid AS INT;
SET @categoryid = (SELECT MIN(categoryid) FROM Production.Categories);
WHILE @categoryid IS NOT NULL
BEGIN
    PRINT CAST(@categoryid AS NVARCHAR);
    SET @categoryid = (SELECT MIN(categoryid) FROM Production.Categories
                       WHERE categoryid > @categoryid);
END;
GO
```

Вы получите на выходе следующий результат:

1
2
3
4
5
6
7
8

В случае со значением `categoryid`, являющимся первичным ключом в таблице, вы знаете, что каждое значение будет уникально. Теперь представьте, что вы хотите выполнять итерацию для имени категории. Измените код процедуры, заменив имена переменных столбцов и указав тип данных для новой переменной `@categoryname`.

```
DECLARE @categoryname AS NVARCHAR(15);
SET @categoryname = (SELECT MIN(categoryname) FROM Production.Categories);
WHILE @categoryname IS NOT NULL
BEGIN
    PRINT @categoryname;
    SET @categoryname = (SELECT MIN(categoryname) FROM Production.Categories
    WHERE categoryname > @categoryname);
END;
GO
```

Вы получите на выходе следующее:

Beverages
Condiments
Confections
Dairy Products
Grains/Cereals
Meat/Poultry
Produce
Seafood

На этот раз мы имеем 8 строк и 8 разных имен категорий. Но это произошло случайно: в реальной таблице `Production.Categories` уникальность имен категорий не обеспечивается, поэтому возможны дубликаты.

Важно отметить, что когда условие `WHILE` тестирует элементы, которые могут содержать дубликаты, оно захватывает только их общее значение, пропуская все остальные. Поэтому, например, не важно, сколько дубликатов имен категорий имеется в таблице `Production.Categories`, предыдущий цикл `WHILE` покажет только различающиеся значения.

Чтобы получить все строки, необходимо выбрать столбец, уникальность которого гарантирована. Это может быть столбец `categoryname` при наличии ограничения уникальности или уникального индекса на этом столбце, или это может быть `categoryid`, являющийся первичным ключом и поэтому гарантированно уникальный.

Команда WAITFOR

Команда WAITFOR в действительности не изменяет управление потоком и не вызывает ветвления, но она рассматривается здесь, поскольку может вызывать паузу в выполнении инструкции на определенный период времени. Команда WAITFOR имеет три варианта: WAITFOR DELAY, WAITFOR TIME и WAITFOR RECEIVE (WAITFOR RECEIVE используется только с компонентом Service Broker).

Опция WAITFOR DELAY вызывает задержку выполнения на указанный период времени. Например, следующий код останавливает выполнение кода на 20 секунд.

```
WAITFOR DELAY '00:00:20';
```

Опция WAITFOR TIME приостанавливает выполнение до указанного времени. Например, следующий код ждет до 23:45.

```
WAITFOR TIME '23:46:00';
```

Инструкция GOTO

Инструкция GOTO позволяет заставить код перепрыгнуть на определенную метку в коде T-SQL. При этом весь промежуточный код T-SQL пропускается. Например, в следующем коде вторая инструкция PRINT пропускается.

```
PRINT 'First PRINT statement';
GOTO MyLabel;
PRINT 'Second PRINT statement';
MyLabel:
PRINT 'End';
```

Не рекомендуется использовать инструкцию GOTO, поскольку она быстро приводит к усложнению и запутыванию кода (неструктурированный код), другие конструкции T-SQL выполняют эту же работу лучше.

Разработка хранимых процедур

При разработке хранимых процедур следует учитывать несколько моментов, влияющих на поведение процедур.

Результаты хранимых процедур

Хранимые процедуры могут возвращать наборы обратно клиенту на основании запроса, выполненного процедурой. В действительности они могут посыпать обратно более одного результирующего набора. Например, следующий код демонстрирует простую процедуру, которая возвращает два результирующих набора, каждый из которых содержит только одну строку.

```
IF OBJECT_ID('Sales.ListSampleResultsSets', 'P') IS NOT NULL
    DROP PROC Sales.ListSampleResultsSets;
GO
CREATE PROC Sales.ListSampleResultsSets
```

```
AS
BEGIN
    SELECT TOP (1) productid, productname, supplierid,
        categoryid, unitprice, discontinued
    FROM Production.Products;
    SELECT TOP (1) orderid, productid, unitprice, qty, discount
    FROM Sales.OrderDetails;
END
GO
EXEC Sales.ListSampleResultsSets
```

К другим типам результатов, которые может возвращать хранимая процедура, относятся значения параметров `OUTPUT` и коды возврата, возвращенные инструкцией `RETURN`, которые были рассмотрены ранее в данном занятии.

Вызов других хранимых процедур

Можно вызывать из хранимых процедур другие хранимые процедуры. В действительности это обычный способ инкапсуляции и повторного использования кода. Но при вызове других хранимых процедур необходимо учитывать следующее.

- Если временная таблица создается в одной хранимой процедуре — например, назовем ее `Proc1`, — эта временная таблица видна всем другим хранимым процедурам, вызываемым из `Proc1`. Но эта временная таблица не видна процедурам, вызывающим `Proc1`.
- Переменные, объявленные в `Proc1`, и параметры процедуры `Proc1` не видны никаким процедурам, вызываемым процедурой `Proc1`.

Хранимые процедуры и обработка ошибок

Для защиты хранимых процедур от ошибок можно использовать блок `TRY/CATCH` с инструкцией `THROW`, так же как и в любом сценарии T-SQL. Подробнее об использовании хранимых процедур и блока `TRY/CATCH` написано в *занятии 2 главы 12*.

Динамический SQL в хранимых процедурах

Так же как и в сценариях T-SQL, динамический SQL можно использовать внутри хранимых процедур. Когда хранимые процедуры, использующие динамический SQL, представляются внешним пользователям, необходимо гарантировать защиту кода от внедрения SQL-кода. Информацию о динамическом SQL и предотвращении внедрения SQL-кода можно найти в *занятии 3 главы 12*.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Назовите два типа параметров для хранимых процедур T-SQL.
2. Может ли хранимая процедура охватывать несколько пакетов T-SQL?

Ответы на контрольные вопросы

- Хранимая процедура T-SQL может иметь входные и выходные параметры.
- Нет, хранимая процедура может содержать только один пакет кода T-SQL.

ПРАКТИКУМ Реализация хранимых процедур

В этом практикуме вам предстоит написать хранимые процедуры двух типов, применяя знания о них, полученные в данном занятии.

Задание 1. Создание хранимой процедуры для выполнения административных задач

В этом задании вам нужно разработать хранимую процедуру для создания резервной копии базы данных. Сначала вы создадите сценарий и цикл WHILE, а затем постепенно будете усовершенствовать код, чтобы в конечном итоге создать хранимую процедуру с параметром, указывающим тип базы данных, для которой нужно создать резервную копию.

- Разработайте цикл WHILE, который выполнит итерацию на наборе несистемных баз данных. Инструкция PRINT является заполнителем (указателем места размещения) команды резервного копирования. Выполните цикл в SSMS для проверки того, что имена несистемных баз данных выводятся на печать.

```
DECLARE @databasename AS NVARCHAR(128);
SET @databasename = (SELECT MIN(name) FROM sys.databases
WHERE name NOT IN ('master', 'model', 'msdb', 'tempdb'));
WHILE @databasename IS NOT NULL
BEGIN
    PRINT @databasename; SET @databasename = (SELECT MIN(name)
    FROM sys.databases WHERE name NOT
    IN ('master', 'model', 'msdb', 'tempdb') AND name > @databasename);
END
GO
```

- В нескольких следующих шагах вам нужно создать имя для файла резервной копии базы данных в формате <database name>_<yyyymmdd>_<hhmmss>.bak. Используйте текущую дату и время и функцию CONVERT() для конвертирования их в строковую переменную.

```
SELECT CONVERT(NVARCHAR, GETDATE(), 120)
```

- Теперь используйте функцию REPLACE() для удаления тире из даты и двоеточий из времени и заменить пробелы подчеркиванием.

```
SELECT REPLACE(REPLACE(REPLACE(CONVERT(NVARCHAR, GETDATE(), 120), ' ', '_'),
':', ''), '-','');
```

- Теперь можно добавить команду BACKUP DATABASE вместо инструкции PRINT. Для простоты выполните полное резервное копирование (FULL) базы данных в один файл. Имя файла резервной копии — это имя базы данных, стоящее перед стро-

кой времени, и затем расширение bak. Для создания строки времени используйте функции CONVERT() и REPLACE() из пп. 2 и 3 и поместите результат в переменную @timecomponent.

```

DECLARE @databasename AS NVARCHAR(128),
        @timecomponent AS NVARCHAR(50),
        @sqlcommand AS NVARCHAR(1000);
SET @databasename = (SELECT MIN(name) FROM sys.databases WHERE name
                     NOT IN ('master', 'model', 'msdb', 'tempdb'));
WHILE @databasename IS NOT NULL
BEGIN
    SET @timecomponent = REPLACE(REPLACE(REPLACE(CONVERT(NVARCHAR,
        GETDATE(), 120), ' ', '_'), ':', ''), '-');
    SET @sqlcommand = 'BACKUP DATABASE ' + @databasename +
        ' TO DISK = ''C:\Backups\' + @databasename + '_' + @timecomponent +
        '.bak''';
    PRINT @sqlcommand; --EXEC(@sqlcommand); SET @databasename =
        (SELECT MIN(name) FROM sys.databases WHERE name
         NOT IN ('master', 'model', 'msdb', 'tempdb')
         AND name > @databasename);
END;
GO

```

5. Теперь конвертируйте сценарий в хранимую процедуру.

```

IF OBJECT_ID('dbo.BackupDatabases', 'P') IS NOT NULL
    DROP PROCEDURE dbo.BackupDatabases
GO
CREATE PROCEDURE dbo.BackupDatabases AS
BEGIN
    DECLARE @databasename AS NVARCHAR(128),
            @timecomponent AS NVARCHAR(50),
            @sqlcommand AS NVARCHAR(1000);
    SET @databasename = (SELECT MIN(name) FROM sys.databases
                          WHERE name NOT IN ('master', 'model', 'msdb', 'tempdb'));
    WHILE @databasename IS NOT NULL
    BEGIN
        SET @timecomponent = REPLACE(REPLACE(REPLACE(
            CONVERT(NVARCHAR, GETDATE(), 120), ' ', '_'), ':', ''), '-');
        SET @sqlcommand = 'BACKUP DATABASE ' + @databasename +
            ' TO DISK = ''C:\Backups\' + @databasename + '_' +
            @timecomponent + '.bak''';
        PRINT @sqlcommand; --EXEC(@sqlcommand);
        SET @databasename = (SELECT MIN(name) FROM sys.databases
                              WHERE name
                               NOT IN ('master', 'model', 'msdb', 'tempdb')
                               AND name > @databasename);
    END;

```

```
    RETURN;
END;
GO
```

6. Выполнив код в п. 5 для создания процедуры, протестируйте ее. Процедура должна выводить на печать набор команд BACKUP DATABASE, по одной на каждую базу данных.

```
EXEC dbo.BackupDatabases
```

7. Наконец, добавьте параметр @databasetype в процедуру; если значение этого параметра равно 'user', сделайте резервную копию всех пользовательских баз данных; если значение этого параметра — 'system', сделайте резервную копию всех системных баз данных.

```
IF OBJECT_ID('dbo.BackupDatabases', 'P') IS NOT NULL
    DROP PROCEDURE dbo.BackupDatabases;
GO
CREATE PROCEDURE dbo.BackupDatabases
    @databasetype AS NVARCHAR(30)
AS
BEGIN
    DECLARE @databasename AS NVARCHAR(128),
        @timecomponent AS NVARCHAR(50),
        @sqlcommand AS NVARCHAR(1000);
    IF @databasetype NOT IN ('User', 'System')
        BEGIN
            THROW 50000,
                'dbo.BackupDatabases: @databasetype must be User or System', 0;
            RETURN;
        END;
    IF @databasetype = 'System'
        SET @databasename = (SELECT MIN(name) FROM sys.databases
            WHERE name IN ('master', 'model', 'msdb'));
    ELSE
        SET @databasename = (SELECT MIN(name) FROM sys.databases
            WHERE name NOT IN ('master', 'model', 'msdb', 'tempdb'));
    WHILE @databasename IS NOT NULL
        BEGIN
            SET @timecomponent = REPLACE(REPLACE(REPLACE(CONVERT(
                NVARCHAR, GETDATE(), 120), ' ', '_'), ':', ''), '-');
            SET @sqlcommand = 'BACKUP DATABASE ' + @databasename +
                ' TO DISK = ''C:\Backups\' + @databasename + '_' +
                @timecomponent + '.bak'''';
            PRINT @sqlcommand;
            --EXEC(@sqlcommand);
            IF @databasetype = 'System'
                SET @databasename = (SELECT MIN(name) FROM sys.databases
                    WHERE name IN ('master', 'model', 'msdb')
                    AND name > @databasename);
```

```

ELSE
    SET @databasename = (SELECT MIN(name) FROM sys.databases
        WHERE name NOT IN ('master', 'model', 'msdb', 'tempdb')
        AND name > @databasename);
END;
RETURN;
END;
GO

```

8. Теперь протестируйте процедуру. Если вы не передадите никаких параметров или передадите параметры, отличные от 'user' и 'system', то должны получить сообщение об ошибке. Если передадите правильные параметры, вы должны получить выведенные на печать команды резервного копирования.

```

EXEC dbo.BackupDatabases;
GO
EXEC dbo.BackupDatabases 'User';
GO
EXEC dbo.BackupDatabases 'System';
GO
EXEC dbo.BackupDatabases 'Unknown'

```

9. Если вы удовлетворены работой хранимой процедуры, можно удалить комментарий из команды EXEC и закомментировать команду PRINT, чтобы создать резервную копию баз данных.

Задание 2. Разработка хранимой процедуры *INSERT* для уровня доступа к данным

В этом задании вам предстоит создать базовую хранимую процедуру INSERT, которая может использоваться приложением для вставки данных в базу данных TSQL2012. Затем нужно добавить в хранимую процедуру тестирование параметра и обработку ошибок.

1. Откройте SSMS и пустое окно запросов. Загрузите или введите следующий код для создания хранимой процедуры. Помните, что инструкция INSERT является абсолютно незащищенной.

```

-- Версия 1. Простая хранимая процедура вставки
USE TSQL2012;
GO
IF OBJECT_ID('Production.InsertProducts', 'P') IS NOT NULL
    DROP PROCEDURE Production.InsertProducts
GO
CREATE PROCEDURE Production.InsertProducts
    @productname AS NVARCHAR(40),
    @supplierid AS INT,
    @categoryid AS INT,
    @unitprice AS MONEY = 0,
    @discontinued AS BIT = 0 AS

```

```

BEGIN
    INSERT Production.Products (productname, supplierid, categoryid,
                                unitprice, discontinued)
        VALUES (@productname, @supplierid, @categoryid, @unitprice,
                @discontinued)
    RETURN;
END;
GO

```

2. Чтобы протестировать процедуру, выполните ее с правильными параметрами, как показано в следующем коде. Проверьте результаты и затем удалите тестовую строку.

```

EXEC Production.InsertProducts
    @productname = 'Test Product',
    @supplierid = 10,
    @categoryid = 1,
    @unitprice = 100,
    @discontinued = 0;
GO
-- Проверка результатов
SELECT * FROM Production.Products WHERE productname = 'Test Product';
GO
-- Удаление новой строки
DELETE FROM Production.Products WHERE productname = 'Test Product';

```

3. Теперь протестируйте хранимую процедуру с неправильным значением параметра. Вы должны получить сообщение об ошибке от SQL Server.

```

EXEC Production.InsertProducts
    @productname = 'Test Product',
    @supplierid = 10,
    @categoryid = 1,
    @unitprice = -100,
    @discontinued = 0

```

4. Теперь добавьте обработку ошибок в процедуру с помощью блока TRY/CATCH. Загрузите или введите следующий код в окне запроса и выполните его, чтобы создать хранимую процедуру.

```

-- Версия 2. С обработкой ошибок
IF OBJECT_ID('Production.InsertProducts', 'P') IS NOT NULL
    DROP PROCEDURE Production.InsertProducts
GO
CREATE PROCEDURE Production.InsertProducts
    @productname AS NVARCHAR(40),
    @supplierid AS INT,
    @categoryid AS INT,
    @unitprice AS MONEY = 0,
    @discontinued AS BIT = 0 AS

```

```

BEGIN
BEGIN TRY
    INSERT Production.Products (productname, supplierid, categoryid,
        unitprice, discontinued)
    VALUES (@productname, @supplierid, @categoryid, @unitprice,
        @discontinued);
END TRY
BEGIN CATCH
    THROW;
    RETURN;
END CATCH;
END;
GO

```

5. Протестируйте хранимую процедуру с помощью неверного параметра unitprice.

```

EXEC Production.InsertProducts
    @productname = 'Test Product', @supplierid = 10, @categoryid = 1,
    @unitprice = -100, @discontinued = 0

```

6. Теперь добавьте тестирование параметра в хранимую процедуру. Загрузите или введите следующий код:

```

-- Версия 3. С тестированием параметров
IF OBJECT_ID('Production.InsertProducts', 'P') IS NOT NULL
    DROP PROCEDURE Production.InsertProducts
GO
CREATE PROCEDURE Production.InsertProducts
    @productname AS NVARCHAR(40),
    @supplierid AS INT,
    @categoryid AS INT,
    @unitprice AS MONEY = 0,
    @discontinued AS BIT = 0
AS
BEGIN
    DECLARE @ClientMessage NVARCHAR(100);
    BEGIN TRY
        -- Тестирование параметров
        IF NOT EXISTS(SELECT 1 FROM Production.Suppliers
            WHERE supplierid = @supplierid)
        BEGIN
            SET @ClientMessage = 'Supplier id ' +
                CAST(@supplierid AS VARCHAR) + ' is invalid';
            THROW 50000, @ClientMessage, 0;
        END
        IF NOT EXISTS(SELECT 1 FROM Production.Categories
            WHERE categoryid = @categoryid)
        BEGIN
            SET @ClientMessage = 'Category id
                + CAST(@categoryid AS VARCHAR) THROW 50000, @ClientMessage, 0;
        END;
    
```

```
IF NOT(@unitprice >= 0)
BEGIN
    SET @ClientMessage = 'Unitprice '
        + CAST(@unitprice AS VARCHAR) + ' is invalid. Must be >= 0.';
    THROW 50000, @ClientMessage, 0;
END;
-- Выполнение вставки
INSERT Production.Products (productname, supplierid, categoryid,
                               unitprice, discontinued)
VALUES (@productname, @supplierid, @categoryid, @unitprice,
        @discontinued);
END TRY
BEGIN CATCH
    THROW;
END CATCH;
END;
GO
```

7. Протестируйте хранимую процедуру с помощью параметра `unitprice`, находящегося вне диапазона.

```
EXEC Production.InsertProducts
    @productname = 'Test Product',
    @supplierid = 10, @categoryid = 1,
    @unitprice = -100, @discontinued = 0
```

8. Протестируйте хранимую процедуру, используя другой неверный параметр, в данном случае — `supplierid`.

```
EXEC Production.InsertProducts
    @productname = 'Test Product',
    @supplierid = 100,
    @categoryid = 1,
    @unitprice = 100,
    @discontinued = 0
```

9. Удалите хранимую процедуру.

Резюме занятия

- Хранимые процедуры T-SQL — это модули кода T-SQL, которые хранятся в базе данных и могут быть выполнены с помощью команды T-SQL `EXECUTE`.
- Хранимые процедуры можно использовать для инкапсуляции кода на стороне сервера, что снижает нагрузку на сеть со стороны приложений; для представления приложениям уровня доступа к данным; для выполнения административных задач и техобслуживания.
- Хранимые процедуры можно определить с помощью параметров. Входные параметры отправляются хранимой процедуре для внутреннего использования

процедуры. Выходные параметры могут использоваться для возвращения информации вызывающей стороне.

- Внутри хранимой процедуры параметры определяются с помощью такого же синтаксиса, что и переменные T-SQL, и на них можно ссылаться и манипулировать ими внутри процедуры точно так же, как это происходит с переменными.
- Каждая хранимая процедура состоит только из одного пакета кода T-SQL. Хранимые процедуры могут вызывать другие хранимые процедуры.
- Где бы ни выполнялась команда `RETURN`, выполнение хранимой процедуры заканчивается и управление возвращается вызывающей стороне.
- Хранимые процедуры могут возвращать вызывающей стороне более одного результирующего набора.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какие из следующих инструкций T-SQL могут использоваться для создания ветвления в хранимой процедуре? (Выберите все подходящие варианты.)
 - A. `WHILE`.
 - B. `BEGIN/END`.
 - C. `IF/ELSE`.
 - D. `GO`.
2. Хранимая процедура вызывает другую хранимую процедуру. Вызываемая хранимая процедура создала временные таблицы, объявила переменные и передала параметры вызываемой хранимой процедуре. Какие данные вызывающей стороны может видеть вызываемая хранимая процедура?
 - A. Хранимая процедура может видеть переменные, временные таблицы и переданные параметры вызывающей стороны.
 - B. Хранимая процедура может видеть переменные, но не может видеть временные таблицы и переданные параметры вызывающей стороны.
 - C. Вызываемая процедура может видеть переданные параметры и временные таблицы, но не может видеть переменные вызывающей стороны.
 - D. Вызываемая процедура не может видеть никаких объектов, созданных вызывающей процедурой.
3. Как можно использовать выходные параметры в хранимых процедурах T-SQL? (Выберите все подходящие варианты.)
 - A. С помощью выходного параметра можно передать данные в процедуру, но нельзя получить информацию обратно из нее.
 - B. С помощью выходного параметра можно передать данные в процедуру, и любые изменения параметра будут возвращены вызывающей процедуре.

- C. С помощью выходного параметра нельзя передать данные в процедуру; он используется только для передачи данных обратно вызывающей стороне.
- D. С помощью выходного параметра нельзя передать данные в процедуру, а также невозможно получить обратно данные из процедуры из выходного параметра.

Занятие 2. Реализация триггеров



Триггер — это особый тип хранимой процедуры, связанный с выбранными событиями языка обработки данных в таблице или представлении. Триггер не может быть выполнен явно. Он срабатывает, когда запускает событие DML, связанное с этим триггером, такое как инструкции `INSERT`, `UPDATE` или `DELETE`. При возникновении любого такого события триггер запускается и его код выполняется.

SQL Server поддерживает триггеры с двумя типами событий:

- события обработки данных (триггеры DML);
- события описания данных (триггеры DDL), такие как `CREATE TABLE`.

Данное занятие посвящено исключительно триггерам DML.

Изучив материал этого занятия, вы сможете:

- ✓ Создавать и изменять триггеры `AFTER` и `INSTEAD OF` языка T-SQL
- ✓ Описать вставленные и удаленные таблицы, используемые триггерами
- ✓ Описать, как работают вложенные триггеры
- ✓ Использовать функцию `UPDATED` в триггере
- ✓ Обрабатывать несколько строк в триггере
- ✓ Описать влияние, которое могут оказывать триггеры на производительность

Продолжительность занятия — 30 минут.



Триггеры DML

Триггер DML — это пакет T-SQL, связанный с таблицей, которая предназначена для того, чтобы отвечать на определенное событие DML, такое как инструкция `INSERT`, `UPDATE` или `DELETE`, или комбинацию этих событий. SQL Server поддерживает два вида триггеров DML:

- `AFTER` — этот триггер срабатывает после того, как событие, с которым он связан, завершается, и может быть определен только для постоянных таблиц;
- `INSTEAD OF` — этот триггер срабатывает вместо события, с которым он связан, и может быть определен в постоянных таблицах и представлениях.

Триггер выполняется только один раз для каждой инструкции DML, независимо от количества задействованных строк. Схема триггера должна быть той же, что и схема таблицы или представления, с которыми связан триггер.

Можно использовать триггеры DML для аудита, реализации сложных правил целостности и т. п.

Оба типа триггеров DML выполняются как часть транзакции, связанной с инструкцией `INSERT`, `UPDATE` или `DELETE`. Триггер рассматривается как часть транзакции, которая включает событие, вызывающее срабатывание триггера.

Запуск команды `ROLLBACK TRAN` в коде триггера вызывает откат всех изменений, которые имели место в триггере, а также откат исходной инструкции DML, с которой связан триггер. Но использование команды `ROLLBACK TRAN` в триггере может иметь нежелательные побочные эффекты. Вместо нее можно запустить `THROW` или `RAISERROR` и контролировать сбой с помощью стандартных процедур обработки ошибок.

Нормальным выходом из триггера является использование инструкции `RETURN`, также, как и в хранимой процедуре.

В коде T-SQL для обоих типов триггеров DML имеется возможность доступа к таблицам, которые называются *вставленными* и *удаленными*. Эти таблицы содержат строки, подвергшиеся изменениям в результате модификации, вызвавшей срабатывание триггера. Вставленная таблица содержит новый образ подвергнувшихся воздействию строк в случае инструкций `INSERT` и `UPDATE`. Удаленная таблица содержит старый образ обработанных строк в случае инструкций `DELETE` и `UPDATE`. При использовании триггеров `INSTEAD OF` вставленные и удаленные таблицы содержат строки, которые могут быть обработаны инструкцией DML.

Триггеры *AFTER*

Триггеры *AFTER* могут быть определены только для таблиц. В триггере *AFTER* код триггера выполняется после того, как инструкция DML передала все ограничения, такие как ограничения первичного и внешнего ключа, ограничение уникальности или проверочные ограничения. Если ограничение нарушено, инструкция выдает ошибку и триггер не выполняется.

Чтобы посмотреть, как работает триггер *AFTER*, можно начать со вставки фрагмента триггера *AFTER* из SSMS. Откройте новое окно запроса, щелкните правой кнопкой мыши, выберите команду **Insert Snippet** (Вставить фрагмент), затем команду **Create Trigger** (Создать триггер) и нажмите клавишу `<Enter>`. Следующий код будет вставлен в окно запроса:

```
CREATE TRIGGER Sales.tr_SalesOrderDetailsDML
ON Sales.OrderDetails
FOR DELETE, INSERT, UPDATE
AS
BEGIN
    SET NOCOUNT ON
END
```

Прежде всего, убедитесь в том, что это триггер *AFTER*. В определении триггера типом по умолчанию является *AFTER*, если указан элемент `FOR`. Но элемент `FOR` может быть заменен либо *AFTER*, либо `INSTEAD OF` для определения типа триггера.

COBET**Подготовка к экзамену**

Когда выполняются инструкции INSERT, UPDATE или DELETE и не обрабатываются никакие строки, нет смысла в продолжении работы триггера. Можно повысить производительность триггера, выполняя проверку @@ROWCOUNT на равенство 0 в самой первой строке триггера. Это должна быть первая строка, поскольку @@ROWCOUNT будет переустановлена в 0 любой дополнительной инструкцией. Когда триггер AFTER начнет работать, @@ROWCOUNT будет содержать число строк, задействованных внешней инструкцией INSERT, UPDATE или DELETE.

Теперь добавьте проверку существования в функцию OBJECT_ID(), используя 'TR' как тип объекта. Определите его в таблице Sales.OrderDetails базы данных TSQSL2012 и назовите sales.tr_salesOrderDetailsDML.

```
IF OBJECT_ID('Sales.tr_SalesOrderDetailsDML', 'TR') IS NOT NULL
    DROP TRIGGER Sales.tr_SalesOrderDetailsDML;
GO
CREATE TRIGGER Sales.tr_SalesOrderDetailsDML
ON Sales.OrderDetails
AFTER DELETE, INSERT, UPDATE
AS
BEGIN
    IF @@ROWCOUNT = 0 RETURN; -- Должна быть первой инструкцией
    SET NOCOUNT ON;
END;
```

Теперь добавьте инструкцию SELECT для вставленных и удаленных таблиц.

```
IF OBJECT_ID('Sales.tr_SalesOrderDetailsDML', 'TR') IS NOT NULL
    DROP TRIGGER Sales.tr_SalesOrderDetailsDML;
GO
CREATE TRIGGER Sales.tr_SalesOrderDetailsDML
ON Sales.OrderDetails
AFTER DELETE, INSERT, UPDATE
AS
BEGIN
    IF @@ROWCOUNT = 0 RETURN;
    SET NOCOUNT ON;
    SELECT COUNT(*) AS InsertedCount FROM Inserted;
    SELECT COUNT(*) AS DeletedCount FROM Deleted;
END;
```

Главная задача этого триггера — предоставить сведения о том, сколько строк имеется во вставленной и удаленной таблицах. Помните, что вы определили его для инструкций INSERT, UPDATE и DELETE.

COBET**Подготовка к экзамену**

Считается не лучшим решением возвращать результирующие наборы из триггеров. В SQL Server 2012 и более ранних версиях возвращение набора строк из триггера разрешено, но на него не следует полагаться. Эту опцию можно запретить с помощью параметра хранимой процедуры sp_configure, который называется **Disallow Results From Triggers** (Запретить результаты триггеров). Кроме того, возможность возвращать результирующие наборы из триггера является устаревшей и будет удалена в версии SQL Server, следующей за SQL Server 2012.

Теперь измените триггер, чтобы он выполнял некоторую работу. Обратите внимание, таблица Production.Categories не имеет ограничения уникальности или уникального индекса на столбце categoryname. Следующий код добавляет уникальность с помощью триггера AFTER. Вам нужно определить триггер для инструкций INSERT и UPDATE.

```
IF OBJECT_ID('Production.tr_ProductionCategories_categoryname', 'TR')  
    IS NOT NULL  
    DROP TRIGGER Production.tr_ProductionCategories_categoryname;  
GO  
CREATE TRIGGER Production.tr_ProductionCategories_categoryname  
ON Production.Categories  
AFTER INSERT, UPDATE  
AS  
BEGIN  
    IF @@ROWCOUNT = 0 RETURN;  
    SET NOCOUNT ON;  
    IF EXISTS (SELECT COUNT(*)  
        FROM Inserted AS I  
        JOIN Production.Categories AS C  
            ON I.categoryname = C.categoryname  
        GROUP BY I.categoryname  
        HAVING COUNT(*) > 1 )  
        BEGIN  
            THROW 50000, 'Duplicate category names not allowed', 0;  
        END;  
    END;  
GO
```

Теперь протестируйте его с помощью следующей команды INSERT:

```
INSERT INTO Production.Categories (categoryname, description)  
VALUES ('TestCategory1', 'Test1 description v1');
```

Команда INSERT работает только один раз, поскольку иначе это приведет к появлению дубликата строки. Теперь попробуйте выполнить команду UPDATE:

```
UPDATE Production.Categories SET categoryname = 'Beverages'  
WHERE categoryname = 'TestCategory1';
```

Команда UPDATE также завершается ошибкой, поскольку она может привести к появлению дубликата. Выполните следующий код для очистки таблицы:

```
DELETE FROM Production.Categories WHERE categoryname = 'TestCategory1';
```

В этом примере мы проверяем объединение вставленных строк и реальной таблицы, но только *после* того, как выполнена вставка — потому что это триггер AFTER! Поэтому можно использовать обычное объединение, и новые строки из вставленной таблицы будут сопоставляться со строками, только что вставленными в базовую таблицу. Если подсчет уникальных объектов для имени категории больше 1,

вы знаете, что вставили дубликат. Кроме того, глядя на подсчет всех строк, вы определяете случай, когда вставлено или удалено несколько строк.

Вложенные триггеры AFTER

Триггеры AFTER могут быть вложенными, т. е. можно иметь триггер для таблицы A, который обновляет таблицу B. Таблица B, в свою очередь, также может иметь триггер, который выполняется. Максимальная глубина выполнения вложенных триггеров равна 32. Если же вложенность является циклической (триггер таблицы A запускает триггер таблицы B, который запускает триггер таблицы C, который запускает триггер таблицы A и т. д.), максимальный уровень вложенности, равный 32, будет достигнут, и выполнение триггера прекратится.

Вложенные триггеры — это параметр конфигурации для всего экземпляра SQL Server. По умолчанию он включен (ON), но его можно запретить для сервера. Можно проверить эту настройку с помощью хранимой процедуры sp_configure.

```
EXEC sp_configure 'nested triggers';
```

Затем нужно выполнить инструкцию RECONFIGURE для того, чтобы эта настройкаступила в силу. Поскольку он не является дополнительным параметром, не нужно устанавливать параметр конфигурации **Show Advanced Options** в ON с помощью sp_configure.

Триггеры INSTEAD OF

Триггер INSTEAD OF выполняет пакет кода T-SQL вместо инструкций INSERT, UPDATE или DELETE. Инструкция может быть выполнена позже в коде.

Хотя триггеры INSTEAD OF могут создаваться как для таблиц, так и для представлений, обычно они используются с представлениями. Причина в том, что когда инструкция UPDATE отправляется к представлению, может быть обновлена только одна таблица за один раз. Кроме того, в представлении могут быть агрегаты функций на столбцах, не допускающие прямого обновления. Триггер INSTEAD OF может взять инструкцию UPDATE применительно к представлению и, вместо ее выполнения, заменить ее двумя инструкциями UPDATE применительно к базовой таблице представления.

Например, возьмем триггер AFTER из предыдущего раздела и перепишем его как триггер INSTEAD OF. Для простоты определим его только для инструкции INSERT.

```
IF OBJECT_ID('Production.tr_ProductionCategories_categoryname', 'TR')  
    IS NOT NULL  
    DROP TRIGGER Production.tr_ProductionCategories_categoryname;  
GO  
CREATE TRIGGER Production.tr_ProductionCategories_categoryname  
ON Production.Categories  
INSTEAD OF INSERT  
AS
```

```

BEGIN
    SET NOCOUNT ON;
    IF EXISTS (SELECT COUNT(*)
        FROM Inserted AS I
            LEFT JOIN Production.Categories AS C
                ON I.categoryname = C.categoryname
        GROUP BY I.categoryname
            HAVING COUNT(*) > 0 )
    BEGIN
        THROW 50000, 'Duplicate category names not allowed', 0;
    END;
ELSE
    INSERT Production.Categories (categoryname, description)
        SELECT categoryname, description FROM Inserted;
END;
GO
-- Очистка
IF OBJECT_ID('Production.tr_ProductionCategories_categoryname', 'TR')
    IS NOT NULL
    DROP TRIGGER Production.tr_ProductionCategories_categoryname;

```

Функции триггеров DML

Для получения информации о том, что происходит в коде, можно использовать в триггере две функции.

- **UPDATE()**. Эту функцию можно использовать, чтобы определить, имеет ли конкретный столбец ссылки от инструкций `INSERT` или `UPDATE`. Например, можно вставить в триггер следующий код:

```
IF UPDATE(qty) PRINT 'Column qty affected';
```

Следующая инструкция делает выражение `UPDATE(qty)` истинным:

```
UPDATE Sales.OrderDetails
    SET qty = 99 WHERE orderid = 10249
        AND productid = 16;
```

Функция `UPDATE()` возвращает значение "истина", даже если значение столбца указывает на себя в инструкции `UPDATE`. Выполняется только проверка, имеются ли ссылки на столбец.

- **COLUMNS_UPDATED()**. Можно использовать эту функцию, если известен последовательный номер столбца в таблице. Вы должны будете использовать битовую операцию "И" (&), чтобы проверить, был ли столбец обновлен.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие два типа триггеров DML могут быть созданы?
2. Если триггер AFTER обнаруживает ошибку, как он препятствует завершению команды?

Ответы на контрольные вопросы

- Можно создать два типа триггеров DML — AFTER и INSTEAD OF.
- Триггер AFTER запускает команду THROW или RAISERROR, чтобы вызывать откат транзакции DML-команды.

ПРАКТИКУМ Написание триггеров DML

В данном практикуме вы напишете два триггера AFTER. Сначала вы научитесь просматривать содержимое вставленных и удаленных таблиц, а затем будете выполнять бизнес-правило с помощью триггера.

Задание 1. Изучение содержимого вставленных и удаленных таблиц

В этом задании вы будете использовать триггер AFTER для изучения содержимого вставленных и удаленных таблиц, которые видны во время выполнения триггера.

ПРИМЕЧАНИЕ Результатирующие наборы в триггерах устарели

Как уже говорилось ранее, возможность возвращения триггерами результатирующих наборов является устаревшим свойством, которое не будет доступно в последующих версиях SQL Server. Это свойство можно запретить с помощью хранимой процедуры sp_configure, поэтому при выполнении этого задания убедитесь в том, что параметр **Disallow Results From Triggers** (Запретить результаты триггеров) выключен.

- Создайте заново триггер на таблице Sales.OrderDetails следующим образом:

```
USE TSQL2012;
GO
IF OBJECT_ID('Sales.tr_SalesOrderDetailsDML', 'TR') IS NOT NULL
    DROP TRIGGER Sales.tr_SalesOrderDetailsDML; GO
CREATE TRIGGER Sales.tr_SalesOrderDetailsDML
ON Sales.OrderDetails
AFTER DELETE, INSERT, UPDATE
AS
BEGIN
    IF @@ROWCOUNT = 0 RETURN;
    SET NOCOUNT ON;
    SELECT COUNT(*) AS InsertedCount FROM Inserted;
    SELECT COUNT(*) AS DeletedCount FROM Deleted;
END
```

- Убедитесь в том, что можно ввести произвольные выбранные данные. Следующих строк нет в исходной таблице базы данных TSQL2012, поэтому если они присутствуют, удалите их.

```
DELETE FROM Sales.OrderDetails
WHERE orderid = 10249 and productid in (15, 16);
GO
```

3. Теперь добавьте в таблицу данные. Когда вы введете эти две строки, то должны увидеть их во вставленной таблице и не видеть в удаленной таблице. (Если вы выполните следующую инструкцию `INSERT` дважды в строке, то получите нарушение ограничения первичного ключа и не увидите никаких выходных данных триггера, поскольку он не будет выполнен.)

```
INSERT INTO Sales.OrderDetails (orderid, productid, unitprice,
                               qty, discount)
VALUES (10249, 16, 9.00, 1, 0.60),
       (10249, 15, 9.00, 1, 0.40);
```

GO

4. Обновите одну из этих строк. Вы должны увидеть одну строку во вставленной таблице (новые данные) и одну строку в удаленной таблице (старые данные).

```
UPDATE Sales.OrderDetails
   SET unitprice = 99
 WHERE orderid = 10249 AND productid = 16;
GO
```

5. Теперь удалите эти две строки. Вы не должны видеть строк во вставленной таблице и две строки в удаленной таблице.

```
DELETE FROM Sales.OrderDetails
WHERE orderid = 10249 AND productid IN (15, 16);
```

6. Наконец, удалите триггер.

```
IF OBJECT_ID('Sales.tr_SalesOrderDetailsDML', 'TR') IS NOT NULL
    DROP TRIGGER Sales.tr_SalesOrderDetailsDML;
GO
```

Задание 2. Создание триггера *AFTER* для выполнения бизнес-правила

В этом задании вы создадите триггер *AFTER* для выполнения бизнес-правила применительно к таблице `Sales.OrderDetails` в базе данных TSQL2012.

1. Вам нужно написать триггер для реализации следующей логики: любой элемент из таблицы `Sales.OrderDetails` со значением `unitprice` меньше 10 не может иметь скидку более .5. Сначала создайте базовый триггер на таблице `Sales.OrderDetails`, как показано ниже (обратите внимание на то, что переменные используются для получения и тестирования значений `discount` и `unitprice`).

```
USE TSQL2012;
GO
-- Шаг 1: базовый триггер
IF OBJECT_ID('Sales.OrderDetails_AfterTrigger', 'TR') IS NOT NULL
    DROP Trigger Sales.OrderDetails_AfterTrigger;
GO
CREATE TRIGGER Sales.OrderDetails_AfterTrigger ON Sales.OrderDetails
```

```

AFTER INSERT, UPDATE
AS
BEGIN
    IF @@ROWCOUNT = 0 RETURN;
    SET NOCOUNT ON;
    -- Выполнение проверки
    DECLARE @unitprice AS money, @discount AS NUMERIC(4,3);
    SELECT @unitprice = unitprice FROM inserted;
    SELECT @discount = discount FROM inserted;
    IF @unitprice < 10 AND @discount > .5
        BEGIN
            THROW 50000, 'Discount must be <= .5 when unitprice < 10', 0;
        END;
    END;
    GO

```

2. Затем протестируйте триггер на двух строках. Триггер находит строку с нарушением ограничения, в которой значение unitprice равно 9.00 и скидка 0.60.

```

INSERT INTO Sales.OrderDetails (orderid, productid, unitprice,
                                qty, discount)
VALUES (10249, 16, 9.00, 1, 0.60),
       (10249, 15, 9.00, 1, 0.40);

```

3. Теперь попробуйте выполнить ту же вставку с инвертированным порядком строк. На этот раз строка с нарушением ограничения не найдена.

```

INSERT INTO Sales.OrderDetails (orderid, productid, unitprice,
                                qty, discount)
VALUES (10249, 15, 9.00, 1, 0.40),
       (10249, 16, 9.00, 1, 0.60) ;

```

4. Удалите неправильно вставленную строку.

```

DELETE FROM Sales.OrderDetails
WHERE orderid = 10249 AND productid IN (15, 16);
GO

```

5. Обновите триггер, чтобы выполнялось получение и проверка всех строк.

```

IF OBJECT_ID('Sales.OrderDetails_AfterTrigger', 'TR') IS NOT NULL
    DROP Trigger Sales.OrderDetails_AfterTrigger;
GO
CREATE TRIGGER Sales.OrderDetails_AfterTrigger ON Sales.OrderDetails
AFTER INSERT, UPDATE
AS
BEGIN
    IF @@ROWCOUNT = 0 RETURN;
    SET NOCOUNT ON;
    -- Проверка всех строк
    IF EXISTS(SELECT * FROM inserted AS I
              WHERE unitprice < 10 AND discount > .5)

```

```
BEGIN  
    THROW 50000, 'Discount must be <= .5 when unitprice < 10', 0;  
END  
END  
GO
```

6. Снова выполните тот же тест на нескольких строках.

```
INSERT INTO Sales.OrderDetails (orderid, productid, unitprice,  
                                qty, discount)  
VALUES (10249, 15, 9.00, 1, 0.40),  
       (10249, 16, 9.00, 1, 0.60);
```

Теперь триггер должен захватить строку или строки с нарушением ограничения независимо от того, сколько строк вставлено или обновлено.

7. Последний шаг — удалите триггер.

```
IF OBJECT_ID('Sales.OrderDetails_AfterTrigger', 'TR') IS NOT NULL  
    DROP Trigger Sales.OrderDetails_AfterTrigger;  
GO
```

Резюме занятия

- Триггер DML — это пакет кода T-SQL, подобный хранимой процедуре, который связан с таблицей и иногда с представлением. Триггеры DML могут использоваться для аудита, реализации сложных правил целостности и т. п.
- Триггеры выполняются, когда происходит определенное событие DML, такое как инструкция INSERT, UPDATE или DELETE.
- SQL Server поддерживает два типа триггеров DML: триггеры AFTER и триггеры INSTEAD OF. Триггеры DML обоих типов выполняются как часть транзакции, связанной с инструкциями INSERT, UPDATE или DELETE.
- В коде T-SQL для обоих типов триггеров DML можно получить доступ к таблицам, называемым вставленными и удаленными таблицами. Эти таблицы содержат строки, модификация которых приводила к запуску триггера.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Как работают вставленные и удаленные таблицы с инструкцией DML в триггере AFTER?
- А. При использовании инструкции DELETE вставленная таблица содержит новые строки, а удаленная таблица — удаленные строки.

- В. Вставленная таблица содержит только строки из инструкции `INSERT`, а удаленная таблица — только строки из инструкции `DELETE`.
- С. При использовании инструкции `INSERT` вставленная таблица содержит новые строки, а удаленная таблица остается пустой.
- Д. При использовании инструкции `UPDATE` вставленная таблица остается пустой, а удаленная таблица содержит все измененные строки.
2. Какие из перечисленных утверждений справедливы для триггера `INSTEAD OF`? (Назовите все возможные варианты.)
- А. Триггеры `INSTEAD OF` можно открывать на представлениях.
- Б. Триггеры `INSTEAD OF` выполняются вместо триггеров `AFTER`.
- С. Триггеры `INSTEAD OF` могут быть объявлены только для инструкций `UPDATE`.
- Д. Триггеры `INSTEAD OF` выполняют код вместо исходной инструкции DML.
3. Как можно отключить встроенные триггеры на экземпляре SQL Server с помощью T-SQL?
- А. Нужно использовать хранимую процедуру `sp_configure` перед '`nested triggers`' и '`OFF`'.
- Б. Нужно использовать хранимую процедуру `sp_configure` перед '`nested triggers`' и `0`.
- С. Нужно использовать хранимую процедуру `sp_configure` перед '`nested triggers`' и '`OFF`', после которых следует инструкция `RECONFIGURE`.
- Д. Нужно использовать хранимую процедуру `sp_configure` перед '`nested triggers`' и `0`, после которых следует инструкция `RECONFIGURE`.

Занятие 3. Реализация определяемых пользователем функций

Определяемые пользователем (пользовательские) функции — это процедуры T-SQL или CLR, которые могут принимать параметры и возвращать либо скалярные значения, либо таблицы. Данное занятие посвящено определяемым пользователем функциям языка T-SQL. Встроенные системные функции для SQL Server рассмотрены в главе 2.

Изучив материал этого занятия, вы сможете:

- ✓ Создавать и изменять определяемые пользователем функции
- ✓ Описать скалярные и табличные значения
- ✓ Использовать детерминированные и недетерминированные функции

Продолжительность занятия — 20 минут.

Основные сведения об определяемых пользователем функциях

Задача определяемых пользователем функций (user-defined functions, UDF) — инкапсулировать повторно используемый код T-SQL и возвращать вызывающей стороне скалярное значение или таблицу.

Как и хранимые процедуры, определяемые пользователем функции могут принимать параметры, и эти параметры могут быть доступны внутри функции как переменные. В отличие от хранимых процедур, определяемые пользователем функции встроены в инструкции T-SQL и выполняются как часть команды T-SQL. Определяемые пользователем функции не могут выполняться с помощью команды EXECUTE.

Определяемые пользователем функции имеют доступ к данным SQL Server, но не могут выполнять DDL, т. е. они не могут создавать таблицы, а также модифицировать таблицы, индексы или любые другие объекты или изменять любые данные в постоянных таблицах с помощью инструкций DML.

Существуют два основных типа определяемых пользователем функций: скалярные функции и функции с табличным значением. Скалярная функция возвращает вызывающей стороне одно значение, тогда как функция с табличным значением возвращает таблицу. И скалярные функции, и функции с табличным значением могут состоять из одной строки кода T-SQL или из нескольких строк.

Определяемые пользователем функции с табличным значением возвращают таблицу. Функции с табличным значением могут появляться в предложении FROM запроса T-SQL. Определяемая пользователем функция с табличным значением, состоящая из одной строки кода, называется встроенной пользовательской функцией с табличным значением. Определяемая пользователем функция с табличным значением, состоящая из нескольких строк кода, называется многооператорной возвращающей табличное значение определяемой пользователем функцией.

Обратите внимание, при ссылке на тип функции либо из столбца типа представления sys.objects, либо из параметра типа функции OBJECT_ID() используются три сокращенных наименования этих функций:

- FN = скалярная функция SQL;
- IF = встроенная функция с табличным значением SQL;
- TF = функция с табличным значением SQL.

Скалярные определяемые пользователем функции

Определяемые пользователем (пользовательские) скалярные функции называются скалярными, поскольку они возвращают одно значение. Пользовательские скалярные функции могут появляться в любом месте запроса, где может появляться выражение, возвращающее одно значение (например, в списке столбца SELECT). Весь

код внутри пользовательской скалярной функции должен быть заключен в блок BEGIN/END.

В SSMS, если щелкнуть правой кнопкой мыши в окне запроса, выбрать команду **Insert Snippet** (Вставить фрагмент) и вставить фрагмент для скалярной функции, то можно увидеть следующие выходные данные:

```
CREATE FUNCTION dbo.FunctionName
( @param1 int,
  @param2 int)
RETURNS INT
AS
BEGIN
    RETURN @param1 + @param2
END
```

Возьмите эти выходные данные и создайте простую скалярную функцию для вычисления стоимости как цены, умноженной на количество, в таблице Sales. Возьмите столбцы unitprice и qty и возвратите результат их умножения друг на друга. Внесите следующие изменения во фрагмент кода:

1. Присвойте имя sales.fn_extension определяемой пользователем функции.
2. Добавьте в начале условную инструкцию DROP.
3. Добавьте параметры для столбцов unitprice и qty в таблице Sales.OrderDetails.
4. Введите оператор умножения для выполнения вычисления.

```
IF OBJECT_ID('Sales.fn_extension', 'FN') IS NOT NULL
DROP FUNCTION Sales.fn_extension
GO
CREATE FUNCTION Sales.fn_extension
( @unitprice AS MONEY,
  @qty AS INT )
RETURNS MONEY
AS
BEGIN
    RETURN @unitprice * @qty
END;
GO
```

Обратите внимание, можно добавить дополнительные строки в пользовательскую скалярную функцию, просто вставив дополнительные строки между инструкциями BEGIN/END. Для вызова функции просто вызовите ее внутри запроса T-SQL, например в инструкции SELECT. Далее приведен один из примеров, в котором используется функция в списке SELECT.

```
SELECT Orderid, unitprice, qty, Sales.fn_extension(unitprice, qty) AS extension
FROM Sales.OrderDetails;
```

Теперь рассмотрите другой пример, в котором используется функция в предложении WHERE для ограничения выходных данных по стоимости более 1000.

```
SELECT Orderid, unitprice, qty, Sales.fn_extension(unitprice, qty)
      AS extension
FROM Sales.OrderDetails
WHERE Sales.fn_extension(unitprice, qty) > 1000;
```

Определяемые пользователем функции с табличным значением

Пользовательская функция с табличным значением возвращает вызывающей стороне не единственное значение, а таблицу. Поэтому она может быть вызвана в запросе T-SQL там, где ожидается табличный результат, как в предложении `FROM`.

Встроенная функция с табличным значением — это единственный тип определяемой пользователем функции, которая может быть написана без блока `BEGIN/END`.

Многооператорная пользовательская функция с табличным значением имеет инструкцию `RETURN` в конце тела функции.

Встроенная пользовательская функция с табличным значением

Встроенная пользовательская функция с табличным значением содержит одну инструкцию `SELECT`, которая возвращает таблицу. Чтобы посмотреть, как работает встроенная пользовательская функция с табличным значением, вставьте следующий фрагмент SSMS для встроенной функции с табличным значением, которая не может быть выполнена сама по себе.

```
CREATE FUNCTION dbo.FunctionName
( @param1 int,
  @param2 char(5) )
RETURNS TABLE AS RETURN
( SELECT @param1 AS c1,
          @param2 AS c2 )
```

Теперь модифицируйте функцию так, чтобы она возвращала только те строки таблицы `Sales.OrderDetails`, у которых количество находится в промежутке между двумя величинами. Поставьте в начале кода инструкцию `DROP`, присвойте функции имя `sales.fn_filteredextension` и добавьте параметры для нижнего и верхнего значений `qty`.

```
IF OBJECT_ID('Sales.fn_FilteredExtension', 'IF') IS NOT NULL
DROP FUNCTION Sales.fn_FilteredExtension;
GO
CREATE FUNCTION Sales.fn_FilteredExtension
( @lowqty AS SMALLINT,
  @highqty AS SMALLINT )
RETURNS TABLE AS RETURN
( SELECT orderid, unitprice, qty
    FROM Sales.OrderDetails
   WHERE qty BETWEEN @lowqty AND @highqty );
GO
```

Чтобы вызвать функцию, встройте ее в предложение FROM инструкции SELECT, но не забудьте представить требуемые параметры. В следующем примере вы увидите строки, имеющие qty между 10 и 20.

```
SELECT orderid, unitprice, qty
FROM Sales.fn_FilteredExtension (10,20);
```

Обратите внимание, поскольку встроенная функция с табличным значением не выполняет никаких других операций, оптимизатор обрабатывает ее точно так же, как представление. Можно даже применять к ней инструкции INSERT, UPDATE и DELETE, как для представления. По этой причине можно относиться к встроенным функциям с табличным значением как к параметризованным представлениям.

Также обратите внимание, как функция задает таблицу возврата из инструкции SELECT с помощью одной инструкции.

```
RETURNS TABLE AS RETURN
( <SELECT ...> );
```

Именно эта возможность возвратить результаты одной инструкции SELECT делает ее *встроенной* пользовательской функцией с табличным значением.

Многооператорная пользовательская функция с табличным значением

Чтобы построить многооператорную пользовательскую функцию с табличным значением, необходимо изменить синтаксис. Взгляните на следующий фрагмент кода SSMS для пользовательской функции с табличным значением.

```
CREATE FUNCTION dbo.FunctionName
( @param1 int,
  @param2 char(5) )
RETURNS @returntable TABLE
( c1 int,
  c2 char(5) )
AS
BEGIN
  INSERT @returntable
  SELECT @param1, @param2
  RETURN
END;
GO
```

Обратите внимание на различия. В пользовательской функции с табличным значением вы обязаны определить таблицу, которая должна быть возвращена, как табличную переменную и вставить данные в эту табличную переменную. Инструкция RETURN только завершает функцию и не используется для передачи каких-либо данных обратно вызывающей стороне.

Возьмите предыдущую встроенную пользовательскую функцию с табличным значением Sales.fn_FilteredExtension и конвертируйте ее в многооператорную пользовательскую функцию с табличным значением.

```
IF OBJECT_ID('Sales.fn_FilteredExtension2', 'TF') IS NOT NULL
    DROP FUNCTION Sales.fn_FilteredExtension2;
GO
CREATE FUNCTION Sales.fn_FilteredExtension2
( @lowqty AS SMALLINT,
  @highqty AS SMALLINT )
RETURNS @returntable TABLE
( orderid INT,
  unitprice MONEY,
  qty SMALLINT )
AS
BEGIN
    INSERT @returntable
        SELECT orderid, unitprice, qty
        FROM Sales.OrderDetails
        WHERE qty BETWEEN @lowqty AND @highqty
    RETURN
END;
GO
```

Теперь используйте многооператорную пользовательскую функцию с табличным значением Sales.fnFilteredExtension2.

```
SELECT orderid, unitprice, qty
FROM Sales.fn_FilteredExtension2 (10, 20);
```

Ограничения для определяемых пользователем функций

Пользователь, создающий функцию, должен иметь права CREATE FUNCTION в базе данных.

Определяемые пользователем функции не могут выполнять следующие действия:

- применять какие-либо изменения к схеме или данным в базе данных;
- изменять состояние базы данных или экземпляра SQL Server;
- создавать временные таблицы или иметь к ним доступ;
- вызывать хранимые процедуры;
- выполнять динамический SQL;
- производить побочные эффекты. Например, функции RAND() и NEWID() используют информацию из предыдущего вызова. Использование предыдущей информации является "побочным эффектом", что недопустимо.

Аргументы определяемой пользователем функции

Для определяемых пользователем функций можно указать пять аргументов.

- ENCRYPTION. Как и в случаях с хранимыми процедурами и триггерами, это не полное шифрование, а только лишь запутывание исходного кода.

- SCHEMABINDING. Связывает схемы всех объектов, имеющих ссылки на функцию.
- RETURNS NULL ON NULL INPUT. Если этот параметр установлен, любые параметры со значением NULL приводят к тому, что пользовательская скалярная функция возвращает NULL, не выполняя тело функции.
- CALLED ON NULL INPUT. Это настройка по умолчанию, при которой тело скалярной функции будет выполняться, даже если в качестве аргумента передано значение NULL.
- EXECUTE AS. Позволяет выполнять определяемую пользователем функцию в разных контекстах безопасности.

Определяемые пользователем функции также могут быть вложенными. Например, пользовательская функция с табличным значением может в процессе выполнения вызывать пользовательскую скалярную функцию, и конечно, пользовательская скалярная функция может вызывать другую пользовательскую скалярную функцию.

Производительность в контексте определяемой пользователем функции

То, как используется функция, может оказывать решающее влияние на производительность выполняемых запросов. В частности, пользовательские скалярные функции должны иметь высокую эффективность, т. к. они выполняются один раз для каждой строки в результирующем наборе или иногда в целой таблице.

Пользовательская скалярная функция в списке SELECT, если она применяется к значениям столбцов, выполняется для каждой извлеченной строки.

Пользовательская скалярная функция в предложении WHERE, которое ограничивает результирующий набор, выполняется один раз для каждой строки в ссылочной (упоминаемой) таблице.

Использование пользовательских скалярных функций предотвращает распараллеливание запросов.

Контрольные вопросы

1. Назовите два типа определяемых пользователем функций с табличным значением.
2. Какой тип определяемой пользователем функции возвращает всего одно значение?

Ответы на контрольные вопросы

1. Можно создавать встроенные или многооператорные определяемые пользователем функции с табличным значением.
2. Скалярная определяемая пользователем функция возвращает только одно значение.

ПРАКТИКУМ Написание определяемых пользователем функций

В данном практикуме вам предстоит написать две определяемые пользователем функции: скалярную определяемую пользователем функцию истроенную определяемую пользователем функцию с табличным значением.

Задание 1. Создание скалярной определяемой пользователем функции для вычисления дисконтированной стоимости

В этом задании вам нужно написать скалярную пользовательскую функцию, которая определяет стоимость единицы товара после применения скидки в таблице Sales.OrderDetails.

1. Начните с написания запроса для определения стоимости элемента после применения скидки. В таблице Sales.SalesOrder в вычислении участвуют три столбца: unitprice (цена за единицу товара), qty (количество проданных единиц) и discount (часть, на которую уменьшается цена).

```
SELECT orderid, productid, unitprice, qty, discount  
FROM Sales.OrderDetails;
```

2. Произведение двух значений (unitprice и qty) является расширенной стоимостью, т. е. это полная стоимость всех единиц продукции для всего заказа. Добавьте его в запрос.

```
SELECT orderid, productid, unitprice, qty, discount,  
       unitprice * qty AS totalcost  
FROM Sales.OrderDetails
```

3. Скидка — это доля, указывающая величину снижения стоимости. Если вы умножите значение unitprice или totalcost на (1 - discount), то получите стоимость после применения скидки. В этом примере примените скидку к вычисляемому значению totalcost.

```
SELECT orderid, productid, unitprice, qty, discount,  
       unitprice * qty as totalcost,  
       (unitprice * qty) * (1 - discount) as costafterdiscount  
FROM Sales.OrderDetails;
```

4. Теперь у нас достаточно материала для вставки в функцию. Функция требует всего три параметра: @unitprice, @qty и @discount.

```
IF OBJECT_ID('Sales.fn_CostAfterDiscount', 'FN') IS NOT NULL  
    DROP FUNCTION Sales.fn_CostAfterDiscount;  
GO  
CREATE FUNCTION Sales.fn_CostAfterDiscount(  
    @unitprice AS MONEY,  
    @qty AS SMALLINT,  
    @discount AS NUMERIC(4,3)  
) RETURNS MONEY
```

```

AS
BEGIN
    RETURN (@unitprice * @qty) * (1 - @discount);
END;
GO

```

5. Проверьте результат.

```

SELECT Orderid, unitprice, qty, discount,
       Sales.fn_CostAfterDiscount(unitprice, qty, discount)
  AS costafterdiscount
 FROM Sales.OrderDetails;

```

6. Удалите функцию.

```

IF OBJECT_ID('Sales.fn_CostAfterDiscount', 'FN') IS NOT NULL
    DROP FUNCTION Sales.fn_CostAfterDiscount;
GO

```

Задание 2. Создание определяемых пользователем функций с табличным значением

В этом задании вам нужно написать встроенную определяемую пользователем функцию с табличным значением.

- Вы должны написать функцию, которая возвращает таблицу строк из Sales.OrderDetails, отфильтрованных по нижнему и верхнему значениям количества, добавляя столбец для цены. Цена — это просто произведение unitprice * qty. Далее приведена базовая инструкция SELECT без фильтров.

```

SELECT orderid, unitprice, qty, (unitprice * qty) AS extension
FROM Sales.OrderDetails;

```

- Для добавления фильтра можно использовать пару переменных, как показано в следующем коде:

```

DECLARE @lowqty AS SMALLINT = 10,
        @highqty AS SMALLINT = 20;
SELECT orderid, unitprice, qty, (unitprice * qty) AS extension
FROM Sales.OrderDetails WHERE qty BETWEEN @lowqty AND @highqty;

```

- Теперь есть все необходимое для функции. Начните со следующего фрагмента SSMS для встроенной функции с табличным значением:

```

CREATE FUNCTION dbo.FunctionName (
    @param1 int, @param2 char(5)
) RETURNS TABLE AS RETURN (
    SELECT @param1 AS c1, @param2 AS c2
)

```

- Используйте в качестве параметров переменные и присвойте има fn_FilteredExtension. Не забудьте удалить присвоенные значения из переменных, когда будете использовать их, как параметры.

```
IF OBJECT_ID('Sales.fn_FilteredExtension', 'FN') IS NOT NULL
    DROP FUNCTION Sales.fn_FilteredExtension;
GO
CREATE FUNCTION Sales.fn_FilteredExtension (
    @lowqty AS SMALLINT,
    @highqty AS SMALLINT )
RETURNS TABLE AS RETURN (
    SELECT orderid, unitprice, qty, (unitprice * qty) AS extension
        FROM Sales.OrderDetails WHERE qty BETWEEN @lowqty AND @highqty );
GO
```

5. Теперь протестируйте эту функцию.

```
SELECT *
FROM Sales.fn_FilteredExtension (10,20);
```

6. Наконец, удалите функцию.

```
IF OBJECT_ID('Sales.fn_FilteredExtension', 'FN') IS NOT NULL
    DROP FUNCTION Sales.fn_FilteredExtension;
GO
```

Резюме занятия

- Определяемые пользователем функции инкапсулируют повторно используемый код T-SQL и возвращают вызывающей стороне скалярное значение или таблицу.
- Как и хранимые процедуры, определяемые пользователем функции могут принимать параметры, доступ к которым можно получить внутри функции как к переменным. В отличие от хранимых процедур, определяемые пользователем функции встроены в инструкции T-SQL и выполняются как часть команды T-SQL. Определяемые пользователем функции нельзя выполнять с помощью команды EXECUTE.
- Определяемые пользователем функции имеют доступ к данным SQL Server, но не могут выполнять никаких DDL, т. е. они не могут выполнять модификацию таблиц, индексов или других объектов, а также изменять таблицы данных с помощью DML.
- Существуют два основных типа определяемых пользователем функций: скалярные и возвращающие табличное значение. Пользовательская скалярная функция возвращает вызывающей стороне одно значение и может вызываться из множества мест, включая список SELECT и предложение WHERE. Функция, возвращающая табличное значение, возвращает таблицу и может появляться в предложении FROM. И определяемые пользователем скалярные функции, и определяемые пользователем функции с табличным значением могут состоять из одной или нескольких строк кода T-SQL.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Что из перечисленного далее справедливо для скалярной пользовательской функции?
 - A. Пользовательские скалярные функции являются и встроенными, и многооператорными.
 - B. Пользовательские скалярные функции возвращают результат инструкции SELECT.
 - C. Пользовательские скалярные функции могут вызываться в списке SELECT или предложении WHERE.
 - D. Пользовательские скалярные функции могут вызываться в предложении FROM инструкции SELECT.
2. Что из перечисленного далее справедливо для определяемых пользователем функций с табличным значением?
 - A. Определяемые пользователем функции с табличным значением могут возвращать скалярные значения или таблицы.
 - B. Определяемые пользователем функции с табличным значением всегда работают с несколькими инструкциями T-SQL.
 - C. Определяемые пользователем функции с табличным значением могут вызываться в списке SELECT или предложении WHERE.
 - D. Определяемые пользователем функции с табличным значением могут вызываться в предложении FROM инструкции SELECT.
3. Какое из высказываний лучше всего описывает разницу между встроенной пользовательской функцией с табличным значением и многооператорной пользовательской функцией с табличным значением?
 - A. Встроенная пользовательская функция с табличным значением определяет схему табличной переменной с именами столбцов и типами данных и вставляет данные в табличную переменную.
 - B. Встроенная пользовательская функция с табличным значением определяет схему постоянной таблицы с именами столбцов и типами данных и вставляет данные в эту таблицу.
 - C. Многооператорная пользовательская функция с табличным значением определяет схему табличной переменной с именами столбцов и типами данных и вставляет данные в табличную переменную.
 - D. Многооператорная пользовательская функция с табличным значением определяет схему постоянной таблицы с именами столбцов и типами данных и вставляет данные в эту таблицу.

Упражнения

В следующих упражнениях вы примените полученные знания о написании кода хранимых процедур, триггеров и определяемых пользователем функций. Ответы на вопросы можно найти в *приложении "Ответы" в конце книги*.

Упражнение 1. Реализация хранимых процедур и определяемых пользователем функций

Вы назначены на новый проект. Как ведущий разработчик баз данных, вы заметили, что почти вся проверка данных в базе данных происходит в клиентских программах. Иногда неустранимые ошибки в клиентском программном обеспечении вызывают несогласованность базы данных, и вы хотите провести рефакторинг системы с помощью хранимых процедур, чтобы помочь защитить базу данных. Ответьте на следующие вопросы о том, какие действия вы можете предпринять для увеличения надежности приложения.

1. Какие шаги можно предпринять для предотвращения появления дубликатов или несогласованностей уникальных ключей и несоответствующих внешних ключей?
2. Как вы можете представить стандартный интерфейс кода приложения к базе данных?
3. Разработчики клиента хотят поместить параметры в представления, но T-SQL не позволяет им это сделать. Что вы можете использовать вместо параметризованных представлений?
4. Существует одна большая таблица, в которой часто выполняется поиск по трем разным столбцам, но пользователь может выбрать любой из столбцов и оставить остальные пустыми. Как можно использовать хранимые процедуры для увеличения эффективности этого поиска?

Упражнение 2. Реализация триггеров

Вас попросили проверить код T-SQL существующего приложения базы данных и дать рекомендации по его усовершенствованию. Ответьте на следующие вопросы по поводу рекомендаций, которые вы можете дать по данному проекту.

1. Вы заметили, что система использует множество триггеров для обеспечения ограничений внешнего ключа, триггеры подвержены ошибкам и их трудно отлаживать. Какие изменения для уменьшения использования триггеров вы можете порекомендовать?
2. Вы также заметили, что имеются сложные операции, использующие вложенные триггеры, которые невозможно заставить работать в приложении корректно. Какие действия вы можете порекомендовать, чтобы избавиться от использования вложенных триггеров?

3. Приложение должно часто выполнять вставку данных в главную таблицу и в несколько вспомогательных таблиц в одном и том же действии, делая код приложения очень сложным. Что вы можете порекомендовать в качестве способа переместить части сложности из приложения в базу данных?
4. Имеется важная таблица, которая при каждом изменении данных требует простых действий по регистрации этих изменений в журнале. Ведение журнала выполняется в настраиваемой таблице, построенной специально, чтобы удовлетворять требованиям приложения. Какие вы можете дать рекомендации для реализации такого ведения журнала?

Рекомендуемые упражнения

Для успешного освоения экзаменационных тем, представленных в данной главе, выполните следующие задания.

Использование хранимых процедур, триггеров и определяемых пользователем функций

Следующие задания расширяют код, с которым вы работали в занятиях и заданиях этой главы. Продолжайте разработку в базе данных TSQL2012.

- Задание 1.** Добавьте блок TRY/CATCH для обработки ошибок в хранимую процедуру резервного копирования dbo.BackupDatabases, которую вы создавали в *задании 1 занятия 1*.
- Задание 2.** Добавьте блок TRY/CATCH для обработки ошибок в триггер AFTER Sales.OrderDetails_AfterTrigger, который вы создавали в *задании 2 занятия 2*.
- Задание 3.** Модифицируйте встроенную пользовательскую функцию с табличным значением Sales.fn_FilteredExtension, которую вы создали в *задании 2 занятия 3*, чтобы она превратилась в многооператорную пользовательскую функцию с табличным значением.

ГЛАВА 14

Использование инструментов анализа производительности запросов

Темы экзамена

- Устранение неполадок и оптимизация.
 - Оптимизация запросов.

Написание запросов требует базовых знаний T-SQL; для написания высокопроизводительных запросов необходимы значительно более обширные знания. Но Microsoft SQL Server 2012 помогает в этом процессе обучения. SQL Server предоставляет информацию о выполнении запроса, такую как количество дисковых операций ввода-вывода и процессорное время, необходимое для выполнения запроса, подробные шаги выполнения, отсутствующие индексы и индексы, которые не используются, и многое другое. В данной главе представлена информация об использовании информации о выполнении запросов, предоставляемой SQL Server.

ПРЕЖДЕ ВСЕГО

Для полного освоения этой главы необходимо иметь:

- понимание основ реляционных баз данных;
- опыт работы с SQL Server Management Studio (SSMS);
- некоторый опыт в написании кода на языке T-SQL;
- доступ к экземпляру SQL Server 2012 с установленной учебной базой данных TSQL2012.

Занятие 1. Основные понятия оптимизации запросов

Люди не очень любят ждать. Они начинают нервничать в пробке. Они не очень довольны, если им приходится ждать, пока им принесут напитки в баре. Аналогично,

они хотят, чтобы их приложения реагировали на запросы настолько быстро, насколько это возможно. Конечные пользователи ощущают проблемы производительности через ожидание.

SQL Server выполняет огромную работу по оптимизации выполнения запросов. Но оптимизатор запросов SQL Server (SQL Server Query Optimizer) несовершенен. В действительности, как вы скоро увидите сами, нахождение подходящего плана выполнения — исключительно сложный процесс. Вы можете помочь оптимизатору, используя подходящую архитектуру базы данных и приложения, соответствующим образом написанные запросы, применяя правильное индексирование, подсказки выполнения запросов и пр. Чтобы выбрать наилучший режим для повышения производительности ваших запросов, вам следует понимать, как SQL Server их выполняет. SQL Server представляет эту информацию самыми разными способами. В данном занятии вы познакомитесь с основами оптимизации запросов и простыми инструментами, которые способны помочь с оптимизацией, а именно SQL Trace (трассировка SQL), приложением SQL Server Profiler и подсистемой расширенных событий SQL Server (SQL Server Extended Events). С прочими инструментами вы познакомитесь в двух последующих занятиях этой главы.

Изучив материал этого занятия, вы сможете:

- ✓ Понимать проблемы оптимизации запросов
- ✓ Описать оптимизатор запросов SQL Server
- ✓ Использовать инструменты SQL Trace и SQL Server Profiler
- ✓ Использовать подсистему расширенных событий SQL Server

Продолжительность занятия — 40 минут.

Проблемы оптимизации запросов и оптимизатор запросов

За исключением очень простых случаев, запрос может быть выполнен различными способами. Сколько таких способов существует? Количество разных способов выполнения или планов выполнения запросов растет в геометрической прогрессии от сложности запроса. Например, внимательно рассмотрите следующий псевдозапрос. (Разумеется, этот запрос не выполнится, если вы его запустите.)

```
SELECT A.col5, SUM(C.col6) AS col6sum
FROM TableA AS A
INNER JOIN TableB AS B
    ON A.col1 = B.col1 INNER JOIN TableC AS C ON B.col2 = c.col2
WHERE A.col3 = constant1
    AND B.col4 = constant2
GROUP BY A.col5;
```

Начнем с части `FROM`. Какие таблицы SQL Server должен объединять в первую очередь: `TableA` и `TableB` или `TableB` и `TableC`? И в каждом объединении, какая из объединенных должна быть левой, а какая — правой? Существует 6 возможных вари-

антов, если эти два объединения оценивать линейно, одно за другим. При оценке нескольких объединений одновременно число всех возможных комбинаций для обработки объединений уже равно 12. Реальная формула для возможных комбинаций оценивания объединений — $n!$ (или n факториал) для линейной оценки и $(2n - 2)!/(n - 1)!$ для параллельной оценки возможных объединений. Кроме того, SQL Server может выполнять объединение разными способами. Он может использовать любой из следующих алгоритмов объединения:

- вложенные циклы (Nested Loops);
- слияние (Merge);
- хэш (Hash);
- хэш оптимизированной фильтрации по битовым картам (Bitmap Filtering Optimized Hash, также называемый оптимизацией типа "звезда").

Это уже дает 2 варианта для каждого объединения. Таким образом, мы имеем $6 \times 4 = 24$ разных вариантов только для части `FROM` этого запроса. Но реальная ситуация еще хуже. SQL Server может выполнять хэш-соединение тремя разными способами. Как уже упоминалось, это только быстрый поверхностный анализ выполнения псевдозапроса, и для первого знакомства с проблемами оптимизации запросов такие подробности просто не нужны.

В предложении `WHERE` два выражения связаны логическим оператором `AND`, который является коммутативным, поэтому SQL Server может оценить сначала второе выражение. И снова имеются два варианта. Объединив все вместе, получим уже $6 \times 4 \times 2 = 48$ вариантов выбора. И еще раз, реальная ситуация еще хуже. Поскольку в псевдозапросе все соединения являются внутренними и выражения в предложении `WHERE` являются коммутативными, SQL Server может начать выполнение запроса с любого выражения в предложении `WHERE`, затем переключиться на предложение `FROM` и выполнить сначала объединение, оценить второе значение из предложения `WHERE` и т. д. Так что число возможных планов уже значительно выше 48.

В рамках нашего поверхностного знакомства с проблемами производительности, перейдем к предложению `GROUP BY`. SQL Server может выполнить эту часть двумя способами — как упорядоченную группу и как хэш-группу.

Можно остановить анализ вариантов выполнения псевдозапроса. Важно то, что вы можете сделать вывод: число возможных отличающихся планов выполнения запроса растет в геометрической прогрессии от сложности запроса. Можно быстро получить миллионы возможных планов выполнения, SQL Server должен решить, какой из них использовать, за очень короткое время. Вряд ли вы захотите ждать, например, целый день, пока SQL Server найдет наилучший возможный план и затем выполнит ваш запрос за 5 секунд вместо 15. Теперь вы можете вообразить сложность проблем, которые должен решать оптимизатор запросов SQL Server для каждого запроса.

Прежде чем продолжить рассматривать процесс оптимизации запросов, в данном занятии мы кратко опишем, как SQL Server выполняет запросы. Этот процесс показан на рис. 14.1.

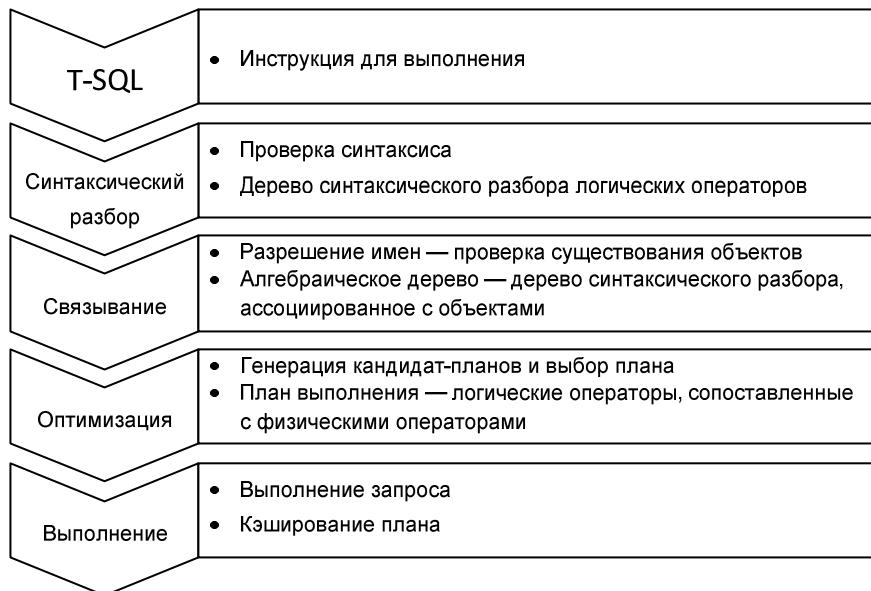


Рис. 14.1. Этапы выполнения запроса

Выполнение запроса начинается с вашего запроса T-SQL. На этапе *синтаксического разбора* SQL Server проверяет запрос на синтаксическую правильность. Результатом этого этапа, при условии прохождения запросом синтаксической проверки, является дерево *логических операторов*, известное как *дерево синтаксического разбора*. На следующем этапе, этапе *привязки*, SQL Server выполняет разрешение имен в запросе. Это означает, что он привязывает объекты к логическим операторам. Разумеется, чтобы этот этап завершился успешно, объекты должны существовать на этом этапе. Результатом этого этапа — *алгебраическое дерево*, представляющее собой дерево логических операторов, связанных с реальными объектами.

Процесс нахождения и оценки разных вариантов (т. е. разных планов) выполнения запроса происходит в течение этапа *оптимизации*. Это этап, на котором оптимизатор запросов выполняет подавляющее большинство работы. На этом этапе SQL Server генерирует кандидат-планы выполнения запросов и оценивает их. Он выбирает наилучший план для следующего этапа. Результатом этого этапа является фактический *план выполнения*, представляющий собой одно дерево с *физическими операторами*.

До сих пор все шаги выполнялись *реляционным модулем*. Реляционный модуль — это внутренний компонент, который работает на логическом уровне. Реальное выполнение осуществляется *подсистемой хранилища*. Разумеется, обе части неделимы и реализованы в одной службе. Подсистема хранилища осуществляет физические операции. Говоря коротко, оптимизатор запросов должен выполнять преобразование из логических в физические операторы. Разумеется, оптимизатор запросов может использовать только физические операторы, которые подсистема хранилища может выполнить. Например, логический оператор может быть

оператором соединения; физический оператор — операцией соединения слиянием (Merge Join).

Результатом конечного этапа, *этапа выполнения*, является желаемый результирующий набор. Кроме того, результатом этапа выполнения также может быть кэшированный план. SQL Server может кэшировать план выполнения для того, чтобы он был готов для последующего выполнения, таким образом избегая повторного выполнения оптимизации. Конечно, SQL Server должен компилировать код в двоичный код перед выполнением; потому кэшируются скомпилированные планы.

Оптимизатор запросов SQL Server основан на *оценке стоимости*. Он присваивает число, называемое стоимостью, каждому возможному плану. Более высокая стоимость означает более сложный план, а более сложный план означает более медленный запрос. Теоретически, SQL Server должен генерировать все возможные планы и затем выбирать план с наименьшей стоимостью. Набор всех возможных планов выполнения называется *областью поиска*. Поскольку количество возможных планов выполнения растет экспоненциально с увеличением сложности запроса, невозможно сгенерировать и проверить все возможные планы для сложных запросов. Оптимизатор запросов ищет баланс между качеством плана и временем, требуемым на оптимизацию. Потому оптимизатор запросов не может гарантировать, что всегда будет выбран наилучший из возможных планов.

SQL Server вычисляет стоимость операции, определяя алгоритм, используемый физическим оператором, и оценивая число строк, которое должно быть обработано. Оценка количества строк также называется *оценкой мощности*. Стоимость выражает использование физических ресурсов, таких как число дисковых операций ввода-вывода, время работы ЦП и объем памяти, требуемый для выполнения. После того как оптимизатор запросов получает стоимость для всех операторов в плане, он может вычислить стоимость всего плана.

Предварительный расчет стоимости может быть довольно непростым. Оптимизатору запросов нужна информация для оценки числа строк, обрабатываемых каждым физическим оператором. Оптимизатор запросов получает эту информацию из статистических данных оптимизатора. SQL Server поддерживает статистику общего количества строк и распределения числа строк по значениям ключа индекса для каждого индекса. Кроме того, SQL Server может генерировать статистику для столбца, даже если это неиндексированный столбец. Также можно сгенерировать и поддерживать статистику вручную. Более подробно об индексах и статистике вы узнаете в главе 15.

Кэширование выбранного плана выполнения в *кэше планов* может ускорить следующее выполнение того же самого запроса или эквивалентного запроса с точки зрения выполнения. SQL Server фактически пытается параметризовать запросы, для того чтобы иметь один план для нескольких эквивалентных запросов. Эквивалентные запросы — это запросы, которые могут быть выполнены тем же способом. Например, два следующих псевдозапроса могут использовать один и тот же план выполнения.

```
SELECT col1 FROM TableA WHERE col2 = 3;  
SELECT col1 FROM TableA WHERE col2 = 5;
```

SQL Server преобразует подобные запросы в параметризованный запрос, такие как следующий псевдозапрос:

```
SELECT col1 FROM TableA WHERE col2 = ?;
```

Конечно, можно также написать собственные параметризованные запросы в хранимых процедурах. Затем можно передать параметры в процессе вызова хранимой процедуры. Кроме того, можно использовать системную хранимую процедуру sys.sp_executesql для параметризации оперативных запросов. Использование хранимых процедур считается наилучшим решением, поскольку автопараметризация имеет множество ограничений, которые не позволяют многим запросам стать параметризованными.

SQL Server кэширует план выполнения отдельно от реального значения, т. е. *контекста выполнения*. Таким образом, SQL Server может повторно использовать тот же план выполнения много раз. Однако использование кэшированных планов не всегда может быть наилучшим решением. Например, число строк в таблице может существенно возрасти. Все планы, включающие сканирование этой таблицы, достаточно быстро для небольшой таблицы, вдруг могут стать неоптимальными.

Планы в кэше также могут устареть, потому что изменяются метаданные в базе данных. Например, в таблицу может быть добавлен индекс или может быть изменено ограничение.

Оптимизатор запросов иногда должен предварительно производить оценку мощности, поскольку он не может с уверенностью определить ее на основании параметров. Эта проблема известна как проблема *перехвата* (снiffeинга) *параметров*. Снiffeинг параметров — это процесс, когда SQL Server пытается перехватить (sniff) значение текущего параметра в процессе компиляции и передает его оптимизатору запросов.

SQL Server проделывает огромную работу по оптимизации запросов и поддерживает кэшированные планы. Но существует множество ситуаций, когда что-то идет не так, как нужно, и выбирается не лучший план, как, например, в следующих сценариях:

- выбран не лучший план, поскольку область поиска планов выполнения была слишком большой;
- статистическая информация не представлена или не обновлена, что ведет к неправильной оценке мощности;
- кэшированный план является неоптимальным для данного значения параметра;
- перехват параметров ведет к неточной оценке мощности;
- оптимизатор запросов недооценивает или переоценивает стоимость алгоритма, реализованного в физическом операторе;
- изменения в аппаратном обеспечении могут больше соответствовать другому плану выполнения. Например, кто-то мог добавить в блок процессоры, и план, который использует большее время центрального процессора (ЦП), может быть более подходящим.

Разумеется, из-за сложности проблемы могут существовать и другие факторы, которые ведут к появлению неоптимального плана. Поэтому всегда есть возможность

оптимизации запросов. Чтобы понять, что произошло не так, необходимо иметь возможность извлечь информацию о расчетных планах и фактически используемых планах выполнения, об использовании индексов, о статистической информации и т. д. SQL Server предоставляет эту информацию разными способами. В оставшейся части этой главы вы узнаете, как извлекать информацию, необходимую для оптимизации запросов.

КОНТРОЛЬНЫЙ ВОПРОС

- Что является результатом этапа синтаксического разбора выполнения запроса?

Ответ на контрольный вопрос

- Результатом этого этапа, при условии, что запрос прошел проверку синтаксиса, является дерево логических операций, известное как дерево синтаксического разбора.

Подсистема расширенных событий SQL Server, трассировка SQL и приложение SQL Server Profiler

Любой мониторинг оказывает влияние на отслеживаемую систему. Если вы следите за системой, уже имеющей проблемы производительности, то можете еще больше замедлить ее работу. Это означает, что необходимо иметь настолько легковесную систему мониторинга, насколько это возможно. Подсистема расширенных событий SQL Server (SQL Server Extended Events) представляет собой очень легкую систему мониторинга производительности. Имея инфраструктуру расширенных событий (Extended Events), вы даже можете коррелировать данные с SQL Server с данными из операционной системы и приложений. Полная система расширенных событий довольно сложна, но поскольку подсистема расширенных событий SQL Server 2012 предоставляет два графических интерфейса пользователя — *мастер создания нового сеанса* (New Session Wizard) и *создание сеанса* (New Session) — можно легко создать сеанс мониторинга и использовать преимущества подсистемы расширенных событий.

Пакет расширенных событий представляет собой контейнер всех объектов подсистемы расширенных событий. К этим объектам относятся следующие.

- **События (Events).** Это точки мониторинга, которые вас интересуют. Можно использовать события для мониторинга или для запуска синхронных или асинхронных действий.
- **Цели (Targets).** Это получатели событий. Можно использовать цели, которые выполняют запись в файл, хранят данные события в буфере памяти или собирают статистические данные о событиях. Цели могут выполнять синхронную или асинхронную обработку данных.
- **Действия (Actions).** Это ответы на события. Они привязаны к событию. Действия могут получать копию стека и проверять данные, сохранять информацию в локальной переменной, собирать статистические данные о событиях и даже до-

бавлять данные к данным события. Например, в SQL Server можно использовать действие обнаружения плана выполнения для нахождения плана выполнения.

- **Предикаты (Predicates).** Это наборы логических правил для фильтрации собранных событий. Для минимизации влияния сессии мониторинга на систему важно захватывать только нужные события.
- **Типы (Types).** Они помогают интерпретировать собранные данные. В действительности, данные — это коллекция байтов, а типы представляют контекст данных. Тип имеют события, действия, цели, предикаты и сами типы.
- **Карты (Maps).** Это внутренние таблицы SQL Server, которые отображают внутренние числовые значения на значимые строки.

Подсистема аудита SQL Server, которая позволяет администратору баз данных (DBA) выполнять облегченный контроль, также основывается на расширенных событиях. Подсистема аудита SQL Server не рассматривается в рамках данной книги. В практикуме к данному занятию мы рассмотрим несколько подробнее подсистему расширенных событий.

Трассировка SQL (SQL Trace) — это встроенный механизм для получения событий. Трассировка SQL будет исключена из последующих версий. Это означает, что она будет по-прежнему доступна в жизненном цикле SQL Server 2012 и следующей версии SQL Server; но в дальнейшем использование приложения SQL Trace может быть прекращено.

Трассировки можно создавать с помощью набора системных хранимых процедур SQL Server. Трассировки можно создавать вручную или через пользовательский интерфейс приложения SQL Server Profiler. Выполняется трассировка *событий* SQL Server. Источником для события трассировки может быть пакет T-SQL или какое-то другое событие SQL Server, такое как блокирование.

После того как событие произошло, трассировка собирает информацию о событии. Информация о событии затем передается очереди. До передачи в очередь события фильтруются в соответствии с установленными *фильтрами*. Из очереди информация о трассировке может передаваться в файл или в таблицу SQL Server, а также использоваться приложениями, такими как SQL Server Profiler.

SQL Server Profiler — это развитое приложение, которое предоставляет пользовательский интерфейс для SQL Trace. С помощью SQL Server Profiler можно создавать трассировки и управлять ими, а также анализировать результаты трассировок. События из сохраненной трассировки можно воспроизводить шаг за шагом. Для запуска трассировки на стороне сервера можно записать созданную в интерфейсе SQL Server Profiler трассировку в сценарий и затем выполнить этот сценарий непосредственно на экземпляре SQL Server.

Использование SQL Server Profiler имеет некоторые недостатки, к которым относятся следующие:

- увеличивается влияние мониторинга на экземпляр SQL Server по сравнению с использованием только трассировки SQL из-за производимых приложением SQL Server Profiler дополнительных затрат;

- при использовании интерфейса SQL Server Profiler на компьютере с установленным экземпляром SQL Server, мониторинг которого выполняется, SQL Server Profiler конкурирует за те же ресурсы;
- при удаленном использовании SQL Server Profiler все события должны проходить по сети, что замедляет другие сетевые операции;
- SQL Server Profiler показывает события в сетке, что может требовать большого количества памяти при получении большого числа событий;
- вы или кто-то еще можете непреднамеренно закрыть профайлер и остановить трассировку, хотя вам нужно получать события в течение длительного времени.

Таким образом, для мониторинга экземпляра SQL Server, находящегося в промышленной эксплуатации, следует использовать трассировку SQL.

Поскольку SQL Server Profiler, так же как и SQL Trace, не будет использоваться в будущих версиях SQL Server, вам следует в ближайшее время перейти на подсистему расширенных событий. Но вы можете продолжать использовать SQL Server Profiler в целях обучения. Например, очень просто запустить трассировку с помощью SQL Server Profiler и проверить, какие команды приложение отправляет на SQL Server. Так вы можете получить информацию о приложении, командах T-SQL и системных процедурах SQL Server.

В следующем списке перечислены термины, используемые как механизмом трассировки SQL, так и приложением SQL Server Profiler.

- *Событие* (event) — это действие на SQL Server. Например, действием может быть отказ во входе в систему, пакет T-SQL, запуск хранимой процедуры и пр.
- *Класс событий* (event class) — это тип события. Класс события определяет данные, которые может сообщить событие.
- *Категория события* (event category) определяет группирование событий.
- *Столбец данных* (data column) — это атрибут события. Если трассировка сохраняется в таблице, событие представляется строкой в таблице, и атрибуты событий являются столбцами этой таблицы.
- *Шаблон* (template) — это сохраненное определение трассировки. SQL Server Profiler поставляется с парой предопределенных шаблонов, которые могут ускорить создание трассировки.
- *Фильтр* (filter). Фильтры ограничивают количество отслеживаемых событий. Можно наложить фильтр на любой столбец события. Для минимизации влияния мониторинга на экземпляр SQL Server необходимо отфильтровать все события, которые не нужны в данной трассировке.
- *Трассировка* (trace) — это коллекция событий, столбцов, фильтров и возвращаемых данных.

COBET

Подготовка к экзамену

Следует использовать подсистему расширенных событий SQL вместо трассировки и приложения SQL Server Profiler, поскольку подсистема расширенных событий является более легкой, а также потому что трассировка SQL и приложение SQL Server Profiler не будут использоваться в следующих версиях SQL Server.

ПРАКТИКУМ Использование подсистемы расширенных событий

В этом практикуме вы будете создавать сеанс расширенных событий. Вам также нужно выполнить инструкцию T-SQL и рассмотреть ее результаты.

Задание 1. Подготовка инструкции T-SQL и создание сеанса расширенных событий

В этом задании вам предстоит подготовить инструкцию T-SQL, которую вы будете анализировать, а также создать и запустить сеанс расширенных событий с помощью мастера.

1. Запустите SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
2. Откройте новое окно запроса, нажав кнопку **New Query** (Создать запрос).
3. Измените контекст на базу данных TSQL2012.
4. Напишите следующий запрос и выполните его, чтобы проверить, работает ли он.

```
SELECT C.custid, C.companyname,
       O.orderid, O.orderdate
  FROM Sales.Customers AS C
    INNER JOIN Sales.Orders AS O
        ON C.custid = O.custid
 ORDER BY C.custid, O.orderid;
```

5. В **Object Explorer** (Обозреватель объектов) раскройте папку **Management** (Управление). Разверните узел **Extended Events** (Расширенные события). Щелкните правой кнопкой мыши на папке **Sessions** (Сеансы) и выберите команду **New Session Wizard** (Мастер новых сеансов).
6. Прочтите информацию на странице **Introduction** (Введение) и нажмите кнопку **Next** (Далее).
7. На странице **Set Session Properties** (Настройка свойств сеанса) присвойте сеансу имя tk461ch14. Нажмите кнопку **Next**.
8. На странице **Choose Template** (Выбор шаблона) установите переключатель **Do not use a template** (Не использовать шаблон). Нажмите кнопку **Next**.
9. На странице **Select Events To Capture** (Выбор событий для отслеживания) введите строку sql в текстовом поле **Event library** (Библиотека событий) для выбора событий, у которых эта строка присутствует в имени. Выберите событие **sql_statement_completed** и переместите его в список **Selected events** (Выбранные события), как показано на рис. 14.2. Нажмите кнопку **Next**.
10. На странице **Capture Global Fields** (Выбор глобальных полей) просмотрите глобальные поля (действия), являющиеся общими для всех событий. Конкретные поля для выбранного события доступны автоматически, поэтому вам не нужно выбирать ни одно из глобальных событий. Нажмите кнопку **Next**.

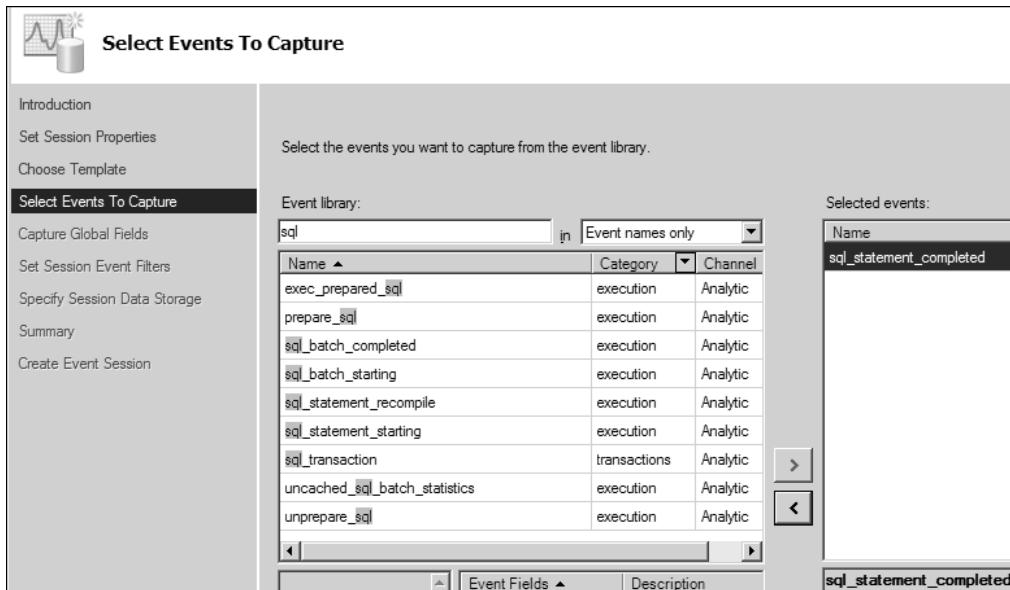


Рис. 14.2. Выбор событий для захвата

11. На странице **Set Session Event Filters** (Установка фильтров событий сеанса) создайте фильтр, который выбирает события с `sqlserver.database_name`, равным значению `TSQL2012`; для оператора поля `sqlserver.sql_text` используйте `like_i_sql_unicode_string` для отбора инструкций, подобных строке `SELECT C.custid, C.companyname%`. Щелкните внутри ячеек столбцов **And/Or**, **Field** и **Operator** в таблице, чтобы выбрать соответствующие значения. Ваш фильтр должен выглядеть аналогично показанному на рис. 14.3. Нажмите кнопку **Next**.

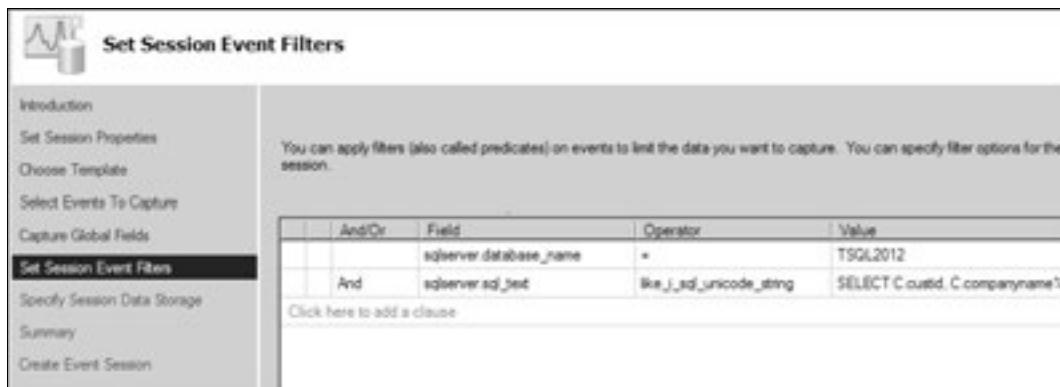


Рис. 14.3. Настройка фильтров сессии события

12. На странице **Specify Session Data Storage** (Указание хранилища данных сеанса) установите флажок **Work with only the most recent data (ring_buffer target)** (Обрабатывать только самые последние данные (целевой объект ring_buffer)). Нажмите кнопку **Next** (Далее).

13. На странице **Summary** (Сводка) проверьте все выбранные данные и нажмите кнопку **Finish** (Готово).
14. На последней странице **Create Event Session** (Создание сеанса событий) установите оба флагка: **Start the event session immediately after session creation** (Запуск сеанса событий непосредственно после создания сеанса) и **Watch live data on the screen as it is captured** (Интерактивный просмотр захваченных данных на экране). Нажмите кнопку **Close** (Закрыть). Окно **Extended Events Live Data** должно открыться в SSMS.

Задание 2. Использование сеанса расширенных событий

В этом задании вы будете использовать сеанс расширенных событий, подготовленный в предыдущем задании. Вы также напишете сценарий сеанса, чтобы понять, как его создать на языке T-SQL.

1. Переключитесь в окно запроса, в котором находится ваш запрос.
2. Выполните запрос.
3. Переключитесь обратно в окно **Extended Events Live Data** и проверьте результаты. Обратите внимание на собранные поля.
4. Закройте окно **Extended Events Live Data**.
5. В **Object Explorer** раскройте папку **Sessions** (Сеансы) расширенных событий. Кроме ваших сеансов, уже определены сеансы по умолчанию.
6. Щелкните правой кнопкой мыши на своем сеансе TK461Ch14 и создайте его сценарий как DDL-инструкцию CREATE в новом окне запроса. Проверьте инструкцию CREATE EVENT SESSION.
7. Закройте окно запроса с инструкцией CREATE EVENT SESSION. Чтобы очистить данные, щелкните правой кнопкой мыши на своем сеансе TK461Ch14 и выберите команду **Delete** (Удалить).

Резюме занятия

- Оптимизатор запросов генерирует кандидат-планы выполнения и оценивает их.
- SQL Server предоставляет множество инструментов, которые помогают пользователям анализировать запросы, включая подсистему расширенных событий, трассировку SQL и приложение SQL Server Profiler.
- Подсистема расширенных событий является более легким механизмом мониторинга, чем трассировка SQL.
- SQL Server Profiler предоставляет пользовательский графический интерфейс для доступа к трассировке SQL.

Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в приложении "Ответы" в конце книги.

1. Какие действия относятся к этапу оптимизации выполнения запроса? (Выберите все подходящие варианты.)
 - A. Генерация алгебраического дерева.
 - B. Генерация кандидат-планов.
 - C. Выбор лучшего кандидат-плана.
 - D. Кэширование плана.
 - E. Выполнение запроса.
2. На каком этапе выполнения запроса SQL Server проверяет, существуют ли объекты, на которые ссылается запрос?
 - A. На этапе грамматического разбора.
 - B. На этапе привязки.
 - C. На этапе оптимизации.
 - D. На этапе выполнения.
3. Что из перечисленного не является частью пакета расширенных событий?
 - A. Предикаты.
 - B. Цели.
 - C. Источники.
 - D. Действия.

Занятие 2. Использование параметров сеанса инструкции *SET* и анализ планов запросов

Как вы уже поняли из предыдущего занятия, оптимизация запросов — это сложный процесс. Производительность запросов можно повысить с помощью правильного построения запросов, соответствующих индексов, обновленной статистики, подсказок к запросам и пр. SQL Server предоставляет информацию о своей деятельности различными способами. В предыдущем занятии вы познакомились с инструментами, которые можно использовать для трассировки деятельности во времени, такими как подсистема расширенных событий, трассировка SQL и SQL Server Profiler. В данном занятии вы узнаете, как выполнять подробный анализ одного запроса с помощью параметров сеанса *SET* и предоставленных планов выполнения.

Изучив материал этого занятия, вы сможете:

- ✓ Использовать параметры сеанса инструкции *SET* для анализа запросов
- ✓ Читать расчетные и реальные планы выполнения

Продолжительность занятия — 40 минут.



Параметры сеанса инструкции **SET**

SQL Server хранит данные на *страницах*. Страница — это физическая единица на диске внутри базы данных SQL Server. Страница имеет фиксированный размер, равный 8192 байт, или 8 Кбайт. Страница принадлежит только одному объекту, т. е. одной таблице, индексу или индексированному представлению. Далее страницы группируются в логические группы по 8 страниц, называемые *экстентами*. Экстент может быть *смешанным*, если страницы в этом экстенте принадлежат нескольким объектам, или *однородным*, когда все страницы этого экстента принадлежат только одному объекту.

Более подробно о содержимом страницы в таблице или индексе вы узнаете в *главе 15*. В данной главе достаточно помнить, что одна из целей оптимизации запроса — уменьшить число дисковых операций ввода-вывода. Это означает, что нужно уменьшить число страниц, которые должен читать SQL Server.

Вы можете получить информацию о числе страниц на таблицу, к которым получают доступ запросы, установив параметр возвращения статистических данных о вводе-выводе. Это можно сделать на уровне сеанса с помощью команды `SET STATISTICS IO` языка T-SQL. Параметр уровня сеанса означает, что этот параметр остается неизменным на протяжении всего сеанса, пока вы не отключитесь от SQL Server, если вы не выключите этот параметр с помощью той же инструкции.

Следующий код проверяет число страниц, которые таблицы `Sales.Customers` и `Sales.Orders` занимают в базе данных TSQL2012.

```
DBCC DROPCLEANBUFFERS;
SET STATISTICS IO ON;
SELECT * FROM Sales.Customers;
SELECT * FROM Sales.Orders;
```

Обратите внимание, инструкция `DBCC DROPCLEANBUFFERS` удаляет данные из кэша. SQL Server кэширует не только данные планов запросов и процедур. Чтобы показать статистику ввода-вывода, лучше не иметь данных в кэше. Но следует соблюдать исключительную осторожность при очистке кэша на рабочем сервере. SQL Server кэширует данные для ускорения выполнения запросов; поскольку часть данных кэшируется, когда они понадобятся в следующий раз, SQL Server может извлечь их из памяти, а не с диска, таким образом выполнив запрос, которому эти данные нужны, значительно быстрее. Далее приведены результаты запроса.

```
(91 row(s) affected)
Table 'Customers'. Scan count 1, logical reads 5, physical reads 1, read-ahead reads 10, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
(830 row(s) affected)
Table 'Orders'. Scan count 1, logical reads 21, physical reads 1, read-ahead reads 19, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

Далее приведены обозначения, используемые в возвращенной запросом информации:

- `scan count` (число просмотров) — количество выполненных просмотров таблицы или индекса;

- logical reads (логических чтений) — количество страниц, считанных из кэша данных. Когда выполняется чтение целой таблицы, как в запросах из приведенного примера, это число дает оценку размера таблицы;
- physical reads (физических чтений) — количество страниц, считанных с диска. Это число меньше, чем фактическое количество страниц, поскольку множество страниц находится в кэше;
- read-ahead reads (упреждающих чтений) — количество страниц, прочитанных SQL Server с упреждением;
- lob logical reads (LOB логических чтений) — число страниц больших объектов (large object page, LOB), считанных из кэша данных. К большим объектам относятся столбцы типов VARCHAR(MAX), NVARCHAR(MAX), VARBINARY(MAX), TEXT, NTEXT, IMAGE, XML или большие типы данных CLR, включая системные пространственные CLR-типы GEOMETRY и GEOGRAPHY;
- lob physical reads (LOB физических чтений) — число страниц типов больших объектов, считанных с диска;
- lob read-ahead reads (LOB упреждающих чтений) — число страниц типов больших объектов, прочитанных SQL Server с упреждением.

Для начала вы можете ориентироваться на число логических чтений. Логические чтения дадут первую приближенную оценку эффективности запроса. Но не следует использовать эту информацию без дополнительных сведений.

Посмотрите на следующие два запроса:

```
SELECT C.custid, C.companyname,
       O.orderid, O.orderdate
  FROM Sales.Customers AS C
    INNER JOIN Sales.Orders AS O
      ON C.custid = O.custid;
```

```
SELECT C.custid, C.companyname,
       O.orderid, O.orderdate
  FROM Sales.Customers AS C
    INNER JOIN Sales.Orders AS O
      ON C.custid = O.custid
 WHERE O.custid < 5;
```

Статистика ввода-вывода для первого запроса показывает только два логических чтения для таблицы Sales.Customers, тогда как для второго запроса показано 60 логических чтений. Однако второй запрос фильтрует строки для клиентов, у которых значение custid меньше 5, поэтому он должен выполнить меньше операций ввода-вывода, верно? Но SQL Server считает каждое обращение к таблице, даже если нужные таблицы уже помещены в кэш. Но когда таблицы находятся в кэше, обращение к ним не требует больших затрат. Вот краткое объяснение полученных чисел: SQL Server использовал во втором запросе алгоритм соединения на основе вложенных циклов, при этом Sales.Customers была в этом соединении вложенной

таблицей. Более подробно о соединениях можно прочесть в главе 17. Пока лишь достаточно знать, что важно не полагаться только на статистику ввода-вывода при анализе производительности запроса. К счастью, есть множество дополнительных возможностей для этого.

Следующая полезная для анализа производительности команда SET уровня сеанса — это команда SET STATISTICS TIME. Следующий код устанавливает этот параметр. Обратите внимание на команду DBCC DROPCLEANBUFFERS. Она удаляет кэшированные планы из кэша. Еще раз напомним, следует соблюдать большую осторожность при использовании этой команды в рабочей среде.

```
SET STATISTICS TIME ON;  
DBCC DROPCLEANBUFFERS;
```

Затем следующий код выполняет первый запрос из предыдущего примера.

```
SELECT C.custid, C.companyname,  
       O.orderid, O.orderdate  
  FROM Sales.Customers AS C  
 INNER JOIN Sales.Orders AS O  
    ON C.custid = O.custid;
```

Результаты могут немного отличаться от выполнения к выполнению запроса. Например, результаты в процессе одного из тестовых выполнений были следующими:

```
SQL Server parse and compile time:  
   CPU time = 0 ms, elapsed time = 0 ms.  
SQL Server Execution Times:  
   CPU time = 0 ms, elapsed time = 92 ms.  
(Время синтаксического анализа и компиляции SQL Server:  
 время ЦП = 0 мс, истекшее время = 0 мс.  
Время работы SQL Server:  
 Время ЦП = 0 мс, затраченное время = 92 мс.)
```

Вы видите, что возвращенная статистика по времени содержит время ЦП (CPU) и общее (затраченное) время, необходимое для выполнения операции. Кроме того, вы можете видеть действительное время выполнения и время, необходимое для предшествующих выполнению этапов, включая грамматический разбор, связывание, оптимизацию и компиляцию. Теперь нужно снова удалить помещенные в кэш планы.

```
DBCC DROPCLEANBUFFERS;
```

Если вы выполните второй запрос из примера, то получите другие результаты.

```
DBCC DROPCLEANBUFFERS;  
SELECT C.custid, C.companyname,  
       O.orderid, O.orderdate  
  FROM Sales.Customers AS C  
 INNER JOIN Sales.Orders AS O  
    ON C.custid = O.custid  
 WHERE O.custid < 5;
```

Вот как выглядят результаты:

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 1 ms.

SQL Server Execution Times:

CPU time = 15 ms, elapsed time = 10 ms.

(Время работы SQL Server:

Время ЦП = 0 мс, затраченное время = 1 мс.

Время работы SQL Server:

Время ЦП = 15 мс, затраченное время = 10 мс.)

Теперь вы видите, что второй запрос работал быстрее. Но есть еще более совершенные инструменты для изучения того, как SQL Server выполняет запросы.

Планы выполнения

Можно получить исчерпывающую информацию о том, как выполняется запрос, анализируя его план выполнения. SQL Server предоставляет *предполагаемые* (расчетные) и *фактические* планы выполнения. Если отображается только предполагаемый план выполнения, запрос не выполняется. Фактический и предполагаемый планы обычно не отличаются друг от друга; но в некоторых случаях расчетный план не может дать такую полную и точную информацию, какую может представить фактический план. Например, если вы создаете и запрашиваете временную таблицу в одном и том же пакете, SQL Server не может оптимизировать доступ к данным временной таблицы, поскольку на момент оптимизации она еще не создана. Кроме того, SQL Server откладывает оптимизацию динамического SQL, потому что неясно, что выполнять и как оптимизировать этот динамический SQL до времени выполнения. Однако в случае больших запросов, выполняемых продолжительное время, расчетные планы могут быть весьма полезными. Вряд ли вы захотите выполнять запрос, который осуществляет чтение миллиарда строк, на рабочей системе только для того, чтобы получить фактический план выполнения.

И расчетный, и фактический план можно получить в трех форматах: текстовом, XML и графическом. SQL Server изначально возвращает план выполнения в формате XML. SSMS представляет этот XML-формат в графическом виде. Текстовое представление планов выполнения является устаревшим и будет удалено в следующих версиях SQL Server. Для включения и отключения текстового формата планов выполнения можно использовать следующие команды SET языка T-SQL уровня сеанса:

- SET SHOWPLAN_TEXT и SET SHOWPLAN_ALL для расчетных планов;
- SET STATISTICS PROFILE для фактических планов.

Включить и отключить XML-планы можно с помощью следующих команд:

- SET SHOWPLAN_XML для расчетных планов;
- SET STATISTICS XML для фактических планов.

Можно включать и отключать расчетные и фактические планы в среде SSMS следующими способами.

- Щелкните правой кнопкой мыши в окне редактора запросов и выберите либо команду **Display Estimated Execution Plan** (Показать предполагаемый план выполнения), либо команду **Include Actual Execution Plan** (Включить действительный план выполнения).
- Нажмите комбинацию клавиш <Ctrl>+<L> для настройки параметра **Display Estimated Execution Plan** (Показать предполагаемый план выполнения) или <Ctrl>+<M> для выбора параметра **Include Actual Execution Plan** (Включить действительный план выполнения).
- Выберите любой из этих параметров в меню **Query** (Запрос).
- Нажмите кнопку **Display Estimated Execution Plan** (Показать предполагаемый план выполнения) или **Include Actual Execution Plan** (Включить действительный план выполнения) на панели инструментов редактора SQL (SQL Editor).

Далее приведен пример запроса.

```
SELECT C.custid, MIN(C.companyname) AS companyname,
       COUNT(*) AS numorders
  FROM Sales.Customers AS C
 INNER JOIN Sales.Orders AS O
    ON C.custid = O.custid
 WHERE O.custid < 5
 GROUP BY C.custid
 HAVING COUNT(*) > 6;
```

Этот запрос создает действительный план выполнения (рис. 14.4).



Рис. 14.4. План выполнения запроса в графическом виде

На плане выполнения можно увидеть физические операторы, используемые в процессе выполнения. План выполнения читается справа налево и сверху вниз. SQL Server начал выполнение этого запроса с поиска кластеризованного индекса в таблице `Sales.Customers` и некластеризованного индекса в таблице `Sales.Orders`. Затем SQL Server объединил результаты предыдущих операций и объединение с использованием вложенных циклов и т. д. Также вы можете увидеть относительную стоимость каждого оператора в общей стоимости запроса, выраженную в процентном

отношении к общим затратам на выполнение запроса. В данном примере первые два оператора имеют 99% (46 + 53) от общей стоимости запроса. Вы можете легко найти наиболее "дорогие" операторы.

Стрелки указывают на поток данных от одного физического оператора к другому. Толщина стрелки соответствует относительному количеству строк, переданных от оператора к оператору. Задержав указатель мыши на операторе или стрелке, вы можете получить значительно больше дополнительной информации. На рис. 14.5 показана подробная информация для оператора вложенных циклов.

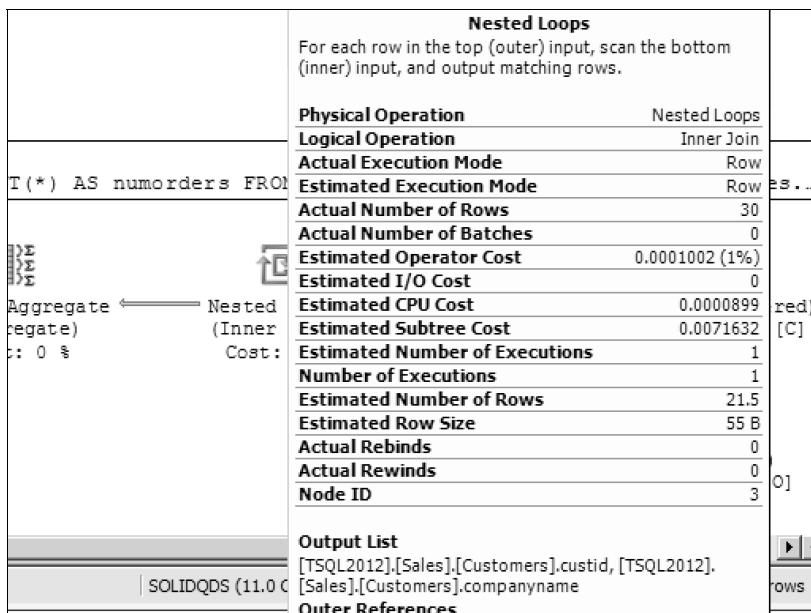


Рис. 14.5. Подробная информация об операторе вложенных циклов

В этих подробных сведениях вы видите используемые физические и логические операторы. Вы видите предполагаемое количество строк (в нашем примере 21.5) и фактическое количество строк (30). Таким образом, вы можете быстро заметить ошибки в оценке мощности. Вы можете увидеть здесь предполагаемую стоимость оператора, предполагаемую стоимость поддерева вплоть до этой точки. Стоимость оператора используется для вычисления процента от полной стоимости. Вы можете получить полную предполагаемую стоимость запроса, если задержите указатель мыши на последнем операторе в плане и прочтете предполагаемую стоимость поддерева. Можно получить еще более подробную информацию о каждом операторе, если открыть окно **Properties** (Свойства) из меню **View** (Вид) или нажать клавишу <F4>.

Обратите внимание, план выполнения изначально представлен в формате XML, за его графическое представление отвечает среда SSMS. Вы можете щелкнуть правой кнопкой мыши на графическом плане и просмотреть его в формате XML, выбрав команду **Show Execution Plan XML** (Показать XML-код плана выполнения...).

Можно сохранить план для последующего анализа с помощью параметра **Save Execution Plan As** (Сохранить план выполнения как...). План выполнения сохраняется в формате XML в файле с расширением sqlplan.

Информацию обо всех допустимых операторах, которые могут появиться в плане выполнения, можно найти в электронной документации по SQL Server 2008 R2 в разделе "Значки графических планов выполнения (среда SQL Server Management Studio)" по адресу

[http://msdn.microsoft.com/ru-ru/library/ms175913\(v=SQL.105\).aspx](http://msdn.microsoft.com/ru-ru/library/ms175913(v=SQL.105).aspx).

В табл. 14.1 показаны некоторые из наиболее популярных операторов, включая их значки на графическом плане, их имена и краткое описание для каждого.

Таблица 14.1. Часто используемые операторы плана выполнения

Значок	Оператор	Описание
	Table Scan (Просмотр строк таблицы)	Просматривает целую таблицу, сохраненную в виде кучи. Таблица может быть в виде кучи или кластеризованного индекса
	Clustered Index Scan	Сканирует целую таблицу, сохраненную как кластеризованный индекс. Индексы сохраняются как сбалансированные деревья
	Clustered Index Seek	SQL Server выполняет поиск первого значения в аргументе поиска (например, значение столбца в предложении WHERE) в некластеризованном индексе и затем выполняет частичное сканирование
	Index Scan (Просмотр индекса columnstore)	Сканирование целого некластеризованного индекса
	Index Seek (Поиск в индексе)	SQL Server выполняет поиск первого значения в аргументе поиска (например, значение столбца в предложении WHERE) в некластеризованном индексе и затем выполняет частичное сканирование
	RID Lookup (Уточняющий запрос RID)	Выполняет поиск одной строки в таблице, сохраненной как куча, используя идентификатор этой строки (RID)
	Key Lookup (Поиск ключа)	Выполняет поиск одной строки в таблице, сохраненной как кластеризованный индекс, используя ключ индекса
	Hash Match Join	Соединения, которые используют хэш-алгоритм
	Merge Join	Соединения, которые используют алгоритм слияния
	Nested Loops	Соединения, которые используют алгоритм вложенных циклов
	Stream Aggregate	Группирование упорядоченных строк
	Hash Match Aggregate	Хэш-алгоритм, используемый для статистических вычислений. Обратите внимание, значок такой же, как у оператора Hash Match Join; но в плане выполнения текст под значком дает информацию о том, выполняя оператор соединение или статистическое вычисление

Таблица 14.1 (окончание)

Значок	Оператор	Описание
	Filter (Фильтр)	Фильтрует строки на основании предиката (например, предикат предложения WHERE)
	Sort (Сортировать)	Сортирует входящие строки

Предпочтительно видеть одни из этих операторов в среде транзакций, другие — в среде хранилищ данных. Ни один из этих операторов не может считаться хорошим или плохим. Вы лучше поймете, когда какой оператор является предпочтительным, прочитав главы 15 и 17. Дополнительную информацию о кучах и кластеризованных и некластеризованных индексах см. в главе 15.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как вы можете быстро измерить количество дисковых операций ввода-вывода, которые выполняет запрос?
2. Как вы можете получить предполагаемый план выполнения в формате XML для дальнейшего анализа?

Ответы на контрольные вопросы

1. Вам следует использовать команду `SET STATISTICS IO`.
2. Вы можете использовать команду `SET SHOWPLAN_XML`.

ПРАКТИКУМ Использование параметров `SET` на уровне сеанса и планов выполнения

В данном практикуме вы будете использовать графический план выполнения и параметры уровня сеанса инструкции `SET`. Вам также нужно рассмотреть дополнительный параметр плана выполнения — предупреждение об отсутствующих индексах.

Задание 1. Подготовка данных

В этом задании вам нужно создать новую таблицу с данными и проиндексировать ее.

1. Если вы закрыли SQL Server Management Studio (SSMS), откройте ее и подключитесь к вашему экземпляру SQL Server.
2. Откройте новое окно запроса, нажав кнопку **New Query** (Создать запрос).
3. Измените контекст на базу данных TSQL2012.
4. Напишите следующий запрос для создания новой таблицы из таблицы `Sales.Orders` и умножьте количество строк исходной таблицы 30 раз.

```
SELECT N1.n * 100000 + O.orderid AS norderid, O.*  
INTO dbo.NewOrders  
FROM Sales.Orders AS O  
CROSS JOIN (VALUES(1), (2), (3), (4), (5), (6), (7), (8), (9),  
(10), (11), (12), (13), (14), (15), (16),  
(17), (18), (19), (20), (21), (22), (23),  
(24), (25), (26), (27), (28), (29), (30)) AS N1(n);
```

5. Создайте на новой таблице некластеризованный индекс с именем `idx_nc_orderid`. Проиндексируйте столбец `orderid`. Используйте следующий код:

```
CREATE NONCLUSTERED INDEX idx_nc_orderid  
ON dbo.NewOrders(orderid);
```

Задание 2. Проанализируйте запрос

В этом задании вам нужно проанализировать запрос, который запрашивает созданную вами в предыдущем задании таблицу.

1. Включите статистику ввода-вывода и статистику по времени.

```
SET STATISTICS IO ON;  
SET STATISTICS TIME ON;
```

2. Выполните следующий запрос:

```
SELECT norderid  
FROM dbo.NewOrders  
WHERE norderid = 110248  
ORDER BY norderid;
```

3. Обратите внимание, запрос использует только столбец `norderid`, тогда как индекс, который вы создали, содержал лишь столбец `orderid`. Проверьте время ЦП и истекшее время, требуемое для грамматического разбора, компиляции и выполнения. Проверьте количество логических чтений. Запомните эти числа и затем выключите статистику ввода-вывода и статистику по времени.

```
SET STATISTICS IO OFF;  
SET STATISTICS TIME OFF;
```

4. Включите действительный план выполнения, когда будете выполнять запрос, нажав кнопку **Include Actual Execution Plan** (Включить действительный план выполнения). Снова выполните тот же запрос.

```
SELECT norderid  
FROM dbo.NewOrders  
WHERE norderid = 110248  
ORDER BY norderid;
```

5. В окне запроса среды SSMS, в панели результатов, щелкните на вкладке **Execution plan** (План выполнения). Проанализируйте план выполнения. Он должен выглядеть подобно изображенному на рис. 14.6.

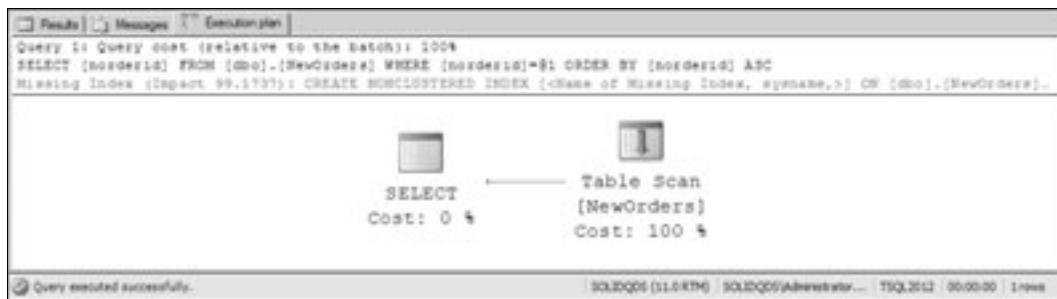


Рис. 14.6. План выполнения с предупреждением об отсутствующих индексах

6. Просмотрите предупреждение об отсутствующих индексах. Щелкните правой кнопкой мыши и выберите команду **Missing Index Details** (Сведения об отсутствующих индексах), чтобы записать инструкцию создания индекса в сценарий.
 7. Создайте отсутствующий индекс с помощью следующего кода:
- ```
CREATE NONCLUSTERED INDEX idx_nc_norderid
 ON dbo.NewOrders(norderid);
```
8. Снова выполните запрос. Изучите новый план выполнения. При желании вы можете также проверить логический ввод-вывод и истекшее время.
  9. Отключите действительный план выполнения.
  10. Очистите базу данных, используя следующий код:

```
DROP TABLE dbo.NewOrders;
```

## Резюме занятия

- Можно использовать параметры инструкции SET уровня сеанса для анализа запросов.
- Можно использовать графические планы выполнения для получения подробных сведений о том, как SQL Server выполняет запрос.
- Можно получить отображение предварительного и действительного плана выполнения.
- В графическом плане выполнения можно найти подробные свойства каждого оператора.

## Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какие параметры инструкции SET уровня сеанса полезны для оптимизации запросов? (Выберите все подходящие ответы.)

- A. SET STATISTICS IO.
  - B. SET STATISTICS EXECUTION\_DETAILS.
  - C. SET IDENTITY\_INSERT.
  - D. SET STATISTICS.
2. Как можно прочесть графический план выполнения?
- A. Сверху вниз, слева направо.
  - B. Сверху вниз, справа налево.
  - C. Слева направо, сверху вниз.
  - D. Справа налево, сверху вниз.
3. Какие команды включают XML-план?
- A. SET EXECUTION\_XML ON.
  - B. SET SHOWPLAN\_XML ON.
  - C. SET XML PLAN ON.
  - D. SET STATISTICS XML ON.

## Занятие 3. Использование динамических административных объектов

Даже расширенные события, трассировка SQL, приложение SQL Server Profiler, параметры инструкции SET уровня сеанса и планы выполнения из набора инструментов SQL Server не являются исчерпывающими возможностями оптимизации запросов. SQL Server постоянно контролирует себя и собирает информацию, полезную для мониторинга состояния экземпляра, находит проблемы, такие как отсутствующие индексы, и оптимизирует запросы. SQL Server предоставляет эту информацию посредством *динамических административных объектов* (dynamic management objects, DMO). К этим объектам относятся *динамические административные представления* и *функции динамического управления*. Функции, которые отличаются от представлений, принимают параметры. Все динамические административные объекты принадлежат системной схеме sys; имена динамических административных объектов начинаются со строки dm\_. Часть информации из динамических административных объектов показывает текущее состояние экземпляра сервера, тогда как другая информация накапливается с момента запуска экземпляра сервера.



**Изучив материал этого занятия, вы сможете:**

- ✓ Получить общее представление о динамических административных объектах
- ✓ Использовать динамические административные объекты для настройки запросов

**Продолжительность занятия — 35 минут.**

## **Введение в динамические административные объекты**

Представьте, что конечный пользователь обратился с жалобой на производительность SQL Server. Вы должны немедленно приступить к изучению проблемы. С чего вы начнете? В рабочей системе конечные пользователи могут отправлять тысячи запросов в час. Какой из запросов вам нужно проанализировать в первую очередь? Можно начать с сеанса мониторинга с помощью расширенных событий. Вы можете использовать трассировку SQL. В обоих случаях вы будете должны ждать длительное время, прежде чем соберете достаточно данных, необходимых, чтобы начать анализ и найти самые проблематичные запросы. А что, если эти проблемные запросы не будут выполняться снова вскоре после начала вашей сессии мониторинга? Вы можете только надеяться, что сможете обнаружить проблемы через какой-то разумный промежуток времени.

Это как раз тот случай, когда динамические административные объекты становятся исключительно полезными. С помощью динамических административных объектов большое количество нужной вам информации уже собрано. Все, что вам нужно сделать, — это запросить соответствующие динамические административные объекты с помощью стандартных запросов T-SQL и извлечь полезную информацию. Динамические административные объекты не сохраняются ни в какой базе данных; динамические административные объекты — это виртуальные объекты, которые предоставляют вам доступ к собранным SQL Server данным в памяти.

Хотя динамические административные объекты действительно очень полезны, у них есть ряд недостатков. Наиболее важная проблема, на которую следует обратить внимание, возникает, когда выполняется перезапуск экземпляра сервера, который вы инспектируете. Накопленные совокупные данные становятся бесполезными, если экземпляр сервера был недавно перезапущен.

В SQL Server 2012 существует более 130 динамических административных объектов, доступных для запроса. Для получения подробной информации обратитесь к электронной документации к разделу "Динамические административные представления и функции (Transact-SQL)" по адресу <http://msdn.microsoft.com/ru-ru/library/windowsazure/ms188754.aspx>. В данном занятии вы познакомитесь с наиболее важными из них с точки зрения настройки запросов и индексов.

## **Наиболее важные динамические административные объекты для настройки объектов**

Динамические административные объекты группируются в множество категорий. К наиболее важным группам для анализа производительности запросов относятся следующие.

- Динамические административные объекты, связанные с операционной системой, и специфичные для SQL Server (SQLOS). SQLOS управляет ресурсами операционной системы, специфическими для SQL Server.

- Динамические административные объекты, связанные с выполнением. Эти динамические административные объекты позволяют заглянуть внутрь запросов, которые были выполнены, включая текст запросов, план выполнения, количество выполнений и пр.
- Динамические административные объекты, связанные с индексами. Эти динамические административные объекты предоставляют полезную информацию об использовании индексов и отсутствующих индексах.

Вы можете начать сеанс анализа со сбора системной информации. Используя относящееся к SQL Server динамическое административное представление `sys.dm_os_sys_info`, можно получить основную информацию об экземпляре сервера, как показано в следующем запросе:

```
SELECT cpu_count AS logical_cpu_count,
 cpu_count / hyperthread_ratio AS physical_cpu_count,
 CAST(physical_memory_kb / 1024. AS int) AS physical_memory_mb,
 sqlserver_start_time
 FROM sys.dm_os_sys_info;
```

Этот запрос возвращает информацию о количестве логических ЦП, физических ЦП, физической памяти и времени запуска SQL Server. Последняя информация говорит о том, имеет ли смысл анализировать накопленную информацию или нет.

Относящееся к SQL Server динамическое административное представление `sys.dm_os_waiting_tasks` предоставляет информацию о сеансах, которые в данный момент ожидают чего-либо. Например, сеансы могут быть заблокированы другим сеансом. Вы можете объединить это динамическое административное представление со связанным с выполнением запросов динамическим административным представлением `sys.dm_exec_sessions` для получения информации о пользователе, хосте и приложении, находящихся в ожидании. Также вы можете использовать флагок `is_user_process` из динамического административного представления `sys.dm_exec_sessions`, чтобы отфильтровать системные сеансы. Следующий запрос предоставляет такую информацию:

```
SELECT S.login_name, S.host_name, S.program_name,
 WT.session_id, WT.wait_duration_ms,
 WT.wait_type, WT.blocking_session_id, WT.resource_description
 FROM sys.dm_os_waiting_tasks AS WT
 INNER JOIN sys.dm_exec_sessions AS S
 ON WT.session_id = S.session_id
 WHERE s.is_user_process = 1;
```

Связанное с выполнением запросов динамическое административное представление `sys.dm_exec_requests` возвращает информацию о выполняющихся в данный момент запросах. Оно включает столбец `sql_handle`, являющийся хэш-картой текста пакета T-SQL, который выполняется. Вы можете использовать этот дескриптор для извлечения полного текста пакета с помощью связанной с выполнением динамической административной функции `sys.dm_exec_sql_text`, которая принимает этот дескриптор в качестве параметра. Следующий запрос объединяет информацию

о текущих запросах, их времени ожидания и текст из пакета SQL с динамическим административным представлением sys.dm\_exec\_sessions, чтобы также получить информацию о пользователе, хосте и приложении.

```
SELECT S.login_name, S.host_name, S.program_name,
 R.command, T.text,
 R.wait_type, R.wait_time, R.blocking_session_id
 FROM sys.dm_exec_requests AS R
 INNER JOIN sys.dm_exec_sessions AS S
 ON R.session_id = S.session_id
 OUTER APPLY sys.dm_exec_sql_text(R.sql_handle) AS T
 WHERE S.is_user_process = 1;
```

Вы можете извлечь множество информации о выполняемых запросах из связанного с выполнением запросов динамического административного представления sys.dm\_exec\_query\_stats. Вы можете извлечь информацию о дисковых операциях ввода-вывода на запрос, использование ЦП на запрос, истекшее время на запрос и т. д. С помощью динамического административного представления sys.dm\_exec\_sql\_text можно также извлечь текст запроса. Вы можете извлечь текст определенного запроса из текста пакета с помощью столбцов statement\_start\_offset и statement\_end\_offset динамического административного представления sys.dm\_exec\_query\_stats. Это довольно сложная процедура. Следующий запрос перечисляет 5 запросов, использовавших большую часть логических операций ввода-вывода, а также выводит текст запроса, извлеченный из текста пакета.

```
SELECT TOP (5)
 (total_logical_reads + total_logical_writes) AS total_logical_IO,
 execution_count,
 (total_logical_reads/execution_count) AS avg_logical_reads,
 (total_logical_writes/execution_count) AS avg_logical_writes,
 (SELECT SUBSTRING(text, statement_start_offset/2 + 1,
 (CASE WHEN statement_end_offset = -1
 THEN LEN(CONVERT(nvarchar(MAX), text)) * 2
 ELSE statement_end_offset
 END - statement_start_offset)/2)
 FROM sys.dm_exec_sql_text(sql_handle)) AS query_text
 FROM sys.dm_exec_query_stats
 ORDER BY (total_logical_reads + total_logical_writes) DESC;
```

SQL Server предоставляет множество полезных связанных с индексом динамических административных объектов. Вы можете находить отсутствующие индексы с помощью sys.dm\_db\_missing\_index\_details, sys.dm\_db\_missing\_index\_columns, sys.dm\_db\_missing\_index\_groups и sys.dm\_db\_missing\_index\_group\_stats динамических административных объектов. Обратите внимание, не следует иметь слишком много индексов; хотя запросы их используют, SQL Server должен их поддерживать. С помощью представления каталога sys.indexes и динамического административного представления sys.dm\_db\_index\_usage\_stats можно найти индексы, которые не используются. Вы будете использовать связанные с индексом динамические административные объекты в практикуме к данному занятию.

**КОНТРОЛЬНЫЙ ВОПРОС**

- Какой динамический административный объект предоставляет подробный текст выполненных запросов?

**Ответ на контрольный вопрос**

- Вы можете извлечь текст выполненных пакетов и запросов из динамического административного представления sys.dm\_exec\_sql\_text.

**ПРАКТИКУМ Использование связанных с индексом динамических административных объектов**

В данном практикуме вы будете использовать связанные с индексом динамические административные объекты. Вы также узнаете, как важно иметь репрезентативную выборку запросов, собранную до использования динамических административных объектов, которые возвращают накопительные значения.

**Задание 1. Нахождение неиспользованных индексов**

В этом задании вам нужно найти неиспользованные индексы.

1. Перезапустите ваш экземпляр SQL Server. (Замечание: не делайте этого на рабочем сервере.) Если вы не закрыли SSMS, это можно сделать из **Object Explorer**, щелкнув правой кнопкой мыши на вашем экземпляре сервера и выбрав команду **Restart** (Перезагрузка). Для этого также можно использовать SQL Server Configuration Manager (Диспетчер конфигурации SQL Server).
2. Подключите ваш экземпляр сервера к SSMS. Откройте новое окно запроса, нажав кнопку **New Query** (Создать запрос).
3. Измените контекст на базу данных TSQL2012.
4. Найдите некластеризованные индексы, которые не использовались с момента последнего запуска экземпляра сервера, с помощью следующего запроса:

```
SELECT OBJECT_NAME(I.object_id) AS objectname,
 I.name AS indexname, I.index_id AS indexid
 FROM sys.indexes AS I
 INNER JOIN sys.objects AS O
 ON O.object_id = I.object_id
 WHERE I.object_id > 100
 AND I.type_desc = 'NONCLUSTERED' AND I.index_id NOT IN
 (SELECT S.index_id
 FROM sys.dm_db_index_usage_stats AS S
 WHERE S.object_id=I.object_id
 AND I.index_id=S.index_id
 AND database_id = DB_ID('TSQL2012'))
 ORDER BY objectname, indexname;
```

Посмотрите на результаты. Запрос возвратил все некластеризованные индексы из базы данных TSQL2012. Конечно, поскольку вы только что перезапустили ваш экземпляр сервера, SQL Server еще не собрал никаких данных об использовании индекса. Это показывает, как важно иметь репрезентативную выборку, когда выполняется анализ кумулятивных данных.

## **Задание 2. Нахождение отсутствующих индексов**

В этом задании вам нужно найти отсутствующие индексы.

- Быстро создайте таблицу и индекс на этой таблице из практикума к предыдущему занятию, но используйте в 10 раз меньше данных. Затем проиндексируйте таблицу и выполните запрос, который может выиграть от использования дополнительного индекса. Вот как выглядит код:

```
SELECT N1.n * 100000 + O.orderid AS norderid, O.*
INTO dbo.NewOrders FROM Sales.Orders AS O
 CROSS JOIN (VALUES(1),(2),(3)) AS N1(n);
GO
CREATE NONCLUSTERED INDEX idx_nc_orderid
 ON dbo.NewOrders(orderid);
GO
SELECT norderid
FROM dbo.NewOrders
WHERE norderid = 110248
ORDER BY norderid;
GO
```

- Найдите отсутствующий индекс с помощью связанных с индексом динамических административных объектов. Используйте следующий запрос:

```
SELECT MID.statement AS [Database.Schema.Table],
 MIC.column_id AS ColumnId,
 MIC.column_name AS ColumnName,
 MIC.column_usage AS ColumnUsage,
 MIGS.user_seeks AS UserSeeks,
 MIGS.user_scans AS UserScans,
 MIGS.last_user_seek AS LastUserSeek,
 MIGS.avg_total_user_cost AS AvgQueryCostReduction,
 MIGS.avg_user_impact AS AvgPctBenefit
FROM sys.dm_db_missing_index_details AS MID
 CROSS APPLY sys.dm_db_missing_index_columns (MID.index_handle) AS MIC
 INNER JOIN sys.dm_db_missing_index_groups AS MIG
 ON MIG.index_handle = MID.index_handle
 INNER JOIN sys.dm_db_missing_index_group_stats AS MIGS
 ON MIG.index_group_handle=MIGS.group_handle
 ORDER BY MIGS.avg_user_impact DESC;
```

- Проанализируйте полученную информацию.

4. После того как закончите анализировать полученные данные, очистите базу данных, используя следующий код:

```
DROP TABLE dbo.NewOrders;
```
5. Выйдите из SSMS.

## Резюме занятия

- Динамические административные объекты помогают быстро получить информацию, собранную SQL Server.
- Для анализа запросов следует использовать SQLoS, относящиеся к выполнению, и связанные с индексом динамические административные объекты.
- Связанные с индексом динамические административные объекты не только предоставляют полезные сведения об использовании индексов, они также информируют об отсутствующих индексах.

## Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какой динамический административный объект предоставляет информацию об использовании индексов?
  - A. sys.dm\_exec\_query\_stats.
  - B. sys.dm\_exec\_query\_text.
  - C. sys.dm\_db\_index\_usage\_stats.
  - D. sys.indexes.
2. Какой недостаток динамических управляющих объектов считается наиболее важным?
  - A. Необходимость иметь достаточное количество собранных данных с момента последнего перезапуска SQL Server.
  - B. Динамические управляющие объекты сложно использовать.
  - C. Динамические управляющие объекты недоступны в стандартной редакции SQL Server.
  - D. Необходимость повторно создавать динамические управляющие объекты перед каждым сбором аналитики.
3. Как вы можете найти текст выполненного запроса с помощью динамических управляющих объектов?
  - A. Эта информация представлена в динамическом административном представлении sys.dm\_exec\_query\_stats.
  - B. Выполнив запрос к динамическому административному представлению sys.dm\_exec\_query\_stats.

- С. Динамическое административное представление sys.dm\_exec\_query\_stats возвращает текст запроса.
- Д. Невозможно найти текст запроса с помощью динамических управляющих объектов.

## Упражнения

---

В следующих упражнениях вам нужно применить полученные знания об инструментах SQL Server, использующихся для анализа запросов с целью их оптимизации. Ответы на вопросы можно найти в *приложении "Ответы" в конце книги*.

### Упражнение 1. Анализ запросов

Вы получили срочный вызов от менеджера компании, в которой занимаетесь поддержкой SQL Server. Менеджер жалуется на то, что база данных SQL Server не отвечает уже несколько часов. Вам нужно оптимизировать только один запрос, но как можно быстрее. Но вы должны найти наиболее проблемный запрос. Вы подключились к экземпляру SQL Server. Вы видите, что сотни пользователей одновременно используют сервер, но при этом не запущены ни подсистема расширенных событий, ни трассировка SQL. Вы также выяснили, что SQL Server работает без перерыва уже 6 месяцев.

1. С чего вы начнете анализировать эту ситуацию?
2. Когда вы найдете наиболее проблемный запрос, что вы будете делать дальше?

### Упражнение 2. Непрерывный мониторинг

Вам нужно постоянно выполнять мониторинг экземпляра SQL Server, для того чтобы выявлять потенциальные проблемные места. Ваш экземпляр SQL Server используется в интенсивном режиме. Вы не должны увеличивать нагрузку на него процедурами мониторинга.

1. Какой инструмент будете вы использовать для мониторинга?
2. Как вы минимизируете влияние этого инструмента?

### Рекомендуемые упражнения

Для успешного освоения экзаменационных тем, представленных в данной главе, выполните следующие задания.

### **Дополнительные сведения о расширенных событиях, планах выполнения и динамических административных объектах**

Вы можете найти большое количество дополнительной информации о расширенных событиях, планах выполнения и динамических управляющих объектах в электронной документации по SQL Server.

- **Задание 1.** Чтобы полностью изучить расширенные объекты SQL Server, прочтите информацию, представленную в электронной документации по SQL Server 2012 в разделе "Расширенные события" по адресу [http://msdn.microsoft.com/ru-ru/library/bb630282\(SQL.110\).aspx](http://msdn.microsoft.com/ru-ru/library/bb630282(SQL.110).aspx).
- **Задание 2.** Чтобы полностью изучить планы выполнения, прочтите информацию, предоставленную в электронной документации по SQL Server 2012 в разделе "Справочник по логическим и физическим операторам Showplan" по адресу <http://msdn.microsoft.com/ru-ru/library/ms191158.aspx>.
- **Задание 3.** Чтобы полностью изучить динамические административные объекты SQL Server, прочтите информацию, предоставленную в электронной документации по SQL Server 2012 в разделе "Динамические административные представления и функции (Transact-SQL)" по адресу <http://msdn.microsoft.com/ru-ru/library/ms188754.aspx>.

# ГЛАВА 15

## Реализация индексов и статистика

### Темы экзамена

- Создание объектов баз данных.
  - Создание и изменение представлений (простые инструкции).
- Устранение неполадок и оптимизация.
  - Оптимизация запросов.

В главе 14 вы познакомились с инструментами, которые помогают находить проблемы производительности. Индексы упоминались там много раз. Это не случайно. Правильное индексирование необходимо для хорошей производительности баз данных. Чтобы создавать оптимальные индексы, надо понимать, как Microsoft SQL Server хранит данные в таблицах и индексах и как затем он получает доступ к этим данным. Этому посвящено самое большое занятие данной главы — занятие 1.

Если вы будете писать неэффективные запросы, вам не помогут никакие индексы. В занятии 2 вы узнаете, как писать аргументы, которые SQL Server может использовать для поиска по индексам. Но даже если у вас есть индексы и соответствующие аргументы поиска, SQL Server все равно может принять решение не использовать индекс. Это может произойти, потому что статистическая информация об индексе не представлена или устарела. В занятии 3 вы узнаете, как получать информацию о статистике и поддерживать ее.

### ПРЕЖДЕ ВСЕГО

Для полного освоения этой главы необходимо иметь:

- понимание основ реляционных баз данных;
- опыт работы с SQL Server Management Studio (SSMS);
- некоторый опыт в написании кода на языке T-SQL;
- доступ к экземпляру SQL Server 2012 с установленной учебной базой данных TSQL2012.

## Занятие 1. Реализация индексов

Данные в файле данных внутренне организованы в SQL Server в страницы. Страница — это модуль размером в 8 Кбайт, принадлежащий только одному объекту, например, таблице или индексу. Таблица является наименьшей единицей для чтения и записи. Страницы далее собраны в экстенты. Экстент состоит из 8 последовательных страниц. Страницы экстента могут принадлежать одному объекту или нескольким объектам. Если страницы принадлежат нескольким объектам, экстент называется *смешанным* экстентом. Если страницы принадлежат только одному объекту, экстент называется *однородным* экстентом. SQL Server хранит первые 8 страниц объекта в смешанных экстентах. Когда размер объекта превышает 8 страниц, SQL Server выделяет объекту дополнительные однородные экстенты. При такой организации небольшие объекты требуют меньше пространства, а большие — меньше фрагментированы.



В то время как предшествующая информация кратко описывает физическую структуру SQL Server, с точки зрения разработчика баз данных, логические структуры имеют намного большее значение. Это занятие посвящено логическим структурам.

**Изучив материал этого занятия, вы сможете:**

- ✓ Объяснить, как SQL Server использует страницы и экстенты
- ✓ Описать кучи и сбалансированные деревья
- ✓ Создавать кластеризованные и некластеризованные индексы
- ✓ Создавать индексированные представления

**Продолжительность занятия — 60 минут.**

### Кучи и сбалансированные деревья

Страницы — это физические структуры. SQL Server организует данные на страницах в логические структуры.



SQL Server организует таблицы в кучи или *сбалансированные деревья*. Таблица, организованная в виде сбалансированного дерева, также известна как кластеризованная таблица или кластеризованный индекс (эти два термина являются взаимозаменяемыми).

Индексы всегда организованы как сбалансированные деревья. Другие индексы, которые не содержат всех данных и служат в качестве указателей на строки в таблицах для быстрого поиска, называются *некластеризованными* индексами.

#### Кучи

Куча — это довольно простая структура. Данные в куче не имеют никакой логической организации. Куча — это просто набор страниц и экстентов.



SQL Server отслеживает, какие страницы и экстенты принадлежат объекту, с помощью специальных системных страниц, называемых страницами карты распределения.

ления индекса (Index Allocation Map, IAM). Каждая таблица или индекс имеет по крайней мере одну страницу IAM, называемую *первой страницей IAM*. Одна страница IAM может указывать примерно на 4 Гбайт пространства. Большие объекты могут иметь более одной страницы IAM. Страницы IAM объекта организованы в *двунаправленный список*; у каждой страницы есть указатель на ее потомка и предшественника. SQL Server хранит указатели на первые страницы IAM в собственных внутренних системных таблицах.



На рис. 15.1 показано, как выглядит демонстрационная таблица для хранения клиентских заказов, когда она организована в виде кучи.

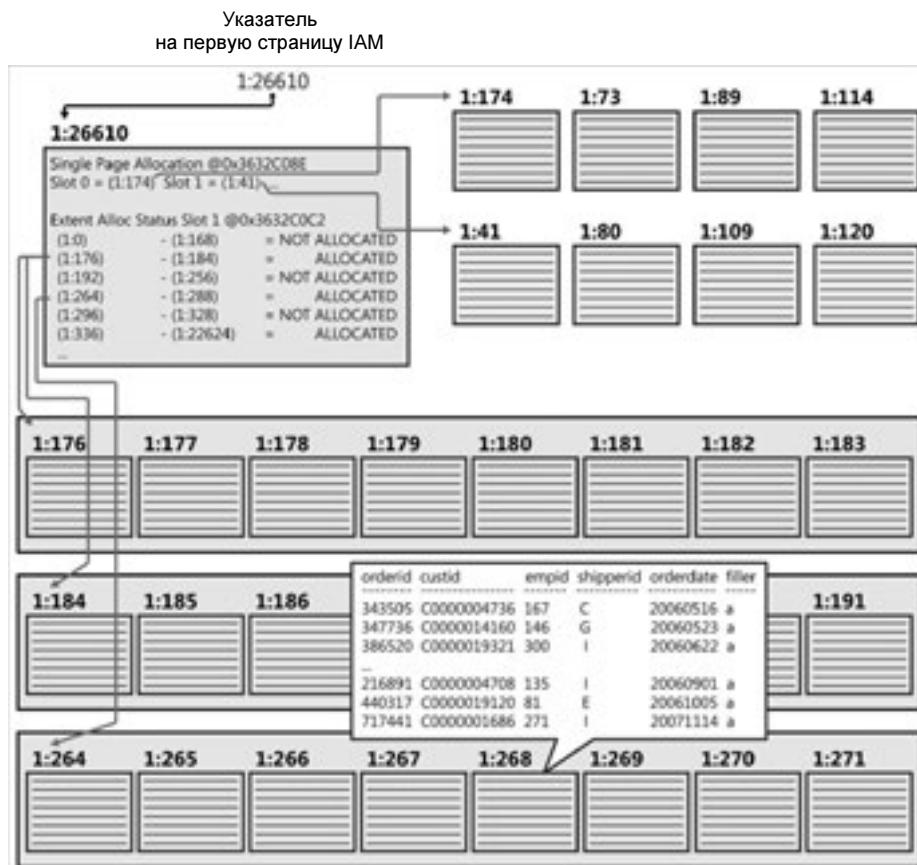


Рис. 15.1. Таблица, организованная как куча

SQL Server может найти данные в куче, только просматривая всю кучу. SQL Server использует страницы IAM для просмотра куч в физическом порядке или в *порядке выделения*. Даже если запросу нужно извлечь только одну строку, SQL Server должен просмотреть всю кучу. SQL Server размещает новые строки в куче произвольно. Он может сохранить новую строку на существующей странице, если она еще не заполнена, либо выделить новую страницу или экстент для объекта, где вы встав-

ляете строку. Разумеется, это означает, что со временем кучи могут стать очень фрагментированными.

Структуру SQL Server проще понять с помощью примеров. Следующий код создает таблицу, организованную в виде кучи.

```
USE tempdb;
GO
CREATE TABLE dbo.TestStructure
(id INT NOT NULL,
 filler1 CHAR(36) NOT NULL,
 filler2 CHAR(216) NOT NULL);
```

Если вы не создаете кластеризованный индекс явно или неявно с помощью первичного ключа либо ограничения уникальности, таблица будет организована как куча. SQL Server не выделяет таблице никаких страниц при ее создании. Он выделяет первую страницу, а также первую IAM-страницу, когда выполняется первая вставка строки в таблицу. Общую информацию о таблицах и индексах можно найти в представлении каталога sys.indexes.

Следующий запрос извлекает основную информацию о таблице dbo.TestStructure, которая была создана в предыдущем коде.

```
SELECT OBJECT_NAME(object_id) AS table_name,
 name AS index_name, type, type_desc
 FROM sys.indexes
 WHERE object_id = OBJECT_ID(N'dbo.TestStructure', N'U');
```

Вот как выглядят результаты этого запроса.

| table_name    | index_name | type | type_desc |
|---------------|------------|------|-----------|
| TestStructure | NULL       | 0    | HEAP      |

Столбец type хранит значение 0 для куч, 1 для кластеризованных таблиц (индексов) и 2 для некластеризованных индексов. Можно узнать, сколько страниц выделено под объект, из функции динамического управления sys.dm\_db\_index\_physical\_stats или с помощью системной процедуры dbo.sp\_spaceused, как показано в следующем коде. Поскольку этот код используется много раз в этом занятии, для простоты мы ссылаемся на него как на "проверку выделения кучи".

```
SELECT index_type_desc, page_count,
 record_count, avg_page_space_used_in_percent
 FROM sys.dm_db_index_physical_stats
 (DB_ID(N'tempdb'), OBJECT_ID(N'dbo.TestStructure'),
 NULL, NULL, 'DETAILED');
EXEC dbo.sp_spaceused @objname = N'dbo.TestStructure', @updateusage = true;
```

Выходные данные этих двух команд выглядят следующим образом:

| index_type_desc | page_count | record_count | avg_page_space_used_in_percent |
|-----------------|------------|--------------|--------------------------------|
| HEAP            | 0          | 0            | 0                              |

| name          | rows | reserved | data | index_size | unused |
|---------------|------|----------|------|------------|--------|
| TestStructure | 0    | 0 KB     | 0 KB | 0 KB       | 0 KB   |

Вы видите, что таблица пустая, а пустая таблица не занимает места. Обратите внимание на последний столбец первого запроса, `avg_space_used_in_percent`. Этот столбец показывает внутреннюю фрагментацию. *Внутренняя фрагментация* означает, что страницы не заполнены. Чем больше строк вы храните на одной странице, тем меньше страниц SQL Server должен читать для извлечения этих строк, и тем меньше памяти он использует для кэшированных страниц при одинаковом количестве строк. В кучах вы не получите большой внутренней фрагментации, поскольку SQL Server, как вы уже знаете, хранит новые строки на существующих страницах, если там имеется достаточно места. Теперь вставьте первую строку.



```
INSERT INTO dbo.TestStructure (id, filler1, filler2)
 VALUES (1, 'a', 'b');
```

Если вы выполните код проверки выделения кучи снова, результат будет следующим:

| index_type_des | page_count | record_count | avg_page_space_used_in_percent |            |        |
|----------------|------------|--------------|--------------------------------|------------|--------|
| HEAP           | 1          | 1            | 3.24932048430937               |            |        |
| name           | rows       | reserved     | data                           | index_size | unused |
| TestStructure  | 1          | 16 KB        | 8 KB                           | 8 KB       | 0 KB   |

Таблица с одной строкой занимает одну страницу. Среднее использование пространства страницы является низким, поскольку на странице только одна строка. Результат процедуры `dbo.sp_spaceused` показывает, что у таблицы есть две зарезервированные страницы: одна для данных и одна для первой страницы IAM. Вы видите, что SQL Server выделяет только страницу, но не экстент для этой таблицы. Теперь заполните страницу с помощью следующего кода:

```
DECLARE @i AS int = 1;
WHILE @i < 30
BEGIN
 SET @i = @i + 1;
 INSERT INTO dbo.TestStructure (id, filler1, filler2)
 VALUES (@i, 'a', 'b');
END;
```

После выделения кучи запустите код снова. Вы получите следующий результат:

| index_type_des | page_count | record_count | avg_page_space_used_in_percent |            |        |
|----------------|------------|--------------|--------------------------------|------------|--------|
| HEAP           | 1          | 30           | 98.1961947121324               |            |        |
| name           | rows       | reserved     | data                           | index_size | unused |
| TestStructure  | 30         | 480 KB       | 320 KB                         | 8 KB       | 0 KB   |

| name          | rows | reserved | data | index_size | unused |
|---------------|------|----------|------|------------|--------|
| TestStructure | 30   | 16 KB    | 8 KB | 8 KB       | 0 KB   |

По-прежнему выделена только одна страница; но эта страница не имеет внутренней фрагментации, поскольку страница не может разместить дополнительные строки. Попробуйте вставить дополнительные строки.

```
INSERT INTO dbo.TestStructure (id, filler1, filler2)
VALUES (31, 'a', 'b');
```

Код проверки выделения кучи возвращает следующие выходные данные:

| index_type_des | page_count | record_count | avg_page_space_used_in_percent |            |        |
|----------------|------------|--------------|--------------------------------|------------|--------|
| HEAP           | 2          | 32           | 50.7227575982209               |            |        |
| name           | rows       | reserved     | data                           | index_size | unused |
| TestStructure  | 31         | 24 KB        | 16 KB                          | 8 KB       | 0 KB   |

Теперь вы видите, что выделена одна дополнительная страница из смешанного экстента. Разумеется, внутренняя фрагментация возросла, поскольку вторая страница почти пуста. Заполните 8 страниц с помощью следующего кода:

```
DECLARE @i AS int = 31;
WHILE @i < 240
BEGIN
 SET @i = @i + 1;
 INSERT INTO dbo.TestStructure (id, filler1, filler2)
 VALUES (@i, 'a', 'b');
END;
```

Результаты выполнения кода проверки выделения кучи выглядят следующим образом:

| index_type_des | page_count | record_count | avg_page_space_used_in_percent |            |        |
|----------------|------------|--------------|--------------------------------|------------|--------|
| HEAP           | 8          | 240          | 98.1961947121324               |            |        |
| name           | rows       | reserved     | data                           | index_size | unused |
| TestStructure  | 240        | 72 KB        | 64 KB                          | 8 KB       | 0 KB   |

8 страниц заполнены. Что произойдет, если вы вставите еще одну строку? Попробуйте это сделать с помощью следующего кода:

```
INSERT INTO dbo.TestStructure (id, filler1, filler2)
VALUES (241, 'a', 'b');
```

Результаты выполнения кода проверки выделения кучи выглядят следующим образом:

| index_type_des | page_count | record_count | avg_page_space_used_in_percent |            |        |
|----------------|------------|--------------|--------------------------------|------------|--------|
| HEAP           | 9          | 241          | 87.6465530022239               |            |        |
| name           | rows       | reserved     | data                           | index_size | unused |
| TestStructure  | 241        | 72 KB        | 64 KB                          | 8 KB       | 0 KB   |

| name          | rows | reserved | data  | index_size | unused |
|---------------|------|----------|-------|------------|--------|
| TestStructure | 241  | 136 KB   | 72 KB | 8 KB       | 56 KB  |

### **ПРИМЕЧАНИЕ    Одинаковые результаты не гарантированы**

При разной конфигурации базы данных — например, если у вас есть два или более файлов данных, — результаты могут несколько различаться.

Теперь вы видите, что хотя таблица занимает только 9 страниц, под нее зарезервированы 16 страниц данных плюс первая страница IAM. Как показывает результат процедуры `dbo.sp_spaceused`, SQL Server зарезервировал 136 Кбайт под таблицу, что означает 17 страниц; 56 Кбайт по-прежнему не используются. Неиспользованные 56 Кбайт пространства означают, что 7 страниц из однородного экстента по-прежнему пусты. Первые 8 страниц размещаются в смешанных экстентах. Поскольку таблица уже больше 8 страниц, SQL Server выделяет однородные экстенты для дополнительного пространства, которое необходимо.

## **Кластеризованные индексы**

Таблица строится как сбалансированное дерево, когда вы создаете кластеризованный индекс. Такая структура называется сбалансированным деревом, потому что она напоминает обратное дерево. Каждое сбалансированное дерево имеет одну или более корневую страницу и хотя бы одну или более конечных страниц. Кроме того, оно может иметь 0 или более промежуточных уровней. Все данные в кластеризованной таблице хранятся в конечных страницах. Данные хранятся в логическом порядке ключа кластеризации. *Ключ кластеризации* может состоять из одного столбца или нескольких столбцов. Если ключ состоит из нескольких столбцов, тогда это *составной ключ*. В ключе может быть до 16 столбцов; размер всех столбцов вместе в составном ключе не должен превышать 900 байт. Обратите внимание, данные хранятся логически и физически никак не упорядочены. SQL Server по-прежнему использует страницы IAM, чтобы следовать физическому выделению.



### **ВАЖНО!                    Кластеризованный индекс — это таблица**

Когда вы создаете кластеризованный индекс, вы не копируете данные; напротив, вы реорганизуете таблицу. Кроме того, данные не отсортированы физически; если вам нужны упорядоченные выходные данные запроса, вы должны использовать предложение `ORDER BY`.

Страницы выше конечного уровня указывают на страницы конечного уровня. Стока на странице выше конечного уровня содержит значение ключа кластеризации и указатель на страницу, где это значение начинается в логически упорядоченном конечном уровне. Если одна страница может указывать на все страницы конечного уровня, тогда выделяется только корневая страница. Если нужно более одной страницы для указания на страницы конечного уровня, SQL Server создает первую страницу промежуточного уровня, которая указывает на страницы конечного уровня. Строки корневой страницы указывают на страницы промежуточного уровня. Если корневая страница не может указывать на все промежуточные страницы первого уровня, SQL Server создает новый промежуточный уровень. Страницы

цы одного уровня организованы как двунаправленный список; таким образом, SQL Server может найти предыдущую и последующую страницы в логической последовательности для любой конкретной страницы. Кроме страниц сбалансированного дерева, SQL Server использует страницы IAM, чтобы отслеживать физическое выделение страниц сбалансированного дерева.

Вы можете использовать столбец или столбцы с уникальными или неуникальными значениями для ключа кластеризованного индекса. Но SQL Server всегда внутренними средствами поддерживает уникальность ключа кластеризации. Он добавляет значение *uniquifier*, которое является последовательным целочисленным значением, к повторяющимся значениям. Первое значение хранится без *uniquifier*; первое повторяющееся значение получает *uniquifier* со значением 1, второе — со значением 2 и т. д. Вы поймете, почему значения ключа кластеризации должны быть внутренне уникальными, когда познакомитесь с некластеризованными индексами далее в этом занятии.

На рис. 15.2 показана кластеризованная структура демонстрационной таблицы для заказов клиентов. Обратите внимание, дата заказа (столбец *od* на рисунке) используется для ключа кластеризации. Поскольку этот столбец не является уникальным, *uniquifier* добавляется к повторяющимся значениям (столбец *upq* на рисунке).

SQL Server может выполнять поиск строки в кластеризованном индексе. Чтобы найти конкретную строку в таблице, представленной на рис. 15.2, SQL Server должен выполнить чтение только трех страниц. Если бы таблица была построена как куча, SQL Server должен был бы выполнять чтение всей таблицы, что было бы со-поставимо с чтением всех страниц на конечном уровне кластеризованного индекса. Разумеется, если вы запросите все строки, SQL Server будет просматривать конечный уровень также и кластеризованного индекса. Просмотр кластеризованного индекса может быть выполнен в логической последовательности либо, если логический порядок не требуется, в физической последовательности или последовательности выделения. Кроме того, SQL Server может выполнить частичный просмотр, если в вашем запросе требуются последовательные строки, упорядоченные по ключу кластеризации. В этом заключаются преимущества кластеризованных индексов перед кучами.

Кластеризованные индексы также имеют некоторые недостатки по сравнению с кучами. Когда вы вставляете новую строку в заполненную страницу, SQL Server должен разбить страницу на две и перенести половину строк на вторую страницу. Это происходит потому, что SQL Server должен поддерживать логический порядок строк. Таким образом возникает внутренняя фрагментация, которую вы не получите в куче. Кроме того, новая страница (или новый однородный экстент для большой таблицы) может быть зарезервирована где угодно в файле данных. Физическая последовательность страниц и экстентов кластеризованной таблицы не должна соответствовать логической последовательности. Если страницы физически не упорядочены, тогда кластеризованная таблица будет *логически* фрагментирована. Это называется *внешней* фрагментацией. Внешняя фрагментация может замедлить полный или частичный поиск в логической последовательности.

Указатель на первую страницу IAM  
Указатель на первую страницу

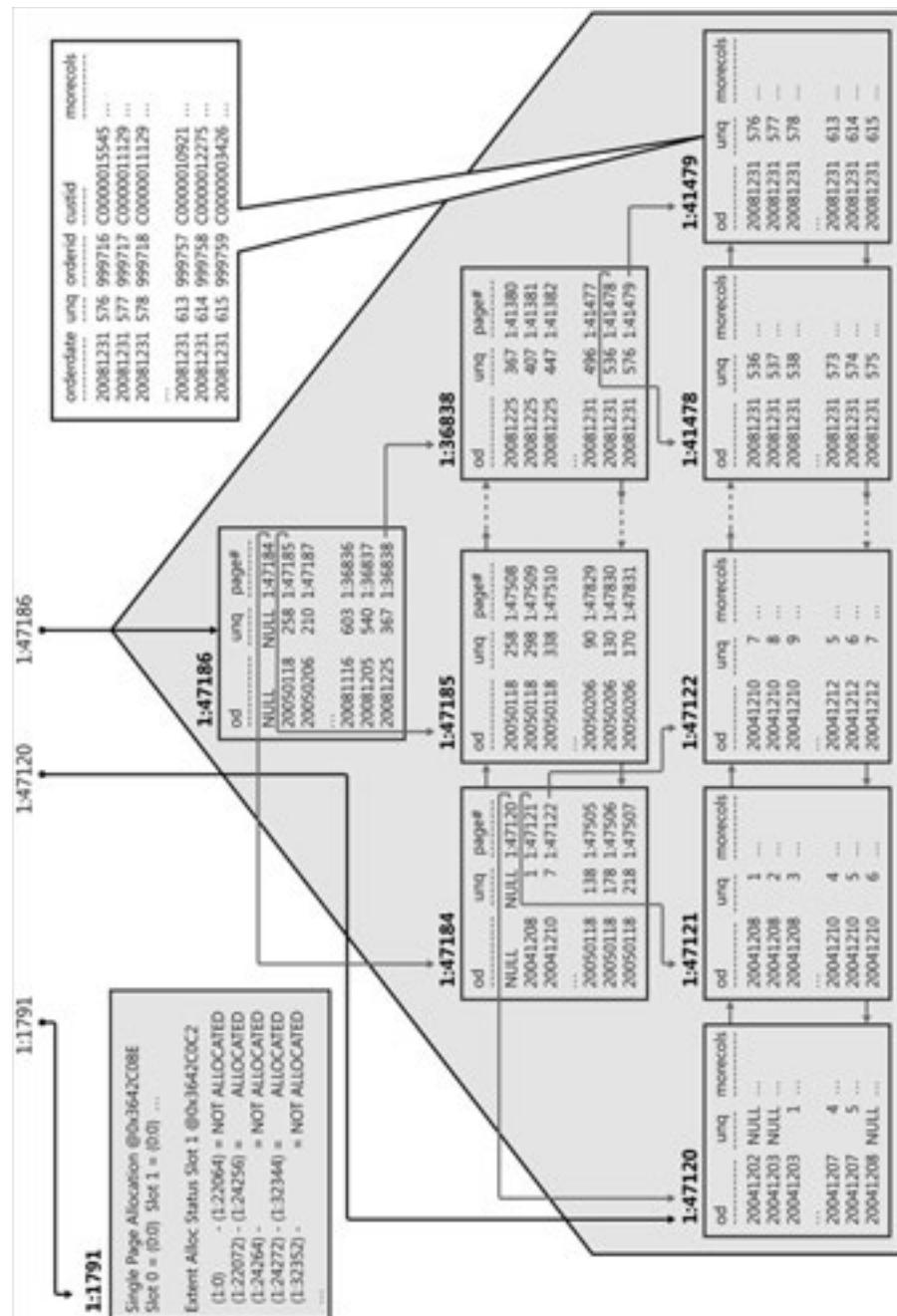


Рис. 15.2. Таблица, организованная как сбалансированное дерево

В большинстве случаев преимущества кластеризованных таблиц превосходят их недостатки. Невозможно контролировать внутреннюю фрагментацию с помощью параметра `FILLFACTOR` для страниц конечного уровня и с помощью параметра `PAD_INDEX` для страниц более высокого уровня инструкции `CREATE INDEX`. Вы можете перестроить или реорганизовать индекс, чтобы избавиться от внешней фрагментации с помощью инструкций `ALTER INDEX...REORGANIZE` или `ALTER INDEX...REBUILD`.

Короткий ключ кластеризации означает, что большее количество строк может поместиться на страницах выше конечного уровня. Таким образом, потенциально необходимо меньшее число уровней. Меньшее число уровней означает более эффективный индекс, потому что SQL Server должен выполнять чтение меньшего числа страниц, чтобы найти строку. Столбец `uniquifier` дополняет ключ; поэтому использование короткого и уникального ключа предпочтительно для поиска. Это очень характерно для приложений оперативной обработки транзакций (*online transaction processing, OLTP*). Для таких приложений выбор последовательного целочисленного значения в качестве ключа кластеризации, как правило, является правильным решением. Однако в сценариях организации хранилищ данных многие запросы выполняют чтение огромного количества данных, как правило, упорядоченных. Например, многие запросы к хранилищам данных выполняют поиск строк в порядке столбца даты или даты-времени. В таком случае, возможно, предпочтительно поддерживать частичный просмотр данных и создать кластеризованный индекс на столбце даты.

**СОВЕТ****Подготовка к экзамену**

---

Убедитесь в том, что вы хорошо поняли, как выбирать ключ кластеризации в разных средах.

Вы лучше разберетесь в кластеризованных таблицах с помощью примеров. Следующий код усекает таблицу, созданную и заполненную в разд. "Кучи" этого занятия, и реорганизует эту таблицу в сбалансированное дерево с помощью столбца `id` как ключа кластеризации.

```
TRUNCATE TABLE dbo.TestStructure;
CREATE CLUSTERED INDEX idx_cl_id ON dbo.TestStructure(id);
```

Вы можете снова проверить эту таблицу в представлении каталога `sys.indexes`.

```
SELECT OBJECT_NAME(object_id) AS table_name,
 name AS index_name, type, type_desc
 FROM sys.indexes
 WHERE object_id = OBJECT_ID(N'dbo.TestStructure', N'U');
```

Запрос возвращает следующие выходные данные.

| table_name    | index_name | type | type_desc |
|---------------|------------|------|-----------|
| TestStructure | idx_cl_id  | 1    | CLUSTERED |

Как видите, значение `type` изменилось на 1, и куча более не существует. Когда вы создаете кластеризованный индекс, то фактически реорганизуете таблицу. Теперь заполните 621 страницу этой таблицы, используя уникальные значения для ключа кластеризации.

```

DECLARE @i AS int = 0;
WHILE @i < 18630
BEGIN
 SET @i = @i + 1;
 INSERT INTO dbo.TestStructure (id, filler1, filler2)
 VALUES (@i, 'a', 'b');
END;

```

Обратите внимание, если вы знаете, что значения должны быть уникальными, вы должны создать первичный ключ или ограничение уникальности для этой таблицы. Вы также можете создать уникальный индекс; но, поскольку уникальность ограничивает значения, следует использовать ограничения.

Основную информацию об индексе можно получить, запросив функцию динамического управления sys.dm\_db\_index\_physical\_stats. Следующий фрагмент кода будет использоваться в этой части занятия много раз, поэтому в дальнейшем будем называть его кодом "проверки выделения кластеризованного индекса".

```

SELECT index_type_desc, index_depth, index_level, page_count,
 record_count, avg_page_space_used_in_percent
FROM sys.dm_db_index_physical_stats
 (DB_ID(N'tempdb'), OBJECT_ID(N'dbo.TestStructure'),
 NULL, NULL, 'DETAILED');

```

Результат выполнения кода проверки выделения кластеризованного индекса будет следующим:

| index_type_desc | index_depth | index_level | page_count | record_count | avg_pg_spc_used_in_pct |
|-----------------|-------------|-------------|------------|--------------|------------------------|
| CLUSTERED INDEX | 2           | 0           | 621        | 18630        | 98.1961947121324       |
| CLUSTERED INDEX | 2           | 1           | 1          | 621          | 99.7158388930072       |

### **ПРИМЕЧАНИЕ Имена столбцов в выходных данных**

В показанном результате некоторые имена столбцов несколько сокращены по сравнению с именами столбцов реальных выходных данных, чтобы поместить эти выходные данные на странице книги.

Вы видите, что индекс имеет только два уровня: конечный уровень и корневую страницу. Корневая страница имеет 621 строку, которые указывают на 621 конечную страницу. В этом случае внутренняя фрагментация отсутствует. Теперь вставьте еще одну строку.

```

INSERT INTO dbo.TestStructure (id, filler1, filler2)
 VALUES (18631, 'a', 'b');

```

Выполнив код проверки выделения кластеризованного индекса, вы получите следующие выходные данные:

| index_type_desc | index_depth | index_level | page_count | record_count | avg_pg_spc_used_in_pct |
|-----------------|-------------|-------------|------------|--------------|------------------------|
| CLUSTERED INDEX | 3           | 0           | 622        | 18631        | 98.0435507783543       |
| CLUSTERED INDEX | 3           | 1           | 2          | 622          | 49.9258710155671       |
| CLUSTERED INDEX | 3           | 2           | 1          | 2            | 0.296515937731653      |

Теперь индекс имеет три уровня. Поскольку новая страница была выделена на конечном уровне, начальная корневая страница не может уже ссылаться на все конечные страницы. SQL Server добавил промежуточный уровень с двумя страницами, указывающими на 622 конечных страницы, и новую корневую страницу, указывающую на две страницы промежуточного уровня.

Чтобы продемонстрировать влияние `uniquifier`, следующий код усекает таблицу и заполняет 423 страницы с помощью неуникальных значений ключа кластеризации.

```
TRUNCATE TABLE dbo.TestStructure;
DECLARE @i AS int = 0;
WHILE @i < 8908
BEGIN
 SET @i = @i + 1;
 INSERT INTO dbo.TestStructure (id, filler1, filler2)
 VALUES (@i % 100, 'a', 'b');
END;
```

Если вы выполните код проверки выделения кластеризованного индекса, то получите следующий результат:

| index_type_desc   | index_depth | index_level | page_count | record_count | avg_pg_spc_used_in_pct |
|-------------------|-------------|-------------|------------|--------------|------------------------|
| CLUSTERED INDEX 2 | 0           | 423         | 8908       |              | 70.9815171732147       |
| CLUSTERED INDEX 2 | 1           | 1           | 423        |              | 99.8393872003954       |

Заметьте, корневая страница может ссылаться только на 423 страницы конечного уровня. Для заполнения двух уровней индекса нужно было только 8908 строк, тогда как с уникальными значениями ключа кластеризации в предыдущем примере SQL Server мог разместить 18 630 строк на двух уровнях.

Чтобы доказать это, добавьте еще одну строку.

```
INSERT INTO dbo.TestStructure (id, filler1, filler2)
VALUES (8909 % 100, 'a', 'b');
```

Код проверки выделения кластеризованного индекса возвращает следующие выходные данные:

| index_type_desc   | index_depth | index_level | page_count | record_count | avg_pg_spc_used_in_pct |
|-------------------|-------------|-------------|------------|--------------|------------------------|
| CLUSTERED INDEX 3 | 0           | 424         | 8909       |              | 70.8220039535458       |
| CLUSTERED INDEX 3 | 1           | 2           | 424        |              | 50.0370644922165       |
| CLUSTERED INDEX 3 | 2           | 1           | 2          |              | 0.395354583642204      |

Вы видите, что если значения ключа не уникальны, SQL Server должен добавить дополнительный уровень к индексу намного раньше.

До сих пор значения ключа кластеризации были последовательными. Что произойдет, если они не будут последовательными? Следующий код усекает таблицу `dbo.TestStructure`, удаляет существующий кластеризованный индекс, создает новый с помощью столбца `filler1` в качестве ключа кластеризации, а затем вставляет

9000 строк в таблицу с уникальными последовательными значениями в ключе кластеризации.

```
TRUNCATE TABLE dbo.TestStructure;
DROP INDEX idx_cl_id ON dbo.TestStructure;
CREATE CLUSTERED INDEX idx_cl_filler1 ON dbo.TestStructure(filler1);
DECLARE @i AS int = 0;
WHILE @i < 9000
BEGIN
 SET @i = @i + 1;
 INSERT INTO dbo.TestStructure (id, filler1, filler2)
 VALUES (@i, FORMAT(@i,'0000'), 'b');
END;
```

Теперь проверьте фрагментацию. Следующий код, который мы дальше будем называть кодом проверки фрагментации, проверяет внутреннюю фрагментацию (столбец avg\_page\_space\_used\_in\_percent) и внешнюю фрагментацию (столбец avg\_fragmentation\_in\_percent).

```
SELECT index_level, page_count,
 avg_page_space_used_in_percent, avg_fragmentation_in_percent
FROM sys.dm_db_index_physical_stats
(DB_ID(N'tempdb'), OBJECT_ID(N'dbo.TestStructure'),
NULL, NULL, 'DETAILED');
```

Выходные данные кода проверки фрагментации в этом случае будут такими:

| index_level | page_count | avg_page_space_used_in_percent | avg_fragmentation_in_percent |
|-------------|------------|--------------------------------|------------------------------|
| 0           | 300        | 98.1961947121324               | 1.66666666666667             |
| 1           | 3          | 55.5720286632073               | 0                            |
| 2           | 1          | 1.64319248826291               | 0                            |

### **ПРИМЕЧАНИЕ    Идентичные результаты не гарантируются**

Значения столбцов avg\_page\_space\_used\_in\_percent и avg\_fragmentation\_in\_percent могут несколько различаться в ваших результатах.

Вы видите, что индекс имеет три уровня. На конечном уровне внутренняя фрагментация отсутствует; кроме того, внешней фрагментации тоже почти нет. Все страницы на конечном уровне заполнены, и физическая последовательность почти такая же, как и логическая. Теперь выполните усечение таблицы и заполните ее случайными значениями в столбце filler1. Следующий код использует функцию NEWID() языка T-SQL, которая генерирует идентификаторы GUID и сохраняет их в столбце filler1.

```
TRUNCATE TABLE dbo.TestStructure;
DECLARE @i AS int = 0;
WHILE @i < 9000
BEGIN
 SET @i = @i + 1;
```

```

INSERT INTO dbo.TestStructure (id, filler1, filler2)
VALUES (@i, CAST(NEWID() AS CHAR(36)), 'b');
END;

```

Идентификаторы GUID, сгенерированные функцией NEWID(), почти случайные. Если вы выполните код проверки фрагментации снова, то получите следующий результат:

| index_level | page_count | avg_page_space_used_in_percent | avg_fragmentation_in_percent |
|-------------|------------|--------------------------------|------------------------------|
| 0           | 432        | 68.1842599456387               | 98.6111111111111             |
| 1           | 4          | 60.0197677291821               | 50                           |
| 2           | 1          | 2.19915987150976               | 0                            |

#### **ПРИМЕЧАНИЕ Идентичные результаты не гарантируются**

Значения столбцов avg\_page\_space\_used\_in\_percent и avg\_fragmentation\_in\_percent могут несколько различаться в ваших результатах.

Вы видите, что страницы конечного уровня занимают только 68% пространства, заполненного строками. Это потому, что SQL Server выполнил разбиение нескольких страниц. Кроме того, внешняя фрагментация составляет около 99%; почти ни одна страница не расположена в правильном логическом порядке. Вы видите, что использование идентификаторов GUID как ключей кластеризации может привести к созданию довольно неэффективных индексов. Внешняя фрагментация в основном замедляет просмотры, что не должно часто происходить в средах OLTP, которые очень важны для хранилищ данных. Внутренняя фрагментация является проблемой в обоих сценариях, потому что таблица намного больше, чем она должна быть с последовательным ключом.

От фрагментации можно избавиться, если перестроить или реорганизовать индекс. Реорганизация индекса — процесс более медленный, но менее ресурсоемкий. В общем случае, следует выполнять реорганизацию индекса, если внешняя фрагментация менее 30%, и перестраивать его, если она больше 30%. Следующий код перестраивает индекс.

```
ALTER INDEX idx_cl_filler1 ON dbo.TestStructure REBUILD;
```

Если же вы предпочтете реорганизовать таблицу, вам нужно просто заменить ключевое слово REBUILD ключевым словом REORGANIZE. Если вы выполните код проверки фрагментации после перестройки индекса, то увидите в выходных данных, что фрагментация практически исчезла.

| index_level | page_count | avg_page_space_used_in_percent | avg_fragmentation_in_percent |
|-------------|------------|--------------------------------|------------------------------|
| 0           | 300        | 98.1961947121324               | 0.6666666666666667           |
| 1           | 2          | 83.3703978255498               | 0                            |
| 2           | 1          | 1.08722510501606               | 0                            |

#### **ПРИМЕЧАНИЕ Идентичные результаты не гарантируются**

Значения столбцов avg\_page\_space\_used\_in\_percent и avg\_fragmentation\_in\_percent могут немного отличаться в ваших результатах.

### КОНТРОЛЬНЫЙ ВОПРОС

- Какой тип ключа кластеризации вы выберете для среды OLTP?

### Ответ на контрольный вопрос

- Для среды OLTP наилучшим выбором может быть короткий, уникальный и последовательный ключ кластеризации.

## Реализация некластеризованных индексов

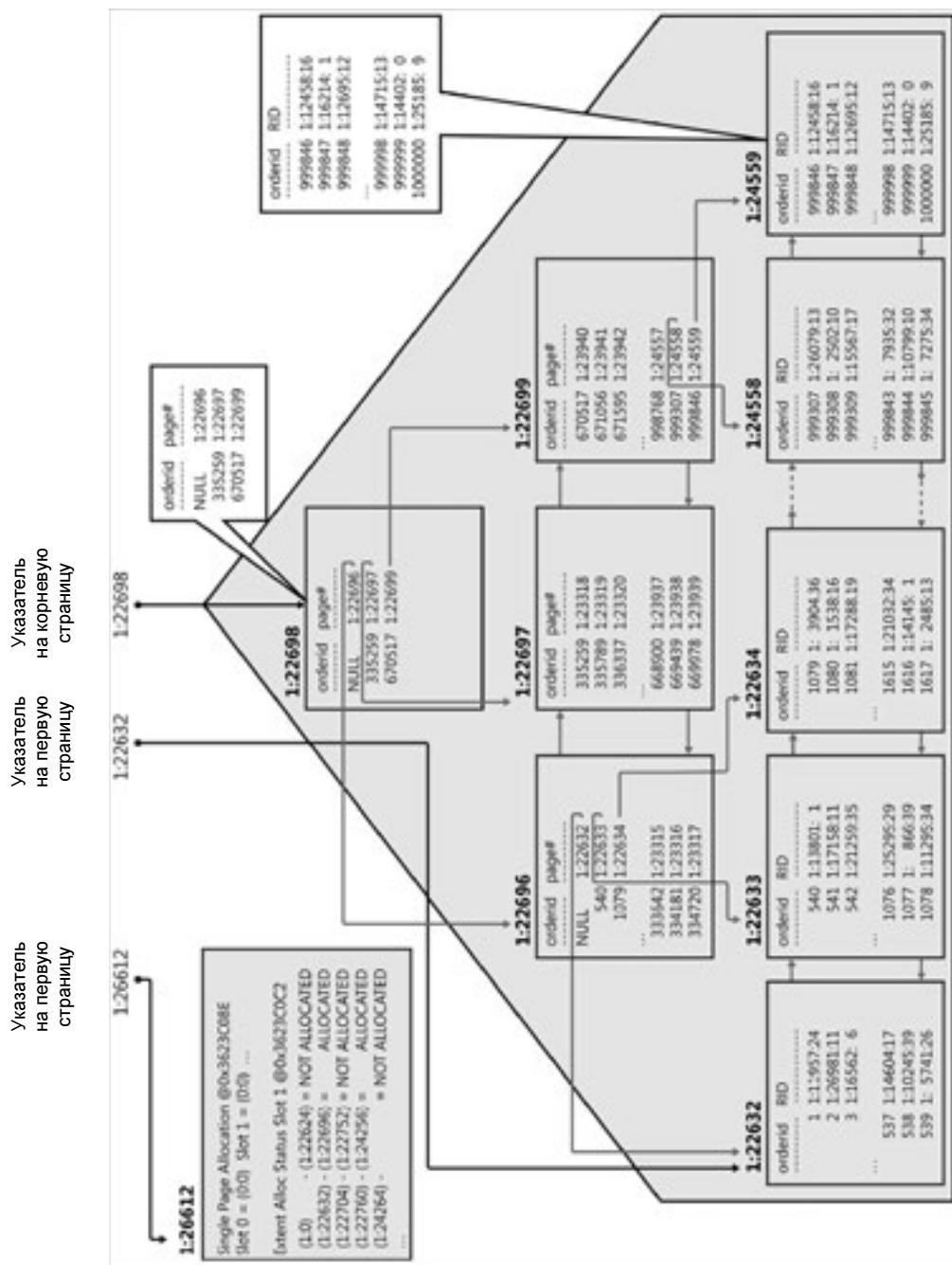
Некластеризованные индексы имеют структуру, очень похожую на структуру кластеризованных индексов. Корневой и промежуточные уровни выглядят фактически так же, как в кластеризованном индексе. Конечный уровень отличается, поскольку он не содержит всех данных. Что хранится на конечном уровне некластеризованного индекса, зависит от организации базовой таблицы, независимо от того, организована она как куча или как сбалансированное дерево. Можно иметь до 999 некластеризованных индексов на одной таблице.

Конечный уровень некластеризованного индекса содержит индексные ключи и *указатели строк*. Еще раз: ключ может иметь до 16 столбцов, и размер всех столбцов вместе в составном ключе не может превышать 900 байт. Указатель строк указывает на строку в базовой таблице. Если таблица является кучей, тогда указатель строк называется идентификатором строк (RID). Это 8-байтный указатель, содержащий идентификатор файла базы данных и идентификатор страницы целевой строки, а также идентификатор целевой строки на этой странице.

На рис. 15.3 показан некластеризованный индекс в куче. Он использует тот же пример таблицы клиентских заказов, что и другие рисунки в этой главе. Столбец `orderid` используется для индексного ключа.

Чтобы выполнить поиск строки, SQL Server должен пройти индекс до конечного уровня и затем прочитать соответствующую страницу из кучи и извлечь строку из страницы. Операция извлечения строки из кучи называется *уточняющим запросом RID*. Если запрос очень избирательный и ищет одну строку или только небольшое количество строк, тогда индексный поиск с помощью уточняющего запроса RID очень эффективен. Поскольку страницы на том же уровне индекса соединены в двунаправленный список, SQL Server также может выполнить частичный или полный упорядоченный поиск на некластеризованном индексе, а затем выполнить уточняющие запросы RID, не начиная поиск с корневой страницы для каждой строки. Но по мере увеличения числа строк, извлекаемых запросом, уточняющий запрос RID становится значительно дороже, потому что стоимость уточняющего запроса RID — как правило, одна страница на строку.

Если таблица организована как сбалансированное дерево, тогда указателем строк является ключ кластеризации. Это означает, что когда SQL Server ищет строку, он должен пройти все уровни на некластеризованном индексе, а затем также все уровни кластеризованного индекса. Эта операция называется *поиском ключа*. На первый взгляд, это выглядит хуже, чем извлечение одной страницы



**Рис. 15.3.** Некластеризованный индекс на куче

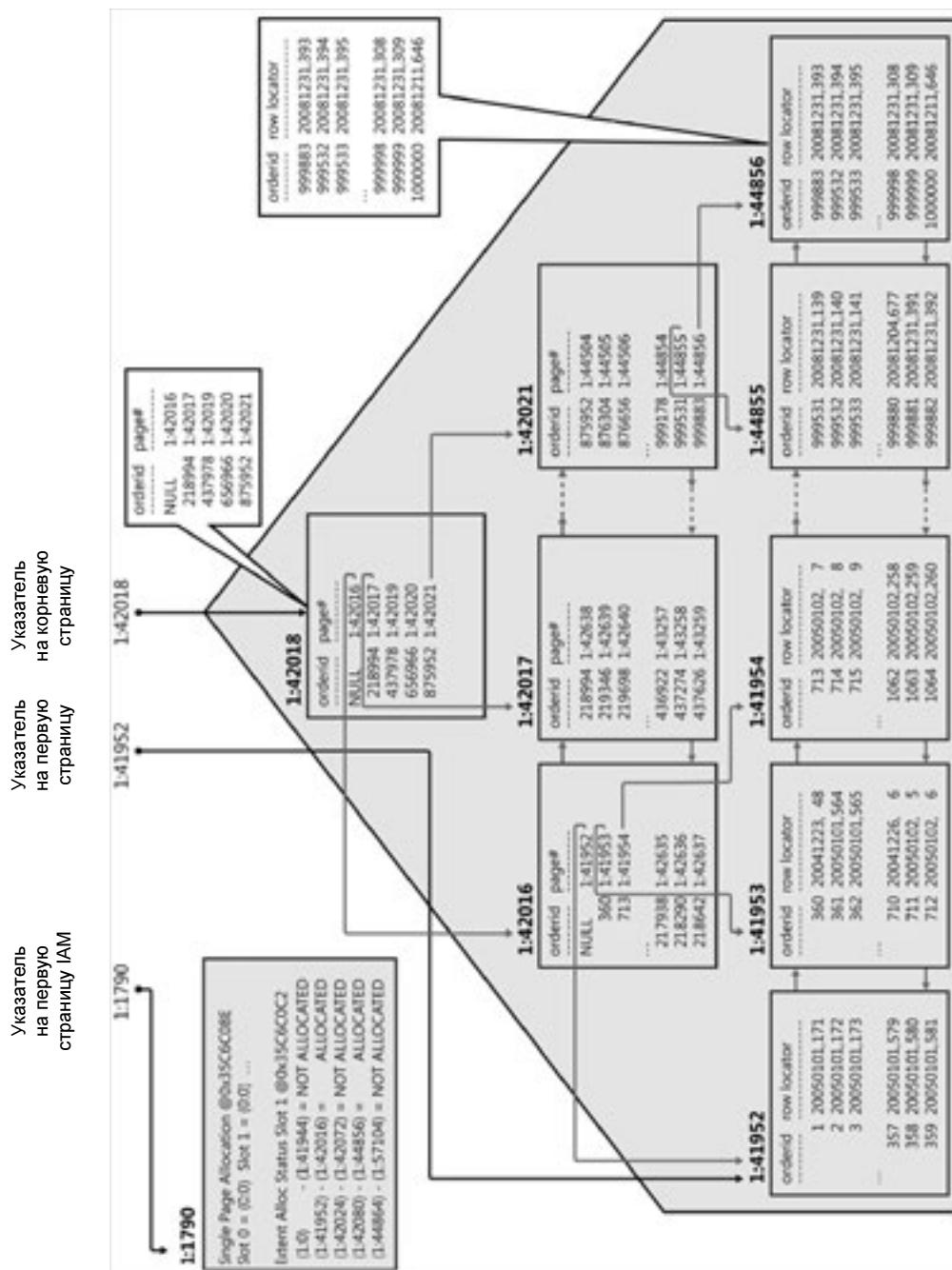
из кучи. Однако, поскольку в этом случае указатель строк указывает на логическую структуру, а не на физическую структуру, не имеет значения, где физически расположена строка в таблице. Это значит, что вы можете легко реорганизовать или перестроить кластеризованный индекс; до тех пор, пока вы не измените ключ кластеризации, SQL Server не должен обновлять некластеризованные индексы. Если строка перемещается в кучу, SQL Server должен обновить все некластеризованные индексы, чтобы соответствовать новой позиции. У SQL Server имеется оптимизация для обновлений кучи; если строка должна переместиться на другую страницу, SQL Server оставляет указатель на новое местоположение на исходной странице, поэтому SQL Server по-прежнему не обязан обновлять все некластеризованные индексы. Однако, даже с такой оптимизацией, более правильным считается организовывать таблицы как сбалансированные деревья. Если ключ кластеризации узкий — например, 4-байтовое целое число, — тогда SQL Server может также разместить больше строк на странице конечного уровня, чем когда используется RID в качестве указателя строк.

На рис. 15.4 продемонстрирован некластеризованный индекс на кластеризованной таблице. Это тот же пример таблицы клиентских заказов; данные заказа используются для ключа кластеризации, а идентификатор заказа — для ключа некластеризованного индекса.

Обратите внимание, ключ кластеризации не является уникальным, и поэтому `uniquifier` добавляется к повторяющимся значениям. Ключ некластеризованного индекса является уникальным. Если запрос искал бы конкретный идентификатор заказа (одна строка) и получил бы ключ кластеризации с датой заказа, одинаковой для нескольких строк, тогда SQL Server возвратил бы неправильный результирующий набор. SQL Server возвратил бы все строки с одинаковой датой заказа, что, разумеется, неприемлемо. Поэтому SQL Server должен поддерживать уникальность ключей кластеризации внутренними средствами.

Ключ кластеризации должен быть кратким и уникальным, поскольку он появляется во всех некластеризованных индексах. Однако заметим еще раз, что это не общее правило; в сценариях, связанных с хранилищами данных, может оказаться более предпочтительным выбрать ключ кластеризации, который поддерживает частые неполные просмотры. В любом случае кластеризация не должна меняться часто, а еще лучше — не меняться совсем. Если вы обновите ключ кластеризации, SQL Server должен обновить все некластеризованные индексы. Вы также должны сначала создать кластеризованный индекс, а затем все некластеризованные индексы. Если вы измените структуру таблицы из кучи на сбалансированное дерево или наоборот посредством создания или удаления кластеризованного индекса, и эта таблица имеет существующие некластеризованные индексы, SQL Server должен заново открыть все некластеризованные индексы.

Вы можете открыть фильтруемый некластеризованный индекс. Фильтруемый индекс охватывает только подмножество значений столбцов, и, следовательно, применяется к подмножеству строк таблицы. Фильтруемые некластеризованные индексы полезны, когда какие-то значения в столбце появляются редко, тогда как другие значения появляются часто. В подобных случаях следует создать фильтруе-



**Рис. 15.4.** Некластеризованный индекс на кластеризованной таблице

мый индекс только на редко появляющихся значениях. SQL Server использует этот индекс для поиска редко появляющихся значений, но выполняет просмотр часто появляющихся значений. Поддержка фильтруемых индексов является недорогой, поскольку SQL Server должен обновлять их только по изменениям редко встречающихся значений. Фильтруемый индекс создается посредством добавления предложения `WHERE` в инструкции `CREATE INDEX`. Можно использовать фильтруемый индекс для обеспечения уникальности фильтрации. Например, представьте, что столбец имеет значения `NULL` в нескольких строках; однако известные значения должны быть уникальными. Вы не можете создать фильтруемый первичный ключ или ограничение уникальности; но вы можете создать фильтруемый уникальный некластеризованный индекс только из известных значений, что разрешит использование нескольких значений `NULL` и отклонит дубликаты известных значений.

В SQL Server 2012 существует способ хранения некластеризованных индексов. Кроме стандартного хранилища строк, SQL Server 2012 может сохранять данные индекса столбец за столбцом в так называемом индексе *columnstore*. Индексы *columnstore* могут многократно ускорить запросы хранилища данных, от 10 до 100 раз.



Индекс *columnstore* — это просто разновидность некластеризованного индекса на таблице. Оптимизатор запросов SQL Server рассматривает использование индекса *columnstore* на этапе оптимизации запроса точно так же, как если бы это был любой другой индекс. Единственное, что нужно сделать, чтобы воспользоваться преимуществами этой функциональности, — создать индекс *columnstore* на таблице.

Индекс *columnstore* хранится в сжатом виде. Степень сжатия может быть до 10 раз от первоначального размера индекса. Когда запрос ссылается на один столбец, являющийся частью индекса *columnstore*, SQL Server извлекает с диска только этот столбец; он не извлекает целые строки, как в случае с хранилищем строк. Это также снижает дисковый ввод-вывод и потребление кэш-памяти. Индексы *columnstore* используют собственный алгоритм сжатия; в индексе *columnstore* нельзя использовать сжатие строк или страниц.

С другой стороны, SQL Server должен возвращать строки. Поэтому строки должны быть восстановлены при выполнении запроса. Такое восстановление строк требует времени и использует ресурсы ЦП и памяти. Избирательные запросы, которые затрагивают только несколько строк, могут не получать выгоды от использования индексов *columnstore*.

Индексы *columnstore* ускоряют запросы хранилища данных, но не рабочую нагрузку OLTP. Из-за восстановления строк и других служебных операций при обновлении сжатых данных, таблицы, содержащие индекс *columnstore*, становятся доступными только для чтения. Если вы хотите обновить таблицу с помощью индекса *columnstore*, вы сначала должны удалить индекс *columnstore*. Если вы используете секционирование таблиц, то можете переключить секцию на другую таблицу, которая не использует индекс *columnstore*, обновить там данные, создать индекс *columnstore* на этой таблице (которая имеет меньший поднабор данных) и затем переключить данные новой таблицы обратно на раздел исходной таблицы.

Индекс columnstore разделен на единицы, называемые сегментами. Сегменты хранятся как большие объекты и состоят из нескольких страниц. *Сегменты* — это единицы передачи с диска в память. Каждый сегмент имеет метаданные, которые хранят минимальное и максимальное значения каждого столбца для этого сегмента. Это делает возможным раннее удаление сегментов в подсистеме хранилища. SQL Server загружает в память только сегменты, требуемые в запросе.



Более подробно об индексах columnstore и их эффективном использовании вы узнаете в главе 17. Вы будете работать с некластеризованными индексами в практикуме к данному занятию.

## Реализация индексированных представлений

Можно оптимизировать запросы, которые выполняют статистическую обработку данных и множественные соединения с помощью постоянного хранения статистических и объединенных данных. Например, можно создать новую таблицу посредством объединенных и статистических данных и затем поддерживать эту таблицу на протяжении ETL-процесса (процесса извлечения, преобразования и загрузки данных).

Однако создание дополнительных таблиц для объединенных и статистических данных нельзя считать наилучшим решением, поскольку использование этих таблиц означает, что вы должны изменить запросы отчетов. К счастью, для хранения объединенных и статистических таблиц можно использовать другую возможность. Вы можете создать представление с запросом, который объединяет и выполняет статистическую обработку данных. Затем это представление можно проиндексировать, чтобы получить *индексированное представление*. С помощью индексации вы материализуете представление. В выпуске SQL Server 2012 Enterprise Edition оптимизатор запросов SQL Server использует индексированное представление автоматически — без необходимости изменять запрос. SQL Server также поддерживает индексированные представления автоматически. Однако, чтобы ускорить загрузку данных, можно удалить или запретить индекс перед загрузкой и затем заново открыть или перестроить его после загрузки.



### **К СВЕДЕНИЮ**    **Возможности, поддерживаемые выпусками SQL Server 2012**

Дополнительную информацию об использовании индексированного представления и прочих возможностях, поддерживаемых другими выпусками SQL Server 2012, можно найти в электронной документации по SQL Server в разделе "Возможности, поддерживаемые различными выпусками SQL Server 2012" по адресу [http://msdn.microsoft.com/ru-ru/library/cc645993\(SQL.110\).aspx](http://msdn.microsoft.com/ru-ru/library/cc645993(SQL.110).aspx).

Индексированные представления имеют множество ограничений и предварительных требований, и вам следует обратиться к электронной документации по SQL Server для их изучения. Но вы можете создать простой тест, который показывает, как могут использоваться индексированные представления. Следующий запрос выполняет статистическую обработку столбца `qty` таблицы `Sales.OrderDetails` и

столбца shipcountry таблицы Sales.Orders в демонстрационной базе данных TSQL2012. Также код устанавливает параметр STATISTICS IO в ON, чтобы измерять ввод-вывод.

```
USE TSQL2012;
SET STATISTICS IO ON;
-- Статистический запрос с соединением
SELECT O.shipcountry, SUM(OD.qty) AS totalordered
FROM Sales.OrderDetails AS OD
INNER JOIN Sales.Orders AS O
 ON OD.orderid = O.orderid
GROUP BY O.shipcountry;
```

Запрос выполняет 11 логических чтений в таблице Sales.OrderDetails и 21 логическое чтение в таблице Sales.Orders. Вы можете создать представление с помощью этого запроса и проиндексировать его, как показано в следующем коде:

```
-- Создание представления
CREATE VIEW Sales.QuantityByCountry
WITH SCHEMABINDING
AS
SELECT O.shipcountry, SUM(OD.qty) AS total_ordered,
 COUNT_BIG(*) AS number_of_rows
FROM Sales.OrderDetails AS OD
INNER JOIN Sales.Orders AS O
 ON OD.orderid = O.orderid
GROUP BY O.shipcountry;
GO
-- Индексация представления
CREATE UNIQUE CLUSTERED INDEX idx_cl_shipcountry
ON Sales.QuantityByCountry(shipcountry);
GO
```

Обратите внимание, представление должно быть создано с параметром SCHEMABINDING, если вы хотите его индексировать. Кроме того, вы должны использовать статистическую функцию COUNT\_BIG. Для получения дополнительной информации ознакомьтесь с предварительными требованиями в электронной документации по SQL Server 2012 в разделе "Создание индексированных представлений" по адресу <http://msdn.microsoft.com/ru-ru/library/ms191432.aspx>. Как бы там ни было, после создания представления и индекса снова выполните статистический запрос и измерьте ввод-вывод. На этот раз запрос выполняет только два логических чтения в представлении Sales.QuantityByCountry.

Проанализировав индексированное представление, установите параметр STATISTICS IO в OFF и очистите вашу базу данных TSQL2012, выполнив следующий код:

```
SET STATISTICS IO OFF;
DROP VIEW Sales.QuantityByCountry;
```

## ПРАКТИКУМ Аналisis некластеризованных индексов

В этом практикуме вы будете анализировать некластеризованные индексы.

### Задание 1. Реализация некластеризованного индекса в куче

В этом задании вам нужно создать некластеризованный индекс на куче.

1. Запустите SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
2. Откройте новое окно запроса, нажав кнопку **New Query** (Создать запрос).
3. Измените контекст на базу данных tempdb. Установите параметр **NOCOUNT** в значение **ON**, чтобы сообщение, показывающее счетчик числа строк, обработанных командой, не выводилось. Тогда операции вставки будут намного быстрее. Используйте следующий код:

```
USE tempdb;
SET NOCOUNT ON;
```

4. Создайте таблицу с именем **dbo.teststructure**, которая имеет ту же структуру, что и таблица, использованная для тестирования кучи и кластеризованного индекса. Используйте следующий код:

```
CREATE TABLE dbo.TestStructure
(id INT NOT NULL,
 filler1 CHAR(36) NOT NULL,
 filler2 CHAR(216) NOT NULL);
GO
```

5. Обратите внимание, таблица организована как куча, поскольку вы не создали кластеризованный индекс. Создайте некластеризованный индекс на столбце **filler1** с помощью следующего кода:

```
CREATE NONCLUSTERED INDEX idx_nc_filler1 ON dbo.TestStructure(fillter1);
```

6. Запросите представление каталога **sys.indexes**, чтобы убедиться, что таблица сохранена как куча и существует некластеризованный индекс.

```
SELECT OBJECT_NAME(object_id) AS table_name,
 name AS index_name, type, type_desc
 FROM sys.indexes
 WHERE object_id = OBJECT_ID(N'dbo.TestStructure', N'U');
```

7. Вставьте 24 472 строки в таблицу. Создайте последовательные значения для столбцов **id** и **filler1**. Используйте следующий код:

```
DECLARE @i AS int = 0;
WHILE @i < 24472
BEGIN
 SET @i = @i + 1;
 INSERT INTO dbo.TestStructure (id, fillter1, fillter2)
 VALUES (@i, FORMAT(@i,'00000'), 'b');
END;
```

8. Используйте функцию динамического управления sys.dm\_db\_index\_physical\_stats для проверки, сколько уровней имеет некластеризованный индекс и сколько страниц и строк существует на каждом уровне. Также проверьте кучу. Вы можете использовать следующий код:

```
SELECT index_type_desc, index_depth, index_level,
 page_count, record_count
 FROM sys.dm_db_index_physical_stats
 (DB_ID(N'tempdb'), OBJECT_ID(N'dbo.TestStructure'),
 NULL, NULL, 'DETAILED');
```

9. Вы должны иметь два уровня некластеризованного индекса. Теперь вставьте другую строку.

```
INSERT INTO dbo.TestStructure (id, filler1, filler2)
 VALUES (24473, '24473', 'b');
```

10. Проверьте количество уровней некластеризованного индекса и кучу еще раз.

```
SELECT index_type_desc, index_depth, index_level,
 page_count, record_count
 FROM sys.dm_db_index_physical_stats
 (DB_ID(N'tempdb'), OBJECT_ID(N'dbo.TestStructure'),
 NULL, NULL, 'DETAILED');
```

Теперь вы должны иметь три уровня в некластеризованном индексе.

## **Задание 2. Реализация некластеризованного индекса на кластеризованной таблице**

В этом задании вам нужно изменить физическую структуру таблицы из предыдущего задания, преобразовав ее из кучи в кластеризованный индекс. Вы увидите разницу между некластеризованным индексом на куче и некластеризованным индексом на кластеризованной таблице.

1. Усеките таблицу, созданную в предыдущем задании, и создайте кластеризованный индекс на столбце id.

```
TRUNCATE TABLE dbo.TestStructure;
CREATE CLUSTERED INDEX idx_cl_id ON dbo.TestStructure(id);
GO
```

2. Запросите представление каталога sys.indexes, чтобы убедиться, что таблица сохранена как сбалансированное дерево и некластеризованный индекс существует.

```
SELECT OBJECT_NAME(object_id) AS table_name,
 name AS index_name, type, type_desc
 FROM sys.indexes
 WHERE object_id = OBJECT_ID(N'dbo.TestStructure', N'U');
```

3. Вставьте 28 864 строки в таблицу. Создайте последовательные значения для столбцов id и filter1. Используйте следующий код:

```
DECLARE @i AS int = 0;
WHILE @i < 28864
BEGIN
 SET @i = @i + 1;
 INSERT INTO dbo.TestStructure (id, filler1, filler2)
 VALUES (@i, FORMAT(@i,'00000'), 'b');
END;
```

4. Проверьте количество уровней некластеризованного и кластеризованного индексов.

```
SELECT index_type_desc, index_depth, index_level,
 page_count, record_count
FROM sys.dm_db_index_physical_stats
(DB_ID(N'tempdb'), OBJECT_ID(N'dbo.TestStructure'),
NULL, NULL, 'DETAILED');
```

5. Кластеризованный индекс должен иметь три уровня, некластеризованный — два. Вы можете разместить больше строк на каждой странице некластеризованного индекса на кластеризованном индексе, в некластеризованном индексе на куче, поскольку ключ кластеризации короче, чем RID. Теперь добавьте еще одну строку.

```
INSERT INTO dbo.TestStructure (id, filler1, filler2)
VALUES (28865, '28865', 'b');
```

6. Проверьте количество уровней некластеризованного индекса и кластеризованного индекса еще раз.

```
SELECT index_type_desc, index_depth, index_level,
 page_count, record_count
FROM sys.dm_db_index_physical_stats
(DB_ID(N'tempdb'), OBJECT_ID(N'dbo.TestStructure'),
NULL, NULL, 'DETAILED');
```

Теперь некластеризованный индекс должен иметь три уровня.

7. Очистите базу данных tempdb.  
8. Закройте окно запроса.

## Резюме занятия

- Можно сохранить таблицу как кучу или как сбалансированное дерево. Если таблица сохранена как сбалансированное дерево, она является кластеризованной; она также называется кластеризованным индексом.
- Можно создать некластеризованный индекс на куче или на кластеризованной таблице.
- Также можно индексировать представление.

## Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какие уровни может иметь индекс? (Выберите все подходящие варианты.)
  - A. Промежуточный уровень.
  - B. Уровень кучи.
  - C. Корневой уровень.
  - D. Конечный уровень.
2. Сколько кластеризованных индексов можно создать на таблице?
  - A. 999.
  - B. 16.
  - C. 1.
  - D. 900.
3. Что является указателем строк, когда таблица сохранена как сбалансированное дерево?
  - A. RID.
  - B. Индексный ключ columnstore.
  - C. Ключ кластеризации.
  - D. Таблица никогда не сохраняется как сбалансированное дерево.

## Занятие 2. Использование аргументов поиска

---

Индексы полезны только в том случае, если запросы их используют. Вы должны знать, какие типы запросов выигрывают от их использования и какие типы запросов их не используют, даже если они существуют. Кроме того, при фильтрации строк нужно писать правильные предикаты, чтобы дать возможность оптимизатору запросов SQL Server использовать индексы.

**Изучив материал этого занятия, вы сможете:**

- ✓ Поддерживать запросы с индексами
- ✓ Использовать в запросах нужные аргументы поиска

**Продолжительность занятия — 35 минут.**

## Поддержка запросов с индексами

Написание эффективных запросов начинается с включения в запрос предложения WHERE для фильтрации строк. Предложение WHERE является одной из важнейших частей запроса, которая может выигрывать от использования индекса. Можно прове-

рить, использовался ли запрос, просмотрев предполагаемый или действительный план запроса. Также можно отслеживать использование индекса, запросив динамическое административное представление `sys.dm_db_index_usage_stats`. Помните, что предоставляемая динамическим управляющим объектом информация накапливается, начиная с последнего запуска SQL Server.

Следующий запрос показывает использование индекса в базе данных TSQL2012. Запрос был выполнен сразу после перезапуска SQL Server.

```
SELECT OBJECT_NAME(S.object_id) AS table_name,
 I.name AS index_name, S.user_seeks,
 S.user_scans, S.user_lookups
 FROM sys.dm_db_index_usage_stats AS S
 INNER JOIN sys.indexes AS i
 ON S.object_id = I.object_id
 AND S.index_id = I.index_id
 WHERE S.object_id = OBJECT_ID(N'Sales.Orders', N'U');
```

Мы будем использовать данный запрос далее в этом занятии, поэтому для простоты будем называть его запросом "использования индексов". Этот запрос не возвращает строк, поскольку у SQL Server пока что нет никакой информации об использовании индекса. Следующий запрос не содержит предложения `WHERE`, он извлекает все строки из таблицы `Sales.Orders`.

```
ELECT orderid, custid, shipcity
 FROM Sales.Orders;
```

План выполнения для этого запроса показывает, что SQL Server использовал сканирование кластеризованного индекса. Была просмотрена целая таблица, хотя на таблице `Sales.Orders` имеется множество индексов. Просмотр был неупорядоченным, или невыделенным, как показано на рис. 15.5. Свойство оператора `Ordered` установлено в значение `False`. Помните, что порядок не гарантируется, не включено предложение `ORDER BY`.

Добавление в запрос предложения `WHERE` не гарантирует, что будет использоваться индекс. Это предложение должно быть поддержано соответствующим индексом с достаточной степенью избирательности. Если запрос возвращает слишком много строк, он менее затратен для SQL Server с точки зрения выполнения сканирования таблицы или кластеризованного индекса, чем поиск по некластеризованному индексу с последующим уточняющим запросом по RID или по ключу. Например, хотя следующий запрос является достаточно избирательным, SQL Server сканирует кластеризованный индекс, потому что предикат `WHERE` не поддерживается индексом. Не существует индекса, который имел бы в качестве столбца `shipcity`.

```
SELECT orderid, custid, shipcity
 FROM Sales.Orders
 WHERE shipcity = N'Vancouver';
```

Предложение `JOIN` запроса может также выигрывать от использования надлежащих индексов. Более подробно о соединениях и их использовании с индексами написано в главе 17.



Рис. 15.5. Просмотр неупорядоченного кластеризованного индекса

Если запрос выполняет статистическую обработку данных и использует предложение `GROUP BY`, следует рассмотреть возможность использования этого предложения с индексом. SQL Server может проводить статистическую обработку данных, используя статистический оператор хэширования или потока. Статистическая обработка потока выполняется быстрее; однако она требует отсортированных входных данных. Статистический запрос может выигрывать от использования индекса, даже если он не содержит предложения `GROUP BY`. Например, если используется статистическая функция `MIN()`, и у вас есть соответствующий индекс, тогда SQL Server может выполнять поиск только первого значения индекса и не обязан сканировать всю таблицу целиком. Следующий статистический запрос не поддерживается индексом, поскольку в таблице `Sales.Orders` нет индекса, который мог бы использовать в качестве ключа столбец `shipregion`.

```
SELECT shipregion, COUNT(*) AS num_regions
FROM Sales.Orders
GROUP BY shipregion;
```

На рис. 15.6 показан план выполнения для этого запроса.

Обратите внимание, результаты предыдущего запроса не упорядочены. Включение предложения `GROUP BY` в запрос не гарантирует отсортированный результирующий набор. Если вам нужен отсортированный результат, используйте предложение

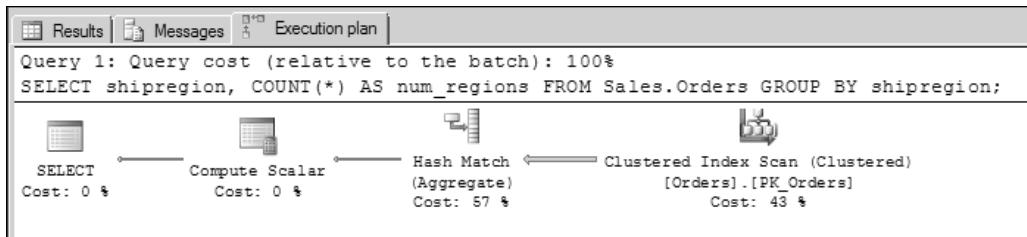


Рис. 15.6. Использование оператора **Hash Match (Aggregate)**, когда статистический запрос не поддерживается индексом

ORDER BY. Но если вы включите это предложение, вам следует рассмотреть также его поддержку индексом. Если подходящего индекса для предложения ORDER BY нет, SQL Server должен отсортировать данные перед их возвращением. Сортировка больших наборов данных может оказаться серьезным ударом по производительности SQL Server. Данные должны быть отсортированы в памяти или сброшены в базу данных tempdb, если они не помещаются в память. Следующий запрос использует предложение ORDER BY, которое не поддержано индексом.

```
SELECT shipregion
FROM Sales.Orders
ORDER BY shipregion;
```

План выполнения для этого запроса включает оператор **Sort** (рис. 15.7).

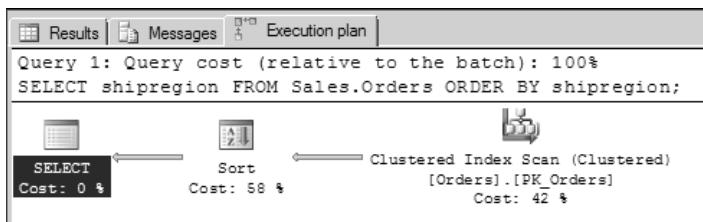


Рис. 15.7. SQL Server, использующий оператор **Sort** для сортировки выходных данных

Выполнение запроса использования индекса возвращает следующий результат:

| table_name | index_name | user_seeks | user_scans | user_lookups |
|------------|------------|------------|------------|--------------|
| Orders     | PK_Orders  | 0          | 4          | 0            |

Все 4 запроса, выполнявшиеся в этом занятии, использовали сканирование кластеризованного индекса. Пора начать поддержку запросов соответствующими индексами. Следующий фрагмент кода создает некластеризованный индекс, используя столбец shipregion.

```
CREATE NONCLUSTERED INDEX idx_nc_shipregion ON Sales.Orders(shipregion);
```

Если вы снова выполните запросы статистической обработки данных и запросы, которые запрашивают отсортированные данные, как показано в следующем примере, вы получите разные планы выполнения.

```
-- Запрос, который статистически обрабатывает данные
SELECT shipregion, COUNT(*) AS num_regions
FROM Sales.Orders
GROUP BY shipregion;
-- Запрос, который сортирует выходные данные
SELECT shipregion
FROM Sales.Orders
ORDER BY shipregion;
```

План выполнения для первого запроса использует оператор **Stream Aggregate** (Статистическое выражение потока), а план выполнения для второго запроса не включает оператор **Sort** (рис. 15.8).

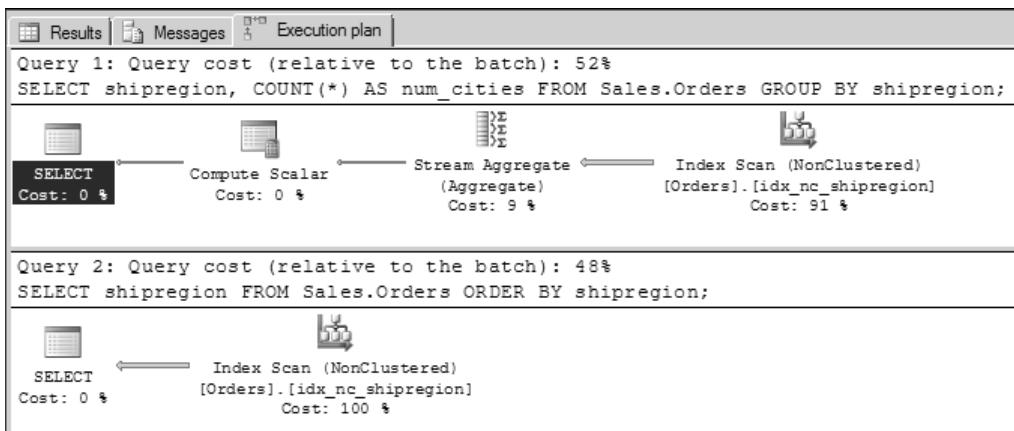


Рис. 15.8. Планы выполнения для запросов GROUP BY и ORDER BY, поддержанные индексом

Выполнение запроса использования индекса возвращает следующий результат:

| table_name | index_name        | user_seeks | user_scans | user_lookups |
|------------|-------------------|------------|------------|--------------|
| Orders     | idx_nc_shipregion | 0          | 2          | 0            |
| Orders     | PK_Orders         | 0          | 4          | 0            |

Обратите внимание, некластеризованный индекс **idx\_nc\_shipregion** использовался для двух просмотров, и никакого другого использования кластеризованного индекса не было. Это также видно из планов выполнения на рис. 15.8. SQL Server нашел все данные для запроса в некластеризованном индексе и не должен был выполнять уточняющий поиск по RID или ключу. Если SQL Server находит все данные в некластеризованных индексах, этот запрос покрывается некластеризованными индексами, а индексы являются покрывающими индексами. Покрытые индексами запросы являются очень эффективными.

Можно добавить дополнительные столбцы в ключ некластеризованного индекса, чтобы покрыть больше запросов. Но при длинном ключе индекс может стать менее эффективным. В SQL Server 2012 есть еще одна возможность. Можно также включить столбец в некластеризованный индекс только на конечном уровне, не как

часть ключа. Это можно сделать с помощью предложения `INCLUDE` инструкции `CREATE INDEX`. Включенный столбец не является частью ключа, и SQL Server не использует его для поиска. Включенные столбцы помогают покрыть запросы. Но следует соблюдать осторожность и не включать слишком много столбцов. Например, если вы включите все столбцы таблицы, вы ее фактически скопируете.

Индекс `idx_nc_shipregion` не будет использоваться в последующих примерах, поэтому его можно удалить с помощью следующего кода:

```
DROP INDEX idx_nc_shipregion ON Sales.Orders;
```

## Аргументы поиска

Включение в запрос предложения `WHERE`, даже если предикат является очень выборочным и поддерживается индексом, не гарантирует, что SQL Server будет использовать индекс. Необходимо написать соответствующий предикат, позволяющий оптимизатору запросов использовать преимущества индексов. Оптимизатор запросов не всемогущ. Он может принять решение использовать индекс только тогда, когда возможен поиск по аргументам в предикате. Вы должны научиться писать подходящие аргументы поиска (search arguments, SARG).

Чтобы написать подходящий аргумент поиска SARG, вы должны быть уверены, что столбец, имеющий индекс, появляется в предикате отдельно, а не как параметр функции. Аргумент поиска SARG должен принимать форму столбца `inclusive_operator <value>` или столбца `<value> inclusive_operator`. Имя столбца должно стоять отдельно на одной стороне выражения, а константа или вычисляемое значение — появляться на другой стороне. В качестве операторов могут использоваться операторы `=`, `>`, `<`, `=>`, `=<`, `BETWEEN` и `LIKE`. Но оператор `LIKE` можно использовать, только если подстановочные символы `%` или `_` не стоят в начале строковой переменной, с которой сравнивается столбец. Например, следующий запрос возвращает заказы для дат 10 июля 2006 и 11 июля 2006.

```
SELECT orderid, custid, orderdate, shipname
FROM Sales.Orders
WHERE DATEDIFF(day, '20060709', orderdate) <= 2
 AND DATEDIFF(day, '20060709', orderdate) > 0;
```

Запрос возвращает только две строки; таким образом, предикат `WHERE` является очень выборочным. На столбце `orderdate` имеется некластеризованный индекс. Но SQL Server его не использовал, как показывает план выполнения на рис. 15.9.

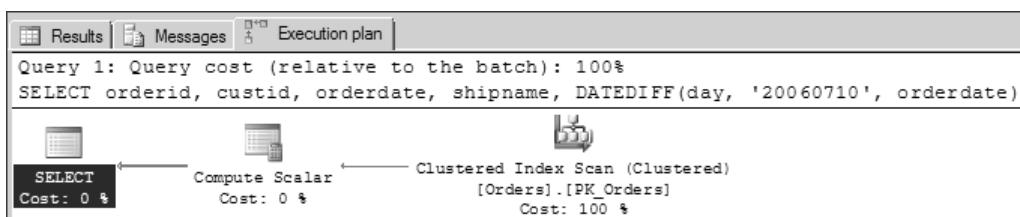


Рис. 15.9. Запрос, в котором предикат не является аргументом поиска SARG

Параметр `orderdate` в предикате не появляется отдельно, а является аргументом функции. Такой запрос можно переписывать много раз. Следующий запрос дает тот же результат, но на этот раз предикат является аргументом поиска SARG.

```
SELECT orderid, custid, orderdate, shipname
FROM Sales.Orders
WHERE DATEADD(day, 2, '20060709') >= orderdate
AND '20060709' < orderdate;
```

Из рис. 15.10 вы видите, что SQL Server использовал индекс `idx_nc_orderdate` для поиска и затем выполнял дополнительный поиск по ключу.

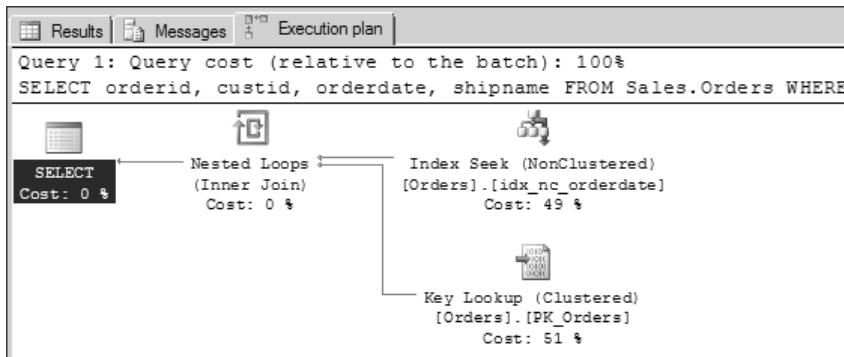


Рис. 15.10. Запрос, в котором предикат является аргументом поиска SARG

Этот запрос можно переписать различными способами. Можно использовать оператор `IN`, чтобы включить список дат, на которые запрос должен извлекать заказы. Также можно использовать оператор эквивалентности для каждой даты и связывать два предиката эквивалентности с помощью логического оператора `OR` (И). В действительности эти два запроса внутренне должны обрабатываться одинаково; оптимизатор запросов конвертирует оператор `IN` в оператор `OR` с отдельным сравнением с каждым элементом списка оператора `IN`. Следующие два запроса возвращают те же две строки и внутренне обрабатываются как эквивалентные.

```
SELECT orderid, custid, orderdate, shipname
FROM Sales.Orders
WHERE orderdate IN ('20060710', '20060711');
```

```
SELECT orderid, custid, orderdate, shipname
FROM Sales.Orders
WHERE orderdate = '20060710'
OR orderdate = '20060711';
```

Вы видите, что SQL Server выполнил оба запроса с помощью одинаковых планов выполнения (рис. 15.11).

Использование оператора `AND` в предикате предложения `WHERE` означает, что каждая часть предиката ограничивает результирующий набор еще больше, чем предыдущая. Например, если первое условие ограничивает запрос до 5 строк, то следующее

условие, соединенное с первым через логический оператор AND, ограничивает запрос не более чем до 5 строк. Оптимизатор запросов понимает, как работает логический оператор AND, и может использовать подходящие индексы.

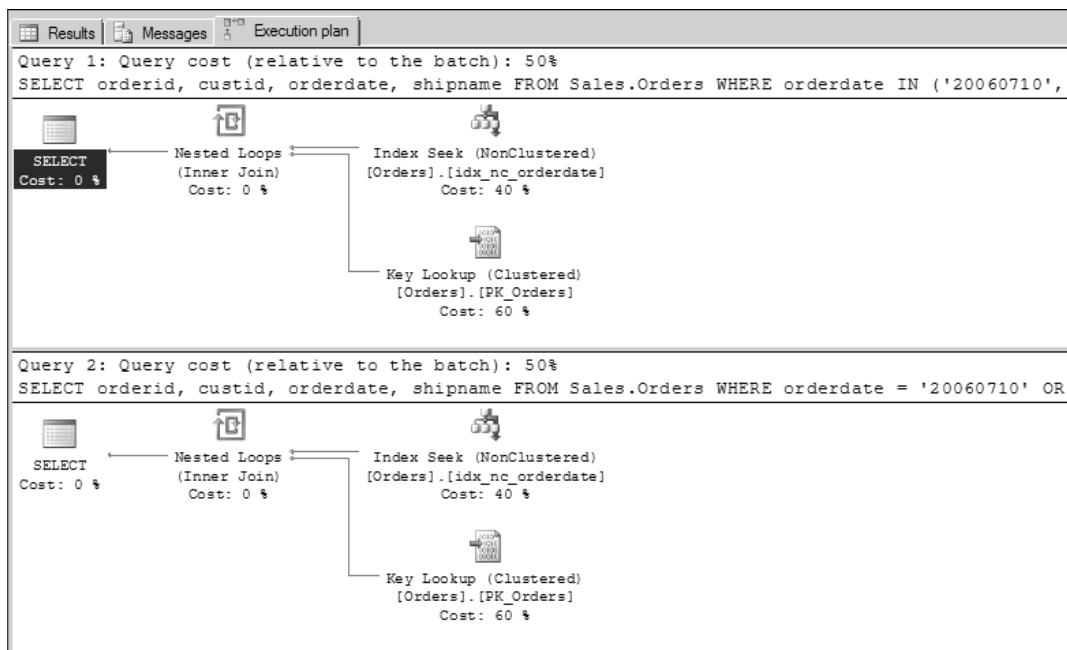


Рис. 15.11. SQL Server выполняет операторы IN и OR одинаково

Но логический оператор OR является включающим. Например, если первое условие в предикате ограничивает запрос до 5 строк, а следующее условие, соединенное с первым с помощью логического оператора OR, ограничивает запрос до 6 строк, результирующий набор будет иметь от 6 до 11 строк. Если два условия используют разные столбцы, тогда SQL Server осмотрительно берет худший вариант и предполагает, что запрос должен возвратить 11 строк. Наличие в предикате нескольких условий, соединенных с помощью оператора OR, снижает возможность для SQL Server использовать индексы. Следует рассмотреть возможность переписать такой предикат на его логический эквивалент, использующий оператор AND.

### КОНТРОЛЬНЫЙ ВОПРОС

- Какие предложения запроса по вашему мнению поддерживаются индексом?

### Ответ на контрольный вопрос

- Список предложений, которые предположительно поддерживаются индексом, включает, но не ограничен предложениями WHERE, JOIN, GROUP BY и ORDER BY.

## ПРАКТИКУМ Использование логических операторов *OR* и *AND*

В этом практикуме вы будете использовать логические операторы *OR* или *AND* для соединения двух условий предиката запроса и затем проверите планы выполнения.

### Задание 1. Поддержка логического оператора *OR*

В этом задании вам нужно проверить влияние логического оператора *OR* на выполнение запроса. Вы планируете поддерживать запросы соответствующими индексами.

- Если вы закрыли SQL Server Management Studio (SSMS), откройте ее и подключитесь к вашему экземпляру SQL Server.
- Откройте новое окно запроса, нажав кнопку **New Query** (Создать запрос).
- Измените контекст на базу данных TSQL2012.
- Создайте некластеризованный индекс на столбце `shipcity` таблицы `Sales.Orders`.

```
CREATE NONCLUSTERED INDEX idx_nc_shipcity ON Sales.Orders(shipcity);
```

- Извлеките столбцы `orderid`, `custid` и `shipcity` для города `Vancouver` (Ванкувер).

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE shipcity = N'Vancouver';
```

- Запрос является выборочным, поскольку он возвращает только три строки. Проверьте, использовался ли только что созданный вами некластеризованный индекс, с помощью следующего кода:

```
SELECT OBJECT_NAME(S.object_id) AS table_name,
 I.name AS index_name, S.user_seeks,
 S.user_scans, S.user_lookups
 FROM sys.dm_db_index_usage_stats AS S
 INNER JOIN sys.indexes AS I
 ON S.object_id = I.object_id
 AND S.index_id = I.index_id
 WHERE S.object_id = OBJECT_ID(N'Sales.Orders', N'U')
 AND I.name = N'idx_nc_shipcity';
```

Вы должны получить одну строку, и это показывает, что индекс использовался для поиска.

- Включите действительный план выполнения. Теперь извлеките те же столбцы для клиента с идентификатором 42.

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE custid = 42;
```

Обратите внимание, возвращены те же три строки. SQL Server на этот раз использовал некластеризованный индекс `idx_nc_custid`, как видно из плана выполнения. Индекс использует столбец `custid` как ключ.

- Снова извлеките тот же результирующий набор. Но теперь используйте оба условия, город Ванкувер и идентификатор клиента 42, в предложении WHERE, соединенные оператором OR, как показано в следующем коде:

```
SELECT orderid, custid, shipcity
FROM Sales.Orders WHERE custid = 42
 OR shipcity = N'Vancouver';
```

- И снова возвращены те же три строки. Но вы видите из плана выполнения, что на этот раз SQL Server просмотрел кластеризованный индекс. Задержав указатель мыши на операторе **Clustered Index Scan** (Просмотр кластеризованного индекса) в плане, вы можете увидеть предполагаемое и действительное число строк (рис. 15.12).

Вы видите, что предполагаемое число строк — около 6, и SQL Server решил просканировать кластеризованный индекс.

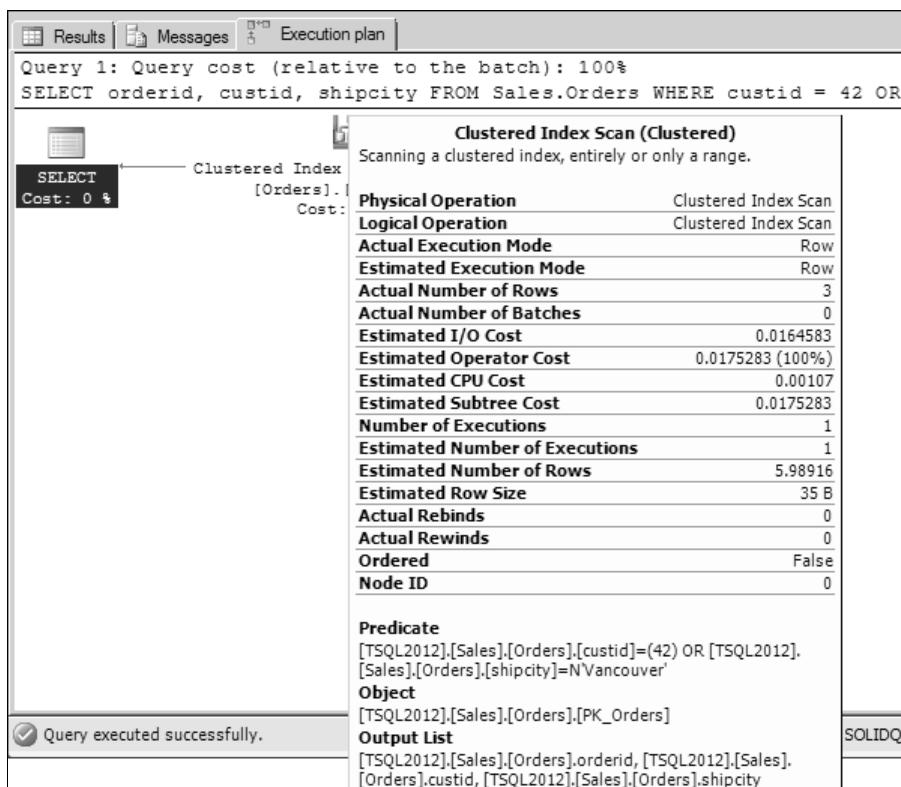


Рис. 15.12. Предполагаемое и действительное число строк оператора **Clustered Index Scan**

## Задание 2. Поддержка логического оператора AND

В этом задании вам нужно проверить влияние логического оператора `AND` на выполнение запроса. Вам также нужно создать индекс, который содержит включенный столбец.

1. Измените последний запрос из предыдущего задания, заменив оператор `OR` на оператор `AND`.

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE custid = 42 AND shipcity = N'Vancouver';
```

Разумеется, вы по-прежнему извлечете те же три строки. Но из плана выполнения вы видите, что SQL Server на этот раз использует некластеризованный индекс `idx_nc_custid`.

2. Удалите некластеризованный индекс на столбце `shipcity` таблицы `Sales.Orders`.

```
DROP INDEX idx_nc_shipcity ON Sales.Orders;
```

3. Создайте некластеризованный индекс на столбце `shipcity` таблицы `Sales.Orders` снова. На этот раз включите столбец `custid`. Используйте следующий код:

```
CREATE NONCLUSTERED INDEX idx_nc_shipcity_i_custid
 ON Sales.Orders(shipcity)
INCLUDE (custid);
```

4. Снова выполните запрос, содержащий оба условия, город Ванкувер и идентификатор клиента 42, в предложении `WHERE`, соединенные оператором `OR`.

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE custid = 42 OR shipcity = N'Vancouver';
```

5. Должно выполниться сканирование некластеризованного индекса. Запрос покрыт индексом с включенным столбцом, который вы только что создали. Но SQL Server опять использовал сканирование, из-за оператора `OR`. Замените оператор `OR` оператором `AND` снова и выполните запрос.

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE custid = 42 AND shipcity = N'Vancouver';
```

На этот раз SQL Server должен использовать оператор **Index Seek** (Поиск в индексе) для поиска первого вхождения Ванкувера, а затем выполнить частичный просмотр.

6. Отключите действительный план выполнения и удалите созданный вами индекс.

```
DROP INDEX idx_nc_shipcity_i_custid ON Sales.Orders;
```

## Резюме занятия

- Разные части запроса могут поддерживаться индексами.
- Следует рассматривать возможность использования предложений запроса WHERE, JOIN, GROUP BY, ORDER BY и SELECT с надлежащими индексами.
- Следует писать подходящие аргументы поиска, не включая в выражения столбцы ключей индексов.

## Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Как можно поддерживать предложение SELECT запроса с помощью некластеризованного индекса, который уже используется для предложения WHERE?
  - A. Нужно использовать SELECT \*.
  - B. Можно модифицировать индекс, который уже используется, включив столбцы из списка SELECT, не являющиеся частью ключа.
  - C. Можно добавить псевдонимы столбцов.
  - D. Предложение SELECT нельзя поддерживать индексами.
2. Если нужна сортировка, где SQL Server сортирует данные?
  - A. В текущей базе данных.
  - B. В главной базе данных.
  - C. В базе данных msdb.
  - D. SQL Server сортирует данные в памяти или сбрасывает их в tempdb, если они не помещаются в памяти.
3. Вы создали индекс для поддержки предложения WHERE запроса. Но SQL Server не использует индекс. В чем заключаются возможные причины? (Выберите все подходящие варианты.)
  - A. По аргументам предиката не может выполняться поиск.
  - B. SQL Server не может использовать индекс для поддержки предложения WHERE.
  - C. Предикат является недостаточно выборочным.
  - D. Вы работаете с базой данных tempdb, и SQL Server не использует индексы в этой базе данных.

## Занятие 3. Основные понятия статистики

Читая и тестируя код из предыдущего занятия, вы, возможно, задавали себе вопрос, откуда SQL Server знает заранее, является ли запрос достаточно выборочным для выполнения поиска? Никакой магии, просто SQL Server поддерживает статистиче-

ские данные о распределении значений ключей в специальных системных статистических страницах. Оптимизатор запросов использует эту статистику для предварительной оценки количества элементов, или числа строк, в результирующем наборе запроса. Вы будете изучать статистику в данном занятии.

**Изучив материал этого занятия, вы сможете:**

- ✓ Сформулировать основы статистики SQL Server
- ✓ Поддерживать статистику вручную

**Продолжительность занятия — 25 минут.**

## Автоматически создаваемая статистика

По умолчанию SQL Server создает статистику автоматически. SQL Server создает статистику для каждого индекса и для отдельных столбцов, используемых как аргументы поиска в запросах. Существуют три параметра баз данных, оказывающие влияние на автоматическое создание статистики.

- AUTO\_CREATE\_STATISTICS.** Когда этот параметр включен, SQL Server создает статистику автоматически. Этот параметр включен по умолчанию и его следует оставлять включенным в подавляющем большинстве случаев.
- AUTO\_UPDATE\_STATISTICS.** Когда этот параметр включен, SQL Server может автоматически обновлять статистику при достаточном количестве изменений в базовых таблицах и индексах. Когда он включен, SQL Server также обновляет представшую быть актуальной статистику в процессе оптимизации запросов. SQL Server проверяет актуальность статистики до компиляции запроса перед выполнением кэшированного запроса. В общем случае тот параметр нужно оставлять установленным.
- AUTO\_UPDATE\_STATISTICS\_ASYNC.** Этот параметр определяет, использует ли SQL Server синхронное или асинхронное обновление статистики, во время оптимизации запроса. Если статистика обновляется асинхронно, SQL Server не может использовать ее для оптимизации запроса, инициировавшего обновление; но SQL Server не ждет обновления статистики на этапе оптимизации. Этот параметр надо включать только в том случае, когда ваши запросы ждут синхронного обновления статистики слишком часто, и это вызывает проблемы производительности. По умолчанию этот параметр выключен и его изменение не имеет никакого значения, пока не включен параметр `AUTO_UPDATE_STATISTICS`.

Каждый объект статистики хранится в большом двоичном объекте статистики и создается на одном или более столбцах. Статистика содержит гистограмму, показывающую распределение значений в первом столбце. Объекты статистики для нескольких столбцов хранят дополнительные статистические сведения о корреляции значений между столбцами. Эти статистические корреляции называют значениями плотности. Они получаются из количества уникальных строк комбинаций значений столбцов составного индекса.

Количество шагов в гистограмме ограничено. Статистика может иметь не более 200 шагов. Объект статистики также имеет заголовок с метаданными о статистике и вектор плотности для измерения корреляции с охватом нескольких столбцов. SQL Server вычисляет предполагаемое число строк, возвращаемых запросом, или предполагаемую мощность, с любыми данными в объекте статистики.

Информацию о статистике можно получить, запросив представления каталога sys.stats и sys.stats\_columns. Команда DBCC SHOW\_STATISTICS позволяет получить подробную информацию о статистике. С помощью команд CREATE, DROP и UPDATE можно поддерживать ручное получение статистики. Также вы можете использовать системную процедуру sys.sp\_updatestats для ручного обновления статистики для всех таблиц в базе данных. Например, следующий код использует курсор на представлении каталога sys.stats, чтобы циклически выполняться на всей автоматически создаваемой статистике для столбцов, которые не используются в качестве индексных ключей для таблицы Sales.Orders, динамически подцепляя команду DROP STATISTICS и удаляет эти статистические данные.

```
DECLARE @statistics_name AS NVARCHAR(128), @ds AS NVARCHAR(1000);
DECLARE acs_cursor CURSOR FOR
SELECT name AS statistics_name
FROM sys.stats
WHERE object_id = OBJECT_ID(N'Sales.Orders', N'U')
 AND auto_created = 1;
OPEN acs_cursor;
FETCH NEXT FROM acs_cursor INTO @statistics_name;
WHILE @@FETCH_STATUS = 0
BEGIN
 SET @ds = N'DROP STATISTICS Sales.Orders.' + @statistics_name +';';
 EXEC(@ds);
 FETCH NEXT FROM acs_cursor INTO @statistics_name;
END;
CLOSE acs_cursor;
DEALLOCATE acs_cursor;
```

### **ВАЖНО!**

### **Обновление статистики для всех таблиц в базе данных**

Использование системной процедуры sys.sp\_updatestats для ручного обновления статистики для всех таблиц в базе данных может потребовать большого количества времени и использовать много ресурсов; поэтому следует соблюдать осторожность при использовании этой команды в большой базе данных. Ее можно использовать только не в часы пиковой нагрузки.

Теперь должна существовать статистика для индексов, как показывает следующий запрос:

```
SELECT OBJECT_NAME(object_id) AS table_name,
 name AS statistics_name, auto_created
 FROM sys.stats
 WHERE object_id = OBJECT_ID(N'Sales.Orders', N'U');
```

Вот как выглядит результат этого запроса:

| table_name | statistics_name       | auto_created |
|------------|-----------------------|--------------|
| Orders     | PK_Orders             | 0            |
| Orders     | idx_nc_custid         | 0            |
| Orders     | idx_nc.empid          | 0            |
| Orders     | idx_nc_shipperid      | 0            |
| Orders     | idx_nc_orderdate      | 0            |
| Orders     | idx_nc.shippeddate    | 0            |
| Orders     | idx_nc.shippostalcode | 0            |

Столбец `auto_created` получает значение 1 для статистики, которую SQL Server генерирует автоматически для одиночных столбцов, используемых в качестве аргумента поиска в процессе выполнения запроса. Прежде чем показать статистику, следующая строка кода перестраивает `idx_nc.empid` и таблицу `Sales.Orders`, чтобы убедиться в том, что SQL Server обновил статистику.

```
ALTER INDEX idx_nc.empid ON Sales.Orders REBUILD;
```

Следующая команда демонстрирует гистограмму статистики `idx_nc.empid`.

```
DBCC SHOW_STATISTICS(N'Sales.Orders',N'idx_nc.empid') WITH HISTOGRAM;
```

Результат команды выглядит следующим образом:

| RANGE_HI_KEY | RANGE_ROWS | EQ_ROWS | DISTINCT_RANGE_ROWS | AVG_RANGE_ROWS |
|--------------|------------|---------|---------------------|----------------|
| 1            | 0          | 123     | 0                   | 1              |
| 2            | 0          | 96      | 0                   | 1              |
| 3            | 0          | 127     | 0                   | 1              |
| 4            | 0          | 156     | 0                   | 1              |
| 5            | 0          | 42      | 0                   | 1              |
| 6            | 0          | 67      | 0                   | 1              |
| 7            | 0          | 72      | 0                   | 1              |
| 8            | 0          | 104     | 0                   | 1              |
| 9            | 0          | 43      | 0                   | 1              |

```
DBCC completed. If DBCC printed error messages, contact your system
execution administrator.
```

В гистограмме только 9 шагов, поскольку столбец `empid` имеет только 9 уникальных значений. Если вы выполните команду `DBCC SHOW_STATISTICS` без параметра `WITH HISTOGRAM`, то получите всю статистическую информацию, включая заголовок и вектор плотности. Из заголовка можно получить такие полезные сведения, как дата последнего обновления статистики, как это делает следующий запрос:

```
DBCC SHOW_STATISTICS(N'Sales.Orders',N'idx_nc.empid') WITH STAT_HEADER;
```

Частичный выход этого запроса (только 7 самых левых столбцов) выглядит следующим образом:

| Name         | Updated            | Rows | Rows Sampled | Steps | Density | Average key length |
|--------------|--------------------|------|--------------|-------|---------|--------------------|
| idx_nc_empid | Mar 24 2012 1:57PM | 830  | 830          | 9     | 0       | 8                  |

Информацию о дате последнего обновления статистики можно получить, используя системную функцию `STATS_DATE()`.

Как уже говорилось, SQL Server автоматически создает статистику для неключевых столбцов поиска в процессе выполнения запроса. Чтобы проверить это, начните со следующего кода, который добавляет некластеризованный индекс в таблицу `Sales.Orders`.

```
CREATE NONCLUSTERED INDEX idx_nc_custid_shipcity
 ON Sales.Orders(custid, shipcity);
```

Следующий запрос извлекает три строки для клиента с идентификатором 42 из таблицы `Sales.Orders`.

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE custid = 42;
```

Теперь проверьте, существует ли автоматически созданная статистика. Следующий запрос возвращает ноль строк, что означает отсутствие автоматически созданной статистики для таблицы `Sales.Orders`. У SQL Server было достаточно информации, полученной через статистику индексов от `idx_nc_custid_shipcity` для выполнения предыдущего запроса.

```
SELECT OBJECT_NAME(object_id) AS table_name,
name AS statistics_name
FROM sys.stats
WHERE object_id = OBJECT_ID(N'Sales.Orders', N'U')
 AND auto_created = 1;
```

Теперь выберите те же три строки из таблицы `Sales.Orders`, но на этот раз используйте столбец `shipcity` как аргумент поиска SARG и ограничьте выходные данные только выборкой для Ванкувера.

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE shipcity = N'Vancouver';
```

Следующий запрос проверяет, существует ли автоматически созданная статистика, и добавляет информацию о столбцах, для которых была создана статистика.

```
SELECT OBJECT_NAME(s.object_id) AS table_name,
 S.name AS statistics_name, C.name AS column_name
FROM sys.stats AS S
INNER JOIN sys.stats_columns AS SC
 ON S.stats_id = SC.stats_id
INNER JOIN sys.columns AS C
 ON S.object_id= C.object_id AND SC.column_id = C.column_id
WHERE S.object_id = OBJECT_ID(N'Sales.Orders', N'U')
 AND auto_created = 1;
```

Этот запрос имеет следующие выходные данные:

| table_name | statistics_name           | column_name |
|------------|---------------------------|-------------|
| Orders     | _WA_Sys_0000000B_20C1E124 | shipcity    |

Вы видите, что SQL Server создал статистику для столбца `shipcity`. Все имена автоматически созданных объектов статистики начинаются строкой `_WA_Sys_`. Очистите данные, удалив созданный для этого теста индекс.

```
DROP INDEX idx_nc_custid_shipcity ON Sales.Orders;
```

Если хотите, можно снова удалить всю созданную автоматически статистику.

## Ручная поддержка статистики

Существует всего несколько причин ручного создания статистики. Например, если предикат запроса содержит несколько столбцов, имеющих перекрестные связи. Статистика по нескольким столбцам способна улучшить план запроса. Она содержит значения плотности, распределенные на несколько столбцов, которые недоступны статистике по одному столбцу. Но если столбцы входят в какой-то индекс, объект статистики для нескольких столбцов уже существует, поэтому вам не нужно создавать дополнительный столбец вручную.

Аналогично фильтруемым индексам, можно создавать фильтруемую статистику. Статистика, создаваемая SQL Server автоматически, всегда формируется по всем строкам таблицы. Если запросы часто делают выборку из поднабора строк, который имеет уникальное распределение строк, фильтруемая статистика может улучшить планы запросов.

Иногда вы можете получать сообщение в плане выполнения о том, что определенная статистика отсутствует. Такую статистику можно создать вручную. Но перед этим необходимо проверить, что параметры базы данных `AUTO_CREATE_STATISTICS` и `AUTO_UPDATE_STATISTICS` включены, а база данных доступна для записи. Если база данных доступна только для чтения, оптимизатор запросов не может сохранить статистику.

Создание статистики вручную можно рассмотреть при следующих обстоятельствах.

- При большом времени выполнения запросов, если вы знаете, что запросы написаны правильно и поддерживаются соответствующими индексами. До использования указаний запроса обновите статистику. SQL Server не использует индекс с неактуальной статистикой. Также проверьте, включено ли автоматическое обновление статистики для вашей базы данных.
- Когда операции вставки происходят на восходящих или убывающих ключевых столбцах. Статистика не обновляется для каждой отдельной строки; поэтому количество вставленных строк может быть слишком маленьким для инициации обновления статистики. Если запросы делают выборку из недавно добавленных строк, у текущей статистики может отсутствовать предварительная оценка ко-

личества элементов для этих новых значений. Кроме того, пакетная вставка строк в таблицу или усечение таблицы может значительно изменить распределение данных. Запросы, выполненные сразу после этих операций, могут получить неоптимальный план выполнения, поскольку статистика еще не была автоматически обновлена.

- После обновления предыдущей версии SQL Server. Статистическая информация может изменяться с новой версией SQL Server; чтобы обезопасить себя, следует обновлять статистику для обновленных баз данных.

#### Контрольный вопрос

- Как можно быстро обновить статистику для целой базы данных после обновления?

#### Ответ на контрольный вопрос

- Необходимо использовать системную процедуру sys.sp\_updatestats.

## ПРАКТИКУМ Ручная поддержка статистики

В данном практикуме вам предстоит вручную поддерживать статистику.

### Задание 1. Запрещение автоматического создания статистики

В этом задании вам нужно запретить автоматическое создание статистики.

1. Если вы закрыли SQL Server Management Studio (SSMS), откройте ее и подключитесь к вашему экземпляру SQL Server.
2. Откройте новое окно запроса, нажав кнопку **New Query** (Создать запрос).
3. Измените контекст на базу данных TSQL2012.
4. Чтобы очистить все данные, удалите автоматически созданную статистику для таблицы Sales.Orders. Используйте следующий код:

```
DECLARE @statistics_name AS NVARCHAR(128), @ds AS NVARCHAR(1000);
DECLARE acs_cursor CURSOR FOR
SELECT name AS statistics_name
FROM sys.stats
WHERE object_id = OBJECT_ID(N'Sales.Orders', N'U')
 AND auto_created = 1;
OPEN acs_cursor;
FETCH NEXT FROM acs_cursor INTO @statistics_name;
WHILE @@FETCH_STATUS = 0
BEGIN
 SET @ds = N'DROP STATISTICS Sales.Orders.' + @statistics_name +';';
 EXEC (@ds);
 FETCH NEXT FROM acs_cursor INTO @statistics_name;
END;
CLOSE acs_cursor;
DEALLOCATE acs_cursor;
```

5. Запретите автоматическое создание статистики для базы данных TSQL2012 с помощью команды ALTER DATABASE.

```
ALTER DATABASE TSQL2012
SET AUTO_CREATE_STATISTICS OFF WITH NO_WAIT;
```

## Задание 2. Наблюдение эффекта от отключения автоматического создания статистики

В этом задании вам нужно проверить, как влияет на использование индексов отключение автоматического создания статистики.

1. Добавьте составной некластеризованный индекс к таблице Sales.Orders, используя столбцы custid и shipcity в качестве ключей.

```
CREATE NONCLUSTERED INDEX idx_nc_custid_shipcity
ON Sales.Orders(custid, shipcity);
```

2. Включите действительный план выполнения. Используйте следующий запрос для выборки трех заказов, у которых параметр shipcity имеет значение Vancouver (Ванкувер).

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE shipcity = N'Vancouver';
```

3. Проверьте план выполнения. Вы должны получить либо оператор **Clustered Index Scan** (Просмотр кластеризованного индекса) со значком предупреждения на нем, либо оператор **Index Scan (NonClustered)** (Просмотр некластеризованного индекса) также со значком предупреждения на нем. Вы можете открыть окно **Properties** (Свойства) в меню **View** (Вид) или нажать клавишу <F4> для проверки свойства предупреждений. Щелкните на многоточии в конце текста свойства **Warnings** (Предупреждения), чтобы получить всплывающее окно для этого свойства (рис. 15.13).

4. Проверьте, существует ли автоматически созданная статистика для таблицы Sales.Orders, используя следующий запрос:

```
SELECT OBJECT_NAME(object_id) AS table_name,
 name AS statistics_name
 FROM sys.stats
 WHERE object_id = OBJECT_ID(N'Sales.Orders', N'U')
 AND auto_created = 1;
```

5. Запрос не должен возвратить никаких строк. Вы можете создать отсутствующую статистику вручную. Кроме создания статистики, вам нужно очистить кэшированный план, чтобы не допустить его повторного использования SQL Server. Используйте следующий код:

```
CREATE STATISTICS st_shipcity ON Sales.Orders(shipcity);
DBCC FREEPROCCACHE;
```

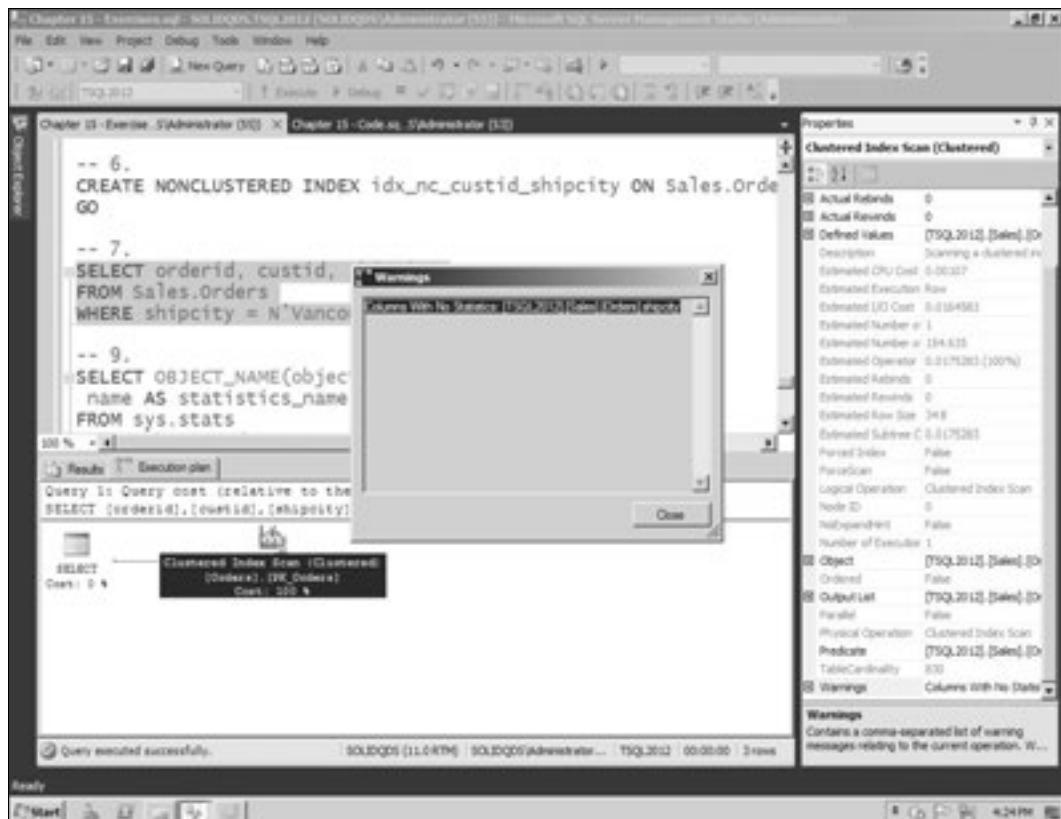


Рис. 15.13. Предупреждение об отсутствующей статистике

6. Выполните снова тот же запрос, который выполняет поиск заказов из Ванкувера.

```
SELECT orderid, custid, shipcity
FROM Sales.Orders
WHERE shipcity = N'Vancouver';
```

7. Проверьте план выполнения. На этот раз предупреждение не должно появиться.

8. Для очистки данных включите автоматическое создание статистики, обновите всю статистику для базы данных TSQL2012 и удалите индекс и статистику, которые вы создали в этом задании. Вы можете использовать следующий код:

```
DROP STATISTICS Sales.Orders.st_shipcity;
DROP INDEX idx_nc_custid_shipcity ON Sales.Orders;
ALTER DATABASE TSQL2012
 SET AUTO_CREATE_STATISTICS ON WITH NO_WAIT;
EXEC sys.sp_updatestats;
```

9. Выйдите из SSMS.

## Резюме занятия

- Оптимизатор запросов SQL Server использует статистику для определения количества элементов в запросе.
- Кроме предоставления права поддержки автоматической статистики SQL Server, можно также поддерживать статистику вручную.

## Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Как может SQL Server рассчитать мощность запроса?
  - A. SQL Server хранит информацию о мощности в конечных страницах индекса.
  - B. SQL Server быстро выполняет запрос на 10% пробных данных.
  - C. SQL Server не может рассчитать мощность запроса, если не представлена табличная подсказка.
  - D. SQL Server использует статистику, чтобы рассчитать мощность запроса.
2. Что из перечисленного ниже не является основанием для ручного обновления статистики?
  - A. Вы только что перестроили индекс.
  - B. Вы выполнили пакетную вставку большого количества данных в таблицу и хотите запросить эту таблицу сразу после вставки.
  - C. Вы обновили базу данных.
  - D. Запросы выполняются медленно, но вы знаете, что запросы написаны правильно и имеют надлежащие индексы.
3. Каково ограничение на количество шагов в статистических гистограммах?
  - A. 200 шагов на гистограмму.
  - B. 200 гистограмм на столбец.
  - C. 200 страниц на гистограмму.
  - D. 200 шагов на гистограмму.

## Упражнения

---

В следующих упражнениях вы примените полученные знания о реализации индексов и статистике. Ответы на вопросы можно найти в *приложении "Ответы" в конце книги*.

### Упражнение 1. Просмотр таблицы

Администраторы баз данных в компании, в которой вы занимаетесь поддержкой баз данных SQL Server, жалуются, что для большинства запросов SQL Server про-

сматривает таблицы целиком, хотя запросы являются выборочными. Производительность недопустимо низкая. Вы должны помочь повысить производительность.

1. Какие физические структуры вам нужно проверить?
2. Нужно ли вам также проверять код?

## Упражнение 2. Медленные обновления

Конечные пользователи в компании, где вы отвечаете за оптимизацию баз данных, жалуются на то, что обновления баз данных происходят очень медленно, даже если обновляется всего 1 строка. Поиск обновленных строк поддерживается соответствующими индексами. Запросы `SELECT` выполняются хорошо. Это именно то, чего вы ожидали, поскольку вы создали некластеризованные индексы на всех столбцах, используемых в этих запросах. Вы должны также повысить производительность базы данных для обновлений.

1. Как вы думаете, что является причиной медленных обновлений?
2. Как вы будете исследовать возможные проблемы?

## Рекомендуемые упражнения

Для успешного освоения экзаменационных тем, представленных в данной главе, выполните следующие задания.

### Узнайте больше об индексах и о том, как статистические данные влияют на выполнение запроса

Правильное индексирование и поддержка статистики — это очень важные задачи администратора баз данных. Получите разобраться в них можно, выполнив два следующих задания.

□ **Задание 1.** Чтобы понять, как статистическая информация влияет на выполнение запросов, создайте тестовую базу данных с тестовой таблицей в ней. Отключите автоматическое создание и поддержку статистики для этой базы данных. Создайте кластеризованный индекс и один или несколько некластеризованных индексов на этой таблице. Заполните таблицу тестовыми данными. Выполните тестовые запросы и проверьте, использовал ли SQL Server индексы. Вручную создайте статистику и выполните эти запросы снова. Проверьте, использовал ли SQL Server индексы на тот раз. Добавьте строки в таблицу и выполните эти запросы еще раз. После проверки использования индексов вручную обновите статистику. Выполните запросы последний раз и проверьте использование индексов.

□ **Задание 2.** Создайте таблицу с не менее чем 10 столбцами. Вставьте 1 млн строк в цикле и измерьте время, потребовавшееся на эту вставку. Создайте некластеризованный индекс на каждом столбце. Вставьте 1 млн строк в цикле и измерьте время, потребовавшееся на эту вставку. Вы должны заметить разницу и понять, что поддержка индексов требует некоторых ресурсов SQL.

# ГЛАВА 16

## Основные сведения о курсорах, наборах данных и временных таблицах

### Темы экзамена

- Устранение неполадок и оптимизация.
  - Сравнительная оценка построчных операций и операций на основе наборов.

В этой главе рассматриваются две основные темы. Первое занятие посвящено различиям между построчными операциями и операциями на основе наборов. Во втором занятии мы рассмотрим использование временных объектов, таких как локальные временные таблицы и табличные переменные, и объясним, когда следует применять каждый из этих объектов.

### ПРЕЖДЕ ВСЕГО

Для полного освоения этой главы необходимо иметь:

- опыт работы с SQL Server Management Studio (SSMS);
- доступ к экземпляру SQL Server 2012 с установленной учебной базой данных TSQL2012;
- понимание фильтрации и сортировки данных;
- представление о комбинировании наборов данных;
- представление о группировании и оконных функциях;
- общие сведения о создании таблиц и обеспечении целостности данных;
- представление о модификации данных;
- общее представление об индексировании.

# Занятие 1. Сравнительная оценка использования решений на основе курсоров/итераций и решений на основе наборов данных

В этом занятии описываются различия между итерационными решениями и решениями на основе наборов данных при работе с запросами. Если вы не новичок в T-SQL, то могли слышать рекомендации использовать решения на основе наборов вместо итерационных решений. В этом занятии мы объясним, что такое решение на основе наборов и почему их стоит использовать. Также мы расскажем о ситуациях, в которых следует рассмотреть возможность применения итерационных конструкций.

## Изучив материал этого занятия, вы сможете:

- ✓ Оценить использование итерационных решений для операций, которые должны выполняться для каждой строки
- ✓ Использовать курсоры для выполнения построчных операций
- ✓ Выполнять построчные операции без помощи курсоров
- ✓ Описать, почему решения на основе наборов для выполняющих запросы задач, как правило, предпочтительны по сравнению с итерационными решениями

Продолжительность занятия — 40 минут.

## Значение выражения "на основе наборов"

Термин "*на основе набора*" используется для описания подхода в выполнении связанных с запросами задач и основывается на принципах реляционной модели. Напомним, что реляционная модель частично основана на математической теории множеств. Решения на основе наборов используют запросы T-SQL, которые выполняют операции на вводных таблицах как наборах строк. Такие решения отличаются от итерационных решений, которые используют курсоры или другие итерационные конструкции для обработки по одной строке за один раз.



В соответствии с теорией множеств, набор должен рассматриваться как единое целое. Это означает, что ваше внимание должно быть сосредоточено на целом наборе, а не на отдельных его элементах. Что касается итерационных решений, этот принцип разбивается на выполнение операций на одном элементе (строке) за один раз. Кроме того, у набора нет определенного порядка расположения элементов. Поэтому при использовании решений на основе наборов данных вы не можете знать, в какой последовательности расположены данные. Аналогично, пока вы не добавите в запрос предложение `ORDER BY`, вы не можете быть уверены, что данные будут возвращены в каком-то определенном порядке. Используя итерационные решения, вы обрабатываете одну строку за один раз и можете делать это в определенном порядке.

Как уже говорилось, обычно рекомендуется по умолчанию использовать решения на основе наборов, оставляя итерационные решения только для исключительных случаев. Одна из причин такой рекомендации заключается в том, что, как говорилось в главе 1, теория множеств лежит в основе реляционной модели, которая, в свою очередь, положена в основу языка SQL — стандартного языка, на котором базируется язык T-SQL. Используя итерационные решения, вы противоречите принципам основ этого языка.

При использовании решений на основе наборов данных вы представляете свой запрос как запрос на простом декларативном языке. В своем запросе вы фокусируетесь на части "что", предоставляя заботу о части "как" компоненту Database Engine. В случае итерационного решения вы должны реализовать обе эти части в своем коде. В результате итерационные решения обычно бывают более длинными, чем основанные на наборах решения, их сложнее выполнять и поддерживать.

Другая причина придерживаться решений на основе наборов весьма прагматична — это производительность. Итерационные конструкции в T-SQL являются очень медленными. Начнем с того, что циклы в T-SQL работают значительно медленнее, чем в других языках программирования, таких как Microsoft .NET. Кроме того, получение строки из курсора с помощью команды `FETCH NEXT` имеет достаточно большие дополнительные расходы. Таких расходов нет, когда SQL Server обрабатывает решение на основе набора, даже если план выполнения для запроса использует итерации. В результате, если вы умеете настраивать запросы, вы часто сможете достигнуть значительно более высокой производительности, чем при использовании итерационных решений. Это будет показано в примере далее в этом занятии.

Нужно отметить, что существуют исключительные случаи, когда итерационные решения выполняются лучше, чем решения, основанные на наборах — даже при всех дополнительных затратах на построчные операции. Это может произойти, если оптимизатору не удается создать эффективный план запроса, и вы не можете найти способ настроить запрос. При работе с итерационными решениями у вас больше возможностей управления запросом, поскольку вы отвечаете за часть "как". Поэтому, если вы хорошо понимаете, как эффективно обрабатывать данные по одной строке за один раз, иногда вы сможете достигнуть лучших результатов, чем способен найти оптимизатор для решения на основе набора. Но, тем не менее, это исключения из правил, а не норма.

## **Итерации для операций, которые должны выполняться построчно**

Следует понимать, что некоторые задачи просто обязательно нужно обработать с помощью итерационных решений. Рассмотрим задачи управления, которые должны быть выполнены для каждого отдельного объекта в наборе, например в наборе баз данных, таблиц или индексов. Вы должны запросить представление каталога или другой системный объект, чтобы возвратить нужный набор объектов, просмотреть результирующие строки по одной за раз и затем выполнить требуемую задачу для каждого объекта. Примером такой задачи управления может слу-

жить восстановление индексов, имеющих более высокий уровень фрагментации, чем определенный вами процент.

В качестве другого примера задачи, требующей итерационного решения, предположим, что вы имеете хранимую процедуру, которая выполняет некоторую работу для входного клиента. В этой работе участвует несколько инструкций, реализованных в теле процедуры. Логика не может быть реализована для нескольких клиентов за один раз; она должна быть реализована только для одного клиента.

Следующий код определяет такую процедуру, имеющую имя Sales.ProcessCustomer.

```
USE TSQSL2012;
IF OBJECT_ID('Sales.ProcessCustomer') IS NOT NULL
 DROP PROC Sales.ProcessCustomer;
GO
CREATE PROC Sales.ProcessCustomer (@custid AS INT) AS
 PRINT 'Processing customer ' + CAST(@custid AS VARCHAR(10));
GO
```

Инструкция PRINT представляет часть, которая, как правило, реализует работу для входного клиента. Теперь представьте, что вам нужно написать код, который выполняет хранимую процедуру для каждого клиента из таблицы Sales.Customers. Вы должны пройти через строки клиентов по одной за один раз, получить идентификатор клиента и выполнить процедуру с этим идентификатором в качестве входных данных.

Вы можете реализовать такое решение с помощью курсора. Сначала нужно использовать команду DECLARE для объявления курсора на основе запроса, который возвращает все идентификаторы клиентов из таблицы Sales.Customers. Вы можете использовать параметр FAST\_FORWARD, чтобы сделать курсор однодirectionalным и доступным только для чтения.

## **К СВЕДЕНИЮ Типы и параметры курсоров**

Дополнительную информацию о типах и параметрах курсоров можно найти в электронной документации к SQL Server 2012 в разделе "DECLARE CURSOR (Transact-SQL)" по адресу <http://msdn.microsoft.com/ru-ru/library/ms180169.aspx>.

Затем вам нужно использовать команду OPEN, чтобы открыть курсор. Далее следует применить команду FETCH NEXT, чтобы выбрать идентификатор клиента из первой записи курсора в переменную. Затем вы должны пройти по записям курсора с помощью цикла WHILE, когда функция @@FETCH\_STATUS возвращает 0. Возможные значения, возвращаемые этой функцией: 0, если предыдущая выборка была выполнена успешно; -1, если строка оказалась вне результирующего набора; -2, когда выбранная строка отсутствует. На каждой итерации цикла выполняется процедура Sales.ProcessCustomer с использованием текущего идентификатора клиента в качестве входного параметра, а затем выбирается следующая запись из курсора. После завершения цикла вы должны использовать команду CLOSE, чтобы закрыть курсор, и команду DEALLOCATE, чтобы освободить используемые курсором ресурсы. Далее приведен код, полностью реализующий это решение.

```
SET NOCOUNT ON;
DECLARE @curcustid AS INT;
DECLARE cust_cursor CURSOR FAST_FORWARD FOR
 SELECT custid
 FROM Sales.Customers;
OPEN cust_cursor;
FETCH NEXT FROM cust_cursor INTO @curcustid;
WHILE @@FETCH_STATUS = 0
BEGIN
 EXEC Sales.ProcessCustomer @custid = @curcustid;
 FETCH NEXT FROM cust_cursor INTO @curcustid;
END;
CLOSE cust_cursor;
DEALLOCATE cust_cursor;
GO
```

**СОВЕТ****Подготовка к экзамену**

Во время экзамена убедитесь в том, что внимательно читаете и точно понимаете, каковы требования вопросов. Довольно просто попасть в ловушку, пытаясь выбрать ответ, представляющий лучший или самый эффективный способ достижения поставленной задачи, но это не всегда предмет вопроса. Возможно, правильный ответ на вопрос заключается не в самом эффективном способе решения задачи.

Ту же задачу можно решить с помощью другого итерационного решения, не использующего курсор. Можно применить запрос с параметром `TOP (1)` и сортировкой по столбцу `custid` для возвращения минимального значения идентификатора клиента. Затем надо выполнить цикл, пока последний запрос не возвратит значение `NULL`. При каждом проходе цикла нужно выполнять хранимую процедуру, используя в качестве входного параметра текущий идентификатор клиента. Чтобы получить следующий идентификатор клиента, нужно выполнить запрос с параметром `TOP (1)`, где значение столбца `custid` больше, чем предыдущее, отсортированный по столбцу `custid`. Вот как выглядит законченное решение.

```
SET NOCOUNT ON;
DECLARE @curcustid AS INT;
SET @curcustid = (SELECT TOP (1) custid
 FROM Sales.Customers
 ORDER BY custid);
WHILE @curcustid IS NOT NULL
BEGIN
 EXEC Sales.ProcessCustomer @custid = @curcustid;
 SET @curcustid = (SELECT TOP (1) custid
 FROM Sales.Customers
 WHERE custid > @curcustid
 ORDER BY custid);
END;
GO
```

Можно подумать, что последнее решение основано на наборах, поскольку в нем нет явного объявления и использования объекта курсора. Но вспомним, что одним из принципов, реализуемых решениями на основе наборов, является то, что они обрабатывают набор как единое целое в противоположность обработке одного элемента за один раз. Здесь этот принцип нарушен. Кроме того, решения на основе наборов не полагаются на порядок сортировки данных, а это решение полагается.

С точки зрения производительности, первое решение (с использованием курсора) не нуждается в специальных индексах для его поддержки. Второе решение (без использования курсора) в них нуждается. Индекс нужен на столбце `custid`. Если вы не создадите его, каждый запрос с параметром `TOP` будет заканчиваться просмотром всех строк таблицы и применять оператор `TOP N Sort` в плане. Другими словами, без индекса второе решение будет выполняться плохо. Даже при наличии индекса второе решение требует большего количества операций ввода-вывода, чем первое, поскольку оно должно выполнять операцию поиска в индексе построчно.

Можно рассмотреть другой вариант; для извлечения минимального и максимального значения идентификатора клиента из таблицы создайте цикл, который увеличивает текущий идентификатор клиента на 1 до тех пор, пока он не станет равным максимальному значению. Если между существующими идентификаторами клиентов будут возникать разрывы (из-за удалений или в связи с генерацией ключей), ваш код придет к попыткам обработки идентификаторов клиентов, не существующих в вашей таблице. Таким образом, этот запрос не рекомендуется, даже если в данный момент нет разрывов между ключами.

## Сравнение курсора и решений на основе наборов в задачах манипулирования данными

Как уже говорилось ранее, чаще всего при работе с запросами нужно использовать решения на основе наборов и рассматривать итерационные решения только как исключения. Мы уже говорили о том, что решения на основе наборов имеют тенденцию быть более точными и, как правило, обеспечивают более высокую производительность. В этом разделе показано, как решить связанную с запросами задачу, используя оба подхода.

Для генерации демонстрационных данных для задачи, которая будет решаться в этом разделе, вам нужно использовать вспомогательную функцию `GetNums`, которая принимает на вход два целочисленных значения `@low` и `@high` и возвращает результирующий набор, содержащий последовательность целых чисел в интервале между этими двумя входными значениями. Используйте следующий код для определения функции `GetNums`:

```
IF OBJECT_ID('dbo.GetNums', 'IF') IS NOT NULL DROP FUNCTION dbo.GetNums;
GO
CREATE FUNCTION dbo.GetNums(@low AS BIGINT, @high AS BIGINT) RETURNS TABLE
AS
RETURN
```

```

GO
WITH
 L0 AS (SELECT c FROM (VALUES(1),(1)) AS D(c)),
 L1 AS (SELECT 1 AS c FROM L0 AS A CROSS JOIN L0 AS B),
 L2 AS (SELECT 1 AS c FROM L1 AS A CROSS JOIN L1 AS B),
 L3 AS (SELECT 1 AS c FROM L2 AS A CROSS JOIN L2 AS B),
 L4 AS (SELECT 1 AS c FROM L3 AS A CROSS JOIN L3 AS B),
 L5 AS (SELECT 1 AS c FROM L4 AS A CROSS JOIN L4 AS B),
 Nums AS (SELECT ROW_NUMBER() OVER(ORDER BY (SELECT NULL))
 AS rownum FROM L5)
SELECT @low + rownum - 1 AS n
FROM Nums
ORDER BY rownum
OFFSET 0 ROWS FETCH FIRST @high - @low + 1 ROWS ONLY;
GO

```

Задача в этом разделе включает выполнение запросов к таблице с именем `Transactions`, в которой хранится информация о транзакциях по банковским счетам. В таблице имеются три столбца: `actid` (идентификатор счета, account ID), `tranid` (увеличивающийся идентификатор транзакции) и `val` (сумма операции — положительная для вклада и отрицательная для снятия). Следующий код создает таблицу `Transactions` и заполняет ее 1 млн строк, представляющих 100 счетов, каждый с 10 тыс. транзакций.

```

IF OBJECT_ID('dbo.Transactions', 'U') IS NOT NULL DROP TABLE dbo.Transactions;
CREATE TABLE dbo.Transactions
(
 actid INT NOT NULL, -- столбец секционирования
 tranid INT NOT NULL, -- столбец сортировки
 val MONEY NOT NULL, -- сумма
 CONSTRAINT PK_Transactions PRIMARY KEY(actid, tranid));
DECLARE
 @num_partitions AS INT = 100,
 @rows_per_partition AS INT = 10000;

TRUNCATE TABLE dbo.Transactions;
INSERT INTO dbo.Transactions WITH (TABLOCK) (actid, tranid, val)
 SELECT NP.n, RPP.n,
 (ABS(CHECKSUM(NEWID())%2)*2-1) * (1 + ABS(CHECKSUM(NEWID())%5))
 FROM dbo.GetNums(1, @num_partitions) AS NP
 CROSS JOIN dbo.GetNums(1, @rows_per_partition) AS RPP;
GO

```

Ваша задача — реализовать решение, которое вычисляет по каждой транзакции остаток на счете на данный момент. Остаток на счете после конкретной транзакции вычисляется, как нарастающий итог всех значений транзакции с начала деятельности по счету и до текущей транзакции включительно.

Следующий код реализует итерационное решение с использованием курсора.

```
SET NOCOUNT ON;
DECLARE @Result AS TABLE
(actid INT,
 tranid INT,
 val MONEY,
 balance MONEY);
DECLARE
 @actid AS INT,
 @prvactid AS INT,
 @tranid AS INT,
 @val AS MONEY,
 @balance AS MONEY;
DECLARE C CURSOR FAST_FORWARD FOR
 SELECT actid, tranid, val
 FROM dbo.Transactions
 ORDER BY actid, tranid;
OPEN C
FETCH NEXT FROM C INTO @actid, @tranid, @val;
SELECT @prvactid = @actid, @balance = 0;
WHILE @@fetch_status = 0
BEGIN
 IF @actid <> @prvactid
 SELECT @prvactid = @actid, @balance = 0;
 SET @balance = @balance + @val;
 INSERT INTO @Result VALUES(@actid, @tranid, @val, @balance);
 FETCH NEXT FROM C INTO @actid, @tranid, @val;
END
CLOSE C;
DEALLOCATE C;
SELECT * FROM @Result;
GO
```

Курсор построен на запросе, который возвращает строки из таблицы `Transactions`, отсортированные по столбцам `actid` и `tranid`. Код проходит по транзакциям по одной за один раз. Пока значение `actid` не изменяется, код добавляет значение текущей транзакции к накопленному на данный момент в переменной `@balance` и сохраняет строку с информацией о данной транзакции и балансом в табличной переменной `@Result`. Если текущее значение `actid` отличается от предыдущего (это означает, что текущая транзакция — первая для нового счета), переменная `@balance` сбрасывается в 0. Когда код выполнит все итерации, он запрашивает табличную переменную `@Result` на предмет возвращения всех транзакций вместе с вычисленными остатками после каждой транзакции.

Из-за медленности итераций в T-SQL и высокими затратами на извлечение каждой строки из курсора на выполнение этого кода в системе, в которой он был запущен, потребовалось 66 секунд. И это после того, как вывод на печать для строк выходно-

го набора был подавлен с помощью щелчка правой кнопкой мыши на пустом пространстве в панели запроса, выбора команд **Query Options | Results | Grid** (Параметры запроса | Результаты | Сетка) и установки параметра **Discard Results After Execution** (Отбросить результаты после выполнения).

Далее приведено решение на основе наборов для той же задачи с использованием расширенных оконных функций статистической обработки в SQL Server 2012, описанных в *главе 5*.

```
SELECT actid, tranid, val,
 SUM(val) OVER(PARTITION BY actid
 ORDER BY tranid
 ROWS UNBOUNDED PRECEDING) AS balance
 FROM dbo.Transactions;
```

Статистическая оконная функция `SUM` вычисляет значение всех транзакций, находящихся в одном счете (предложение `PARTITION BY actid`), начиная с первой транзакции в счете и до текущей транзакции (предложение `ORDER BY tranid ROWS UNBOUNDED PRECEDING`). Интересно то, что этот запрос оптимизирован с помощью единственного просмотра данных. Он не требует больших дополнительных затрат, связанных с решением на основе курсора. Этот запрос выполнился за 4 секунды в системе, на которой он был запущен.

Ключ к эффективности этого решения — в разнице между строками, которые должны обрабатываться теоретически, и тем, что оптимизатор запросов SQL Server делает на практике. Теоретически, для каждой строки оконная функция генерирует фрейм (рамку) из всех строк, представляющих транзакции, принадлежащие одному счету, с начала деятельности со счетом и до текущей транзакции. Но на практике оптимизатор обнаруживает, что он может просто один раз просмотреть данные, и для того чтобы вычислить текущий нарастающий итог, он может просто добавить значение текущей строки к нарастающему итогу, который был подсчитан для предыдущей строки. Это означает, что данное решение имеет линейное масштабирование. То есть если число строк на один счет увеличивается на фактор  $f$ , количество работы также увеличивается на фактор  $f$ . Таким образом, время выполнения будет изменяться по такому же принципу.

Следует отметить, что некоторые решения на основе наборов не имеют такого хорошего масштабирования. Например, рассмотрите следующее решение на основе наборов для той же задачи.

```
SELECT T1.actid, T1.tranid, T1.val,
 SUM(T2.val) AS balance
 FROM dbo.Transactions AS T1
 JOIN dbo.Transactions AS T2
 ON T2.actid = T1.actid
 AND T2.tranid <= T1.tranid
 GROUP BY T1.actid, T1.tranid, T1.val;
```

Теоретически это решение сопоставляет каждой строке из экземпляра таблицы `Transactions` с псевдонимом `T1` все строки, которые имеют то же значение `actid`,

что и текущая строка, и значение `tranid`, которое меньше или равно текущей строке. Проблема в том, что в отличие от решения с использованием оконной функции, план для этого запроса также физически обрабатывает это множество строк. Здесь оптимизатор не использует ускоренную оптимизацию, которая просматривает данные только один раз. Вы заканчиваете квадратичным ( $N^2$ ) масштабированием. То есть если число строк на один счет увеличивается на фактор  $f$ , количество работы увеличивается на фактор  $f^2$ , и аналогично возрастает время выполнения. На выполнение этого запроса понадобилось 46 минут и 53 секунды.

Решение с использованием оконной функции не поддерживалось выпусками до SQL Server 2012. Поэтому в предыдущих версиях решение на основании курсора в действительности быстрее, если только число строк на один счет не очень мало. Но в SQL Server 2012 решение на основе наборов с использованием оконной функции значительно быстрее и выполняется по линейному закону, поэтому оно является предпочтительным.

### КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие команды требуются для работы с курсором?
2. Что означает использование параметра `FAST_FORWARD` в команде объявления курсора с точки зрения свойств курсора?

### Ответы на контрольные вопросы

1. `DECLARE, OPEN, FETCH в цикле, CLOSE и DEALLOCATE.`
2. `Что этот курсор является однодirectional и доступным только для чтения.`

## ПРАКТИКУМ Сравнительная оценка решений на основе курсора и решений на основе наборов данных

В этом практикуме вам нужно дать сравнительную оценку использования решений на основе курсора/итераций и решений на основе наборов данных.

### Задание 1. Расчет статистического выражения с помощью курсора

В этом задании вы будете использовать таблицу `dbo.Transactions` из данного занятия. Вам нужно проанализировать производительность вычисления максимального значения (столбец `val`) по отдельному счету (столбец `actid`) с использованием решения на основе курсора.

1. Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
2. Создайте индекс на столбцах `actid` и `val` для поддержки вычисления максимального статистического значения.

```
USE TSQL2012;
CREATE INDEX idx_actid_val ON dbo.Transactions(actid, val);
```

3. Напишите решение на основе курсора для выполнения поставленной задачи. Вы можете использовать пример курсора, приведенный в данном занятии, в качестве основы, чтобы пройти все строки в таблице Transactions. Но на этот раз вам нужно рассчитать максимальное значение транзакции на один счет. Ваше решение должно выглядеть следующим образом:

```

SET NOCOUNT ON;
DECLARE @Result AS TABLE
(actid INT,
 mx MONEY);
DECLARE
 @actid AS INT,
 @val AS MONEY,
 @prevactid AS INT,
 @prevval AS MONEY;
DECLARE tx_cursor CURSOR FAST_FORWARD FOR
 SELECT actid, val
 FROM dbo.Transactions
 ORDER BY actid, val;
OPEN tx_cursor;
FETCH NEXT FROM tx_cursor INTO @actid, @val;
SELECT @prevactid = @actid, @prevval = @val;
WHILE @@FETCH_STATUS = 0
BEGIN
 IF @actid <> @prevactid
 INSERT INTO @Result(actid, mx) VALUES(@prevactid, @prevval);
 SELECT @prevactid = @actid, @prevval = @val;
 FETCH NEXT FROM tx_cursor INTO @actid, @val;
END
IF @prevactid IS NOT NULL
 INSERT INTO @Result(actid, mx) VALUES(@prevactid, @prevval);
CLOSE tx_cursor;
DEALLOCATE tx_cursor;
SELECT actid, mx
FROM @Result;
GO

```

4. Отметьте время выполнения этого решения в вашей системе. На тестовом компьютере это решение выполнилось за 21 секунду. Большая часть времени выполнения была затрачена на итерации и издержки курсора.

## **Задание 2. Расчет статистического выражения с помощью решения на основе наборов данных**

Здесь вам нужно решить ту же задачу, что и в предыдущем задании, но на этот раз используя решение на основе наборов данных.

1. Напишите решение на основе наборов для задачи, представленной в предыдущем задании. Решение на основе наборов в этом случае — это очень простой группирующий запрос.

Ваш код должен выглядеть следующим образом.

```
SELECT actid, MAX(val) AS mx
FROM dbo.Transactions
GROUP BY actid;
```

2. Отметьте время выполнения решения на основе наборов в вашей системе и сравните его с производительностью решения на основе курсора. На тестовом компьютере это решение выполнилось меньше чем за 1 секунду. Напомним, что решение на основе курсора выполнилось за 21 секунду.

## Резюме занятия

- Вы можете использовать один из двух основных подходов к решению связанных с запросами задач; один — использование решения на основе наборов данных, другой — использование итерационных решений.
- Решения на основе наборов используют SQL-запросы, которые следуют принципам реляционной модели. Они взаимодействуют с входными таблицами (наборами) как с единым целым, в отличие от построчного взаимодействия. Они также не ожидают, что данные будут приниматься или возвращаться в определенном порядке.
- Некоторые задачи должны решаться с помощью итерационных решений, к ним относятся задачи управления, которые должны выполняться для каждого объекта, или хранимые процедуры, которые нужно выполнять в таблице построчно.
- Что касается связанных с запросами задач, обычно по умолчанию рекомендуется использовать решения на основе наборов, а итерационные решения применять только в исключительных случаях.

## Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. При извлечении строки из курсора как можно определить, что больше нет строк для извлечения?
  - А. Когда функция @@FETCH\_STATUS возвращает 0.
  - Б. Когда функция @@FETCH\_STATUS возвращает -1.
  - С. Когда функция @@FETCH\_STATUS возвращает -2.
  - Д. Когда функция @@FETCH\_STATUS генерирует ошибку.
2. Почему важно предпочитать решения на основе наборов итерационным решениям для связанных с запросами задач? (Выберите все подходящие варианты.)
  - А. Потому что решения на основе наборов основываются на реляционной модели, которая положена в основу языка T-SQL.

- В. Потому что решения на основе наборов всегда обеспечивают более высокую производительность, чем итерационные решения.
- С. Потому что решения на основе наборов обычно требуют написания меньшего количества кода, чем итерационные решения.
- Д. Потому что решения на основе наборов дают возможность опереться на порядок данных.
3. Когда вам нужно обрабатывать одну строку за один раз, что вы можете использовать в качестве альтернативы курсора?
- А. Использовать конструкцию цикла FOR EACH.
- Б. Извлечь минимальный и максимальный ключи, а затем выполнить цикл со счетчиком, который начинает с минимального значения и увеличивается на 1 в каждой итерации, пока не достигнет максимума.
- С. Использовать запрос TOP (1), упорядоченный по ключу, для выбора первой строки. Затем использовать цикл, пока последний возвращенный ключ не равен NULL. В каждой итерации цикла обрабатывать текущую строку и затем использовать запрос TOP (1) с ключом, большим последнего значения, отсортированный по этому ключу, для выборки следующей строки.
- Д. Определить построчный триггер SELECT.

## **Занятие 2. Сравнение использования временных таблиц и табличных переменных**

SQL Server поддерживает множество вариантов, которые вы можете использовать для временного хранения данных. Вы можете использовать временные таблицы и табличные переменные. В данном занятии описываются эти типы объектов и различия между ними.

### **Изучив материал этого занятия, вы сможете:**

- ✓ Описать разницу между временными таблицами и табличными переменными с точки зрения области действия
- ✓ Описать поддержку временных объектов языком DDL и индексами
- ✓ Описать физическое представление временных таблиц, табличных переменных и табличных выражений
- ✓ Описать транзакционную поддержку временных объектов
- ✓ Описать, как обрабатывается статистика временных объектов

**Продолжительность занятия — 40 минут.**

## **Область действия**

SQL Server поддерживает два типа временных таблиц — локальные и глобальные, а также табличные переменные. В этом разделе рассматривается разница между этими тремя типами объектов с точки зрения области действия, или видимости.

Локальные временные таблицы имеют имена, начинающиеся с префикса в виде одного символа решетки #, например #T1. Они видны только создавшему их сеансу. Разные сеансы в действительности могут создавать временные таблицы с одинаковыми именами, и каждый сеанс будет видеть только собственную таблицу. SQL Server добавляет внутренние уникальные суффиксы, чтобы эти имена были уникальными в базе данных, но это прозрачно для сеанса.

Локальные временные таблицы видны всему уровню, который их создал, в пакетах и во всех внутренних уровнях стека вызовов. Поэтому, если вы создадите временную таблицу на конкретном уровне в коде и затем выполните динамический пакет или хранимую процедуру, внутренний пакет может иметь доступ к временной таблице. Если временную таблицу не удалить явно, она будет удалена после завершения создавшего ее уровня.

Имена глобальных временных таблиц имеют префикс в виде двух знаков решетки ##, например ##T1. Они видны всем сессиям. Они удаляются при завершении создавшего их сеанса и при условии, что активные ссылки на них отсутствуют.

Табличные переменные объявляются, а не создаются. Их имена имеют префикс в виде знака at (@), например @T1. Они видны только пакету, который их объявил, и уничтожаются автоматически в конце пакета. Они не видны другим пакетам на том же уровне, а также не видны внутренним уровням в стеке вызова.

Следующий код демонстрирует, что локальные временные таблицы видны пакетам одного уровня, а также внутренним уровням в стеке вызовов.

```
CREATE TABLE #T1
(col1 INT NOT NULL);
INSERT INTO #T1(col1) VALUES(10);
EXEC('SELECT col1 FROM #T1;');
GO
SELECT col1 FROM #T1;
GO
DROP TABLE #T1;
GO
```

Код выполняется успешно, без ошибок.

Следующий код демонстрирует, что табличные переменные не видны внутренним уровням в стеке вызовов.

```
DECLARE @T1 AS TABLE
(col1 INT NOT NULL);
INSERT INTO @T1(col1) VALUES(10);
EXEC('SELECT col1 FROM @T1;');
GO
```

Попытка сослаться на табличную переменную из динамического пакета генерирует следующую ошибку:

```
Msg 1087, Level 15, State 2, Line 1
Must declare the table variable "@T1".
```

Представленный далее код демонстрирует, что табличные переменные не видны даже другим пакетам одного уровня.

```
DECLARE @T1 AS TABLE
(col1 INT NOT NULL);
INSERT INTO @T1(col1) VALUES(10);
GO
SELECT col1 FROM @T1;
GO
```

Запрос `SELECT` к табличной переменной выполняется со следующей ошибкой:

```
Msg 1087, Level 15, State 2, Line 2
Must declare the table variable "@T1".
```

## Язык DDL и индексы

Этот раздел посвящен взаимоотношениям языка описания данных (data definition language, DDL) и индексов с временными таблицами и табличными переменными. В нем рассматриваются вопросы именования ограничений и возможность определения индексов на временных объектах.

Вам следует знать, что удивляет некоторых людей в отношении использования имен ограничений во временных таблицах. Оказывается, имена ограничений считаются именами объектов в схеме, а имена объектов должны быть уникальными на уровне схемы — не на уровне таблицы. Таким образом, две таблицы не могут иметь ограничений с одним и тем же именем в одной и той же схеме.

Временные таблицы создаются в базе данных tempdb в схеме `dbo`. Как уже упоминалось, вы можете создать две временные таблицы с одним именем в разных сессиях, поскольку SQL Server добавляет уникальный суффикс к каждому имени в соответствии с внутренним алгоритмом. Но если вы создаете временные таблицы в разных сессиях с одним и тем же именем ограничения, только одна будет создана, а вторая попытка закончится ошибкой.

В качестве примера попробуйте создать следующую временную таблицу в двух разных сессиях.

```
CREATE TABLE #T1
(col1 INT NOT NULL,
 col2 INT NOT NULL,
 col3 DATE NOT NULL,
 CONSTRAINT PK_#T1 PRIMARY KEY(col1));
```

Первая попытка прошла успешно, но вторая завершилась следующей ошибкой:

```
Msg 2714, Level 16, State 5, Line 1
There is already an object named 'PK_#T1' in the database.
(В базе данных уже есть объект с именем 'PK_#T1')
Msg 1750, Level 16, State 0, Line 1
Could not create constraint. See previous errors.
(Невозможно создать ограничение, см. предыдущую ошибку.)
```

Причина ошибки заключается в том, что в коде используется имя ограничения первичного ключа `PK_#T1`, и SQL Server не разрешает двух появлений одного и того же имени ограничения в одной схеме.

Выполните следующий код для удаления таблицы в сеансе, в котором она была успешно создана.

```
DROP TABLE #T1;
```

Если вы определите ограничение без имени, SQL Server самостоятельно создаст уникальное имя для него. Поэтому мы рекомендуем не давать имена ограничениям во временных таблицах. Далее приведено исправленное определение таблицы, на этот раз без присвоения имен ограничениям.

```
CREATE TABLE #T1
(col1 INT NOT NULL,
 col2 INT NOT NULL,
 col3 DATE NOT NULL,
 PRIMARY KEY(col1));
```

Выполните этот код в двух разных сессиях. Вы увидите, что обе попытки на этот раз завершились успешно.

В SQL Server можно создавать индексы на временных таблицах после создания таблиц. Например, следующий код создает индекс на таблице `#T1`, которую вы только что сформировали.

```
CREATE UNIQUE NONCLUSTERED INDEX idx_col2 ON #T1(col2);
```

Вы также можете изменить определение таблицы и применить изменения определений, такие как добавление ограничения или столбца.

По окончании выполните следующий код для удаления таблицы `#T1` во всех сессиях, в которых вы ее создали.

```
DROP TABLE #T1;
```

Для табличных переменных SQL Server не разрешает явное присвоение имен ограничениям — даже в одном сеансе. Попробуйте выполнить следующий код, чтобы попытаться объявить табличную переменную, у которой есть именованное ограничение.

```
DECLARE @T1 AS TABLE
(col1 INT NOT NULL,
 col2 INT NOT NULL,
 col3 DATE NOT NULL,
 CONSTRAINT PK_@T1 PRIMARY KEY(col1));
```

Попытка завершилась следующей ошибкой:

```
Msg 156, Level 15, State 2, Line 6
Incorrect syntax near the keyword 'CONSTRAINT'.
(Nеправильный синтаксис в окружении ключевого слова 'CONSTRAINT'.)
```

Попробуйте еще раз, без присвоения имен ограничениям.

```
DECLARE @T1 AS TABLE
(col1 INT NOT NULL,
 col2 INT NOT NULL,
 col3 DATE NOT NULL,
 PRIMARY KEY(col1));
```

На этот раз таблица объявлена успешно.

SQL Server не разрешает создавать индексы на табличной переменной после объявления таблицы. Но напомним, что когда вы определяете ограничение первичного ключа, SQL Server обеспечивает его уникальность по умолчанию с помощью кластеризованного индекса. Когда вы определяете ограничение уникальности, SQL Server по умолчанию обеспечивает его уникальность посредством уникального некластеризованного индекса. Поэтому если вы хотите определить индексы на табличной переменной, то можете сделать это косвенным образом, через определение ограничений. Следующий пример показывает это посредством объявления табличной переменной, которая имеет ограничение первичного ключа и ограничение уникальности, создавая кластеризованный и некластеризованный индексы соответственно.

```
DECLARE @T1 AS TABLE
(col1 INT NOT NULL,
 col2 INT NOT NULL,
 col3 DATE NOT NULL,
 PRIMARY KEY(col1),
 UNIQUE(col2));
```

## Физическое представление в базе данных tempdb

Существует общее заблуждение, что только временные таблицы имеют физическое представление в базе данных tempdb, а табличные переменные размещаются лишь в памяти. Это не так. И временные таблицы, и табличные переменные имеют физическое представление в базе данных tempdb.

Представление sys.objects содержит записи для внутренних таблиц, которые SQL Server создает в базе данных tempdb для реализации временных таблиц и табличных переменных. В качестве примера следующий код сначала создает временную таблицу с именем #T1, а затем запрос к представлению sys.objects в базе данных tempdb для поиска имен таблиц с именами, начинающимися на #.

```
CREATE TABLE #T1
(col1 INT NOT NULL);
INSERT INTO #T1(col1) VALUES(10);
SELECT name FROM tempdb.sys.objects WHERE name LIKE '#%';
DROP TABLE #T1;
GO
```

Этот код генерирует следующие выходные данные в тестовой системе:

```
name

#T1_000000000018
```

Как уже говорилось ранее, SQL Server добавляет суффикс к присвоенному пользователем имени таблицы в соответствии с внутренним алгоритмом, чтобы предотвратить конфликт имен в случае, если несколько сеансов создают временную таблицу с одним и тем же именем.

Следующий код демонстрирует, как объявить табличную переменную, и затем запрашивает представление sys.objects.

```
DECLARE @T1 AS TABLE
(col1 INT NOT NULL);
INSERT INTO @T1(col1) VALUES(10);
SELECT name FROM tempdb.sys.objects WHERE name LIKE '#%';
```

При выполнении на тестовой системе этот код дает следующий результат:

```
name

#BD095663
```

Как видите, SQL Server создает таблицу в базе данных tempdb для реализации табличной переменной, которую вы объявили.

Общий вопрос: материализуются ли табличные выражения, такие как обобщенные табличные выражения (common table expressions, CTE), аналогично временными таблицам и табличным переменным? Ответ — нет. Когда SQL Server оптимизирует запрос к табличному выражению, он раскрывает логику внутреннего запроса и взаимодействует напрямую с базовыми таблицами. Это означает, что в отличие от временных таблиц и табличных переменных, табличные выражения не имеют физической реализации.

## Транзакции

Временные таблицы и табличные переменные взаимодействуют с транзакциями по-разному. Временные таблицы подобны стандартным таблицам в этом отношении. Изменения, применяемые к временной таблице, отменяются при откате транзакции. Это показано в следующем примере:

```
CREATE TABLE #T1
(col1 INT NOT NULL);
BEGIN TRAN
 INSERT INTO #T1(col1) VALUES(10);
ROLLBACK TRAN
SELECT col1 FROM #T1;
DROP TABLE #T1;
GO
```

Код создает временную таблицу, открывает транзакцию, вставляет строку, откатывает транзакцию и запрашивает таблицу. Этот код генерирует следующие выходные данные, которые показывают, что выполненная транзакцией работа была отменена.

```
col1
```

```

```

Так же как в случае с обычными переменными, изменения, примененные к табличным переменным в транзакции, не отменяются, если выполнен откат транзакции. Это показывает следующий тест:

```
DECLARE @T1 AS TABLE
(col1 INT NOT NULL);
BEGIN TRAN
 INSERT INTO @T1(col1) VALUES(10);
ROLLBACK TRAN
SELECT col1 FROM @T1;
```

Этот код генерирует следующие выходные данные, которые показывают, что выполненная транзакцией работа не была отменена.

```
col1
```

```

```

```
10
```

Обратите внимание, единственная инструкция к табличной переменной должна быть атомарной, поэтому, если эта инструкция выдаст ошибку до завершения обработки, частичное изменение отменяется. Но если эта единственная инструкция завершится, а пользовательская транзакция будет откачена, такое изменение не отменяется. Это весьма полезно при обработке ошибок, если нужно записывать в журнал информацию в транзакции, откат которой выполняется.

## Статистика

Что касается производительности, имеется существенная разница между временными таблицами и табличными переменными. SQL Server поддерживает статистику распространения (*гистограммы*) для временных таблиц, но не для табличных переменных. Это означает, что, вообще говоря, планы для временных таблиц бывают более оптимальными. Это определяется стоимостью поддержки гистограмм и стоимостью повторной компиляции, связанной с обновлением гистограмм.

Чтобы показать это, приведем пример, в котором существование гистограммы ведет к созданию оптимального плана, а отсутствие гистограммы приводит к получению неоптимального плана. Для измерения стоимости ввода-вывода запросов в вашем сеансе выполните следующий код:

```
SET STATISTICS IO ON;
```



Используйте следующий код, чтобы создать временную таблицу и заполнить ее миллионом строк.

```

CREATE TABLE #T1
(
 col1 INT NOT NULL,
 col2 INT NOT NULL,
 col3 DATE NOT NULL,
 PRIMARY KEY(col1),
 UNIQUE(col2);
)
INSERT INTO #T1(col1, col2, col3)
SELECT n, n * 2, CAST(SYSDATETIME() AS DATE)
FROM dbo.GetNums(1, 1000000);

```

Этот код определяет ограничение первичного ключа со значением `col1` в качестве ключа, создавая кластеризованный индекс. Он также определяет ограничение уникальности со значением `col2` в качестве ключа, создавая некластеризованный индекс. Установите параметр **Include Actual Query Plan** (Включить действительный план выполнения), нажав комбинацию клавиш `<Ctrl>+<M>` в среде SSMS, и затем выполните следующий код:

```

SELECT col1, col2, col3
FROM #T1
WHERE col2 <= 5;

```

SQL создает план этого запроса, показанный на рис. 16.1.

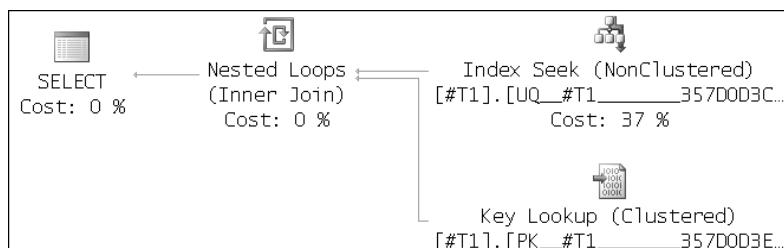


Рис. 16.1. План выполнения для временной таблицы

Это очень эффективный план. Оптимизатор проверил гистограмму на столбце `col2` и предварительно рассчитал, что поддерживается очень мало строк, которые должны быть отфильтрованы. План решил использовать индекс на `col2`, поскольку фильтр является очень выборочным. Требуется только небольшое количество уточняющих запросов ключа, чтобы получить соответствующие строки данных. Для такого выборочного фильтра это более подходящий план по сравнению с тем, который выполняет полный просмотр кластеризованного индекса. Параметр `STATISTICS IO` показывает, что потребовалось только 9 операций логического чтения для выполнения этого плана.

Выполните следующий код для удаления временной таблицы:

```
DROP TABLE #T1;
```

Используйте следующий код, чтобы выполнить такой же тест, но на этот раз примените табличную переменную.

```

DECLARE @T1 AS TABLE
(col1 INT NOT NULL,
 col2 INT NOT NULL,
 col3 DATE NOT NULL,
 PRIMARY KEY(col1),
 UNIQUE(col2));

INSERT INTO @T1(col1, col2, col3)
SELECT n, n * 2, CAST(SYSDATETIME() AS DATE)
FROM dbo.GetNums(1, 1000000);

SELECT col1, col2, col3
FROM @T1
WHERE col2 <= 5;
GO

```

План для этого запроса показан на рис. 16.2.

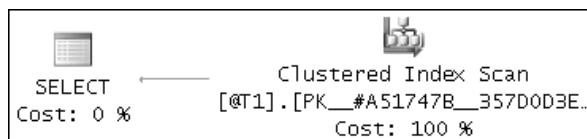


Рис. 16.2. План выполнения для табличной переменной

В отличие от временных таблиц, SQL Server не поддерживает гистограммы для табличных переменных. Не имея возможности точно рассчитать выборочность фильтра, оптимизатор полагается на жестко заданный предварительный расчет, который предполагает довольно низкую выборочность (30%). В результате оптимизатор выбирает выполнение полного просмотра кластеризованного индекса, стоимость которого составляет 2485 операций логического чтения. Он просто не понял, что в действительности фильтр очень выборочный и значительно более эффективным был бы план, подобный показанному на рис. 16.1 для временной таблицы.

Теперь вы можете отключить вывод стоимости ввода-вывода в данном сеансе, выполнив следующий код:

```
SET STATISTICS IO OFF;
```

Из этого примера можно сделать вывод, что когда эффективность плана зависит от существования гистограмм, нужно использовать временные таблицы. Табличные переменные хорошо использовать в двух общих случаях. Один — когда объем данных настолько маленький, как страница или две, что эффективность плана не имеет значения. Другой случай — когда план очень простой. *Простой план* означает, что существует только один разумный план, и оптимизатору фактически не нужны гистограммы для принятия решения. Пример такого плана — просмотр диапазона данных в кластеризованном индексе или покрывающем индексе. Такой план не зависит от выборочности фильтра — просто это всегда лучшее решение, чем полный просмотр.

## КОНТРОЛЬНЫЕ ВОПРОСЫ

- Как вы создадите локальную временную таблицу и как — глобальную?
- Можно ли присваивать имена ограничениям в локальных временных таблицах и в табличных переменных?

## Ответы на контрольные вопросы

- Нужно присвоить имя локальной временной таблице, используя один знак решетки # в качестве префикса, а глобальной локальной переменной — с использованием двух знаков решетки в качестве префикса.
- Можно присваивать имена ограничениям в локальных временных таблицах, хотя это не рекомендуется делать, поскольку это может привести к конфликту имен. Присваивать имена ограничениям в табличных переменных нельзя.

## ПРАКТИКУМ Выбор оптимального временного объекта

В данном практикуме вам предстоит применить полученные знания о временных объектах.

### Задание 1. Сравнение текущего количества заказов с количеством заказов за предыдущий год с использованием СТЕ

В этом задании вам поставлена задача, для решения которой вы должны использовать табличное выражение СТЕ. В задачу входит запрос к таблице Sales.Orders, позволяющий вычислить количество заказов в год. Вы должны возвратить количество заказов за текущий год для каждого года и разницу между количествами заказов за текущий и предыдущий годы. Решение должно быть совместимо с SQL Server 2005 и SQL Server 2008, поэтому вы не можете использовать функциональность, добавленную только в SQL Server 2012.

- Откройте SQL Server Management Studio (SSMS) и подключитесь к учебной базе данных TSQL2012.
- Напишите запрос, который вычисляет количество заказов за год, и запросите включение действительного плана выполнения, нажав комбинацию клавиш <Ctrl>+<M> в среде SSMS. Ваш запрос должен выглядеть следующим образом:

```
SELECT YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
FROM Sales.Orders
GROUP BY YEAR(orderdate);
```

SQL Server генерирует план для этого запроса, показанный на рис. 16.3.

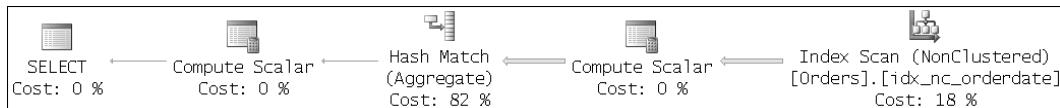


Рис. 16.3. План для сгруппированного запроса

Этот план показывает, что просматривается индекс `idx_nc_orderdate` для получения всех дат заказов, а затем эти данные группируются и статистически обрабатываются.

- Напишите решение для этой задачи с помощью СТЕ. А именно, определите СТЕ на основе запроса из шага 2. Во внешнем запросе соедините два экземпляра СТЕ для сопоставления каждой строки года с соответствующей строкой из предыдущего года. Затем вычислите разницу между количеством заказов за текущий и предыдущий годы. Ваше решение должно выглядеть следующим образом:

```
WITH YearlyCounts AS
(SELECT YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
 FROM Sales.Orders
 GROUP BY YEAR(orderdate))
SELECT C.orderyear, C.numorders, C.numorders - P.numorders AS diff
FROM YearlyCounts AS C
INNER JOIN YearlyCounts AS P
 ON C.orderyear = P.orderyear + 1;
```

- Изучите план выполнения, показанный на рис. 16.4, который SQL Server сгенерировал для этого запроса.

Обратите внимание, работа по просмотру индекса, а также группирование и статистическая обработка данных были выполнены дважды.

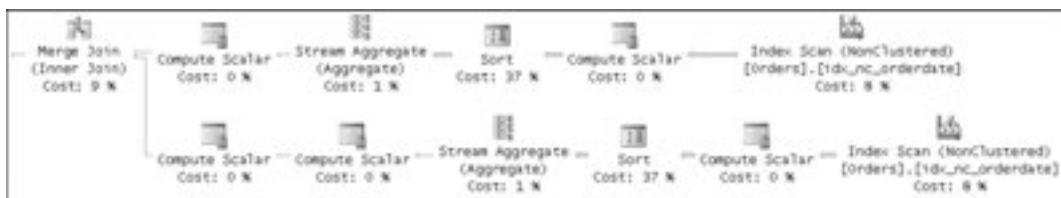


Рис. 16.4. План выполнения для решения с использованием табличных выражений СТЕ

## **Задание 2. Сравнение текущего количества заказов с количеством заказов за прошлый год с использованием табличных переменных**

Здесь вам нужно выполнить ту же задачу, что и в предыдущем задании, но с помощью табличной переменной.

- Решение, использовавшее табличное выражение СТЕ, предполагало просмотр таблицы два раза. Вам нужно найти решение, избегающее дублирования работы (представьте себе таблицы `Orders` значительно большего размера, чем в демонстрационной базе данных). Для достижения этого вам нужно сохранить результат сгруппированного запроса во временной таблице или табличной переменной и затем соединить два экземпляра временных объектов. Поскольку результирующий набор, который требуется сохранить, очень мал на этот раз, табличная переменная может это сделать. Ваше решение должно выглядеть следующим образом:

```

DECLARE @YearlyCounts AS TABLE
(orderyear INT NOT NULL,
 numorders INT NOT NULL,
 PRIMARY KEY(orderyear));

INSERT INTO @YearlyCounts(orderyear, numorders)
 SELECT YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
 FROM Sales.Orders
 GROUP BY YEAR(orderdate);
SELECT C.orderyear, C.numorders, C.numorders - P.numorders AS diff
 FROM @YearlyCounts AS C
 INNER JOIN @YearlyCounts AS P
 ON C.orderyear = P.orderyear + 1;

```

## 2. Изучите сгенерированный SQL Server план для этого решения (рис. 16.5).

Обратите внимание, работа по просмотру, группированию и статистической обработке данных выполнена только один раз (верхняя часть плана). Результат сохранен в табличной переменной. Нижняя часть плана показывает соединение двух экземпляров маленьких табличных переменных.

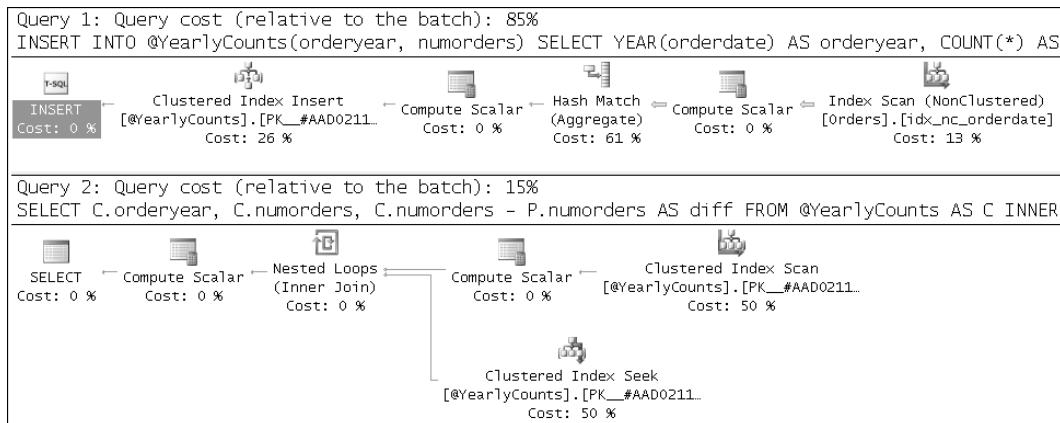


Рис. 16.5. План выполнения для решения, использующего табличные переменные

## Резюме занятия

- Вы можете использовать временные таблицы и табличные переменные, когда вам нужно временно сохранить данные, такие как промежуточный результирующий набор запроса.
- Временные таблицы и табличные переменные имеют множество различий, включая область действия, язык DDL и индексирование, взаимодействие с транзакциями и статистику распространения.
- Локальные временные таблицы видны уровню, который их создал, всем пакетам этого уровня, а также внутренним и уровням в стеке вызовов. Табличные переменные видны только пакету, который их объявил.

- Можно применять язык DDL к временной таблице после ее создания, включая создание индексов и другие изменения DDL. Нельзя применять изменения DDL к табличной переменной после ее объявления. В табличной переменной можно создавать индексы косвенным образом посредством ограничений первичного ключа и уникальности.
- Изменения, примененные к временной таблице в транзакции, отменяются при откате этой транзакции. Изменения, примененные к табличной переменной, не отменяются при откате транзакции.
- SQL Server поддерживает статистику распространения на временных таблицах, но не на табличных переменных. В результате, планы запросов, использующих временные таблицы, более оптимальны, чем планы с использованием табличных переменных.

## Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Какой из перечисленных вариантов подходит для использования табличных переменных? (Выберите все подходящие варианты.)
  - A. Когда таблицы очень маленькие, а план — простой.
  - B. Когда таблицы очень маленькие, а план — непростой.
  - C. Когда таблицы большие, а план — простой.
  - D. Когда таблицы большие, а план — непростой.
2. Можно ли создать индексы на табличных переменных?
  - A. Нет.
  - B. Да, выполнив команду `CREATE INDEX`.
  - C. Да, косвенно, определив первичный ключ и ограничения уникальности.
  - D. Да, с помощью определения внешнего ключа.
3. Перед вами поставлена задача реализовать триггер. Как часть кода триггера в особых условиях, вам нужно выполнять откат транзакции. Но вам требуется скопировать данные из вставленных и удаленных таблиц в таблицы аудита, чтобы отслеживать изменения. Как вы можете сделать это?
  - A. Выполнить откат транзакции и затем скопировать данные из удаленных и вставленных таблиц в таблицы аудита.
  - B. Скопировать данные из вставленных и удаленных таблиц в таблицы аудита и затем выполнить откат транзакции.
  - C. Скопировать строки из вставленных и удаленных таблиц во временные таблицы, выполнить откат транзакции и затем скопировать данные из временных таблиц в таблицы аудита.

- D. Скопировать строки из вставленных и удаленных таблиц в табличные переменные, выполнить откат транзакции и затем скопировать данные из табличных переменных в таблицу аудита.

## Упражнения

---

В следующих упражнениях вы примените полученные знания о курсорах, наборах данных и временных таблицах. Ответы на вопросы можно найти в *приложении "Ответы" в конце книги*.

### **Упражнение 1. Рекомендации по повышению производительности для курсоров и временных объектов**

Вы наняты в качестве консультанта в новую компанию, которая разрабатывает приложение, использующее SQL Server в качестве базы данных. Компания в настоящее время сталкивается с проблемами производительности и масштабируемости. Вы изучили код и выяснили следующее.

Почти все решения выполнены как курсоры. Изучив эти решения, вы увидели, что это не те случаи, которые должны быть реализованы с помощью итерационной логики.

Некоторые решения хранят промежуточные результаты в табличных переменных и затем запрашивают эти табличные переменные. Большое количество строк хранится в табличных переменных.

1. Можете ли вы дать рекомендации по поводу факта использования большинством решений курсоров?
2. Можете ли вы дать рекомендации по поводу использования табличных переменных?
3. Можете ли вы объяснить клиенту, при каких обстоятельствах должны использоваться курсоры и табличные переменные?

### **Упражнение 2. Указать неточности в ответах**

Вы участвуете в конференции и присутствуете на лекции, посвященной языку T-SQL. В конце конференции лектор предлагает задать вопросы и отвечает на них. Далее приведены вопросы, которые слушатели задавали лектору, и ответы лектора на них. Укажите неточности в ответах лектора.

1. Вопрос: в чем заключается разница между временными таблицами и табличными переменными с точки зрения производительности?

Ответ: никакой разницы. Компания Microsoft просто предлагает немного разные способы решения одной и той же задачи.

2. Вопрос: у меня есть многострочный триггер UPDATE, который устанавливает значение столбца lastmod в измененных строках в значение, возвращенное функцией SYSDATETIME(). Этот триггер применяет курсор к вставленным таблицам для построчной обработки. Триггер работает очень медленно. Можете дать рекомендации по увеличению производительности триггера?

Ответ: вместо использования курсора напишите решение на основе наборов данных, которое использует цикл WHERE и запрос TOP для выполнения итераций по ключам по одному за один раз.

3. Вопрос: приведите, пожалуйста, пример, когда полезны табличные выражения?

Ответ: один из примеров — ситуация, когда требуется сохранить результат ресурсоемкого запроса и затем многократно ссылаться на этот результат.

## Рекомендуемые упражнения

Для успешного освоения экзаменационных тем, представленных в данной главе, выполните следующие задания.

### Укажите различия

Здесь вы проверите, насколько хорошо поняли, чем отличаются временные объекты, а также реляционные и итерационные решения в задачах выполнения запросов.

- Задание 1.** Не используя текст занятия, постарайтесь заполнить табл. 16.1 характеристиками временных таблиц, табличных переменных и табличных выражений для перечисленных свойств.

**Таблица 16.1.** Сравнение временных объектов

|                                                | Временная таблица | Табличная переменная | Табличное выражение |
|------------------------------------------------|-------------------|----------------------|---------------------|
| Область действия                               |                   |                      |                     |
| Может применять DDL после создания/объявления? |                   |                      |                     |
| Может иметь индексы?                           |                   |                      |                     |
| Зависит от ROLLBACK?                           |                   |                      |                     |
| Имеет статистику распространения?              |                   |                      |                     |
| Присутствует физически в tempdb?               |                   |                      |                     |
| Для таблиц какого размера подходит?            |                   |                      |                     |

- Задание 2.** Используя только свою память, перечислите различия между реляционными и итерационными решениями в задачах выполнения запросов.

# ГЛАВА 17

## Основные сведения о дальнейших аспектах оптимизации

### Темы экзамена

- Устранение неполадок и оптимизация.
  - Оценка использования операций на основе строк по сравнению с операциями на основе наборов.

В главе 15 вы знакомились с внутренними компонентами индексов и статистики в Microsoft SQL Server. В главе 14 вы узнали об инструментах, которые помогают анализировать процесс выполнения запроса.

В этой главе вы узнаете, как SQL Server находит данные, которые требуются запросу. SQL Server имеет разные методы доступа для извлечения данных. Кроме того, SQL Server реализует различные алгоритмы соединения. Данная глава рассказывает о них, а также их сильных и слабых местах.

SQL Server может повторно использовать план запроса, даже если последующий запрос не такой. Например, последующий запрос может изменить значение условия поиска в предложении `WHERE` запроса. SQL Server пытается параметризовать оперативные запросы, чтобы позволить повторное использование плана запроса. Помимо такой автопараметризации вы можете выполнить параметризацию ваших запросов, например, используя их в хранимых процедурах.

Можно влиять на выполнение запросов с помощью *подсказок*, используемых в SQL Server. Подсказки — это указания на то, как выполнить запрос. Можно применять *табличные подсказки*, для которых вы указываете, как использовать конкретную таблицу в запросе, и *указания запросов*, являющиеся подсказками на уровне запроса. Для последних можно указать, например, какой должен использоваться алгоритм соединения для конкретного запроса. Также можно применять *указания соединений* только для одного соединения. Наконец, можно предписать выполнение целого запроса с помощью структур плана.



## ПРЕЖДЕ ВСЕГО

Для полного освоения этой главы необходимо иметь:

- понимание основ реляционных баз данных;
- опыт работы с SQL Server Management Studio (SSMS);
- некоторый опыт в написании кода на языке T-SQL;
- доступ к экземпляру SQL Server 2012 с установленной учебной базой данных TSQL2012.

# Занятие 1. Общие сведения об итераторах планов

Как вам уже известно из *главы 14*, SQL Server выполняет запрос с помощью набора физических операторов. Поскольку эти операторы выполняют итерации на наборах строк, их также называют *итераторами*. В данном занятии вы узнаете подробности о наиболее важных итераторах: используемых для доступа к данным, выполнения соединений и выполнения прочих действий, предназначенных для извлечения желаемых результатов.



**Изучив материал этого занятия, вы сможете:**

- ✓ Объяснить разные методы доступа, используемые в SQL Server
- ✓ Описать алгоритмы соединений
- ✓ Описать другие важные итераторы планов

**Продолжительность занятия — 50 минут.**

## Методы доступа

Если таблица организована как куча, тогда единственным методом доступа, возможным в SQL Server, является *просмотр таблицы*. Просмотр производится без определенной логической последовательности; SQL Server использует страницы *карты распределения индекса* (Index Allocation Map, IAM) для выполнения поиска в физическом *порядке выделения*. SQL Server также может использовать просмотр в порядке выделения, когда таблица является кластеризованной. Просмотр в порядке выделения работает быстрее, если таблица меньше фрагментирована физически; просмотр будет более медленным, если физическая фрагментация велика. На просмотр в порядке выделения не влияет логическая фрагментация. Следующий код создает кучу посредством выбора всех строк в таблице Sales.OrderDetails базы данных TSQL2012 и помещает их в новую таблицу.



```
SELECT orderid, productid, unitprice, qty, discount
INTO Sales.OrderDetailsHeap
FROM Sales.OrderDetails;
```

Даже если вы выбираете только несколько столбцов из этой таблицы и используете очень выборочное предложение WHERE, ограничивающее результирующий набор

одной строкой, как показано в следующем запросе, SQL Server использует итератор **Table Scan** (Просмотр таблицы) для извлечения данных.

```
SELECT orderid, productid
FROM Sales.OrderDetailsHeap
WHERE orderid = 10248
 AND productid = 11;
```

На рис. 17.1 представлен план выполнения для этого запроса.

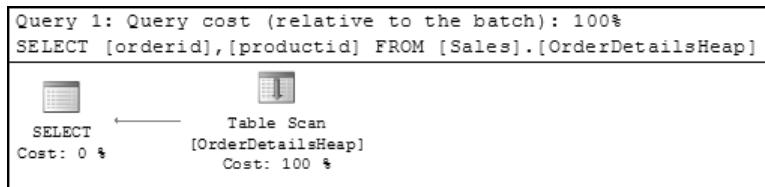


Рис. 17.1. Итератор **Table Scan** и метод доступа

SQL Server может использовать просмотр в порядке выделения для кластеризованной таблицы, если таблица содержит более 64 страниц и в запросе нет требования определенного порядка, а также если уровень изоляции — Read Uncommitted (Незафиксированное чтение) или если вы работаете в среде без разрешения записи. Когда SQL Server просматривает кластеризованный индекс, он также может выполнять просмотр в логической последовательности индекса с помощью просмотра в порядке индекса. В обоих случаях используется итератор **Clustered Index Scan** (Просмотр кластеризованного индекса). SQL Server использует индексный связанный список конечного уровня для выполнения просмотра в порядке индекса. На просмотр в порядке индекса негативно влияют и логическая, и физическая фрагментация. Следующий запрос не требует специально упорядоченного результата; вы видите, что свойство **Ordered** (Отсортировано) оператора **Clustered Index Scan** (Просмотр кластеризованного индекса) имеет значение **False** (Ложь), и это означает, что SQL Server не должен был возвращать данные упорядоченными, как показано на рис. 17.2.

```
SELECT orderid, productid, unitprice
FROM Sales.OrderDetails;
```

Некластеризованный индекс может покрывать запрос. *Покрытие* запроса означает, что SQL Server может найти все нужные запросу данные в некластеризованном индексе и не обязан выполнять уточняющий поиск в базовой таблице. SQL Server использует итератор **Index Scan** (Поиск индекса) для просмотра некластеризованного индекса. Как в случае с итератором **Clustered Index Scan** (Просмотр кластеризованного индекса), SQL Server может выполнить просмотр в порядке выделения или в порядке индекса при просмотре некластеризованного индекса. Следующий запрос просматривает некластеризованный индекс и возвращает данные в индексном порядке (рис. 17.3).

```
SELECT orderid, productid
FROM Sales.OrderDetails
ORDER BY productid;
```

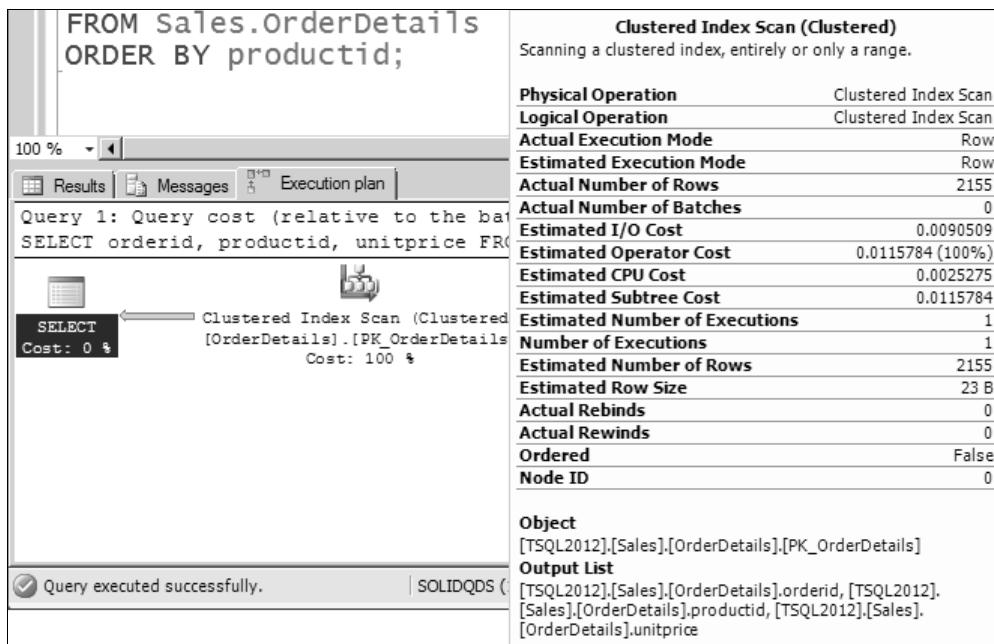


Рис. 17.2. Просмотр кластеризованного индекса с возвращенным неупорядоченным набором строк

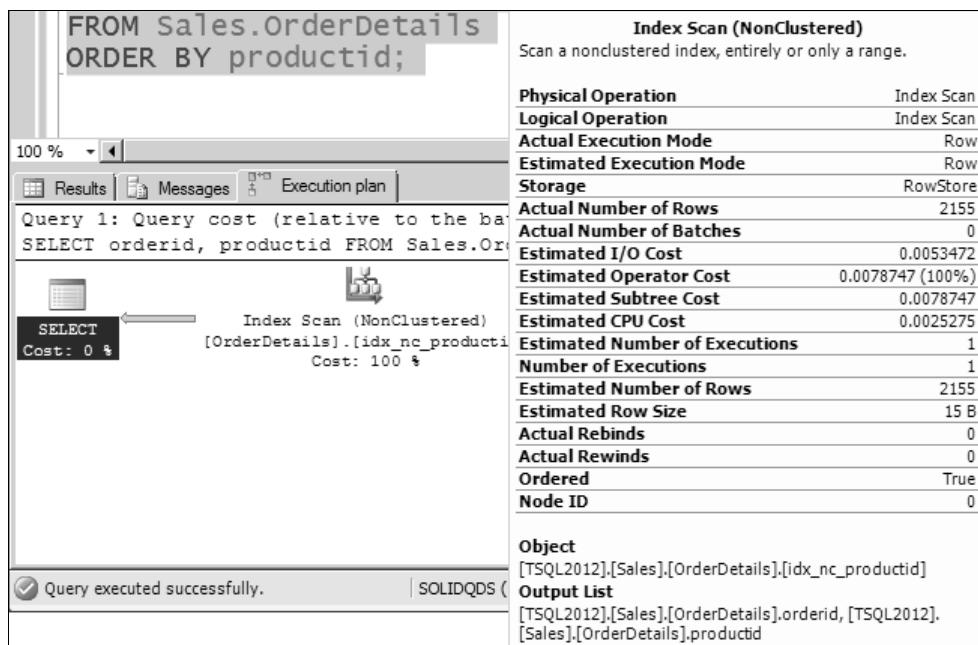


Рис. 17.3. Оператор Index Scan (NonClustered) с данными, возвращенными в порядке индексации

В некоторых случаях оптимизатор запросов SQL Server даже может решить покрыть запрос с несколькими кластеризованными индексами. SQL Server может объединять некластеризованные индексы. Все некластеризованные индексы таблицы всегда имеют какие-нибудь общие данные, которые можно использовать для соединения. Если таблица организована как куча, этими данными является идентификатор строки (row identifier, RID); если таблица кластеризованная, это ключ кластеризации. Вы также можете улучшить покрытие запроса с помощью включенных столбцов в некластеризованном индексе.

Обратите внимание, просмотры в порядке выделения могут быть небезопасными. Используя просмотр в порядке выделения, SQL Server может пропустить некоторые строки и прочитать другие строки несколько раз. Это может произойти, если используется уровень изоляции Read Uncommitted (Незафиксированное чтение) в среде с возможностью чтения/записи. Пока один запрос выполняет просмотр в порядке выделения, другая команда может обновить данные и привести к перемещению одной или более строк. Выполняющий просмотр запрос мог уже прочитать эти строки и может прочесть их снова после этого перемещения. Или выполняющий просмотр запрос, возможно, уже передал страницу, в которую была перемещена строка с еще неотсканированной страницы, и этот запрос никогда не сможет прочесть такую строку. Страна может перемещаться по разным причинам. Например, команда может обновить столбец переменной длины и заменить короткое значение длинным. Страница может быть заполнена, и тогда обновленная строка должна переместиться на другую страницу. Набор строк может переместиться, если существуют вставки в кластеризованную таблицу и происходит разбиение страницы; примерно половина строк переместится на новую страницу. Следует соблюдать большую осторожность при использовании уровня изоляции Read Uncommitted в среде с разрешенными чтением и записью.

При просмотре индекса SQL Server не ограничен возможностью полного поиска. Если вы ограничите возвращаемый запросом набор данных и просмотр является упорядоченным, SQL Server может выполнить поиск первого требуемого значения в наборе строк, а затем осуществить частичный просмотр последующих значений в логической последовательности индекса. SQL Server может использовать поиск и частичный просмотр и для кластеризованных индексов, и для покрывающих некластеризованных индексов. Рассмотрите следующий запрос:

```
SELECT orderid, productid, unitprice
FROM Sales.OrderDetails
WHERE orderid BETWEEN 10250 AND 10390
ORDER BY orderid, productid;
```

Этот запрос создал план выполнения, показанный на рис. 17.4. Обратите внимание, использовался оператор **Clustered Index Seek** (Поиск в кластеризованном индексе); но, вы видите, что значение **Actual Number of Rows** (Фактическое количество строк) равно 377. Таким образом, после того как был найден первый нужный заказ, SQL Server не стал искать последующие заказы, вместо этого он выполнял частичный просмотр.

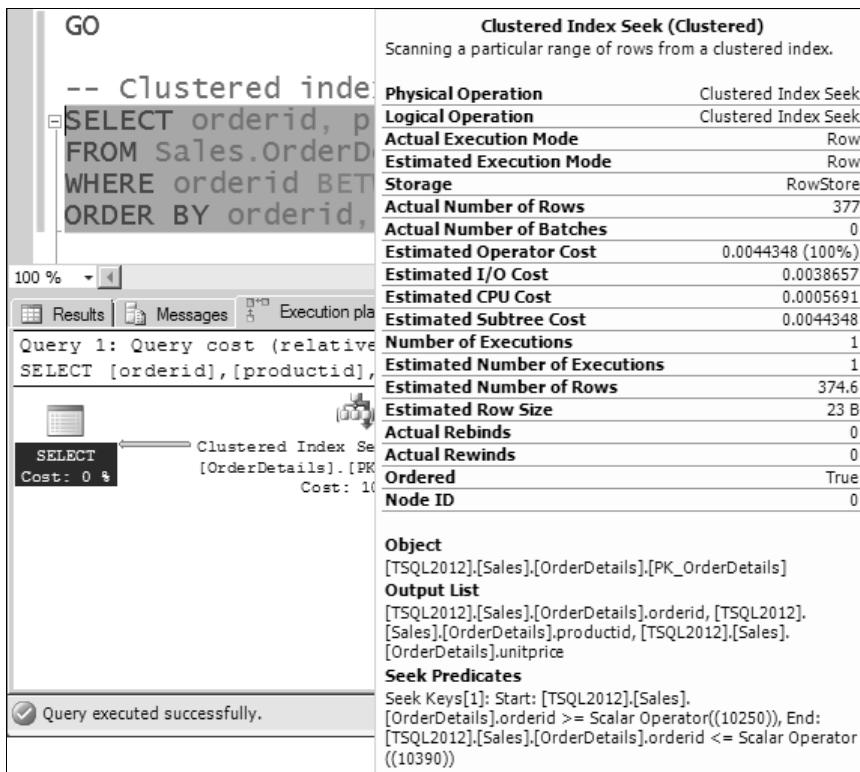


Рис. 17.4. Оператор **Clustered Index Seek** с частичным поиском и упорядоченными данными

Как уже говорилось, SQL Server может использовать тот же метод доступа, поиск в индексе и затем — частичный упорядоченный просмотр покрывающего некластеризованного индекса, как показано в следующем запросе:

```

SELECT orderid, productid
FROM Sales.OrderDetails
WHERE productid BETWEEN 10 AND 30
ORDER BY productid;

```

На рис. 17.5 представлен план выполнения для этого запроса. Обратите внимание, использовался оператор **Index Seek** (Поиск в индексе), и свойство **Actual Number of Rows** (Фактическое количество строк) равно 593.

Возможно, наиболее распространенный метод доступа, который использует SQL Server в среде оперативной обработки транзакций (online transaction processing, OLTP), — это поиск по некластеризованному индексу с упорядоченным частичным просмотром и последующим уточняющим запросом в базовую таблицу по каждой строке, найденной в некластеризованном индексе. Такие планы обычно используются для выборочных запросов. Базовая таблица может быть организована как куча или сбалансированное дерево. Если таблица организована как куча, SQL Server использует оператор **RID Lookup** (Уточняющий запрос RID) для извлечения строк из базовой таблицы. SQL Server находит строки в базовой таблице с помощью их

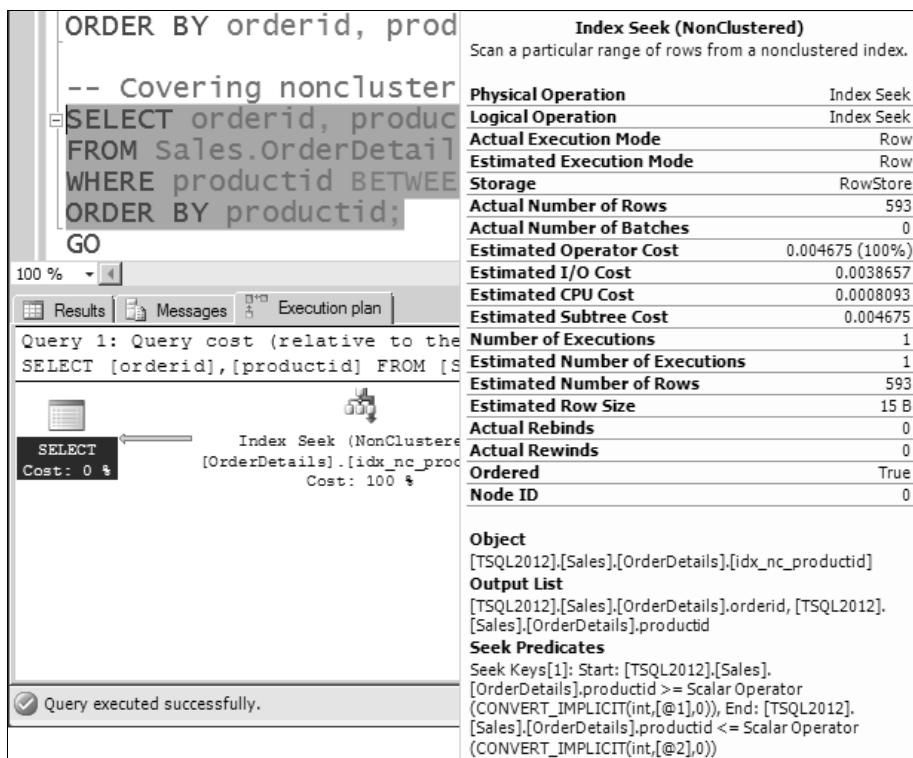


Рис. 17.5. Оператор **Index Seek** в покрывающем некластеризованном индексе с частичным просмотром и упорядоченными данными

идентификаторов RID. Следующий фрагмент кода создает некластеризованный индекс на куче и затем запрашивает таблицу с помощью поиска в некластеризованном индексе с упорядоченным частичным просмотром и уточняющим запросом RID.

```
CREATE NONCLUSTERED INDEX idx_nc_qtyheap ON Sales.OrderDetailsHeap(qty);
SELECT orderid, productid, unitprice, qty
FROM Sales.OrderDetailsHeap
WHERE qty = 52;
```

Этот запрос создал план выполнения, показанный на рис. 17.6.

Если таблица кластеризованная, SQL Server использует оператор уточняющего запроса ключа **Key Lookup** вместо оператора **RID Lookup** (Уточняющий запрос RID). Следующий код создает некластеризованный индекс на кластеризованной таблице и затем запрашивает данные, используя поиск по некластеризованному индексу с упорядоченным частичным просмотром и уточняющим запросом по ключу.

```
CREATE NONCLUSTERED INDEX idx_nc_qty ON Sales.OrderDetails(qty);
SELECT orderid, productid, unitprice, qty
FROM Sales.OrderDetails
WHERE qty = 52;
```

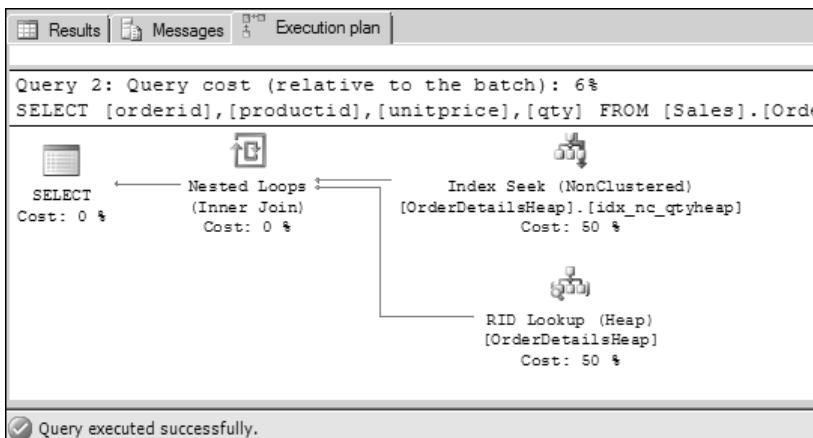


Рис. 17.6. Оператор поиска в некластеризованном индексе Index Seek (NonClustered) с частичным просмотром и оператором RID Lookup

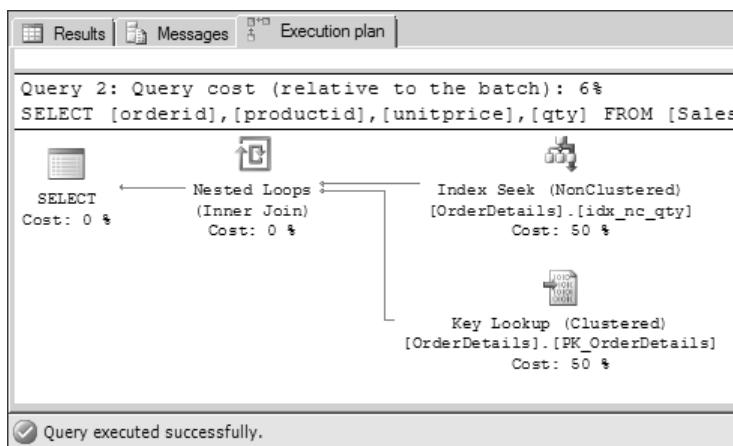


Рис. 17.7. Оператор Index Seek (NonClustered) с частичным просмотром и оператором Key Lookup

На рис. 17.7 показан план выполнения, созданный предыдущим запросом.

Очень редко SQL Server может принять решение использовать просмотр неупорядоченного некластеризованного индекса и затем выполнить либо уточняющий запрос по ключу, либо уточняющий запрос RID в базовой таблице. Чтобы получить такой план, запрос должен быть достаточно выборочным, не должно быть оптимального покрывающего некластеризованного индекса, и используемый индекс не должен поддерживать искомые ключи.

Следующий код чистит базу данных TSQL2012 после тестирования разных методов доступа.

```
DROP INDEX idx_nc_qtyheap ON Sales.OrderDetailsHeap;
DROP INDEX idx_nc_qty ON Sales.OrderDetails;
DROP TABLE Sales.OrderDetailsHeap;
```

## Алгоритмы соединений

SQL Server использует различные алгоритмы при выполнении соединений. SQL Server поддерживает три основных алгоритма: вложенные циклы, соединения слиянием и хэш-соединения. Хэш-соединение может быть далее оптимизировано с помощью *битовой фильтрации*; а хэш-соединение с битовой фильтрацией можно рассматривать как четвертый алгоритм, или как расширение этих трех алгоритмов. Эти три алгоритма рассматриваются в данном разделе, а оптимизация хэш-соединения — в следующем занятии данной главы.

*Алгоритм вложенных циклов* очень прост и во многих случаях эффективен. SQL Server использует одну таблицу для внешнего цикла — как правило, таблицу с несколькими строками. Для каждой строки в этой внешней входной таблице SQL Server ищет соответствующие строки во второй таблице, которая является внутренней. SQL Server использует условие соединения для нахождения совпадающих строк. Это соединение может быть *соединением по неравенству*, что означает, что оператор **Equals** (Равно) не должен быть частью предиката соединения. Если внутренняя таблица не имеет поддерживающего индекса для выполнения поиска, SQL Server просматривает внутренние входные данные для каждой строки внешних входных данных. Этот сценарий нельзя назвать эффективным. Соединение с использованием вложенных циклов эффективно, когда SQL Server может выполнять поиск в индексе во внутренних входных данных. Следующий запрос использует оператор **Nested Loops** (Вложенные циклы) для соединения таблиц Sales.Orders и Sales.OrderDetails. Обратите внимание, запрос фильтрует заказы для создания меньшего набора входных данных; без предложения WHERE, SQL Server использовал бы алгоритм соединения слиянием.

```
SELECT O.custid, O.orderdate, OD.orderid, OD.productid, OD.qty
FROM Sales.Orders AS O
INNER JOIN Sales.OrderDetails AS OD
 ON O.orderid = OD.orderid
WHERE O.orderid < 10250;
```

Запрос создает план выполнения, показанный на рис. 17.8.

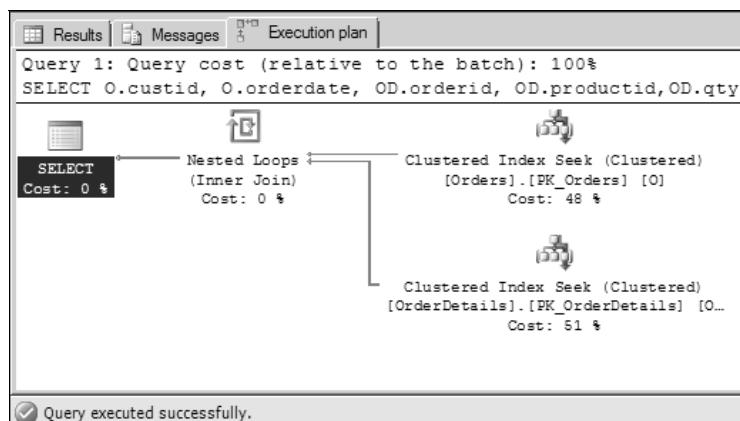


Рис. 17.8. Итератор Nested Loops



Соединение слиянием представляет собой очень эффективный алгоритм соединения. Но он имеет также и ограничения. Ему нужен хотя бы один предикат соединения по равенству и отсортированные входные данные с обеих сторон. Это означает, что соединение слиянием должно поддерживаться индексами в обеих таблицах, участвующих в соединении. Кроме того, если один входной набор намного меньше другого, соединение с использованием вложенных циклов может быть более эффективным, чем соединение слиянием.

В сценарии "один-к-одному" или "один-ко-многим" соединение слиянием просматривает оба входных набора только один раз. Оно начинает с нахождения первых строк с обеих сторон. Если конец входного набора не достигнут, соединение слиянием проверяет предикат соединения для определения, совпадают ли строки. Если строки совпадают, они добавляются в выходные данные. Затем этот алгоритм проверяет следующие строки с обеих сторон и добавляет их в выходной набор до тех пор, пока они удовлетворяют условиям предиката. Если строки из входных данных не совпадают, алгоритм считывает следующую строку на стороне с меньшим значением. Он выполняет чтение на этой стороне и сравнивает строку со строкой на другой стороне, пока значение не станет больше, чем значение на другой стороне. Затем он продолжает чтение с другой стороны и т. д. В сценарии "многие-ко-многим" алгоритм соединения слиянием использует рабочую таблицу, чтобы отложить строки с одной входной стороны для повторного использования, если существуют двойные совпадающие строки в другом входном наборе.

Следующий запрос использует итератор **Merge Join** (Соединение слиянием) для соединения таблиц Sales.Orders и Sales.OrderDetails. Запрос использует соединение по равенству. Оба входа поддерживаются кластеризованным индексом.

```
SELECT O.custid, O.orderdate, OD.orderid, OD.productid, OD.qty
FROM Sales.Orders AS O
INNER JOIN Sales.OrderDetails AS OD
ON O.orderid = OD.orderid;
```

Запрос генерирует план, показанный на рис. 17.9.

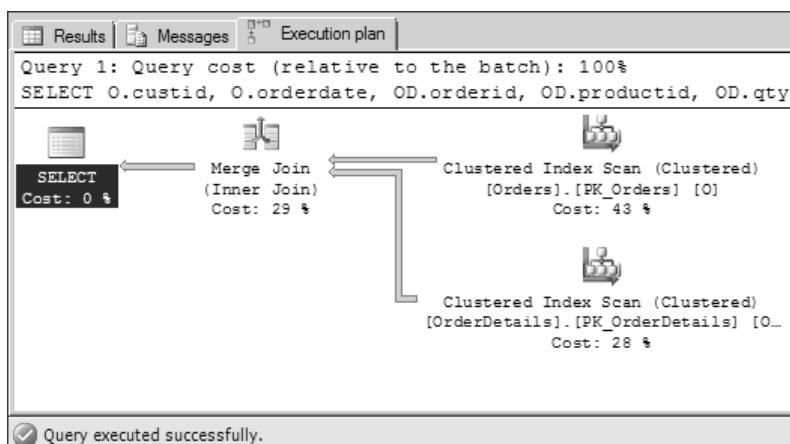


Рис. 17.9. Итератор Merge Join

Если ни один из входов не поддерживается индексом и используется предикат соединения по равенству, тогда более эффективным может быть алгоритм хэш-соединения. Он использует структуру поиска под названием *хэш-таблица*. Этую поисковую структуру вы не можете построить, как сбалансированное дерево, используемое для индексов. SQL Server строит хэш-таблицу внутренним образом. Он использует хэш-функцию для разбиения строк из меньшего входного набора на сегменты. Это этап *построения*. SQL Server использует меньший входной набор для построения хэш-таблицы, поскольку SQL Server хочет держать хэш-таблицу в памяти. Если понадобится сбрасывать ее на диск, алгоритм может стать значительно медленнее. Хэш-функция создает сегменты примерно одинакового размера.

После того как хэш-таблица построена, SQL Server применяет хэш-функцию к каждой строке другого входного набора. Он проверяет, в какой сегмент помещается строка. Затем он просматривает все строки из сегмента. Этот этап называется *пробным*.

Хэш-соединение — это компромисс между созданием индекса полностью сбалансированного дерева с последующим использованием другого алгоритма соединения и выполнением полного просмотра входных данных одной стороны на предмет нахождения каждой строки другого входного набора. По крайней мере на первом этапе используется поиск соответствующего сегмента. Вы можете подумать, что алгоритм хэш-соединения неэффективен. Действительно, в режиме с одним потоком он, как правило, медленнее, чем алгоритмы слияния и соединения с использованием вложенных циклов, поддерживаемые существующими индексами. Но SQL Server может разбить строки из пробного входного набора заранее и выполнить частичные соединения в нескольких потоках. Хэш-соединение в действительности является очень масштабируемым. Этот тип оптимизации хэш-соединения называется *хэш-соединением с битовой фильтрацией*. Обычно оно используется в сценариях хранилищ данных, где имеются большие объемы входных данных в запросах и несколько одновременных пользователей, так что SQL Server может выполнять запрос параллельно. Хотя обычное хэш-соединение также может использоваться параллельно, хэш-соединение с битовой фильтрацией значительно более эффективно, поскольку SQL Server может использовать битовые карты для раннего удаления строк, не используемых в соединении, из больших таблиц, участвующих в соединении.

Следующие два запроса создают две кучи, которые не имеют индекса, из таблиц Sales.Orders и Sales.OrderDetails.

```
SELECT orderid, productid, unitprice, qty, discount
INTO Sales.OrderDetailsHeap
FROM Sales.OrderDetails;
SELECT orderid, custid, orderdate
INTO Sales.OrdersHeap
FROM Sales.Orders;
```



Следующий запрос использует алгоритм хэш-соединения для соединения таблиц.

```
SELECT O.custid, O.orderdate, OD.orderid, OD.productid, OD.qty
FROM Sales.OrdersHeap AS O
```

```
INNER JOIN Sales.OrderDetailsHeap AS OD
ON O.orderid = OD.orderid;
```

Результатом запроса является план, показанный на рис. 17.10.

Следующий код чистит базу данных TSQL2012 после тестирования разных алгоритмов соединений.

```
DROP TABLE Sales.OrderDetailsHeap;
DROP TABLE Sales.OrdersHeap;
```

### **COBET**

### **Подготовка к экзамену**

Только алгоритм соединения с использованием вложенных циклов поддерживает соединения по неравенству.

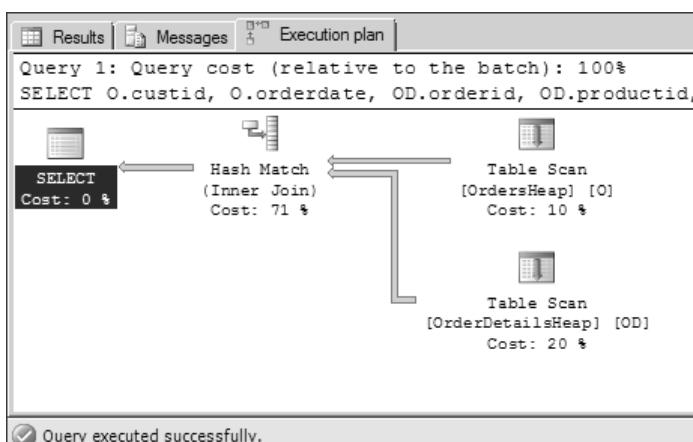


Рис. 17.10. Итератор Hash Match (Inner Join) — итератор, выполняющий хэш-соединение

## **Другие итераторы плана выполнения**

Существует множество других итераторов плана выполнения. В данном занятии представлены три дополнительных важных итератора.

### **К СВЕДЕНИЮ Значки операторов плана выполнения**

Полный список операторов плана выполнения и их графических обозначений, используемых в SQL Server 2012, можно найти в электронной документации к SQL Server 2012 в разделе "Справочник по логическим и физическим операторам Showplan" по адресу <http://msdn.microsoft.com/ru-ru/library/ms191158.aspx>.

SQL Server использует оператор **Sort** (Сортировать) при необходимости сортировать входные данные. Существует много причин сортировать входные данные. Например, SQL Server может решить отсортировать входные данные так, чтобы использовать алгоритм соединения слиянием. Очень типичный пример использования оператора **Sort** (Сортировать) — для запросов, которые запрашивают упорядоченный набор строк, когда порядок не поддерживается индексом. Оператор сорти-

ровки может быть очень дорогим; чтобы обеспечить высокую производительность, надо быть уверенным, что оператор **Sort** (Сортировать) используется только для небольших сходных наборов. Следующий код запрашивает упорядоченный набор строк. Набор строк должен быть упорядочен по столбцу *qty* таблицы Sales.*OrderDetails*. Но у таблицы нет индекса на этом столбце.

```
SELECT orderid, productid, qty
FROM Sales.OrderDetails
ORDER BY qty;
```

На рис. 17.11 представлен план выполнения этого запроса. Обратите внимание, стоимость этого оператора **Sort** (Сортировать) составляет около 81% от общей стоимости запроса.

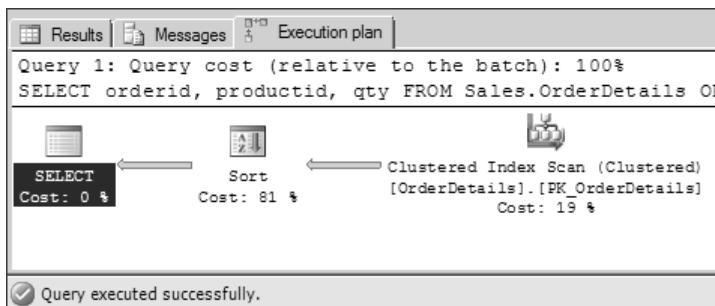


Рис. 17.11. Итератор Sort

SQL Server использует еще два алгоритма для статистических расчетов. Если входные данные упорядочены по столбцам, указанным в предложении GROUP BY, SQL Server использует *алгоритм статистической обработки потока*, реализованный оператором **Stream Aggregate** (Статистическое выражение потока). Статистическая обработка потока очень эффективна. SQL Server может принять решение сортировать входные данные перед выполнением статистической обработки, чтобы была возможность использования оператора **Stream Aggregate**.

Следующий запрос использует оператор **Stream Aggregate**. Обратите внимание, он группирует строки таблицы Sales.OrderDetails по столбцу *productid*. Некластеризованный индекс существует на этом столбце. Кроме того, поскольку запрос не требует других столбцов, индекс является покрывающим.

```
SELECT productid, COUNT(*) AS num
FROM Sales.OrderDetails
GROUP BY productid;
```

На рис. 17.12 представлен план выполнения этого запроса.

Если входные данные для статистической обработки не отсортированы и их настолько много, что сортировка была бы неэффективной, SQL Server использует *алгоритм статистической обработки хэша*. Для этого типа статистической обработки используется оператор **Hash Match Aggregate** (Статистическая обработка хэш-совпадений). Его значок такой же, как у оператора **Hash Match**.

**Join** (Соединение с хэш-совпадениями). Алгоритм статистической обработки строит хэш-таблицу из входных данных так же, как для хэш-соединения. Но сегменты используются для хранения групп.

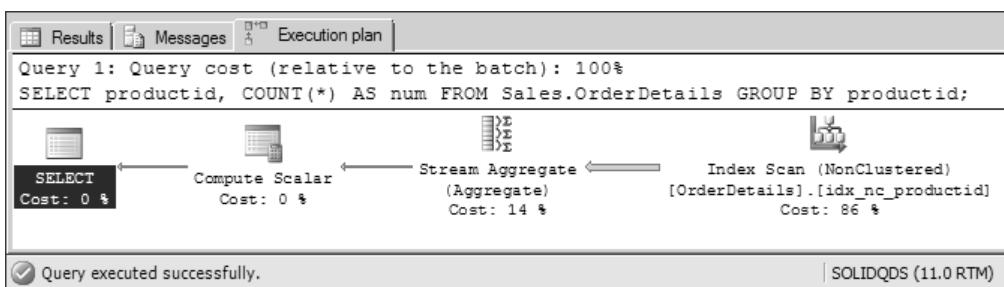


Рис. 17.12. Итератор Stream Aggregate

Аналогично хэш-соединению, статистическая обработка хэша также является масштабируемой. Подобно алгоритму статистической обработки потока, алгоритм статистической обработки хэша может вычислять несколько групп одновременно в нескольких потоках. Следующий запрос группирует строки из таблицы Sales.OrderDetails по столбцу qty; статистическая обработка не поддерживается индексом.

```
SELECT qty, COUNT(*) AS num
FROM Sales.OrderDetails
GROUP BY qty;
```

На рис. 17.13 представлен план выполнения этого запроса. Обратите внимание, SQL Server использовал оператор **Hash Match (Aggregate)** (Статистика по хэш-совпадениям). Также следует заметить, что относительная стоимость статистической обработки хэша значительно выше, чем статистической обработки потока, ранее показанной на рис. 17.12. Если статистическая обработка потока составляет примерно 14% от общей стоимости запроса, то статистическая обработка хэша составляет 71%.

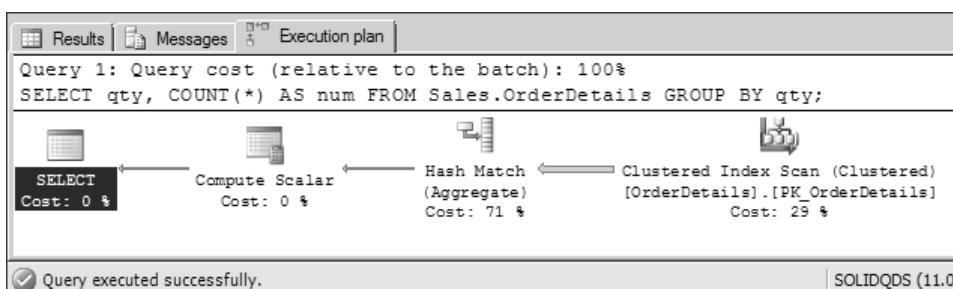


Рис. 17.13. Итератор Hash Match (Aggregate)

## ПРАКТИКУМ Определение итераторов плана выполнения

В этом практикуме вам нужно проанализировать несколько запросов.

### Задание 1. Прогноз плана выполнения

В этом задании вам нужно выполнить несколько различных запросов в контексте базы данных TSQL2012. Но прежде попробуйте определить, какие итераторы будет использовать SQL Server. Затем выполните запросы и проверьте действительный план выполнения.

1. Запустите SQL Server Management Studio (SSMS) и подключитесь к вашему экземпляру SQL Server.
2. Откройте новое окно запроса, нажав кнопку **New Query** (Создать запрос).
3. Измените контекст на базу данных TSQL2012.
4. Проанализируйте столбцы и индексы таблиц `Sales.Customers` и `Sales.Orders`. Посмотрите на следующий запрос:

```
SELECT C.custid, C.companyname, C.address, C.city,
 O.orderid, O.orderdate
 FROM Sales.Customers AS C
 INNER JOIN Sales.Orders AS O
 ON C.custid = O.custid;
```

Какие операторы вы ожидаете увидеть в плане выполнения? Какой алгоритм соединения должен использовать SQL Server? Ожидаете ли вы увидеть итератор **Sort** (Сортировать) в плане выполнения?

5. Включите действительный план выполнения и выполните запрос. На рис. 17.14 представлен план выполнения этого запроса.

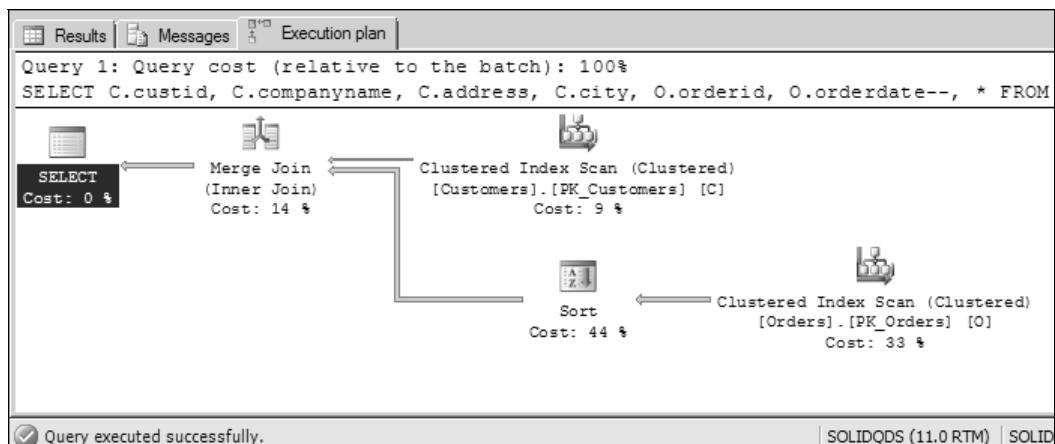


Рис. 17.14. Действительный план выполнения запроса из шага 4

Возможно, вы правильно предположили, что SQL Server будет просматривать кластеризованную таблицу Sales.Customers. Вы могли ожидать, что может использоваться некластеризованный индекс на столбце custid таблицы Sales.Orders. Но запрос также включил столбец orderdate, и этот некластеризованный индекс не мог покрыть запрос. Поэтому SQL Server нужно было использовать оператор **Key Lookup** (Уточняющий запрос ключа). Оптимизатор запросов решил, что дешевле выполнить просмотр кластеризованного индекса на таблице Sales.Orders, отсортировать строки и затем использовать алгоритм соединения слиянием.

Следующий запрос не включает столбец orderdate. Какого типа операторы плана выполнения вы предполагаете использовать в этом запросе?

```
SELECT C.custid, C.companyname, C.address, C.city, O.orderid
FROM Sales.Customers AS C
INNER JOIN Sales.Orders AS O
 ON C.custid = O.custid;
```

6. Выполните запрос и проверьте план выполнения. Как вы, возможно, предполагали, SQL Server выполнил просмотр кластеризованной таблицы Sales.Customers, далее — некластеризованного покрывающего индекса на столбце orderdate таблицы Sales.Orders, а затем использовал итератор **Merge Join** (Соединение слиянием) для соединения данных. На рис. 17.15 представлен план выполнения этого запроса.

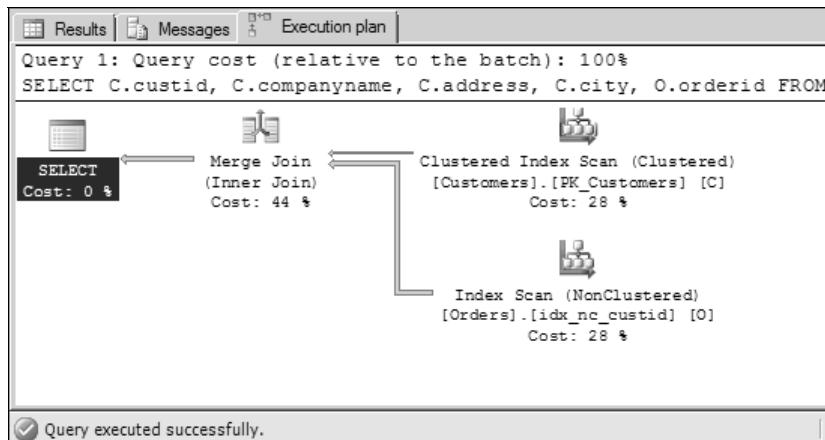


Рис. 17.15. Действительный план выполнения для запроса из шага 5

## Задание 2. Анализ плана выполнения

В этом задании вам нужно выполнить несколько различных запросов в контексте базы данных TSQL2012 и проанализировать действительный план выполнения.

1. Выполните следующий запрос и проверьте план выполнения. Обратите внимание, запрос является достаточно выборочным; клиенты из Берлина не имеют много заказов.

```

SELECT C.custid, C.companyname, C.address, C.city,
 O.orderid, O.orderdate
 FROM Sales.Customers AS C
 INNER JOIN Sales.Orders AS O
 ON C.custid = O.custid
 WHERE C.city = N'Berlin';

```

Поскольку запрос является очень выборочным, SQL Server принял решение, что оператор **Key Lookup** (Уточняющий запрос ключа) не должен быть дорогим. Обратите внимание, даже всего лишь при 6 возвращенных строках с всего лишь 7 уточняющими запросами по ключу, стоимость оператора **Key Lookup** (Уточняющий запрос ключа) составляет примерно 74% от полной стоимости запроса (рис. 17.16).

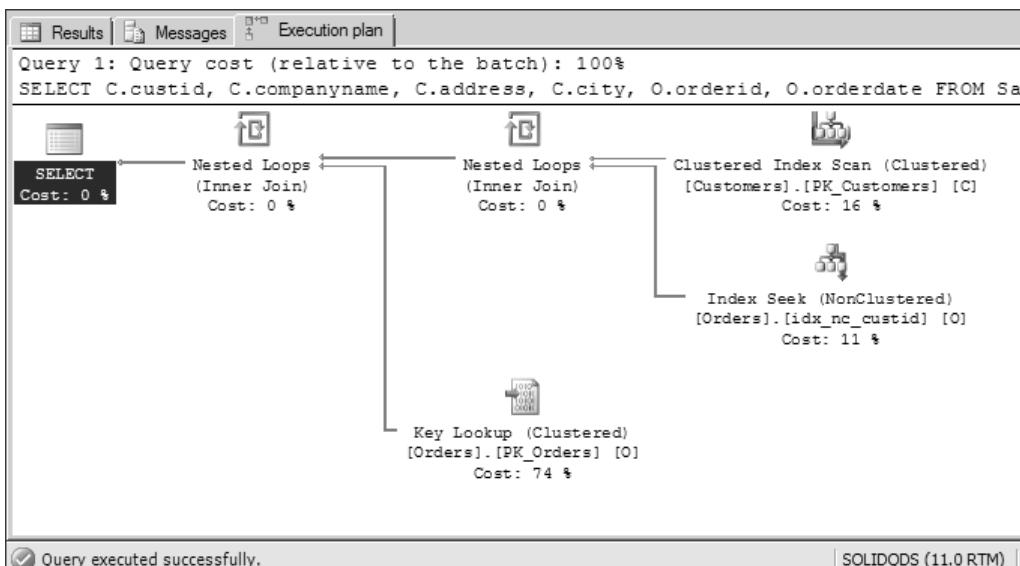


Рис. 17.16. Действительный план выполнения для запроса из шага 1

2. Проверьте следующий запрос и попытайтесь выяснить, какой тип плана выполнения может использовать SQL Server.

```

SELECT C.city, MIN(O.orderid) AS minorderid
 FROM Sales.Customers AS C
 INNER JOIN Sales.Orders AS O
 ON C.custid = O.custid
 GROUP BY C.city;

```

Обратите внимание, запрос покрыт двумя некластеризованными индексами. Один некластеризованный индекс имеется на столбце city для таблицы Sales.Customers, которая также включает столбец custid из кластеризованного первичного ключа, а второй некластеризованный индекс существует на столбце custid для таблицы Sales.Orders. Поскольку входные данные для статистической обработки отсортированы, SQL Server использует итератор **Stream**

**Aggregate** (Статистическое выражение потока). На рис. 17.17 представлен план выполнения этого запроса.

3. Закройте окно запроса.

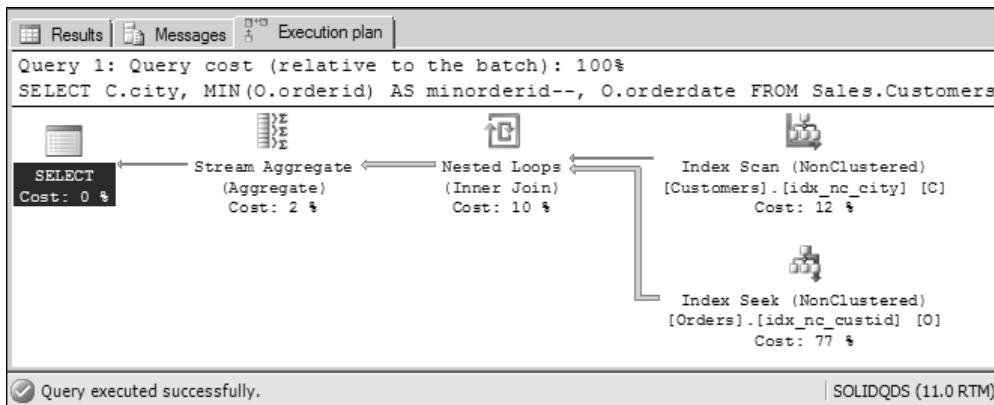


Рис. 17.17. Действительный план выполнения для запроса из шага 2

## Резюме занятия

- SQL Server использует много различных методов доступа к данным.
- SQL Server использует разные алгоритмы соединений и статистической обработки.
- Не существует "хороших" и "плохих" итераторов. Любой итератор может подходить наилучшим образом для конкретного запроса и конкретных данных.

## Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в приложении "Ответы" в конце книги.

1. Какие алгоритмы статистической обработки использует SQL Server? (Выберите все подходящие варианты.)
  - A. Статистическая обработка слияний.
  - B. Статистическая обработка потока.
  - C. Статистическая обработка хэша.
  - D. Статистическая обработка вложенных циклов.
2. Какой оператор используется, когда SQL Server выполняет поиск по некластеризованному индексу для нахождения строки, но также ему нужны данные из базовой таблицы, организованной как кластеризованный индекс?
  - A. Уточняющий запрос RID.
  - B. Просмотр кластеризованного индекса.

- С. Соединение слиянием.
- D. Уточняющий запрос по ключу.
3. Как называется просмотр, когда SQL Server просматривает кластеризованный индекс в логической последовательности индекса?
- A. Просмотр в порядке выделения.
- B. Просмотр по кластеризованному индексу.
- C. Просмотр в порядке индекса.
- D. Поиск в порядке индекса.

## Занятие 2. Использование параметризованных запросов и пакетных операций

Как вы узнали из главы 14, оптимизатор запросов SQL Server проделывает большую работу для определения хорошего плана выполнения. После того как план построен, SQL Server помещает его в кэш для повторного использования при необходимости. SQL Server старается параметризовать запросы и таким образом увеличить вероятность повторного использования плана. Вы можете помочь SQL Server, если выполните параметризацию самостоятельно.

В среде OLTP большинство работы по оптимизации связано со снижением дискового ввода-вывода. В сценариях хранилищ данных запросы обычно считывают большие объемы данных, выполняя при этом просмотры данных. Но запросы к хранилищам данных часто выполняются в параллельном режиме. Таким образом, производительность ЦП может резко понизиться. В SQL Server 2012 представлен режим пакетной обработки, уменьшающий нагрузку на ЦП. Он очень полезен при использовании с индексами columnstore.

**Изучив материал этого занятия, вы сможете:**

- ✓ Объяснить параметризацию запросов
- ✓ Объяснить обработку строк пакетов
- ✓ Использовать итераторы обработки пакетов

**Продолжительность занятия — 40 минут.**

### Параметризованные запросы

SQL Server выполняет параметризацию оперативных запросов автоматически. Но он очень консервативен в отношении повторного использования планов. Он не хочет использовать плохой план. SQL Server принимает решение повторно использовать план, только если уверен в том, что кэшированный план правильный и подходит для запроса. Изменение типа данных параметра, некоторых параметров инструкции SET, в настройках безопасности и т. д. могут привести к созданию нового плана, когда вы ожидаете повторного использования существующего кэшированного плана.

Вы можете получить информацию о кэшированных планах и количестве их повторных использований, запросив функцию динамического управления sys.dm\_exec\_query\_stats. Функция динамического управления sys.dm\_exec\_sql\_text может предоставить точный текст запроса. Когда вы тестируете кэширование и повторное использование плана, можете воспользоваться командой DBCC FREEPROCCACHE языка T-SQL для очистки кэша, как показано в следующем примере:

```
DBCC FREEPROCCACHE;
```

**ВАЖНО!****Использование команды DBCC FREEPROCCACHE  
в рабочей среде**

---

Команда T-SQL DBCC FREEPROCCACHE очень полезна для тестирования. Но следует соблюдать большую осторожность при ее использовании в рабочей среде. Если вы очистите кэш на рабочем сервере, SQL Server должен будет оптимизировать и скомпилировать новые последовательные запросы, хранимые процедуры, функции и триггеры. Пользователи могут почувствовать серьезное снижение производительности.

Каждый из следующих трех запросов извлекает одну строку из таблицы Sales.Orders в базе данных TSQL2012. Все они используют в качестве параметра предложения WHERE столбец orderid. Таблица имеет первичный ключ, определенный на столбце orderid. Таким образом, SQL Server знает, что запрос возвращает только одну строку. Но первые два запроса используют для параметра целое число, тогда как третий запрос определяет для этого параметра десятичное число.

```
-- Параметр INT
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderid = 10248;

-- Параметр INT
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderid = 10249;

-- Параметр DECIMAL
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderid = 10250.0;
```

Вы можете проверить планы в кэше и количество выполнений с помощью следующего запроса:

```
SELECT qs.execution_count AS cnt, qt.text
FROM sys.dm_exec_query_stats AS qs
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) AS qt
WHERE qt.text LIKE N'%Orders%'
AND qt.text NOT LIKE N'%qs.execution_count%'
ORDER BY qs.execution_count;
```

Поскольку этот запрос используется в последующих примерах в данном занятии, мы будем называть его запросом "плана повторного использования". Запрос плана повторного использования возвращает следующий результат:

```
cnt text
--- -----
1 (@1 numeric(6,1))SELECT [orderid],[custid],[empid],[orderdate] FROM
 [Sales].[Orders] WHERE [orderid]=@1
2 (@1 smallint)SELECT [orderid],[custid],[empid],[orderdate] FROM
 [Sales].[Orders] WHERE [orderid]=@1
```

Вы видите, что SQL Server выполнил параметризацию этих запросов. Он повторно использовал план выполнения там, где параметр был целым числом. Там, где параметр был десятичным числом, SQL Server сгенерировал новый план.

Чтобы продемонстрировать, насколько SQL Server консервативен в вопросах повторного использования плана, рассмотрим следующие три запроса, использующие столбец `custid` в предложении `WHERE`. Первый запрос возвращает 1 строку, второй — 2 строки, третий — 31 строку. SQL Server не может быть уверен в выборочности столбца `custid` в таблице `Sales.Orders`.

```
-- 1 строка
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE custid = 13;
GO

-- 2 строки
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE custid = 33;
GO

-- 31 строка
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE custid = 71;
GO
```

Если вы очистили кэш до того, как выполните предыдущие три запроса, то запрос повторного использования плана возвратит следующие выходные данные:

```
cnt text
--- -----
1 SELECT orderid, custid, empid, orderdate FROM Sales.Orders
 WHERE custid = 33;
1 SELECT orderid, custid, empid, orderdate FROM Sales.Orders
 WHERE custid = 13;
1 SELECT orderid, custid, empid, orderdate FROM Sales.Orders
 WHERE custid = 71;
```

Обратите внимание, запросы не были параметризованы.

### **К СВЕДЕНИЮ   Почему SQL Server иногда не параметризирует запросы**

Существует множество причин, по которым SQL Server не выполняет параметризацию запросов. Их полный список можно найти в *приложении A* к разделу MSDN "Plan Caching

in SQL Server 2008" по адресу [http://msdn.microsoft.com/en-us/library/ee343986\(SQL.100\).aspx](http://msdn.microsoft.com/en-us/library/ee343986(SQL.100).aspx) (хотя статья написана для SQL Server 2008, она также подходит для SQL Server 2012).

Покажем, что влияет на повторное использование плана. Для этого возьмем следующие два запроса — те же самые, что два первых запроса из первого примера в данном разделе, когда повторно использовался план. Но параметр SET, который может влиять на результат запроса, изменен перед выполнением второго запроса.

```
-- Параметризованный запрос
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderid = 10248;

-- Изменения в параметре SET
SET CONCAT_NULL_YIELDS_NULL OFF;
-- Запрос, который мог бы использовать тот же план
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderid = 10249;

-- Восстановление параметра SET
SET CONCAT_NULL_YIELDS_NULL ON;
```

Если вы очистили кэш до того, как выполнили предыдущие три запроса, запрос плана повторного использования возвратит следующий результат:

| cnt | text                                                                                                    |
|-----|---------------------------------------------------------------------------------------------------------|
| 1   | (@1 numeric(6,1))SELECT [orderid],[custid],[empid],[orderdate] FROM [Sales].[Orders] WHERE [orderid]=@1 |
| 2   | (@1 smallint)SELECT [orderid],[custid],[empid],[orderdate] FROM [Sales].[Orders] WHERE [orderid]=@1     |

Хотя SQL Server выполнил параметризацию запросов, он не использовал первый план повторно.

Вам может показаться, что SQL Server слишком консервативен в отношении повторного использования планов. Но все не так плохо. Большинство запросов поступает от приложений, и приложения обычно всегда генерируют свои запросы одинаково, с теми же параметрами. Кроме того, вы можете помочь SQL Server, используя системную процедуру `sys.sp_executesql` для выполнения параметризованного динамического SQL. В действительности, следует рассмотреть использование системной процедуры `sys.sp_executesql` как значительно более правильное решение, чем использование оперативных запросов.

В следующем примере параметризованный запрос создан как SQL-строка и затем дважды выполнен с использованием разных параметров. В первом случае параметр — целое число, во втором — десятичное число.

```
DECLARE @v INT;
DECLARE @s NVARCHAR(500);
DECLARE @p NVARCHAR(500);
```

```
-- Построение SQL-строки
SET @s = N'
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderid = @orderid';
SET @p = N'@orderid INT';
-- Параметр - целое число
SET @v = 10248;
EXECUTE sys.sp_executesql @s, @p, @orderid = @v;
-- Параметр - десятичное число
SET @v = 10249.0;
EXECUTE sp_executesql @s, @p, @orderid = @v;
```

Если вы дважды очистили кэш перед вызовом процедуры sys.sp\_executesql, запрос плана повторного использования возвратит следующий результат:

```
cnt text
--- ---
2 (@orderid INT) SELECT orderid, custid, empid, orderdate FROM
 Sales.Orders WHERE orderid = @orderid
```

Обратите внимание, хотя второй выход неявно использовал десятичный параметр, SQL Server знал, что в действительности параметр был целым числом, поскольку он был явно определен как целое число для процедуры sys.sp\_executesql. Разумеется, значение должно было быть неявно конвертируемым в тип данных INTEGER или второй запрос закончился бы ошибкой.

Использование динамического SQL — не лучшее решение. Чтобы инициировать повторное использование плана, нужно использовать программные объекты, такие как хранимые процедуры. Следующий код создает процедуру, которая упаковывает запрос, извлекающий один заказ, в хранимую процедуру.

```
CREATE PROCEDURE Sales.GetOrder
(@orderid INT)
AS
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderid = @orderid;
```

Вы можете дважды вызвать эту процедуру, один раз — неявно передавая параметр как целое число, второй раз — как десятичное число.

```
EXEC Sales.GetOrder @orderid = 10248;
EXEC Sales.GetOrder @orderid = 10249.0;
```

Если вы дважды очистили кэш перед вызовом процедуры, запрос плана повторного использования возвращает следующий результат:

```
cnt text
--- ---
2 CREATE PROCEDURE Sales.GetOrder (@orderid INT) AS SELECT orderid,
 custid, empid, orderdate
 FROM Sales.Orders WHERE orderid = @orderid;
```

Вы видите, что план с успехом использовался повторно. Хранимые процедуры инициируют повторное использование планов. Но иногда может оказаться предпочтительным, чтобы последовательные вызовы хранимой процедуры не использовали кэшированный план. Процедура может возвращать разное число строк в зависимости от значения параметра. Для некоторых значений запрос внутри процедуры может быть очень выборочным, а для других значений — не выборочным совсем. Поэтому, возможно, вы захотите иметь разные планы выполнения для каждого вызова. Вы можете побудить SQL Server перекомпилировать хранимую процедуру, если создадите ее с параметром `WITH RECOMPILE`. Кроме того, повторную компиляцию можно вызвать на уровне запроса. Вместо повторной компиляции всей процедуры вы можете перекомпилировать только критические инструкции. Повторная компиляция процедуры рассматривается в практикуме к данному занятию, а повторная компиляция запросов — в следующем занятии данной главы.

После тестирования процедуры следует очистить базу данных TSQL2012.

```
DROP PROCEDURE Sales.GetOrder;
```

## Обработка пакетов

В занятии 1 рассматривались три основных алгоритма соединений, используемых в SQL Server. Вы также узнали, что хэш-соединение может быть далее оптимизировано с помощью выполнения частичных соединений в нескольких потоках, и даже с ранним удалением строк, использованных в соединении, из большей таблицы, с помощью битовых фильтров. Это называется хэш-соединением с битовой фильтрацией. Вы уже знаете, что оно обычно используется в сценариях хранилищ данных, где возможны большие объемы данных на входе запроса и несколько одновременных пользователей, так что SQL Server может выполнять запрос в параллельном режиме. Такой тип соединения также иногда называют *соединением типа "звезда"* в честь типичной схемы хранилищ данных, напоминающей звезду с одной центральной таблицей на стороне "многие" отношений и несколькими связанными окружающими таблицами на стороне "один" отношений. Следующий код создает четыре таблицы в очень простой схеме типа "звезда" и заполняет их большим количеством строк. Он использует вспомогательную табличную функцию, которая возвращает таблицу чисел, используемую для заполнения таблиц хранилища данных.

```
-- Распределение данных для хранилища данных
DECLARE
 @dim1rows AS INT = 100
 @dim2rows AS INT = 50
 @dim3rows AS INT = 200
-- Первое измерение
CREATE TABLE dbo.Dim1
(key1 INT NOT NULL CONSTRAINT PK_Dim1 PRIMARY KEY,
attr1 INT NOT NULL,
filler BINARY(100) NOT NULL DEFAULT (0x));
```

```
-- Второе измерение
CREATE TABLE dbo.Dim2
(key2 INT NOT NULL CONSTRAINT PK_Dim2 PRIMARY KEY,
attr1 INT NOT NULL,
filler BINARY(100) NOT NULL DEFAULT (0x));
-- Третье измерение
CREATE TABLE dbo.Dim3
(key3 INT NOT NULL CONSTRAINT PK_Dim3 PRIMARY KEY,
attr1 INT NOT NULL,
filler BINARY(100) NOT NULL DEFAULT (0x));
-- Таблица фактов
CREATE TABLE dbo.Fact
(
key1 INT NOT NULL CONSTRAINT FK_Fact_Dim1 FOREIGN KEY REFERENCES dbo.Dim1,
key2 INT NOT NULL CONSTRAINT FK_Fact_Dim2 FOREIGN KEY REFERENCES dbo.Dim2,
key3 INT NOT NULL CONSTRAINT FK_Fact_Dim3 FOREIGN KEY REFERENCES dbo.Dim3,
measure1 INT NOT NULL,
measure2 INT NOT NULL,
measure3 INT NOT NULL,
filler BINARY(100) NOT NULL DEFAULT (0x),
CONSTRAINT PK_Fact PRIMARY KEY(key1, key2, key3)
);
-- Заполнение первого измерения
INSERT INTO dbo.Dim1(key1, attr1)
SELECT n, ABS(CHECKSUM(NEWID()))% 20 + 1
FROM dbo.GetNums(1, @dim1rows);
-- Заполнение второго измерения
INSERT INTO dbo.Dim2(key2, attr1)
SELECT n, ABS(CHECKSUM(NEWID()))% 10 + 1
FROM dbo.GetNums(1, @dim2rows);
-- Заполнение третьего измерения
INSERT INTO dbo.Dim3(key3, attr1)
SELECT n, ABS(CHECKSUM(NEWID()))% 40 + 1
FROM dbo.GetNums(1, @dim3rows);
-- Заполнение таблицы фактов
INSERT INTO dbo.Fact WITH (TABLOCK)
(key1, key2, key3, measure1, measure2, measure3)
SELECT N1.n, N2.n, N3.n,
ABS(CHECKSUM(NEWID())) % 1000000 + 1,
ABS(CHECKSUM(NEWID())) % 1000000 + 1,
ABS(CHECKSUM(NEWID())) % 1000000 + 1
FROM dbo.GetNums(1, @dim1rows) AS N1
CROSS JOIN dbo.GetNums(1, @dim2rows) AS N2
CROSS JOIN dbo.GetNums(1, @dim3rows) AS N3;
```

На рис. 17.18 показана схема для этих четырех таблиц. Обратите внимание, схема напоминает звезду. Именно поэтому она называется схемой типа "звезда".

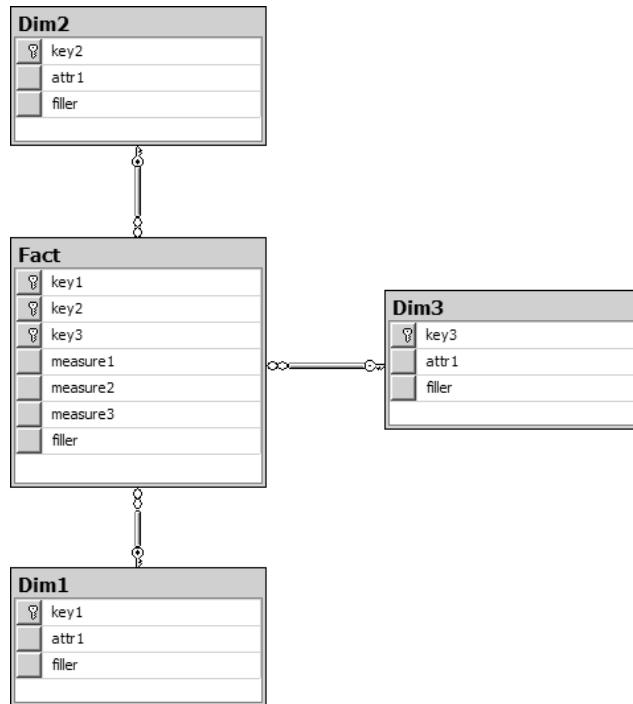


Рис. 17.18. Схема типа "звезды"

Следующий запрос соединяет все четыре таблицы и выполняет статистическую обработку данных. Также измеряются статистика ввода-вывода и времена ЦП (параметры STATISTICS IO и STATISTICS TIME).

```

-- Измерение ввода-вывода и времени
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
-- Запрос, демонстрирующий соединение типа "звезда"
SELECT D1.attr1 AS x, D2.attr1 AS y, D3.attr1 AS z,
 COUNT(*) AS cnt, SUM(F.measure1) AS total
 FROM dbo.Fact AS F
 INNER JOIN dbo.Dim1 AS D1
 ON F.key1 = D1.key1
 INNER JOIN dbo.Dim2 AS D2
 ON F.key2 = D2.key2
 INNER JOIN dbo.Dim3 AS D3
 ON F.key3 = D3.key3
 WHERE D1.attr1 <= 10
 AND D2.attr1 <= 15
 AND D3.attr1 <= 10
 GROUP BY D1.attr1, D2.attr1, D3.attr1;

```

Запрос был выполнен на компьютере с четырехъядерным процессором и использованием технологии hyper-threading (гиперпоточность). SQL Server использовал

8 логических процессоров. На рис. 17.19 представлен частичный план выполнения этого запроса. Обратите внимание, запрос выполнялся в параллельном режиме (итератор **Parallelism** (Параллелизм), использовалось хэш-соединение (итератор **Hash Match**), а также в одном случае перед выполнением соединения (оператор **Bitmap**) использовался фильтр по битовым картам. В SQL Server 2012 также есть два дополнительных свойства для итераторов. На рис. 17.19 вы видите свойства **Actual Execution Mode** (Действительный режим выполнения) и **Estimated Execution Mode** (Предполагаемый режим выполнения) для оператора **Hash Match (Inner Join)** (Внутреннее соединение по хэш-совпадениям). Значением для этих двух свойств является **Row** (Строка).

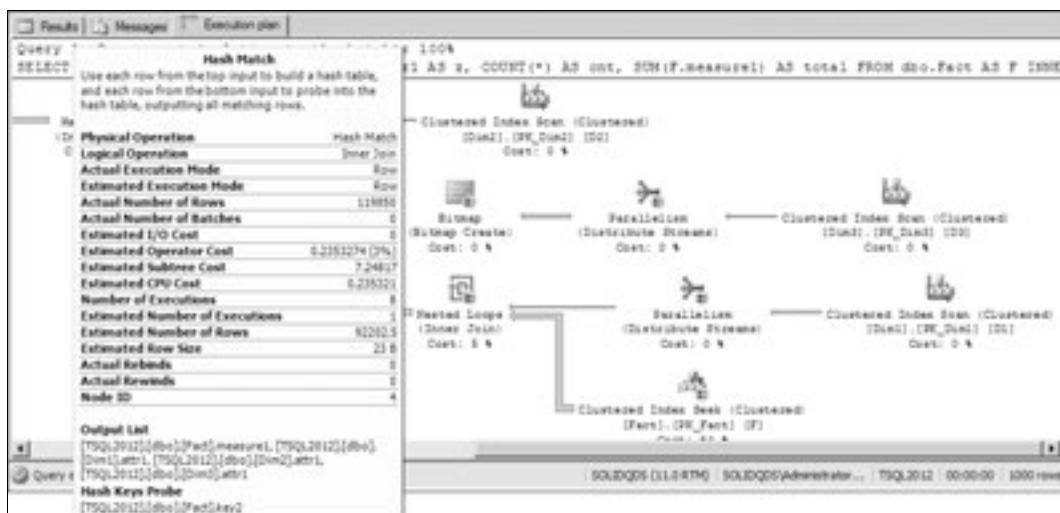


Рис. 17.19. Частичный план для соединения типа "звезда"

Вот как выглядят сокращенные результаты расчета статистики ввода-вывода и времени ЦП (параметры STATISTICS IO и STATISTICS TIME) для данного результата.

```
Table 'Dim2'. Scan count 1, logical reads 2, ...
Table 'Dim3'. Scan count 1, logical reads 5, ...
Table 'Dim1'. Scan count 1, logical reads 4, ...
Table 'Worktable'. Scan count 0, logical reads 0, ...
Table 'Fact'. Scan count 47, logical reads 8152, ...
Table 'Worktable'. Scan count 0, logical reads 0, ...
SQL Server Execution Times:
 CPU time = 671 ms, elapsed time = 255 ms.
```

Обратите внимание на огромное количество операций логического чтения в таблице dbo.Fact. Кроме того, заметьте, что время ЦП почти в три раза больше, чем время исполнения этого запроса. Поскольку запрос выполнялся в параллельном режиме, нагрузка на ЦП была очень высокой, и ЦП мог стать узким местом для этого сценария. Представьте, что было бы, если бы таблица была еще и скатая. Тогда SQL Server должен был бы ее распаковать. Вы можете также создать на этой таблице

индекс columnstore. Тогда SQL Server будет должен повторно создать строки для выходного набора. Подводя итог, ЦП может стать узким местом в сценарии хранилища данных.

SQL Server 2012 предлагает решение проблемы нагрузки на ЦП. Он позволяет использовать итераторы, которые обрабатывают не каждую строку по отдельности, а пакет строк за один раз. Таким образом, ЦП должен обрабатывать метаданные для строки только один раз на пакет. Пакетная обработка ортогональна по отношению к индексам; она также может поддерживать хранение строк. Но наилучшие результаты дает использование индексов columnstore. С помощью индексов columnstore SQL Server иногда может выполнять пакетные операции непосредственно на сжатых данных, пропуская таким образом операцию их распаковки. SQL Server 2012 может использовать одновременно пакетные и строковые операторы и динамически переключаться с пакетного режима на строковый.

Следующие операторы поддерживают обработку в пакетном режиме в SQL Server 2012:

- Filter** (Фильтр);
- Project** (Проект);
- Scan** (Просмотр);
- Local hash (partial) aggregation** (Логическая (частичная) статистическая обработка хэша);
- Hash inner join** (Внутреннее хэш-соединение);
- Batch hash table build** (Построение хэш-таблицы пакета).

Чтобы проверить работу пакетных операторов, следующий код строит индекс columnstore на таблице dbo.Fact.

```
CREATE COLUMNSTORE INDEX idx_cs_fact
 ON dbo.Fact(key1, key2, key3, measure1, measure2, measure3);
```

После создания индекса columnstore снова выполните тот же запрос типа "звезда".

```
SELECT D1.attr1 AS x, D2.attr1 AS y, D3.attr1 AS z,
 COUNT(*) AS cnt, SUM(F.measure1) AS total
FROM dbo.Fact AS F
 INNER JOIN dbo.Dim1 AS D1
 ON F.key1 = D1.key1
 INNER JOIN dbo.Dim2 AS D2
 ON F.key2 = D2.key2
 INNER JOIN dbo.Dim3 AS D3
 ON F.key3 = D3.key3
WHERE D1.attr1 <= 10
 AND D2.attr1 <= 15
 AND D3.attr1 <= 10
GROUP BY D1.attr1, D2.attr1, D3.attr1;
```

На рис. 17.20 показан частичный план выполнения запроса типа "звезда". Обратите внимание, запрос использует оператор **Columnstore Index Scan** (Просмотр индекса

columnstore). Кроме того, как видно из свойств одного из операторов **Hash Match (Inner Join)** (Внутреннее соединение по хэш-совпадениям), SQL Server использует пакетный режим выполнения.

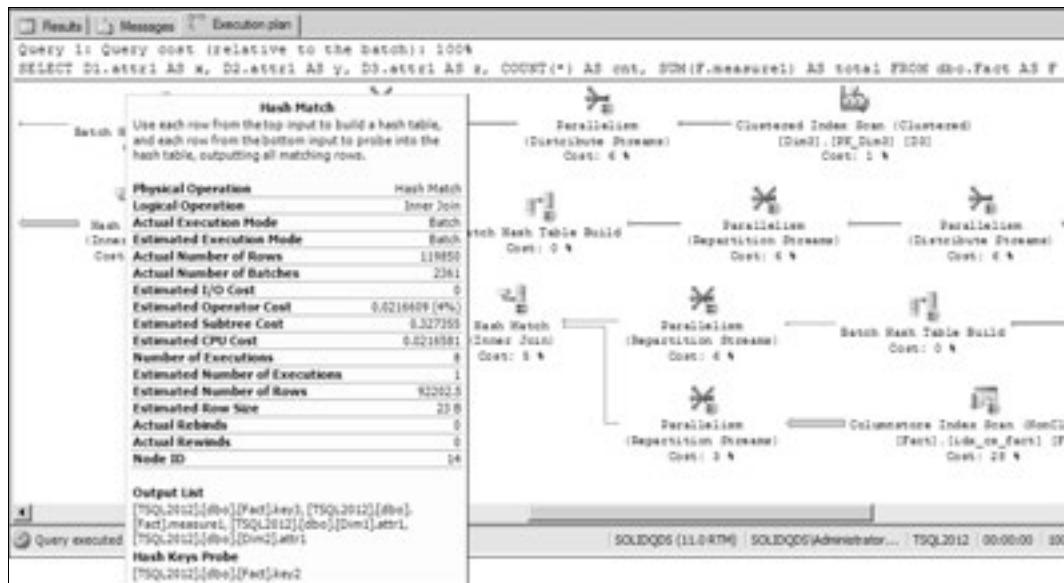


Рис. 17.20. Частичный план запроса соединения типа "звезды", который использует индекс columnstore и пакетную обработку

Также проверьте сокращенные результаты расчета статистики ввода-вывода и времени ЦП этого выполнения.

```

Table 'Dim3'. Scan count 1, logical reads 5, ...
Table 'Dim2'. Scan count 1, logical reads 2, ...
Table 'Dim1'. Scan count 1, logical reads 4, ...
Table 'Fact'. Scan count 8, logical reads 993, ...
Table 'Worktable'. Scan count 0, logical reads 0, ...
Table 'Worktable'. Scan count 0, logical reads 0, ...
SQL Server Execution Times:
CPU time = 63 ms, elapsed time = 186 ms.

```

Вы видите, что количество логических операций ввода-вывода в таблице `dbo.Fact` значительно снизилось по сравнению с первым выполнением, где индекс columnstore не использовался. Время выполнения стало меньше, чем было при первом выполнении запроса. Но особое внимание стоит обратить на время ЦП, которое стало примерно в 10 раз меньше, чем при выполнении запроса без использования пакетного режима.

Следующий код чистит базу данных TSQL2012 и устанавливает параметры STATISTICS IO и TIME в значение OFF.

```

SET STATISTICS IO OFF;
SET STATISTICS TIME OFF;

```

```
DROP TABLE dbo.Fact;
DROP TABLE dbo.Dim1;
DROP TABLE dbo.Dim2;
DROP TABLE dbo.Dim3;
```

### КОНТРОЛЬНЫЙ ВОПРОС

- Как можно определить, использовал ли SQL Server режим пакетной обработки для конкретного итератора?

### Ответ на контрольный вопрос

- Можно проверить свойство итератора **Actual Execution Mode** (Действительный режим выполнения).

## ПРАКТИКУМ Работа с параметризацией запросов и хранимыми процедурами

В данном практикуме вы узнаете, как принудительно запустить перекомпиляцию хранимой процедуры, когда не требуется повторное использование плана.

### Задание 1. Работа с запросами, для которых SQL Server не использует повторно план выполнения

В этом задании вам нужно написать два запроса, для которых SQL Server не будет повторно использовать планы.

1. Если вы закрыли среду SSMS, запустите ее и подключитесь к вашему экземпляру SQL Server.
2. Откройте новое окно запроса, нажав кнопку **New Query** (Создать запрос).
3. Измените контекст на базу данных TSQL2012.
4. Выполните следующий запрос и проверьте план выполнения.

```
ELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE custid = 13
```

Вы должны увидеть в плане операторы **Index Seek** (Поиск в индексе) и **Key Lookup** (Уточняющий запрос ключа).

5. Измените параметр и выполните следующий запрос с включенным параметром **Actual Execution Plan** (Действительный план выполнения).

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE custid = 71;
```

На этот раз вы должны увидеть итератор **Clustered Index Scan** (Просмотр кластеризованного индекса).

## Задание 2. Изучение повторной компиляции хранимой процедуры

В этом задании вам нужно создать хранимую процедуру так, что SQL Server будет повторно использовать план этой процедуры. Но вы увидите, что план не должен быть повторно использован и поэтому укажете параметр WITH RECOMPILE для вынужденной повторной компиляции процедуры.

1. Создайте параметризованную процедуру из запроса, использовавшегося в предыдущем задании.

```
CREATE PROCEDURE Sales.GetCustomerOrders (@custid INT)
AS
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE custid = @custid;
```

2. Очистите кэш с помощью команды DBCC FREEPROCCACHE. (Убедитесь, что не делаете это на рабочем сервере.)
3. Выполните процедуру дважды в том же пакете, один раз для клиента 13 и один — для клиента 71.

```
EXEC Sales.GetCustomerOrders @custid = 13;
EXEC Sales.GetCustomerOrders @custid = 71;
```

Вы должны получить одинаковый план выполнения, план с операторами **Index Seek** (Поиск в индексе) и **Key Lookup** (Уточняющий запрос ключа), для обоих запросов. Этот план создается первым вызовом процедуры, все последующие вызовы используют тот же план.

4. Измените процедуру, чтобы инициировать повторную компиляцию для каждого вызова процедуры.

```
ALTER PROCEDURE Sales.GetCustomerOrders (@custid INT)
WITH RECOMPILE
AS
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE custid = @custid;
```

5. Обратите внимание, поскольку вы изменили процедуру, вам не нужно чистить кэш. SQL Server не использует кэшированный план процедуры, которую вы только что изменили. Выполните процедуру дважды в том же пакете, один раз для клиента 13 и один — для клиента 71.

```
EXEC Sales.GetCustomerOrders @custid = 13;
EXEC Sales.GetCustomerOrders @custid = 71;
```

На этот раз вы должны получить разные планы для каждого вызова процедуры.

6. Очистите базу данных TSQL2012.

```
DROP PROCEDURE Sales.GetCustomerOrders;
```

## Резюме занятия

- SQL Server выполняет параметризацию запросов для улучшения повторного использования планов выполнения.
- Вы можете выполнить параметризацию запросов самостоятельно.
- Режим пакетной обработки, который появился только в SQL Server 2012, может значительно повысить производительность запросов к хранилищам данных, особенно снижением использования ЦП.
- Пакетная обработка хорошо совмещается с индексами columnstore.

## Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы" в конце книги*.

1. Укажите возможные причины, по которым SQL Server не использует повторно существующий кэшированный план? (Выберите все подходящие варианты.)
  - A. Потому что параметр SET, который влияет на результат запроса, был изменен.
  - B. Потому что запрос был параметризован вручную в хранимой процедуре, и не используется параметр повторной компиляции.
  - C. Потому что SQL Server не может определить выборочность параметризованного предиката.
  - D. Потому что в параметризованном предикате для параметра использовался другой тип данных.
2. Каково главное преимущество пакетного режима обработки?
  - A. Он снижает дисковый ввод-вывод.
  - B. Он ускоряет передачу по сети.
  - C. Он снижает использование ЦП.
  - D. Он использует меньше памяти.
3. Какая из следующих команд SET препятствует повторному использованию плана выполнения?
  - A. SET STATISTICS IO ON.
  - B. SET STATISTICS PROFILE ON.
  - C. SET CONCAT\_NULL\_YIELDS\_NULL OFF.
  - D. SET STATISTICS IO OFF.

## Занятие 3. Использование подсказок оптимизатора и структур планов



Оптимизатор запросов SQL Server не всегда может найти лучший план выполнения из всех возможных. В некоторых случаях можно выбрать лучший план, используя подсказки оптимизатора. Чтобы эти подсказки использовать, надо изменить запрос. Кроме того, имеется еще одна возможность влиять на выполнение запроса — это *структуры планов*. Можно использовать структуры планов, когда вы не хотите или не можете изменить текст запроса, например, когда вам нужно оптимизировать запросы, созданные приложением от стороннего поставщика. SQL Server использует структуры планов, чтобы присоединить к запросу указания запроса или постоянный план запроса.

**Изучив материал этого занятия, вы сможете:**

- ✓ Объяснять и использовать подсказки оптимизатора
- ✓ Объяснять и использовать структуры планов

**Продолжительность занятия — 30 минут.**

### Подсказки оптимизатора



Подсказки оптимизатора имеют немного неудачное название. Это не просто подсказки, в действительности, это указания по выполнению запроса. Их можно использовать с инструкцией `SELECT` и с инструкциями языка изменения данных. Существует три вида подсказок: *табличные подсказки*, *указания запросов* и *указания соединений*.

**ВАЖНО!**

**Используйте подсказки оптимизатора с осторожностью**

При использовании подсказки вы изменяете запрос. SQL Server должен выполнять запрос или его часть всегда одинаково. Запрос может быть частью приложения, поэтому его бывает сложно изменить. Распределение данных может со временем меняться и, хотя подсказка могла иметь более высокую производительность в прошлом, ее производительность могла со временем понизиться. Лучше использовать другие средства, такие как создание соответствующих индексов, создание и обновление статистики и даже использование структур планов, прежде чем перейти к подсказкам. Применяйте подсказки как последнюю возможность и после их использования, время от времени проверяйте, действительно ли они все еще полезны.

Указания запросов задаются как часть предложения `OPTION` инструкций `SELECT`, `INSERT`, `UPDATE`, `DELETE` и `MERGE`. Использовать указания запросов во вложенных запросах нельзя, можно только в самом внешнем запросе. Если в операции `UNION` существует несколько запросов, предложение `OPTION` можно указать только после последнего запроса. Нельзя задавать указания запросов в инструкции `INSERT` за исключением случаев, когда предложение `SELECT` используется внутри инструкции.

В SQL Server 2012 поддерживаются следующие указания запросов:

- { HASH | ORDER } GROUP;
- { CONCAT | HASH | MERGE } UNION;

- { LOOP | MERGE | HASH } JOIN;
- EXPAND VIEWS;
- FAST *number\_rows*;
- FORCE ORDER;
- IGNORE\_NONCLUSTERED\_COLUMNSTORE\_INDEX;
- KEEP PLAN;
- KEEPFIXED PLAN;
- MAXDOP *number\_of\_processors*;
- MAXRECURSION *number*;
- OPTIMIZE FOR (@variable\_name {UNKNOWN | = *literal\_constant*} [, ...n ]);
- OPTIMIZE FOR UNKNOWN;
- PARAMETERIZATION { SIMPLE | FORCED };
- RECOMPILE;
- ROBUST PLAN;
- USE PLAN N'xml\_plan';
- TABLE HINT(*exposed\_object\_name* [, <table\_hint> [ [, ]...n ]]).

### ***К СВЕДЕНИЮ Подробно об указаниях запросов***

---

Подробное рассмотрение каждой из подсказок выходит за рамки данной книги. Но вы познакомитесь с некоторыми из них и их использованием на примерах кода и с помощью практикума к данному занятию. Подробные сведения о каждом из указаний запросов см. в электронной документации к SQL Server в разделе "Указания запросов (Transact-SQL)" по адресу <http://msdn.microsoft.com/ru-ru/library/ms181714.aspx>.

Следующие два запроса возвращают один и тот же статистически обработанный набор строк; первый запрос предоставляет возможность SQL Server принять решение о том, какую технологию статистической обработки применить — SQL Server решает использовать статистическую обработку хэша, — тогда как второй запрос принудительно применяет статистическую обработку потока.

```
-- Статистика по хэш-совпадениям
SELECT qty, COUNT(*) AS num
FROM Sales.OrderDetails
GROUP BY qty;
-- Принудительная статистическая обработка потока
SELECT qty, COUNT(*) AS num
FROM Sales.OrderDetails
GROUP BY qty OPTION (ORDER GROUP);
```

На рис. 17.21 показан план выполнения обоих запросов вместе, потому что они были выполнены в пакете.

Во втором запросе SQL Server использовал оператор **Stream Aggregate** (Статистическое выражение потока). Но поскольку этому оператору нужны упорядоченные

входные данные, SQL Server также добавил в план оператор Sort (Сортировать). Хотя статистическая обработка потока может выполняться быстрее, чем статистическая обработка хэша, второй запрос может быть более медленным из-за использования дополнительного оператора сортировки.

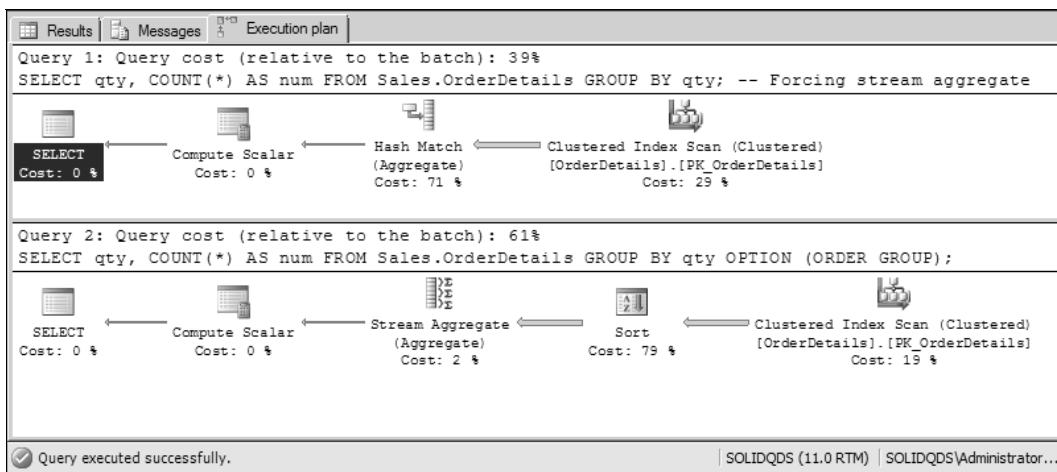


Рис. 17.21. План статистической обработки хэша и принудительной статистической обработки потока

Вы можете дать SQL Server подсказку для одной таблицы в запросе. Табличные подсказки влияют на блокировки и метод доступа только для одной таблицы или представления. Можно использовать табличные подсказки в предложении `FROM` и представлять их с помощью ключевого слова `WITH`. SQL Server поддерживает следующие табличные подсказки:

- NOEXPAND;
- INDEX (`index_value [ ,...n ]`) | INDEX = (`index_value`);
- FORCESEEK [`(index_value (index_column_name [ ,... ]))`];
- FORCESCAN;
- FORCESEEK;
- KEEPIDENTITY;
- KEEPDFAULTS;
- IGNORE\_CONSTRAINTS;
- IGNORE\_TRIGGERS;
- HOLDLOCK;
- NOLOCK;
- NOWAIT;
- PAGLOCK;

- READCOMMITTED;
- READCOMMITTEDLOCK;
- READPAST;
- READUNCOMMITTED;
- REPEATABLEREAD;
- ROWLOCK;
- SERIALIZABLE;
- SPATIAL\_WINDOW\_MAX\_CELLS = *integer*;
- TABLOCK;
- TABLOCKX;
- UPDLOCK;
- XLOCK.

### ***К СВЕДЕНИЮ Подробно о табличных подсказках***

---

Подробные сведения о каждой из табличных подсказок можно найти в электронной документации к SQL Server в разделе "Табличные указания (Transact-SQL)" по адресу <http://msdn.microsoft.com/ru-ru/library/ms187373.aspx>.

Возможно, наиболее популярной подсказкой оптимизатора является табличная подсказка, которая вызывает принудительное использование определенного индекса. Следующие два запроса демонстрируют пример предоставления SQL Server возможности выбора метода доступа и принудительного использования некластеризованного индекса.

```
-- Просмотр по кластеризованному индексу
SELECT orderid, productid, qty
FROM Sales.OrderDetails
WHERE productid BETWEEN 10 AND 30
ORDER BY productid;

-- Принудительное использование некластеризованного индекса
SELECT orderid, productid, qty
FROM Sales.OrderDetails WITH (INDEX(idx_nc_productid))
WHERE productid BETWEEN 10 AND 30
ORDER BY productid;
```

На рис. 17.22 представлен план выполнения для этого пакета.

SQL Server 2012 также поддерживает следующие указания соединений в предложении `FROM`:

- LOOP;
- HASH;
- MERGE;
- REMOTE.

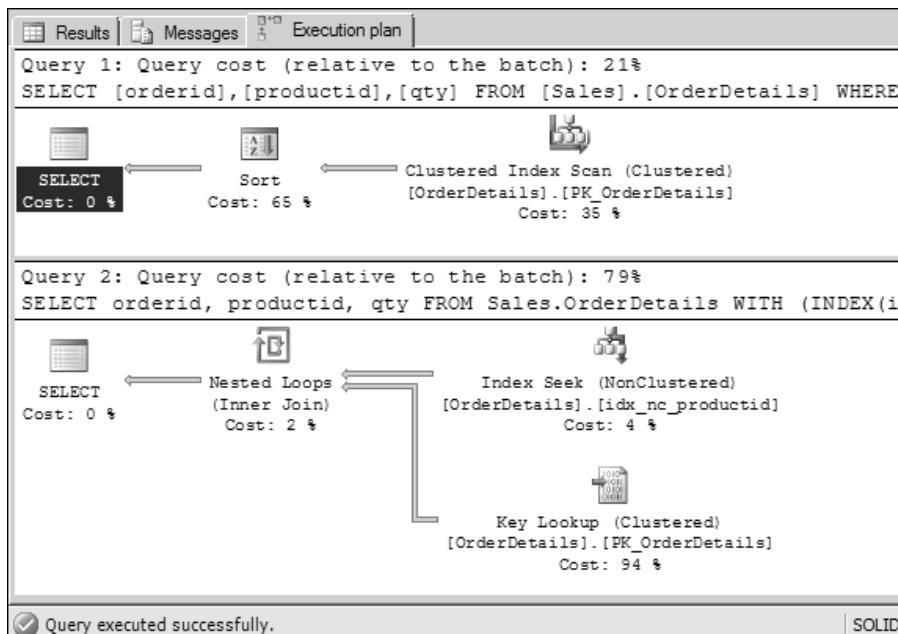


Рис. 17.22. План для просмотра и вынужденного поиска в некластеризованном индексе

### К СВЕДЕНИЮ Подробно об указаниях соединений

Подробные сведения о каждом из указаний соединений электронной документации к SQL Server см. в разделе "Указания в соединении (Transact-SQL)" по адресу <http://msdn.microsoft.com/ru-ru/library/ms173815.aspx>.

Следующие два запроса снова возвращают тот же результатирующий набор. В первом запросе выбор алгоритма соединения оставлен SQL Server — SQL Server принимает решение использовать соединение с применением вложенных циклов, а второй запрос инициирует соединение слиянием.

```
-- Соединение с использованием вложенных циклов
SELECT O.custid, O.orderdate, OD.orderid, OD.productid, OD.qty
FROM Sales.Orders AS O
INNER JOIN Sales.OrderDetails AS OD
 ON O.orderid = OD.orderid
WHERE O.orderid < 10250;
-- Вынужденное соединение слиянием
SELECT O.custid, O.orderdate, OD.orderid, OD.productid, OD.qty
FROM Sales.Orders AS O
INNER MERGE JOIN Sales.OrderDetails AS OD
 ON O.orderid = OD.orderid
WHERE O.orderid < 10250;
```

На рис. 17.23 представлен план выполнения для этого пакета.

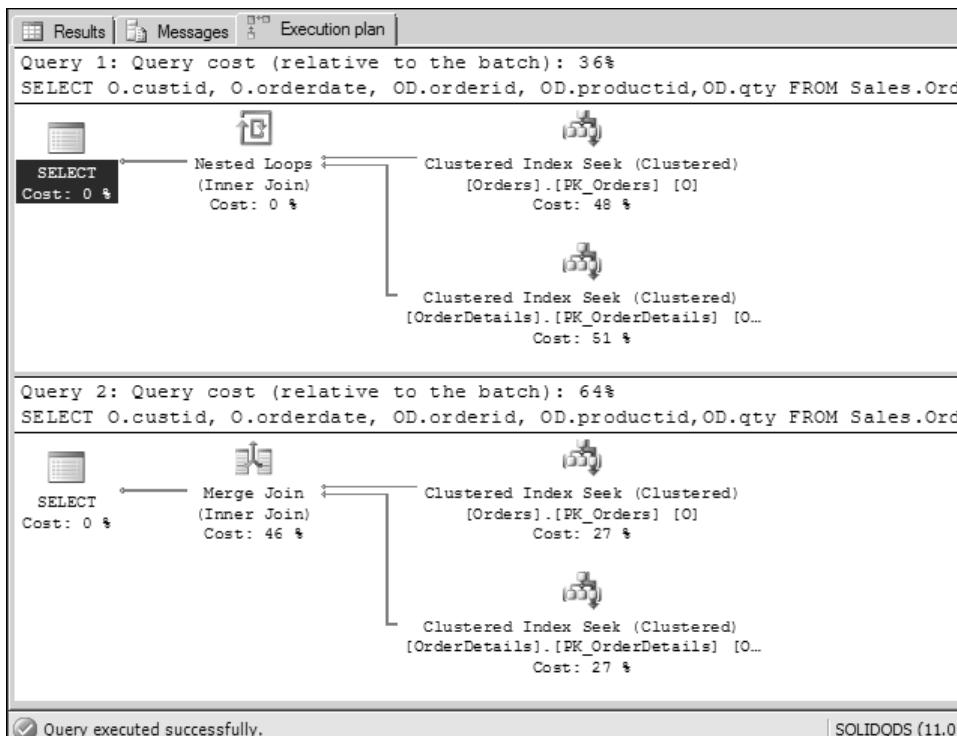


Рис. 17.23. План для вложенных циклов и вынужденное соединение слиянием

## Структуры планов

В структуре плана можно указать либо предложение **OPTION**, либо план запроса для инструкции, которую вы хотите оптимизировать. Также нужно указать инструкцию T-SQL, для которой предназначена структура плана. Оптимизатор запросов SQL Server сопоставляет существующую инструкцию T-SQL с инструкцией, указанной в структуре плана, и затем использует данную структуру для создания плана выполнения. Обратите внимание, вы не можете использовать структуры планов в редакции SQL Server 2012 Express.

Можно создать следующие типы структур планов.

- Структуры планов **OBJECT** используются оптимизатором запросов для запросов внутри хранимых процедур, скалярных определяемых пользователем функций, многооператорных возвращающих табличное значение определяемых пользователем функций и триггеров DML.
- Структуры планов **SQL** используются оптимизатором запросов для изолированных запросов или запросов, принадлежащих пакетам, не входящим ни в один объект базы данных.
- Структуры планов **TEMPLATE** используются оптимизатором запросов для изолированных запросов, которые могут быть параметризованы в указанную форму. Параметризацию можно инициировать с помощью структур шаблонов.

Структуры планов создаются с помощью системной процедуры sys.sp\_create\_plan\_guide. Посредством системной процедуры sys.sp\_control\_plan\_guide можно запретить, разрешить или удалить структуру плана. Можно создать структуру плана из кэшированного плана запроса с помощью системной процедуры sys.sp\_create\_plan\_guide\_from\_handle. Вы можете проверить допустимость плана посредством системной функции sys.fn\_validate\_plan\_guide. Структура плана может стать недопустимой из-за изменения схемы базы данных. Вы можете использовать системную процедуру sys.sp\_get\_query\_template для получения параметризованной формы запроса. Эта процедура особенно полезна для получения параметризованного запроса для структуры плана TEMPLATE.

Рассмотрите следующую хранимую процедуру:

```
CREATE PROCEDURE Sales.GetCustomerOrders (@custid INT)
AS
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE custid = @custid;
```

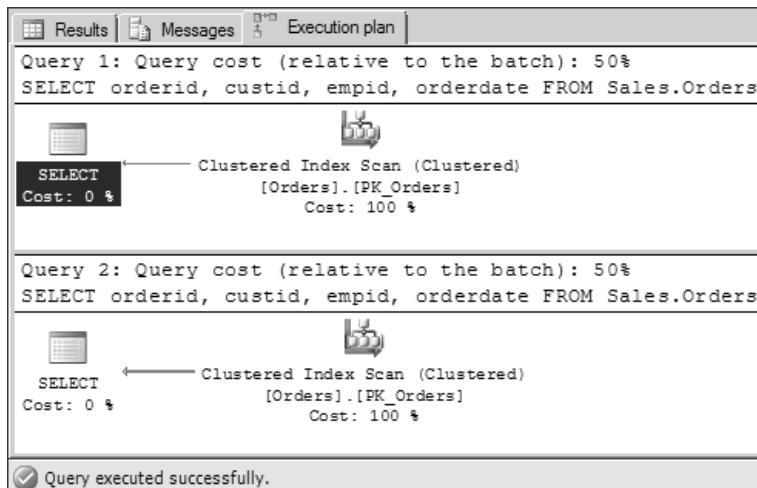
Для подавляющего большинства клиентов, например для клиента с `custid`, равным 71, запрос в процедуре является не очень выборочным; поэтому более подходящим будет использование просмотра таблицы или кластеризованного индекса. Но для некоторых редких клиентов, у которых есть всего по несколько заказов, например для клиента с `custid`, равным 13, больше подойдет поиск в индексе с уточняющим запросом. Если пользователь выполняет эту процедуру сначала для пользователя с `custid`, равным 13, тогда план процедуры в кэше не подойдет для большинства будущих выполнений. Создав структуру плана, которая использует указание запроса, инициирующее оптимизацию запроса в процедуре для клиента с `custid`, равным 71, вы оптимизируете выполнение хранимой процедуры для большинства клиентов. Следующий код создает структуру плана:

```
EXEC sys.sp_create_plan_guide
 @name = N'Cust71',
 @stmt = N'
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE custid = @custid;',
 @type = N'OBJECT',
 @module_or_batch = N'Sales.GetCustomerOrders',
 @params = NULL,
 @hints = N'OPTION (OPTIMIZE FOR (@custid = 71))';
```

Если вы выполните процедуру, используя разные параметры, после очистки кэша, чтобы гарантировать отсутствие прежнего плана для этой процедуры в кэше, SQL Server всегда оптимизирует запрос для значения `custid`, равного 71, и, таким образом, будет использовать просмотр кластеризованного индекса. Это справедливо, даже если вы сначала выполните запрос со значением `custid`, равным 13, как показано в следующем коде:

```
-- Очистка кэша
DBCC FREEPROCCACHE;
-- Выполнение процедуры с разными параметрами
EXEC Sales.GetCustomerOrders @custid = 13;
EXEC Sales.GetCustomerOrders @custid = 71;
```

На рис. 17.24 показан план выполнения для этого пакета. Видно, что в обоих выполнениях использовался итератор **Clustered Index Scan** (Просмотр кластеризованного индекса).



**Рис. 17.24.** План для двух выполнений хранимой процедуры с использованием структуры плана

Список всех структур планов в базе данных можно получить, запросив представление каталога sys.plan\_guides. Также можно получить список всех подсказок, используемых в каждой структуре плана, как показано в следующем запросе:

```
SELECT plan_guide_id, name, scope_type_desc, is_disabled,
 query_text, hints
 FROM sys.plan_guides;
```

Следующий код чистит базу данных TSQL2012:

```
EXEC sys.sp_control_plan_guide N'DROP', N'Cust71';
DROP PROCEDURE Sales.GetCustomerOrders;
```

### КОНТРОЛЬНЫЙ ВОПРОС

- Почему может быть предпочтительным использование структуры планов вместо подсказок оптимизатора?

### Ответ на контрольный вопрос

- Используя структуры планов, вам не нужно менять текст запроса.

## ПРАКТИКУМ Использование подсказок оптимизатора

В данном практикуме вы рассмотрите пример, в котором подсказка оптимизатора может быть полезной.

### Задание 1. Создание процедуры с указанием подсказки запроса RECOMPILE

В этом задании вам нужно создать процедуру с запросом, использующим указание запроса RECOMPILE.

- Если вы закрыли среду SSMS, запустите ее и подключитесь к вашему экземпляру SQL Server.
- Откройте новое окно запроса, нажав кнопку **New Query** (Создать запрос).
- Измените контекст на базу данных TSQL2012.
- Создайте процедуру для получения всех заказов для одного клиента, подобную той, которую вы создавали в практикуме к предыдущему занятию. Но на этот раз используйте предложение **OPTION**, чтобы инициировать повторную компиляцию одной инструкции в процедуре. Обратите внимание, процедура может включать несколько инструкций, и повторная компиляция всей процедуры может быть не лучшим решением. Использование структуры плана тоже может быть не самым удачным решением, поскольку план из структуры плана может оказаться не самым эффективным для всех возможных значений параметра запроса. Указание запроса в данном случае может быть наиболее подходящим решением. Используйте следующий код для создания процедуры:

```
CREATE PROCEDURE Sales.GetCustomerOrders (@custid INT)
AS
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE custid = @custid
OPTION (RECOMPILE);
```

### Задание 2. Тестирование процедуры с указанием запроса RECOMPILE

В этом задании вам нужно протестировать процедуру с запросом, использующим указание запроса RECOMPILE.

- Выполните процедуру, которую вы создали в предыдущем задании, дважды, дав разные значения параметра `@custid`. Используйте значения 13 и 71.

```
EXEC Sales.GetCustomerOrders @custid = 13;
EXEC Sales.GetCustomerOrders @custid = 71;
```

Вы должны получить два разных плана выполнения (рис. 17.25).

- Очистите базу данных TSQL2012 с помощью следующего кода:

```
DROP PROCEDURE Sales.GetCustomerOrders;
```

- Выходите из SSMS.

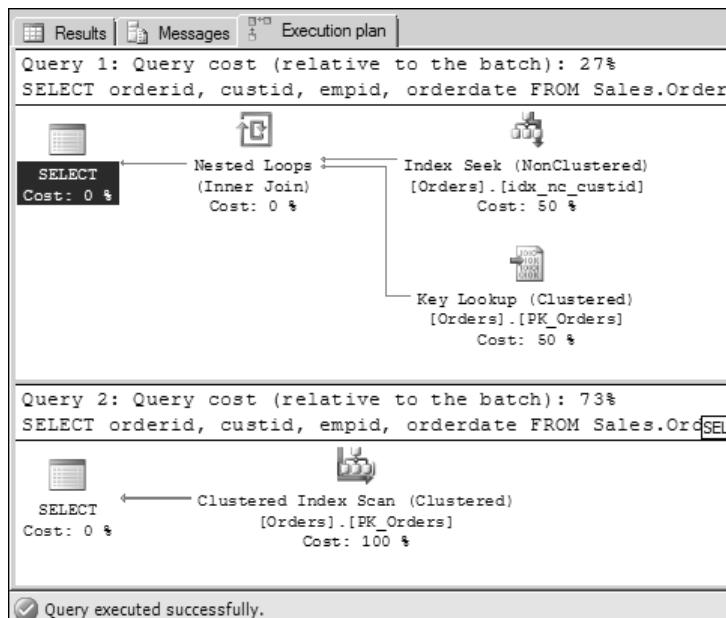


Рис. 17.25. Два разных плана выполнения для двух вызовов хранимой процедуры

## Резюме занятия

- Можно заставить SQL Server выполнять запрос конкретным способом с помощью подсказок оптимизатора.
- Используя структуры планов, можно инициировать определенный план выполнения без изменения текста запроса.

## Закрепление материала

Для проверки знания материала, изложенного в данном занятии, ответьте на следующие вопросы. Ответы на эти вопросы и пояснения, почему тот или иной ответ верен или неверен, можно найти в *приложении "Ответы"* в конце книги.

1. Какие типы подсказок оптимизатора поддерживает SQL Server 2012? (Выберите все подходящие варианты.)  
 А. Запроса.  
 Б. Соединения.  
 С. Очереди.  
 Д. Таблицы.
2. Что не является типом структуры плана?  
 А. JOIN.  
 Б. TEMPLATE.

- C. SQL.
  - D. ОБЪЕКТ.
3. Что инициирует указание запроса OPTION (ORDER GROUP)?
- A. Использование индексов для соединений.
  - B. Статистическая обработка хэша.
  - C. Статистическая обработка в очереди, т. е. последовательный план статистической обработки.
  - D. Статистическая обработка потока.

## Упражнения

---

В следующих упражнениях вы примените полученные знания основ дополнительных аспектов оптимизации. Ответы на вопросы можно найти в *приложении "Ответы" в конце книги*.

### Упражнение 1. Оптимизация запроса

Администраторы базы данных в компании, где вы занимаетесь поддержкой SQL Server, жалуются, что SQL Server не использует индексы для некоторых запросов от стороннего приложения. Запросы сгенерированы непосредственно в приложении, поэтому вы не можете их изменить.

1. Какие действия вы можете предпринять?
2. Можно ли оптимизировать запросы, которые вы не можете модифицировать?

### Упражнение 2. Табличная подсказка

В хранимой процедуре есть запрос, для которого, как вам кажется, SQL Server не использует оптимальный план. Хотя очень избирательное предложение WHERE запроса поддерживается некластеризованным индексом, SQL Server не использует его. Статистика для всех индексов в актуальном состоянии. Вам нужно оптимизировать этот запрос.

1. Можете ли вы модифицировать запрос, чтобы использовать индекс?
2. Как вы выполните эту задачу?

### Рекомендуемые упражнения

Для успешного освоения экзаменационных тем, представленных в данной главе, выполните следующие задания.

## Анализ планов выполнения запросов и их принудительное выполнение

Понимание планов выполнения является непростой задачей. Нахождение лучшего плана, чем использование оптимизатора запросов SQL Server — еще более сложная задача. Чтобы лучше разобраться в планах выполнения и структурах планов, вам нужно выполнить следующие задания.

- **Задание 1.** Чтобы понять, как SQL Server выполняет запросы, используйте базу данных TSQL2012 и напишите дополнительные запросы, которые объединяют, фильтруют, статистически обрабатывают и сортируют данные. Постарайтесь предсказать, какие итераторы будет использовать SQL. Проверьте правильность ваших прогнозов, выполнив запросы и проанализировав действительные планы выполнения. Постарайтесь понять, почему SQL Server принял решение использовать конкретные планы.
- **Задание 2.** Подумайте о рабочей системе, которую вы знаете. Использует ли эта система хранимые процедуры? Какие вы имеете возможности для оптимизации? Встроены ли запросы в приложение? Какие у вас есть возможности оптимизировать их? Как бы вы использовали структуры планов в каждом случае?

# Ответы

## Глава 1

---

### Занятие 1

#### Закрепление материала

1. Правильные ответы: В и D.
  - A. **Неправильно:** следует использовать стандартный код.
  - B. **Правильно:** использование стандартного кода упрощает его перенос между платформами, поскольку требует меньшего количества исправлений.
  - C. **Неправильно:** нет гарантии, что стандартный код будет более эффективным.
  - D. **Правильно:** при использовании стандартного кода проще адаптироваться к новой среде, потому что элементы стандартного кода похожи на разных платформах.
2. Правильный ответ: D.
  - A. **Неправильно:** отношение имеет заголовок с набором атрибутов, и кортежи отношения имеют тот же заголовок. Набор неупорядочен, поэтому порядковые номера не имеют смысла и вступают в противоречие с реляционной моделью. Следует обращаться к атрибутам по их именам.
  - B. **Неправильно:** считается, что запрос должен вернуть отношение. Отношение имеет тело с набором кортежей. Набор не содержит дубликатов. Возвращение дублированных строк является нарушением реляционной модели.
  - C. **Неправильно:** отсутствие ключа в таблице делает возможным наличие дублированных строк в этой таблице, и так же как в предыдущем случае, это противоречит реляционной модели.
  - D. **Правильно:** поскольку ожидается, что атрибуты будут идентифицироваться их именами, гарантия наличия имен у всех атрибутов является реляционной, поэтому несоответствия реляционной модели нет.

3. Правильный ответ: В.

- A. **Неправильно:** язык T-SQL не является стандартным, а SQL — это не диалект в Microsoft SQL Server.
- B. **Правильно:** язык SQL является стандартным, а T-SQL — диалект в Microsoft SQL Server.
- C. **Неправильно:** T-SQL не является стандартным языком.
- D. **Неправильно:** SQL это не диалект в Microsoft SQL Server.

## Занятие 2

### Закрепление материала

1. Правильный ответ: В.

- A. **Неправильно:** логическая обработка запроса не начинается с предложения SELECT.
- B. **Правильно:** логическая обработка запроса начинается с предложения FROM и затем переходит к WHERE, GROUP BY, HAVING, SELECT и ORDER BY.
- C. **Неправильно:** предложение ORDER BY не оценивается перед предложением SELECT.
- D. **Неправильно:** логическая обработка запроса не начинается с предложения SELECT.

2. Правильные ответы: С и D.

- A. **Неправильно:** T-SQL позволяет обращаться к атрибуту, сгруппированному в предложении WHERE.
- B. **Неправильно:** T-SQL позволяет выполнять группировку по выражению.
- C. **Правильно:** если в запрос есть группирование, на этапах, обрабатываемых после этапа GROUP BY, каждый атрибут, к которому вы обращаетесь, должен присутствовать либо в списке GROUP BY, либо в агрегатной функции.
- D. **Правильно:** поскольку предложение HAVING оценивается раньше предложения SELECT, ссылка на псевдоним, определенный в предложении SELECT внутри предложения HAVING, является неправильной.

3. Правильный ответ: А.

- A. **Правильно:** запрос с предложением ORDER BY не возвращает реляционный результат. Чтобы результат был реляционным, запрос должен удовлетворять ряду требований, включая следующие: запрос не должен содержать предложение ORDER BY, все атрибуты должны иметь имена, все имена атрибутов должны быть уникальными, и дубликаты не должны присутствовать в результате.
- B. **Неправильно:** запрос, имеющий предложение DISTINCT в предложении SELECT, может возвратить дубликаты.

- C. **Неправильно:** запрос, не имеющий предложения ORDER BY, не может гарантировать последовательность строк на выходе.
- D. **Неправильно:** запрос, не имеющий предложения ORDER BY, не может гарантировать последовательность строк на выходе.

## Упражнения

### Упражнение 1. Важность знания теории

1. Одна из самых распространенных ошибок, которую делают разработчики T-SQL, заключается во мнении, что запрос без предложения ORDER BY всегда возвращает данные в определенном порядке, например, как у кластеризованного индекса. Но как понятно из теории множеств, набор не имеет определенного порядка своих элементов, и вы знаете, что не следует делать таких заключений. В SQL единственным способом гарантировать, что строки будут возвращены в определенном порядке, является добавление предложения ORDER BY. Это один из множества примеров аспектов T-SQL, которые можно лучше понять, если вы разбираетесь в основах этого языка.
2. Хотя язык T-SQL основывается на реляционной модели, он во многом отклоняется от нее. Но он дает вам достаточно инструментов, которые, при условии понимания реляционной модели, позволят вам писать в реляционной манере. Следование реляционной модели поможет писать код более правильно. Далее приведено несколько примеров:
  - не следует полагаться на последовательность столбцов или строк;
  - всегда нужно давать имена результирующим столбцам;
  - следует устранять дубликаты, если возможно их появление в результате запроса.

### Упражнение 2. Собеседование на должность специалиста по анализу кода

1. Важно использовать стандартный код SQL. В таком случае и сам код, и ваши знания станут более переносимыми. Особенно в случаях, когда используются как стандартные, так и нестандартные формы элементов языка, рекомендуется использовать стандартные формы.
2. Использование порядковых номеров в предложении ORDER BY не рекомендуется. С точки зрения реляционности, предполагается обращение к атрибутам по имени, а не по порядковому номеру. Кроме того, что если список SELECT будет изменен в будущем и разработчик забудет откорректировать список ORDER BY соответственно?
3. Если запрос не имеет предложения ORDER BY, не существует гарантии какой-либо определенной последовательности в результате запроса. Порядок вывода следует считать произвольным. Также надо отметить, что интервьюер использовал

неправильный термин "запись" вместо "строки". Возможно, следует упомянуть об этом, поскольку интервьюер мог сделать это, чтобы проверить ваши знания.

4. С чисто реляционной точки зрения, это может быть допустимо и, возможно, даже рекомендовано. Но с практической точки зрения, существует вероятность, что SQL Server попытается удалить дубликаты, даже если их нет, что приведет к лишним затратам. Поэтому рекомендуется добавлять предложение DISTINCT только тогда, когда дубликаты возможны в результирующем наборе, но они не должны быть возвращены.

## Глава 2

---

### Занятие 1

#### Закрепление материала

1. Правильные ответы: В и D.
  - A. **Неправильно:** создание псевдонимов атрибутов позволяет удовлетворять требования реляционности, поэтому это явно более чем просто эстетическое свойство.
  - B. **Правильно:** Реляционная модель требует, чтобы все атрибуты имели имена.
  - C. **Неправильно:** язык T-SQL позволяет результирующему атрибуту не иметь имени, если выражение основывается на вычислении без псевдонима.
  - D. **Правильно:** с помощью псевдонима можно назначить результирующему атрибуту имя по своему выбору.
2. Правильный ответ: С.
  - A. **Неправильно:** предложения FROM и SELECT являются обязательными в запросе SELECT по стандарту SQL, но не T-SQL.
  - B. **Неправильно:** предложение WHERE необязательное в T-SQL.
  - C. **Правильно:** в соответствии с T-SQL только предложение SELECT является обязательным.
  - D. **Неправильно:** оба предложения, и FROM, и WHERE, необязательные в T-SQL.
3. Правильные ответы: С и D.
  - A. **Неправильно:** присвоение псевдонимов столбцам с помощью предложения AS является стандартной процедурой и рекомендуется к применению.
  - B. **Неправильно:** присвоение псевдонимов таблицам с помощью предложения AS является стандартной процедурой и рекомендуется к применению.
  - C. **Правильно:** неиспользование псевдонима для столбца, который является результатом вычисления, не является реляционным и не рекомендуется к применению.
  - D. **Правильно:** использование звездочки (\*) в списке SELECT считается плохой практикой.

## Занятие 2

### Закрепление материала

1. Правильный ответ: В.
  - A. **Неправильно:** как правило, форматирование не входит в область ответственности уровня типов данных или данных; за это отвечает уровень представления данных.
  - B. **Правильно:** тип данных должен рассматриваться как ограничение, поскольку он ограничивает разрешенные значения.
  - C. **Неправильно:** сам по себе тип данных не препятствует появлению дубликатов. Для предупреждения появления дубликатов необходимо использовать первичный ключ или ограничение уникальности.
  - D. **Неправильно:** тип данных не препятствует появлению значений NULL. Чтобы достичь этого, следует использовать ограничение NOT NULL.
2. Правильные ответы: А и В.
  - A. **Правильно:** функция NEWID создает индексы GUID в произвольном порядке. Ее стоит применять, когда дополнительный размер памяти не имеет особого значения, а приоритетной является возможность генерировать уникальное значение во времени и пространстве, из любого места и в произвольном порядке.
  - B. **Правильно:** функция NEWSEQUENTIALID генерирует идентификаторы GUID в машине в возрастающей последовательности. Она позволяет уменьшить фрагментацию и хорошо работает, когда данные загружаются в одной сессии и количество дисков невелико. Однако следует тщательно рассмотреть возможность использования другого генератора ключей, такого как объект последовательности, причем с меньшим типом данных, где это возможно.
  - C. **Неправильно:** нет никакой гарантии, что функция GETDATE сгенерирует уникальные значения; поэтому это не лучший вариант генерации ключей.
  - D. **Неправильно:** функция CURRENT\_TIMESTAMP — это просто стандартная версия функции GETDATE, так что она тоже не гарантирует уникальности.
3. Правильный ответ: В.
  - A. **Неправильно:** выражения CASE ничего не делают с моделью восстановления базы данных.
  - B. **Правильно:** разница между этими формами заключается в том, что простая форма сравнивает выражения, а поисковая форма использует предикаты.
  - C. **Неправильно:** оба выражения CASE разрешены везде, где разрешены скалярные выражения — в любом месте запроса.
  - D. **Неправильно:** оба выражения CASE разрешены везде, где разрешены скалярные выражения — в любом месте запроса.

## Упражнения

### Упражнение 1. Анализ использования типов данных

1. Тип данных DATETIME использует 8 байт памяти. SQL Server 2012 поддерживает тип данных DATE, который использует 3 байта памяти. Для всех атрибутов, представляющих только дату, рекомендуется перейти на использование типа данных DATE. Чем меньше требования к памяти, тем лучше будет выполняться чтение.

Что касается прочих рекомендаций, общее правило "чем меньше, тем лучше, при условии, что покрываются потребности атрибута в длительной перспективе" подходит с точки зрения производительности чтения данных. Например, если у вас есть описания переменных длин данных, хранящихся в типе данных CHAR или NCHAR, следует рассмотреть переход на тип данных VARCHAR или NVARCHAR соответственно. Также если в данный момент вы используете типы данных в кодировке Unicode, но вам нужно хранить строковые данные только для одного языка — скажем, US English, — рассмотрите использование стандартных символов.

2. Тип данных UNIQUEIDENTIFIER большой — 16 байт. И поскольку это еще и ключ кластеризованного индекса, он копируется во все некластеризованные индексы. Кроме того, из-за произвольной последовательности, в которой функция NEWID генерирует значения, скорее всего, уровень фрагментации индекса велик. Можно рассмотреть (и протестировать!) другой подход — перейти на целочисленный тип и, используя объект последовательности, генерировать ключи, которые не конфликтуют в таблицах. Из-за меньшего размера типа данных, с учетом эффекта умножения для кластеризованных индексов, производительность чтения, вероятно, станет выше. Значения будут расти, и в результате уменьшится фрагментация, что также положительно повлияет на чтение данных.

### Упражнение 2. Анализ использования функций

Для улучшения переносимости кода важно использовать стандартный код, когда это возможно, и конечно, это особенно относится к использованию встроенных функций. Например, используйте функцию COALESCE, а не ISNULL, функцию CURRENT\_TIMESTAMP, а не GETDATE, и функцию CASE, а не IIF.

---

## Глава 3

### Занятие 1

#### Закрепление материала

1. Правильный ответ: В.  
А. **Неправильно:** NULL не является частью трех возможных логических результатов предиката в T-SQL.

- B. **Правильно:** троичная логика оперирует значениями "истина", "ложь" и неизвестное значение.
- C. **Неправильно:** 1, 0 и NULL не являются частью трех возможных логических результатов предиката.
- D. **Неправильно:** -1, 0 и 1 не являются частью трех возможных логических результатов предиката.
2. Правильные ответы: A, B и C.
- A. **Правильно:** форма записи '2012-02-12' не зависит от языка для типов данных DATE, DATETIME2 и DATETIMEOFFSET, но зависит от языка для типов данных DATETIME и SMALLDATETIME.
- B. **Правильно:** форма записи '02/12/2012' зависит от языка.
- C. **Правильно:** форма записи '12/02/2012' зависит от языка.
- D. **Неправильно:** форма записи '20120212' не зависит от языка.
3. Правильные ответы: B и E.
- A. **Неправильно:** этот предикат применяет обработку к фильтруемому столбцу, следовательно, это не аргумент поиска.
- B. **Правильно:** предикат LIKE является аргументом поиска, когда шаблон начинается с известного префикса.
- C. **Неправильно:** предикат LIKE не является аргументом поиска, когда шаблон начинается с подстановочного знака.
- D. **Неправильно:** предикат LIKE не является аргументом поиска, когда шаблон начинается с подстановочного знака.
- E. **Правильно:** предикат является аргументом поиска, поскольку к фильтруемому столбцу не применяется никаких манипуляций.

## Занятие 2

### Закрепление материала

1. Правильный ответ: A.
- A. **Правильно:** без предложения ORDER BY сортировка не гарантирована и может быть произвольной — это зависит от оптимизации.
- B. **Неправильно:** без предложения ORDER BY сортировка не гарантирована.
- C. **Неправильно:** без предложения ORDER BY сортировка не гарантирована.
- D. **Неправильно:** без предложения ORDER BY сортировка не гарантирована.
2. Правильный ответ: C.
- A. **Неправильно:** здесь используется сортировка по возрастанию для orderdate и по убыванию для orderid.
- B. **Неправильно:** это неверный синтаксис.

- C. **Правильно:** правильный синтаксис — указать DESC после каждого выражения, для которого направление сортировки должно быть по убыванию.
- D. **Неправильно:** это неверный синтаксис.
3. Правильные ответы: B, C и D.
- A. **Неправильно:** это неверный синтаксис.
- B. **Правильно:** по умолчанию выполняется сортировка по возрастанию, поэтому данное предложение использует сортировку по возрастанию и для orderdate, и для orderid.
- C. **Правильно:** это предложение явно использует сортировку по возрастанию и для orderdate, и для orderid.
- D. **Правильно:** по умолчанию выполняется сортировка по возрастанию, поэтому данное предложение использует сортировку по возрастанию и для orderdate, и для orderid.

## Занятие 3

### Закрепление материала

1. Правильный ответ: B.
- A. **Неправильно:** если в результате запроса есть хотя бы три строки, не содержащие TOP, запрос возвратит три строки.
- B. **Правильно:** если в результате запроса меньше трех строк, не содержащих TOP, запрос возвратит эти строки. Если таких строк три или более, запрос возвратит три строки.
- C. **Неправильно:** если в результате запроса меньше трех строк, не содержащих TOP, запрос возвратит эти строки.
- D. **Неправильно:** если не будет использоваться опция WITH TIES, запрос не будет возвращать больше, чем требуемое число строк.
- E. **Неправильно:** если не будет использоваться опция WITH TIES, запрос не будет возвращать больше, чем требуемое число строк.
- F. **Неправильно:** если не будет использоваться опция WITH TIES, запрос не будет возвращать больше, чем требуемое число строк.
2. Правильный ответ: F.
- A. **Неправильно:** если в результате запроса есть хотя бы три строки, не содержащие TOP, запрос возвратит хотя бы три строки.
- B. **Неправильно:** если в результате запроса более трех строк, не содержащих TOP, и имеются связи с третьей строкой, запрос возвратит более трех строк.
- C. **Неправильно:** если в результате запроса меньше трех строк, не содержащих TOP, запрос возвратит только эти строки. Если в результате запроса более трех строк, не содержащих TOP, и имеются связи с третьей строкой, запрос возвратит более трех строк.

- D. **Неправильно:** если в результате запроса меньше трех строк, не содержащих `TOP`, запрос возвратит только эти строки.
- E. **Неправильно:** если в результате запроса три или меньше строк, не содержащих `TOP`, запрос не возвратит более трех строк.
- F. **Правильно:** если в результате запроса меньше трех строк, не содержащих `TOP`, запрос возвратит только эти строки. Если в результате запроса есть хотя бы три строки и не существует связей с третьей строкой, запрос возвратит три строки. Если в результате запроса более трех строк и имеются связи с третьей строкой, запрос возвратит более трех строк.
3. Правильные ответы: А и С.
- A. **Правильно:** T-SQL поддерживает использование предложения `OFFSET` без предложения `FETCH`.
- B. **Неправильно:** в отличие от стандартного SQL, язык T-SQL не поддерживает предложение `FETCH` без предложения `OFFSET`.
- C. **Правильно:** T-SQL поддерживает использование предложений `OFFSET` и `FETCH`.
- D. **Неправильно:** T-SQL не поддерживает `OFFSET...FETCH` без предложения `ORDER BY`.

## Упражнения

### Упражнение 1. Рекомендации по улучшению производительности фильтрации и сортировки

1. Прежде всего, в базе данных должно выполняться как можно больше фильтрации данных. Выполнение большей части фильтрации данных на клиенте означает, что вы сканируете больше данных, что увеличивает нагрузку на подсистему хранения данных, а также что вы увеличиваете нагрузку на сеть. При выполнении фильтрации в базе данных, например, с помощью предложения `WHERE`, следует использовать аргументы поиска, которые повышают вероятность эффективного применения индексов. Следует, насколько возможно, избегать манипулирования фильтруемыми столбцами.
2. Добавление предложения `ORDER BY` означает, что SQL Server должен гарантировать возвращение строк в запрашиваемой последовательности. Если нет существующих индексов для поддержки требований сортировки, SQL Server не будет иметь другой возможности, кроме как сортировать данные. На больших наборах сортировка является дорогой. Поэтому главная рекомендация — избегать добавления предложений `ORDER BY` в запросы, если нет требований сортировки данных. И когда действительно нужно возвращать строки в определенном порядке, следует рассмотреть возможность организации поддерживающих индексов, чтобы избавить SQL Server от необходимости выполнять дорогостоящие операции сортировки.

3. Главный способ помочь хорошей работе запросов, содержащих `TOP` и `OFFSET...FETCH`, — организация индексов для поддержки элементов сортировки. Это, в дополнение к сортировке, может предотвратить сканирование всех данных.

## Упражнение 2. Обучение разработчика-стажера

1. Чтобы понять, почему нельзя ссылаться на псевдоним, определенный в списке `SELECT` в предложении `WHERE`, надо понимать логическую обработку запросов. Несмотря на то, что последовательность ввода предложений — `SELECT...FROM...WHERE...GROUP BY...HAVING...ORDER BY`, последовательность логической обработки запросов `FROM...WHERE...GROUP BY...HAVING...SELECT...ORDER BY`. Как видите, предложение `WHERE` оценивается раньше предложения `SELECT`, и таким образом, псевдонимы, определенные в предложении `SELECT`, не видны предложению `WHERE`.
2. Последовательность логической обработки запросов объясняет, почему предложение `ORDER BY` может ссылаться на псевдонимы, определенные в предложении `SELECT`. Это происходит потому, что предложение `ORDER BY` логически оценивается после предложения `SELECT`.
3. Предложение `ORDER BY` является обязательным, если используется `OFFSET...FETCH`, поскольку это стандартное предложение, и стандартный SQL решил сделать его обязательным. Компания Microsoft просто следует стандарту. Что касается конструкции `TOP`, это частное свойство, и когда компания Microsoft разрабатывала его, ее сотрудники решили разрешить использование `TOP` в совершенно недетерминированной манере — без предложения `ORDER BY`. Обратите внимание, то, что `OFFSET...FETCH` требует наличия предложения `ORDER BY`, не означает, что вы должны использовать детерминированную сортировку. Например, если список `ORDER BY` не является уникальным, сортировка не будет детерминированной. И если нужно, чтобы сортировка полностью была недетерминированной, можно указать выражение `ORDER BY (SELECT NULL)`, и это эквивалентно тому, что предложение `ORDER BY` не указано совсем.

---

# Глава 4

## Занятие 1

### Закрепление материала

1. Правильный ответ: D.
  - A. **Неправильно:** оба предложения используют троичную логику.
  - B. **Неправильно:** оба предложения используют троичную логику.
  - C. **Неправильно:** `ON` определяет соответствие, а `WHERE` — фильтрацию.
  - D. **Правильно:** `ON` определяет соответствие, а `WHERE` — фильтрацию.

2. Правильные ответы: С и D.

- A. **Неправильно:** ключевое слово JOIN не может быть опущено в новом синтаксисе соединений.
- B. **Неправильно:** если ключевое слово CROSS опущено в CROSS JOIN, одно ключевое слово JOIN означает внутреннее соединение, а не пересекающееся соединение.
- C. **Правильно:** если ключевое слово INNER опущено в INNER JOIN, значение сохраняется.
- D. **Правильно:** если ключевое слово OUTER опущено в LEFT OUTER JOIN, RIGHT OUTER JOIN и FULL OUTER JOIN, значение сохраняется.

3. Правильный ответ: А.

- A. **Правильно:** синтаксис с ключевым словом JOIN согласуется только со стандартным синтаксисом, доступным для внешних соединений, и менее подвержен ошибкам.
- B. **Неправильно:** внешние соединения не имеют стандартного синтаксиса с использованием запятых.
- C. **Неправильно:** таких рекомендаций не существует. Пересекающиеся и внутренние соединения имеют право на существование.
- D. **Неправильно:** таких доказательств нет.

## Занятие 2

### Закрепление материала

1. Правильный ответ: А.

- A. **Правильно:** запрос завершается ошибкой при выполнении, указывая, что возвращено более одного значения.
- B. **Неправильно:** запрос завершается ошибкой.
- C. **Неправильно:** запрос завершается ошибкой.
- D. **Неправильно:** скалярный подзапрос конвертируется в значение NULL, когда он возвращает пустой набор — не несколько значений.

2. Правильные ответы: В и С.

- A. **Неправильно:** все типы табличных выражений с точки зрения оптимизации обрабатываются одинаково — у них нет вложенности.
- B. **Правильно:** если нужно сослаться на одну производную таблицу из другой, следует вложить их одна в другую. При использовании СТЕ вы их отделяете друг от друга запятыми, так что код становится модульным и проще для выполнения.
- C. **Правильно:** поскольку имя СТЕ определено раньше внешнего запроса, который его использует, внешний запрос имеет право ссылаться на несколько экземпляров одного и того же имени СТЕ.
- D. **Неправильно:** СТЕ видны только той инструкции, которая их определяет.

3. Правильный ответ: D.

- A. **Неправильно:** оба возвращают все комбинации.
- B. **Неправильно:** оба возвращают пустой набор.
- C. **Неправильно:** оба возвращают пустой набор.
- D. **Правильно:** оба возвращают одинаковый результат, когда не существует связи, поскольку оператор CROSS APPLY применяет все строки из T2 к каждой строке из T1.

## Занятие 3

### Закрепление материала

1. Правильные ответы: A, C и D.

- A. **Правильно:** оператор UNION удаляет дубликаты.
- B. **Неправильно:** оператор UNION ALL не удаляет дубликаты.
- C. **Правильно:** оператор INTERSECT удаляет дубликаты.
- D. **Правильно:** оператор EXCEPT удаляет дубликаты.

2. Правильный ответ: D.

- A. **Неправильно:** при использовании оператора UNION порядок входных запросов не имеет значения.
- B. **Неправильно:** при использовании оператора UNION ALL порядок входных запросов не имеет значения.
- C. **Неправильно:** при использовании оператора INTERSECT порядок входных запросов не имеет значения.
- D. **Правильно:** при использовании оператора EXCEPT порядок входных запросов имеет значение.

3. Правильный ответ: B.

- A. **Неправильно:** без круглых скобок оператор INTERSECT имеет более высокий приоритет над другими операторами; если используются круглые скобки, он оценивается последним.
- B. **Правильно:** без круглых скобок, оператор INTERSECT имеет более высокий приоритет над другими операторами; то же самое и при использовании круглых скобок.
- C. **Неправильно:** без круглых скобок оператор INTERSECT имеет более высокий приоритет над другими операторами; если используются круглые скобки, первым оценивается оператор EXCEPT.
- D. **Неправильно:** без круглых скобок оператор UNION оценивается вторым (после оператора INTERSECT), а с круглыми скобками оператор UNION оценивается последним.

## Упражнения

### Упражнение 1. Анализ кода

- Чтобы избавиться от сложности вложенных производных таблиц, в дополнение к дублированию кода производных таблиц, вы можете использовать СТЕ. СТЕ не имеют вложенности; напротив, они обладают большей модульностью. Вы также можете определить СТЕ один раз и ссылаться на него много раз во внешнем запросе. Что касается запросов, которые повторяются в разных частях в коде, для многократного использования кода вы можете применять представления и встроенные табличные функции. Если вам не нужно передавать параметры, используйте первый, если нужно — второй.
- Клиент должен рассмотреть использование оператора `APPLY` вместо курсора и запроса на каждую строку. Оператор `APPLY` требует меньше кода и поэтому улучшает сопровождение, а также он не влечет снижения производительности, которое, как правило, характерно для курсоров.
- Клиент должен проверить связи по внешнему ключу и рассмотреть создание индексов на столбцах внешнего ключа.

### Упражнение 2. Объяснение операторов работы с наборами

- Оператор `UNION` возвращает различающиеся строки. Когда объединенные наборы не пересекаются, нет дубликатов, которые надо удалять, но оптимизатор запросов SQL Server может не знать этого. Попытки удалять дубликаты, даже если их нет, приводят к дополнительным затратам. Поэтому когда наборы не пересекаются, важно использовать оператор `UNION ALL`, а не оператор `UNION`. Кроме того, добавление ограничений `CHECK`, которые определяют диапазоны, поддерживаемые каждой таблицей, может помочь оптимизатору понять, что наборы не пересекаются. Тогда даже при использовании оператора `UNION` оптимизатор может понять, что не нужно удалять дубликаты.
- Операторы работы с наборами имеют множество преимуществ. Они позволяют писать более простой код, поскольку не нужно явно сравнивать столбцы двух входных наборов, как при использовании соединений. Также когда операторы работы с наборами сравнивают два значения `NULL`, они считают их равными, что не имеет места в случае соединений. Когда требуется именно такое поведение, проще применять операторы работы с наборами. При использовании соединений для получения такого же поведения надо добавлять предикаты.

## Глава 5

### Занятие 1

#### Закрепление материала

- Правильный ответ: D.  
А. **Неправильно:** можно группировать строки без вызова статистической функции.

- B. **Неправильно:** запрос может включать статистическую функцию без предложения GROUP BY. Если группирование подразумевается — все строки составляют одну группу.
- C. **Неправильно:** не существует требования, чтобы элементы группирования появлялись в списке SELECT, хотя, как правило, возвращают элементы, которые группируются.
- D. **Правильно:** запрос группирования возвращает только одну строку на группу. Поэтому все выражения, появляющиеся на этапах, которые оцениваются после предложения GROUP BY (HAVING, SELECT и ORDER BY), должны гарантировать возвращение одного значения на группу. Отсюда и происходит это ограничение.
2. Правильные ответы: B, C и D.
- A. **Неправильно:** эти функции не могут быть использованы в предложении GROUP BY.
- B. **Правильно:** когда функции возвращают 1 бит, для замещения используется значение NULL; когда они возвращают 0 бит, значение NULL происходит из таблицы.
- C. **Правильно:** каждый набор группирования может быть указан уникальной комбинацией 1 и 0, возвращенной этими функциями.
- D. **Правильно:** эти функции можно использовать для сортировки, поскольку они возвращают 0 бит для детального элемента и 1 бит для агрегированного элемента. Поэтому если нужно сначала видеть детали, следует сортировать по результату функции в порядке возрастания.

3. Правильный ответ: A.

- A. **Правильно:** функция COUNT(\*) не обрабатывает входное выражение; она вычисляет число строк в группе. Функция COUNT(<expression>) обрабатывает выражение и игнорирует значения NULL. Интересно то, что функция COUNT(<expression>) возвращает 0, когда все входные значения — NULL, тогда как другие общие функции работы с наборами данных, такие как MIN, MAX, SUM, и AVG, возвращают в подобном случае значение NULL.
- B. **Неправильно:** функция COUNT(\*) подсчитывает строки.
- C. **Неправильно:** функция COUNT(\*) подсчитывает строки.
- D. **Неправильно:** очевидно, что эти функции отличаются в смысле обработки значений NULL.

## Занятие 2

### Закрепление материала

1. Правильный ответ: B.

- A. **Неправильно:** функция GROUPING имеет отношение к наборам группирования — не к сведению данных.

- B. **Правильно:** оператор PIVOT определяет группирующий элемент методом исключения, как остающийся после выделения распределяемого и агрегатного элементов.
- C. **Неправильно:** при использовании оператора PIVOT группирование для сведения данных происходит как часть оператора PIVOT — перед тем, как оценивать предложение GROUP BY.
- D. **Неправильно:** оператор PIVOT не рассматривает определения ограничений для определения элемента группирования.
2. Правильные ответы: A, B, C и D.
- A. **Правильно:** нельзя задавать вычисление как вход для статистической функции, а только лишь имя столбца из входной таблицы.
- B. **Правильно:** нельзя задавать вычисление как распределяемый элемент, а только лишь имя столбца из входной таблицы.
- C. **Правильно:** нельзя задавать подзапрос в предложении IN, а только лишь статический список.
- D. **Правильно:** нельзя указывать несколько статистических функций, а только одну.
3. Правильный ответ: D.
- A. **Неправильно:** тип данных столбца значений не всегда INT.
- B. **Неправильно:** тип данных столбца значений NVARCHAR(128) — это относится к столбцу имен.
- C. **Неправильно:** тип данных столбца значений — не SQL\_VARIANT.
- D. **Правильно:** тип данных столбца значений такой же, как тип данных столбцов, к которым применяется отмена сведения, поэтому они должны все иметь одинаковый тип данных.

## Занятие 3

### Закрепление материала

1. Правильные ответы: С и D.
- A. **Неправильно:** ограничение по умолчанию использует единицу измерения окна RANGE.
- B. **Неправильно:** ограничение по умолчанию использует единицу измерения окна RANGE.
- C. **Правильно:** это ограничение по умолчанию.
- D. **Правильно:** это сокращенная форма ограничения по умолчанию, имеющая тот же смысл.
2. Правильный ответ: B.
- A. **Неправильно:** эти определения на 1 меньше правильных.
- B. **Правильно:** это правильные определения.

- C. **Неправильно:** эти определения на 2 меньше правильных.
- D. **Неправильно:** правильным является противоположное утверждение — эти две функции возвращают одинаковый результат при уникальной сортировке.
3. Правильный ответ: A.
- A. **Правильно:** оконные функции должны обрабатывать результирующий набор базового запроса. С точки зрения логической обработки запросов, этот результирующий набор получается на этапе SELECT.
- B. **Неправильно:** стандартный язык SQL определяет это ограничение, так что с недостатком времени компании Microsoft это никак не связано.
- C. **Неправильно:** существуют практические основания для фильтрации или группировки данных с помощью оконных функций.
- D. **Неправильно:** в SQL нет программ несанкционированного доступа (door-backdoor functions).

## Упражнения

### Упражнение 1. Усовершенствование операций анализа данных

- Статистические оконные функции прекрасно подходят для таких вычислений. Что касается вещей, которых следует остерегаться, в текущей реализации SQL Server 2012 следует избегать использования единицы измерения окна RANGE. А также помнить, что без явного предложения оконного кадра вы будете использовать вариант RANGE по умолчанию, поэтому следует явно указывать параметр ROWS.
- Операторы PIVOT и UNPIVOT подходят для запросов к перекрестным таблицам. При использовании оператора PIVOT следует помнить, что группирующий элемент определяется методом исключения — это то, что осталось от входной таблицы и не было определено ни как распределяющий, ни как агрегатный элемент. Поэтому рекомендуется всегда определять табличное выражение, возвращающее группирующий, распределяющий и агрегатный элементы, и использовать эту таблицу как вход для оператора PIVOT.
- Функции LAG и LEAD вполне естественны для этой цели.

### Упражнение 2. Прохождение собеседования на позицию разработчика

- Функция ROW\_NUMBER не чувствительна к связям в значениях сортировки окна. Поэтому такое вычисление является детерминированным только тогда, когда сортировка окна уникальна. Если сортировка окна неуникальна, эта функция не является детерминированной. Функция RANK чувствительна к связям и дает одноковое значение ранга для всех строк с одинаковым значением сортировки. Поэтому она является детерминированной, даже если сортировка окна неуникальна.

2. Разница между предложениями `ROWS` и `RANGE` в действительности подобна разнице между `ROW_NUMBER` и `RANK` соответственно. Если сортировка окна неуникальна, `ROWS` не включает одноранговые члены и, таким образом, не является детерминированной, тогда как `RANGE` включает одноранговые члены и, таким образом, является детерминированной. Также опция `ROWS` может быть оптимизирована с эффективной подкачкой "в памяти"; опция `RANGE` оптимизируется с подкачкой "на диске" и, следовательно, как правило, является более медленной.
3. Оконные функции разрешены только в предложениях `SELECT` и `ORDER BY`, поскольку исходное окно, с которым они должны работать, является результирующим набором базового запроса. Если вам нужно отфильтровать строки с помощью оконной функции, следует использовать табличное выражение, такое как СТЕ или производная таблица. Нужно указать оконную функцию в предложении `SELECT` внутреннего запроса и присвоить псевдоним целевому столбцу. Затем можно фильтровать строки, ссылаясь на этот псевдоним столбца в предложении `WHERE` внешнего запроса.

## Глава 6

---

### Занятие 1

#### Закрепление материала

1. Правильные ответы: А и D.
  - A. **Правильно:** стоп-слова содержат лишние слова.
  - B. **Неправильно:** тезаурус используется для синонимов.
  - C. **Неправильно:** парадигматический модуль используется генерации словоформ слов.
  - D. **Правильно:** стоп-слова группируются в стоп-списки.
2. Правильный ответ: С.
  - A. **Неправильно:** база данных msdb устанавливается по умолчанию и используется агентом SQL Server.
  - B. **Неправильно:** база данных распространителя устанавливается по умолчанию и используется для репликации.
  - C. **Правильно:** вам нужна база данных semanticsdb для того, чтобы разрешить семантический поиск.
  - D. **Неправильно:** база данных tempdb устанавливается по умолчанию и используется для всех временных объектов.
3. Правильный ответ: А.
  - A. **Правильно:** можно добавить синонимы, редактируя файл тезауруса.
  - B. **Неправильно:** полнотекстовый поиск применяет для синонимов файлы тезауруса и не использует для этого таблицы.

- C. **Неправильно:** нельзя использовать стоп-слова для синонимов.  
D. **Неправильно:** полнотекстовый поиск использует синонимы.

## Занятие 2

### Закрепление материала

1. Правильный ответ: E.
  - A. **Неправильно:** FORMSOF — это допустимое ключевое слово предиката CONTAINS.
  - B. **Неправильно:** THESAURUS — это допустимое ключевое слово.
  - C. **Неправильно:** NEAR — это допустимое ключевое слово.
  - D. **Неправильно:** PROPERTY — это допустимое ключевое слово.
  - E. **Правильно:** TEMPORARY — это недопустимое ключевое слово предиката CONTAINS.
2. Правильный ответ: A.
  - A. **Правильно:** это выражение с учетом расположения определяет как расстояние между искомыми словами, так и их порядок.
  - B. **Неправильно:** это недопустимый синтаксис.
  - C. **Неправильно:** это выражение с учетом расположения определяет только расстояние между искомыми словами.
  - D. **Неправильно:** это выражение с учетом расположения не определяет ни расстояние между искомыми словами, ни и их порядок.
3. Правильные ответы: A, B, D и E.
  - A. **Правильно:** можно выполнять поиск словоформ слова.
  - B. **Правильно:** можно выполнять поиск синонимов искомого слова.
  - C. **Неправильно:** полнотекстовый поиск не поддерживает перевод слов.
  - D. **Правильно:** можно выполнять поиск текста, в котором искомое слово находится рядом с другим искомым словом.
  - E. **Правильно:** можно выполнять поиск только префикса слова или фразы.

## Занятие 3

### Закрепление материала

1. Правильный ответ: A.
  - A. **Правильно:** функция CONTAINSTABLE используется для ранжирования документов на основании расстояния между словами.
  - B. **Неправильно:** функция FREETEXTTABLE применяется для ранжирования документов на основании нахождения в них (включения) слов.

- C. **Неправильно:** функция SEMANTICKEYPHRASETABLE используется для возвращения ключевых фраз, связанных со столбцом с полнотекстовым индексированием.
- D. **Неправильно:** функция SEMANTICSIMILARITYTABLE используется для извлечения документов на основании схожести с указанным документом.
- E. **Неправильно:** функция SEMANTICSIMILARITYDETAILTABLE используется для возвращения ключевых фраз, которые являются общими для двух документов.
2. Правильный ответ: D.
- A. **Неправильно:** функция CONTAINSTABLE используется для ранжирования документов на основании расстояния между словами.
- B. **Неправильно:** функция FREETEXTTABLE используется для ранжирования документов на основании нахождения в них (включения) слов.
- C. **Неправильно:** функция SEMANTICKEYPHRASETABLE используется для возвращения ключевых фраз, связанных со столбцом с полнотекстовым индексированием.
- D. **Правильно:** функция SEMANTICSIMILARITYTABLE используется для извлечения документов на основании схожести с указанным документом.
- E. **Неправильно:** функция SEMANTICSIMILARITYDETAILTABLE используется для возвращения ключевых фраз, которые являются общими для двух документов.
3. Правильный ответ: C.
- A. **Неправильно:** функция CONTAINSTABLE используется для ранжирования документов на основании расстояния между словами.
- B. **Неправильно:** функция FREETEXTTABLE используется для ранжирования документов на основании нахождения в них (включения) слов.
- C. **Правильно:** функция SEMANTICKEYPHRASETABLE используется для возвращения ключевых фраз, связанных со столбцом с полнотекстовым индексированием.
- D. **Неправильно:** функция SEMANTICSIMILARITYTABLE используется для извлечения документов на основании схожести с указанным документом.
- E. **Неправильно:** функция SEMANTICSIMILARITYDETAILTABLE используется для возвращения ключевых фраз, которые являются общими для двух документов.

## Упражнения

### Упражнение 1. Расширение поиска

1. Необходимо использовать механизм полнотекстового поиска (компонент Full-Text Search) SQL Server.

2. Необходимо исправить запросы, включив в них полнотекстовые предикаты, или использовать в них табличные функции полнотекстового и семантического поиска.

## Упражнение 2. Использование семантического поиска

1. Выбор T-SQL является более простым решением в этом задании, потому что компоненты полнотекстового и семантического поиска SQL Server поддерживают требуемую функциональность по умолчанию.
2. Следует использовать функцию SEMANTICSIMILARITYTABLE.

# Глава 7

---

## Занятие 1

### Закрепление материала

1. Правильные ответы: А и D.
  - A. **Правильно:** режим FOR XML AUTO является подходящим для создания автоматически отформатированного XML.
  - B. **Неправильно:** режима FOR XML MANUAL не существует.
  - C. **Неправильно:** режима FOR XML DOCUMENT не существует.
  - D. **Правильно:** опция FOR XML PATH позволяет явно форматировать XML.
2. Правильный ответ: С.
  - A. **Неправильно:** режим FOR XML AUTO автоматически форматирует возвращаемый XML.
  - B. **Неправильно:** с помощью параметра ROOT можно задать имя корневого элемента.
  - C. **Правильно:** используйте параметр ELEMENTS для получения элементного XML.
  - D. **Неправильно:** параметр XMLSCHEMA позволяет получить встроенный XSD.
3. Правильные ответы: В и D.
  - A. **Неправильно:** режим FOR XML AUTO автоматически форматирует возвращаемый XML.
  - B. **Правильно:** режим FOR XML EXPLICIT позволяет вручную форматировать возвращаемый XML.
  - C. **Неправильно:** режим FOR XML ROW автоматически форматирует возвращаемый XML.
  - D. **Правильно:** режим FOR XML PATH позволяет вручную форматировать возвращаемый XML.

## Занятие 2

### Закрепление материала

1. Правильный ответ: D.
  - A. **Неправильно:** предложение `for` относится к FLWOR.
  - B. **Неправильно:** предложение `let` относится к FLWOR.
  - C. **Неправильно:** предложение `where` относится к FLWOR.
  - D. **Правильно:** предложение `over` не относится к FLWOR; о здесь означает предложение `order by`.
  - E. **Неправильно:** предложение `return` относится к FLWOR.
2. Правильный ответ: C.
  - A. **Неправильно:** с помощью звездочки (\*) извлекаются все основные узлы.
  - B. **Неправильно:** с помощью `comment()` извлекаются узлы комментариев.
  - C. **Правильно:** тест типа узла `node()` позволяет извлекать все узлы.
  - D. **Неправильно:** с помощью `text()` извлекаются текстовые узлы.
3. Правильный ответ: B.
  - A. **Неправильно:** `if` не является выражением XQuery.
  - B. **Правильно:** XQuery поддерживает условное выражение `if...then...else`.
  - C. **Неправильно:** `CASE` не является выражением XQuery.
  - D. **Неправильно:** `switch` не является выражением XQuery.

## Занятие 3

### Закрепление материала

1. Правильный ответ: A.
  - A. **Правильно:** `merge()` не является методом типа данных XML.
  - B. **Неправильно:** `nodes()` является методом типа данных XML.
  - C. **Неправильно:** `exist()` является методом типа данных XML.
  - D. **Неправильно:** `value()` является методом типа данных XML.
2. Правильные ответы: А и В.
  - A. **Правильно:** индекс PRIMARY XML должен создаваться раньше всех других XML-индексов.
  - B. **Правильно:** индекс PATH XML особенно полезен, если в запросах указывается выражение пути.
  - C. **Неправильно:** не существует общего индекса ATTRIBUTE XML.
  - D. **Неправильно:** не существует общего индекса PRINCIPALNODES XML.

3. Правильный ответ: В.

- A. **Неправильно:** метод `modify()` используется для обновления XML-данных.
- B. **Правильно:** метод `nodes()` используется для дробления XML-данных.
- C. **Неправильно:** метод `exist()` используется для проверки, существует ли узел.
- D. **Неправильно:** метод `value()` используется для извлечения скалярного значения из XML-данных.

## Упражнения

### Упражнение 1. Создание отчетов из XML-данных

1. Вы можете использовать метод `value()` типа данных XML для извлечения скалярных значений, требуемых в отчете.
2. Вы должны рассмотреть возможность использования XML-индексов для того, чтобы максимально повысить производительность отчета.

### Упражнение 2. Динамическая схема

1. Вы можете использовать XML-тип данных `column` для хранения атрибутов переменной в XML-формате.
2. Вам следует проверить соответствие вашего XML коллекции схем XML.

## Глава 8

---

### Занятие 1

#### Закрепление материала

1. Правильные ответы: А, С и D.
  - A. **Правильно:** регулярный идентификатор может состоять из всех символов алфавита.
  - B. **Неправильно:** регулярный идентификатор не может включать в себя пробел.
  - C. **Правильно:** регулярный идентификатор может включать в себя знак доллара (\$).
  - D. **Правильно:** регулярный идентификатор может содержать подчеркивание (\_).
2. Правильный ответ: В.
  - A. **Неправильно:** тип данных VARBINARY предназначен для хранения двоичных данных общего назначения и не может заменить тип TIMESTAMP.
  - B. **Правильно:** тип данных ROWVERSION заменяет устаревший тип TIMESTAMP.
  - C. **Неправильно:** тип данных DATETIME2 хранит типы данных даты и времени и не может заменить тип TIMESTAMP.

- D. **Неправильно:** тип данных TIME хранит только данные в формате времени и не может заменить тип TIMESTAMP.
3. Правильный ответ: D.
- A. **Неправильно:** указание NULL должно идти после типа данных.
- B. **Неправильно:** ALLOW NULL не является допустимой конструкцией инструкции CREATE TABLE.
- C. **Неправильно:** PERMIT NULL не является допустимой конструкцией инструкции CREATE TABLE..
- D. **Правильно:** следует указывать NULL сразу после типа данных.

## Занятие 2

### Закрепление материала

1. Правильные ответы: В и D.
- A. **Неправильно:** суррогатные ключи должны быть незначащими, а время — это значащее число. Кроме того, нельзя гарантировать, что две строки не могут быть добавлены почти в одно и то же время.
- B. **Правильно:** автоматически увеличивающееся целочисленное значение обычно используется в качестве суррогатного ключа, поскольку оно не влияет на значащие данные строки и будет уникальным для каждой строки.
- C. **Неправильно:** суррогатный ключ не должен иметь значащих данных, таких как часть идентификатора пользователя и имя пользователя.
- D. **Правильно:** уникальный идентификатор (GUID) тоже может использоваться как суррогатный ключ, когда он уникальным образом генерируется для каждой строки.
2. Правильный ответ: C.
- A. **Неправильно:** ограничение уникальности только обеспечивает уникальность и не может проверять существование значения в другой таблице.
- B. **Неправильно:** ограничение по умолчанию только задает значение по умолчанию. Оно не может проверять существование значения в другой таблице.
- C. **Правильно:** ограничение внешнего ключа проверяет существование значения в другой таблице.
- D. **Неправильно:** ограничение первичного ключа гарантирует уникальность и не может проверять существование значения в другой таблице.
3. Правильные ответы: А, С и D.
- A. **Правильно:** sys.key\_constraints перечисляет все первичные ключи и ограничения уникальности в базе данных.

- В.** **Неправильно:** `sys.indexes` не перечисляет ограничения уникальности в базе данных.
- С.** **Правильно:** `sys.default_constraints` перечисляет ограничения уникальности в базе данных.
- Д.** **Правильно:** `sys.foreign_keys` перечисляет все первичные ключи в базе данных.

## Упражнения

### Упражнение 1. Работа с ограничениями таблиц

1. Можно обеспечить уникальность определенных столбцов или их комбинаций в таблице, применив ограничения первичного ключа и ограничения уникальности. Также можно применить уникальный индекс. Обычно предпочтительно использовать объявленное ограничение первичного ключа и уникальности, поскольку они могут быть легко найдены и распознаны в метаданных и административных инструментах SQL Server. Если уникальность строки не может быть задана с помощью ограничения или уникального индекса, вы можете попробовать применить триггер.
2. Для простых ограничений диапазонов в таблице можно использовать проверочное ограничение. Затем вы можете указать это ограничение в значении выражения ограничения.
3. Для обеспечения правильности искомых значений в основном следует использовать ограничения внешнего ключа. Ограничения внешнего ключа — это объявленные ограничения и поэтому известны SQL Server и оптимизатору запросов через метаданные. При соединении таблицы, которая имеет ограничение внешнего ключа, с ее таблицей уточняющих запросов (lookup table) полезно добавить индекс на столбец внешнего ключа для повышения производительности соединения.
4. Нельзя обеспечить применение ограничения первичного ключа для каждой таблицы. Но вы можете запросить `sys.key_constraints` с целью контроля, что каждая таблица действительно содержит первичный ключ.

### Упражнение 2. Использование ограничений уникальности и ограничений по умолчанию

1. Можно создать ограничение уникальности на столбце или наборе столбцов для обеспечения уникальности их значений, в дополнение к первичному ключу.
2. Можно предотвратить наличие значений `NULL` в столбце, изменив таблицу и переопределив столбец как `NOT NULL`.
3. Можно создать ограничение по умолчанию на столбце и, таким образом, обеспечить, что если не вставлены никакие значения, на их место будут вставлены значения по умолчанию.

# Глава 9

## Занятие 1

### Закрепление материала

1. Правильные ответы: А, С и D.
  - A. **Правильно:** представление может содержать предложение WHERE.
  - B. **Неправильно:** представление может содержать инструкцию ORDER BY, если используется предложение SELECT TOP, но реальная сортировка результатов не гарантируется.
  - C. **Правильно:** инструкции SELECT можно объединить в представлении с помощью операций UNION и UNION ALL.
  - D. **Правильно:** представление может содержать предложение GROUP BY.
2. Правильный ответ: С.
  - A. **Неправильно:** вы всегда можете изменить представление, не изменяя базовой таблицы или таблиц.
  - B. **Неправильно:** если даже вы изменяете представление, в случае применения к представлению предложения WITH SCHEMABINDING, базовая таблица не может быть изменена.
  - C. **Правильно:** предложение WITH SCHEMABINDING означает, что схемы базовой таблицы зафиксированы представлением. Для изменения этой таблицы нужно сначала удалить представление.
  - D. **Неправильно:** никогда не нужно удалять таблицы, чтобы изменить представление.
3. Правильный ответ: С.
  - A. **Неправильно:** предложение WITH CHECK OPTION не препятствует обновлению данных через представление.
  - B. **Неправильно:** предложение WITH CHECK OPTION не ограничивает все обновления только столбцами первичного ключа.
  - C. **Правильно:** назначение предложения WITH CHECK OPTION — предотвратить любые обновления, в результате которых строки могут нарушать правильность предложения WHERE представления. Оно также препятствует обновлению любых строк, находящихся вне действия фильтра предложения WHERE.
  - D. **Неправильно:** предложение WITH CHECK OPTION не имеет никакого отношения к проверочным ограничениям таблицы.

## Занятие 2

### Закрепление материала

1. Правильные ответы: А и С.
  - A. **Правильно:** синонимы могут ссылаться на хранимые процедуры.

- B. **Неправильно:** синонимы не могут ссылаться на индексы; индексы не являются объектами базы данных, находящимися в пределах области действия имен схем.
- C. **Правильно:** синонимы могут ссылаться на временные таблицы.
- D. **Неправильно:** пользователи базы данных не являются объектами базы данных, находящимися в пределах области действия имен схем.
2. Правильные ответы: A и D.
- A. **Правильно:** синонимы — это просто имена, они не хранят код T-SQL или какие-либо данные.
- B. **Неправильно:** синонимы — это объекты базы данных, ограниченные областью действия схем базы данных, так же как таблицы, представления, функции и хранимые процедуры, поэтому им нужны имена схем.
- C. **Неправильно:** имя синонима (имя схемы плюс имя объекта) не может быть таким же, как у любого другого действующего в пределах схемы объекта базы данных, включая другие синонимы.
- D. **Правильно:** синонимы могут ссылаться на другие объекты базы данных с помощью трехкомпонентных имен и на объекты через связанные серверы, используя имена, состоящие из четырех частей.
3. Правильный ответ: C.
- A. **Неправильно:** только представления могут быть созданы с опцией WITH SCHEMABINDING, но не синонимы.
- B. **Неправильно:** синонимы не могут ссылаться на другие синонимы; цепочки синонимов не разрешены.
- C. **Правильно:** можно создать синоним, который ссылается на несуществующий объект. Но чтобы использовать такой синоним, необходимо обеспечить существование объекта.
- D. **Неправильно:** синонимы всегда требуют имени схемы.

## Упражнения

### Упражнение 1. Сравнение представлений, встроенных функций и синонимов

- Чтобы избавить разработчиков от необходимости использовать сложные соединения, им можно предоставить представления и встроенные функции, которые скрывают сложность соединений. Поскольку для обновления данных они будут использовать хранимые процедуры, вам не требуется обеспечивать обновляемость представлений.
- Вы можете изменить имена или определения представлений и изменить имена таблиц, не затрагивая приложение, если приложение использует синонимы. Вы должны будете удалить и снова создать синоним при изменении имени базовой

таблицы или представления, и это должно делаться, когда приложение находится в автономном режиме.

- Вы можете использовать встроенные функции для создания подобных представлению объектов, которые могут фильтроваться по параметрам. Хранимые процедуры не нужны, поскольку пользователи могут ссылаться на встроенные функции из предложения `FROM` запроса.

## Упражнение 2. Преобразование синонимов в другие объекты

1. Для фильтрования данных, получаемых из таблицы, можно создать представление или встроенную функцию, которые фильтруют соответствующим образом данные, и заново создать синоним для ссылки на это представление или функцию.
2. Чтобы синонимы работали даже в случае изменения имен столбцов, можно создать представление, которое ссылается на таблицы, и заново создать синоним для ссылки на это представление.
3. Синонимы не могут показывать метаданные. Поэтому при просмотре базы данных в среде SSMS пользователи не увидят имена столбцов и их типы данных под синонимом. Чтобы предоставить пользователям возможность видеть типы данных столбцов базовых таблиц, вы должны заменить синонимы представлениями.

# Глава 10

---

## Занятие 1

### Закрепление материала

1. Правильный ответ: D.
  - A. **Неправильно:** при желании вы имеете возможность не указывать столбец и позволить ограничению по умолчанию сгенерировать значение, но это не означает, что вы должны пропустить его. Если хотите, вы можете указать собственное значение.
  - B. **Неправильно:** еще раз, при желании вы имеете возможность не указывать столбец и позволить SQL Server присвоить значение `NULL` столбцу, но это не означает, что вы должны пропустить его. Если хотите, вы можете указать собственное значение.
  - C. **Неправильно:** если столбец не принимает значения `NULL` и не получает свое значение автоматически каким-либо способом, вы должны указать его.
  - D. **Правильно:** если столбец имеет свойство `IDENTITY`, обычно нужно пропустить его в инструкции `INSERT` и предоставить свойству возможность присвоить значение. Чтобы указать собственное значение, следует установить параметр `IDENTITY_INSERT`, но это, как правило, не используется.

2. Правильные ответы: А, В и D.

- А. **Правильно:** инструкция SELECT INTO не копирует индексы.
- В. **Правильно:** инструкция SELECT INTO не копирует ограничения.
- С. **Неправильно:** инструкция SELECT INTO копирует свойство IDENTITY.
- Д. **Правильно:** инструкция SELECT INTO не копирует триггеры.

3. Правильные ответы: А и С.

- А. **Правильно:** инструкция SELECT INTO поддерживает только ограниченный контроль над определением целевого столбца, в отличие от ее альтернативы, имеющей полный контроль над ним.
- В. **Неправильно:** инструкция INSERT SELECT обычно действует не быстрее, чем инструкция SELECT INTO. В действительности, во многих случаях инструкция SELECT INTO может выигрывать за счет ведения минимального журнала.
- С. **Правильно:** инструкция SELECT INTO блокирует как данные, так и метаданные, и поэтому может вызывать блокирование, связанное с тем и другим. Если инструкции CREATE TABLE и INSERT SELECT выполняются в разных транзакциях, блокировка на метаданных держится только в течение очень короткого периода времени.
- Д. **Неправильно:** все наоборот — инструкция SELECT INTO требует меньшего кодирования, поскольку вам не нужно определять целевую таблицу.

## Занятие 2

### Закрепление материала

1. Правильный ответ: С.

- А. **Неправильно:** поддержка объединений в обновлении не обеспечивает решение задачи за одно обращение к строке.
- В. **Неправильно:** поддержка обновлений на основе табличных выражений не обеспечивает решение задачи за одно обращение к строке.
- С. **Правильно:** инструкция UPDATE с использованием переменной может изменить значение столбца, и поместить результат в переменную за одно обращение к строке.
- Д. **Неправильно:** задача может быть решена, как описано в ответе С.

2. Правильные ответы: А и С.

- А. **Правильно:** объединение может использоваться для фильтрации обновленных данных.
- В. **Неправильно:** нельзя выполнить обновление нескольких таблиц в одной инструкции UPDATE.
- С. **Правильно:** объединение дает доступ к информации в других таблицах, которые могут использоваться в исходных выражениях для присвоений.

- D. **Неправильно:** когда несколько исходных строк соответствуют одной целевой строке, получается недетерминированное обновление, в котором используется только одна исходная строка. Кроме того, факт, что такое обновление не завершается ошибкой, должен рассматриваться как недостаток, а не преимущество.
3. Правильный ответ: В.
- A. **Неправильно:** инструкция UPDATE на основе объединения не может ссылаться на оконные функции в предложении SET.
- B. **Правильно:** используя инструкцию UPDATE на основе табличного выражения, можно вызвать оконную функцию в списке SELECT внутреннего запроса. Затем вы можете сослаться на псевдоним, который вы присвоили результирующему столбцу во внешнем запросе в предложении SET внешней инструкции UPDATE.
- C. **Неправильно:** инструкция UPDATE с переменной не может ссылаться на оконные функции в предложении SET.
- D. **Неправильно:** задача может быть решена, как описано в ответе B.

## Занятие 3

### Закрепление материала

1. Правильный ответ: В.
- A. **Неправильно:** нельзя напрямую сослаться на функцию ROW\_NUMBER в предложении WHERE инструкции DELETE.
- B. **Правильно:** используя табличное выражение, можно создать результирующий столбец с помощью функции ROW\_NUMBER, а затем сослаться на псевдоним этого столбца в фильтре внешней инструкции.
- C. **Неправильно:** инструкция TRUNCATE не имеет фильтра.
- D. **Неправильно:** задача может быть решена, как описано в ответе B.
2. Правильный ответ: А.
- A. **Правильно:** инструкция DELETE записывает больше информации в журнал транзакций, чем инструкция TRUNCATE.
- B. **Неправильно:** инструкция DELETE не сбрасывает свойство IDENTITY.
- C. **Неправильно:** инструкция DELETE разрешена даже тогда, когда имеется внешний ключ, указывающий на таблицу, если нет строк, связанных с удаляемыми.
- D. **Неправильно:** инструкция DELETE разрешена, когда существует индексированное представление на основе целевой таблицы.
3. Правильные ответы: В, С и D.
- A. **Неправильно:** инструкция TRUNCATE в отличие от инструкции DELETE использует оптимизированное ведение журнала.

- В.** Правильно: инструкция TRUNCATE сбрасывает свойство IDENTITY.
- С.** Правильно: инструкция TRUNCATE запрещена, когда имеется внешний ключ, указывающий на таблицу.
- Д.** Правильно: инструкция TRUNCATE запрещена, когда существует индексированное представление на основе таблицы.

## Упражнения

### Упражнение 1. Использование модификаций, поддерживающих оптимизированное ведение журнала

1. Что касается процесса удаления, если необходимо очистить целую таблицу, клиенту стоит рассмотреть использование инструкции TRUNCATE, которая выполняется с минимальным протоколированием.
2. Что касается процесса вставки, он может осуществляться очень медленно, т. к. не использует преимуществ минимального протоколирования. Клиенту нужно оценить целесообразность использования вставки с минимальным протоколированием, например, с помощью инструкции SELECT INTO (что может потребовать сначала удалить целевую таблицу), инструкции INSERT SELECT с параметром TABLOCK и др. Обратите внимание, должна использоваться простая модель восстановления данных для базы данных или модель восстановления с неполным протоколированием, поэтому клиент должен оценить, приемлемо ли это с точки зрения требований к возможностям восстановления данных, принятых в компании.

### Упражнение 2. Усовершенствование процесса обновления данных

1. Клиент должен рассмотреть возможность разработки процесса, который организует выполнение больших обновлений по блокам. Если выполнять обновление в одной большой транзакции, процесс скорее всего приведет к значительному увеличению размера журнала транзакций. Кроме того, такой процесс, вероятно, вызовет укрупнение блокировок, которое может привести к возникновению проблем блокирования.
2. Клиент должен рассмотреть возможность использования инструкции UPDATE на основе объединений вместо существующего применения вложенных запросов. Объем кода значительно уменьшится, а производительность может возрасти. Каждый вложенный запрос требует отдельного обращения к соответствующей строке. Поэтому использование нескольких подзапросов для получения значений из нескольких столбцов приведет к нескольким обращениям к данным. С помощью объединений только за одно обращение к соответствующей строке вы можете получить любое количество значений столбцов, какое вам нужно.

# Глава 11

## Занятие 1

### Закрепление материала

1. Правильный ответ: D.
  - A. **Неправильно:** максимальное значение в таблице — это не обязательно последнее сгенерированное значение идентификатора.
  - B. **Неправильно:** функция `SCOPE_IDENTITY` определяется не на уровне таблицы, а на уровне сеанса и области.
  - C. **Неправильно:** функция `@@IDENTITY` определяется не на уровне таблицы, а на уровне сеанса.
  - D. **Правильно:** функция `IDENT_CURRENT` принимает имя таблицы на вход и возвращает последнее значение идентификатора, сгенерированное в таблице.
2. Правильные ответы: B, C и D.
  - A. **Неправильно:** оба не могут гарантировать отсутствие разрывов.
  - B. **Правильно:** одно из преимуществ использования объекта последовательности вместо свойства `IDENTITY` — это то, что можно присоединить ограничение `DEFAULT`, которое имеет вызов функции `NEXT VALUE FOR` к существующему столбцу, а также удалить это ограничение для столбца.
  - C. **Правильно:** можно сгенерировать новое значение последовательности до его использования с помощью присвоения этого значения переменной и последующего использования этой переменной в инструкции `INSERT`.
  - D. **Правильно:** вы можете указать собственное значение для столбца, имеющего свойство `IDENTITY`, но для этого требуется включить параметр сеанса `IDENTITY_INSERT`, который в свою очередь требует дополнительных разрешений. Объект последовательности является более гибким. Вы можете вставлять свои значения в столбец, который обычно получает их из объекта последовательности. И это — без необходимости включать какие-либо специальные параметры или предоставлять дополнительные разрешения.
3. Правильный ответ: A.
  - A. **Правильно:** с помощью предложения `OVER` можно контролировать порядок задания значений последовательности в инструкции `INSERT SELECT`.
  - B. **Неправильно:** добавление предложения `ORDER BY` в конец запроса не обеспечивает того, что значения последовательности будут сгенерированы в том же порядке.
  - C. **Неправильно:** использование параметра `TOP` с предложением `ORDER BY` не обеспечивает того, что значения последовательности будут сгенерированы в том же порядке.

- D. **Неправильно:** использование параметра TOP с предложением ORDER BY не обеспечивает того, что значения последовательности будут сгенерированы в том же порядке.

## Занятие 2

### Закрепление материала

1. Правильный ответ: В.
  - A. **Неправильно:** требуется только одно предложение как минимум.
  - B. **Правильно:** требуется только одно предложение как минимум, и это может быть любое из предложений WHEN.
  - C. **Неправильно:** не существует специального предложения WHEN, которое необходимо; как минимум требуется одно любое предложение.
  - D. **Неправильно:** не существует специального предложения WHEN, которое необходимо; как минимум требуется одно любое предложение.
2. Правильные ответы: А, В и D.
  - A. **Правильно:** разрешены таблицы, табличные переменные и временные таблицы.
  - B. **Правильно:** табличные выражения разрешены.
  - C. **Неправильно:** хранимые процедуры не разрешены в качестве источника в инструкции MERGE.
  - D. **Правильно:** табличные функции разрешены.
3. Правильный ответ: С.
  - A. **Неправильно:** предложение WHEN MATCHED является стандартным.
  - B. **Неправильно:** предложение WHEN NOT MATCHED является стандартным.
  - C. **Правильно:** предложение WHEN NOT MATCHED BY SOURCE не является стандартным.
  - D. **Неправильно:** предложение WHEN NOT MATCHED BY SOURCE не является стандартным.

## Занятие 3

### Закрепление материала

1. Правильный ответ: А.
  - A. **Правильно:** при ссылке на элементы из вставленных строк всегда необходимо использовать ключевое слово inserted как префикс к имени столбца.
  - B. **Неправильно:** не существует случаев, когда можно опустить ключевое слово inserted, даже если это инструкция INSERT.

- C. **Неправильно:** верно то, что вы должны добавлять префикс в виде ключевого слова `inserted` к вставленным элементам в инструкции UPDATE, но не только в инструкции UPDATE.
- D. **Неправильно:** верно то, что вы должны добавлять префикс в виде ключевого слова `inserted` к вставленным элементам в инструкции MERGE, но не только в инструкции MERGE.
2. Правильные ответы: С и D.
- A. **Неправильно:** также поддерживаются другие типы таблиц.
- B. **Неправильно:** также поддерживаются другие типы таблиц.
- C. **Правильно:** целевая таблица не может участвовать в отношении по внешнему ключу.
- D. **Правильно:** целевая таблица не может иметь триггеры, определенные на ней.
3. Правильный ответ: А.
- A. **Правильно:** на элементы из исходной таблицы можно ссылаться только в предложении OUTPUT инструкции MERGE.
- B. **Неправильно:** это также можно сделать в инструкции UPDATE.
- C. **Неправильно:** присвоение псевдонимов целевым столбцам в предложении OUTPUT разрешено во всех инструкциях.
- D. **Неправильно:** компонуемый DML поддерживает все инструкции.

## Упражнения

### Упражнение 1. Лучшее решение для генерации ключей

1. Все существующие проблемы, связанные со свойством IDENTITY, можно решить с помощью объекта последовательности.
2. С помощью объекта последовательности можно генерировать значения до их использования, с помощью вызова функции NEXT VALUE FOR и сохранения результата в переменной. В отличие от свойства IDENTITY, вы можете обновить столбец, который обычно получает свои значения из объекта последовательности. Также, поскольку объект последовательности не привязан ни к какому конкретному столбцу в определенной таблице, а напротив, является независимым объектом базы данных, вы можете генерировать значения из одной последовательности и использовать их в различных таблицах.

### Упражнение 2. Усовершенствование модификаций

1. Рекомендуемое решение — использовать инструкцию MERGE. Определите источник для инструкции MERGE как производную таблицу на основе предложения VALUES, со строкой, составленной из входных параметров для процедуры. Укажите табличную подсказку HOLDLOCK или SERIALIZABLE по отношению к целевой таблице, чтобы предотвратить такие конфликты, как существующие в системе

в данный момент. Затем используйте предложение `WHEN MATCHED` для выполнения действия `UPDATE`, если целевая строка существует, и предложение `WHEN NOT MATCHED`, чтобы выполнить действие `INSERT`, если целевая строка отсутствует.

2. Одна из возможностей — использовать уровень изоляции `SERIALIZABLE`, обрабатывая и инструкцию, которая копирует строки в архив, и инструкцию, которая удаляет строки, в одной транзакции. Но более простым решением является выполнение обеих задач в одной инструкции — инструкции `DELETE` с предложением `OUTPUT INTO`. Это гарантирует, что только строки, которые скопированы в архивную таблицу, будут удалены. И если по какой-то причине произойдет сбой копирования строк в архивную таблицу, операция удаления тоже потерпит неудачу, поскольку оба действия являются частями одной транзакции.

## Глава 12

---

### Занятие 1

#### Закрепление материала

1. Правильные ответы: А, С и D.
  - A. **Правильно:** команда `ALTER TABLE` — это команда языка DDL, которая изменяет метаданные и всегда выполняется как транзакция.
  - B. **Неправильно:** команда `PRINT` не изменяет данные и поэтому не выполняется отдельно в транзакции.
  - C. **Правильно:** инструкция `UPDATE` изменяет данные и выполняется как транзакция.
  - D. **Правильно:** инструкция `SET` влияет только на параметры сеанса и не изменяет данные, поэтому она не выполняется как транзакция.
2. Правильный ответ: В.
  - A. **Неправильно:** одна операция `COMMIT` фиксирует только самый внутренний уровень транзакции и не будет фиксировать всю вложенную транзакцию.
  - B. **Правильно:** одна операция `ROLLBACK` выполняет откат всей внешней транзакции во вложенной транзакции.
  - C. **Неправильно:** одна операция `COMMIT` фиксирует только самый внешний уровень вложенной транзакции.
  - D. **Неправильно:** одна операция `ROLLBACK` не выполняет откат только одного уровня транзакции; напротив, она осуществляет откат всей транзакции.
3. Правильные ответы: А, В и D.
  - A. **Правильно:** добавление табличной подсказки `READUNCOMMITTED` не вызывает совмещаемых блокировок, которые могут использоваться инструкцией.
  - B. **Правильно:** параметр `READ COMMITTED SNAPSHOT` выполняет чтение зафиксированных данных из версий, а не с помощью запроса совмещаемых блокировок.

- C. **Неправильно:** уровень изоляции REPEATABLE READ в действительности удерживает совмещаемые блокировки до конца транзакции и, таким образом, может увеличить блокирование и взаимоблокирование.
- D. **Правильно:** уровень изоляции SNAPSHOT также снижает совмещаемые блокировки посредством чтения зафиксированных данных из зафиксированных версий, а не с помощью совмещаемых блокировок, поэтому он также может уменьшить блокирование и взаимоблокирование.

## Занятие 2

### Закрепление материала

1. Правильный ответ: В.
- A. **Неправильно:** действительно, команда THROW не принимает параметры в блоке CATCH, но это не обязательно является преимуществом.
- B. **Правильно:** инструкция THROW в блоке CATCH может повторно вызвать ошибку и таким образом позволить создать сообщение об ошибке в блоке TRY без необходимости сохранения какой-либо предварительной информации. Это позволяет выполнять обработку ошибок в блоке CATCH.
- C. **Неправильно:** инструкция THROW всегда дает уровень серьезности 16, но это не обязательно является преимуществом. Команда RAISERROR является более гибкой, предоставляя диапазон уровней серьезности.
- D. **Неправильно:** требование наличия точки с запятой в предыдущей инструкции T-SQL — возможно, правильное требование написания кода, но это не является преимуществом, предоставляемым командой THROW.
2. Правильные ответы: А, В, С и D.
- A. **Правильно:** значение @@ERROR изменяется с каждой успешной командой, поэтому если доступ к нему может быть получен в самой первой инструкции блока CATCH, можно получить исходное сообщение об ошибке.
- B. **Правильно:** ERROR\_NUMBER() возвращает номер ошибки для исходной ошибки, которая привела к передаче управления блоку CATCH.
- C. **Правильно:** ERROR\_MESSAGE() возвращает текст исходной ошибки.
- D. **Правильно:** XACT\_STATE() сообщает состояние транзакции в блоке CATCH, в частности, является ли транзакция фиксируемой.
3. Правильный ответ: В.
- A. **Неправильно:** ошибка T-SQL с уровнем серьезности более 16 не вызывает прерывания всей транзакции.
- B. **Правильно:** ошибка T-SQL с уровнем серьезности более 10 вызывает прерывание транзакции.

- C. **Неправильно:** когда транзакция прервана параметром XACT\_ABORT, никакие другие инструкции в транзакции не будут выполнены.
- D. **Неправильно:** когда транзакция прервана параметром XACT\_ABORT, никакие другие инструкции в транзакции не будут выполнены.

## Занятие 3

### Закрепление материала

1. Правильный ответ: В.
  - A. **Неправильно:** строка комментария должна стоять последней, а внедренная одиночная кавычка — первой.
  - B. **Правильно:** начальная одиночная кавычка прерывает входную строку, а конечный знак комментария уничтожает эффект прерывающей одиночной кавычки. Поэтому вредоносный код может быть вставлен между ними.
  - C. **Неправильно:** вредоносный код должен стоять после первой одиночной кавычки и перед конечным знаком комментария.
2. Правильные ответы: А и С.
  - A. **Правильно:** параметризация — главное преимущество хранимой процедуры sp\_executesql над инструкцией EXEC(), поскольку она гарантирует, что любой внедренный код будет виден только как значение строкового параметра, а не как исполняемый код.
  - B. **Неправильно:** хотя хранимая процедура sp\_executesql требует строк Unicode в качестве параметров, это не обязательно является преимуществом. Команда EXECUTE принимает и формат Unicode, и отличные от него форматы, и таким образом может считаться более гибкой.
  - C. **Правильно:** использование выходных параметров разрешает многие ограничения команды EXECUTE. Команда EXECUTE не может напрямую возвращать информацию вызываемому сеансу.
3. Правильные ответы: А и D.
  - A. **Правильно:** если параметр QUOTED\_IDENTIFIER установлен в ON, можно использовать двойные кавычки для разделения идентификаторов T-SQL, таких как имена таблиц и столбцов.
  - B. **Неправильно:** если параметр QUOTED\_IDENTIFIER установлен в OFF, нельзя использовать двойные кавычки для разделения идентификаторов T-SQL, таких как имена таблиц и столбцов.
  - C. **Неправильно:** если параметр QUOTED\_IDENTIFIER установлен в ON, нельзя использовать двойные кавычки для разделения строк.
  - D. **Правильно:** если параметр QUOTED\_IDENTIFIER установлен в OFF, можно использовать двойные кавычки для разделения строк.

## Упражнения

### Упражнение 1. Реализация обработки ошибок

1. Всякий раз, когда происходит более чем одно изменение данных в хранимой процедуре, и важно, чтобы изменения данных обрабатывались как логическая единица работы, необходимо добавлять логику транзакции в хранимую процедуру.
2. Вам необходимо привести уровни изоляции в соответствие требованиям согласованности транзакций. Вы должны изучить существующее приложение и базу данных на наличие элементов блокирования и особенно взаимоблокирования. Если вы найдете взаимоблокировки и установите, что они вызваны не ошибками в кодах T-SQL, можно использовать различные способы снижения уровня изоляции, чтобы сделать взаимоблокировки менее вероятными. Но следует учитывать, что некоторые транзакции могут требовать более высоких уровней изоляции.
3. Вам следует использовать блоки TRY/CATCH в каждой хранимой процедуре, где могут возникать ошибки, и рекомендовать вашей команде сделать такое использование стандартом. Направляя все ошибки в блок CATCH, можно обрабатывать ошибки в одном месте кода.
4. Проверьте хранимые процедуры на использование динамического SQL и где возможно замените вызовы команды EXECUTE хранимой процедурой sp\_executesql.

### Упражнение 2. Реализация транзакций

1. Чтобы гарантировать, что ни при каком чтении данных в транзакции данные не будут изменяться до конца транзакции, можно использовать уровень изоляции транзакции REPEATABLE READ. Это наименее строгий уровень, который будет удовлетворять требованиям.
2. При развертывании новых объектов базы данных с помощью сценариев T-SQL можно запаковать пакеты в одну транзакцию и использовать настройку SET XACT\_ABORT ON сразу после инструкции BEGIN TRANSACTION. Затем, если возникнет какая-либо ошибка T-SQL, вся транзакция будет прервана, и вам не придется добавлять сложную обработку ошибок.
3. Чтобы гарантировать, что для чтения диапазона значений транзакцией никакие строки, для которых выполняется чтение, не были изменены и что никакие строки не смогут быть вставлены либо удалены, можно использовать уровень изоляции SERIALIZABLE. Это наиболее строгий уровень изоляции, который может привести к множеству блокировок, поэтому следует убедиться, что транзакции будут завершаться как можно быстрее.

## Глава 13

---

### Занятие 1

#### Закрепление материала

1. Правильные ответы: А и С.
  - A. **Правильно:** инструкция WHILE запускает циклическую структуру.
  - B. **Неправильно:** BEGIN и END не вызывают ветвления, они используются только для группирования инструкций.
  - C. **Правильно:** IF и ELSE вызывают ветвление выполнения кода на основе условия, указанного в предложении IF.
  - D. **Неправильно:** инструкция GO только прерывает пакет. Сама по себе она не влияет на выполнение кода.
2. Правильный ответ: С.
  - A. **Неправильно:** переменные вызывающей процедуры не могут быть видны вызываемой процедурой.
  - B. **Неправильно:** временные таблицы видны, но передаваемые параметры также видны.
  - C. **Правильно:** вызываемая процедура видит временные таблицы и параметры, передаваемые ей вызывающей процедурой.
  - D. **Неправильно:** вызываемая процедура может видеть временные таблицы и передаваемые параметры из вызывающей процедуры.
3. Правильный ответ: В.
  - A. **Неправильно:** можно использовать выходной параметр для возвращения информации из хранимой процедуры.
  - B. **Правильно:** вы можете передавать данные в хранимую процедуру и извлекать информацию из нее с помощью выходного параметра.
  - C. **Неправильно:** выходной параметр используется не только для передачи данных обратно вызывающей стороне хранимой процедуры. Он также используется для передачи данных от вызывающей стороны в хранимую процедуру.
  - D. **Неправильно:** вы можете передавать данные в хранимую процедуру и извлекать информацию из нее с помощью выходного параметра.

### Занятие 2

#### Закрепление материала

1. Правильный ответ: С.
  - A. **Неправильно:** в случае инструкции DELETE вставленная таблица будет пустой, поскольку нет новых или измененных строк.

- B. **Неправильно:** вставленная и удаленная таблицы содержат строки не только для инструкций `INSERT` и `DELETE`, но и для инструкции `UPDATE`.
- C. **Правильно:** инструкция `INSERT` содержит все вставленные строки во вставленной таблице и не содержит строк в удаленной таблице.
- D. **Неправильно:** в случае инструкции `UPDATE`, которая обновляет строки в таблице, изменяемые строки появятся во вставленной таблице со своими новыми значениями, и в удаленной таблице со своими старыми значениями.
2. Правильные ответы: A и D.
- A. **Правильно:** вы можете создать триггеры `INSTEAD OF` на представлениях для перенаправления вставок или обновлений в базовые таблицы.
- B. **Неправильно:** триггеры `INSTEAD OF` выполняются вместо инструкций DML, а не вместо триггеров `AFTER`.
- C. **Неправильно:** триггеры `INSTEAD OF` могут быть объявлены для всех инструкций DML — `INSERT`, `UPDATE` и `DELETE`.
- D. **Правильно:** используя триггеры `INSTEAD OF`, можно подставить код триггера вместо исходной инструкции DML.
3. Правильный ответ: D.
- A. **Неправильно:** `OFF` не является допустимым значением второго параметра хранимой процедуры `sp_configure`.
- B. **Неправильно:** Для параметра '`nested triggers`' требуется дополнительная инструкция `RECONFIGURE`.
- C. **Неправильно:** не является допустимым значением второго параметра хранимой процедуры `sp_configure`.
- D. **Правильно:** После хранимой процедуры `sp_configure`, стоящей перед '`nested triggers`' и 0, вы должны также выполнить инструкцию `RECONFIGURE`.

## Занятие 3

### Закрепление материала

1. Правильный ответ: C.
- A. **Неправильно:** пользовательские скалярные функции никогда не бывают встроенными. Только определяемые пользователем функции с табличным значением могут быть встроенными.
- B. **Неправильно:** результатом инструкции `SELECT` может быть таблица, а пользовательские скалярные функции не могут возвращать таблицу.
- C. **Правильно:** можно вызвать пользовательскую скалярную функцию в списке `SELECT` или в условном предложении `WHERE`, в любом месте, где скалярное значение применимо.
- D. **Неправильно:** предложение `FROM` требует таблицу, а пользовательские скалярные функции не могут возвращать таблицу.

2. Правильный ответ: D.

- A. **Неправильно:** определяемые пользователем функции с табличным значением возвращают только таблицы.
- B. **Неправильно:** встроенные определяемые пользователем функции с табличным значением состоят только из одной инструкции T-SQL. Даже многооператорные определяемые пользователем функции с табличным значением требуют только одной инструкции T-SQL.
- C. **Неправильно:** вызов в списке SELECT или предложении WHERE будет требовать скалярного значения, а определяемые пользователем функции с табличным значением возвращают только таблицы.
- D. **Правильно:** предложение FROM требует таблицу, и определяемые пользователем функции с табличным значением возвращают таблицы.

3. Правильный ответ: C.

- A. **Неправильно:**строенная пользовательская функция с табличным значением не определяет схему табличной структуры, которую она возвращает.
- B. **Неправильно:**строенная пользовательская функция с табличным значением не может создать постоянную таблицу.
- C. **Правильно:** многооператорная пользовательская функция с табличным значением определяет явную схему табличной переменной и затем вставляет данные в эту табличную переменную.
- D. **Неправильно:** многооператорная пользовательская функция с табличным значением не может создать постоянную таблицу.

## Упражнения

### Упражнение 1. Реализация хранимых процедур и определяемых

1. Чтобы предотвратить несогласованность базы данных, следите за тем, чтобы были установлены нужные ограничения: ограничения первичного и уникального ключей на таблицах, проверочные ограничения на столбцах и ограничения внешнего ключа между таблицами. Прочие более сложные бизнес-правила можно обеспечить с помощью триггеров.
2. Чтобы представить стандартный интерфейс к базе данных, используйте хранимую процедуру уровня данных, т. е. используйте стандартные хранимые процедуры вставки, обновления и удаления для каждой таблицы. Клиентское программное обеспечение должно только изменять данные в таблицах с помощью этих хранимых процедур.
3. Вы можете использовать функции, возвращающие табличное значение, вместо представлений и определить параметры для удовлетворения требований разработчиков приложения. Затем вы можете вызвать функцию изнутри хранимой процедуры, которая принимает эти параметры и отправляет результаты обратно клиенту.

4. Рассмотрите возможность создания хранимой процедуры поиска, которая состоит из драйвера, и заставьте ее вызывать вложенные процедуры, по одной для каждой комбинации параметра. Эти вложенные процедуры всегда будут иметь тот же план запроса, поэтому перекомпиляция процедур не потребуется.

## Упражнение 2. Реализация триггеров

1. Ограничения внешнего ключа можно реализовать с помощью триггеров, но код может стать сложным и ненадежным. Вместо того вы можете порекомендовать разработчикам базы данных реализовать ссылочную целостность с помощью объявленных ограничений внешнего ключа T-SQL вместо триггеров.
2. Вы можете рекомендовать, чтобы приложение запретило вложенные триггеры на сервере разработки, так чтобы разработчики баз данных привыкли к выполнению всех необходимых действий в пределах одного уровня триггера. Это поможет упростить код триггера и повысить возможность его отладки.
3. Когда приложение вставляет данные в одну таблицу и должно также выполнять вставку во вспомогательные таблицы в рамках того же действия, вы можете порекомендовать разработчикам базы данных использовать для этого триггер `INSTEAD OF`. В этом триггере несколько вставок могут выполняться перед вставкой в главную таблицу.
4. Для поддержки простого ведения журнала вы можете рекомендовать разработчикам использовать DML-триггер `AFTER`. Триггер этого типа выполняется после инструкций `INSERT`, `UPDATE` или `DELETE` и может осуществлять запись в таблицу ведения журнала.

# Глава 14

---

## Занятие 1

### Закрепление материала

1. Правильные ответы: В и С.
  - А. **Неправильно:** алгебраическое дерево генерируется на этапе связывания.
  - В. **Правильно:** на этапе оптимизации SQL Server генерирует кандидат-планы.
  - С. **Правильно:** на этапе оптимизации SQL Server выбирает план выполнения из коллекции кандидат-планов.
  - Д. **Неправильно:** план кэшируется на этапе выполнения.
  - Е. **Неправильно:** запрос выполняется на этапе выполнения.
2. Правильный ответ: В.
  - А. **Неправильно:** на этапе синтаксического разбора SQL Server выполняет проверку на синтаксическую корректность.

- B. **Правильно:** SQL Server выполняет разрешение имен объектов и их связывание в логические операторы на этапе связывания.
- C. **Неправильно:** на этапе оптимизации SQL Server генерирует кандидат-планы и выбирает план выполнения.
- D. **Неправильно:** на этапе выполнения SQL Server выполняет запрос и кэширует план выполнения.
3. Правильный ответ: C.
- A. **Неправильно:** предикаты — это объекты пакета расширенных событий.
- B. **Неправильно:** цели — это объекты пакета расширенных событий.
- C. **Правильно:** источники не входят в состав пакета расширенных событий.
- D. **Неправильно:** действия — это объекты пакета расширенных событий.

## Занятие 2

### Закрепление материала

1. Правильные ответы: A и D.
- A. **Правильно:** параметр сеанса SET STATISTICS IO полезен для анализа производительности запроса.
- B. **Неправильно:** не существует параметра SET STATISTICS EXECUTION\_DETAILS.
- C. **Неправильно:** параметр SET IDENTITY\_INSERT используется для предоставления значения для столбца, имеющего свойство идентификатора.
- D. **Правильно:** параметр сеанса SET STATISTICS TIME полезен для анализа производительности запроса.
2. Правильный ответ: D.
- A. **Неправильно:** планы выполнения следует читать справа налево, сверху вниз.
- B. **Неправильно:** планы выполнения следует читать справа налево, сверху вниз.
- C. **Неправильно:** планы выполнения следует читать справа налево, сверху вниз.
- D. **Правильно:** планы выполнения следует читать справа налево, сверху вниз.
3. Правильные ответы: B и D.
- A. **Неправильно:** не существует команды SET EXECUTION\_XML.
- B. **Правильно:** команда SET SHOWPLAN\_XML используется для включения предлагаемых XML-планов.
- C. **Неправильно:** не существует команды SET XML PLAN.
- D. **Правильно:** команда SET STATISTICS XML используется для включения фактических XML-планов.

## Занятие 3

### Закрепление материала

1. Правильный ответ: С.
  - A. **Неправильно:** динамическое административное представление `sys.dm_exec_query_stats` предоставляет статистические данные о запросах, а не об индексах.
  - B. **Неправильно:** динамическое административное представление `sys.dm_exec_query_text` предоставляет текст пакетов и запросов.
  - C. **Правильно:** динамическое административное представление `sys.dm_db_index_usage_stats` предоставляет информацию об использовании индексов.
  - D. **Неправильно:** `sys.indexes` — это представление каталога, а не динамическое административное представление.
2. Правильный ответ: А.
  - A. **Правильно:** отсутствие необходимого количества данных является самым важным недостатком динамического управляющего объекта.
  - B. **Неправильно:** хотя некоторые запросы, использующие динамические управляющие объекты, становятся довольно сложными, с этим легко справиться, узнав больше о T-SQL и динамических управляющих объектах.
  - C. **Неправильно:** динамические управляющие объекты доступны во всех выпусках SQL Server.
  - D. **Неправильно:** динамические управляющие объекты — это системные объекты; вы не можете удалять или создавать их.
3. Правильный ответ: В.
  - A. **Неправильно:** динамическое административное представление `sys.dm_exec_query_stats` не предоставляет текст запроса.
  - B. **Правильно:** текст запроса можно получить, выполнив запрос к функции динамического управления `sys.dm_exec_sql_text`.
  - C. **Неправильно:** динамическое административное представление `sys.dm_exec_query_plan` не предоставляет текст запроса.
  - D. **Неправильно:** текст запроса можно получить, выполнив запрос к функции динамического управления `sys.dm_exec_sql_text`.

## Упражнения

### Упражнение 1. Анализ запросов

1. Вам нужно использовать связанные с выполнением динамические административные объекты для нахождения наиболее проблемных запросов.
2. Вы можете использовать графический предполагаемый план выполнения для этого запроса, чтобы найти операторы, имеющие наивысшую стоимость. Также

вы можете проверить, имеются ли отсутствующие индексы, найденные связанными с индексом динамическими административными представлениями, которые могут быть полезны для того запроса.

## Упражнение 2. Непрерывный мониторинг

1. Вам следует использовать подсистему расширенных событий SQL Server, как наиболее легкую систему мониторинга производительности.
2. В вашем случае вы должны выполнять мониторинг только нескольких самых важных событий. Вам следует собирать данные только для необходимых вам полей. Вы также должны отфильтровать события сеанса, чтобы включать только события, которые вам действительно нужны.

---

# Глава 15

## Занятие 1

### Закрепление материала

1. Правильные ответы: А, С и D.
  - Правильно: индекс может иметь 0 и более промежуточных уровней.
  - Неправильно: куча — это отдельная структура, а не уровень индекса.
  - Правильно: каждый индекс имеет корневой уровень с одной корневой страницей.
  - Правильно: самый нижний уровень индекса — это конечный уровень.
2. Правильный ответ: С.
  - Неправильно: можно создать 999 некластеризованных индексов на таблице.
  - Неправильно: можно иметь до 16 столбцов в составном ключе.
  - Правильно: может быть только 1 кластеризованный индекс, поскольку это сама таблица, построенная на сбалансированном дереве.
  - Неправильно: размер столбцов в ключе не должен превышать 900 байт.
3. Правильный ответ: С.
  - Неправильно: RID используется для куч.
  - Неправильно: столбцы в индексе columnstore не используются как указатели ключа.
  - Правильно: ключ кластеризации является указателем строки, когда таблица хранится как сбалансированное дерево.
  - Неправильно: кластеризованная таблица хранится как сбалансированное дерево.

## Занятие 2

### Закрепление материала

1. Правильный ответ: В.
  - A. **Неправильно:** использование инструкции `SELECT *` очень нежелательно и конечно не помогает SQL Server использовать индексы.
  - B. **Правильно:** вы можете модифицировать индекс, который уже используется, так, чтобы включить столбцы из списка `SELECT`, не являющиеся частью ключа.
  - C. **Неправильно:** добавление псевдонимов столбцов не влияет на использование индексов.
  - D. **Неправильно:** вы можете поддерживать предложение `SELECT` индексами.
2. Правильный ответ: D.
  - A. **Неправильно:** SQL Server сортирует данные в памяти или сбрасывает данные в tempdb, если они не помещаются в памяти.
  - B. **Неправильно:** SQL Server сортирует данные в памяти или сбрасывает данные в tempdb, если они не помещаются в памяти.
  - C. **Неправильно:** SQL Server сортирует данные в памяти или сбрасывает данные в tempdb, если они не помещаются в памяти.
  - D. **Правильно:** SQL Server сортирует данные в памяти или сбрасывает данные в tempdb, если они не помещаются в памяти.
3. Правильные ответы: А и С.
  - A. **Правильно:** SQL Server не использует индекс для поддержки предложения `WHERE`, если по аргументам предиката не может выполняться поиск.
  - B. **Неправильно:** SQL Server поддерживает предложение `WHERE` индексами.
  - C. **Правильно:** SQL Server может решить не использовать индекс для поддержки предложения `WHERE`, если запрос является недостаточно выборочным.
  - D. **Неправильно:** SQL Server рассматривает использование индексов в контексте базы данных tempdb точно так же, как в контексте любой другой базы данных.

## Занятие 3

### Закрепление материала

1. Правильный ответ: D.
  - A. **Неправильно:** информации о мощности нет на страницах конечного уровня или в индексах.
  - B. **Неправильно:** SQL Server не выполняет запрос заранее на данных образца.
  - C. **Неправильно:** SQL Server может предварительно рассчитать мощность запроса.
  - D. **Правильно:** SQL Server использует статистику для предварительного расчета мощности запроса.

2. Правильный ответ: А.

- A. **Правильно:** при перестройке индекса SQL Server автоматически обновляет статистику.
- B. **Неправильно:** после пакетной вставки большого количества данных в таблицу, если вы хотите выполнить запрос к ней сразу после этого, нужно обновить статистику для этой таблицы.
- C. **Неправильно:** нужно обновить статистику для всей базы данных после ее обновления.
- D. **Неправильно:** нужно обновить статистику, если запросы выполняются медленно и вы знаете, что они написаны правильно и поддерживаются надлежащими индексами.

3. Правильный ответ: D.

- A. **Неправильно:** в гистограмме можно иметь до 200 шагов.
- B. **Неправильно:** можно иметь 1 гистограмму на статистику.
- C. **Неправильно:** существует ограничение количества шагов на гистограмму.
- D. **Правильно:** в гистограмме можно иметь до 200 шагов.

## Упражнения

### Упражнение 1. Просмотр таблицы

1. Вам нужно проверить, поддерживаются ли запросы индексами. Кроме того, вам следует проверить, создается и поддерживается ли статистика для индексов.
2. Вам надо проверить, используют ли запросы надлежащие аргументы поиска.

### Упражнение 2. Медленные обновления

1. Слишком большое число индексов может замедлить обновления. Вероятно, вы создали множество ненужных индексов, но поскольку SQL Server должен их поддерживать, обновления выполняются медленно.
2. Вы можете запросить динамическое административное представление `sys.dm_db_index_usage_stats`, чтобы определить, какие индексы используются для поиска, а какие — только для обновления.

## Глава 16

---

### Занятие 1

#### Закрепление материала

1. Правильный ответ: В.

- A. **Неправильно:** 0 означает, что последняя выборка была успешной. Могут быть другие строки для выборки.
- B. **Правильно:** -1 означает, что строка находится вне результирующего набора.

- C. **Неправильно:** –2 означает, что выбранная строка отсутствует. Также могут быть другие строки для выборки.
- D. **Неправильно:** функция не должна генерировать сообщение об ошибке.
2. Правильные ответы: А и С.
- A. **Правильно:** решения на основе наборов базируются на принципах реляционной модели, которая положена в основу языков SQL (стандартный язык) и T-SQL (диалект языка в SQL Server).
- B. **Неправильно:** хотя это исключение из правила, но иногда итерационные решения могут быть более быстрыми, чем решения на основе наборов.
- C. **Правильно:** поскольку решения на основе наборов являются декларативными, а итерационные — императивными, решения на основе наборов имеют тенденцию использовать меньшее количество кода.
- D. **Неправильно:** решения на основе наборов не могут делать никаких заключений о последовательности данных, поскольку наборы данных являются неупорядоченными.
3. Правильный ответ: С.
- A. **Неправильно:** T-SQL не поддерживает цикл FOR EACH.
- B. **Неправильно:** при наличии разрывов между ключами этот подход приведет к попытке обрабатывать несуществующие ключи.
- C. **Правильно:** подход с использованием параметра TOP действительно является хорошей альтернативой курсора. Но следует принимать во внимание, что он вызывает увеличение нагрузки на ввод-вывод.
- D. **Неправильно:** в языке T-SQL не существует триггера SELECT или построчных триггеров.

## Занятие 2

### Закрепление материала

1. Правильные ответы: А, В и С.
- A. **Правильно:** табличные переменные пригодны для использования, когда таблицы очень маленькие.
- B. **Правильно:** табличные переменные пригодны для использования, когда таблицы очень маленькие.
- C. **Правильно:** при простом плане табличные переменные по-прежнему пригодны для использования, даже если они большие.
- D. **Неправильно:** если таблицы большие и план непростой, предпочтительными являются временные таблицы.
2. Правильный ответ: С.
- A. **Неправильно:** на табличных переменных могут существовать индексы.
- B. **Неправильно:** команда CREATE INDEX не поддерживается для табличных переменных.

- C. **Правильно:** можно создать индексы косвенным образом, определив ограничения первичного ключа и уникальности.
- D. **Неправильно:** внешние ключи не создают индексы; более того, они не поддерживаются на временных таблицах и табличных переменных.
3. Правильный ответ: D.
- A. **Неправильно:** после отката транзакции в триггере вставленные и удаленные таблицы очищаются.
- B. **Неправильно:** откат транзакции вызывает отмену копирования в таблицы аудита.
- C. **Неправильно:** изменения во временных таблицах отменяются после отката транзакции.
- D. **Правильно:** изменения в табличных переменных не отменяются после отката транзакции, поэтому такое решение является правильным.

## Упражнения

### Упражнение 1. Рекомендации по повышению производительности для курсоров и временных объектов

1. Клиент должен оценить использование решений на основе наборов вместо решений на основе курсоров. Если большинство существующих решений использует курсоры, возможно, проблема заключается в недостатке понимания концепций реляционной модели. Можно порекомендовать компании организовать курс обучения для разработчиков по этой теме.
2. Когда большое количество строк должно быть сохранено во временных объектах, возможность оптимизатора выполнить точный прогноз выборочности становится более важным для эффективности плана. Исключение — если план простой. SQL Server не поддерживает статистику распределения (гистограммы) на табличных переменных, поэтому прогноз может быть неточным. Неточный прогноз может привести к неоптимальному плану. Клиент должен проверить планы запросов и найти неточные прогнозы. И если они будут найдены, необходимо оценить, стоит ли в этих случаях использовать временные таблицы вместо табличных переменных. SQL Server не поддерживает гистограммы для временных таблиц, и поэтому план выполнения для них более оптимальный.
3. В некоторых случаях уместно использовать табличные переменные, например, если объем данных очень мал — страница или две. Также если таблица большая, а план простой, оптимизатору не нужны гистограммы для выбора оптимального плана. То, что табличные переменные не имеют гистограмм, дает некоторые преимущества. Вы избежите затрат на их поддержку. Также не будет затрат на повторную компиляцию планов выполнения, связанных с обновлениями гистограмм.

Что касается курсоров, в некоторых случаях необходимо выполнять обработку каждой отдельной строки в таблице. Например, в целях обслуживания может

понадобиться выполнять некоторую работу для каждого индекса, таблицы, базы данных или другого объекта. Для таких случаев предназначены курсоры. Что касается манипулирования данными, возможны случаи, когда оптимизатор запросов SQL Server не может выполнить оптимизацию запроса хорошо, и вы не можете найти способ помочь оптимизатору сгенерировать эффективный план. С помощью курсоров, несмотря на большие издержки, иногда можно получить лучшие результаты, поскольку у вас больше возможностей управления процессом. Но такие случаи являются исключениями из правил.

## Упражнение 2. Указать неточности в ответах

1. Между временными таблицами и табличными переменными существуют различия с точки зрения производительности. Одно важное различие — это поддержка в SQL Server статистики распространения (гистограмм) применительно к временным таблицам, но не к табличным переменным. Это означает, что для временных таблиц оптимизатор может сделать лучший прогноз выборочности. Поэтому планы, включающие временные таблицы, более оптимальны по сравнению с планами, включающими табличные переменные.
2. Решение с использованием циклов с запросом TOP вместо курсора не будет основано на наборах и не всегда более эффективно, чем существующее решение с использованием курсора. Вы по-прежнему обрабатываете строки по одной за один раз. Лучшим решением может быть применение одной инструкции UPDATE или MERGE, которая соединяет вставленную таблицу с базовой таблицей и обновляет все целевые строки с помощью одной операции на основе набора.
3. Результат внутреннего запроса табличного выражения не сохраняется в рабочей таблице. SQL Server раскрывает все ссылки на табличные выражения и взаимодействует с базовыми объектами напрямую. Многократные ссылки раскрываются несколько раз, поэтому работа повторяется. Если требуется сохранить результат ресурсоемкого запроса, чтобы избежать повторной работы, следует рассмотреть возможность использования временных таблиц или табличных переменных. Например, полезно использовать табличные выражения, когда необходимо ссыльаться на псевдонимы столбцов, сгенерированные в списке SELECT в предложениях, которые логически обрабатываются раньше инструкции SELECT, таких как WHERE, GROUP BY и HAVING.

# Глава 17

## Занятие 1

### Закрепление материала

1. Правильные ответы: В и С.
  - А. **Неправильно:** не существует алгоритма статистической обработки слияния.
  - В. **Правильно:** SQL Server использует алгоритмы статистической обработки потока и хэша.

- C. **Правильно:** SQL Server использует алгоритмы статистической обработки потока и хэша.
- D. **Неправильно:** не существует алгоритма статистической обработки вложенных циклов.
2. Правильный ответ: D.
- A. **Неправильно:** SQL Server использует оператор **RID Lookup** (Уточняющий запрос RID) для получения данных из базовой таблицы после поиска по некластеризованному индексу, если базовая таблица организована как куча.
- B. **Неправильно:** нет необходимости в выполнении полного просмотра кластеризованного индекса после того, как строка найдена в некластеризованном индексе.
- C. **Неправильно:** оператор **Merge Join** (Соединение слиянием) используется при выполнении соединения слиянием.
- D. **Правильно:** SQL Server использует оператор **Key Lookup** (Уточняющий запрос ключа) для получения данных из базовой кластеризованной таблицы после поиска по некластеризованному индексу.
3. Правильный ответ: C.
- A. **Неправильно:** в физической последовательности этот просмотр называется просмотром в порядке выделения.
- B. **Неправильно:** **Clustered Index Scan** (Просмотр кластеризованного индекса) — это оператор, который может просматривать данные в физической или логической последовательности.
- C. **Правильно:** когда SQL Server просматривает данные в логическом порядке индекса, такой просмотр называется просмотром в порядке индекса.
- D. **Неправильно:** оператор **Seek** (Поиск) не используется для просмотра.

## Занятие 2

### Закрепление материала

1. Правильные ответы: A, C и D.
- A. **Правильно:** изменение параметра `SET` может быть причиной того, что SQL Server не использует повторно существующий кэшированный план.
- B. **Неправильно:** SQL Server повторно использует кэшированные планы хранимой процедуры, если вы не запускаете повторную компиляцию или используете в процедуре динамический SQL.
- C. **Правильно:** SQL Server не использует повторно кэшированный план, если он не может определить применяемость параметризованного предиката.
- D. **Правильно:** если в параметризованном предикате используется другой тип данных для параметра, SQL Server повторно не использует кэшированный план.

2. Правильный ответ: С.

- A. **Неправильно:** обработка в пакетном режиме не уменьшает дисковый ввод-вывод.
- B. **Неправильно:** обработка в пакетном режиме не оказывает влияния на сеть.
- C. **Правильно:** снижение использования ЦП — главное преимущество обработки в пакетном режиме.
- D. **Неправильно:** использование памяти не связано напрямую с пакетной обработкой.

3. Правильный ответ: С.

- A. **Неправильно:** команда SET STATISTICS IO только включает и отключает статистическую информацию о дисковом вводе-выводе.
- B. **Неправильно:** когда параметр STATISTICS PROFILE установлен в ON, каждый выполненный запрос возвращает свой обычный результирующий набор, стоящий перед дополнительным результирующим набором, который показывает профиль выполнения запроса; он не влияет на повторную используемость плана выполнения.
- C. **Правильно:** команда SET CONCAT\_NULL\_YIELDS\_NULL влияет на результат запроса и, следовательно, ее изменение не позволяет повторно использовать план.
- D. **Неправильно:** команда SET STATISTICS IO только включает и отключает статистическую информацию о дисковом вводе-выводе.

## Занятие 3

### Закрепление материала

1. Правильные ответы: А, В и D.

- A. **Правильно:** SQL Server поддерживает подсказки запросов, таблиц и соединений.
- B. **Правильно:** SQL Server поддерживает подсказки запросов, таблиц и соединений.
- C. **Неправильно:** не существует подсказки очереди.
- D. **Правильно:** SQL Server поддерживает подсказки запросов, таблиц и соединений.

2. Правильный ответ: А.

- A. **Правильно:** не существует типа структуры плана JOIN.
- B. **Неправильно:** SQL Server поддерживает структуру плана TEMPLATE.
- C. **Неправильно:** SQL Server поддерживает структуру плана SQL.
- D. **Неправильно:** SQL Server поддерживает структуру плана OBJECT.

3. Правильный ответ: D.

- A. **Неправильно:** подсказка запроса OPTION (ORDER GROUP) не влияет на соединения.
- B. **Неправильно:** подсказка запроса OPTION (ORDER GROUP) инициирует статистическую обработку потока.
- C. **Неправильно:** подсказка запроса OPTION (ORDER GROUP) не влияет на последовательность или параллельность плана выполнения.
- D. **Правильно:** подсказка запроса OPTION (ORDER GROUP) инициирует статистическую обработку потока.

## Упражнения

### Упражнение 1. Оптимизация запроса

- 1. Можно проверить, являются ли индексы наиболее подходящими для запросов. Также вы можете проверить, обновлены ли статистические данные для индексов. Кроме того, вы можете создать структуры планов для проблематичных запросов.
- 2. Нет, вы не можете использовать подсказки оптимизатора, поскольку не можете изменить текст проблематичных запросов.

### Упражнение 2. Табличная подсказка

- 1. Да. Поскольку запрос находится внутри хранимой процедуры и не встроен в приложение, вы можете изменить его.
- 2. Вы можете использовать табличную подсказку, чтобы инициировать использование индекса.

# Предметный указатель

## A

aggregation element 164  
AUTO\_CREATE\_STATISTICS 562  
AUTO\_UPDATE\_STATISTICS 562  
AUTO\_UPDATE\_STATISTICS\_ASYNC 562

## C

CASE, выражение 56  
Common table expression (CTE)  
*См. Обобщенное табличное выражение*

## D

Data definition language (DDL) 246  
Data manipulation language (DML)  
◊ компонуемый 384  
Document Object Model (DOM) 226  
Dynamic management objects (DMO) 517

## F

Full-Text Search, компонент 190

## G

Grouping element 164

## I

iFilters 191  
Index Allocation Map (IAM) 528, 600

## N

NULL 17, 68, 71, 267

## Q

QName 232

## R

RANGE 178

## S

Spreading element 164  
SQL:  
◊ динамический 433  
◊ история названия 22  
◊ код, внедрение 438  
SQL Server Extended Events 500  
SQL Server Profiler 501

## T

TRY/CATCH 423  
T-SQL 9, 21

## U

UNPIVOT 169

## X

XML Schema Description (XSD) 219  
XML-документ 216, 246  
XQuery 231

**A**

- Автофиксация 397
- Алгоритм:
  - ◊ вложенных циклов 607
  - ◊ статистической обработки:
    - потока 611
    - хэша 611
- Аргумент поиска 67, 70, 555
- Атомарность 394
- Атрибут XML-кода 218

**Б**

- База данных tempdb 586
- Блокирование 405
- Блокировка:
  - ◊ кратковременная 49
  - ◊ монопольная 404
- Блокировки совмещаемые 403

**В**

- Взаимоблокировка 405
- Выражение:
  - ◊ взвешенное 190
  - ◊ префиксное 190
  - ◊ производное 190
  - ◊ простое 190
  - ◊ с учетом расположения слов или фраз 190
  - ◊ табличное 123, 129
  - ◊ тезауруса 190

**Г**

- Гистограмма 590

**Д**

- Данные:
  - ◊ вставка 316
  - ◊ обновление с помощью табличных выражений 335
  - ◊ символьные 46
  - ◊ сортировка 80
  - ◊ удаление 344, 345
- Дата и время, форматирование строк 56
- Дерево:
  - ◊ алгебраическое 497
  - ◊ сбалансированное 527, 532
  - ◊ синтаксического разбора 497
- Дескриптор документа 226
- Дробление 226

**З**

- Запись 18
- Запрос:
  - ◊ RID, уточняющий 540
  - ◊ вложенный 119
    - скалярный 120
  - ◊ групповой 160
  - ◊ детерминированный 94
  - ◊ именованный 123
  - ◊ логическая обработка 21, 22, 106
  - ◊ оптимизация 506
  - ◊ покрытие 601
  - ◊ с несколькими соединениями 114
- Запросы эквивалентные 498

**И**

- Идентификатор:
  - ◊ регулярный 265
  - ◊ с разделителем 265
  - ◊ свойство 268
- Изоляция 394
- Индекс:
  - ◊ columnstore 544
  - ◊ XML 253
  - ◊ кластеризованный 533
  - ◊ некластеризованный 540, 601
  - ◊ полнотекстовый 190, 193
- Инструкция:
  - ◊ CREATE TABLE 261
  - ◊ DELETE 344
    - на основе объединений 346
    - с табличными выражениями 347
  - ◊ FLWOR 239
  - ◊ INSERT EXEC 320
  - ◊ INSERT SELECT 319
  - ◊ INSERT VALUES 317
  - ◊ MERGE 367, 373
  - ◊ SELECT 22, 26, 293
  - ◊ SELECT INTO 322
  - ◊ TRUNCATE 345
  - ◊ UPDATE 328–330
    - и принцип единовременности 339
    - недетерминированная 332
- Итератор 600

**К**

- Кадрирование 175
- Карта распределения индекса 600
- Каталог, полнотекстовый 193

**Ключ:**

- ◊ бизнес-ключ 276
- ◊ внешний 278
- ◊ естественный 276
- ◊ кластеризации 532
- ◊ непоследовательный 50
- ◊ первичный 277
- ◊ последовательный 49
- ◊ суррогатный 48, 276
  - генерация 48

Конструкция OFFSET...FETCH 94

Код возврата 454

**Команда:**

- ◊ ALTER TABLE 270
- ◊ EXECUTE 437
- ◊ THROW 420

Курсор 15, 28, 575, 579

Куча 527

**Л**

Литерал 46, 73, 75

**Логика:**

- ◊ ветвления 456
- ◊ двоичная 68
- ◊ троичная 69

**М**

Метаданные 219

Методы типа данных XML 248

Множество 11, 12

Модель реляционная 11, 12, 17

**Модуль:**

- ◊ парадигматический 191
- ◊ реляционный 497

Мультимножество 14

Мультисоединение 114

**Н**

Набор группирования 150, 155, 157

Навигация по XML-документу 236

Номер ошибки 418

**О**

Область поиска 498

Обобщенное табличное выражение 127

**Обработка:**

- ◊ в пакетном режиме 626
- ◊ пакетная 617

Объединение символьных строк 53

**Объект:**

- ◊ динамический административный 517
  - категории 518
  - связанный с выполнением 519
  - связанный с индексом 519
- ◊ последовательности 353, 362

**Ограничение:**

- ◊ внешнего ключа 279
- ◊ по умолчанию 282
- ◊ проверочное 281
- ◊ уникальности 277

**Оператор:**

- ◊ APPLY 131
- ◊ CROSS APPLY 131
- ◊ EXCEPT 142
- ◊ INTERSECT 142
- ◊ OUTER APPLY 133
- ◊ PIVOT 166
- ◊ UNPIVOT 167
- ◊ работы с наборами 139, 140

Оптимизатор запросов 497, 498

Отмена сведения данных 167, 168

**Отношение 11**

- ◊ заголовок 11, 13, 15

Оценка мощности 498

**П****Параметр:**

- ◊ PERCENT 91
- ◊ кэширования 361

Переменная, табличная 260, 585

**План:**

- ◊ выполнения 510
  - операторы 513
  - предполагаемый 510
  - форматы 510
- ◊ кэшированный 498
- ◊ простой 592
- ◊ структура 631, 636

Подзапрос 119

- ◊ коррелированный 121
- ◊ независимый 120
- ◊ связанный 121

Подсистема расширенных событий

SQL Server 500

Подсказка 599

- ◊ оптимизатора 631
- ◊ табличная 633

**Поиск:**

- ◊ ключа 540
- ◊ полнотекстовый 190
- ◊ статистический семантический 194

Поле 18

Последовательность 358  
Предикат 12, 17, 67, 68, 70, 79, 107, 108  
◊ CONTAINS 201  
◊ FREETEXT 202  
◊ LIKE 74  
◊ в XML 238  
◊ объединение 71  
Предикаты, комбинирование 72  
Предложение:  
◊ DISTINCT 14, 27  
◊ FOR XML 220  
◊ FROM 23, 35  
◊ GROUP BY 25  
◊ HAVING 26  
◊ ORDER BY 15, 19, 27, 82, 85  
◊ OUTPUT 379  
◊ SELECT 36  
◊ TOP 90  
◊ WHERE 24, 67  
◊ WITH CHECK OPTION 293  
◊ ROWS 178  
Представление 129, 290  
◊ атрибутивное 218  
◊ имя 293  
◊ индексированное 260, 294, 545  
◊ ограничение 294  
◊ параметры 292  
◊ секционированное 296  
◊ создание 290  
◊ удаление 295  
◊ элементное 218  
Пространство имен 218  
Псевдоним 16  
◊ выражения 38  
◊ столбца 108

**P**

Разбор синтаксический 497  
Разделение идентификаторов 39  
Ранг документа 207  
Режим:  
◊ FOR XML AUTO 221  
◊ FOR XML EXPLICIT 224  
◊ FOR XML PATH 224  
◊ FOR XML RAW 221  
Решение:  
◊ итерационное 574, 577  
◊ на основе наборов 573, 577

**C**

Самосоединение 106  
Свойства транзакций 394

Свойство IDENTITY 354  
Сегмент 545  
Сжатие данных в таблице 269  
Синоним 306  
◊ создание 306  
Слияние данных 367  
Сниффинг параметров 499  
Согласованность 394  
Соединение:  
◊ внешнее 110  
▫ левое 110  
▫ полное 113  
▫ правое 113  
◊ внутреннее 106  
◊ "звезда" 622  
◊ перекрестное 104  
◊ слиянием 608  
Сообщение об ошибке 418  
Сортировка 90  
◊ данных 80  
◊ детерминированная 83  
Состояние 418  
Список двунаправленный 528  
Средство разбиения по словам 191  
Статистика:  
◊ автоматическое создание 562  
◊ ввода-вывода 508  
◊ по времени 509  
◊ распространения 590  
◊ ручное создание 566  
Столбец 18  
◊ вычисляемый 268  
Стоп-слово 192  
Стоп-список 192  
Страница 507  
Строка 18  
◊ удаление 344, 345  
Структура плана 631, 636  
Схема:  
◊ XSD 223  
◊ базы данных 263  
◊ встроенная 223  
◊ динамическая 249

**T**

Таблица:  
◊ базовая 260  
◊ временная 260, 584–586  
▫ глобальная 585  
▫ локальная 585  
◊ производная 124  
◊ создание 261  
◊ уточняющих запросов 279

Тег, открывающий 216

Тезаурус 192

Тест узла 237

Тип данных 42, 44, 47

◊ XML 247

◊ XQuery 234

◊ для ключей 48

◊ столбцов 266

Тип ограничения 275

Типы:

◊ атомарные 234

◊ переменных 45

◊ узлов 234

Транзакция 393

◊ вложенная 400

◊ локальная 403

◊ межбазовая 403

◊ откат 393

◊ пользовательская 396

◊ распределенная 403

◊ режим автоматической фиксации 397

◊ режим неявных транзакций 398

◊ системная 396

◊ точки сохранения 402

◊ уровень 396

▫ изоляции 408

◊ фиксация 393

◊ явная 399

Трассировка SQL 501

Триггер 471

◊ AFTER 472

◊ DML 471

◊ INSTEAD OF 475

◊ вложенный 475

## У

Указание:

◊ соединения 634

◊ запроса 631

Указатель строк 540

Уровень серьезности 418

Устойчивость 394

## Ф

Фильтр 70

Фильтрация 73, 90

◊ битовая 607

◊ даты и времени 75

◊ символьных данных 73

◊ строк 26

Фрагментация:

◊ внешняя 533

◊ внутренняя 530, 539

Функция:

◊ \$action 383

◊ OPENXML 226

◊ встроенная 298

▫ табличная 129

◊ групповая агрегатная 174

◊ даты и времени 50

◊ оконная 173

▫ смещения 180

▫ ранжирующая 178

▫ статистическая 174

◊ пользовательская 482

▫ встроенная с табличным значением 484

▫ с табличным значением 484

▫ скалярная 482

◊ с табличным значением 482

◊ семантического поиска 209

◊ символьных типов данных 53

◊ системная @@ERROR 422

◊ скалярная 482

◊ триггера 476

◊ языка XQuery 234

## Х

Хранимая процедура 449

◊ sp\_configure 454, 473

◊ sp\_executesql 440

◊ параметры 452

Хэш-соединение 607

◊ с битовой фильтрацией 609

## Ч

Чтение фантомное 409

## Э

Экстент 507, 527

Элемент:

◊ агрегатный 164

◊ группирующий 164

◊ закрепленный 128

◊ распределяющий 164

◊ рекурсивный 128

# Об авторах



**Ицик Бен-Ган (Itzik Ben-Gan)** — один из создателей и руководителей компании SolidQ. Имея статус Microsoft SQL Server MVP с 1999 г., Ицик провел по всему миру огромное число тренингов, посвященных созданию запросов, их настройке и программированию на языке T-SQL. Ицик также является автором нескольких книг по T-SQL. Кроме статей и информационно-технических документов для MSDN и "The SolidQ Journal", он написал ряд статей по SQL Server Pro. Ицик выступает с лекциями на конференциях

Tech-Ed, SQL PASS (Professional Association for SQL Server), SQL Server Connections, проводит презентации для разных групп пользователей SQL Server и для различных событий, организуемых SolidQ. Ицик является экспертом в SolidQ по всем видам деятельности, связанным с T-SQL. Он организовал для SolidQ курсы "Advanced T-SQL" (Расширенный курс T-SQL) и "T-SQL Fundamentals" (Основы T-SQL) и регулярно проводит их по всему миру.



**Деян Сарка (Dejan Sarka)**, имея статус MCT и SQL Server MVP, специализируется на разработке баз данных и приложениях для бизнес-анализа. Помимо работы над проектами, он тратит большую часть времени на проведение тренингов и консультаций. Деян является основателем компании Slovenian SQL Server and .NET Users Group. Он — автор и соавтор 11 книг о базах данных и SQL Server. Кроме того, он разработал два курса и множество семинаров для компаний Microsoft и компании SolidQ.



**Рон Талмейдж (Ron Talmage)** живет в Сиэтле и является консультантом в области баз данных в компании SolidQ. Он — инструктор и один из основателей этой компании, имеет звание SQL Server MVP, является региональным инструктором PASS и главой сообщества "Seattle SQL Server User Group". Он работает в области SQL Server, с тех самых пор, как появилась версия SQL Server 4.21a, и написал множество статей и информационно-технических документов по этой тематике.