ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ОРЕНБУРГСКИЙ ГОСУДАРСТВЕННЫЙ АГРАРНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ОБУЧАЮЩИХСЯ ПО ОСВОЕНИЮ ДИСЦИПЛИНЫ

Б1.В.20 SQL-программирование

Направление подготовки 09.03.01 Информатика и вычислительная техника

Профиль образовательной программы *автоматизированные системы обработки информации и управления*

Форма обучения *очная*

СОДЕРЖАНИЕ

1. КОНСПЕКТ ЛЕКЦИЙ

1.1. Лекция № 1 (2 часа)

Тема: «Определение структурированного языка запросов SQL»

1.1.1. Вопросы лекции:

1. Реляционная база данных, СУБД.
2. Классификация команд SQL.

1.1.2. Краткое содержание вопросов:

1. Реляционная база данных, СУБД.

Управление основными потоками информации осуществляется с помощью так называемых систем управления реляционными базами данных, которые берут свое начало в традиционных системах управления базами данных. Именно объединение реляционных баз данных и клиент-серверных технологий позволяет современному предприятию успешно управлять собственными данными, оставаясь конкурентоспособным на рынке товаров и услуг.

Реляционные БД имеют мощный теоретический фундамент, основанный на математической теории отношений. Появление теории реляционных баз данных дало толчок к разработке ряда языков запросов, которые можно отнести к двум классам:

- алгебраические языки, позволяющие выражать запросы средствами специализированных операторов, применяемых к отношениям;
- языки исчисления предикатов, представляющие собой набор правил для записи выражения, определяющего новое отношение из заданной совокупности существующих отношений.

СУБД (система управления базами данных) – программное обеспечение, с помощью которого пользователи могут определять, создавать и поддерживать базу данных, а также получать к ней контролируемый доступ.

2. Классификация команд SQL.

Основные категории команд языка SQL:

• DDL – язык определения данных;
• DML – язык манипулирования данными;
• DQL – язык запросов;
• DCL – язык управления данными;
• команды администрирования данных;
• команды управления транзакциями

Определение структур базы данных (DDL)

Язык определения данных (DataDefinitionLanguage, DDL) позволяет создавать и изменять структуру объектов базы данных, например, создавать и удалять таблицы.

Основными командами языка DDL являются следующие: CREATE TABLE, ALTER TABLE, DROP TABLE, CREATE INDEX, ALTER INDEX, DROP INDEX.

Манипулирование данными (DML)

Язык манипулирования данными (DataManipulationLanguage, DML) используется для манипулирования информацией внутри объектов реляционной базы данных посредством трех основных команд: INSERT, UPDATE, DELETE.

Выборка данных (DQL) Язык запросов DQL наиболее известен пользователям реляционной базы данных, несмотря на то, что он включает всего одну команду SELECT. Эта команда вместе со своими многочисленными опциями и предложениями используется для формирования запросов к реляционной базе данных.

Язык управления данными (DCL - DataControlLanguage)

Команды управления данными позволяют управлять доступом к информации, находящейся внутри базы данных. Как правило, они используются для создания объектов,

связанных с доступом к данным, а также служат для контроля над распределением привилегий между пользователями. Команды управления данными следующие: GRANT, REVOKE.

Команды администрирования данных

С помощью команд администрирования данных пользователь осуществляет контроль за выполняемыми действиями и анализирует операции базы данных; они также могут оказаться полезными при анализе производительности системы.

Команды управления транзакциями

Существуют следующие команды, позволяющие управлять транзакциями базы данных: COMMIT, ROLLBACK, SAVEPOINT, SET TRANSACTION.

1.2. Лекция № 2 (2 часа)

Тема: «Эффективное выполнение запросов для извлечения данных».

1.2.1. Вопросы лекции:

1. Синтаксис оператора SELECT.
2. Построение условий выбора данных с применением оператора сравнения, логических операторов и логических связок.

1.2.2. Краткое содержание вопросов:

1. Синтаксис оператора SELECT.

Оператор SELECT – один из наиболее важных и самых распространенных операторов SQL. Он позволяет производить выборки данных из таблиц и преобразовывать к нужному виду полученные результаты. Будучи очень мощным, он способен выполнять действия, эквивалентные операторам реляционной алгебры, причем в пределах единственной выполняемой команды. При его помощи можно реализовать сложные и громоздкие условия отбора данных из различных таблиц.

Оператор SELECT – средство, которое полностью абстрагировано от вопросов представления данных, что помогает сконцентрировать внимание на проблемах доступа к данным.

Примеры его использования наглядно демонстрируют один из основополагающих принципов больших (промышленных) СУБД: средства хранения данных и доступа к ним отделены от средств представления данных. Операции над данными производятся в масштабе наборов данных, а не отдельных записей.

Оператор SELECT имеет следующий формат:

SELECT [ALL | DISTINCT] {*|[имя_столбца
[AS новое_имя]]} [,...n]
FROM имя_таблицы [[AS] псевдоним] [,...n]
[WHERE <условие_поиска>]
[GROUPBYимя_столбца [,...n]]
[HAVING <критерии выбора групп>]
[ORDER BY имя_столбца [,...n]]

Оператор SELECT определяет поля (столбцы), которые будут входить в результат выполнения запроса. В списке они разделяются запятыми и приводятся в такой очередности, в какой должны быть представлены в результате запроса. Если используется имя поля, содержащее пробелы или разделители, его следует заключить в квадратные скобки. Символом * можно выбрать все поля, а вместо имени поля применить выражение из нескольких имен. Если обрабатывается ряд таблиц, то (при наличии одноименных полей в разных таблицах) в списке полей используется полная спецификация поля, т.е. Имя_таблицы.Имя_поля.

2. Построение условий выбора данных с применением оператора сравнения, логических операторов и логических связок.

Сравнение

В языке SQL можно использовать следующие операторы сравнения: = – равенство; < –меньше; > – больше; <= – меньше или равно; >= – больше или равно; <> – не равно.

Пример 4.3. Показать все операции отпуска товаров объемом больше 20.

```
SELECT * FROM Сделка
WHERE Количество>20
```

Пример 4.3. Операции отпуска товаров объемом больше 20.

Более сложные предикаты могут быть построены с помощью логических операторов AND, OR или NOT, а также скобок, используемых для определения порядка вычисления выражения.

Вычисление выражения в условиях выполняется по следующим правилам:

- Выражение вычисляется слева направо.
- Первыми вычисляются подвыражения в скобках.
- Операторы NOT выполняются до выполнения операторов AND и OR.
- Операторы AND выполняются до выполнения операторов OR.

Для устранения любой возможной неоднозначности рекомендуется использовать скобки.

Пример 4.4. Вывести список товаров, цена которых больше или равна 100 и меньше или равна 150.

```
SELECT Название, Цена
FROM Товар
WHERE Цена>=100 And Цена<=150
```

Пример 4.4. Список товаров, цена которых больше или равна 100 и меньше или равна 150.

Пример 4.5. Вывести список клиентов из Москвы или из Самары.

```
SELECT Фамилия, ГородКлиента
FROM Клиент
WHERE ГородКлиента="Москва" Or
                ГородКлиента="Самара"
```

Пример 4.5. Список клиентов из Москвы или из Самары.

Диапазон

Оператор BETWEEN используется для поиска значения внутри некоторого интервала, определяемого своими минимальным и максимальным значениями. При этом указанные значения включаются в условие поиска.

Пример 4.6. Вывести список товаров, цена которых лежит в диапазоне от 100 до 150 (запрос эквивалентен примеру 4.4).

```
SELECT Название, Цена
FROM Товар
WHERE Цена Between 100 And 150
```

Пример 4.6. Список товаров, цена которых лежит в диапазоне от 100 до 150.

При использовании отрицания NOT BETWEEN требуется, чтобы проверяемое значение лежало вне границ заданного диапазона.

Пример 4.7. Вывести список товаров, цена которых не лежит в диапазоне от 100 до 150.

```
SELECT Товар.Название, Товар.Цена
FROM Товар
WHERE Товар.Цена Not Between 100 And 150
Или (что эквивалентно)
SELECT Товар.Название, Товар.Цена
```

FROM Товар
WHERE (Товар.Цена<100) OR (Товар.Цена>150)

1.3. Лекция № 3 (2 часа)

Тема: «Построение нетривиальных запросов».

1.3.1. Вопросы лекции:

1. Построение нетривиальных запросов.
2. Способ построения подзапросов, возвращающих множественные и единичные значения с использованием операторов EXISTS, ALL, ANY.

1.3.2. Краткое содержание вопросов:

1. Построение нетривиальных запросов.

Понятие подзапроса

Часто невозможно решить поставленную задачу путем одного запроса. Это особенно актуально, когда при использовании условия поиска в предложении WHERE значение, с которым надо сравнивать, заранее не определено и должно быть вычислено в момент выполнения оператора SELECT. В таком случае приходят на помощь законченные операторы SELECT, внедренные в тело другого оператора SELECT. Внутренний подзапрос представляет собой также оператор SELECT, а кодирование его предложений подчиняется тем же правилам, что и основного оператора SELECT.

Внешний оператор SELECT использует результат выполнения внутреннего оператора для определения содержания окончательного результата всей операции. Внутренние запросы могут быть помещены непосредственно после оператора сравнения (=, <, >, <=, >=, <>) в предложения WHERE и HAVING внешнего оператора SELECT – они получают название подзапросов или вложенных запросов. Кроме того, внутренние операторы SELECT могут применяться в операторах INSERT, UPDATE и DELETE.

Подзапрос – это инструмент создания временной таблицы, содержимое которой извлекается и обрабатывается внешним оператором. Текст подзапроса должен быть заключен в скобки.

К подзапросам применяются следующие правила и ограничения:

- фраза ORDER BY не используется, хотя и может присутствовать во внешнем подзапросе;
- список в предложении SELECT состоит из имен отдельных столбцов или составленных из них выражений – за исключением случая, когда в подзапросе присутствует ключевое слово EXISTS;
- по умолчанию имена столбцов в подзапросе относятся к таблице, имя которой указано в предложении FROM. Однако допускается ссылка и на столбцы таблицы, указанной во фразе FROM внешнего запроса, для чего применяются квалифицированные имена столбцов (т.е. с указанием таблицы);
- если подзапрос является одним из двух операндов, участвующих в операции сравнения, то запрос должен указываться в правой части этой операции.

Существует два типа подзапросов:

- Скалярный подзапрос возвращает единственное значение. В принципе, он может использоваться везде, где требуется указать единственное значение.
- Табличный подзапрос возвращает множество значений, т.е. значения одного или нескольких столбцов таблицы, размещенные в более чем одной строке. Он возможен везде, где допускается наличие таблицы.

2. Способ построения подзапросов, возвращающих множественные и единичные значения с использованием операторов EXISTS, ALL, ANY.

Использование ключевых слов ANY и ALL

Ключевые слова ANY и ALL могут использоваться с подзапросами, возвращающими один столбец чисел.

Если подзапросу будет предшествовать ключевое слово ALL, условие сравнения считается выполненным, только когда оно выполняется для всех значений в результирующем столбце подзапроса.

Если записи подзапроса предшествует ключевое слово ANY, то условие сравнения считается выполненным, когда оно выполняется хотя бы для одного из значений в результирующем столбце подзапроса.

Если в результате выполнения подзапроса получено пустое значение, то для ключевого слова ALL условие сравнения будет считаться выполненным, а для ключевого слова ANY – невыполненным. Ключевое слово SOME является синонимом слова ANY.

Пример 7.14. Определить клиентов, совершивших сделки с максимальным количеством товара (эквивалентно запросу 7.3.)

```
SELECT Клиент.Фамилия, Сделка.Количество
FROM Клиент INNER JOIN Сделка
ON Клиент.КодКлиента=Сделка.КодКлиента
WHERE Сделка.Количество>=ALL(SELECT Количество
FROM Сделка)
```

В примере определены клиенты, в сделках которых количество товара больше или равно количеству товара в каждой из всех сделок.

Использование операций EXISTS и NOT EXISTS

Ключевые слова EXISTS и NOT EXISTS предназначены для использования только совместно с подзапросами. Результат их обработки представляет собой логическое значение TRUE или FALSE. Для ключевого слова EXISTS результат равен TRUE в том и только в том случае, если в возвращаемой подзапросом результирующей таблице присутствует хотя бы одна строка. Если результирующая таблица подзапроса пуста, результатом обработки операции EXISTS будет значение FALSE. Для ключевого слова NOT EXISTS используются правила обработки, обратные по отношению к ключевому слову EXISTS. Поскольку по ключевым словам EXISTS и NOT EXISTS проверяется лишь наличие строк в результирующей таблице подзапроса, то эта таблица может содержать произвольное количество столбцов.

Пример 7.18. Определить список имеющихся на складе товаров (запрос эквивалентен примеру 7.7).

```
SELECT Название
FROM Товар
WHERE EXISTS (SELECT КодТовара
FROM Склад
WHERE Товар.КодТовара=Склад.КодТовара)
```

1.4. Лекция № 4 (2 часа)

Тема: «Запросы модификации данных в реляционной таблице».

1.4.1. Вопросы лекции:

1. Целостность данных.
2. Целостность сущностей и ссылочная целостность**.**

1.4.2. Краткое содержание вопросов:

1. Целостность данных.

Введение в понятие "целостность данных"

Выполнение операторов модификации данных в таблицах базы данных INSERT, DELETE и UPDATE может привести к нарушению целостности данных и их корректности, т.е. к потере их достоверности и непротиворечивости.

Чтобы информация, хранящаяся в базе данных, была однозначной и непротиворечивой, в реляционной модели устанавливаются некоторые ограничительные условия – правила, определяющие возможные значения данных и обеспечивающие логическую основу для поддержания корректных значений.

Ограничения целостности позволяют свести к минимуму ошибки, возникающие при обновлении и обработке данных.

В базе данных, построенной на реляционной модели, задается ряд правил целостности, которые, по сути, являются ограничениями для всех допустимых состояний базы данных и гарантируют корректность данных.

Рассмотрим следующие типы ограничений целостности данных:

- обязательные данные;
- ограничения для доменов полей;
- корпоративные ограничения;
- целостность сущностей;
- ссылочная целостность.

2. Целостность сущностей и ссылочная целостность.

Целостность сущностей

Это ограничение целостности касается первичных ключей базовых таблиц. По определению, первичный ключ – минимальный идентификатор (одно или несколько полей), который используется для уникальной идентификации записей в таблице.

Таким образом, никакое подмножество первичного ключа не может быть достаточным для уникальной идентификации записей.

Целостность сущностей определяет, что в базовой таблице ни одно поле первичного ключа не может содержать отсутствующих значений, обозначенных NULL.

Если допустить присутствие определителя NULL в любой части первичного ключа, это равносильно утверждению, что не все его поля необходимы для уникальной идентификации записей, и противоречит определению первичного ключа.

Ссылочная целостность

Указанное ограничение целостности касается внешних ключей. Внешний ключ – это поле (или множество полей) одной таблицы, являющееся ключом другой (или той же самой) таблицы. Внешние ключи используются для установления логических связей между таблицами. Связь устанавливается путем присвоения значений внешнего ключа одной таблицы значениям ключа другой.

Между двумя или более таблицами базы данных могут существовать отношения подчиненности, которые определяют, что для каждой записи главной таблицы (называемой еще родительской) может существовать одна или несколько записей в подчиненной таблице (называемой также дочерней).

Существует три разновидности связи между таблицами базы данных:

- "один-ко-многим";
- "один-к-одному";
- "многие-ко-многим".

Отношение "один-ко-многим" имеет место, когда одной записи родительской таблицы может соответствовать несколько записей дочерней. Связь "один-ко-многим" иногда называют связью "многие-к-одному". И в том, и в другом случае сущность связи между таблицами остается неизменной. Связь "один-ко-многим" наиболее распространена для реляционных баз данных. Она позволяет моделировать также иерархические структуры данных.

Отношение "один-к-одному" имеет место, когда одной записи в родительской таблице соответствует одна запись в дочерней. Это отношение встречается намного реже, чем отношение "один-ко-многим". Его используют, если не хотят, чтобы таблица БД "распухала" от второстепенной информации. Использование связи "один-к-одному"

приводит к тому, что для чтения связанной информации в нескольких таблицах приходится производить несколько операций чтения вместо одной, когда данные хранятся в одной таблице.

Отношение "многие-ко-многим" имеет место в следующих случаях:

 одной записи в родительской таблице соответствует более одной записи в дочерней таблице;

 одной записи в дочерней таблице соответствует более одной записи в родительской таблице.

Считается, что всякая связь "многие-ко-многим" может быть заменена на связь "один-ко-многим" (одну или несколько).

Часто связь между таблицами устанавливается по первичному ключу, т.е. значениям внешнего ключа одной таблицы присваиваются значения первичного ключа другой. Однако это не является обязательным – в общем случае связь может устанавливаться и с помощью вторичных ключей. Кроме того, при установлении связей между таблицами не требуется непременная уникальность ключа, обеспечивающего установление связи.

Поля внешнего ключа не обязаны иметь те же имена, что и имена ключей, которым они соответствуют. Внешний ключ может ссылаться на свою собственную таблицу – в таком случае внешний ключ называется рекурсивным.

Ссылочная целостность определяет: если в таблице существует внешний ключ, то его значение должно либо соответствовать значению первичного ключа некоторой записи в базовой таблице, либо задаваться определителем NULL.

1.5. Лекция № 5 (2 часа)

Тема: «Понятие представлений».

1.5.1. Вопросы лекции:

1. Роль представлений в вопросах безопасности данных.
2. Процесс управления представлениями: создание, изменение, применение, удаление представлений**.**

1.5.2. Краткое содержание вопросов:

1. Определение представлений.

Представления, или просмотры (VIEW), представляют собой временные, производные (иначе - виртуальные) таблицы и являются объектами базы данных, информация в которых не хранится постоянно, как в базовых таблицах, а формируется динамически при обращении к ним. Обычные таблицы относятся к **базовым,** т.е. содержащим данные и постоянно находящимся на устройстве хранения информации. Представление не может существовать само по себе, а определяется только в терминах одной или нескольких таблиц. Применение представлений позволяет разработчику базы данных обеспечить каждому пользователю или группе пользователей наиболее подходящие способы работы с данными, что решает проблему простоты их использования и безопасности. Содержимое представлений выбирается из других таблиц с помощью выполнения запроса, причем при изменении значений в таблицах данные в представлении автоматически меняются. Представление - это фактически тот же запрос, который выполняется всякий раз при участии в какой-либо команде. Результат выполнения этого запроса в каждый момент времени становится содержанием представления. У пользователя создается впечатление, что он работает с настоящей, реально существующей таблицей.

Преимущества и недостатки представлений

Механизм представления - мощное средство СУБД, позволяющее скрыть реальную структуру БД от некоторых пользователей за счет определения представлений. Любая

реализация представления должна гарантировать, что состояние представляемого отношения точно соответствует состоянию данных, на которых определено это представление. Обычно вычисление представления производится каждый раз при его использовании. Когда представление создается, информация о нем записывается в каталог БД под собственным именем. Любые изменения в данных адекватно отобразятся в представлении - в этом его отличие от очень похожего на него запроса к БД. В то же время запрос представляет собой как бы <мгновенную фотографию> данных и при изменении последних запрос к БД необходимо повторить. Наличие представлений в БД необходимо для обеспечения логической независимости данных.

Если система обеспечивает физическую независимость данных, то изменения в физической структуре БД не влияют на работу пользовательских программ. Логическая независимость подразумевает тот факт, что при изменении логической структуры данных влияние на пользовательские программы также не оказывается, а значит, система должна уметь решать проблемы, связанные с ростом и реструктуризацией БД. Очевидно, что с увеличением количества данных, хранимых в БД, возникает необходимость ее расширения за счет добавления новых атрибутов или отношений - это называется ростом БД. Реструктуризация данных подразумевает сохранение той же самой информации, но изменяется ее расположение, например, за счет перегруппировки атрибутов в отношениях. Предположим, некоторое отношение в силу каких-либо причин необходимо разделить на два. Соединение полученных отношений в представлении воссоздает исходное отношение, а у пользователя складывается впечатление, что никакой реструктуризации не производилось. Помимо решения проблемы реструктуризации представление можно применять для просмотра одних и тех же данных разными пользователями и в различных вариантах. С помощью представлений пользователь имеет возможность ограничить объем данных для удобства работы. Наконец, механизм представлений позволяет скрыть служебные данные, не интересные пользователям.

В случае выполнения СУБД на отдельно стоящем персональном компьютере использование представлений обычно имеет целью лишь упрощение структуры запросов к базе данных. Однако в случае многопользовательской сетевой СУБД представления играют ключевую роль в определении структуры базы данных и организации защиты информации. Рассмотрим основные преимущества применения представлений в подобной среде.

Независимость от данных

С помощью представлений можно создать согласованную, неизменную картину структуры базы данных, которая будет оставаться стабильной даже в случае изменения формата исходных таблиц (например, добавления или удаления столбцов, изменения связей, разделения таблиц, их реструктуризации или переименования).

Если в таблицу добавляются или из нее удаляются не используемые в представлении столбцы, то изменять определение этого представления не потребуется.

Если структура исходной таблицы переупорядочивается или таблица разделяется, можно создать представление, позволяющее работать с виртуальной таблицей прежнего формата. В случае разделения исходной таблицы, прежний формат может быть виртуально воссоздан с помощью представления, построенного на основе соединения вновь созданных таблиц - конечно, если это окажется возможным. Последнее условие можно обеспечить с помощью помещения во все вновь созданные таблицы первичного ключа прежней таблицы.

2. Процесс управления представлениями: создание, изменение, применение, удаление представлений.

Актуальность

Изменения данных в любой из таблиц базы данных, указанных в определяющем запросе, немедленно отображается на содержимом представления.

Повышение защищенности данных

Права доступа к данным могут быть предоставлены исключительно через ограниченный набор представлений, содержащих только то подмножество данных, которое необходимо пользователю. Подобный подход позволяет существенно ужесточить контроль за доступом отдельных категорий пользователей к информации в базе данных.

Снижение стоимости

Представления позволяют упростить структуру запросов за счет объединения данных из нескольких таблиц в единственную виртуальную таблицу. В результате многотабличные запросы сводятся к простым запросам к одному представлению.

Дополнительные удобства

Создание представлений может обеспечивать пользователей дополнительными удобствами - например, возможностью работы только с действительно нужной частью данных. В результате можно добиться максимального упрощения той модели данных, которая понадобится каждому конечному пользователю.

Возможность настройки

Представления являются удобным средством настройки индивидуального образа базы данных. В результате одни и те же таблицы могут быть предъявлены пользователям в совершенно разном виде.

Обеспечение целостности данных

Если в операторе CREATE VIEW будет указана фраза WITH CHECK OPTION, то СУБД станет осуществлять контроль за тем, чтобы в исходные таблицы базы данных не была введена ни одна из строк, не удовлетворяющих предложению WHERE в определяющем запросе. Этот механизм гарантирует целостность данных в представлении. Практика ограничения доступа некоторых пользователей к данным посредством создания специализированных представлений, безусловно, имеет значительные преимущества перед предоставлением им прямого доступа к таблицам базы данных. Однако использование представлений в среде SQL не лишено *недостатков.*

Ограниченные возможности обновления

В некоторых случаях представления не позволяют вносить изменения в содержащиеся в них данные.

Структурные ограничения

Структура представления устанавливается в момент его создания. Если определяющий запрос представлен в форме SELECT * FROM_, то символ * ссылается на все столбцы, существующие в исходной таблице на момент создания представления. Если впоследствии в исходную таблицу базы данных добавятся новые столбцы, то они не появятся в данном представлении до тех пор, пока это представление не будет удалено и вновь создано.

Снижение производительности

Использование представлений связано с определенным снижением производительности. В одних случаях влияние этого фактора совершенно незначительно, тогда как в других оно может послужить источником существенных проблем. Например, представление, определенное с помощью сложного многотабличного запроса, может потребовать значительных затрат времени на обработку, поскольку при его разрешении потребуется выполнять соединение таблиц всякий раз, когда понадобится доступ к данному представлению. Выполнение разрешения представлений связано с использованием дополнительных вычислительных ресурсов.

1.6. Лекция № 6 (2 часа)

Тема: «Определение функций пользователя, примеры их создания и использования».

1.6.1. Вопросы лекции:

1. Типы функций.
2. Встроенные функции языка SQL**.**

1.6.2. Краткое содержание вопросов:

1. Типы функций.

Функции пользователя представляют собой самостоятельные объекты базы данных, такие, например, как хранимые процедуры или триггеры. Функция пользователя располагается в определенной базе данных и доступна только в ее контексте.

В SQL Server имеются следующие классы функций пользователя:

 Scalar – функции возвращают обычное скалярное значение, каждая может включать множество команд, объединяемых в один блок с помощью конструкции BEGIN...END;

 Inline– функции содержат всего одну команду SELECT и возвращают пользователю набор данных в виде значения типа данных TABLE;

 Multi-statement – функции также возвращают пользователю значение типа данных TABLE, содержащее набор данных, однако в теле функции находится множество команд SQL (INSERT, UPDATE и т.д.).

Именно с их помощью и формируется набор данных, который должен быть возвращен после выполнения функции.

Пользовательские функции сходны с хранимыми процедурами, но, в отличие от них, могут применяться в запросах так же, как и системные встроенные функции.

Пользовательские функции, возвращающие таблицы, могут стать альтернативой просмотрам. Просмотры ограничены одним выражением SELECT, а пользовательские функции способны включать дополнительные выражения, что позволяет создавать более сложные и мощные конструкции.

2. Встроенные функции языка SQL

Встроенные функции

Встроенные функции, имеющиеся в распоряжении пользователей при работе с SQL, можно условно разделить на следующие группы:

 математические функции;
 строковые функции;
 функции для работы с датой и временем;
 функции конфигурирования;
 функции системы безопасности;
 функции управления метаданными;
 статистические функции.

Математические функции

Краткий обзор математических функций представлен в таблице.

Таблица 11.1.

ABS	вычисляет абсолютное значение числа
ACOS	вычисляет арккосинус
ASIN	вычисляет арксинус
ATAN	вычисляет арктангенс
ATN2	вычисляет арктангенс с учетом квадратов
CEILING	выполняет округление вверх
COS	вычисляет косинус угла
COT	возвращает котангенс угла
DEGREES	преобразует значение угла из радиан в градусы
EXP	возвращает экспоненту

FLOOR	выполняет округление вниз
LOG	вычисляет натуральный логарифм
LOG10	вычисляет десятичный логарифм
PI	возвращает значение "пи"
POWER	возводит число в степень
RADIANS	преобразует значение угла из градуса в радианы
RAND	возвращает случайное число
ROUND	выполняет округление с заданной точностью
SIGN	определяет знак числа
SIN	вычисляет синус угла
SQUARE	выполняет возведение числа в квадрат
SQRT	извлекает квадратный корень
TAN	возвращает тангенс угла

Строковые функции

Краткий обзор строковых функций представлен в таблице.

Таблица 11.2.

ASCII	возвращает код ASCII левого символа строки
CHAR	по коду ASCII возвращает символ
CHARINDEX	определяет порядковый номер символа, с которого начинается вхождение подстроки в строку
DIFFERENCE	возвращает показатель совпадения строк
LEFT	возвращает указанное число символов с начала строки
LEN	возвращает длину строки
LOWER	переводит все символы строки в нижний регистр
LTRIM	удаляет пробелы в начале строки
NCHAR	возвращает по коду символ Unicode
PATINDEX	выполняет поиск подстроки в строке по указанному шаблону
REPLACE	заменяет вхождения подстроки на указанное значение
QUOTENAME	конвертирует строку в формат Unicode
REPLICATE	выполняет тиражирование строки определенное число раз
REVERSE	возвращает строку, символы которой записаны в обратном порядке
RIGHT	возвращает указанное число символов с конца строки
RTRIM	удаляет пробелы в конце строки
SOUNDEX	возвращает код звучания строки
SPACE	возвращает указанное число пробелов
STR	выполняет конвертирование значения числового типа в символьный формат
STUFF	удаляет указанное число символов, заменяя новой подстрокой
SUBSTRING	возвращает для строки подстроку указанной длины с заданного символа
UNICODE	возвращает Unicode-код левого символа строки
UPPER	переводит все символы строки в верхний регистр

Функции для работы с датой и временем

Краткий обзор основных функций для работы с датой и временем представлен в таблице.

Таблица 11.3.

DATEADD	добавляет к дате указанное значение дней, месяцев, часов и т.д.
DATEDIFF	возвращает разницу между указанными частями двух дат
DATENAME	выделяет из даты указанную часть и возвращает ее в символьном формате
DATEPART	выделяет из даты указанную часть и возвращает ее в числовом формате
DAY	возвращает число из указанной даты

GETDATE	возвращает текущее системное время
ISDATE	проверяет правильность выражения на соответствие одному из возможных форматов ввода даты
MONTH	возвращает значение месяца из указанной даты
YEAR	возвращает значение года из указанной даты

1.7. Лекция № 7 (2 часа)

Тема: «Хранимые процедуры».

1.7.1. Вопросы лекции:

1. Примеры создания, изменения и использования хранимых процедур с параметрами.
2. Определение входных и выходных параметров.
3. Примеры создания и вызова хранимых процедур.

1.7.2. Краткое содержание вопросов:

1. Примеры создания, изменения и использования хранимых процедур с параметрами.

Хранимые процедуры представляют собой группы связанных между собой операторов SQL, применение которых делает работу программиста более легкой и гибкой, поскольку выполнить хранимую процедуру часто оказывается гораздо проще, чем последовательность отдельных операторов SQL. Хранимые процедуры представляют собой набор команд, состоящий из одного или нескольких операторов SQL или функций и сохраняемый в базе данных в откомпилированном виде. Выполнение в базе данных хранимых процедур вместо отдельных операторов SQL дает пользователю следующие преимущества:

- необходимые операторы уже содержатся в базе данных;
- все они прошли этап синтаксического анализа и находятся в исполняемом формате; перед выполнением хранимой процедуры SQL Server генерирует для нее план исполнения, выполняет ее оптимизацию и компиляцию;
- хранимые процедуры поддерживают модульное программирование, так как позволяют разбивать большие задачи на самостоятельные, более мелкие и удобные в управлении части;
- хранимые процедуры могут вызывать другие хранимые процедуры и функции;
- хранимые процедуры могут быть вызваны из прикладных программ других типов;
- как правило, хранимые процедуры выполняются быстрее, чем последовательность отдельных операторов;
- хранимые процедуры проще использовать: они могут состоять из десятков и сотен команд, но для их запуска достаточно указать всего лишь имя нужной хранимой процедуры. Это позволяет уменьшить размер запроса, посылаемого от клиента на сервер, а значит, и нагрузку на сеть.

Выполнение хранимой процедуры

Для выполнения хранимой процедуры используется команда:

[[EXEC [UTE] имя_процедуры [;номер]
[[@имя_параметра=]{значение | @имя_переменной}
[OUTPUT]|[DEFAULT]][,...n]

Если вызов хранимой процедуры не является единственной командой в пакете, то присутствие команды EXECUTE обязательно. Более того, эта команда требуется для вызова процедуры из тела другой процедуры или триггера.

Использование ключевого слова OUTPUT при вызове процедуры разрешается только для параметров, которые были объявлены при создании процедуры с ключевым словом OUTPUT.

Когда же при вызове процедуры для параметра указывается ключевое слово DEFAULT, то будет использовано значение по умолчанию. Естественно, указанное слово DEFAULT разрешается только для тех параметров, для которых определено значение по умолчанию.

Из синтаксиса команды EXECUTE видно, что имена параметров могут быть опущены при вызове процедуры. Однако в этом случае пользователь должен указывать значения для параметров в том же порядке, в каком они перечислялись при создании процедуры.

Присвоить параметру значение по умолчанию, просто пропустив его при перечислении нельзя. Если же требуется опустить параметры, для которых определено значение по умолчанию, достаточно явного указания имен параметров при вызове хранимой процедуры. Более того, таким способом можно перечислять параметры и их значения в произвольном порядке.

Отметим, что при вызове процедуры указываются либо имена параметров со значениями, либо только значения без имени параметра. Их комбинирование не допускается.

Пример 12.1. Процедура без параметров. Разработать процедуру для получения названий и стоимости товаров, приобретенных Ивановым.

```
CREATE PROC my_proc1
AS
SELECT Товар.Название,
Товар.Цена*Сделка.Количество
AS Стоимость, Клиент.Фамилия
FROM Клиент INNER JOIN
(Товар INNER JOIN Сделка
ON Товар.КодТовара=Сделка.КодТовара)
ON Клиент.КодКлиента=Сделка.КодКлиента
WHERE Клиент.Фамилия=’Иванов’
```

Пример 12.1. Процедура для получения названий и стоимости товаров, приобретенных Ивановым.

Для обращения к процедуре можно использовать команды:
EXEC my_proc1 или my_proc1
Процедура возвращает набор данных.

2. Определение входных и выходных параметров

Пример 12.2. Процедура без параметров. Создать процедуру для уменьшения цены товара первого сорта на 10%.

```
CREATE PROC my_proc2
AS
UPDATE Товар SET Цена=Цена*0.9
WHERE Сорт=’первый’
```

Пример 12.2. Процедура для уменьшения цены товара первого сорта на 10%.

Для обращения к процедуре можно использовать команды:
EXEC my_proc2 или my_proc2
Процедура не возвращает никаких данных.

Пример 12.3. Процедура с входным параметром. Создать процедуру для получения

названий и стоимости товаров, которые приобрел заданный клиент.

```
CREATE PROC my_proc3
@k VARCHAR(20)
AS
SELECT Товар.Название,
        Товар.Цена*Сделка.Количество
AS Стоимость, Клиент.Фамилия
FROM Клиент INNERJOIN
(Товар INNERJOIN Сделка
ON Товар.КодТовара=Сделка.КодТовара)
ON Клиент.КодКлиента=Сделка.КодКлиента
WHERE Клиент.Фамилия=@k
```

Пример 12.3. Процедура для получения названий и стоимости товаров, которые приобрел заданный клиент.

Для обращения к процедуре можно использовать команды:

```
EXEC my_proc3 'Иванов' или
my_proc3 @k='Иванов'
```

Пример 12.4. Процедура с входными параметрами. Создать процедуру для уменьшения цены товара заданного типа в соответствии с указанным %.

```
CREATE PROC my_proc4
@t VARCHAR(20), @p FLOAT
AS
UPDATE Товар SET Цена=Цена*(1-@p)
WHERE Тип=@t
```

Пример 12.4. Процедура для уменьшения цены товара заданного типа в соответствии с указанным %.

Для обращения к процедуре можно использовать команды:

```
EXEC my_proc4 'Вафли',0.05 или
EXEC my_proc4 @t='Вафли', @p=0.05
```

3. Примеры создания и вызова хранимых процедур.

Пример 12.5. Процедура с входными параметрами и значениями по умолчанию. Создать процедуру для уменьшения цены товара заданного типа в соответствии с указанным %.

```
CREATE PROC my_proc5
@t VARCHAR(20)=’Конфеты',
@p FLOAT=0.1
AS
UPDATE Товар SET Цена=Цена*(1-@p)
WHERE Тип=@t
```

Пример 12.5. Процедура с входными параметрами и значениями по умолчанию. Создать процедуру для уменьшения цены товара заданного типа в соответствии с указанным %.

Для обращения к процедуре можно использовать команды:

```
EXEC my_proc5 'Вафли',0.05 или
```

EXEC my_proc5 @t='Вафли', @p=0.05 или
EXEC my_proc5 @p=0.05
В этом случае уменьшается цена конфет (значение типа не указано при вызове процедуры и берется по умолчанию).
EXEC my_proc5
В последнем случае оба параметра (и тип, и проценты) не указаны при вызове процедуры, их значения берутся по умолчанию.

Пример 12.6. Процедура с входными и выходными параметрами. Создать процедуру для определения общей стоимости товаров, проданных за конкретный месяц.

```
CREATE PROC my_proc6
@m INT,
@s FLOAT OUTPUT
AS
SELECT @s=Sum(Товар.Цена*Сделка.Количество)
FROM Товар INNER JOIN Сделка
ON Товар.КодТовара=Сделка.КодТовара
GROUP BY Month(Сделка.Дата)
HAVING Month(Сделка.Дата)=@m
```

Пример 12.6. Процедура с входными и выходными параметрами. Создать процедуру для определения общей стоимости товаров, проданных за конкретный месяц.

Для обращения к процедуре можно использовать команды:

```
DECLARE @st FLOAT
EXEC my_proc6 1,@st OUTPUT
SELECT @st
```

Этот блок команд позволяет определить стоимость товаров, проданных в январе (входной параметр месяц указан равным 1).

Создать процедуру для определения общего количества товаров, приобретенных фирмой, в которой работает заданный сотрудник.

Сначала разработаем процедуру для определения фирмы, где работает сотрудник.

```
CREATE PROC my_proc7
@n VARCHAR(20),
@f VARCHAR(20) OUTPUT
AS
SELECT @f=Фирма
FROM Клиент
WHERE Фамилия=@n
```

Пример 12.7. Использование вложенных процедур. Создать процедуру для определения общего количества товаров, приобретенных фирмой, в которой работает заданный сотрудник.

Затем создадим процедуру, подсчитывающую общее количество товара, который закуплен интересующей нас фирмой.

```
CREATEPROCmy_proc8
@fam VARCHAR(20),
@kol INT OUTPUT
```

```
AS
DECLARE @firm VARCHAR(20)
EXEC my_proc7 @fam,@firm OUTPUT
SELECT @kol=Sum(Сделка.Количество)
FROM Клиент INNERJOIN Сделка
ONКлиент.КодКлиента=Сделка.КодКлиента
GROUP BY Клиент.Фирма
HAVING Клиент.Фирма=@firm
```

Пример 12.7. Создание процедуры для определения общего количества товаров, приобретенных фирмой, в которой работает заданный сотрудник.

Вызов процедуры осуществляется с помощью команды:

```
DECLARE @k INT
EXEC my_proc8 ‘Иванов’,@k OUTPUT
SELECT @k
```

1.8. Лекция № 8 (2 часа)

Тема: «Хранимые процедуры».

1.8.1. Вопросы лекции:

1. Определение триггера, область его использования, место и роль триггера в обеспечении целостности данных.
2. Типы триггеров.
3. Программирование триггера.

1.8.2. Краткое содержание вопросов:

1. Определение триггера, область его использования, место и роль триггера в обеспечении целостности данных.

Триггеры являются одной из разновидностей хранимых процедур. Их исполнение происходит при выполнении для таблицы какого-либо оператора языка манипулирования данными (DML). Триггеры используются для проверки целостности данных, а также для отката транзакций.

Триггер – это откомпилированная SQL-процедура, исполнение которой обусловлено наступлением определенных событий внутри реляционной базы данных. Применение триггеров большей частью весьма удобно для пользователей базы данных. И все же их использование часто связано с дополнительными затратами ресурсов на операции ввода/вывода. В том случае, когда тех же результатов (с гораздо меньшими непроизводительными затратами ресурсов) можно добиться с помощью хранимых процедур или прикладных программ, применение триггеров нецелесообразно.

Триггеры – особый инструмент SQL-сервера, используемый для поддержания целостности данных в базе данных. С помощью ограничений целостности, правил и значений по умолчанию не всегда можно добиться нужного уровня функциональности. Часто требуется реализовать сложные алгоритмы проверки данных, гарантирующие их достоверность и реальность. Кроме того, иногда необходимо отслеживать изменения значений таблицы, чтобы нужным образом изменить связанные данные.

Триггеры можно рассматривать как своего рода фильтры, вступающие в действие после выполнения всех операций в соответствии с правилами, стандартными значениями и т.д.

Триггер представляет собой специальный тип хранимых процедур, запускаемых сервером автоматически при попытке изменения данных в таблицах, с которыми триггеры связаны. Каждый триггер привязывается к конкретной таблице. Все производимые им модификации данных рассматриваются как одна транзакция. В случае обнаружения

ошибки или нарушения целостности данных происходит откат этой транзакции. Тем самым внесение изменений запрещается. Отменяются также все изменения, уже сделанные триггером. Создает триггер только владелец базы данных. Это ограничение позволяет избежать случайного изменения структуры таблиц, способов связи с ними других объектов и т.п.

Триггер представляет собой весьма полезное и в то же время опасное средство. Так, при неправильной логике его работы можно легко уничтожить целую базу данных, поэтому триггеры необходимо очень тщательно отлаживать. В отличие от обычной подпрограммы, триггер выполняется неявно в каждом случае возникновения триггерного события, к тому же он не имеет аргументов. Приведение его в действие иногда называют запуском триггера. *С помощью триггеров достигаются следующие цели:*

 проверка корректности введенных данных и выполнение сложных ограничений целостности данных, которые трудно, если вообще возможно, поддерживать с помощью ограничений целостности, установленных для таблицы;

 выдача предупреждений, напоминающих о необходимости выполнения некоторых действий при обновлении таблицы, реализованном определенным образом;

 накопление аудиторской информации посредством фиксации сведений о внесенных изменениях и тех лицах, которые их выполнили;

 поддержка репликации.

Основной формат команды CREATE TRIGGER показан ниже:

<Определение_триггера>::=

CREATE TRIGGER имя_триггера
BEFORE | AFTER <триггерное_событие>
ON <имя_таблицы>
[REFERENCING
<список_старых_или_новых_псевдонимов>]
[FOR EACH { ROW | STATEMENT}]
[WHEN(условие_триггера)]
<тело_триггера>

триггерные события состоят из вставки, удаления и обновления строк в таблице. В последнем случае для триггерного события можно указать конкретные имена столбцов таблицы. Время запуска триггера определяется с помощью ключевых слов BEFORE (триггер запускается до выполнения связанных с ним событий) или AFTER (после их выполнения).

Выполняемые триггером действия задаются для каждой строки (FOR EACH ROW), охваченной данным событием, или только один раз для каждого события (FOR EACH STATEMENT).

Обозначение <список_старых_или_новых_псевдонимов> относится к таким компонентам, как старая или новая строка (OLD / NEW) либо старая или новая таблица (OLD TABLE / NEW TABLE). Ясно, что старые значения не применимы для событий вставки, а новые – для событий удаления.

2. Типы триггеров.

В SQL Server существует два параметра, определяющих поведение триггеров:

 AFTER. Триггер выполняется после успешного выполнения вызвавших его команд. Если же команды по какой-либо причине не могут быть успешно завершены, триггер не выполняется. Следует отметить, что изменения данных в результате выполнения запроса пользователя и выполнение триггера осуществляется в теле одной транзакции: если произойдет откат триггера, то будут отклонены и пользовательские изменения. Можно определить несколько AFTER-триггеров для каждой операции (INSERT, UPDATE, DELETE). Если для таблицы предусмотрено выполнение нескольких AFTER-триггеров, то

с помощью системной хранимой процедуры sp_settriggerorder можно указать, какой из них будет выполняться первым, а какой последним. По умолчанию в SQL Server все триггеры являются AFTER-триггерами.
 INSTEAD OF. Триггер вызывается вместо выполнения команд. В отличие от AFTER-триггера INSTEAD OF-триггер может быть определен как для таблицы, так и для просмотра. Для каждой операции INSERT, UPDATE, DELETE можно определить только один INSTEAD OF-триггер. Триггеры различают по типу команд, на которые они реагируют.
Существует три типа триггеров:
 INSERT TRIGGER – запускаются при попытке вставки данных с помощью команды INSERT.
 UPDATE TRIGGER – запускаются при попытке изменения данных с помощью команды UPDATE.
 DELETE TRIGGER – запускаются при попытке удаления данных с помощью команды DELETE.

Конструкции [DELETE] [,] [INSERT] [,] [UPDATE] и FOR | AFTER | INSTEADOF } { [INSERT] [,] [UPDATE] определяют, на какую команду будет реагировать триггер. При его создании должна быть указана хотя бы одна команда. Допускается создание триггера, реагирующего на две или на все три команды.

Аргумент WITH APPEND позволяет создавать несколько триггеров каждого типа. При создании триггера с аргументом NOT FOR REPLICATION запрещается его запуск во время выполнения модификации таблиц механизмами репликации.

Конструкция AS sql_оператор[...n] определяет набор SQL- операторов и команд, которые будут выполнены при запуске триггера.
Отметим, что внутри триггера не допускается выполнение ряда операций, таких, например, как:
 создание, изменение и удаление базы данных;
 восстановление резервной копии базы данных или журнала транзакций.

Выполнение этих команд не разрешено, так как они не могут быть отменены в случае отката транзакции, в которой выполняется триггер. Это запрещение вряд ли может каким-то образом сказаться на функциональности создаваемых триггеров. Трудно найти такую ситуацию, когда, например, после изменения строки таблицы потребуется выполнить восстановление резервной копии журнала транзакций.

3. Программирование триггеров.

При выполнении команд добавления, изменения и удаления записей сервер создает две специальные таблицы: inserted и deleted. В них содержатся списки строк, которые будут вставлены или удалены по завершении транзакции. Структура таблиц inserted и deleted идентична структуре таблиц, для которой определяется триггер.

Для каждого триггера создается свой комплект таблиц inserted и deleted, поэтому никакой другой триггер не сможет получить к ним доступ. В зависимости от типа операции, вызвавшей выполнение триггера, содержимое таблиц inserted и deleted может быть разным:
 команда INSERT – в таблице inserted содержатся все строки, которые пользователь пытается вставить в таблицу; в таблице deleted не будет ни одной строки; после завершения триггера все строки из таблицы inserted переместятся в исходную таблицу;
 команда DELETE – в таблице deleted будут содержаться все строки, которые пользователь попытается удалить; триггер может проверить каждую строку и определить, разрешено ли ее удаление; в таблице inserted не окажется ни одной строки;
 команда UPDATE – при ее выполнении в таблице deleted находятся старые значения строк, которые будут удалены при успешном завершении триггера.

Новые значения строк содержатся в таблице inserted. Эти строки добавятся в исходную таблицу после успешного выполнения триггера. Для получения информации о количестве строк, которое будет изменено при успешном завершении триггера, можно использовать функцию @@ROWCOUNT; она возвращает количество строк, обработанных последней командой. Следует подчеркнуть, что триггер запускается не при попытке изменить конкретную строку, а в момент выполнения команды изменения. Одна такая команда воздействует на множество строк, поэтому триггер должен обрабатывать все эти строки.

Если триггер обнаружил, что из 100 вставляемых, изменяемых или удаляемых строк только одна не удовлетворяет тем или иным условиям, то никакая строка не будет вставлена, изменена или удалена. Такое поведение обусловлено требованиями транзакции – должны быть выполнены либо все модификации, либо ни одной.

Триггер выполняется как неявно определенная транзакция, поэтому внутри триггера допускается применение команд управления транзакциями. В частности, при обнаружении нарушения ограничений целостности для прерывания выполнения триггера и отмены всех изменений, которые пытался выполнить пользователь, необходимо использовать команду ROLLBACK TRANSACTION.

Для получения списка столбцов, измененных при выполнении команд INSERT или UPDATE, вызвавших выполнение триггера, можно использовать функцию COLUMNS_UPDATED(). Она возвращает двоичное число, каждый бит которого, начиная с младшего, соответствует одному столбцу таблицы (в порядке следования столбцов при создании таблицы). Если бит установлен в значение "1", то соответствующий столбец был изменен. Кроме того, факт изменения столбца определяет и функция UPDATE (имя_столбца).

Для удаления триггера используется команда DROPTRIGGER {имя_триггера} [,...n]

2. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ПРОВЕДЕНИЮ ПРАКТИЧЕСКИХ ЗАНЯТИЙ

2.1 Лабораторная работа № 1, 2 (4 часа).

Тема: «Определение структурированного языка запросов SQL».

2.1.1 Цель работы: Получить практический опыт написания SQL запросов.

2.1.2 Задачи работы:

1. Изучить предложенную структуру БД;

2. Создать реляционную базу данных на основе MS SQL Server 2005.

2.1.3 Описание (ход) работы:

Базы данных составляют основу для построения информационных систем любого масштаба и предназначения. В теории баз данных одними из основных являются вопросы, связанные с анализом предметной области и моделированием структуры данных, управлением данными и их анализом.

Основой любой базы данных является реализованная в ней модель данных, представляющая собой множество структур данных, ограничений целостности и операций манипулирования данными. С помощью модели данных могут быть представлены объекты предметной области и существующие между ними связи.

Результатом лабораторной работы будет создание реляционной базы данных на основе MS SQL Server 2005.

В реляционной базе данных данные представлены в виде собрания таблиц. Таблица состоит из определенного числа столбцов (полей) и произвольного числа строк (записей).

Планируемая база данных будет представлять собой информационное хранилище данных об успеваемости студентов и состоять из следующих таблиц:

- Speciality (специальность)
- Course (курс)
- Group (группа)
- Discipline (дисциплина)
- Account (тип отчетности)
- Mark (отметка)
- Status (академический статус студента)
- Position (должность)
- People (люди)
- Student (студент)
- Teacher (преподаватель)
- SemesterResults (результаты сессии, семестра)

Структура данных таблиц приведена в Приложении.

1. Начало работы в Microsoft SQL Server Management Studio

Для создания баз данных используем среду Microsoft SQL Server Management Studio. На запрос соединения с сервером выбираем (рис. 1):

Тип сервера: Компонент Database Engine

Имя сервера: SQL-MS.

Под таким именем в домене fizmat.vspu.ru. доступна машина, на которой установлены серверные компоненты MS SQL Server 2005. Можно попробовать выбрать сервер из выпадающего списка серверов. Можно также обратиться к этой машине по IP-адресу 192.168.10.152 из локальной сети.

Проверка подлинности: Проверка подлинности SQL Server.

Такая настройка позволяет создавать пользователей данного экземпляра SQL Server независимо от компьютера, с которого производится вход.

Имя входа: studentMBS21.

Пароль: student.

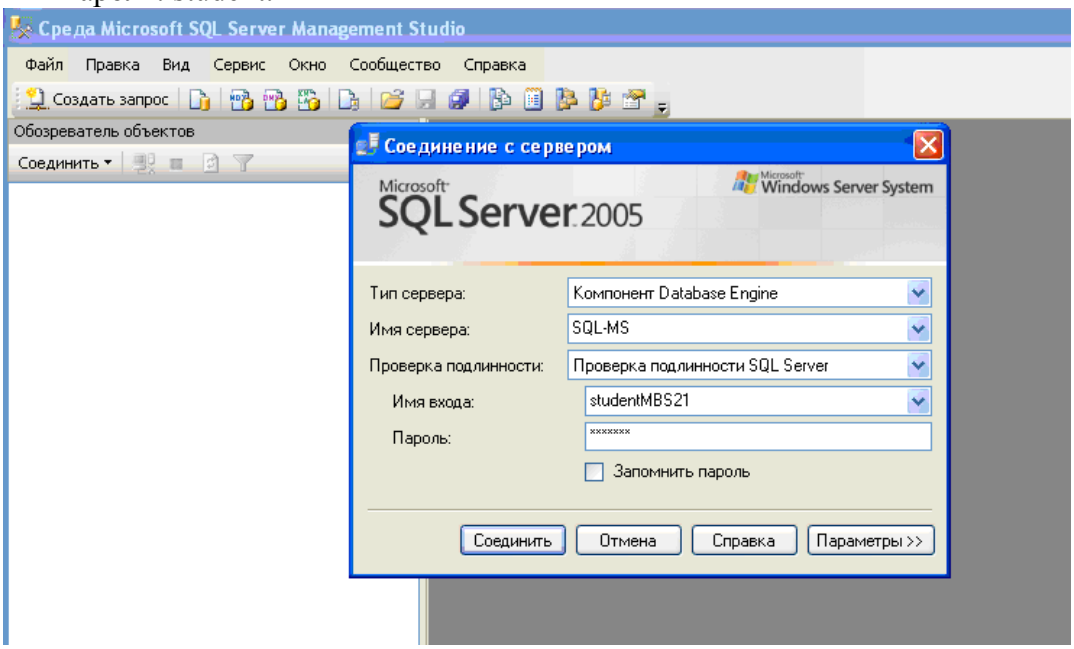


Рисунок 1. Окно входа в Microsoft SQL Server Management Studio 2005

Примечание. Пользователь studentMBS21 обладает большими полномочиями на этом сервере, поэтому пользоваться им надо очень аккуратно. Под этим пользователем мы создадим базу данных, а заполнять её и производить поиск по ней мы будем под другими пользователями. Предпочтительнее всего использовать свою учетную запись в домене fizmat.vspu.ru. В этом случае надо выбирать проверку подлинности Windows.

Теперь нажимаем кнопку «Параметры» и выбираем (рис. 2):

Соединение с базой данных → Обзор сервера… → Пользовательские базы данных → trial_base.

Сетевой протокол → TCP/IP

Нажимаем кнопку «Соединить».

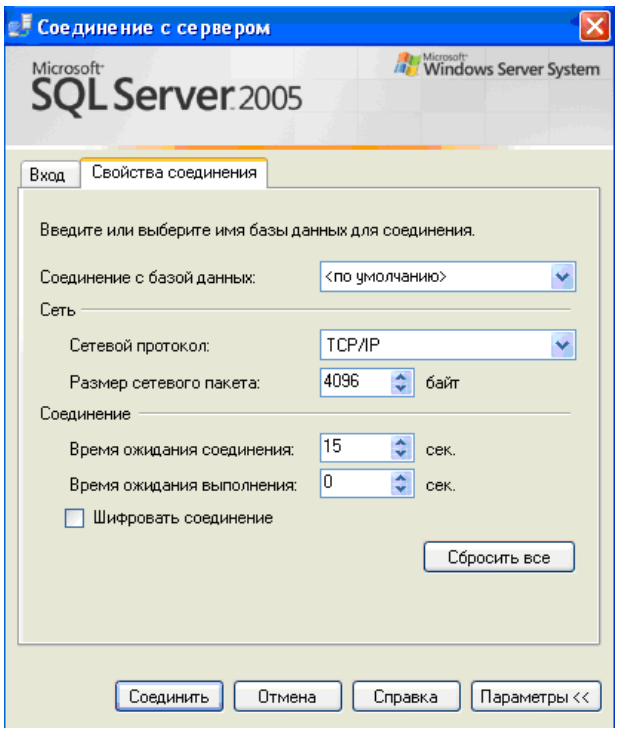


Рисунок 2. Окно входа в Microsoft SQL Server Management Studio 2005 (вкладка Параметры)

Примечание. База данных trial_base является базой данной по умолчанию для пользователя studentMBS21, она была создана при регистрации этого пользователя. В случае, когда права доступа пользователя не ограничены (как в рассматриваемом случае), вкладку Параметры можно не открывать. Если же пользователь имеет доступ только к определенным базам данных, при подключении к серверу нужно одну из этих баз указывать.

После успешного соединения с базой данных на экране видим следующую картинку (рис. 3):

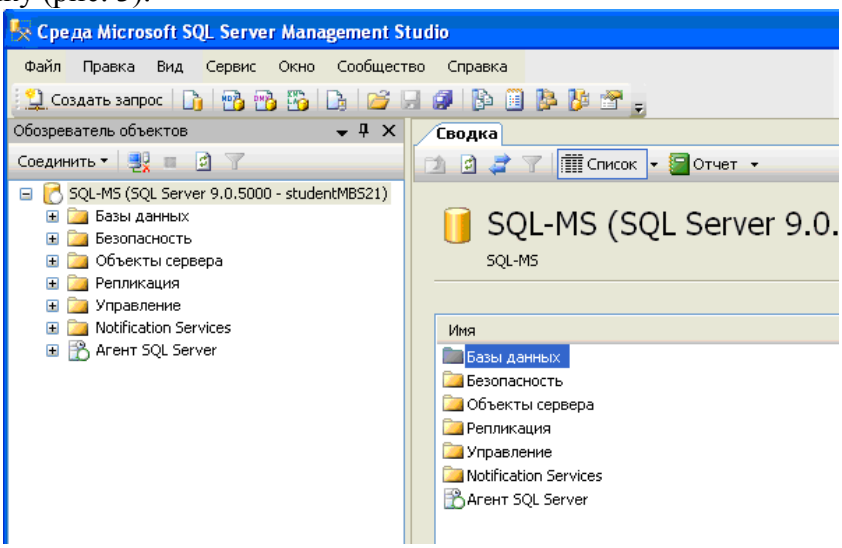


Рисунок 3. Подключение к SQL - серверу установлено

Среда MS SQL Management Studio предоставляет удобный инструментарий для создания, редактирования, заполнения баз данных. Но настоящие профессионалы в своей работе редко пользуются этой средой, а для выполнения своих задач используют SQL-запросы. Мы будем пользоваться, когда это удобно и

наглядно, графическим режимом, но основной упор будем делать на освоении базы языка SQL.

2. Создание базы данных в среде Microsoft SQL Server Management Studio

В разделе «Базы данных» правой кнопкой выбираем «Создать базу данных…» (рис. 4). Назовем базу данных по индексу группы – mbs21. Владельцем базы данных назначим пользователя, под именем которого был произведен вход – studentMBS21. В разделе «Параметры» выбираем тип сортировки Cyrillic_General_BIN (для примера), нажимаем ОК.

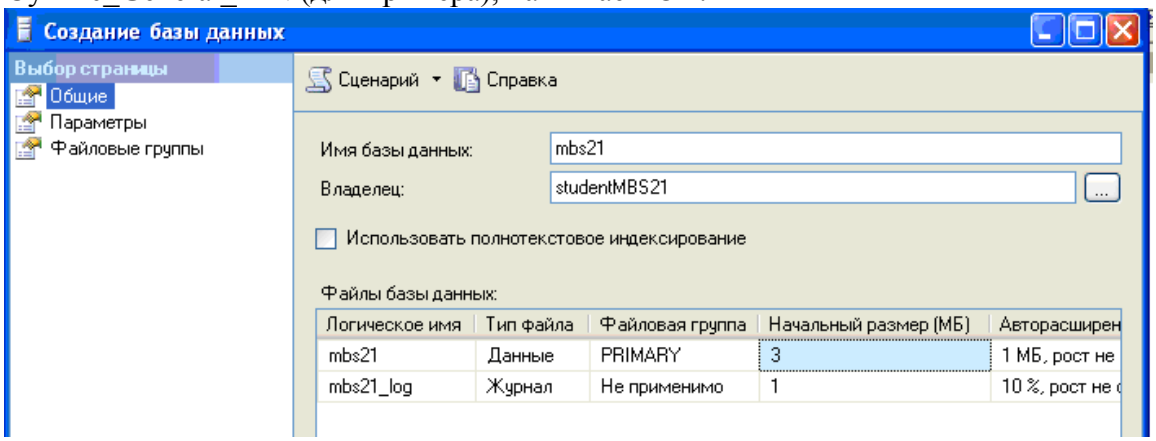


Рисунок 4. Создание базы данных

В разделе «Базы данных» Обозревателя объектов появилась вновь созданная mbs21 (проверьте!):

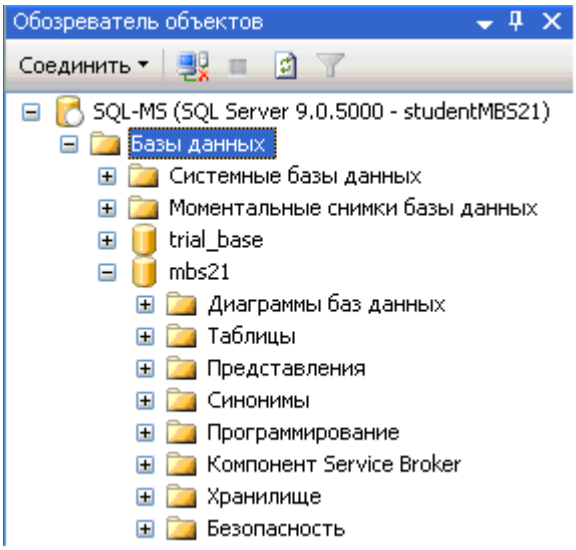


Рисунок 5. Обозреватель объектов

3. Создание таблиц базы данных в среде Microsoft SQL Server Management Studio

Начнем с создания таблицы Speciality. Структура таблицы приведена ниже:

Имя поля (столбца)	Содержание	Тип данных	Возможность содержать NULL
Num	Первичный ключ	int	нет
Name	Название специальности	varchar(60)	нет

В реляционных базах данных первичный ключ используется как уникальный идентификатор записи. Это поле является обязательным, оно

используется для связи таблиц по внешним ключам (примеры такого связывания будут рассмотрены далее). Первичный ключ должен иметь целочисленный тип (в данном случае - int). Во втором поле будет храниться название специальности - некоторая строка, поэтому мы выбираем для этого поля тип varchar(60). Число в скобках означает максимальное число символов в строке. Детальную информацию об этих типах можно посмотреть в справке.

Простейшим образом можно создавать таблицы средствами MS SQL Server Management Studio (правая кнопка мыши на заголовке «Таблицы» > Создать таблицу.). Получаем следующее:

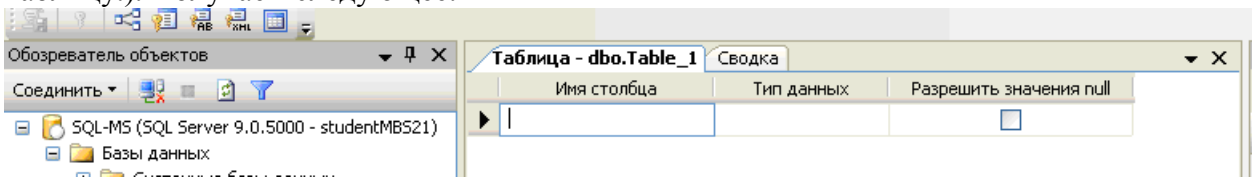


Рисунок 6. Создание таблицы

Вводим имя первого столбца Num (первичный ключ – в том столбце хранится номер записи), выбираем из выпадающего списка тип данных int. Первичный ключ не может быть пустым, поэтому и оставляем неотмеченным поле «Разрешить значения null». Затем аналогичным образом вводим имя второго столбца, задаем тип, запрещаем полю иметь значение null. Таблица принимает следующий вид:

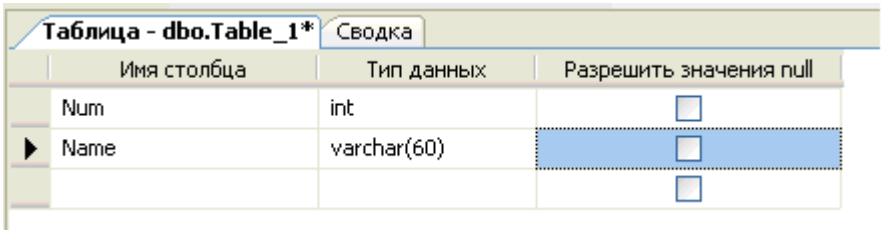


Рисунок 7.

Теперь необходимо указать, что поле Num будет являться первичным ключом. Правой кнопкой мыши щелкаем по этому полю и выбираем «Задать первичный ключ»:

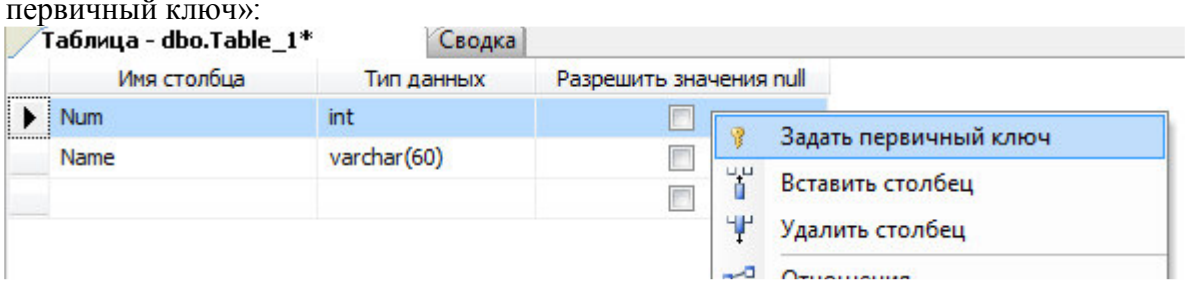


Рисунок 8.

Сохраняем таблицу под именем Speciality (после этого таблица должна появиться в обозревателе объектов). Теперь можно перейти к заполнению этой таблицы (для этого нужно в обозревателе объектов выбрать эту таблицу и в контекстном меню нажать «Открыть таблицу»):

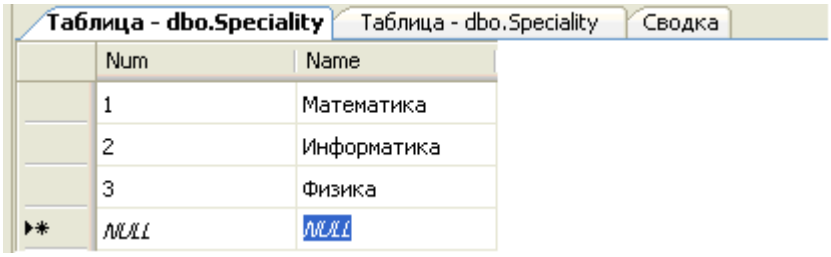


Рисунок 9.

При заполнении вы обнаружите, что каждый раз приходится вводить не только полезную информацию (название специальности), но и номер записи. Чтобы вводить номер записи автоматически, нужно задать спецификацию идентифицирующего столбца. Для этого необходимо в свойствах столбца указать, что данный столбец является идентифицирующим (рис. 10):

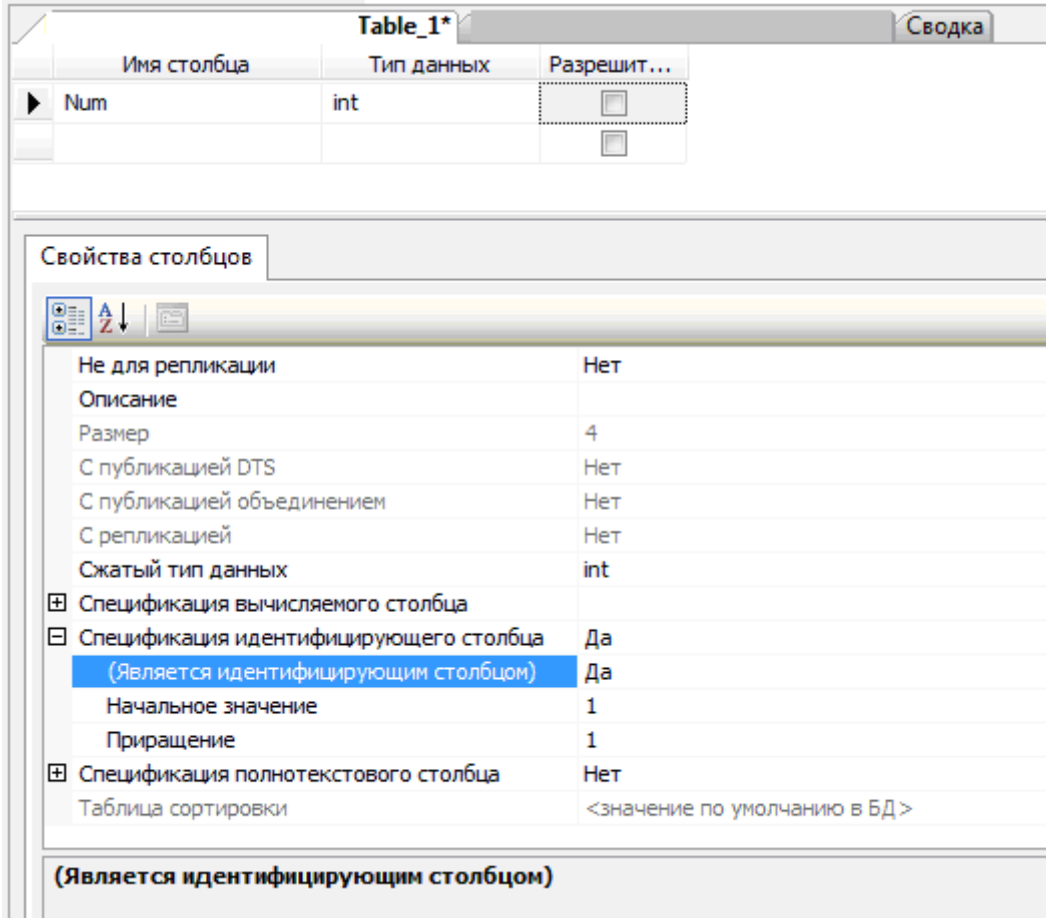


Рисунок 10. Определение свойств идентифицирующего столбца

4. Создание таблиц базы данных с помощью SQL-запроса

Создание таблиц в графическом режиме, безусловно, удобно, однако не универсально. При использовании других средств разработки баз данных (например, IBM DB2) придется привыкать к новым приемам работы. Использование конструкций языка SQL позволяет работать с базами данных, исходя из единого подхода, в любой среде управления базами данных.

Выберите на панели инструментов «Создать запрос»:

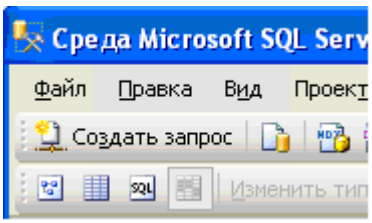


Рисунок 11.

Создадим новую базу данных запросом. Напишем

CREATE DATABASE mbs21_query

и нажмем F5. В обозревателе объектов должна появиться новая база (если сразу не появилась, то надо выделить мышью раздел «Базы данных» и в контекстном меню выбрать «Обновить»).

Теперь создадим таблицу Speciality. Упрощенный синтаксис создания таблиц следующий:

```
CREATE TABLE <имя таблицы> (
   <имя столбца 1> <тип данных> [NOT NULL] [DEFAULT <значение по умолчанию>],
   <имя столбца 2> <тип данных> [NOT NULL] [DEFAULT <значение по умолчанию>],
   ...
)
```

Введем новый запрос:

```
/* создание таблицы Специальность*/
USE mbs21_query                          -- определяем базу данных, в которую входит таблица
CREATE  TABLE  Speciality(
   Num  INT IDENTITY(1,1) PRIMARY KEY NOT NULL, -- первичный ключ
   NameSpec VARCHAR(60)                  -- название специальности
)
```

В обозревателе объектов видим, что таблица действительно создана. Файл с SQL-запросом сохраняем в своей папке (в конце работы необходимо показать запросы, которые были выполнены, преподавателю). Слово IDENTITY(1,1) добавлено, чтобы поле первичного ключа Num автоматически нумеровалось начиная с единицы (фактически, эта конструкция определяет спецификацию идентифицирующего столбца).

Таким же образом необходимо создать остальные таблицы. Рассмотрим таблицу Course.

Таблица Course (курс)

Имя поля (столбца)	Содержание	Тип данных	Возможность содержать NULL
Num	Первичный ключ	int	нет
Name	Название специальности	varchar(60)	нет

YearEntry	Год поступления	int	нет
YearFinal	Год выпуска	int	да
Speciality	Специальность (внешний ключ ссылается на первичный ключ таблицы Speciality)	int	нет

Эта таблица содержит поле Speciality, которое ссылается на первичный ключ таблицы Speciality. Чтобы создать такую таблицу, необходимо выполнить запрос:

```
/* создание таблицы Курс */
USE mbs21_query                    -- определяем базу данных, в которую входит таблица
CREATE  TABLE  Course(
   Num  INT IDENTITY(1,1) PRIMARY KEY NOT NULL, -- первичный ключ
   YearEntry INT NOT NULL,          -- год поступления
   YearFinal INT,                   -- год окончания
   Speciality INT FOREIGN KEY REFERENCES Speciality(Num) -- специальность,
   -- ссылка по внешнему ключу на поле Num таблицы Speciality
)
```

Примечание. Ссылку можно создать только на существующую таблицу. Задать ссылку по внешнему ключу можно и после создания таблицы (подробно будет рассмотрено в следующей лабораторной работе).

Задание. Создайте все остальные таблицы, указанные в Приложении, используя SQL – запросы.

Приложение. Структура данных

Таблица Speciality (специальность)

Имя поля (столбца)	Содержание	Тип данных	Возможность содержать NULL
Num	Первичный ключ	int	Нет
Name	Название	varchar(60)	Нет

Таблица Course (курс)

Имя поля (столбца)	Содержание	Тип данных	Возможность содержать NULL
Num	Первичный ключ	int	нет
Name	Название специальности	varchar(60)	нет
YearEntry	Год поступления	int	нет
YearFinal	Год выпуска	int	да
Speciality	Специальность (внешний ключ ссылается на первичный ключ таблицы Speciality)	int	нет

Таблица Group (группа)

Имя поля (столбца)	Содержание	Тип данных	Возможность содержать NULL
Num	Первичный ключ	int	нет

Name	Название специальности	varchar(60)	нет
Course	Курс (внешний ключ ссылается на первичный ключ таблицы Course)	int	нет

Таблица Discipline (дисциплина)

Имя поля (столбца)	Содержание	Тип данных	Возможность содержать NULL
Num	Первичный ключ	int	Нет
Name	Название (возможные значения: программирование, алгебра…)	varchar(60)	Нет

Таблица Account (тип отчетности)

Имя поля (столбца)	Содержание	Тип данных	Возможность содержать NULL
Num	Первичный ключ	int	Нет
Name	Название (возможные значения: экзамен, зачет, дифференцированный зачет…)	varchar(30)	Нет

Таблица Mark (отметка)

Имя поля (столбца)	Содержание	Тип данных	Возможность содержать NULL
Num	Первичный ключ	int	Нет
Name	Название (возможные значения: зачтено, не зачтено, отлично, хорошо…)	varchar(30)	Нет
Value	Значение (возможные значения: 0, 1, …, 5)	int	Нет

Таблица Status (академический статус студента)

Имя поля (столбца)	Содержание	Тип данных	Возможность содержать NULL
Num	Первичный ключ	int	Нет
Name	Название (возможные значения: обучается, отчислен, в академическом отпуске, в отпуске по уходу за ребенком)	varchar(60)	Нет

Таблица Position (должность)

Имя поля (столбца)	Содержание	Тип данных	Возможность содержать NULL
Num	Первичный ключ	int	Нет
Name	Название (возможные значения: ассистент, старший преподаватель, доцент…)	varchar(60)	Нет

Таблица People (люди)

Имя поля (столбца)	Содержание	Тип данных	Возможность содержать NULL
Num	Первичный ключ	int	Нет
LastName	Фамилия	varchar(30)	Нет

FirstName	Имя	varchar(30)	Нет
MiddleName	Отчество	varchar(30)	Да
Male	Пол	char(1)	Нет
BrthDate	День рождения	datetime	Да
Addr	Адрес	varchar(100)	Да

Таблица Student (студент)

Имя поля (столбца)	Содержание	Тип данных	Возможность содержать NULL
Num	Первичный ключ	int	Нет
People	Человек (внешний ключ ссылается на первичный ключ таблицы People)	int	Нет
Group	Группа (внешний ключ ссылается на первичный ключ таблицы Group)	int	Нет
StudNum	Номер студенческого билета	varchar(30)	Нет
Status	Академический статус студента (внешний ключ ссылается на первичный ключ таблицы Status)	int	Нет

Таблица Teacher (преподаватель)

Имя поля (столбца)	Содержание	Тип данных	Возможность содержать NULL
Num	Первичный ключ (табельный номер сотрудника)	int	Нет
People	Человек (внешний ключ ссылается на первичный ключ таблицы People)	int	Нет
Position	Должность (внешний ключ ссылается на первичный ключ таблицы Position)	int	Нет

Таблица SemesterResults (результаты сессии)

Имя поля (столбца)	Содержание	Тип данных	Возможность содержать NULL
Num	Первичный ключ	int	Нет
Student	Студент (внешний ключ ссылается на первичный ключ таблицы Student)	int	Нет
Semester	Порядковый номер семестра	int	Нет
Account	Тип отчетности (внешний ключ ссылается на первичный ключ таблицы Account)	int	Нет
Discipline	Дисциплина (внешний ключ ссылается на первичный ключ таблицы Discipline)	int	Нет
Teacher	Преподаватель (внешний ключ ссылается на первичный ключ таблицы Teacher)	int	Нет
Mark	Отметка (внешний ключ ссылается на первичный ключ таблицы Mark)	int	Нет
Date	Дата сдачи отчетности	DateTime	Нет

2.2 Лабораторная работа № 3, 4 (4 часа).

Тема: «Эффективное выполнение запросов для извлечения данных».

2.2.1 Цель работы: Получить практический опыт написания SQL запросов для извлечения данных из базы данных.

2.2.2 Задачи работы:

1.Изучить теоретический материал по написанию SQL запросов для извлечения данных;

2. Написать несколько SQL запросов для извлечения данных в базе данных на основе MS SQL Server 2005.

2.2.3 Описание (ход) работы:

SQL — это аббревиатура выражения Structured Query Language (язык структурированных запросов). SQL основывается на реляционной алгебре и специально разработан для взаимодействия с реляционными базами данных.

SQL является, прежде всего, информационно-логическим языком, предназначенным для описания хранимых данных, их извлечения и модификации. SQL не является языком программирования. Вместе с тем конкретные реализации языка, как правило, включают различные процедурные расширения.

Язык SQL представляет собой совокупность операторов, которые можно разделить на четыре группы:

- DDL (Data Definition Language) - операторы определения данных
- DML (Data Manipulation Language) - операторы манипуляции данными
- DCL (Data Control Language) - операторы определения доступа к данным
- TCL (Transaction Control Language) - операторы управления транзакциями

SQLявляется стандартизированным языком. Стандартный SQL поддерживается комитетом стандартов ANSI (Американский национальный институт стандартов), и соответственно называется ANSI SQL.

Многие разработчики СУБД расширили возможности SQL, введя в язык дополнительные операторы или инструкции. Эти расширения необходимы для выполнения дополнительных функций или для упрощения выполнения определенных операций. И хотя часто они очень полезны, эти расширения привязаны к определенной СУБД и редко поддерживаются более чем одним разработчиком. Все крупные СУБД и даже те, у которых есть собственные расширения, поддерживают ANSI SQL (в большей или меньшей степени). Отдельные же реализации носят собственные имена (PL-SQL, Transact-SQL и т.д.). Transact-SQL(T-SQL) – реализация языкаSQLкорпорацииMicrosoft, используемая, в частности, и вSQLServer.

Запросы на выборку данных (оператор select)

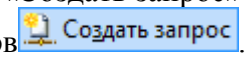
SELECT– наиболее часто используемыйSQLоператор. Он предназначен для выборки информации из таблиц. Чтобы при помощи оператора SELECT извлечь данные из таблицы, нужно указать как минимум две вещи — что вы хотите выбрать и откуда.

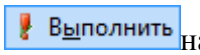
Выборка отдельных столбцов

```
SELECT [Description]
FROM Product
```

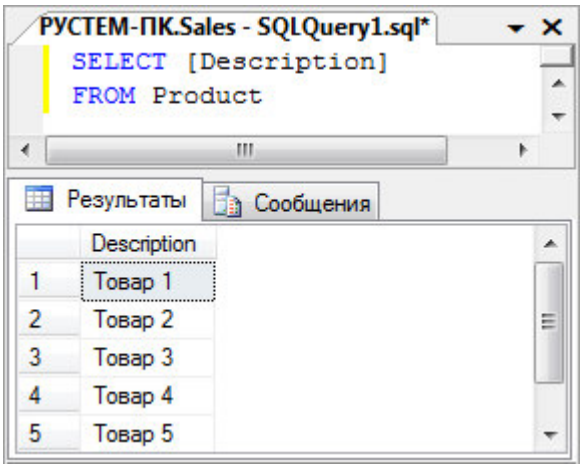
В приведенном выше операторе используется оператор SELECT для выборки одного столбца под названием Description из таблицы Product. Искомое имя столбца указывается сразу после ключевого слова SELECT, а ключевое слово FROM указывает на имя таблицы, из которой выбираются данные.

Для создания и тестирования данного запроса в ManagementStudioвыполните следующие шаги:

1. В контекстном меню базы Salesвыберите команду «Создать запрос» или щелкните соответствующую кнопку на панели инструментов 

.
2. В открывшемся окне создания нового запроса введите представленные выше инструкции SQL.
3. Для запуска запроса на выполнение щелкните кнопку 

на панели инструментов или нажмите клавишуF5. В нижней части экрана должны появиться результаты.



4. ManagementStudioпозволяет сохранять пакетыSQL. Это полезно для сохранения сложных запросов, которые будут повторно запускаться в будущем.

Для этого щелкните кнопку  на панели инструментов. По умолчанию файлы запросов сохраняются с расширением .sql. В дальнейшем сохраненный запрос может быть открыт командой «Открыть файл».

Выборка нескольких столбцов

Для выборки из таблицы нескольких столбцов используется тот же оператор SELECT. Отличие состоит в том, что после ключевого слова SELECT необходимо через запятую указать несколько имен столбцов.

```
SELECT [Description], InStock
FROM Product
```

Выборка всех столбцов

Помимо возможности осуществлять выборку определенных столбцов (одного или нескольких), при помощи оператора SELECT можно запросить все столбцы, не перечисляя каждый из них. Для этого вместо имен столбцов вставляется групповой символ “звездочка” (*). Это делается следующим образом.

```
SELECT *
FROM Product
```

Сортировка данных

В результате выполнения запроса на выборку данные выводятся в том порядке, в котором они находятся в таблице. Для точной сортировки выбранных при помощи оператора SELECT данных используется предложение ORDER BY. В этом предложении указывается имя одного или нескольких столбцов, по которым необходимо отсортировать результаты. Взгляните на следующий пример.

```
SELECT IdProd, [Description], InStock
FROM Product
```

ORDER BY InStock

Это выражение идентично предыдущему, за исключением предложения ORDERBY, которое указывает СУБД отсортировать данные по возрастанию значений столбцаInStock.

Сортировка по нескольким столбцам

Чтобы осуществить сортировку по нескольким столбцам, просто укажите их имена через запятую. В следующем коде выбираются три столбца, а результат сортируется по двум из них — сначала по количеству, а потом по названию.

SELECT IdProd, [Description], InStock
FROM Product
ORDER BY InStock, [Description]

Важно понимать, что при сортировке по нескольким столбцам порядок сортировки будет таким, который указан в запросе. Другими словами, в примере, приведенном выше, продукция сортируется по столбцу Description, только если существует несколько строк с одинаковыми значениямиInStock. Если никакие значения столбцаInStockне совпадают, данные по столбцуDescriptionсортироваться не будут.

Указание направления сортировки

В предложении ORDER BY можно также использовать порядок сортировки по убыванию. Для этого необходимо указать ключевое слово DESC. В следующем примере продукция сортируется по количеству в убывающем порядке плюс по названию продукта.

SELECT IdProd, [Description], InStock
FROM Product
ORDER BY InStock DESC, [Description]

Ключевое слово DESCприменяется только к тому столбцу, после которого оно указано. В предыдущем примере ключевое словоDESCбыло указано для столбцаInStock, но не дляDescription.Таким образом, столбецInStockотсортирован в порядке убывания, а столбецDescriptionв возрастающем порядке (принятым по умолчанию).

Фильтрация данных

В таблицах баз данных обычно содержится много информации и довольно редко возникает необходимость выбирать все строки таблицы. Гораздо чаще бывает нужно извлечь какую-то часть данных таблицы для каких-либо действий или отчетов. Выборка только необходимых данных включает в себя критерий поиска, также известный под названием предложение фильтрации. В операторе SELECT данные фильтруются путем указания критерия поиска в предложении WHERE. Предложение WHERE указывается сразу после названия таблицы (предложения FROM) следующим образом:

SELECT IdProd, [Description], InStock
FROM Product
WHERE InStock = 0

Этот оператор извлекает значения всех столбцов из таблицы товаров, но показывает не все строки, а только те, значение в столбце InStock(Количество товаров на складе) которых равно 0, т.е. только список отсутствующих на складе товаров.

При совместном использовании предложений ORDER BY и WHERE, предложение ORDER BY должно следовать после WHERE.

В предыдущем примере проводилась проверка на равенство, т.е. определялось, содержится ли в столбце указанное значение. SQL поддерживает весь спектр условных (логических) операций, которые приведены в следующей таблице.

Операция	Описание
=	Равенство
<>	Неравенство

!=	Неравенство
<	Меньше
<=	Меньше или равно
!<	Не меньше
>	Больше
>=	Больше или равно
!>	Не больше
BETWEEN	Между двумя указанными значениями
IS NULL	Значение NULL

В следующем примере осуществляется выборка всех клиентов, для которых не указан контактный телефон.

```
SELECT FName, LName, Phone
FROM Customer
WHERE PHONE IS NULL
```

Для поиска диапазона значений можно использовать операцию BETWEEN. Ее синтаксис немного отличается от других операций предложения WHERE, так как для нее требуются два значения: начальное и конечное. Например, операцию BETWEEN можно использовать для поиска товаров, количество которых находится в промежутке между 5 и 10.

```
SELECT IdProd, [Description], InStock
FROM Product
WHERE InStock BETWEEN 5 AND 10
```

Для объединения в предложении WHEREнескольких условий необходимо использовать логические операторыANDи (или)OR. ОператорANDтребует одновременного выполнения обоих условий. Запишем предыдущий запрос посредством объединения двух операции сравнения операторомAND.

```
SELECT IdProd, [Description], InStock
FROM Product
WHERE (InStock >= 5) AND (InStock <= 10)
```

Ключевое слово AND указывает СУБД возвращать только те строки, которые удовлетворяют всем перечисленным критериям отбора. В данном случае будут выбраны только те товары, количество которых находится в промежутке от 5 до 10.

Оператор OR указывает СУБД выбирать только те строки, которые удовлетворяют хотя бы одному из условий.

```
SELECT IdCity, CityName
FROM City
WHERE (CityName = 'Москва') OR (CityName = 'Казань')
```

Посредством этого SQLзапроса из справочника городов выбираются только Москва и Казань. Ключевое словоORуказывает СУБД использовать какое-то одно условие, а не сразу два. Если бы здесь использовалось ключевое словоAND, мы бы не получили никаких данных.

Если вы внимательно рассмотрите выражение в предыдущем предложении WHERE, то заметите, что значения, с которыми сравниваются названия городов, заключены в одинарные кавычки. Одинарные кавычки используются для определения

границ строки (строковой константы). При работе со строковыми константами их всегда необходимо отделять одинарными кавычками.

Предложения WHERE могут содержать любое количество логических операторов AND и OR. Комбинируя их можно создавать сложные фильтры. Однако при комбинировании ключевых слов AND и OR необходимо учитывать, что оператор ANDвыполняется раньше операторaOR, т.е. имеет более высокий приоритет. Изменить приоритет можно с помощью круглых скобок.

В следующем примере осуществляется выборка из таблицы клиентов всех Ивановых и Петровых, для которых не указан контактный телефон.

```
SELECT FName, LName, Phone
FROM Customer
WHERE (LName = 'Иванов' OR LName = 'Петров') AND PHONE IS NULL
```

В случае отсутствия скобок результат был бы не верным, а именно включал бы в себя всех Петровых без контактного телефона и всех Ивановых без каких либо ограничений.

Для определения входит ли сравниваемое значение в определенное заданное множество можно воспользоваться оператором IN. При этом все допустимые значения, заключенные в скобки, перечисляются через запятую. В частности предыдущий пример с использованием операторaINможет быть записан в более компактной форме.

```
SELECT FName, LName, Phone
FROM Customer
WHERE LName IN ('Иванов','Петров') AND PHONE IS NULL
```

Для отрицания какого-то условия используется логический оператор NOT. Поскольку NOT никогда не используется сам по себе (а только вместе с другими логическими операторами), его синтаксис немного отличается от синтаксиса остальных операторов. В отличие от них, NOT вставляется перед названием столбца, значения которого нужно отфильтровать, а не после. В следующем примере отбираются все клиенты, для которых имеются сведения об их контактом телефоне.

```
SELECT FName, LName, Phone
FROM Customer
WHERE NOT PHONE IS NULL
```

Для фильтрации данных по критерию соответствия определенной символьной строки заданному шаблону используется оператор LIKE. Шаблон может включать обычные символы и символы-шаблоны. Во время сравнения с шаблоном необходимо, чтобы его обычные символы в точности совпадали с символами, указанными в строке. Символы-шаблоны могут совпадать с произвольными элементами символьной строки. Использование символов-шаблонов с оператором LIKE предоставляет больше возможностей, чем использование обычных операторов сравнения. Шаблон может включать в себя следующие символы-шаблоны.

Символ-шаблон	Описание	Пример
%	Любое количество символов	Инструкция WHERE FNameLIKE 'A%' выполняет поиск и выдает всех клиентов, имена которых начинаются на букву ‘A’.
_	Любой одиночный символ	Инструкция WHERE LNameLIKE '_етров' выполняет поиск и выдает всех клиентов, фамилии которых состоят из шести букв и заканчиваются сочетанием ‘етров’ (Петров, Ветров и т.п.).
[]	Любой символ, указанный в квадратных скобках	Инструкция WHERE LNameLIKE '[Л-С]омов' выполняет поиск и выдает всех клиентов, фамилии которых заканчиваются на ‘омов’ и начинаются на любую букву в промежутке от ‘Л’ до ‘С’, например Ломов, Ромов, Сомов и т.п.
[^]	Любой символ, кроме перечисленных в квадратных скобках	Инструкция WHERE LNameLIKE 'ив[^а]%' выполняет поиск и выдает всех клиентов, фамилии которых начинаются на ‘ив’ и третья буква отличается от ‘а’.

В следующем примере осуществляется выборка всех товаров, названия которых начинаются на букву Т.

```
SELECT *
FROM Product
WHERE [Description] LIKE 'T%'
```

Создание вычисляемых полей

Конструкция SELECTкроме имен столбцов таблиц может также включать так называемые вычисляемые поля. В отличие от всех выбранных нами ранее столбцов, вычисляемых полей на самом деле в таблицах базы данных нет. Они создаются "на лету" SQL-оператором SELECT. Рассмотрим следующий пример.

```
SELECT IdCust AS 'Номер клиента', FName + ' ' +LName AS 'Фамилия и имя клиента'
FROM Customer
```

Здесь создается вычисляемое поле, которому с помощью ключевого слова ASдан псевдоним ‘Фамилия и имя клиента’. Оно позволяет объединить (произвести конкатенацию) с помощью оператора + фамилию, пробел и имя клиента в одно поле (столбец). Псевдоним может быть задан и для обычного столбца таблицы. В частности здесь столбцуIdCustзадан псевдоним ‘Номер клиента’.

Еще одним способом использования вычисляемых полей является выполнение математических операций над выбранными данными. Рассмотрим пример.

```
SELECT IdProd, Qty, Price, Qty * Price AS 'Стоимость'
FROM OrdItem
WHERE IdOrd = 1
```

Здесь с помощью оператора умножения * вычисляется общая стоимость каждого товара в заказе с кодом 1 как произведение количества на цену.

Исключение дублирующих записей

Для исключения из результата выборки повторяющихся строк используется ключевое слово DISTINCT, которое указывается сразу послеSELECT. В следующем примере осуществляется вывод всех фамилий клиентов. Даже если среди них есть однофамильцы, каждая фамилия будет выведена только один раз.

```
SELECT DISTINCT LName
FROM Customer
```

Задание для самостоятельной работы:Сформулируйте на языкеSQLзапросы на выборку следующих данных:

- Список всех заказов за определенный период времени (например, сентябрь 2010 года) отсортированный по дате заказа;
- Список всех товаров, названия которых включают слово ‘монитор’ с указанием их остатка на складе.

Использование агрегатных функций

В SQL определено множество встроенных функций различных категорий, среди которых особое место занимают агрегатные функции, оперирующие значениями столбцов множества строк и возвращающие одно значение. Аргументами агрегатных функций могут быть как столбцы таблиц, так и результаты выражений над ними. Агрегатные функции и сами могут включаться в другие арифметические выражения. В следующей таблице приведены наиболее часто используемые стандартные унарные агрегатные функции.

Функция	**Возвращаемое значение**
COUNT	Количество значений в столбце или строк в таблице
SUM	Сумма
AVG	Среднее
MIN	Минимум
MAX	Максимум

Общий формат унарной агрегатной функции следующий:

имя_функции([ALL| DISTINCT] выражение)

где DISTINCT указывает, что функция должна рассматривать только различные значения аргумента, а ALL — все значения, включая повторяющиеся (этот вариант используется по умолчанию). Например, функция AVG с ключевым словом DISTINCT для строк столбца со значениями 1, 1, 1 и 3 вернет 2, а при наличии ключевого слова ALL вернет 1,5.

Агрегатные функции применяются во фразах SELECT и HAVING. Здесь мы рассмотрим их использование во фразе SELECT. В этом случае выражение в аргументе функции применяется ко всем строкам входной таблицы фразы SELECT. Кроме того, во фразе SELECT нельзя использовать и агрегатные функции, и столбцы таблицы (или выражения с ними) при отсутствии фразы GROUP BY, которую мы рассмотрим в следующем разделе.

Функция COUNT имеет два формата. В первом случае возвращается количество строк входной таблицы, во втором случае — количество значений аргумента во входной таблице:

- COUNT(*)
- COUNT([DISTINCT | ALL] выражение)

Простейший способ использования этой функции — подсчет количества строк в таблице (всех или удовлетворяющих указанному условию). Для этого используется первый вариант синтаксиса.

Запрос: Количество видов продукции, информация о которых имеется в базе данных.

```
SELECT COUNT(*) AS 'Количество видов продукции'
FROM Product
```

Во втором варианте синтаксиса функции COUNT в качестве аргумента может быть использовано имя отдельного столбца. В этом случае подсчитывается количество либо всех значений в этом столбце входной таблицы, либо только неповторяющихся (при использовании ключевого слова DISTINCT).

Запрос: Количество различных имен, содержащихся в таблице Customer.

```
SELECT COUNT(DISTINCT FNAME)
FROM Customer
```

Использование остальных унарных агрегатных функции аналогично COUNT за тем исключением, что для функций MIN и MAX использование ключевых слов DISTINCT и ALL не имеет смысла. С функциями COUNT, MAX и MIN кроме числовых могут использоваться и символьные поля. Если аргумент агрегатной функции не содержит значений, функция COUNT возвращает 0, а все остальные - значение NULL.

Запрос: Дата последнего заказа до 1 сентября 2010 года.

```
SELECT MAX(OrdDate)
FROM [Order]
WHERE OrdDate<'1.09.2010'
```

Задание для самостоятельной работы: Сформулируйте на языке SQL запросы на выборку следующих данных:

- Суммарная стоимость всех заказов;
- Количество различных городов, содержащихся в таблице Customer.

Запросы с группировкой строк

Описанные выше агрегатные функции применялись ко всей таблице. Однако часто при создании отчетов появляется необходимость в формировании промежуточных итоговых значений, то есть относящихся к данным не всей таблицы, а ее частей. Для этого предназначена фраза GROUP BY. Она позволяет все множество строк таблицы разделить на группы по признаку равенства значений одного или нескольких столбцов (и выражений над ними). Фраза GROUP BY должна располагаться вслед за фразой WHERE (если она отсутствует, то за фразой FROM).

При наличии фразы GROUP BY фраза SELECT применяется к каждой группе, сформированной фразой группировки. В этом случае и действие агрегатных функций, указанных во фразе SELECT, будет распространяться не на всю результирующую таблицу, а только на строки в пределах каждой группы. Каждое выражение в списке фразы SELECT должно принимать единственное значение для группы, то есть оно может быть:

- константой;
- агрегатной функцией, которая оперирует всеми значениями аргумента в пределах группы и агрегирует их в одно значение (например, в сумму);
- выражением, идентичным стоящему во фразе GROUP BY;
- выражением, объединяющим приведенные выше варианты.

Самым простым вариантом использования фразы GROUP BY является группировка по значениям одного столбца.

Запрос Количество клиентов по городам.

```
SELECT IdCity, COUNT(*) AS 'Кол-во клиентов'
FROM Customer
GROUP BY IdCity
```

Если в запросе используются фразы и WHERE, и GROUP BY, строки, не удовлетворяющие условию фразы WHERE, исключаются до выполнения группировки. Вследствие этого группировка производится только по тем строкам, которые удовлетворяют условию.

Запрос: Количество клиентов по городам с фамилией ‘Иванов’.

```
SELECT IdCity, COUNT(*) AS 'Кол-во клиентов'
FROM Customer
WHERE LName = 'Иванов'
GROUP BY IdCity
```

SQL позволяет группировать строки таблицы и по нескольким столбцам. В этом случае имена столбцов перечисляются во фразе GROUP BY через запятую.

Запрос: Количество клиентов по каждой фамилии и имени.

```
SELECT LName, FName, COUNT(*)
FROM Customer
GROUP BY LName, FName
```

Для отбора строк среди полученных групп применяется фраза HAVING. Она играет такую же роль для групп, что и фраза WHERE для исходных таблиц, и может использоваться лишь при наличии фразы GROUP BY. В предложении SELECT фразы WHERE, GROUP BY и HAVING обрабатываются в следующем порядке.

1. Фразой WHERE отбираются строки, удовлетворяющие указанному в ней условию;
2. Фраза GROUP BY группирует отобранные строки;
3. Фразой HAVING отбираются группы, удовлетворяющие указанному в ней условию.

Значение условия, указываемого во фразе HAVING, должно быть уникальным для всех строк каждой группы. Поэтому правила использования имен столбцов и агрегатных функций во фразе HAVING такие же, как и для фразы SELECT при наличии фразы GROUP BY. Это значит, что во фразе HAVING в качестве операндов сравнения можно использовать только группируемые столбцы или агрегатные функции.

Запрос: Список городов, количество клиентов из которых больше 10.

```
SELECT IdCity
FROM Customer
GROUP BY IdCity
HAVING COUNT(*)>10
```

2.3 Лабораторная работа № 5, 6 (4 часа).

Тема: «Построение нетривиальных запросов».

2.3.1 Цель работы: Получить практический опыт написания нетривиальных SQL запросов в базе данных.

2.3.2 Задачи работы:

1. Изучить теоретический материал по написанию нетривиальных SQL запросов;
2. Написать несколько нетривиальных SQL запросов в базе данных на основе MS SQL Server 2005.

2.3.3 Описание (ход) работы:

Понятие подзапроса

Часто невозможно решить поставленную задачу путем одного запроса. Это особенно актуально, когда при использовании условия поиска в предложении WHERE значение, с которым надо сравнивать, заранее не определено и должно быть вычислено в момент выполнения оператора SELECT. В таком случае приходят на помощь законченные операторы SELECT, внедренные в тело другого оператора SELECT. Внутренний подзапрос представляет собой также оператор SELECT, а кодирование его предложений подчиняется тем же правилам, что и основного оператора SELECT. Внешний оператор SELECT использует результат выполнения внутреннего оператора для определения содержания окончательного результата всей операции. Внутренние запросы могут быть помещены непосредственно после оператора сравнения (=, <, >, <=, >=, <>) в предложения WHERE и HAVING внешнего оператора SELECT – они получают название подзапросов или вложенных запросов. Кроме того, внутренние операторы SELECT могут применяться в операторах INSERT, UPDATE и DELETE.

Подзапрос – это инструмент создания временной таблицы, содержимое которой извлекается и обрабатывается внешним оператором. Текст подзапроса должен быть заключен в скобки. К подзапросам применяются следующие правила и ограничения:

- фраза ORDER BY не используется, хотя и может присутствовать во внешнем подзапросе;
- список в предложении SELECT состоит из имен отдельных столбцов или составленных из них выражений – за исключением случая, когда в подзапросе присутствует ключевое слово EXISTS;
- по умолчанию имена столбцов в подзапросе относятся к таблице, имя которой указано в предложении FROM. Однако допускается ссылка и на столбцы таблицы, указанной во фразе FROM внешнего запроса, для чего применяются квалифицированные имена столбцов (т.е. с указанием таблицы);
- если подзапрос является одним из двух операндов, участвующих в операции сравнения, то запрос должен указываться в правой части этой операции.

Существует два типа подзапросов:

- Скалярный подзапрос возвращает единственное значение. В принципе, он может использоваться везде, где требуется указать единственное значение.
- Табличный подзапрос возвращает множество значений, т.е. значения одного или нескольких столбцов таблицы, размещенные в более чем одной строке. Он возможен везде, где допускается наличие таблицы.

Использование подзапросов, возвращающих единичное значение

Пример 7.1. Определить дату продажи максимальной партии товара.

SELECT Дата, Количество FROM Сделка WHERE Количество=(SELECT Max(Количество) FROM Сделка)

Пример 7.1. Определение даты продажи максимальной партии товара.

Во вложенном подзапросе определяется максимальное количество товара. Во внешнем подзапросе – дата, для которой количество товара оказалось равным максимальному. Необходимо отметить, что нельзя прямо использовать предложение WHERE Количество=Max(Количество), поскольку применять обобщающие функции в предложениях WHERE запрещено. Для достижения желаемого результата следует создать подзапрос, вычисляющий максимальное значение количества, а затем использовать его во внешнем операторе SELECT, предназначенном для выборки дат сделок, где количество товара совпало с максимальным значением.

Пример 7.2. Определить даты сделок, превысивших по количеству товара среднее значение и указать для этих сделок превышение над средним уровнем.

SELECT Дата, Количество, Количество-(SELECT Avg(Количество) FROM Сделка) AS Превышение FROM Сделка WHERE Количество> (SELECT Avg(Количество) FROM Сделка)

Пример 7.2. Определение даты сделок, превысивших по количеству товара среднее значение и указать для этих сделок превышение над средним уровнем.

В приведенном примере результат подзапроса, представляющий собой среднее значение количества товара по всем сделкам вообще, используется во внешнем операторе SELECT как для вычисления отклонения количества от среднего уровня, так и для отбора сведений о датах.

Пример 7.3. Определить клиентов, совершивших сделки с максимальным количеством товара.

SELECT Клиент.Фамилия FROM Клиент INNER JOIN Сделка ON Клиент.КодКлиента=Сделка.КодКлиента WHERE Сделка.Количество= (SELECT Max(Сделка.Количество) FROM Сделка)

Пример 7.3. Определение клиентов, совершивших сделки с максимальным количеством товара.

Здесь показан пример использования подзапроса при выборке данных из разных таблиц.

Пример 7.4. Определить клиентов, в сделках которых количество товара отличается от максимального не более чем на 10%.

SELECT Клиент.Фамилия, Сделка.Количество FROM Клиент INNER JOIN Сделка ON Клиент.КодКлиента= Сделка.КодКлиента WHERE Сделка.Количество>=0.9* (SELECT Max(Сделка.Количество) FROM Сделка)

Пример 7.4. Определение клиентов, в сделках которых количество товара отличается от максимального не более чем на 10%.

Покажем, как применяются подзапросы в предложении HAVING.

Пример 7.5. Определить даты, когда среднее количество проданного за день товара оказалось больше 20 единиц.

SELECT Сделка.Дата, Avg(Сделка.Количество) AS Среднее_за_день FROM Сделка GROUP BY Сделка.Дата HAVING Avg(Сделка.Количество)>20

Пример 7.5. Определение даты, когда среднее количество проданного за день товара оказалось больше 20 единиц.

За каждый день определяется среднее количество товара, которое сравнивается с числом 20. Добавим в запрос подзапрос.

Пример 7.6. Определить даты, когда среднее количество проданного за день товара оказалось больше среднего показателя по всем сделкам вообще.

SELECT Сделка.Дата, Avg(Сделка.Количество) AS Среднее_за_день FROM Сделка GROUP BY Сделка.Дата HAVING Avg(Сделка.Количество)> (SELECT Avg(Сделка.Количество) FROM Сделка)

Пример 7.6. Определение даты, когда среднее количество проданного за день товара оказалось больше среднего показателя по всем сделкам вообще.

Внутренний подзапрос определяет средний по всем сделкам показатель, с которым во внешнем запросе сравнивается среднее за каждый день количество товара.

Использование подзапросов, возвращающих множество значений

Во многих случаях значение, подлежащее сравнению в предложениях WHERE или HAVING, представляет собой не одно, а несколько значений. Вложенные подзапросы генерируют непоименованное промежуточное отношение, временную таблицу. Оно может использоваться только в том месте, где появляется в подзапросе. К такому отношению невозможно обратиться по имени из какого-либо другого места запроса. Применяемые к подзапросу операции основаны на тех операциях, которые, в свою очередь, применяются к множеству, а именно:

- { WHERE | HAVING } выражение [NOT] IN (подзапрос);
- { WHERE | HAVING } выражение оператор_сравнения { ALL | SOME | ANY }(подзапрос);
- {WHERE | HAVING } [NOT] EXISTS (подзапрос);

Использование операций IN и NOT IN

Оператор IN используется для сравнения некоторого значения со списком значений, при этом проверяется, входит ли значение в предоставленный список или сравниваемое значение не является элементом представленного списка.

Пример 7.7. Определить список товаров, которые имеются на складе.

SELECT Название FROM Товар WHERE КодТовара In (SELECT КодТовара FROM Склад)

Пример 7.7. Определение списка товаров, которые имеются на складе.

Пример 7.8. Определить список отсутствующих на складе товаров.

SELECT Название FROM Товар WHERE КодТовара Not In (SELECT КодТовара FROM Склад)

Пример 7.8. Определение списка отсутствующих на складе товаров.

Пример 7.9. Определить товары, которые покупают клиенты из Москвы.

SELECT DISTINCT Товар.Название, Клиент.ГородКлиента FROM Товар INNER JOIN (Клиент INNER JOIN Сделка ON Клиент.КодКлиента=Сделка.КодКлиента) ON Товар.КодТовара=Сделка.КодТовара WHERE Клиент.ГородКлиента='Москва'

Пример 7.9. Определение товаров, которые покупают клиенты из Москвы.

В результат включаются товары, приобретенные клиентами из Москвы, однако не исключено, что покупателями таких товаров были и клиенты из других городов.

Введение в запрос фразы "только" требует использования операции NOT IN.

Пример 7.10. Определить товары, покупку которых осуществляют только клиенты из Москвы, и никто другой.

SELECT DISTINCT Товар.Название, Клиент.ГородКлиента FROM Товар INNER JOIN (Клиент INNER JOIN Сделка ON Клиент.КодКлиента=Сделка.КодКлиента) ON Товар.КодТовара=Сделка.КодТовара WHERE Товар.Название NOT IN (SELECT Товар.Название FROM Товар INNER JOIN (Клиент INNER JOIN Сделка ON Клиент.КодКлиента=Сделка.КодКлиента) ON Товар.КодТовара=Сделка.КодТовара WHERE Клиент.ГородКлиента<>'Москва')

Пример 7.10. Определение товаров, покупку которых осуществляют только клиенты из Москвы, и никто другой.

Пример 7.11. Какие товары ни разу не купили московские клиенты?

SELECT DISTINCT Товар.Название, Клиент.ГородКлиента FROM Товар INNER JOIN (Клиент INNER JOIN Сделка ON Клиент.КодКлиента=Сделка.КодКлиента) ON Товар.КодТовара=Сделка.КодТовара WHERE Товар.Название NOT IN (SELECT Товар.Название FROM Товар INNER JOIN (Клиент INNER JOIN Сделка ON Клиент.КодКлиента=Сделка.КодКлиента) ON Товар.КодТовара=Сделка.КодТовара WHERE Клиент.ГородКлиента='Москва')

Пример 7.11. Определение товаров, которые ни разу не купили московские клиенты?

Во вложенном запросе определяется список товаров, приобретаемых клиентами из Москвы. Во внешнем запросе выбираются только те товары, которые не входят в этот список.

Пример 7.12. Определить фирмы, покупающие товары местного производства.

SELECT DISTINCT Клиент.Фирма, Клиент.ГородКлиента, Товар.ГородТовара FROM Товар INNER JOIN (Клиент INNER JOIN Сделка ON Клиент.КодКлиента=Сделка.КодКлиента) ON Товар.КодТовара=Сделка.КодТовара WHERE Клиент.ГородКлиента=Товар.ГородТовара

Пример 7.12. Определение фирм, покупающих товары местного производства.

В результате выполнения запроса перечисляются сделки, когда клиенту был продан товар, изготовленный в его городе, что совсем не исключает наличие сделок этих же клиентов, связанных с приобретением товара из другого города.

Введем в запрос фразу "только" – сразу потребуется привлечение операции NOT IN.

Пример 7.13. Определить фирмы, которые покупают только товары, произведенные в своем городе, и никакие другие.

SELECT DISTINCT Клиент.Фирма, Клиент.ГородКлиента, Товар.ГородТовара FROM Товар INNER JOIN (Клиент INNER JOIN Сделка ON Клиент.КодКлиента=Сделка.КодКлиента) ON Товар.КодТовара=Сделка.КодТовара WHERE Клиент.ГородКлиента NOT IN (SELECT DISTINCT Клиент.ГородКлиента FROM Товар INNER JOIN (Клиент INNER JOIN Сделка ON Клиент.КодКлиента=Сделка.КодКлиента) ON Товар.КодТовара=Сделка.КодТовара WHERE Клиент.ГородКлиента<> Товар.ГородТовара)

Пример 7.13. Определение фирм, которые покупают только товары, произведенные в своем городе, и никакие другие.

Во вложенном запросе определяется множество фирм, совершивших хотя бы одну покупку товара из чужого города. Затем определяются фирмы, не входящие в это множество.

Использование ключевых слов ANY и ALL

Ключевые слова ANY и ALL могут использоваться с подзапросами, возвращающими один столбец чисел.

Если подзапросу будет предшествовать ключевое слово ALL, условие сравнения считается выполненным, только когда оно выполняется для всех значений в результирующем столбце подзапроса.

Если записи подзапроса предшествует ключевое слово ANY, то условие сравнения считается выполненным, когда оно выполняется хотя бы для одного из значений в результирующем столбце подзапроса.

Если в результате выполнения подзапроса получено пустое значение, то для ключевого слова ALL условие сравнения будет считаться выполненным, а для ключевого слова ANY – невыполненным. Ключевое слово SOME является синонимом слова ANY.

Пример 7.14. Определить клиентов, совершивших сделки с максимальным количеством товара (эквивалентно запросу 7.3.)

SELECT Клиент.Фамилия, Сделка.Количество FROM Клиент INNER JOIN Сделка ON Клиент.КодКлиента=Сделка.КодКлиента WHERE Сделка.Количество>=ALL(SELECT Количество FROM Сделка)

Пример 7.14. Определение клиентов, совершивших сделки с максимальным количеством товара.

В примере определены клиенты, в сделках которых количество товара больше или равно количеству товара в каждой из всех сделок.

Пример 7.15. Найти фирму, купившую товаров на сумму, превышающую 10000 руб.

SELECT Клиент.Фирма, Sum(Товар.Цена*Сделка.Количество) AS Общ_стоимость FROM Товар INNER JOIN (Клиент INNER JOIN Сделка ON Клиент.КодКлиента=Сделка.КодКлиента) ON Товар.КодТовара=Сделка.КодТовара GROUP BY Клиент.Фирма HAVING Sum(Товар.Цена*Сделка.Количество)>10000

Пример 7.15. Определение фирмы, купившей товаров на сумму, превышающую 10000 руб.

Добавим в запрос подзапрос.

Пример 7.16. Найти фирму, которая приобрела товаров на самую большую сумму.

SELECT Клиент.Фирма, Sum(Товар.Цена*Сделка.Количество) AS Общ_стоимость FROM Товар INNER JOIN (Клиент INNER JOIN Сделка ON Клиент.КодКлиента=Сделка.КодКлиента) ON Товар.КодТовара=Сделка.КодТовара GROUP BY Клиент.Фирма HAVING Sum(Товар.Цена*Сделка.Количество)>=

ALL(SELECT Sum(Товар.Цена*Сделка.Количество) FROM Товар INNER JOIN Сделка ON Товар.КодТовара=Сделка.КодТовара GROUP BY Сделка.КодКлиента)

Пример 7.16. Определение фирмы, которая приобрела товаров на самую большую сумму.

Вложенный подзапрос подсчитывает общую стоимость покупок каждого клиента. Внешний подзапрос также подсчитывает общую стоимость покупок каждого клиента и определяет тех, для кого эта сумма, по сравнению с другими покупателями, оказалась больше или точно такой же.

Пример 7.17. Найти фирмы, в сделках которых количество товара превышает такой же показатель хотя бы в одной сделке клиентов из Самары.

SELECT Клиент.Фирма, Сделка.Количество FROM Клиент INNER JOIN Сделка ON Клиент.КодКлиента=Сделка.КодКлиента WHERE Сделка.Количество> ANY(SELECT Сделка.Количество FROM Клиент INNER JOIN Сделка ON Клиент.КодКлиента=Сделка.КодКлиента WHERE Клиент.ГородКлиента='Самара')

Пример 7.17. Определение фирм, в сделках которых количество товара превышает такой же показатель хотя бы в одной сделке клиентов из Самары.

Использование операций EXISTS и NOT EXISTS

Ключевые слова EXISTS и NOT EXISTS предназначены для использования только совместно с подзапросами. Результат их обработки представляет собой логическое значение TRUE или FALSE. Для ключевого слова EXISTS результат равен TRUE в том и только в том случае, если в возвращаемой подзапросом результирующей таблице присутствует хотя бы одна строка. Если результирующая таблица подзапроса пуста, результатом обработки операции EXISTS будет значение FALSE. Для ключевого слова NOT EXISTS используются правила обработки, обратные по отношению к ключевому слову EXISTS. Поскольку по ключевым словам EXISTS и NOT EXISTS проверяется лишь наличие строк в результирующей таблице подзапроса, то эта таблица может содержать произвольное количество столбцов.

Пример 7.18. Определить список имеющихся на складе товаров (запрос эквивалентен примеру 7.7).

SELECT Название FROM Товар WHERE EXISTS (SELECT КодТовара FROM Склад WHERE Товар.КодТовара=Склад.КодТовара)

Пример 7.18. Определение списка имеющихся на складе товаров.

Пример 7.19. Определить список отсутствующих на складе товаров (запрос эквивалентен примеру 7.8).

```
SELECT Название
FROM Товар
WHERE NOT EXISTS (SELECT КодТовара
   FROM Склад
WHERE Товар.КодТовара=Склад.КодТовара)
```

Пример 7.19. Определение списка отсутствующих на складе товаров.

2.4 Лабораторная работа № 7, 8 (4 часа).

Тема: «Запросы модификации данных в реляционной таблице».

2.4.1 Цель работы: Получить практический опыт написания SQL запросов модификации данных в реляционной таблице.

2.4.2 Задачи работы:

1.Изучить теоретический материал по написанию SQL запросов модификации данных в реляционной таблице;

2. Написать несколько SQL запросов модификации данных в реляционной таблице на основе MS SQL Server 2005.

2.4.3 Описание (ход) работы:

Запросы действия представляют собой достаточно мощное средство, так как позволяют оперировать не только отдельными строками, но и набором строк. С помощью запросов действия пользователь может добавить, удалить или обновить блоки данных. Существует три вида запросов действия:

- INSERT INTO – запрос добавления;
- DELETE – запрос удаления;
- UPDATE – запрос обновления.

Запрос добавления

Оператор INSERT применяется для добавления записей в таблицу. Формат оператора:

<оператор_вставки>::=INSERT INTO <имя_таблицы>
[(имя_столбца [,...n])]
{VALUES (значение[,...n])|
<SELECT_оператор>}

Здесь параметр имя_таблицы представляет собой либо имя таблицы базы данных, либо имя обновляемого представления.

Первая форма оператора INSERT с параметром VALUES предназначена для вставки единственной строки в указанную таблицу. Список столбцов указывает столбцы, которым будут присвоены значения в добавляемых записях. Список может быть опущен, тогда подразумеваются все столбцы таблицы (кроме объявленных как счетчик), причем в определенном порядке, установленном при создании таблицы. Если в операторе INSERT указывается конкретный список имен полей, то любые пропущенные в нем столбцы должны быть объявлены при создании таблицы как допускающие значение NULL, за исключением тех случаев, когда при описании столбца использовался параметр DEFAULT. Список значений должен следующим образом соответствовать списку столбцов:

- количество элементов в обоих списках должно быть одинаковым;
- должно существовать прямое соответствие между позицией одного и того же элемента в обоих списках, поэтому первый элемент списка значений должен относиться к первому столбцу в списке столбцов, второй – ко второму столбцу и т.д.
- типы данных элементов в списке значений должны быть совместимы с типами данных соответствующих столбцов таблицы.

Пример 8.1. Добавить в таблицу ТОВАР новую запись.

INSERT INTO Товар (Название, Тип, Цена)
VALUES (" Славянский ", " шоколад ", 12)

Пример 8.1. Добавление в таблицу ТОВАР новой записи.

Если столбцы таблицы ТОВАР указаны в полном составе и в том порядке, в котором они перечислены при создании таблицы ТОВАР, оператор можно упростить.

```
INSERT INTO Товар VALUES (" Славянский ",
   " шоколад ", 12)
```

Вторая форма оператора INSERT с параметром SELECT позволяет скопировать множество строк из одной таблицы в другую. Предложение SELECT может представлять собой любой допустимый оператор SELECT. Вставляемые в указанную таблицу строки в точности должны соответствовать строкам результирующей таблицы, созданной при выполнении вложенного запроса. Все ограничения, указанные выше для первой формы оператора SELECT, применимы и в этом случае.

Поскольку оператор SELECT в общем случае возвращает множество записей, то оператор INSERT в такой форме приводит к добавлению в таблицу аналогичного числа новых записей.

Пример 8.2. Добавить в итоговую таблицу сведения об общей сумме ежемесячных продаж каждого наименования товара.

```
INSERT INTO Итог
  (Название, Месяц, Стоимость )
SELECT Товар.Название, Month(Сделка.Дата)
  AS Месяц, Sum(Товар.Цена*Сделка.Количество)
  AS Стоимость
FROM Товар INNER JOIN Сделка
  ON Товар.КодТовара= Сделка.КодТовара
GROUP BY Товар.Название, Month(Сделка.Дата)
```

Пример 8.2. Добавление в итоговую таблицу сведения об общей сумме ежемесячных продаж каждого наименования товара.

Запрос удаления

Оператор DELETE предназначен для удаления группы записей из таблицы.

Формат оператора:

```
<оператор_удаления> ::=DELETE
   FROM <имя_таблицы>[WHERE <условие_отбора>]
```

Здесь параметр имя_таблицы представляет собой либо имя таблицы базы данных, либо имя обновляемого представления.

Если предложение WHERE присутствует, удаляются записи из таблицы, удовлетворяющие условию отбора. Если опустить предложение WHERE, из таблицы будут удалены все записи, однако сама таблица сохранится.

Пример 8.3. Удалить все прошлогодние сделки.

```
DELETE
FROM Сделка
WHERE Year(Сделка.Дата)=Year(GETDATE())-1
```

Пример 8.3. Удаление всех прошлогодних сделок.

В приведенном примере условие отбора формируется с учетом года (функция Year) от текущей даты (функция GETDATE()).

Запрос обновления

Оператор UPDATE применяется для изменения значений в группе записей или в одной записи указанной таблицы.

Формат оператора:

```
<оператор_изменения> ::=
   UPDATE имя_таблицы SET имя_столбца=
      <выражение>[,...n]
   [WHERE <условие_отбора>]
```

Параметр имя_таблицы – это либо имя таблицы базы данных, либо имя обновляемого представления. В предложении SET указываются имена одного и более столбцов, данные в которых необходимо изменить. Предложение WHERE является необязательным. Если оно опущено, значения указанных столбцов будут изменены во

всех строках таблицы. Если предложение WHERE присутствует, то обновлены будут только те строки, которые удовлетворяют условию отбора. Выражение представляет собой новое значение соответствующего столбца и должно быть совместимо с ним по типу данных.

Пример 8.4. Для товаров первого сорта установить цену в значение 140 и остаток – в значение 20 единиц.

```
UPDATE Товар SET Товар.Цена=140, Товар.Остаток=20
WHERE Товар.Сорт=" Первый "
```

Пример 8.4. Обновление выбранных записей.

Пример 8.5. Увеличить цену товаров первого сорта на 25%.

```
UPDATE Товар SET Товар.Цена=Товар.Цена*1.25
WHERE Товар.Сорт=" Первый "
```

Пример 8.5. Обновление выбранных записей.

Пример 8.6. В сделке с максимальным количеством товара увеличить число товаров на 10%.

```
UPDATE Сделка SET Сделка.Количество=
   Сделка.Количество*1.1
WHERE Сделка.Количество=
   (SELECT Max(Сделка.Количество) FROM Сделка)
```

Пример 8.6. Обновление выбранных записей.

Введение в понятие "целостность данных"

Выполнение операторов модификации данных в таблицах базы данных INSERT, DELETE и UPDATE может привести к нарушению целостности данных и их корректности, т.е. к потере их достоверности и непротиворечивости.

Чтобы информация, хранящаяся в базе данных, была однозначной и непротиворечивой, в реляционной модели устанавливаются некоторые ограничительные условия – правила, определяющие возможные значения данных и обеспечивающие логическую основу для поддержания корректных значений. Ограничения целостности позволяют свести к минимуму ошибки, возникающие при обновлении и обработке данных.

В базе данных, построенной на реляционной модели, задается ряд правил целостности, которые, по сути, являются ограничениями для всех допустимых состояний базы данных и гарантируют корректность данных. Рассмотрим следующие типы ограничений целостности данных:

- обязательные данные;
- ограничения для доменов полей;
- корпоративные ограничения;
- целостность сущностей;
- ссылочная целостность.

Обязательные данные

Некоторые поля всегда должны содержать одно из допустимых значений, другими словами, эти поля не могут иметь пустого значения.

Ограничения для доменов полей

Каждое поле имеет свой домен, представляющий собой набор его допустимых значений.

Корпоративные ограничения целостности

Существует понятие "корпоративные ограничения целостности" как дополнительные правила поддержки целостности данных, определяемые пользователями, принятые на предприятии или администраторами баз данных. Ограничения предприятия называются бизнес-правилами.

Целостность сущностей

Это ограничение целостности касается первичных ключей базовых таблиц. По определению, первичный ключ – минимальный идентификатор (одно или несколько полей), который используется для уникальной идентификации записей в таблице. Таким образом, никакое подмножество первичного ключа не может быть достаточным для уникальной идентификации записей.

Целостность сущностей определяет, что в базовой таблице ни одно поле первичного ключа не может содержать отсутствующих значений, обозначенных NULL.

Если допустить присутствие определителя NULL в любой части первичного ключа, это равносильно утверждению, что не все его поля необходимы для уникальной идентификации записей, и противоречит определению первичного ключа.

Ссылочная целостность

Указанное ограничение целостности касается внешних ключей. Внешний ключ – это поле (или множество полей) одной таблицы, являющееся ключом другой (или той же самой) таблицы. Внешние ключи используются для установления логических связей между таблицами. Связь устанавливается путем присвоения значений внешнего ключа одной таблицы значениям ключа другой.

Между двумя или более таблицами базы данных могут существовать отношения подчиненности, которые определяют, что для каждой записи главной таблицы (называемой еще родительской) может существовать одна или несколько записей в подчиненной таблице (называемой также дочерней).

Существует три разновидности связи между таблицами базы данных:

- "один-ко-многим";
- "один-к-одному";
- "многие-ко-многим".

Отношение "один-ко-многим" имеет место, когда одной записи родительской таблицы может соответствовать несколько записей дочерней. Связь "один-ко-многим" иногда называют связью "многие-к-одному". И в том, и в другом случае сущность связи между таблицами остается неизменной.

Связь "один-ко-многим" наиболее распространена для реляционных баз данных. Она позволяет моделировать также иерархические структуры данных.

Отношение "один-к-одному" имеет место, когда одной записи в родительской таблице соответствует одна запись в дочерней. Это отношение встречается намного реже, чем отношение "один-ко-многим". Его используют, если не хотят, чтобы таблица БД "распухала" от второстепенной информации. Использование связи "один-к-одному" приводит к тому, что для чтения связанной информации в нескольких таблицах приходится производить несколько операций чтения вместо одной, когда данные хранятся в одной таблице.

Отношение "многие-ко-многим" имеет место в следующих случаях:

- одной записи в родительской таблице соответствует более одной записи в дочерней таблице;
- одной записи в дочерней таблице соответствует более одной записи в родительской таблице.

Считается, что всякая связь "многие-ко-многим" может быть заменена на связь "один-ко-многим" (одну или несколько).

Часто связь между таблицами устанавливается по первичному ключу, т.е. значениям внешнего ключа одной таблицы присваиваются значения первичного ключа другой. Однако это не является обязательным – в общем случае связь может устанавливаться и с помощью вторичных ключей. Кроме того, при установлении связей между таблицами не требуется непременная уникальность ключа, обеспечивающего установление связи. Поля внешнего ключа не обязаны иметь те же имена, что и имена ключей, которым они соответствуют. Внешний ключ может ссылаться на свою собственную таблицу – в таком случае внешний ключ называется рекурсивным.

Ссылочная целостность определяет: если в таблице существует внешний ключ, то его значение должно либо соответствовать значению первичного ключа некоторой записи в базовой таблице, либо задаваться определителем NULL.

Существует несколько важных моментов, связанных с внешними ключами. Во-первых, следует проанализировать, допустимо ли использование во внешних ключах пустых значений. В общем случае, если участие дочерней таблицы в связи является обязательным, то рекомендуется запрещать применение пустых значений в соответствующем внешнем ключе. В то же время, если имеет место частичное участие дочерней таблицы в связи, то помещение пустых значений в поле внешнего ключа должно быть разрешено. Например, если в операции фиксации сделок некоторой торговой фирмы необходимо указать покупателя, то поле КодКлиента должно иметь атрибут NOT NULL. Если допускается продажа или покупка товара без указания клиента, то для поля КодКлиента можно указать атрибут NULL.

Следующая проблема связана с организацией поддержки ссылочной целостности при выполнении операций модификации данных в базе. Здесь возможны следующие ситуации:

1. Вставка новой строки в дочернюю таблицу. Для обеспечения ссылочной целостности необходимо убедиться, что значение внешнего ключа новой строки дочерней таблицы равно пустому значению либо некоторому конкретному значению, присутствующему в поле первичного ключа одной из строк родительской таблицы.
2. Удаление строки из дочерней таблицы. Никаких нарушений ссылочной целостности не происходит.
3. Обновление внешнего ключа в строке дочерней таблицы. Этот случай подобен описанной выше первой ситуации. Для сохранения ссылочной целостности необходимо убедиться, что значение внешнего ключа в обновленной строке дочерней таблицы равно пустому значению либо некоторому конкретному значению, присутствующему в поле первичного ключа одной из строк родительской таблицы.
4. Вставка строки в родительскую таблицу. Такая вставка не может вызвать нарушения ссылочной целостности. Добавленная строка просто становится родительским объектом, не имеющим дочерних объектов.
5. Удаление строки из родительской таблицы. Ссылочная целостность окажется нарушенной, если в дочерней таблице будут существовать строки, ссылающиеся на удаленную строку родительской таблицы. В этом случае может использоваться одна из следующих стратегий:

- NO ACTION. Удаление строки из родительской таблицы запрещается, если в дочерней таблице существует хотя бы одна ссылающаяся на нее строка.
- CASCADE. При удалении строки из родительской таблицы автоматически удаляются все ссылающиеся на нее строки дочерней таблицы. Если любая из удаляемых строк дочерней таблицы выступает в качестве родительской стороны в какой-либо другой связи, то операция удаления применяется ко всем строкам дочерней таблицы этой связи и т.д. Другими словами, удаление строки родительской таблицы автоматически распространяется на любые дочерние таблицы.
- SET NULL. При удалении строки из родительской таблицы во всех ссылающихся на нее строках дочернего отношения в поле внешнего ключа, соответствующего первичному ключу удаленной строки, записывается пустое значение. Следовательно, удаление строк из родительской таблицы вызовет занесение пустого значения в соответствующее поле дочерней таблицы. Эта стратегия может использоваться, только когда в поле внешнего ключа дочерней таблицы разрешается помещать пустые значения.
- SET DEFAULT. При удалении строки из родительской таблицы в поле внешнего ключа всех ссылающихся на нее строк дочерней таблицы автоматически помещается значение, указанное для этого поля как значение по умолчанию. Таким

образом, удаление строки из родительской таблицы вызывает помещение принимаемого по умолчанию значения в поле внешнего ключа всех строк дочерней таблицы, ссылающихся на удаленную строку. Эта стратегия применима лишь в тех случаях, когда полю внешнего ключа дочерней таблицы назначено некоторое значение, принимаемое по умолчанию.

- NO CHECK. При удалении строки из родительской таблицы никаких действий по сохранению ссылочной целостности данных не предпринимается.

6. Обновление первичного ключа в строке родительской таблицы. Если значение первичного ключа некоторой строки родительской таблицы будет обновлено, нарушение ссылочной целостности случится при том условии, что в дочернем отношении существуют строки, ссылающиеся на исходное значение первичного ключа. Для сохранения ссылочной целостности может применяться любая из описанных выше стратегий. При использовании стратегии CASCADE обновление значения первичного ключа в строке родительской таблицы будет отображено в любой строке дочерней таблицы, ссылающейся на данную строку.

Существует и другой вид целостности – смысловая (семантическая) целостность базы данных. Требование смысловой целостности определяет, что данные в базе данных должны изменяться таким образом, чтобы не нарушалась сложившаяся между ними смысловая связь.

Уровень поддержания целостности данных в разных системах существенно варьируется.

Идеология архитектуры клиент-сервер требует переноса максимально возможного числа правил целостности данных на сервер. К преимуществам такого подхода относятся:

- гарантия целостности базы данных, поскольку все правила сосредоточены в одном месте (в базе данных);
- автоматическое применение определенных на сервере ограничений целостности для любых приложений;
- отсутствие различных реализаций ограничений в разных клиентских приложениях, работающих с базой данных;
- быстрое срабатывание ограничений, поскольку они реализованы на сервере и, следовательно, нет необходимости посылать данные клиенту, увеличивая при этом сетевой трафик;
- доступность внесенных в ограничения на сервере изменений для всех клиентских приложений, работающих с базой данных, и отсутствие необходимости повторного распространения измененных приложений клиентов среди пользователей.

К недостаткам хранения ограничений целостности на сервере можно отнести:

- отсутствие у клиентского приложения возможности реагировать на некоторые ошибочные ситуации, возникающие на сервере при реализации тех или иных правил (например, ошибок при выполнении хранимых процедур на сервере);
- ограниченность возможностей языка SQL и языка хранимых процедур и триггеров для реализации всех возникающих потребностей определения целостности данных.

На практике в клиентских приложениях реализуют лишь такие правила, которые тяжело или невозможно реализовать с применением средств сервера. Все остальные ограничения целостности данных переносятся на сервер.

2.5 Лабораторная работа № 9, 10 (4 часа).

Тема: «Понятие представлений».

2.5.1 Цель работы: Получить практический опыт написания SQL представлений в базе данных.

2.5.2 Задачи работы:

1. Изучить теоретический материал по написанию SQL представлений;

2. Написать несколько SQL представлений в базе данных на основе MS SQL Server 2005.

2.5.3 Описание (ход) работы:

Представления – это виртуальные таблицы, но они могут быть доступны многим пользователям и существуют в базе данных до тех пор, пока не будут принудительно удалены. Они во всем похожи на обычные таблицы базы данных, за исключением того, что не являются физическими объектами хранения данных.

Данные в представлениях выбираются из таблиц, т.е. представляются в том или ином виде. Они применяются, чтобы скрыть от пользователя некоторые столбцы, скомбинировать из нескольких таблиц одну, которая часто нужна пользователю, а запрос для неё очень сложен. Таким образом, представления используются как надстроечные средства для адаптации базы данных к различным категориям пользователей.

Представления создаются с помощью оператора ***CREATEVIEW*** (создать вид, представление).

CREATEVIEW<*имя представления*>AS<*запрос*>

Пример 1. Создать представление, которое выводит фамилии и соответствующие оценки студентов.

CREATE VIEW OCENKI AS

SELECTDANNIE.FAM, USPEV.OCENKA FROM DANNIE, USPEV WHERE DANNIE.KOD_STUDENT = USPEV. KOD_STUDENT.

Создана виртуальная таблица *OCENKI,* к которой можно обращаться с запросами как к обычной таблице.

Пример 2. Вывести из представления *OCENKI*, только фамилии и хорошие оценки.

*SELECT * FROM OCENKI WHERE OCENKA IN (4,5)*

Рассмотренное представление является многотабличным, поскольку создано на основе не одной, а двух таблиц. На практике используются более простые однотабличные представления, в которых скрываются некоторые столбцы и/или добавляются, значения которых вычисляются.

Пример 3. Создать представление Rod, в котором будут отображены фамилии родителей и их телефоны.

CREATE VIEW ROD AS SELECT FIO_ROD AS FIO, TEL FROM RODITELI.

Название представлений и таблиц не должны совпадать. Данное представление можно заменить обычным запросом

SELECT FIO_ROD AS FIO, TEL FROM RODITELI

Однако относительно полученного набора данных нельзя задать какой–нибудь запрос, поскольку этот набор является виртуальной таблицей, отличной от представления.

Оператор *CREATEVIEW*допускает и такую форму синтаксиса:

CREATEVIEW<*имя представления*> (<*столбец1*>, <*столбец2*>,…, <*столбецN*>) AS <*запрос*>

Пример 4. Создать представление, которое выводит фамилию и дату рождения студентов.

CREATE VIEW DATE (FIO, DATE) AS SELECT FAM, DATE_ROGNEN FROM DANNIE

В представлении указываются имена столбцов, которые могут быть отличны от имён столбцов в таблице.

Удаление представления осуществляется командой:

DROPVIEW<*имя представления*>

Порядок выполнения работы:

1.Создать представление DAN, которое выводит фамилию, паспортные данные студента, название улицы, на которой проживает студент.

2.В представлении DAN вывести отсортировать данные о студентах по улицам в алфавитном порядке.

3.В представлении DAN вывести студентов, проживающих на улицах, начинающих на букву К.

4.Создать представление MINMAX, которое выводит фамилию студента, минимальную, максимальную оценку.

5.В представлении MINMAX найти среднее минимальное и максимальное значение оценок.

6.В представлении MINMAX найти среднее минимальное и максимальное значение оценок для каждого студента.

7.Создать представление OBUCH (fio, gruppa, spec) , включающее поля фамилия студента, название группы, название специальности.

8.В представлении OBUCH подсчитать количество студентов в каждой группе.

9.Создатьпредставление ADRES (fio, region, gorod, ulica, dom, kvart).

10. В представление ADRES вывести студентов по городам. Вывести количество студентов проживающих в разных городах.

11. В представление ADRES вывести студентов по регионам. Вывести количество студентов проживающих в разных регионах.

12. Вывести студентов проживающих в одном городе, на одной улице.

2.6 Лабораторная работа № 11, 12 (4 часа).

Тема: «Определение функций пользователя, примеры их создания и использования».

2.6.1 Цель работы: Определить функций пользователя, написать на языке SQL примеры их создания и использования.

2.6.2 Задачи работы:

1. Изучить теоретический материал по написанию SQL функций пользователя;
2. Написать несколько примеров создания и использования SQL функций в базе данных на основе MS SQL Server 2005.

2.6.3 Описание (ход) работы:

Понятие функции пользователя

При реализации на языке SQL сложных алгоритмов, которые могут потребоваться более одного раза, сразу встает вопрос о сохранении разработанного кода для дальнейшего применения. Эту задачу можно было бы реализовать с помощью хранимых процедур, однако их архитектура не позволяет использовать процедуры непосредственно в выражениях, т.к. они требуют промежуточного присвоения возвращенного значения переменной, которая затем и указывается в выражении. Естественно, подобный метод применения программного кода не слишком удобен. Многие разработчики уже давно хотели иметь возможность вызова разработанных алгоритмов непосредственно в выражениях.

Возможность создания пользовательских функций была предоставлена в среде MS SQL Server 2000. В других реализациях SQL в распоряжении пользователя имеются только встроенные функции, которые обеспечивают выполнение наиболее распространенных алгоритмов: поиск максимального или минимального значения и др.

Функции пользователя представляют собой самостоятельные объекты базы данных, такие, например, как хранимые процедуры или триггеры. Функция пользователя располагается в определенной базе данных и доступна только в ее контексте.

В SQL Server имеются следующие классы функций пользователя:

- Scalar – функции возвращают обычное скалярное значение, каждая может включать множество команд, объединяемых в один блок с помощью конструкции BEGIN...END;
- Inline – функции содержат всего одну команду SELECT и возвращают пользователю набор данных в виде значения типа данных TABLE;
- Multi-statement – функции также возвращают пользователю значение типа данных TABLE, содержащее набор данных, однако в теле функции находится множество команд SQL (INSERT, UPDATE и т.д.). Именно с их помощью и формируется набор данных, который должен быть возвращен после выполнения функции.

Пользовательские функции сходны с хранимыми процедурами, но, в отличие от них, могут применяться в запросах так же, как и системные встроенные функции. Пользовательские функции, возвращающие таблицы, могут стать альтернативой просмотрам. Просмотры ограничены одним выражением SELECT, а пользовательские функции способны включать дополнительные выражения, что позволяет создавать более сложные и мощные конструкции.

Функции Scalar

Создание и изменение функции данного типа выполняется с помощью команды:

```
<определение_скаляр_функции>::=
{CREATE | ALTER } FUNCTION [владелец.]
   имя_функции
( [ { @имя_параметра скаляр_тип_данных
   [=default]}[,...n]])
RETURNS скаляр_тип_данных
```

```
[WITH {ENCRYPTION | SCHEMABINDING}
   [,...n] ]
[AS]
BEGIN
<тело_функции>
RETURN скаляр_выражение
END
```

Рассмотрим назначение параметров команды.

Функция может содержать один или несколько входных параметров либо не содержать ни одного. Каждый параметр должен иметь уникальное в пределах создаваемой функции имя и начинаться с символа "@". После имени указывается тип данных параметра. Дополнительно можно указать значение, которое будет автоматически присваиваться параметру (DEFAULT), если пользователь явно не указал значение соответствующего параметра при вызове функции.

С помощью конструкции RETURNS скаляр_тип_данных указывается, какой тип данных будет иметь возвращаемое функцией значение.

Дополнительные параметры, с которыми должна быть создана функция, могут быть указаны посредством ключевого слова WITH. Благодаря ключевому слову ENCRYPTION код команды, используемый для создания функции, будет зашифрован, и никто не сможет просмотреть его. Эта возможность позволяет скрыть логику работы функции. Кроме того, в теле функции может выполняться обращение к различным объектам базы данных, а потому изменение или удаление соответствующих объектов может привести к нарушению работы функции. Чтобы избежать этого, требуется запретить внесение изменений, указав при создании этой функции ключевое слово SCHEMABINDING.

Между ключевыми словами BEGIN...END указывается набор команд, они и будут являться телом функции.

Когда в ходе выполнения кода функции встречается ключевое слово RETURN, выполнение функции завершается и как результат ее вычисления возвращается значение, указанное непосредственно после слова RETURN. Отметим, что в теле функции разрешается использование множества команд RETURN, которые могут возвращать различные значения. В качестве возвращаемого значения допускаются как обычные константы, так и сложные выражения. Единственное условие – тип данных возвращаемого значения должен совпадать с типом данных, указанным после ключевого слова RETURNS.

Пример 11.1. Создать и применить функцию скалярного типа для вычисления суммарного количества товара, поступившего за определенную дату. Владелец функции – пользователь с именем user1.

```
CREATE FUNCTION
   user1.sales(@data DATETIME)
RETURNS INT
AS
BEGIN
DECLARE @c INT
SET @c=(SELECT SUM(количество)
      FROM Сделка
      WHERE дата=@data)
RETURN (@c)
END
```

Пример 11.1. Создание функции скалярного типа для вычисления суммарного количества товара, поступившего за определенную дату.

В качестве входного параметра используется дата. Функция возвращает значение целого типа, полученное из оператора SELECT путем суммирования количества товара из таблицы Сделка. Условием отбора записей для суммирования является равенство даты сделки значению входного параметра функции.

Проиллюстрируем обращение к функции пользователя: определим количество товара, поступившего за 02.11.01:

```
DECLARE @kol INT
SET @kol=user1.sales ('02.11.01')
SELECT @kol
```

Функции Inline

Создание и изменение функции этого типа выполняется с помощью команды:

```
<определение_табл_функции>::=
{CREATE | ALTER } FUNCTION [владелец.]
   имя_функции
( [ { @имя_параметра скаляр_тип_данных
   [=default]}[,...n]])
RETURNS TABLE
[ WITH {ENCRYPTION | SCHEMABINDING}
   [,...n] ]
[AS]
RETURN [(] SELECT_оператор [)]
```

Основная часть параметров, используемых при создании табличных функций, аналогична параметрам скалярной функции. Тем не менее создание табличных функций имеет свою специфику.

После ключевого слова RETURNS всегда должно указываться ключевое слово TABLE. Таким образом, функция данного типа должна строго возвращать значение типа данных TABLE. Структура возвращаемого значения типа TABLE не указывается явно при описании собственно типа данных. Вместо этого сервер будет автоматически использовать для возвращаемого значения TABLE структуру, возвращаемую запросом SELECT, который является единственной командой функции.

Особенность функции данного типа заключается в том, что структура значения TABLE создается автоматически в ходе выполнения запроса, а не указывается явно при определении типа после ключевого слова RETURNS.

Возвращаемое функцией значение типа TABLE может быть использовано непосредственно в запросе, т.е. в разделе FROM.

Пример 11.2. Создать и применить функцию табличного типа для определения двух наименований товара с наибольшим остатком.

```
CREATE FUNCTION user1.itog()
RETURNS TABLE
AS
RETURN (SELECT TOP 2 Товар.Название
  FROM Товар INNER JOIN Склад
  ON Товар.КодТовара=Склад.КодТовара
  ORDER BY Склад.Остаток DESC)
```

Пример 11.2. Создание функции табличного типа для определения двух наименований товара с наибольшим остатком.

Использовать функцию для получения двух наименований товара с наибольшим остатком можно следующим образом:

```
SELECT Название
FROM user1.itog()
```

Функции Multi-statement

Создание и изменение функций типа Multi-statement выполняется с помощью следующей команды:

```
<определение_мульти_функции>::=
{CREATE | ALTER }FUNCTION [владелец.]
   имя_функции
( [ { @имя_параметра скаляр_тип_данных
   [=default]}[,...n]])
RETURNS @имя_параметра TABLE
   <определение_таблицы>
[WITH {ENCRYPTION | SCHEMABINDING}
   [,...n] ]
[AS]
BEGIN
<тело_функции>
RETURN
END
```

Использование большей части параметров рассматривалось при описании предыдущих функций.

Отметим, что функции данного типа, как и табличные, возвращают значение типа TABLE. Однако, в отличие от табличных функций, при создании функций Multi-statement необходимо явно задать структуру возвращаемого значения. Она указывается непосредственно после ключевого слова TABLE и, таким образом, является частью определения возвращаемого типа данных. Синтаксис конструкции <определение_таблицы> полностью соответствует одноименным структурам, используемым при создании обычных таблиц с помощью команды CREATE TABLE.

Набор возвращаемых данных должен формироваться с помощью команд INSERT, выполняемых в теле функции. Кроме того, в теле функции допускается использование различных конструкций языка SQL, которые могут контролировать значения, размещаемые в выходном наборе строк. При работе с командой INSERT требуется явно указать имя того объекта, куда необходимо вставить строки. Поэтому в функциях типа Multi-statement, в отличие от табличных, необходимо присвоить какое-то имя объекту с типом данных TABLE – оно и указывается как возвращаемое значение.

Завершение работы функции происходит в двух случаях: если возникают ошибки выполнения и если появляется ключевое слово RETURN. В отличие от функций скалярного типа, при использовании команды RETURN не нужно указывать возвращаемое значение. Сервер автоматически возвратит набор данных типа TABLE, имя и структура которого была указана после ключевого слова RETURNS. В теле функции может быть указано более одной команды RETURN.

Необходимо отметить, что работа функции завершается только при наличии команды RETURN. Это утверждение верно и в том случае, когда речь идет о достижении конца тела функции – самой последней командой должна быть команда RETURN.

Пример 11.3. Создать и применить функцию (типа multi-statement), которая для некоторого сотрудника выводит список всех его подчиненных (подчиненных как непосредственно ему, так и опосредствованно через других сотрудников).

Список сотрудников с указанием каждого руководителя представлен в таблице emp_mgr со следующей структурой:

```
CREATE TABLE emp_mgr
(emp CHAR(2) PRIMARY KEY,-- сотрудник
mgr CHAR(2))          -- руководитель
```

Пример данных в таблице emp_mgr показан ниже. Для упрощения иллюстрации имена сотрудников и их начальников представлены буквами латинского алфавита. У директора организации начальника нет (NULL).

```
emp     mgr
---------
a     NULL
b     a
c     a
d     a
e     f
f     b
g     b
i     c
k     d

CREATE FUNCTION fn_findReports(@id_emp
   CHAR(2))
RETURNS @report TABLE(empid CHAR(2)
   PRIMARY KEY,
   mgrid CHAR(2))
AS
BEGIN
   DECLARE @r INT
   DECLARE @t TABLE(empid CHAR(2)
      PRIMARY KEY,
      mgrid CHAR(2),
        pr   INT DEFAULT 0)
INSERT @t SELECT emp,mgr,0
      FROM emp_mgr
      WHERE emp=@id_emp
SET @r=@@ROWCOUNT
WHILE @r>0
BEGIN
   UPDATE @t   SET pr=1   WHERE pr=0
   INSERT @t SELECT e.emp, e.mgr,0
      FROM  emp_mgr e, @t t
      WHERE e.mgr=t.empid
         AND t.pr=1
   SET @r=@@ROWCOUNT
   UPDATE @t SET pr=2 WHERE pr=1
END
INSERT @report SELECT empid, mgrid
   FROM @t
RETURN
END
```

Пример 11.3. Создание функции, которая для некоторого сотрудника выводит список всех его подчиненных.

Применим созданную функцию для определения списка подчиненных сотрудника ‘b’:

```
SELECT * FROM fn_findReports('b')
```

Оператор возвращает следующие значения:

```
emp     mgr
-----------
b     a
e     f
```

```
f    b
g    b
```
Список подчиненных сотрудника ‘a’ создается с помощью оператора
```
SELECT * FROM fn_findReports('a')
emp     mgr
---------
a     NULL
b     a
c     a
d     a
e     f
f     b
g     b
i     c
k     d
```
Другой оператор формирует список подчиненных сотрудника ‘e’:
```
SELECT * FROM fn_findReports('e')
emp     mgr
--------
e     f
```
Список подчиненных сотрудника ‘c’ создает следующий оператор:
```
SELECT * FROM fn_findReports('c')
emp     mgr
--------
c     a
i     c
```
Удаление любой функции осуществляется командой:
```
DROP FUNCTION {[ владелец.] имя_функции }
   [,...n]
```
Встроенные функции

Встроенные функции, имеющиеся в распоряжении пользователей при работе с SQL, можно условно разделить на следующие группы:

- математические функции;
- строковые функции;
- функции для работы с датой и временем;
- функции конфигурирования;
- функции системы безопасности;
- функции управления метаданными;
- статистические функции.

Математические функции

Краткий обзор математических функций представлен в таблице.

Таблица 11.1.	
ABS	вычисляет абсолютное значение числа
ACOS	вычисляет арккосинус
ASIN	вычисляет арксинус
ATAN	вычисляет арктангенс
ATN2	вычисляет арктангенс с учетом квадратов
CEILING	выполняет округление вверх
COS	вычисляет косинус угла
COT	возвращает котангенс угла
DEGREES	преобразует значение угла из радиан в градусы
EXP	возвращает экспоненту
FLOOR	выполняет округление вниз
LOG	вычисляет натуральный логарифм
LOG10	вычисляет десятичный логарифм
PI	возвращает значение "пи"
POWER	возводит число в степень
RADIANS	преобразует значение угла из градуса в радианы
RAND	возвращат случайное число
ROUND	выполняет округление с заданной точностью
SIGN	определяет знак числа
SIN	вычисляет синус угла
SQUARE	выполняет возведение числа в квадрат
SQRT	извлекает квадратный корень
TAN	возвращает тангенс угла

```
SELECT Товар.Название, Сделка.Количество,
Round(Товар.Цена*Сделка.Количество
   *0.05,1)
AS Налог
FROM Товар INNER JOIN Сделка
   ON Товар.КодТовара=
   Сделка.КодТовара
```

Пример 11.4. Использование функции округления до одного знака после запятой для расчета налога.

Строковые функции

Краткий обзор строковых функций представлен в таблице.

Таблица 11.2.	
ASCII	возвращает код ASCII левого символа строки
CHAR	по коду ASCII возвращает символ
CHARINDEX	определяет порядковый номер символа, с которого начинается вхождение подстроки в строку
DIFFERENCE	возвращает показатель совпадения строк
LEFT	возвращает указанное число символов с начала строки

LEN	возвращает длину строки
LOWER	переводит все символы строки в нижний регистр
LTRIM	удаляет пробелы в начале строки
NCHAR	возвращает по коду символ Unicode
PATINDEX	выполняет поиск подстроки в строке по указанному шаблону
REPLACE	заменяет вхождения подстроки на указанное значение
QUOTENAME	конвертирует строку в формат Unicode
REPLICATE	выполняет тиражирование строки определенное число раз
REVERSE	возвращает строку, символы которой записаны в обратном порядке
RIGHT	возвращает указанное число символов с конца строки
RTRIM	удаляет пробелы в конце строки
SOUNDEX	возвращает код звучания строки
SPACE	возвращает указанное число пробелов
STR	выполняет конвертирование значения числового типа в символьный формат
STUFF	удаляет указанное число символов, заменяя новой подстрокой
SUBSTRING	возвращает для строки подстроку указанной длины с заданного символа
UNICODE	возвращает Unicode-код левого символа строки
UPPER	переводит все символы строки в верхний регистр

```
SELECT Фирма, [Фамилия]+""
   +Left([Имя],1)+"."
   +Left([Отчество],1)
   +"." AS ФИО
FROM Клиент
```

Пример 11.5. Использование функции LEFT для получения инициалов клиентов.

Функции для работы с датой и временем

Краткий обзор основных функций для работы с датой и временем представлен в таблице.

Таблица 11.3.

DATEADD	добавляет к дате указанное значение дней, месяцев, часов и т.д.
DATEDIFF	возвращает разницу между указанными частями двух дат
DATENAME	выделяет из даты указанную часть и возвращает ее в символьном формате
DATEPART	выделяет из даты указанную часть и возвращает ее в числовом формате
DAY	возвращает число из указанной даты
GETDATE	возвращает текущее системное время
ISDATE	проверяет правильность выражения на соответствие одному из возможных форматов ввода даты
MONTH	возвращает значение месяца из указанной даты
YEAR	возвращает значение года из указанной даты

```
SELECT Year(Дата) AS Год, Month(Дата)
  AS Месяц,
  Sum(Количество) AS Общ_Количество
FROM Сделка
GROUP BY Year(Дата), Month(Дата)
```

Пример 11.6. Использование функций YEAR и MONTH для определения общего количества товара, проданного за каждый месяц каждого года.

```
DECLARE @d DATETIME
DECLARE @y INT
SET @d=’29.10.03’
SET @y=DATEPART(yy,@d)
SELECT @y
```

Пример 11.7. Пример выделения из даты значения года.

2.7 Лабораторная работа № 13, 14 (4 часа).

Тема: «Хранимые процедуры».

2.7.1 Цель работы: Получить практические навыки применения хранимых процедур на языке SQL.

2.7.2 Задачи работы:

1. Изучить теоретический материал по написанию и применению хранимых процедур на языке SQL;

2. Написать несколько примеров написания и применения хранимых процедур SQL в базе данных на основе MS SQL Server 2005.

2.7.3 Описание (ход) работы:

Понятие хранимой процедуры

Хранимые процедуры представляют собой группы связанных между собой операторов SQL, применение которых делает работу программиста более легкой и гибкой, поскольку выполнить хранимую процедуру часто оказывается гораздо проще, чем последовательность отдельных операторов SQL. Хранимые процедуры представляют собой набор команд, состоящий из одного или нескольких операторов SQL или функций и сохраняемый в базе данных в откомпилированном виде. Выполнение в базе данных хранимых процедур вместо отдельных операторов SQL дает пользователю следующие преимущества:

· необходимые операторы уже содержатся в базе данных;

· все они прошли этап синтаксического анализа и находятся в исполняемом формате; перед выполнением хранимой процедуры SQL Server генерирует для нее план исполнения, выполняет ее оптимизацию и компиляцию;

· хранимые процедуры поддерживают модульное программирование, так как позволяют разбивать большие задачи на самостоятельные, более мелкие и удобные в управлении части;

· хранимые процедуры могут вызывать другие хранимые процедуры и функции;

· хранимые процедуры могут быть вызваны из прикладных программ других типов;

· как правило, хранимые процедуры выполняются быстрее, чем последовательность отдельных операторов;

· хранимые процедуры проще использовать: они могут состоять из десятков и сотен команд, но для их запуска достаточно указать всего лишь имя нужной хранимой процедуры. Это позволяет уменьшить размер запроса, посылаемого от клиента на сервер, а значит, и нагрузку на сеть.

Хранение процедур в том же месте, где они исполняются, обеспечивает уменьшение объема передаваемых по сети данных и повышает общую производительность системы. Применение хранимых процедур упрощает сопровождение программных комплексов и внесение изменений в них. Обычно все ограничения целостности в виде правил и алгоритмов обработки данных реализуются на сервере баз данных и доступны конечному приложению в виде набора хранимых процедур, которые и представляют интерфейс обработки данных. Для обеспечения целостности данных, а

также в целях безопасности, приложение обычно не получает прямого доступа к данным – вся работа с ними ведется путем вызова тех или иных хранимых процедур.

Подобный подход делает весьма простой модификацию алгоритмов обработки данных, тотчас же становящихся доступными для всех пользователей сети, и обеспечивает возможность расширения системы без внесения изменений в само приложение: достаточно изменить хранимую процедуру на сервере баз данных. Разработчику не нужно перекомпилировать приложение, создавать его копии, а также инструктировать пользователей о необходимости работы с новой версией. Пользователи вообще могут не подозревать о том, что в систему внесены изменения.

Хранимые процедуры существуют независимо от таблиц или каких-либо других объектов баз данных. Они вызываются клиентской программой, другой хранимой процедурой или триггером. Разработчик может управлять правами доступа к хранимой процедуре, разрешая или запрещая ее выполнение. Изменять код хранимой процедуры разрешается только ее владельцу или члену фиксированной роли базы данных. При необходимости можно передать права владения ею от одного пользователя к другому.

Хранимые процедуры в среде MS SQL Server

При работе с SQL Server пользователи могут создавать собственные процедуры, реализующие те или иные действия. Хранимые процедуры являются полноценными объектами базы данных, а потому каждая из них хранится в конкретной базе данных. Непосредственный вызов хранимой процедуры возможен, только если он осуществляется в контексте той базы данных, где находится процедура.

Типы хранимых процедур

B SQL Server имеется несколько типов хранимых процедур.

· Системные хранимые процедуры предназначены для выполнения различных административных действий. Практически все действия по администрированию сервера выполняются с их помощью. Можно сказать, что системные хранимые процедуры являются интерфейсом, обеспечивающим работу с системными таблицами, которая, в конечном счете, сводится к изменению, добавлению, удалению и выборке данных из системных таблиц как пользовательских, так и системных баз данных. Системные хранимые процедуры имеют префикс sp_, хранятся в системной базе данных и могут быть вызваны в контексте любой другой базы данных.

· Пользовательские хранимые процедуры реализуют те или иные действия. Хранимые процедуры – полноценный объект базы данных. Вследствие этого каждая хранимая процедура располагается в конкретной базе данных, где и выполняется.

· Временные хранимые процедуры существуют лишь некоторое время, после чего автоматически уничтожаются сервером. Они делятся на локальные и глобальные. Локальные временные хранимые процедуры могут быть вызваны только из того соединения, в котором созданы. При создании такой процедуры ей необходимо дать имя, начинающееся с одного символа #. Как и все временные объекты, хранимые процедуры этого типа автоматически удаляются при отключении пользователя, перезапуске или остановке сервера. Глобальные временные хранимые процедуры доступны для любых соединений сервера, на котором имеется такая же процедура. Для ее определения достаточно дать ей имя, начинающееся с символов ##. Удаляются эти процедуры при перезапуске или остановке сервера, а также при закрытии соединения, в контексте которого они были созданы.

Создание хранимой процедуры предполагает решение следующих задач:

· определение типа создаваемой хранимой процедуры: временная или пользовательская. Кроме этого, можно создать свою собственную системную хранимую процедуру, назначив ей имя с префиксом sp_ и поместив ее в системную базу данных. Такая процедура будет доступна в контексте любой базы данных локального сервера;

· планирование прав доступа. При создании хранимой процедуры следует учитывать, что она будет иметь те же права доступа к объектам базы данных, что и создавший ее пользователь;

· определение параметров хранимой процедуры. Подобно процедурам, входящим в состав большинства языков программирования, хранимые процедуры могут обладать входными и выходными параметрами;

· разработка кода хранимой процедуры. Код процедуры может содержать последовательность любых команд SQL, включая вызов других хранимых процедур.

Создание новой и изменение имеющейся хранимой процедуры осуществляется с помощью следующей команды:

<определение_процедуры>::={CREATE | ALTER } [PROCEDURE] имя_процедуры [;номер][{@имя_параметра тип_данных } [VARYING] [=default][OUTPUT]][,...n][WITH { RECOMPILE | ENCRYPTION | RECOMPILE, ENCRYPTION }][FOR REPLICATION]AS sql_оператор [...n]

Рассмотрим параметры данной команды.

Используя префиксы sp_, #, ##, создаваемую процедуру можно определить в качестве системной или временной. Как видно из синтаксиса команды, не допускается указывать имя владельца, которому будет принадлежать создаваемая процедура, а также имя базы данных, где она должна быть размещена. Таким образом, чтобы разместить создаваемую хранимую процедуру в конкретной базе данных, необходимо выполнить команду CREATE PROCEDURE в контексте этой базы данных. При обращении из тела хранимой процедуры к объектам той же базы данных можно использовать укороченные имена, т. е. без указания имени базы данных. Когда же требуется обратиться к объектам, расположенным в других базах данных, указание имени базы данных обязательно.

Номер в имени – это идентификационный номер хранимой процедуры, однозначно определяющий ее в группе процедур. Для удобства управления процедурами логически однотипные хранимые процедуры можно группировать, присваивая им одинаковые имена, но разные идентификационные номера.

Для передачи входных и выходных данных в создаваемой хранимой процедуре могут использоваться параметры, имена которых, как и имена локальных переменных, должны начинаться с символа @. В одной хранимой процедуре можно задать множество параметров, разделенных запятыми. В теле процедуры не должны применяться локальные переменные, чьи имена совпадают с именами параметров этой процедуры.

Для определения типа данных, который будет иметь соответствующий параметр хранимой процедуры, годятся любые типы данных SQL, включая определенные пользователем. Однако тип данных CURSOR может быть использован только как выходной параметр хранимой процедуры, т.е. с указанием ключевого слова OUTPUT.

Наличие ключевого слова OUTPUT означает, что соответствующий параметр предназначен для возвращения данных из хранимой процедуры. Однако это вовсе не означает, что параметр не подходит для передачи значений в хранимую процедуру. Указание ключевого слова OUTPUT предписывает серверу при выходе из хранимой процедуры присвоить текущее значение параметралокальной переменной, которая была указана при вызове процедуры в качестве значения параметра. Отметим, что при указании ключевого слова OUTPUT значение соответствующего параметра при вызове процедуры может быть задано только с помощью локальной переменной. Не разрешается использование любых выражений или констант, допустимое для обычных параметров.

Ключевое слово VARYING применяется совместно с параметром OUTPUT, имеющим тип CURSOR. Оно определяет, чтовыходным параметром будет результирующее множество.

Ключевое слово DEFAULT представляет собой значение, которое будет принимать соответствующий параметр по умолчанию. Таким образом, при вызове процедуры можно не указывать явно значение соответствующего параметра.

Так как сервер кэширует план исполнения запроса и компилированный код, при последующем вызове процедуры будут использоваться уже готовые значения. Однако в некоторых случаях все же требуется выполнять перекомпиляцию кода процедуры. Указание ключевого слова RECOMPILE предписывает системе создавать план выполнения хранимой процедуры при каждом ее вызове.

Параметр FOR REPLICATION востребован при репликации данных и включении создаваемой хранимой процедуры в качестве статьи в публикацию.

Ключевое слово ENCRYPTION предписывает серверу выполнить шифрование кода хранимой процедуры, что может обеспечить защиту от использования авторских алгоритмов, реализующих работу хранимой процедуры.

Ключевое слово AS размещается в начале собственно тела хранимой процедуры, т.е. набора команд SQL, с помощью которых и будет реализовываться то или иное действие. В теле процедуры могут применяться практически все команды SQL, объявляться транзакции, устанавливаться блокировки и вызываться другие хранимые процедуры. Выход из хранимой процедуры можно осуществить посредством команды RETURN.

Для выполнения хранимой процедуры используется команда:

[[EXEC [UTE] имя_процедуры [;номер][[@имя_параметра=]{значение | @имя_переменной} [OUTPUT]|[DEFAULT]][,...n]

Если вызов хранимой процедуры не является единственной командой в пакете, то присутствие команды EXECUTE обязательно. Более того, эта команда требуется для вызова процедуры из тела другой процедуры или триггера.

Использование ключевого слова OUTPUT при вызове процедуры разрешается только для параметров, которые были объявлены при создании процедуры с ключевым словом OUTPUT.

Когда же при вызове процедуры для параметра указывается ключевое слово DEFAULT, то будет использовано значение по умолчанию. Естественно, указанное слово DEFAULT разрешается только для тех параметров, для которых определено значение по умолчанию.

Из синтаксиса команды EXECUTE видно, что имена параметров могут быть опущены при вызове процедуры. Однако в этом случае пользователь должен указывать значения для параметров в том же порядке, в каком они перечислялись при создании процедуры. Присвоить параметру значение по умолчанию, просто пропустив его при перечислении нельзя. Если же требуется опуститьпараметры, для которых определено значение по умолчанию, достаточно явного указания имен параметров при вызове хранимой процедуры. Более того, таким способом можно перечислять параметры и их значения в произвольном порядке.

Отметим, что при вызове процедуры указываются либо имена параметров со значениями, либо только значения без именипараметра. Их комбинирование не допускается.

Пример 12.1. Процедура без параметров. Разработать процедуру для получения названий и стоимости товаров, приобретенных Ивановым.

CREATE PROC my_proc1ASSELECT Товар.Название, Товар.Цена*Сделка.Количество AS Стоимость, Клиент.Фамилия FROM Клиент INNER JOIN (Товар INNER JOIN Сделка ON Товар.КодТовара=Сделка.КодТовара) ON Клиент.КодКлиента=Сделка.КодКлиента WHERE Клиент.Фамилия=’Иванов’

Пример 12.1. Процедура для получения названий и стоимости товаров, приобретенных Ивановым.

Для обращения к процедуре можно использовать команды:

EXEC my_proc1 или my_proc1

Процедура возвращает набор данных.

Пример 12.2. Процедура без параметров. Создать процедуру для уменьшения цены товара первого сорта на 10%.

CREATE PROC my_proc2ASUPDATE Товар SET Цена=Цена*0.9 WHERE Сорт=’первый’

Пример 12.2. Процедура для уменьшения цены товара первого сорта на 10%.

Для обращения к процедуре можно использовать команды:

EXEC my_proc2 или my_proc2

Процедура не возвращает никаких данных.

Пример 12.3. Процедура с входным параметром. Создать процедуру для получения названий и стоимости товаров, которые приобрел заданный клиент.

CREATE PROC my_proc3 @k VARCHAR(20)ASSELECT Товар.Название, Товар.Цена*Сделка.Количество AS Стоимость, Клиент.Фамилия FROM Клиент INNER JOIN (Товар INNER JOIN Сделка ON Товар.КодТовара=Сделка.КодТовара) ON Клиент.КодКлиента=Сделка.КодКлиента WHERE Клиент.Фамилия=@k

Пример 12.3. Процедура для получения названий и стоимости товаров, которые приобрел заданный клиент.

Для обращения к процедуре можно использовать команды:

EXEC my_proc3 'Иванов' или my_proc3 @k='Иванов'

Пример 12.4. Процедура с входными параметрами. Создать процедуру для уменьшения цены товара заданного типа в соответствии с указанным %.

CREATE PROC my_proc4 @t VARCHAR(20), @p FLOATASUPDATE Товар SET Цена=Цена*(1-@p) WHERE Тип=@t

Пример 12.4. Процедура для уменьшения цены товара заданного типа в соответствии с указанным %.

Для обращения к процедуре можно использовать команды:

EXEC my_proc4 'Вафли',0.05 илиEXEC my_proc4 @t='Вафли', @p=0.05

Пример 12.5. Процедура с входными параметрами и значениями по умолчанию. Создать процедуру для уменьшения цены товара заданного типа в соответствии с указанным %.

CREATE PROC my_proc5 @t VARCHAR(20)=’Конфеты', @p FLOAT=0.1ASUPDATE Товар SET Цена=Цена*(1-@p) WHERE Тип=@t

Пример 12.5. Процедура с входными параметрами и значениями по умолчанию. Создать процедуру для уменьшения цены товара заданного типа в соответствии с указанным %.

Для обращения к процедуре можно использовать команды:

EXEC my_proc5 'Вафли',0.05 илиEXEC my_proc5 @t='Вафли', @p=0.05 илиEXEC my_proc5 @p=0.05

В этом случае уменьшается цена конфет (значение типа не указано при вызове процедуры и берется по умолчанию).

EXEC my_proc5

В последнем случае оба параметра (и тип, и проценты) не указаны при вызове процедуры, их значения берутся по умолчанию.

Пример 12.6. Процедура с входными и выходными параметрами. Создать процедуру для определения общей стоимости товаров, проданных за конкретный месяц.

CREATE PROC my_proc6 @m INT, @s FLOAT OUTPUTASSELECT @s=Sum(Товар.Цена*Сделка.Количество) FROM Товар INNER JOIN Сделка ON Товар.КодТовара=Сделка.КодТовара GROUP BY Month(Сделка.Дата) HAVING Month(Сделка.Дата)=@m

Пример 12.6. Процедура с входными и выходными параметрами. Создать процедуру для определения общей стоимости товаров, проданных за конкретный месяц.

Для обращения к процедуре можно использовать команды:

DECLARE @st FLOATEXEC my_proc6 1,@st OUTPUTSELECT @st

Этот блок команд позволяет определить стоимость товаров, проданных в январе (входной параметр месяц указан равным 1).

Создать процедуру для определения общего количества товаров, приобретенных фирмой, в которой работает заданный сотрудник.

Сначала разработаем процедуру для определения фирмы, где работает сотрудник.

CREATE PROC my_proc7 @n VARCHAR(20), @f VARCHAR(20) OUTPUTASSELECT @f=Фирма FROM Клиент WHERE Фамилия=@n

Пример 12.7. Использование вложенных процедур. Создать процедуру для определения общего количества товаров, приобретенных фирмой, в которой работает заданный сотрудник.

Затем создадим процедуру, подсчитывающую общее количество товара, который закуплен интересующей нас фирмой.

CREATE PROC my_proc8 @fam VARCHAR(20), @kol INT OUTPUTASDECLARE @firm VARCHAR(20)EXEC my_proc7 @fam,@firm OUTPUTSELECT @kol=Sum(Сделка.Количество) FROM Клиент INNER JOIN Сделка ON Клиент.КодКлиента=Сделка.КодКлиента GROUP BY Клиент.Фирма HAVING Клиент.Фирма=@firm

Пример 12.7. Создание процедуры для определения общего количества товаров, приобретенных фирмой, в которой работает заданный сотрудник.

Вызов процедуры осуществляется с помощью команды:

DECLARE @k INTEXEC my_proc8 ‘Иванов’,@k OUTPUTSELECT @k

2.8 Лабораторная работа № 15 (2 часа).

Тема: «Триггеры».

2.8.1 Цель работы: Получить практические навыки применения триггеров на языке SQL.

2.8.2 Задачи работы:

1. Изучить теоретический материал по написанию и применению триггеров на языке SQL;

2. Написать несколько примеров написания и применения триггеров SQL в базе данных на основе MS SQL Server 2005.

2.8.3 Описание (ход) работы

Определение триггера в стандарте языка SQL

Триггеры являются одной из разновидностей хранимых процедур. Их исполнение происходит при выполнении для таблицы какого-либо оператора языка манипулирования данными (DML). Триггеры используются для проверки целостности данных, а также для отката транзакций.

Триггер – это откомпилированная SQL-процедура, исполнение которой обусловлено наступлением определенных событий внутри реляционной базы данных. Применение триггеров большей частью весьма удобно для пользователей базы данных. И все же их использование часто связано с дополнительными затратами ресурсов на операции ввода/вывода. В том случае, когда тех же результатов (с гораздо меньшими непроизводительными затратами ресурсов) можно добиться с помощью хранимых процедур или прикладных программ, применение триггеров нецелесообразно.

Триггеры – особый инструмент SQL-сервера, используемый для поддержания целостности данных в базе данных. С помощью ограничений целостности, правил и значений по умолчанию не всегда можно добиться нужного уровня функциональности. Часто требуется реализовать сложные алгоритмы проверки данных, гарантирующие их достоверность и реальность. Кроме того, иногда необходимо отслеживать изменения значений таблицы, чтобы нужным образом изменить связанные данные. Триггеры можно рассматривать как своего рода фильтры, вступающие в действие после выполнения всех операций в соответствии с правилами, стандартными значениями и т.д.

Триггер представляет собой специальный тип хранимых процедур, запускаемых сервером автоматически при попытке изменения данных в таблицах, с которыми триггеры связаны. Каждый триггер привязывается к конкретной таблице. Все производимые им модификации данных рассматриваются как одна транзакция. В случае обнаружения ошибки или нарушения целостности данных происходит откат этой транзакции. Тем самым внесение изменений запрещается. Отменяются также все изменения, уже сделанные триггером.

Создает триггер только владелец базы данных. Это ограничение позволяет избежать случайного изменения структуры таблиц, способов связи с ними других объектов и т.п.

Триггер представляет собой весьма полезное и в то же время опасное средство. Так, при неправильной логике его работы можно легко уничтожить целую базу данных, поэтому триггеры необходимо очень тщательно отлаживать.

В отличие от обычной подпрограммы, триггер выполняется неявно в каждом случае возникновения триггерного события, к тому же он не имеет аргументов. Приведение его в действие иногда называют запуском триггера. С помощью триггеров достигаются следующие цели:

- проверка корректности введенных данных и выполнение сложных ограничений целостности данных, которые трудно, если вообще возможно, поддерживать с помощью ограничений целостности, установленных для таблицы;
- выдача предупреждений, напоминающих о необходимости выполнения некоторых действий при обновлении таблицы, реализованном определенным образом;
- накопление аудиторской информации посредством фиксации сведений о внесенных изменениях и тех лицах, которые их выполнили;
- поддержка репликации.

Основной формат команды CREATE TRIGGER показан ниже:

```
<Определение_триггера>::=
  CREATE TRIGGER имя_триггера
  BEFORE | AFTER <триггерное_событие>
  ON <имя_таблицы>
  [REFERENCING
    <список_старых_или_новых_псевдонимов>]
  [FOR EACH { ROW | STATEMENT}]
  [WHEN(условие_триггера)]
  <тело_триггера>
```

триггерные события состоят из вставки, удаления и обновления строк в таблице. В последнем случае для триггерного события можно указать конкретные имена столбцов таблицы. Время запуска триггера определяется с помощью ключевых слов BEFORE (триггер запускается до выполнения связанных с ним событий) или AFTER (после их выполнения).

Выполняемые триггером действия задаются для каждой строки (FOR EACH ROW), охваченной данным событием, или только один раз для каждого события (FOR EACH STATEMENT).

Обозначение <список_старых_или_новых_псевдонимов> относится к таким компонентам, как старая или новая строка (OLD / NEW) либо старая или новая таблица (OLD TABLE / NEW TABLE). Ясно, что старые значения не применимы для событий вставки, а новые – для событий удаления.

При условии правильного использования триггеры могут стать очень мощным механизмом. Основное их преимущество заключается в том, что стандартные функции сохраняются внутри базы данных и согласованно активизируются при каждом ее

обновлении. Это может существенно упростить приложения. Тем не менее следует упомянуть и о присущих триггеру недостатках:

- сложность: при перемещении некоторых функций в базу данных усложняются задачи ее проектирования, реализации и администрирования;
- скрытая функциональность: перенос части функций в базу данных и сохранение их в виде одного или нескольких триггеров иногда приводит к сокрытию от пользователя некоторых функциональных возможностей. Хотя это в определенной степени упрощает его работу, но, к сожалению, может стать причиной незапланированных, потенциально нежелательных и вредных побочных эффектов, поскольку в этом случае пользователь не в состоянии контролировать все процессы, происходящие в базе данных;
- влияние на производительность: перед выполнением каждой команды по изменению состояния базы данных СУБД должна проверить триггерное условие с целью выяснения необходимости запуска триггера для этой команды. Выполнение подобных вычислений сказывается на общей производительности СУБД, а в моменты пиковой нагрузки ее снижение может стать особенно заметным. Очевидно, что при возрастании количества триггеров увеличиваются и накладные расходы, связанные с такими операциями.

Неправильно написанные триггеры могут привести к серьезным проблемам, таким, например, как появление "мертвых" блокировок. Триггеры способны длительное время блокировать множество ресурсов, поэтому следует обратить особое внимание на сведение к минимуму конфликтов доступа.

Реализация триггеров в среде MS SQL Server

В реализации СУБД MS SQL Server используется следующий оператор создания или изменения триггера:

```
<Определение_триггера>::=
{CREATE | ALTER} TRIGGER имя_триггера
ON {имя_таблицы | имя_просмотра }
[WITH ENCRYPTION ]
{
{ { FOR | AFTER | INSTEAD OF }
{ [ DELETE] [,] [ INSERT] [,] [ UPDATE] }
[ WITH APPEND ]
[ NOT FOR REPLICATION ]
AS
   sql_оператор[...n]
} |
{ {FOR | AFTER | INSTEAD OF } { [INSERT] [,]
  [UPDATE] }
[ WITH APPEND]
[ NOT FOR REPLICATION]
AS
{ IF UPDATE(имя_столбца)
[ {AND | OR} UPDATE(имя_столбца)] [...n]
|
IF (COLUMNS_UPDATES(){оператор_бит_обработки}
  бит_маска_изменения)
{оператор_бит_сравнения }бит_маска [...n]}
sql_оператор [...n]
}
}
```

Триггер может быть создан только в текущей базе данных, но допускается обращение внутри триггера к другим базам данных, в том числе и расположенным на удаленном сервере.

Рассмотрим назначение аргументов из команды CREATE | ALTER TRIGGER.

Имя триггера должно быть уникальным в пределах базы данных. Дополнительно можно указать имя владельца.

При указании аргумента WITH ENCRYPTION сервер выполняет шифрование кода триггера, чтобы никто, включая администратора, не мог получить к нему доступ и прочитать его. Шифрование часто используется для скрытия авторских алгоритмов обработки данных, являющихся интеллектуальной собственностью программиста или коммерческой тайной.

Типы триггеров

В SQL Server существует два параметра, определяющих поведение триггеров:

- AFTER. Триггер выполняется после успешного выполнения вызвавших его команд. Если же команды по какой-либо причине не могут быть успешно завершены, триггер не выполняется. Следует отметить, что изменения данных в результате выполнения запроса пользователя и выполнение триггера осуществляется в теле одной транзакции: если произойдет откат триггера, то будут отклонены и пользовательские изменения. Можно определить несколько AFTER-триггеров для каждой операции (INSERT, UPDATE, DELETE). Если для таблицы предусмотрено выполнение нескольких AFTER-триггеров, то с помощью системной хранимой процедуры sp_settriggerorder можно указать, какой из них будет выполняться первым, а какой последним. По умолчанию в SQL Server все триггеры являются AFTER-триггерами.
- INSTEAD OF. Триггер вызывается вместо выполнения команд. В отличие от AFTER-триггера INSTEAD OF-триггер может быть определен как для таблицы, так и для просмотра. Для каждой операции INSERT, UPDATE, DELETE можно определить только один INSTEAD OF-триггер.

Триггеры различают по типу команд, на которые они реагируют.

Существует три типа триггеров:

- INSERT TRIGGER – запускаются при попытке вставки данных с помощью команды INSERT.
- UPDATE TRIGGER – запускаются при попытке изменения данных с помощью команды UPDATE.
- DELETE TRIGGER – запускаются при попытке удаления данных с помощью команды DELETE.

Конструкции [DELETE] [,] [INSERT] [,] [UPDATE] и FOR | AFTER | INSTEAD OF } { [INSERT] [,] [UPDATE] определяют, на какую команду будет реагировать триггер. При его создании должна быть указана хотя бы одна команда. Допускается создание триггера, реагирующего на две или на все три команды.

Аргумент WITH APPEND позволяет создавать несколько триггеров каждого типа.

При создании триггера с аргументом NOT FOR REPLICATION запрещается его запуск во время выполнения модификации таблиц механизмами репликации.

Конструкция AS sql_оператор[...n] определяет набор SQL- операторов и команд, которые будут выполнены при запуске триггера.

Отметим, что внутри триггера не допускается выполнение ряда операций, таких, например, как:

- создание, изменение и удаление базы данных;
- восстановление резервной копии базы данных или журнала транзакций.

Выполнение этих команд не разрешено, так как они не могут быть отменены в случае отката транзакции, в которой выполняется триггер. Это запрещение вряд ли может

каким-то образом сказаться на функциональности создаваемых триггеров. Трудно найти такую ситуацию, когда, например, после изменения строки таблицы потребуется выполнить восстановление резервной копии журнала транзакций.

Программирование триггера

При выполнении команд добавления, изменения и удаления записей сервер создает две специальные таблицы: inserted и deleted. В них содержатся списки строк, которые будут вставлены или удалены по завершении транзакции. Структура таблиц inserted и deleted идентична структуре таблиц, для которой определяется триггер. Для каждого триггера создается свой комплект таблиц inserted и deleted, поэтому никакой другой триггер не сможет получить к ним доступ. В зависимости от типа операции, вызвавшей выполнение триггера, содержимое таблиц inserted и deleted может быть разным:

- команда INSERT – в таблице inserted содержатся все строки, которые пользователь пытается вставить в таблицу; в таблице deleted не будет ни одной строки; после завершения триггера все строки из таблицы inserted переместятся в исходную таблицу;
- команда DELETE – в таблице deleted будут содержаться все строки, которые пользователь попытается удалить; триггер может проверить каждую строку и определить, разрешено ли ее удаление; в таблице inserted не окажется ни одной строки;
- команда UPDATE – при ее выполнении в таблице deleted находятся старые значения строк, которые будут удалены при успешном завершении триггера. Новые значения строк содержатся в таблице inserted. Эти строки добавятся в исходную таблицу после успешного выполнения триггера.

Для получения информации о количестве строк, которое будет изменено при успешном завершении триггера, можно использовать функцию @@ROWCOUNT; она возвращает количество строк, обработанных последней командой. Следует подчеркнуть, что триггер запускается не при попытке изменить конкретную строку, а в момент выполнения команды изменения. Одна такая команда воздействует на множество строк, поэтому триггер должен обрабатывать все эти строки.

Если триггер обнаружил, что из 100 вставляемых, изменяемых или удаляемых строк только одна не удовлетворяет тем или иным условиям, то никакая строка не будет вставлена, изменена или удалена. Такое поведение обусловлено требованиями транзакции – должны быть выполнены либо все модификации, либо ни одной.

Триггер выполняется как неявно определенная транзакция, поэтому внутри триггера допускается применение команд управления транзакциями. В частности, при обнаружении нарушения ограничений целостности для прерывания выполнения триггера и отмены всех изменений, которые пытался выполнить пользователь, необходимо использовать команду ROLLBACK TRANSACTION.

Для получения списка столбцов, измененных при выполнении команд INSERT или UPDATE, вызвавших выполнение триггера, можно использовать функцию COLUMNS_UPDATED(). Она возвращает двоичное число, каждый бит которого, начиная с младшего, соответствует одному столбцу таблицы (в порядке следования столбцов при создании таблицы). Если бит установлен в значение "1", то соответствующий столбец был изменен. Кроме того, факт изменения столбца определяет и функция UPDATE (имя_столбца).

Для удаления триггера используется команда

```
DROP TRIGGER {имя_триггера} [,...n]
```

Приведем примеры использования триггеров.

Пример 14.1. Использование триггера для реализации ограничений на значение. В добавляемой в таблицу Сделка записи количество проданного товара должно быть не меньше, чем его остаток из таблицы Склад.

Команда вставки записи в таблицу Сделка может быть, например, такой:

```
INSERT INTO Сделка
  VALUES (3,1,-299,'01/08/2002')
```

Создаваемый триггер должен отреагировать на ее выполнение следующим образом: необходимо отменить команду, если в таблице Склад величина остатка товара оказалась меньше продаваемого количества товара с введенным кодом (в примере код товара=3). Во вставляемой записи количество товара указывается со знаком "+", если товар поставляется, и со знаком "-", если он продается. Представленный триггер настроен на обработку только одной добавляемой записи.

```
CREATE TRIGGER Триггер_ins
ON Сделка FOR INSERT
AS
IF @@ROWCOUNT=1
BEGIN
  IF NOT EXISTS(SELECT *
      FROM inserted
      WHERE -inserted.количество<=ALL(SELECT
        Склад.Остаток
    FROM Склад,Сделка
    WHERE Склад.КодТовара=
      Сделка.КодТовара))
    BEGIN
      ROLLBACK TRAN
    PRINT
     'Отмена поставки: товара на складе нет'
    END
END
```

Пример 14.1. Использование триггера для реализации ограничений на значение.

Пример 14.2. Использования триггера для сбора статистических данных.

Создать триггер для обработки операции вставки записи в таблицу Сделка, например, такой команды:

```
INSERT INTO Сделка
  VALUES (3,1,200,'01/08/2002')
```

поставляется товар с кодом 3 от клиента с кодом 1 в количестве 200 единиц.

При продаже или получении товара необходимо соответствующим образом изменить количество его складского запаса. Если товара на складе еще нет, необходимо добавить соответствующую запись в таблицу Склад. Триггер обрабатывает только одну добавляемую строку.

```
ALTER TRIGGER Триггер_ins
ON Сделка FOR INSERT
AS
DECLARE @x INT, @y INT
IF @@ROWCOUNT=1
--в таблицу Сделка добавляется запись
--о поставке товара
BEGIN
--количество проданного товара должно быть не
--меньше, чем его остаток из таблицы Склад
IF NOT EXISTS(SELECT *
           FROM inserted
           WHERE -inserted.количество<
  =ALL(SELECT Склад.Остаток
         FROM Склад,Сделка
         WHERE Склад.КодТовара=
```

```
                Сделка.КодТовара))
     BEGIN
        ROLLBACK TRAN
          PRINT 'откат товара нет '
      END
--если записи о поставленном товаре еще нет,
--добавляется соответствующая запись
--в таблицу Склад
  IF NOT EXISTS ( SELECT *
            FROM Склад C, inserted i
            WHERE C.КодТовара=i.КодТовара )
       INSERT INTO Склад (КодТовара,Остаток)
  ELSE
--если запись о товаре уже была в таблице
--Склад, то определяется код и количество
--товара издобавленной в таблицу Сделка записи
  BEGIN
    SELECT @y=i.КодТовара, @x=i.Количество
    FROM Сделка C, inserted i
    WHERE C.КодТовара=i.КодТовара
--и производится изменения количества товара в
--таблице Склад
          UPDATE Склад
          SET Остаток=остаток+@x
          WHERE КодТовара=@y
    END
END
```

Пример 14.2. Использования триггера для сбора статистических данных.

Пример 14.3. Создать триггер для обработки операции удаления записи из таблицы Сделка, например, такой команды:

```
DELETE FROM Сделка WHERE КодСделки=4
```

Для товара, код которого указан при удалении записи, необходимо откорректировать его остаток на складе. Триггер обрабатывает только одну удаляемую запись.

```
CREATE TRIGGER Триггер_del
ON Сделка FOR DELETE
AS
IF @@ROWCOUNT=1 -- удалена одна запись
BEGIN
  DECLARE @y INT,@x INT
--определяется код и количество товара из
--удаленной из таблицы Склад записи
  SELECT @y=КодТовара, @x=Количество
  FROM deleted
--в таблице Склад корректируется количество
--товара
     UPDATE Склад
     SET Остаток=Остаток-@x
     WHERE КодТовара=@y
END
```

Пример 14.3. Триггер для обработки операции удаления записи из таблицы

Пример 14.4. Создать триггер для обработки операции изменения записи в таблице Сделка, например, такой командой:

```
UPDATE Сделка SET количество=количество-10
WHERE КодТовара=3
```

во всех сделках с товаром, имеющим код, равный 3, уменьшить количество товара на 10 единиц.

Указанная команда может привести к изменению сразу нескольких записей в таблице Сделка. Поэтому покажем, как создать триггер, обрабатывающий не одну запись. Для каждой измененной записи необходимо для старого (до изменения) кода товара уменьшить остаток товара на складе на величину старого (до изменения) количества товара и для нового (после изменения) кода товара увеличить его остаток на складе на величину нового (после изменения) значения. Чтобы обработать все измененные записи, введем курсоры, в которых сохраним все старые (из таблицы deleted) и все новые значения (из таблицы inserted).

```
CREATE TRIGGER Триггер_upd
ON Сделка FOR UPDATE
AS
DECLARE @x INT, @x_old INT, @y INT, @y_old INT
-- курсор с новыми значениями
DECLARE CUR1 CURSOR FOR
    SELECT КодТовара,Количество
    FROM inserted
-- курсор со старыми значениями
DECLARE CUR2 CURSOR FOR
    SELECT КодТовара,Количество
    FROM deleted
OPEN CUR1
OPEN CUR2
-- перемещаемся параллельно по обоим курсорам
    FETCH NEXT FROM CUR1 INTO @x, @y
    FETCH NEXT FROM CUR2 INTO @x_old, @y_old
    WHILE @@FETCH_STATUS=0
      BEGIN
--для старого кода товара уменьшается его
--количество на складе
      UPDATE Склад
      SET Остаток=Остаток-@y_old
      WHERE КодТовара=@x_old
--для нового кода товара, если такого товара
--еще нет на складе, вводится новая запись
      IF NOT EXISTS (SELECT * FROM Склад
         WHERE КодТовара=@x)
      INSERT INTO Склад(КодТовара,Остаток)
        VALUES (@x,@y)
      ELSE
--иначе для нового кода товара увеличивается
--его количество на складе
        UPDATE Склад
        SET Остаток=Остаток+@y
        WHERE КодТовара=@x
      FETCH NEXT FROM CUR1 INTO @x, @y
    FETCH NEXT FROM CUR2 INTO @x_old, @y_old
END
CLOSE CUR1
CLOSE CUR2
```

```
DEALLOCATE CUR1
DEALLOCATE CUR2
```

Пример 14.4. триггер для обработки операции изменения записи в таблице

В рассмотренном триггере отсутствует сравнение количества товара при изменении записи о сделке с его остатком на складе.

Пример 14.5. Исправим этот недостаток. Для генерирования сообщения об ошибке используем в теле триггера команду MS SQL Server RAISERROR, аргументами которой являются текст сообщения, уровень серьезности и статус ошибки.

```
ALTER TRIGGER Триггер_upd
ON Сделка FOR UPDATE
AS
DECLARE @x INT, @x_old INT, @y INT,
        @y_old INT ,@o INT
DECLARE CUR1 CURSOR FOR
    SELECT КодТовара,Количество
    FROM inserted
DECLARE CUR2 CURSOR FOR
    SELECT КодТовара,Количество
    FROM deleted
OPEN CUR1
OPEN CUR2
    FETCH NEXT FROM CUR1 INTO @x, @y
    FETCH NEXT FROM CUR2 INTO @x_old, @y_old
    WHILE @@FETCH_STATUS=0
      BEGIN
        SELECT @o=остаток
        FROM Склад
        WHERE кодтовара=@x
        IF @o<-@y
        BEGIN
          RAISERROR('откат',16,10)
          CLOSE CUR1
          CLOSE CUR2
          DEALLOCATE CUR1
          DEALLOCATE CUR22
          ROLLBACK TRAN
          RETURN
        END
          UPDATE Склад
        SET Остаток=Остаток-@y_old
        WHERE КодТовара=@x_old
        IF NOT EXISTS (SELECT * FROM Склад
          WHERE КодТовара=@x)
        INSERT INTO Склад(КодТовара,Остаток)
          VALUES (@x,@y)
      ELSE
        UPDATE Склад
        SET Остаток=Остаток+@y
        WHERE КодТовара=@x
    FETCH NEXT FROM CUR1 INTO @x, @y
    FETCH NEXT FROM CUR2 INTO @x_old, @y_old
END
CLOSE CUR1
```

```
CLOSE CUR2
DEALLOCATE CUR1
DEALLOCATE CUR2
```

Пример 14.5. Исправленный вариант триггера для обработки операции изменения записи в таблице

Пример 14.6. В примере 14.5 происходит отмена всех изменений при невозможности реализовать хотя бы одно из них. Создадим триггер, позволяющий отменять изменение только некоторых записей и выполнять изменение остальных.

В этом случае триггер выполняется не после изменения записей, а вместо команды изменения.

```
ALTER TRIGGER Триггер_upd
ON Сделка INSTEAD OF UPDATE
AS
DECLARE @k INT, @k_old INT
DECLARE @x INT, @x_old INT, @y INT
DECLARE @y_old INT ,@o INT
DECLARE CUR1 CURSOR FOR
    SELECT КодСделки, КодТовара,Количество
    FROM inserted
DECLARE CUR2 CURSOR FOR
    SELECT КодСделки, КодТовара,Количество
    FROM deleted
OPEN CUR1
OPEN CUR2
    FETCH NEXT FROM CUR1 INTO @k,@x, @y
    FETCH NEXT FROM CUR2 INTO @k_old,@x_old,
      @y_old
    WHILE @@FETCH_STATUS=0
      BEGIN
        SELECT @o=остаток
        FROM Склад
        WHERE КодТовара=@x
        IF @o>=-@y
        BEGIN
        RAISERROR('изменение',16,10)
        UPDATE Сделка SET количество=@y,
         КодТовара=@x
        WHERE КодСделки=@k

        UPDATE Склад
        SET Остаток=Остаток-@y_old
        WHERE КодТовара=@x_old

        IF NOT EXISTS (SELECT * FROM Склад
          WHERE КодТовара=@x)
        INSERT INTO Склад(КодТовара, Остаток)
          VALUES (@x,@y)
        ELSE
        UPDATE Склад
        SET Остаток=Остаток+@y
        WHERE КодТовара=@x
      END
      ELSE
```

```
          RAISERROR('запись не изменена',16,10)
        FETCH NEXT FROM CUR1 INTO @k,@x, @y
        FETCH NEXT FROM CUR2 INTO @k_old,@x_old,
          @y_old
END
CLOSE CUR1
CLOSE CUR2
DEALLOCATE CUR1
DEALLOCATE CUR2
```

Пример 14.6. Триггер, позволяющий отменять изменение только некоторых записей и выполнять изменение остальных.