ТРЗБД

Технология разработки и защиты БД

Тузова Диана Анатольевна

Основные понятия БД

Банк данных – является разновидностью информационной системы или ИС в которой реализованы функции централизованного хранения и накопление информации записанных в одну или несколько БД.

Банк Данных в общем случае состоит из:

Базы Данных, СУБД, словаря данных, администратора, вычислительной системы и обслуживающего персонала.

База данных (БД) – совокупность специальным образом организованных данных хранимых в памяти вычислительной системы и отображающих состояние объекта и их взаимосвязей в предметной области. Логическую структур, хранимых в базе данных, называют моделью представления данных.

К основным моделям относят:

* Иерархическую
* Сетевую
* Табличную (реляционную)
* Пост реляционную
* Многомерную
* Объектно-ориентированную
* Нереляционную

СУБД – это комплекс языковых и программных средств, предназначенных для создания, ведение и использования БД многими пользователями.

Различают СУБД по моделям данных.

Словарь данных – представляет собой подсистему Банка данных, предназначенную для централизованного хранения информации о структурах данных, взаимосвязях данных, типов, форматов данных, принадлежности к пользователю, кодах и способах защиты данных.

Администратор БД – это лицо или группа лиц, отвечающих за выработку требований к БД, создание, проектирование, эффективное использование и сопровождение.

Вычислительная система – представляет собой совокупность взаимосвязанных и согласованных компьютером и других устройств, обеспечивающих автоматизацию процессов приема, обработки и выдачи информации.

Обслуживающий персонал – выполняет функции поддержания технических и программных средств в работоспособном состоянии.

Архитектура Информационной Системы

Эффективность функционирования ИС во многом зависит от ее архитектуры. Наиболее развитой (широко распространенной) на данный момент является архитектура «клиент-сервер». Эта архитектура предполагает наличие компьютерной сети и распределенной БД.

Сервером обеденного ресурса в компьютерной сети называется программа или компьютер, управляющий этим ресурсом.

Клиентом – программа или компьютер использующие этот ресурс.

В качестве ресурса могут быть:

* БД
* Файловые системы
* Почтовые службы

Тип сервера определяется типом ресурса.

Достоинствами организации ИС по архитектуре «клиент-сервер» является удачное сочетание централизованного хранения , обслуживания и коллективного доступа к общей информации с индивидуальной работой пользователя.

Организована архитектура «клиент-сервер» может быть по-разному.

Классификация СУБД

К СУБД относят следующие основные виды программ:

* Полнофункциональные СУБД
* Серверы БД
* Клиенты БД
* Средства разработки программ с БД

Полнофункциональные СУБД представляют собой традиционные СУБД, которые сначала появились для больших компьютеров, а затем и для персональных. Они являются наиболее многочисленными и мощными по своим возможностям.

Серверы БД предназначены для организации центров обработки данных в компьютерных сетях. Серверы реализуют функции управления базами данных, запрашиваемых клиентскими программами. В роли клиентских программ могут служить: текстовые процессоры, электронные таблицы, БД, электронная почта и т.д.

Клиентские или персональные СУБД обычно обеспечивают возможность создания персональных БД и недорогих приложений для работы с ними. Персональные СУБД могут выступать в роли клиента в многопользовательской СУБД.

В функции Персональных СУБД входит:

* Управление данными во внешней памяти.
* Управление буферами оперативной памяти.
* Управление транзакциями.
* Ведение журнала изменения БД.
* Обеспечение целостности и безопасности БД.

Реализация функций управления данными во внешней памяти может различаться в разных системах (на уровне управления ресурсами и на уровне управления данными).

Необходимость в буферизации данных и как следствие реализация функции управления буферами оперативной памяти обусловлено тем, что объем оперативной памяти значительно меньше объема внешней памяти.

Буферы – это области оперативной памяти, предназначенные для быстрого обмена данными между внешней и оперативной памяти. В них временно хранятся фрагменты БД.

Механизм транзакции используется в СУБД для поддержания целостности.

Транзакцией называется неделимая последовательность операций над данными, которая отслеживается СУБД от начала до конца.

Транзакции присуще три основных свойства:

* Атомарность – выполняются все входящие в транзакцию операции или не одна.
* Сериализуемость – отсутствует взаимное влияние выполняемых в одно время транзакций.
* Долговечность – даже крах системы не приводит к утрате результатов зафиксированных транзакций.

Контроль транзакций важен в однопользовательских и многопользовательских СУБД, где транзакции могут быть запущены параллельно. В таком случае говорят о сериализуемости транзакции.

Составляется план по их выполнению, таким образом чтобы суммарный эффект реализации транзакции был эквивалентен эффекту их последовательного выполнения.

При параллельном выполнении могут возникать конфликты (блокировки), разрешение которых является функцией СУБД. При обнаружении таких случаев проводится «откат» путем отмены изменений, внесённых в базу.

Ведение журнала изменений ведётся для обеспечения надёжности хранения данных при сбоях и отказах.

Журнал СУБД – особая БД или часть основной БД, непосредственно недоступная пользователю и используемая для записи изменений в БД.

Целостность БД – это свойство, означающее, что в базе содержится полная, непротиворечивая и адекватно отображающая предметную область информации.

Поддержание целостности обеспечивается с помощью обеспечения целостности в виде условий, которым должны удовлетворять хранимые в базе данные.

Обеспечение безопасности СУБД достигается шифрованием данных, защитой паролем, управлением уровнем доступа к БД и ее элементам.

Модели Данных

К основным моделям представления данных относят:

* Иерархическую
* Сетевую
* Табличную или реляционную модель
* Пост реляционную
* Многомерную
* Объектно-ориентированную
* Нереляционную

Иерархическая

В иерархической модели связи между данными можно представить в виде дерева (упорядоченного графа).

http://images.myshared.ru/28/1304455/slide\_4.jpg

Для описания структуры иерархической БД на языке программирования используют тип данных дерево.

Тип дерево является составным, который включает в себя подтипы, которые тоже являются типом дерево 🌲. Элементарные типы, входящие в дерево называются связи.

Простая запись может быть одного типа данных, например число, а может объединять совокупность типов, например целая, строку данных и указатель.

Корневым называется тип, который имеет подчиненные типы, а сам подтипом не является.

Подчинённый тип является потомком по отношению к родителю или предку.

Иерархическая БД представляет собой упорядоченную совокупность экземпляров данных типа дерево, содержащих записи.

В иерархических СУБД может использоваться терминология, с помощью которой применяется группа методов:

* Представление линейным списком, последовательным распределением памяти.
* Представление связанными линейными и списками с использованием указателей и справочников.

К основным операциям манипулирования иерархическими данными относят:

* Поиск указанного экземпляра.
* Переход от одного дерева к другому.
* Переход от одной записи к другой внутри дерева.
* Вставка, удаление записей и т.д.

В иерархических моделях контроль целостности поддерживается автоматически т.е. потомок не может существовать без родителя, у некоторых родителей может не быть потомков. Между деревьями контроль целостности отсутствует.

К достоинствам иерархической относят:

* Эффективное использование памяти компьютера.
* Неплохие показатели времени выполнения основных операций.

Недостатком является громосткость модели со сложными связями и сложность понимания.

Сетевая модель

Сетевая модель данных позволяет отображать разнообразные взаимосвязи элементов данных в виде произвольного графа

<https://avatars.mds.yandex.net/i?id=b3e9290139ea2d8952ed779a9c679971_l-5306229-images-thumbs&n=13>

Для описания схемы сетевой БД используют две группы типов

* «Запись»
* «Связь»

Тип связь определяется для двух типов записи предка и потомка.

Сетевая БД состоит из набора записей и набора соответствующих связей. Ограничения на количество связей не накладывается.

К числу важнейших операций манипулирования данными сетевого типа относят:

* Поиск записей
* Переход от предка к первому потомку
* Переход от потомка к предку
* Создание, удаление, обновление записей и т.д.

Достоинством сетевой модели является возможность эффективной реализации по показателям затрат памяти и оперативности. По сравнению с иерархической позволяет устанавливать произвольные связи.

Недостатком сетевой модели является высокая сложность и сложность схемы БД и сложность для понимания.

Широкое распространение не получила.

Реляционная модель

Отношение представляет собой множество элементов, называемых кортежами.

<https://avatars.mds.yandex.net/i?id=bb2374bbd1e11a6e0f410633d4ba072d_l-4522636-images-thumbs&n=13>

Каждая строка таблицы имеет одинаковую структуру и состоит из полей.

Строкам таблиц соответствуют картежи (строки) а столбцам атрибут.

С помощью таблиц удобно описать деление одного объекта и связи между ними.

Достоинства табличной модели:

* Простота
* Удобность физической реализации
* Понятность

Основными недостатками реляционной модели данных являются:

* Отсутствие стандартных средств идентификации отдельных записей.
* Сложность описания иерархических и сетевых связей.

Примеры табличной модели:

* Языки группы d-Base, fox-base, paradox fox-pro, Microsoft Access.
* Объектно-реляционная – Oracle.

Пост реляционная модель

Классическая табличная модель предполагает неделимость данных хранящихся в таблице.

Это означает, что данные в таблице находятся в первой нормальной форме (1НФ).

Существует ряд случаев, когда это ограничение мешает эффективной реализации приложений.

Пост реляционная модель данных

Представляет собой расширенную табличную модель, снимающую ограничение неделимости данных.

Например имеются две таблицы «товары» и «поставщики», но для работы неудобно использовать их отдельно. Существует данные товары поставщиков, здесь данные хранятся более эффективно, потому что при обработке не нужно проводить операцию соединения таблиц.

Помимо обеспечения вложенности полей модель поддерживает многозначные ассоциированные поля.

Совокупность ассоциированных полей называется ассоциацией, при этом в строке первое значение одного столбца ассоциаций соответствует первым значениям всех других столбцов. Аналогичным образом связаны все другие значения столбцов и т.д.

Таким образом структура данных и таблиц имеет большую гибкость.

При использовании не ненормализованных данных возникает проблема поддержания целостности и непротиворечивости данных. Проблема решается использованием хранимых процедур.

Достоинством пост реляционной модели является возможность представления совокупности связанных реляционных таблиц одной пост реляционной. Это обеспечивает:

* Высокую наглядность представления информации
* Повышение эффективности обработки информации

Недостатком такой модели является сложность решения проблемы обеспечения целостности и непротиворечивости хранимых данных.

Многомерная модель

Многомерный подход к представлению данных в базе появился практически одновременно с табличным, но сначала реально работающих многомерных СУБД было мало. С середины 90х они получили широкое развитие.

Можно выделить два направления развития таких моделей:

* Системы оперативной транзакционной обработки.
* Системы поддержки принятия решений.

При оперативной транзакционной обработке чаще применялись табличные модели.

Для аналитической обработки данных они показали себя неповоротливыми и недостаточно гибкими.

Многомерные СУБД являются узкоспециализированными предназначенными для интерактивной аналитической обработки информации.

Основные понятия используемые в таких СУБД:

* Агрегируемость - обобщение информации на разных ее уровнях, степень общения зависит от уровня пользователя.
* Историчность – предполагает обеспечение высокого уровня статичности (неизвестности) данных и их взаимосвязей в зависимости от времени. Временная привязка необходима для частого выполнения запросов по периодам времени.
* Прогнозируемость – подразумевает задание функций прогнозирования и применения их к различным временным интервалам.

Многомерность модели данных означает не многомерность визуализации цифровых данных, многомерное логическое представления структура информации при опрсонии и операции. По сравнению с предыдущим моделям обладает высокой наглядностью и информативностью.

Объектно-ориентированная модель

В объектно-ориентированной модели представления данных имеет возможность идентифицировать отдельные записи БД. Между записями и функциями из обработки устанавливается взаимосвязь, подобно механизмам в объектно-ориентированных языках программирования.

Структура объектно-ориентированной БД графически представляется в виде дерева, узлами которого являются объекты.

Основным достоинством по сравнению с табличной моделью является отображение информации о сложных взаимосвязях объекта.

Недостатком – понятийная сложность, неудобство обработки данных и низкая скорость выполнения запросов.

Нереляционная модель (Oracle No SQL, berkly DB, Mongo DB, NoKach)

Поддерживают высокую разделяемость данных, обеспечивая беспрецедентное горизонтальное масштабирование, недостижимое при использовании других типов БД. Появились в 00х годах попытка решить проблему масштабируемости (расширение) при сохранении простоты и гибкости.

Поисковые БД (noSQL) предназначены для аналитики частично структурированных данных.

В зависимости от модели и способов распределенности выделяют:

* Модели типа «ключ»-«значения»
* Документно-ориетированные
* Семейство столбцов
* Графовые

БД noSQL широко используются в веб- приложениях реального времени и больших данных, поскольку их основным преимуществом является высокая доступность и масштабируемость.

Не реляционные БД размещают информацию в коллекциях документов .json. Модель данных не требует таблиц и связей.

Самая популярная noSQL включена в веб-сайты Google, Yahoo, Amazon.

Основные причины появления:

* Масштабирование (расширение).
* Работа с данными ускорилась.
* Избавление от ограниченности реляционной БД.

Структурированный язык запросов SQL

Стандартный язык для работы с табличным БД.

SQL появился в 70х годах 20го века, позже принял в качестве стандарта для работы с табличными БД.

Язык сначала юридически , а затем и фактически применяется для реляционных БД. В основе языка SQL лежат определения математического аппарата линейной алгебры.

Язык SQL является непроцедурным ЯП, то есть он ориентирован на обработку записей.

Язык ориентирован на множество записей, в результате работы над отношениями также образуются отношения.

Язык SQL состоит из двух частей:

* DDL язык определения данных – используется для определения основных объектов БД, так же сюда входит:
  + Ограничения, накладываемые на элементы БД.
  + Определение связей между отношениями.
* DML – язык обработки данных – используется для выборки данных при заданных условиях, управления данными

Так же определяют интерактивный и встроенный SQL.

* Интерактивный – применяется для интерактивной обработки информации в БД
* Встроенный – служит для встраивания запросов в другие ЯП (C++, Delphi, C#). Это делает использование таких средств наиболее эффективным, гибким и удобным. Однако необходим интерфейс с этим ЯП.

Верхнюю строку отношения называют заголовком отношения

Количество кортежей – кардинальными числом отношения

Количество атрибутов – степенью отношений.

R в степени n, m снизу.

В отношении всегда должен быть первичный ключ, то есть атрибут или набор атрибутов, значения в которых не повторяются. Или уникальный индекс для каждой строки.

В отношении всегда имеется внешний ключ для связи с другим отношением, значения должны совпадать.

Домен-отношения – это набор значений из которого могут выбираться значения атрибутов, несколько атрибутов могут находиться на одном домене. Операции манипулирования данными, как правило, проводятся на одном домене.

В отношении кортежи не упорядочены так же как и атрибут, нет такого понятия как первый, второй кортежей.

Упорядоченность кортежей осуществляется только по ключу, то есть в отношении всегда есть запись, позволяющая однозначно идентифицировать кортеж.

Типы данных SQL

* Текстовые строки, тип char – поддерживает строки длиной до 2000 символов постоянной длинны
* Barchar – строки переменной длинны, после указания типа, в квадратных скобках размер. Строки переменной длинны удобно использовать, так как в памяти забивается каждый раз конкретное количество символов вводимых в поле.
* Числовые данные – к силовым типам относят:
  + Float (real) – вещественный тип данных.
  + Decimal – десятичный тип данных.
  + Number – действительные числа.
  + Double precision – числа с двойной точностью.
* Date/time – данные формата дата/время, их диапазон, например в Oracle, изменяется от 1 января 4712 д.н.э. до 31 декабря 4712 н.э., если время не указано, принимается время полуночи. Существует специальная арифметика дат и времён.
* Null - обозначает неопределенные или пропущенные значения.

«Атрибут имеет значение Null» - означает, если значение пропущено, отсутствует или неизвестно.

* Int – целые от -2 в 31 до 2 в 32

Основные правила Null

* В агрегируемых функциях Null игнорируется.
* Для того чтобы определить, является ли значение неопределенным, используют операторы «is Null» или «is Not Null»
* Все операторы, кроме оператора конкотенации, возвращают Null, если один из операндов Null.
* Операторы булевой логики, расширяются до трёхзначной – истина , ложь, неизвестно.

Ключевые слова, предоставления, разделители

Ключевые слова – это операторы языка SQL, которые не могут использоваться в качестве наименований таблиц строк столбцов.

Предложения языка SQL начинаются с ключевого слова и представляют собой запись.

К разделителям относят:

* Квадратные скобки – обозначают необязательность конструкции.
* Многоточие – обозначает , что указанная конструкция может повториться некоторое число раз.
* Вертикальная черта – обозначает что может использоваться один из элементов списка.
* Фигурные скобки – список элементов.
* Двоеточие, знак равенства – определения.

Выборка данных

Основным оператором выборки данных в SQL является оператор «select». После него указываются через запятую наименования полей таблицы из которой производится выборка.

From (из) – после него указываются наименования таблиц.

В конструкции также могут использовать следующие операторы:

* Where (где?) <условие>.
* Having <условие>.
* Group by <формирование группы> группировка по
* Order by <наименование поля> упорядочивание по.

Если данные в результате выборки, могут быть получены с повторяющимися значениями, то используют ключевое слово «distinct» - отличный от.

Для объединения таблиц в результате запроса осуществляется с помощью «union».

Простейшие запросы на выборку

Имеется таблица студент, в которой содержится данные, которой содержится данные (id, фамилия, имя, отчество, курс, дата рождения, наименование института, город проживания, размер стипендии)

Необходимо получить фамилию и имя студента

Select фамилия, имя

From студент;

Если необходимо выбрать города проживания студентов

Select город проживания

From студент;

В результате несколько одинаковых городов. Для исключения повторяющихся записей, запишем запрос в следующем виде:

Select distinct город

From студент;

Для того чтобы вернуть назад удаленные значения используют ключевое слово «all».

Для того чтобы осуществить выбор по заданному условию используем ключевое слово «Where».

Выбрать имена студентов, обучающихся на втором и третьем курсе.

Select distinct имя

From студент

Where курс =2 and курс=3;

Выбрать имена и фамилии студентов, проживающих в Воронеже.

Select имя, фамилия

From студент

Where город проживания= 'Воронеж';

Выбрать имя и фамилию студентов у которых стипендия равна 2000

Select имя

From студент

Where стипендия= 2000;

Выбрать фамилии студентов родившихся в промежутке с 2005 по 2006 год рождения.

Select Фамилия

From студент

Where год рождения= 2005 between 2006;

Выбрать даты рождения студентов обучающихся на третьем курсе

Select год рождения

From студент

Where курс=3;

Есть таблица экзаменационные оценки студентов (id студента, наименование предмета, дата сдачи, оценка)

Выбрать оценки студентов по предмету математика.

Select оценка

From экзаменационные оценки

Where наименование предмета='математика';

Для того чтобы получить все данные из таблицы необязательно перечислять все наименования всех полей, можно использовать шаблон «\*».

Например, получить все данные из таблицы экзаменационные оценки.

Select \*

From экзаменационные оценки;

Выбрать из таблицы студент фамилии студентов первого, третьего и четвертого курсов.

Select фамилия

From студент

Where курс in (1,3,4);

Для того чтобы указать в условии запроса значения, которые не входят в заданный список, используют ключевое слово not

Выбрать фамилии студентов, которые не являются первого, третьего и четвертого курсов.

Select фамилия

From студент

Where курс not in (1,3,4);

Выбрать наименования предметов по которым на экзамене не было двоек.

Select предметы

From экзаменационные оценки

Where оценка in (3,4,5);

Необходимо получить фамилии студентов, которые начинаются на букву «а».

Select фамилия

From студент

Where фамилия like 'А\*';

Звёздочка или процент обозначает несколько символов обозначает несколько символов указанных в шаблоне.

Знак подчеркивания обозначает один символ, указанных в шаблоне.

Выбрать наименования предметов, которые начинаются на букву «М» и «Р».

Select предмет

From экзаменационные оценки

Where наименование предмета like 'М\*' or 'R\*';

Для того чтобы определить имеется ли значение в поле таблиц, используют оператор Null.

Получить список имён студентов у которых неизвестна дата рождения

Select имя

From студент

Where дата рождения is Null;

Выбрать наименования предметов которые сдавались студентами на сессии.

Select наименование предмета

From экзаменационные оценки

Where оценка is not Null;

SQL встроенные функции и преобразования данных

В SQL реализованы операторы преобразования данных в строенные функции, предназначенные для работы со значениями строк и столбцов и константами в выражении.

Их применение допустимо везде, где можно использовать наименование столбцов.

Созданные столбцы становятся виртуальными столбцами.

Нужно выбрать фамилии студентов с указанием констант

Select ' фамилия ' , фамилия

'имя' , имя, 1000

From студент;

(Картинка с доски)

Так в запросах можно осуществлять преобразования данных top char(), to number(), to Date().

В запросах так же можно использовать арифметические операции (+-\*/ унарный минус)

Так же можно применять конкатенации.

В SQL можно вкладывать запросы в запрос

Нужно узнать оценки студента по фамилии Петров, не зная его номер

Select\*

From exam\_marx

Where student\_Id=(select student id

From student

Where surname=»Петров»);

Как работает запрос SQL со связанным подзапросом?

* Выбирается строка из таблицы, имя которой указано во внешнем запросе
* Выполняется подзапрос и полученное значение применяется для анализа этой строки и условия предложения Where внешнего запроса
* По результату оценки этого условия принимается решение о включении или не включении строки в состав выходных данных
* Процедура повторяется для следующей строки таблицы внешнего запроса

В результате работы подзапроса должно получаться единственное значение

Для гарантии получения единственного значения можно использовать оператор distinct

Одним из видов функций которые позволяют получать единственное значение являются агрегирующие функции

Оператор In широко применяется в подзапросах, он задаёт список значений с которыми сравниваются другие значения для определения истинности задаваемого этим оператором предиката

Данные обо всех оценках (таблица exam\_marcs) студентов из Воронежа можно выбрать с помощью следующего запроса

Select\*

From exam\_marck

Where student \_id in

(Select student\_id

From student

Where City= «воронеж»);

Подзапросы можно применять внутри предложения Having

Пусть требуется определить количество предметов обучения с оценкой, превышающей среднее значение оценки студента с идентификатором 301:

Select count (distinct subj\_id), mark

From exam\_marks

Group by mark

Having mark>

(Select Avg (Mark)

From exam\_marks

Where student \_id = 301);

При использовании подзапросов во внутреннем запросе можно ссылаться на таблицу, имя которой указано в предложении FROM внешнего запроса. В этом случае такой связанный подзапрос выполняется по одному разу для каждой строки таблицы основного запроса.

Пример: выбрать сведения обо всех предметах обучения, по

которым проводился экзамен 20 января 1999 г.

SELECT \*

FROM SUBJECT SU

WHERE '20/01/1999' IN

(SELECT EXAM\_DATE

PROM EXAM\_MARKS EX

WHERE SU.SUBJ\_ID = EX.SUBJ\_ID);

В некоторых СУБД для выполнения этого запроса может

потребоваться преобразование значения даты в символьный

тип. В приведенном запросе su и ЕХ являются псевдонимами

(алиасами), то есть специально вводимыми именами, которые

могут быть использованы в данном запросе вместо настоящих

имен. В приведенном примере они используются вместо имен

таблиц SUBJECT и EXAM\_MARKS.

Эту же задачу можно решить с помощью операции соединения таблиц:

SELECT DISTINCT SU.SUBJ\_ID, SUBJ\_NAME, HOUR, SEMESTER

FROM SUBJECT FIRST, EXAM\_MARKS SECOND

WHERE FIRST.SUBJ\_ID = SECOND.SUBJ\_ID

AND SECOND.EXAM\_DATE = '20/01/1999';

В этом выражении псевдонимами (алиасами) таблиц являются имена FIRST и SECOND.

Можно использовать подзапросы, связывающие таблицу со

своей собственной копией.

Например, надо найти идентификаторы, фамилии и стипендии студентов, получающих стипендию выше средней на курсе, на котором они учатся.

SELECT DISTINCT STUDENT\_ID, SURNAME, STIPEND

FROM STUDENT El

WHERE STIPEND >

(SELECT AVG(STIPEND)

FROM STUDENT E2

WHERE El.KURS = E2.KURS);

Тот же результат можно получить с помощью следующего запроса:

SELECT DISTINCT STUDENT\_ID, SURNAME, STIPEND

FROM STUDENT El,

(SELECT KURS, AVG (STIPEND) AS AVG\_STIPEND

FROM STUDENT E2

GROUP BY E2.KURS) E3

WHERE El.STIPEND > AVG\_STIPEND AND El.KURS=E3.KURS;

Обратите внимание — второй запрос будет выполнен гораздо

быстрее. Дело в том, что в первом варианте запроса агрегирующая функция AVG выполняется над таблицей, указанной в подзапросе, для каждой строки внешнего запроса.

В другом варианте вторая таблица (алиас Е2) обрабатывается агрегирующей

функцией один раз, в результате чего формируется вспомогательная таблица (в запросе она имеет алиас ЕЗ), со строками которой

затем соединяются строки первой таблицы (алиас Е1). Следует

иметь в виду, что реальное время выполнения запроса в большой

степени зависит от оптимизатора запросов конкретной СУБД.

Связанные подзапросы в HAVING

Предложение GROUP BY позволяет группировать выводимые SELECT-запросом записи по значению некоторого поля. Использование предложения HAVING позволяет при выводе осуществлять фильтрацию таких групп.

Предикат предложения HAVING оценивается не для каждой строки результата, а для каждой группы выходных записей, сформированной предложением GROUP BY внешнего запроса.

Пусть, например, необходимо по данным из таблицы

EXAM\_MARKS определить сумму полученных студентами оценок

(значений поля MARK), сгруппировав значения оценок по датам

экзаменов и исключив те дни, когда число студентов, сдававших в течение дня экзамены, было меньше 10.

SELECT EXAM\_DATE, SUM(MARK)

FROM EXAM\_MARKS A

GROUP BY EXAM\_DATE

HAVING 10 <

(SELECT COUNT(MARK)

FROM EXAM\_MARKS В

WHERE A.EXAM\_DATE = В.EXAM\_DATE);

Подзапрос вычисляет количество строк с одной и той же датой, совпадающей с датой, для которой сформирована очередная группа основного запроса.

Использовании операторов EXIST

Генерирует значения истина или ложь подобно булевому выражению используя подзапросы в качестве аргумента.

Оператор EXIST оценивает результат как истинный если подзапрос возвращает хотя бы одно значение, в противном случае результат подзапроса будет ложный, оператор EXIST не может принимать значения неизвестный.

Нужно извлечь из таблицы EXAM\_MARKS данные о студентах, получивших хотя бы одну неудовлетворительную оценку:

SELECT DISTINCT STUDENT\_ID

FROM EXAM\_MARKS A

WHERE EXISTS

(SELECT \* FROM EXAM\_MARKS В

WHERE MARK < 3

AND B.STUDENT\_ID = A.STUDENT\_ID);

Получить идентификаторы предметов обучения, экзамены по которым сдавались не одним, а несколькими студентами:

SELECT DISTINCT SUBJ\_ID

FROM EXAM\_MARKS A

WHERE EXISTS

(SELECT \*

FROM EXAM\_MARKS В

WHERE A.SUBJ\_\_ID = B.SUBJ\_ID -связываем таблицы по данным полям

AND A.STUDENT\_ID < > В.STUDENT\_ID);

<>-это !=

Для того что бы найти одного студента нужно перед EXIST поставить оператор «Not».

Из таблицы STUDENT требуется извлечь строки для каждого студента, сдавшего более одного предмета.

SELECT \*

FROM STUDENT FIRST

WHERE EXISTS

(SELECT SUBJ\_ID

FROM EXAM\_MARKS SECOND

GROUP BY SUBJ\_ID

HAVING COUNT (SUBJ\_ID) > 1

WHERE FIRST.STUDENT ID = SECOND.STUDENT ID);

Операторы сравнения с множеством значений

IN, ANY, All

Операторы сравнения с множеством значений имеют следующий смысл.

|  |  |
| --- | --- |
| IN | Равно любому из значений, полученных во внутреннем запросе. |
| NOT IN | Не равно ни одному из значений, полученных во внутреннем запросе. |
| = ANY | То же, что и IN, соответствует логическому оператору OR. |
| > ANY, > = ANY | Больше, чем (либо больше или равно) любое полученное число. Эквивалентно > или > = для самого меньшего полученного числа. |
| < ANY, < = ANY | Меньше, чем (либо меньше или равно) любое полученное число. |
| =ALL | Равно всем полученным значениям. Эквивалентно логическому оператору AND |
| >ALL, >=ALL | Больше, чем (либо больше или равно) все полученные числа. Эквивалент> или > = для самого  большего полученного числа. |
| <ALL, <=ALL | Меньше, чем (либо меньше или равно) все полученные числа. Эквивалентно <или< = самого  меньшего полученного числа. |

Следует иметь в виду, что в некоторых СУБД поддерживаются не все из этих операторов!

Выбрать сведения о студентах, проживающих в городе, где

расположен университет, в котором они учатся.

SELECT \*

FROM STUDENT S

WHERE CITY = ANY

(SELECT CITY

FROM UNIVERSITY U

WHERE U.UNIV\_ID = S.UNIV\_ID);

Другой вариант этого запроса:

SELECT \*

FROM STUDENT S

WHERE CITY IN

(SELECT CITY

FROM UNIVERSITY U

WHERE U.UNIV\_ID = S. UNIV\_ID);

Оператор ALL, как правило, эффективно используется с неравенствами, а не с равенствами, поскольку значение равно всем, которое должно получиться в этом случае в результате выполнения подзапроса, может иметь место, только если все результаты идентичны.

Такая ситуация практически не может быть реализована,

Так как если подзапрос генерирует множество различных значений, то никакое одно значение не может быть равно сразу всем значениям в обычном смысле.

В SQL выражение < > ALL реально означает не равно ни одному из результатов подзапроса.

Подзапрос, выбирающий данные о названиях всех университетов с рейтингом более высоким, чем рейтинг любого университета Воронежа:

SELECT \*

FROM UNIVERSITY

WHERE RATING > ALL

(SELECT RATING

FROM UNIVERSITY

WHERE CITY = 'Воронеж');

В этом запросе вместо ALL можно использовать ANY (проанализируйте, как в этом случае изменится смысл приведенного запроса):

SELECT \*

FROM UNIVERSITY

WHERE NOT RATING <= ANY

(SELECT RATING

FROM UNIVERSITY

WHERE CITY = 'Воронеж');

Особенности применения операторов ANY, ALL, EXISTS для NULL-значений

Необходимо иметь в виду, что при обработке NULL-значений следует учитывать различие реакции на них операторов EXISTS, ANY и ALL.

Когда правильный подзапрос не генерирует никаких выходных данных, оператор ALL автоматически принимает значение истина, а оператор ANY — значение ложь.

Следующий запрос

SELECT \*

FROM UNIVERSITY

WHERE RATING > ANY

(SELECT RATING

FROM UNIVERSITY

WHERE CITY = 'New York');

не генерирует выходных данных, в то время как запрос

SELECT \*

FROM UNIVERSITY

WHERE RATING > ALL

(SELECT RATING

FROM UNIVERSITY

WHERE CITY = 'New York');

полностью воспроизведет таблицу UNIVERSITY.

Использование NULL-значений создает определенные проблемы для рассматриваемых операторов. Когда в SQL сравниваются два значения, одно из которых NULL-значение, результат принимает значение UNKNOWN (неизвестно).

Предикат UNKNOWN, так же, как и FALSE-предикат, создает ситуацию, когда строка не включается в состав выходных данных, но результат при этом будет различен для разных типов запросов, в зависимости от использования в них ALL или ANY вместо EXISTS. Рассмотрим в качестве примера две реализации запроса:

Найти все данные об университетах, рейтинг которых меньше рейтинга любого университета в Москве.

SELECT \*

FROM UNIVERSITY

WHERE RATING < ANY

(SELECT RATING

FROM UNIVERSITY

WHERE CITY = 'Москва');

SELECT \*

FROM UNIVERSITY A

WHERE NOT EXISTS

(SELECT \*

FROM UNIVERSITY В

WHERE A.RATING >= B.RATING

AND B.CITY = 'Москва');

При отсутствии в таблицах NULL оба эти запроса ведут себя совершенно одинаково. Пусть теперь в таблице UNIVERSITY есть строка с NULL-значениями в столбце RATING. В версии запроса с ANY в основном запросе, когда выбирается поле RATING с NULL, предикат принимает значение UNKNOWN и строка не включается в состав выходных данных. Во втором же варианте запроса, когда NOT EXISTS выбирает эту строку в основном запросе, NULL-значение используется в предикате подзапроса, присваивая ему значение UNKNOWN. Поэтому в результате выполнения подзапроса не будет получено ни одного значения, и подзапрос примет значение ложь. Это в свою очередь сделает NOT EXISTS истинным, и, следовательно, строка с NULL-значением в поле RATING попадет в выходные данные. По смыслу запроса такой результат является неправильным, так как на самом деле рейтинг университета, описываемого данной строкой, может быть и больше рейтинга какого-либо московского университета (он просто неизвестен). Указанная проблема связана с тем, что значение EXISTS всегда принимает значения истина или ложь, и никогда — UNKNOWN .

Это является доводом для использования в таких случаях оператора ANY вместо EXISTS.

При отсутствии Null значений- значений оператор EXIST может быть использован вместо ANY и ALL.

Так же вместо EXIST и NOT EXIST могут быть использованы те же самые запросы, но с использованием COUNT(\*) в предложении SELECT.

Например запрос

SELECT \* FROM UNIVERSISTY A

WHETE NOT EXIST

(SELECT\*FROM UNIVERSISTY B

WHERE A.RATING>= B.RATING and B.CITY =»Москва»);

Может быть представлен в следующем виде:

Select \* FROM UNOVERSITY A

WHERE 1>

Использование COUNT вместо EXISTS

При отсутствии NULL-значений оператор EXISTS может быть использован вместо ANY и ALL.

Также вместо EXISTS и NOT EXISTS могут быть использованы те же самые подзапросы, но с использованием COUNT (\*) в предложении SELECT. Например, запрос

SELECT \* FROM UNIVERSITY A

WHERE NOT EXISTS

(SELECT \* FROM UNIVERSITY В

WHERE A.RATING > = B.RATING AND B.CITY = 'Москва');

может быть представлен и в следующем виде:

SELECT \* FROM UNIVERSITY A

WHERE 1 >

(SELECT COUNT (\*) FROM UNIVERSITY В

WHERE A.RATING > = B.RATING AND B.CITY = 'Москва');

Оператор объединения UNION

Оператор UNION используется для объединения выходных данных двух или более SQL-запросов в единое множество строк и столбцов.

Например, для того чтобы получить в одной таблице фамилии и идентификаторы студентов и преподавателей из Москвы, можно использовать следующий запрос:

SELECT 'Студент ', SURNAME, STUDENT\_ID

FROM STUDENT

WHERE CITY = 'Москва'

UNION

SELECT 'Преподаватель', SURNAME, LECTURER\_ID

FROM LECTURER

WHERE CITY = 'Москва';

Обратите внимание на то, что символом «;» оканчивается только последний запрос. Отсутствие этого символа в конце SELECT-запроса означает, что следующий за ним запрос так же, как и он сам, является частью общего запроса с UNION.

Использование оператора UNION возможно только при объединении запросов, соответствующие столбцы которых *совместимы по объединению*, то есть соответствующие числовые поля должны *иметь полностью совпадающие тип и размер,* символьные поля должны иметь точно *совпадающее количество символов*. Если NULL-значения запрещены для столбца хотя бы одного любого подзапроса объединения, то они должны быть запрещены и для всех соответствующих столбцов в других подзапросах объединения.

Устранение дублирования в UNION

В отличие от обычных запросов UNION автоматически исключает из выходных данных дубликаты строк, например, в запросе

SELECT CITY

FROM STUDENT

UNION

SELECT CITY

FROM LECTURER;

совпадающие наименования городов будут исключены.

Если все же необходимо в каждом запросе вывести все строки независимо от того, имеются ли такие же строки в других объединяемых запросах, то следует использовать во множественном запросе конструкцию с оператором UNION ALL.

Так, в запросе

SELECT CITY

FROM STUDENT

UNION ALL

SELECT CITY

FROM LECTURER;

дубликаты значений городов, выводимые второй частью запроса, не будут исключаться.

Приведем еще один пример использования оператора UNION.

Пусть необходимо составить отчет, содержащий для каждой даты сдачи экзаменов сведения по каждому студенту, получившему

максимальную или минимальную оценки.

SELECT 'МАКСОЦ', A.STUDENT\_ID, SURNAME, MARK, EXAM\_DATE

FROM STUDENT A, EXAM\_MARKS В

WHERE (A.STUDENT\_ID = B.STUDENT\_ID

AND B.MARK =

(SELECT MAX (MARK)

FROM EXAM\_MARKS С

WHERE C.EXAM\_DATE = B.EXAM\_DATE))

UNION ALL

SELECT 'МИНОЦ’, A.STUDENT\_ID, SURNAME, MARK, EXAM\_DATE

FROM STUDENT A, EXAM\_MARKS В

WHERE (A.STUDENT\_ID = B.STUDENT\_ID

AND B.MARK =

(SELECT MIN (MARK)

FROM EXAM\_MARKS С

WHERE C.EXAM\_DATE = В.EXAM\_DATE));

Для отличия строк, выводимых первой и второй частями запроса, в них вставлены текстовые константы 'Макс оц' и 'Мин оц'.

В приведенном запросе агрегирующие функции используются в подзапросах. Это является нерациональным с точки зрения времени, затрачиваемого на выполнение запроса. Более эффективна форма запроса, возвращающего аналогичный результат:

SELECT 'МаКСОЦ', A.STUDENT\_ID, SURNAME, E.MARK, E.EXAM\_DATE

FROM STUDENT A,

(SELECT B.STUDENT\_ID, B.MARK, B.EXAM\_DATE

FROM EXAM\_MARKS B,

(SELECT MAX (MARK) AS MAX\_MARK, C.EXAM\_DATE

FROM EXAM\_\_MARKS С

GROUP BY C.EXAM\_DATE) D

WHERE В.EXAM\_DATE=D.EXAM\_DATE

AND B.MARK=MAX\_MARK) E

WHERE A.STUDENT\_ID=E.STUDENT\_ID

UNION ALL

SELECT 'МИНОЦ', A.STUDENT\_\_ID, SURNAME, E.MARK, E .EXAM\_DATE

FROM STUDENT A,

(SELECT B.STUDENT\_ID, B.MARK, B.EXAM\_DATE

FROM EXAM\_MARKS B,

(SELECT MIN (MARK) AS MIN\_MARK, C.EXAM\_DATE

FROM EXAM\_MARKS С

GROUP BY C.EXAM\_DATE) D

WHERE B.EXAM\_DATE=D.EXAM\_DATE

AND B.MARK=MIN\_MARK) E

WHERE A.STUDENT\_ID=E.STUDENT\_ID;

Использование UNION с ORDER BY

Предложение ORDER BY применяется для упорядочения выходных данных объединения запросов так же, как и для отдельных запросов. Последний пример, при необходимости упорядочения выходных данных запроса по фамилиям студентов и датам экзаменов, может выглядеть следующим образом:

SELECT 'МАКСОЦ', A.STUDENT\_ID, SURNAME, E.MARK, E.EXAM\_DATE

FROM STUDENT A,

(SELECT B.STUDENT\_ID, B.MARK, B.EXAM\_DATE

FROM EXAM\_MARKS B,

(SELECT MAX (MARK) AS MAX\_MARK, C.EXAM\_DATE

FROM EXAM\_MARKS С

GROUP BY C.EXAM\_DATE) D

WHERE B.EXAM\_DATE=D.EXAM\_DATE

AND B.MARK=MAX\_MARK) E

WHERE A.STUDENT\_ID=E.STUDENT\_ID

UNION ALL

SELECT 'МИНОЦ', A.STUDENT\_ID, SURNAME, E.MARK, E.EXAM\_DATE

FROM STUDENT A,

(SELECT B.STUDENT\_ID, B.MARK, B.EXAM\_DATE

FROM EXAM\_MARKS B,

(SELECT MIN (MARK) AS MIN\_MARK, C.EXAM\_DATE

FROM EXAM\_MARKS С

GROUP BY C.EXAM\_DATE) D

WHERE В.EXAM\_DATE=D.EXAM\_DATE

AND B.MARK=MIN\_MARK) E

WHERE A.STUDENT\_ID=E.STUDENT\_ID

ORDER BY SURNAME, E.EXAM\_DATE;

Внешнее объединение

Часто бывает полезна операция объединения двух запросов, в которой второй запрос выбирает строки, исключенные первым. Такая операция называется внешним объединением.

Рассмотрим пример.

Пусть в таблице STUDENT имеются записи о студентах, в которых не указан идентификатор университета.

Требуется составить список студентов с указанием наименований университетов для тех студентов, у которых эти данные есть, не исключая тех, у которых UNIV\_ID не указан.

Формируем объединение из двух запросов, один из которых выполняет выборку студентов с названиями UNIV\_ID , а второй с NULL-значениями в поле UNIV\_ID.

SELECT SURNAME, NAME, UNIV\_NAME

FROM STUDENT, UNIVERSITY

WHERE STUDENT.UNIV\_ID=UNIVERSITY.UNIV\_ID

UNION

SELECT SURNAME, NAME, ‘Неизвестен’

FROM STUDENT

WHERE UNIV\_ID IS NULL

ORDER BY 1;

Для совместимости столбцов объединяемых запросов константу ‘Неизвестен’ во втором запросе следует дополнить пробелами так, чтобы ее длина соответствовала длине поля UNIV\_NAME, или использовать функцию CAST. В некоторых СУБД согласование типов поля и замещающей его текстовой константы осуществляется автоматически.

Соединение таблиц с использованием JOIN

Если в операторе SELECT после ключевого слова FROM указывается не одна, а две таблицы, то в результате выполнения запроса, в котором отсутствует предложение WHERE каждая строка одной таблицы будет соединена с каждой строкой второй.

Такая операция называется *декартовым произведением или полным (CROSS) соединением таблиц.*

В случае использования этой операции в ответе на запрос количество записей будет равно произведению числа записей в соединяемых таблицах, т.е. может оказаться чрезвычайно большим.

Соединение таблиц имеет смысл в том случае, если соединяются не все строки исходных таблиц, а только интересующие пользователя.

Ограничение осуществляется предложением WHERE. Таким образом SQL позволяет выводить информацию из нескольких таблиц, связывая их по значениям определенных полей.

Например, если необходимо получить фамилии студентов и для каждого студента названия университетов, расположенных в городе, где живет студент:

SELECT STUDENT.SURNAME, UNIVERSITY.UNIV\_NAME, STUDENT.CITY

FROM STUDENT, UNIVERSITY

WHERE STUDENT.CITY=UNIVERSITY.CITY;

Соединение, использующее предикаты, основанные на равенствах, называется *эквисоединением*.

Рассмотренный пример соединения таблиц относится к виду так называемого внутреннего (INNER) соединения. При этом соединяются только те строки таблиц, для которых истинным является предикат, задаваемый в предложении ON выполняемого запроса.

Приведенный выше запрос может быть записан иначе, с использованием ключевого слова JOIN.

SELECT STUDENT.SURNAME, UNIVERSITY.UNIV\_NAME,

STUDENT.CITY

FROM STUDENT INNER JOIN UNIVERSITY

ON STUDENT.CITY = UNIVERSITY.CITY;

Ключевое слово INNER в запросе может быть опущено, так как эта опция в операторе JOIN действует по умолчанию.

Рассмотренный выше случай полного соединения (декартова произведения) таблиц с использованием ключевого слова JOIN будет выглядеть следующим образом:

SELECT \* FROM STUDENT JOIN UNIVERSITY;

что эквивалентно

SELECT \* FROM STUDENT, UNIVERSITY;

Заметим, что в СУБД Oracle задаваемый стандартом языка SQL оператор JOIN не поддерживается.

Операции соединения таблиц посредством ссылочной целостности

Информация в таблицах STUDENT и EXAM\_MARKS уже связана посредством поля STUDENT\_ID. В таблице STUDENT поле STUDENT\_ID является первичным ключом, а в таблице EXAM\_MARKS — ссылающимся на него внешним ключом.

Состояние связанных таким образом таблиц называется *состоянием ссылочной целостности.*

В данном случае ссылочная целостность этих таблиц подразумевает, что каждому значению поля STUDENT\_ID в таблице EXAM\_MARKS обязательно соответствует такое же значение поля STUDENT ID в таблице STUDENT.

Другими словами, в таблице EXAM\_MARKS не может быть записей, имеющих идентификаторы студентов, которых нет в таблице STUDENT. Стандартное применение операции соединения состоит в извлечении данных в терминах этой связи.

Чтобы получить список фамилий студентов с полученными ими оценками и идентификаторами предметов, можно использовать следующий запрос:

SELECT SURNAME, MARK, SUBJ\_ID

FROM STUDENT, EXAM\_MARKS

WHERE STUDENT.STUDENT\_ID = EXAM\_MARKS.STUDENT\_ID;

Тот же результат может быть получен при использовании в запросе для задания операции соединения таблиц ключевого слова JOIN. Запрос с оператором JOIN выглядит следующим образом:

SELECT SURNAME, MARK

FROM STUDENT JOIN EXAM\_MARKS

ON STUDENT.STUDENT\_ID = EXAM\_MARKS.STUDENT\_ID;

Хотя выше речь шла о соединении двух таблиц, можно сформировать запросы путем соединения более чем двух таблиц.

Пусть требуется найти фамилии всех студентов, получивших неудовлетворительную оценку, вместе с названиями предметов обучения, по которым получена эта оценка.

SELECT SUBJ\_NAME, SURNAME, MARK

FROM STUDENT, SUBJECT, EXAM\_MARKS

WHERE STUDENT.STUDENT\_ID = EXAM\_MARKS.STUDENT\_ID

AND SUBJECT.SUBJ\_ID = EXAM\_MARKS.SUBJ\_ID

AND EXAM\_MARKS.MARK = 2;

Toже самое с использованием оператора JOIN:

SELECT SUBJ\_NAME, SURNAME, MARK

FROM STUDENT JOIN SUBJECT JOIN EXAM\_MARKS

ON STUDENT.STUDENT\_ID = EXAM\_MARKS.STUDENT\_ID

AND SUBJECT.SUBJ\_ID = EXAM\_MARKS.SUBJ\_ID

AND EXAM\_MARKS.MARK = 2;

Как отмечалось ранее, при использовании внутреннего (INNER) соединения таблиц соединяются только те их строки, в которых совпадают значения полей, задаваемые в запросе предложением WHERE.

Однако во многих случаях это может привести к нежелательной потере информации. Рассмотрим еще раз приведенный выше пример запроса на выборку списка фамилий студентов с полученными ими оценками и идентификаторами предметов. При использовании, как это было сделано в рассматриваемом примере, внутреннего соединения в результат запроса не попадут студенты, которые еще не сдавали экзамены, и которые, следовательно, отсутствуют в таблице EXAM\_MARKS.

Если же необходимо иметь записи об этих студентах в выдаваемом запросом списке, то можно присоединить сведения о студентах, не сдававших экзамен, путем использования оператора UNION с соответствующим запросом. Например, следующим образом:

SELECT SURNAME, CAST MARK AS CHAR(1), CAST SUBJ\_ID AS CHAR (10)

FROM STUDENT, EXAM\_MARKS

WHERE STUDENT.STUDENT\_ID = EXAM\_MARKS.STUDENT\_ID

UNION

SELECT SURNAME, CAST NULL AS CHAR (1), CAST NULL AS CHAR (10)

FROM STUDENT

WHERE NOT EXISTS

(SELECT \*

FROM EXAM\_MARKS

WHERE STUDENT.STUDENT\_ID = EXAM\_MARKS.STUDENT\_ID);

(здесь функция преобразования типов CAST используется для обеспечения совместимости типов полей объединяемых запросов).

Нужный результат может быть получен и путем использования внешнего соединения, точнее, одной из его разновидностей —левого внешнего соединения, с применением которого запрос будет выглядеть следующим образом:

SELECT SURNAME, MARK

FROM STUDENT LEFT OUTER JOIN EXAM\_MARKS

ON STUDENT.STUDENT\_ID = EXAM\_

MARKS.STUDENT ID;

При использовании левого соединения расширение выводимой таблицы осуществляется за счет записей входной таблицы, имя которой указано слева от оператора JOIN.

Приведенный выше запрос может быть реализован и с применением правого внешнего соединения. Он будет иметь следующий вид:

SELECT SURNAME, MARK

FROM EXAM\_MARKS RIGHT OUTER JOIN STUDENT

ON EXAM\_MARKS.STUDENT\_ID = STUDENT.STUDENT\_ID;

Здесь таблица STUDENT, за счет записей которой осуществляется расширение выводимой таблицы, указана справа от оператора JOIN.

Иногда возникает необходимость включения в результат запроса записей из обеих (правой и левой) соединяемых таблиц, для которых не удовлетворяется условие соединения. Такое соединение называется *полным внешним соединением* и осуществляется указанием в запросе ключевых слов FULL OUTER JOIN или UNION JOIN.

Использование псевдонимов при соединении таблиц

Часто при запросе информации необходимо осуществлять соединение таблицы с ее же копией.

Например, это требуется в случае, когда нужно найти фамилии студентов, имеющих одинаковые имена. При соединении таблицы с ее же копией вводят псевдонимы (алиасы) таблицы. Запрос для поиска фамилий студентов, имеющих одинаковые имена, выглядит следующим образом:

SELECT FIRST.SURNAME, SECOND.NAME

FROM STUDENT FIRST, STUDENT SECOND

WHERE FIRST.NAME = SECOND.NAME;

В этом запросе введены два псевдонима для одной таблицы STUDENT, что позволяет корректно задать выражение, связывающее две копии таблицы.

Чтобы исключить повторения строк в выводимом результате запроса из-за повторного сравнения одной и той же пары студентов, необходимо задать порядок следования для двух значений так, чтобы одно значение было меньше, чем другое, что делает предикат асимметричным.

SELECT FIRST.SURNAME, SECOND.NAME

FROM STUDENT FIRST, STUDENT SECOND

WHERE FIRST.NAME = SECOND.NAME

AND FIRST.SURNAME < SECOND.SURNAME;

Команды манипулирования данными

В SQL для выполнения операций ввода данных в таблицу, их изменения и удаления предназначены три команды языка манипулирования данными (DML).

Это команды **INSERT** (вставить), **UPDATE** (обновить), **DELETE** (удалить).

Команда **INSERT** осуществляет вставку в таблицу новой строки. В простейшем случае она имеет вид:

INSERT INTO <имя таблицы> VALUES (<значение>, <значение>,);

При такой записи указанные в скобках после ключевого слова VALUES значения вводятся в поля добавленной в таблицу новой строки в том порядке, в котором соответствующие столбцы указаны при создании таблицы, то есть в операторе CREATE TABLE.

Например, ввод новой строки в таблицу STUDENT может быть осуществлен следующим образом:

INSERT INTO STUDENT

VALUES (101,'Иванов','Александр', 2000, 3,'Москва’, '6/10/2004', 15);

Чтобы такая команда могла быть выполнена, таблица с указанным в ней именем (STUDENT) должна быть предварительно определена (создана) командой CREATE TABLE.

Если в какое-либо поле необходимо вставить NULL-значение, то оно вводится как обычное значение:

INSERT INTO STUDENT

VALUES (101, 'Иванов', NULL, 2000, 3, 'Москва','6/10/2004', 15);

В случаях, когда необходимо ввести значения полей в порядке, отличном от порядка столбцов, заданного командой CREATE TABLE, или требуется ввести значения не во все столбцы, следует использовать следующую форму команды INSERT:

INSERT INTO STUDENT (STUDENT\_ID, CITY, SURNAME, NAME)

VALUES (101, 'Москва', 'Иванов', 'Саша');

Столбцам, наименования которых не указаны в приведенном в скобках списке, автоматически присваивается значение по умолчанию, если оно назначено при описании таблицы (команда CREATE TABLE), либо значение NULL.

С помощью команды INSERT можно извлечь значение из одной таблицы и разместить его в другой, например, запросом следующего вида:

INSERT INTO STUDENT1

SELECT \*

FROM STUDENT

WHERE CITY = 'Москва';

При этом таблица STUDENTI должна быть предварительно создана командой CREATE TABLE и иметь структуру, идентичную таблице STUDENT.

Удаление строк из таблицы осуществляется с помощью команды **DELETE.**

Следующее выражение удаляет все строки таблицы EXAM\_MARKS1.

DELETE FROM EXAM\_MARKS1;

В результате таблица становится пустой (после этого она может быть удалена командой DROP TABLE).

Для удаления из таблицы сразу нескольких строк, удовлетворяющих некоторому условию, можно воспользоваться предложением WHERE:

DELETE FROM EXAM\_MARKS1

WHERE STUDENT\_ID = 103;

Можно удалить группу строк:

DELETE FROM STUDENT1

WHERE CITY = 'Москва';

Команда **UPDATE** позволяет изменять, то есть обновлять значения некоторых или всех полей в существующей строке или строках таблицы.

Например, чтобы для всех университетов, сведения о которых находятся в таблице UNIVERSITYI, изменить рейтинг на значение 200, можно использовать конструкцию:

UPDATE UNIVERSITY1

SET RATING = 200;

Для указания конкретных строк таблицы, значения полей которых должны быть изменены, в команде UPDATE можно использовать предикат, указываемый в предложении WHERE.

UPDATE UNIVERSITY1

SET RATING = 200

WHERE CITY = 'Москва';

В результате выполнения этого запроса будет изменен рейтинг только у университетов, расположенных в Москве.

Команда UPDATE позволяет изменять не только один, но и множество столбцов. Для указания конкретных столбцов, значения которых должны быть модифицированы, используется предложение SET.

Например, наименование предмета обучения 'Математика' (для него SUBJ\_ID = 43) должно быть заменено на название 'Высшая математика', при этом идентификационный номер необходимо сохранить, но в соответствующие поля строки таблицы ввести новые данные об этом предмете обучения. Запрос будет выглядеть следующим образом:

UPDATE SUBJECT1

SET SUBJ\_NAME = ‘Высшая математика', HOUR = 36, SEMESTER = 1

WHERE SUBJ\_ID = 43;

В предложении SET команды UPDATE можно использовать скалярные выражения, указывающие способ изменения значений поля, в которые могут входить значения изменяемого и других полей.

UPDATE UNIVERSITY1

SET RATING = RATING\*2;

Например, для увеличения в таблице STUDENT1 значения поля STIPEND в два раза для студентов из Москвы можно использовать запрос:

UPDATE STUDENT 1

SET STIPEND = STIPEND\*2

WHERE CITY = 'Москва';

Предложение SET не является предикатом, поэтому в нем можно указать значение NULL следующим образом:

UPDATE UNIVERSITYl

SET RATING = NULL

WHERE CITY = 'Москва';

Применение оператора INSERT с подзапросом позволяет загружать сразу несколько строк в одну таблицу, используя информацию из другой таблицы.

В то время как оператор INSERT, использующий VALUES, добавляет только одну строку, INSERT с подзапросом добавляет в таблицу столько строк, сколько подзапрос извлекает из другой таблицы.

При этом количество и тип возвращаемых подзапросом столбцов должен соответствовать количеству и типу столбцов таблицы, в которую вставляются данные.

Например, пусть таблица STUDENT1 имеет структуру, полностью совпадающую со структурой таблицы STUDENT.

Запрос, позволяющий заполнить таблицу STUDENT1 записями обо всех

студентах из Москвы из таблицы STUDENT, выглядит следующим образом:

INSERT INTO STUDENT1

SELECT \*

FROM STUDENT

WHERE CITY = 'Москва';

Чтобы добавить в таблицу STUDENT1 сведения обо всех студентах, которые учатся в Москве, можно использовать в предложении WHERE соответствующий подзапрос.

INSERT INTO STUDENT1

SELECT \*

FROM STUDENT

WHERE UNIV\_ID IN

(SELECT UNIV\_ID

FROM UNIVERSITY

WHERE CITY = 'Москва');

Предположим, существует таблица SSTUD, в которой хранятся сведения о студентах, обучающихся в том же городе, в котором они живут. Можно заполнить эту таблицу данными из таблицы STUDENT, используя связанные подзапросы, следующим образом:

INSERT INTO SSTUD

SELECT \*

FROM STUDENT A

WHERE CITY IN

(SELECT CITY

FROM UNIVERSITY В

WHERE A.UNIV\_ID = B.UNIV\_ID);

Предположим, требуется выбрать список студентов, имеющих максимальный балл на каждый день сдачи экзаменов, и разместить его в другой таблице с именем EXAM. Это можно осуществить с помощью запроса

INSERT INTO EXAM

SELECT EXAM\_ID, STUDENT\_ID, SUBJ\_ID, MARK, EXAM\_DATE

FROM EXAM\_MARKS A

WHERE MARK =

(SELECT MAX (MARK)

FROM EXAM\_MARKS В

WHERE A.EXAM\_DATE = В.EXAM\_DATE);

Использование подзапросов с DELETE

Пусть филиал университета в Нью-Васюках ликвидирован и требуется удалить из таблицы STUDENT записи о студентах, которые там учились. Эту операцию можно выполнить с помощью запроса

DELETE

FROM STUDENT

WHERE UNIV\_ID IN

(SELECT UNIV\_ID

FROM UNIVERSITY

WHERE CITY = 'Нью-Васюки');

В предикате предложения FROM (подзапроса) нельзя ссылаться на таблицу, из которой осуществляется удаление. Однако можно ссылаться на текущую строку из таблицы, являющуюся кандидатом на удаление, то есть на строку, которая в настоящее время проверяется в основном предикате.

DELETE

FROM STUDENT

WHERE EXISTS

(SELECT \*

FROM UNIVERSITY

WHERE RATING = 401

AND STUDENT.UNIV\_ID = UNIVERSITY. UNIV\_ID);

Часть AND предиката внутреннего запроса ссылается на таблицу STUDENT. Команда удаляет данные о студентах, которые учатся в университетах, имеющих рейтинг, равный 401.

Существуют и другие способы решения этой задачи.

DELETE

FROM STUDENT

WHERE 401 IN

(SELECT RATING

FROM UNIVERSITY

WHERE STUDENT. UNIV\_ID = UNIVERSITY. UNIV\_\_ID);

Пусть нужно найти наименьшее значение оценки, полученной в каждый день сдачи экзаменов, и удалить из таблицы сведения о студенте, который получил эту оценку. Запрос будет иметь вид:

DELETE

FROM STUDENT

WHERE STUDENT\_ID IN

(SELECT STUDENT\_ID

FROM EXAM\_MARKS A

WHERE MARK=

(SELECT MIN (MARK)

FROM EXAM\_MARKS В

WHERE A.EXAM\_DATE = В.EXAMJDATE));

Так как столбец STUDENT\_ID является первичным ключом,

то удаляется единственная строка.

Если в какой-то день сдавался только один экзамен (то есть получена только одна минимальная оценка) и по какой-либо причине запись, в которой находится эта оценка, требуется оставить, то решение будет иметь вид:

DELETE

FROM STUDENT

WHERE STUDENT\_ID IN

(SELECT STUDENT\_ID

FROM EXAM\_MARKS A

WHERE MARK =

(SELECT MIN (MARK)

FROM EXAM\_MARKS В

WHERE A.EXAM\_DATE = B.EXAM\_DATE

AND 1 <

(SELECT COUNT (SUBJ\_ID)

FROM EXAM\_MARKS В

WHERE A.EXAM\_DATE = В.EXAM\_DATE)));

С помощью команды UPDATE можно применять подзапросы в любой форме, приемлемой для команды DELETE.

Например, используя связанные подзапросы, можно увеличить значение размера стипендии на 2000 в записях студентов, сдавших экзамены на 4 и 5.

UPDATE STUDENT1

SET STIPEND = STIPEND + 2000

WHERE 4 <=

(SELECT MIN (MARK)

FROM EXAM\_MARKS

WHERE EXAM\_MARKS.STUDENT\_ID = STUDENT1.STUDENT\_ID);

Другой запрос: «Уменьшить величину стипендии на 20 всем

студентам, получившим на экзамене минимальную оценку».

UPDATE STUDENT1

SET STIPEND = STIPEND - 20

WHERE STUDENT\_ID IN

(SELECT STUDENT\_ID

FROM EXAM\_MARKS A

WHERE MARK =

(SELECT MIN (MARK)

FROM EXAM\_MARKS В

WHERE A.EXAM DATE = B.EXAM DATE));

Защита данных

Тема Определение прав доступа пользователей к данным

Пользователи и привилегии. Стандартные привилегии

SQL обычно применяется в многопользовательских средах, которые требуют разграничения прав пользователей с точки зрения доступа к данным и прав на выполнение с ними тех или иных манипуляций. Для этих целей в SQL реализованы средства, позволяющие устанавливать и контролировать привилегии пользователей баз данных.

Каждый пользователь в среде SQL имеет специальное имя (идентификатор), с помощью которого осуществляется идентификация пользователя с целью установки и определения его прав с точки зрения доступа к данным. Каждая посланная к СУБД команда SQL-запроса ассоциируется СУБД с идентификатором доступа к данным конкретного пользователя.

Пользователь определяется с помощью следующей команды:

CREATE USER <username> IDENTIFIED BY <password>

Пользователь становится известен БД, но не может выполнять никаких операций.

Удаление пользователя производится командой:

DROUP USER<username>

Назначаемые привилегии –это разрешение на выполнение указанным пользователем данной команды над определенным объектом БД.

Имеется несколько типов привилегий, соответствующих нескольким типам операций.

Привилегии даются и отменяются двумя командами SQL:

GRANT –установка привилегий;

REVOKE-отмена привилегий.

Привилегии над объектом означают, что пользователь имеет право (привилегию) на выполнение данной команды только на определенном объекте в БД.

Привилегии объекта связаны одновременно и с пользователями, и с таблицами БД. То есть привилегия дается определенному пользователю в определенной таблице.

Это может быть, как базовая таблица, так и представление (т.е. таблица, полученная в результате запроса).

Пользователь, создавший таблицу любого вида, является **владельцем** этой таблицы.

То есть имеет все привилегии, относящиеся к таблице, может передавать привилегии на работу с этой таблицей другим пользователям.

Могут быть назначены следующие привилегии:

SELECT, INSERT, UPDATE, DELETE и другие.

Могут быть нестандартные привилегии объекта (INDEX, SYNONIM, ALTER, EXECUTE).

Назначение пользователям этих привилегий осуществляется с помощью команды GRANT.

Команда GRANT, REVOKE

Пользователь, являющийся владельцем таблицы STUDENT, может передать другому пользователю (IVANOV) привилегию SELECT с помощью следующей команды:

GRANT SELECT ON STUDENT TO IVANOV;

Без наличия других привилегий IVANOV может только выбирать (SELECT) значения, но не может выполнять любые действия, которые бы воздействовали на значения в таблице STUDENT, включая использование таблицы в качестве родительской таблицы внешнего ключа.

Для команды GRANT сначала проверяется допустимость этой команды для давшего ее пользователя. IVANOV дать этой команды не может.

Он также не может предоставить право SELECT другому пользователю, так как таблица принадлежит не ему. Команда:

GRANT INSERT ON EXAM\_MARKS TO IVANOV;

позволяет вводить в таблицу новые строки.

Можно передавать несколько привилегий нескольким пользователям:

GRANT SELECT, INSERT ON SUBJECT

TO IVANOV, PETROV;

Однако, в ANSI **невозможно предоставить привилегии для нескольких таблиц** одной командой GRANT.

Чтобы предоставить все привилегии в таблице используют команду ALL PRIVILEGES или ALL.

Например,

GRANT ALL PRIVILEGES ON STUDENT TO IVANOV;

Или:

GRANT ALL ON STUDENT TO IVANOV;

Передает пользователю Иванов все привилегии в таблице Студент.

Аргумент PUBLIC используется для передачи указанных в команде привилегий всем остальным пользователям. Чаще применяется для привилегии SELECT в базовых таблицах или представлениях, которые необходимо сделать доступными пользователям:

GRANT SELECT ON EXAM\_MARKS TO PUBLIC;

Позволяет любому пользователю получить информацию из таблицы EXAM\_MARKS.

Предоставление **всех** привилегий в таблице **всем** пользователям обычно является нежелательным. Так как это может вызвать определенные проблемы обеспечения безопасности и защиты данных. Поэтому для ограничения доступа к таблице лучше всего предоставить привилегии, отличные от SELECT, только индивидуальным пользователям.

Отмена привилегии осуществляется с помощью команды REVOKE, синтаксис аналогичный GRANT.

Другие типы привилегий

Теперь следует рассмотреть следующие вопросы:

-кто имеет право создавать таблицы;

-изменять, удалять, ограничивать таблицы;

-должны ли права создания базовых таблиц отличаться от прав создания представлений;

-должен ли существовать суперпользователь, т.е. пользователь, отвечающий за поддержку БД, имеющий наибольшие (полные привилегии), которые не предоставляются обычному пользователю.

Привилегии, которые не определяются в терминах специальных объектов данных, называются *привилегиями системы или правами БД.*

Эти привилегии включают в себя право создавать объекты данных, отличающиеся от базовых таблиц (обычно создаваемых несколькими пользователями) и представлений (создаваемых большинством пользователей).

Привилегии системы должны дополнять, а не заменять привилегии объекта, которые стандарт требует от создателей представлений.

В любой системе всегда имеются некоторые типы суперпользователей. То есть пользователей, которые имеют большинство или все привилегии и могут передать их (администратор БД).

Общий подход определяет три базовые привилегии системы:

CONNECT-подключить;

RESOURCE-ресурс;

DBA-администратор БД.

CONNECT-право зарегистрироваться и права создать представления и синонимы;

RESOURCE-создать базовые таблицы;

DBA-привилегия администратора, может иметь одного или более пользователей.

В большинстве реализаций SQL нового пользователя создает DBA, т.е. администратор БД, автоматически предоставляющий новому пользователю привилегию CONNECT.

В этом случае добавляется IDENTIFIED BY ‘PASSWORD’.

Например,

GRANT CONNECT TO PETROV IDENTIFIED BY ‘PASSWORD’;

Преимущества SQL-сервера

Особенности применения первичного ключа:

* Не обновлять значения в столбцах первичного ключа
* Не употреблять одни и те же значения в столбцах первичного ключа
* Не использовать в первичном ключе значения, которые могут измениться

Преимущества SQL перед другими СУБД:

* Не является оригинальным т.е. поддерживается во всех реляционных БД
* Простой в изучении
* Эффективный, позволяет выполнять сложные операции с БД

Причины использования SQL-сервера

* Высокая производительность, которую можно увеличивать
* Надежность
* Интеграция с продуктами Microsoft и с облачными сервисами типа Microsoft Azure
* Простота в установке, эксплуатации и что-то там еще, инструменты

Причины, по которым не следует использовать SQL-сервер:

* Если серверы работают от другой ОС
* Дорогой коммерческий продукт (ДЕНЬГИ0\_0!!)

СУБД делят на две категории:

* Файловые
* Клиент-серверные

Серверная часть это ПО которое отвечает за доступ ко всем данным и манипулированием ими. Все запросы на ввод, удаление, обновление данных пропускаются через ПО.

ПО SQL-сервер делится на две части:

* В качестве серверного ПО служит СУБД
* В качестве клиента служат инструментальные средства в составе SQL-сервера т.е. языки написания сценариев, ЯП, языки разработки веб-приложений

При изучении SQL-сервера лучше пользоваться утилитой, предназначенной для выполнения простых сценариев.

Утилита – это программа поводырь, помогает что-то делать. (Microsoft SQL-Server Management Studio)

Получить доступ к серверу можно следующими способами:

* Загрузить и установить SQL-сервер на локальный ПК
* Если использовать установленную администратором программу необходимо иметь учетную запись (регистрационное имя пользователя и пароль)
* SQL-сервер может быть размещена на популярных платформах для облачных вычислений Azure, Amazon, Clone.

Где бы ни был установлен сервер БД, потребуется его адрес (для локальной установки имя вашего ПК, адрес и учетные данные для доступа к нему).

Для выполнения команд транзакт-SQL необходимо использовать утилиту Management Studio. На сайте есть чтобы скачать.

Для того чтобы пользоваться базой используют ключевое слово USE\_имя базы.

Оператор USE не возвращает никаких результатов, а только сообщение о успешно выполненной команде.

Таким образом клиентские приложения получают перечень доступных БД.

Сведения о БД, таблицах, пользователях привилегиях, находятся внутри самих БД т.е. для хранения этой информации SQL-сервер пользуется собственными ресурсами. Это внутренние таблицы находятся в главной БД и недоступны для обычного пользователя. Для получения подобных сведений SQL-сервер имеется набор хранимых процедур, например, чтобы получить список доступных БД, в которых включены базы, предназначенные для внутреннего применения, используют команды:

* SP\_datebases;
* SP\_tables;
* И много чего еще.

Каждый оператор SQL состоит из одного или нескольких ключевых слов. Наиболее часто применяемых является оператор SELECT.

Извлеченные данные не фильтруются и не сортируются по умолчанию.

Регистр имеет значения что-то про C#, объекты с большой, а продолжать маленькими.

Метасимвол подстановки (\* - звездочка) следует избегать во всех случаях, когда не требуется действительно извлечь все столбцы, так как это снижает эффективность обработки данных и производительность всего приложения, однако использование метасимвола дает преимущество, когда имена столбцов неизвестны.

**ОСНОВЫ ПРОГРАММИРОВАНИЯ С ПОМОЩЬЮ ВСТРОЕННОГО ЯЗЫКА TRANSACT-SQL В MICROSOFT SQL SERVER**

**Цель работы**

* Изучить используемый в реляционных СУБД встроенный язык программирования Transact-SQL для написания программ в MS SQL Server.
* Изучить правила построения идентификаторов, правила объявления переменных и их типов.
* Изучить принципы работы с циклами и ветвлениями. Изучить работу с переменными типа Table.
* Изучить синтаксис и семантику функций и хранимых процедур Transact–SQL: способов их идентификации, методов задания и спецификации параметров и возвращаемых значений и вызовов функций и хранимых процедур.

Для освоения программирования используем пример базы данных**,** которая была создана в предыдущих работах. При выполнении примеров и заданий обращайте внимание на соответствие названий БД, таблиц и других объектов проекта.

**Идентификаторы** - это имена объектов, на которые можно ссылаться в программе, написанной на языке Transact SQL. Первый символ может состоять из букв английского алфавита или "**@**", "**#**". Остальные дополнительно из цифр и «$».

Имя идентификатора не должно совпадать с зарезервированным словом.

Для ограничителей идентификаторов при установленном параметре **SET QUOTED\_IDENTIFIER ON** можно использовать как квадратные скобки, так и одинарные кавычки, а строковые значения только в одинарных кавычках (режим по умолчанию).

Если использовать установленный параметр в режиме **SET QUOTED\_IDENTIFIER OFF**, то в качестве ограничителей идентификаторов можно использовать только квадратные скобки, а строковые значения указываются в одинарных или двойных кавычках.

**Переменные** используются для сохранения промежуточных данных в хранимых процедурах и функциях. Все переменные считаются локальными. Имя переменной должно начинаться с **@**.

**Объявление переменных**

Синтаксис в обозначениях MS SQL Server:

**DECLARE @имя\_переменной1 тип\_переменной, @имя\_переменнойN тип\_переменной**

Если тип переменной предполагает указание размера, то используется следующий синтаксис для объявления переменных:

**DECLARE @ имя\_переменной1 тип\_переменной (размер), .,**

@ имя\_переменнойN тип\_переменной(размер)

**Пример 1**: Введите в редактор SQL запросов следующее объявление переменных:

**DECLARE @a INT, @b numeric(10,2)**

**DECLARE @str CHAR(20)**

**Присвоение значений переменным и вывод значений на экран**

Присвоение с помощью **SET** - обычное присвоение, синтаксис:

**SET @имя\_переменной = значение**

Допишите в редакторе SQL объявление переменных с вводом их значений и выводом результата

**DECLARE @a INT, @b numeric (10,2)**

**SET @a = 20**

**SET @b = (@a+@a)/15**

**SELECT @b --вывод на экран результата**

Запустить эту конструкцию на выполнение.

Присвоение с помощью **SELECT** - помещение результата запроса в переменную. Если в результате выполнения запроса не будет возвращено ни одной строки, то значение переменной не меняется, т.е. остается старым.

Рассмотрим пример, в котором переменной присвоим результат выполнения агрегированной функции над таблицами базы данных.

**Пример 2:** Вывести количество строк в таблице **Студенты** и присвоить это значение переменной **@a.**

**DECLARE @a INT**

**SELECT @a = COUNT(\*) FROM Student**

**SELECT @a**

**Пример 3:**

**DECLARE @str CHAR(30)**

**SELECT @str = Sutname FROM Student**

**SELECT @str**

В данном примере в переменную поместится последнее значение из результата запроса.

**Сочетание ключевых слов SET и SELECT**

Измените код запроса из примера 2 на следующий:

**DECLARE @a INT**

**SET @a = (SELECT COUNT(\*) FROM Authors)**

**SELECT @a**

**Создание временной таблицы через переменную типа TABLE**

Объявляется через DECLARE с указанием в скобках столбцов таблицы, их типов, размеров, значений по умолчанию, а также индексов типа PRIMARY KEY или UNIQUE.

**Пример 4:** Создание временной таблицы с двумя полями, в которую будут добавлены 2 строки с данными:

**DECLARE @mytable TABLE (id INT, myname CHAR(20) DEFAULT 'Иванов Иван')**

**INSERT INTO @mytable(id) VALUES (1)**

**INSERT INTO @mytable(id, myname) VALUES (2,'Игорь Троцкий')**

**SELECT \* FROM @mytable**

Выполните и сохраните запрос.

**Пример 5:** Создание временной таблицы с двумя полями, в которую будут добавлены строки, как результат выполнения запроса выборки данных из таблицы **факультет.**

**DECLARE @mytable TABLE(id INT, myname CHAR(255) DEFAULT 'Введите название')**

**INSERT @mytable SELECT kod\_faculteta, name\_faculteta FROM facultet**

**SELECT \* FROM @mytable**

**Операторские скобки**

**BEGIN**

/\* в них нельзя помещать команды, изменяющие структуры объектов БД. Операторские скобки должны содержать хотя бы один оператор. Требуются для конструкций поливариантных ветвлений, условных и циклических конструкций

\*/

**END**

**Условная конструкция IF**

Синтаксис:

**IF условие**

Набор операторов**1 ELSE**

**Набор операторов2**

**Пример 6:** Использование операторских скобок и условных конструкций для поиска ответа – количество кафедр больше 10, да или нет

**DECLARE @a INT**

**DECLARE @str CHAR(30)**

**SET @a = (SELECT COUNT(\*) FROM kafedra)**

**IF @a >10 BEGIN**

**SET @str = 'Количество кафедр больше 10'**

**SELECT @str**

**END ELSE**

**BEGIN**

**SET @str = 'Количество кафедр = ' + str(@a)**

**SELECT @str**

**END**

**Цикл WHILE**

Синтаксис:

**WHILE Условие**

**Набор операторов**1 **BREAK**

**Набор опреторов**2 **CONTINUE**

Конструкции **BREAK** и **CONTINUE** являются необязательными.

Цикл можно принудительно остановить, если в его теле выполнить команду **BREAK**. Если же нужно начать цикл заново, не дожидаясь выполнения всех команд в теле, необходимо выполнить команду **CONTINUE**.

**Пример 7:** Объявление переменной **а**, проверка в цикле, чтобы значение **а** не превосходило 100. Переменная **а** увеличивает свое значение в случайном порядке. Также организуется проверка на условие.

DECLARE @a INT

SET @a = 1

WHILE @a <100

BEGIN

PRINT @a -- вывод на экран значения переменной

IF (@a>40) AND (@a<50)

BREAK --выход и выполнение 1-й команды за циклом

ELSE

SET @a = @a+rand()\*10

CONTINUE

END

PRINT @a

**Функции и хранимые процедуры** используются в SQL Server для реализации на языке Transact-SQL сложных часто используемых алгоритмов обработки данных или различных административных действий создания учетных записей, получения информации об объектах базы данных, управления свойствами сервера и баз данных, управления подсистемой репликации и автоматизации и т.д.

**Они хранятся в виде исходного текста и являются программными модулями**, существующими независимо от таблиц или каких-либо других объектов баз данных.

Хранение функций и хранимых процедур в виде исходных модулей языка Transact – SQL на сервере и в соответствующих базах данных **позволяет уменьшить размер запроса**, посылаемого по сети от клиента на сервер, а, следовательно, и **нагрузку на сеть**, что повышает общую производительность системы. Это также позволяет **упростить сопровождение программных комплексов** и внесение изменений в исходный текст модулей, причем большинство изменений не отразится на работоспособности клиентских приложений.

Значительная часть функций и хранимых процедур поставляются в составе SQL Server. Они называются **системными**, или **встроенными (built - in)**.

Кроме того, пользователю предоставляется возможность разрабатывать и включать в свою базу данных собственные, или **пользовательские (user-defined) функции и хранимые процедуры**, реализующие специальные алгоритмы обработки данных.

Таким образом, пользовательские функции и хранимые процедуры становятся объектами той базы данных, в которой они создавались. Поэтому при их создании, если необходимо, требуемую базу данных следует сделать текущей с помощью команды **USE** имя базы данных. Системные же функции хранятся на экземпляре сервера, а системные хранимые процедуры – в базе MASTER этого же экземпляра сервера.

**Функции и хранимые процедуры могут быть вызваны клиентскими программами**, другими функциями или хранимыми процедурами, а также **триггерам**и.

В любом случае необходимо указать имя функции или хранимой процедуры и список аргументов, которые сопоставляются параметрам соответствующей функции или хранимой процедуры, при этом типы аргументов и параметров должны совпадать или допускать автоматические преобразования типов. Если для некоторого параметра задано значение по умолчанию и это значение подходит для данного вызова, то соответствующий аргумент может быть опущен.

Поскольку функция возвращает значение, она используется в качестве операнда некоторого выражения в виде вызова функций, состоящего из имени этой функции и списка аргументов, заключенного в круглые скобки, при этом в качестве аргументов могут быть любые выражения языка Transact – SQL, дающие в результате значения требуемых типов.

Аргументы в вызове функции отделяются запятыми.

Если список аргументов пуст, то круглые скобки после имени функции, как правило, задаются.

Исключения составляют некоторые системные функции, для которых круглые скобки не задаются, когда нет аргументов.

Хранимые процедуры могут вызываться только командой **EXECUTE**, или сокращенно **EXEC**. За этой командой должны быть указаны имя процедуры и через пробел список аргументов, если вызывается процедура с параметрами. Аргументы разделяются запятой. Если для параметра задано значение по умолчанию, то аргумент либо совсем не задается (в конце списка), либо используется слово **DEFAULT**(в середине списка).

Процедура может возвращать результаты только через параметры с ключевым словом **OUTPUT**, при этом и аргумент должен быть задан с таким же ключевым словом.

Создание, изменение и удаление функций и хранимых процедур производится соответственно командами:

**CREATE FUNCTION,**

**CREATE PROCEDURE,**

**ALTER FUNCTION,**

**ALTER PROCEDURE,**

**DROP FUNCTION,**

**DROP PROCEDURE**.

**При создании функции** указывается тип возвращаемого значения и в теле функции обязательно задается команда **RETURN**, за которой следует выражения для вычисления возвращаемого значения.

**В теле процедуры** использование команды **RETURN** (конечно, без последующего выражения) вовсе не обязательно. Когда этой команды нет, выход из процедуры будет происходить после исполнения последней команды процедуры.

Тело, как функции, так и хранимой процедуры начинается ключевым словом **AS.**

Поскольку каждая из них хранится как отдельный объект, то для указания конца тела не требуется записывать какое-либо специальное ключевое слово или знак.

За командами создания функции или хранимой процедуры перечисляются имена параметров, начинающиеся с символа **@**, и их типы, а также важно значение по умолчанию.

Для функции этот список заключается в круглые скобки, после которых записывается ключевое слово **RETURNS** (возвращает) и тип возвращаемого значения.

Для хранимой процедуры круглые скобки не используются, и задавать тип возвращаемого значение не требуется.

Для тела функции часто используют ключевое слово **begin** после ключевого слова **as** и ключевое слово **end** в конце тела.

Дополнительные опции функции или хранимой процедуры задаются ключевым словом **with** до начала тела.

В SQL Server 2003 можно создавать **функции трех классов**:

**- Scalar** – возвращают обычное скалярное значение;

**- Inline** – возвращают таблицу динамической структуры, создаваемую единственной командой тела функции SELECT;

**- Multi – statement** – возвращает обычную таблицу заданной структуры, при этом количество команд в теле функции не ограничивается.

**СОЗДАНИЕ ФУНКЦИЙ**

Команда:

**CREATE FUNCTION (Transact-SQL)**

создает **определяемую пользователем функцию** в SQL Server. Определяемая пользователем функция представляет собой подпрограмму Transact-SQL или среды CLR, которая принимает параметры, выполняет действия, такие как сложные вычисления, а затем возвращает результат этих действий в виде значения. Возвращаемое значение может быть скалярным значением или таблицей. При помощи этой инструкции можно создать подпрограмму, которую можно повторно использовать следующими способами:

 В инструкциях Transact-SQL, например **SELECT**.

 В приложениях, вызывающих функцию.

 В определении другой пользовательской функции.

 Для параметризации представления или улучшения функциональности индексированного представления.

 Для определения столбца таблицы.

 Для определения ограничения **CHECK** на столбец.

 Для замены хранимой процедуры.

Синтаксис создания скалярной функции.

Аргументы

**schema\_name** - Имя схемы, к которой принадлежит определяемая пользователем функция.

**function\_name** - Имя определяемой пользователем функции. Имена функций должны удовлетворять правилам построения идентификаторов и должны быть уникальными в пределах базы данных и схемы.

**@parameter\_name** - Аргумент пользовательской функции. Может быть объявлен один или несколько аргументов.

Для функций допускается не более 2 100 параметров. При выполнении функции значение каждого из объявленных параметров должно быть указано пользователем, если для них не определены значения по умолчанию.

[ **type\_schema\_name**. ] **parameter\_data\_type -** Тип данных параметра

**[ =default ] -** Значение по умолчанию для аргумента. Если определено значение default, то функция выполняется даже в том случае, если для данного аргумента значение не указано.

**READONLY -** Указывает, что параметр не может быть обновлен или изменен при определении функции. Если тип параметра является определяемым пользователем табличным типом, то должно быть указано ключевое слово READONLY.

**return\_data\_type** - Возвращаемое значение скалярной функции, определяемой пользователем.

**function\_body** - Указывает серию инструкций Transact-SQL

**scalar\_expression** - Указывает скалярное значение, возвращаемое скалярной функцией.

**Предложение EXECUTE AS** - Указывает контекст безопасности, в котором выполняется определяемая пользователем функция. Иными словами, есть возможность управлять тем, какую учетную запись пользователя SQL Server использует при определении разрешений на объекты базы данных, на которые ссылается функция.

**Синтаксис функции, которая возвращает табличное значение**:

**Аргументы:**

**TABLE -** Указывает, что возвращаемым значением функции с табличным значением, является таблица. Функциям с табличным значением, могут передаваться только константы и @local\_variables.

Во встроенных функциях с табличным значением возвращаемое значение TABLE определяется при использовании единственной инструкции **SELECT**. Встроенные функции не имеют соответствующих возвращаемых переменных.

**select\_stmt -** Одиночная инструкция **SELECT**, определяющая возвращаемое значение встроенной функции с табличным значением.

В функциях допустимы следующие инструкции.

 Инструкции присваивания.

 Инструкции управления потоком, за исключением инструкций **TRY...CATCH**.

 Инструкции **DECLARE**, объявляющие локальные переменные и локальные курсоры.

 Инструкции **SELECT**, которые содержат списки выбора с выражениями, присваивающими значения локальным переменным.

 Операции над локальными курсорами, которые объявляются, открываются, закрываются и освобождаются в теле функции. Допустимы только те инструкции **FETCH,** которые предложением **INTO** присваивают значения локальным переменным. Инструкции **FETCH**, возвращающие данные клиенту, недопустимы.

 Инструкции **INSERT, UPDATE** и **DELETE**, которые изменяют локальные табличные переменные.

 Инструкции **EXECUTE**, вызывающие расширенные хранимые процедуры.

**Ограничения**

Определяемые пользователем функция не может выполнять действия, изменяющие состояние базы данных.

Определяемые пользователем функции не могут содержать предложение **OUTPUT INTO**, целью которого является таблица.

Определяемые пользователем функции могут быть **вложенными**, то есть из одной функции может быть вызвана другая. Вложенность определяемых пользователем функций не может превышать 32 уровней.

**Пример 1.** Применение скалярной определяемой пользователем функции, вычисляющей месяц.

В следующем примере показано создание определяемой пользовательской функции **ISOweek,** которая получает в качестве аргумента дату и вычисляет номер месяца, а затем по номеру определяет название месяца. Для правильной работы этой функции перед ее вызовом должна быть выполнена инструкция **SET DATEFIRST 1.**

Следующий пример также показывает использование предложения **EXECUTE AS** для указания контекста безопасности, в котором может быть выполнена хранимая процедура. В этом примере параметр **CALLER** указывает, что процедура будет выполнена в контексте пользователя, который ее вызывает.

В начале, прежде чем создавать функцию, будет выполнена проверка на наличие в базе данных функции под таким же именем. Если функция с таким именем уже была создана ранее, то она будет удалена с помощью команды **DROP FUNCTION** и создана заново **CREATE FUNCTION**.

Откройте sql-редактор. Создайте новый запрос.

**USE University;**

**GO**

**IF OBJECT\_ID (N'dbo.ISOweek', N'FN') IS NOT NULL**

**DROP FUNCTION dbo.ISOweek;**

**GO**

**CREATE FUNCTION dbo.ISOweek (@DATE date)**

**RETURNS CHAR(15)**

**WITH EXECUTE AS CALLER**

**AS**

**BEGIN**

**DECLARE @man int;**

**DECLARE @ISOweek char(15);**

**SET @man= MONTH(@DATE)**

**IF (@man=1) SET @ISOweek='Январь';**

**IF (@man=2) SET @ISOweek='Февраль';**

**IF (@man=3) SET @ISOweek='Март';**

**IF (@man=4) SET @ISOweek='Апрель';**

**IF (@man=5) SET @ISOweek='Май';**

**IF (@man=6) SET @ISOweek='Июнь';**

**IF (@man=7) SET @ISOweek='Июль';**

**IF (@man=8) SET @ISOweek='Август';**

**IF (@man=9) SET @ISOweek='Сентябрь';**

**IF (@man=10) SET @ISOweek='Октябрь';**

**IF (@man=11) SET @ISOweek='Ноябрь';**

**IF (@man=12) SET @ISOweek='Декабрь';**

**RETURN(@ISOweek);**

**END;**

Для того чтобы увидеть результат, выведем на экран вычисление значение функции от произвольной даты, например, '12.04.2004'. для этого ниже в sql-редакторе пишем:

**GO**

**SET DATEFIRST 1;**

**SELECT dbo.ISOweek('12.04.2004') AS 'Месяц';**

Ниже приводится результирующий набор.

Просмотреть все функции, которые пользователь может создавать, откройте в окне обозревателя объектов в базе данных **University** группу **Программирование/Функции/Скалярные функции**.

**Пример 2. Создание пользовательской функции с табличным значением**.

Например, создадим пользовательскую функцию, которая будет разрешена к использованию всеми пользователями с правами роли «**dekan**». Функция будет возвращать результат в виде **таблицы** – вывод на всех кафедрах суммирующей зарплаты по каждой должности всех преподавателей. При этом функция имеет один параметр **@storeid**, с помощью которого введем ограничение на вычисление, а именно зарплата должна быть больше 100.

Создайте новый запрос:

**USE University;**

**GO**

**IF OBJECT\_ID (N'ufn\_SalesByStore', N'IF') IS NOT NULL**

**DROP FUNCTION dekan.ufn\_SalesByStore;**

**GO**

**CREATE FUNCTION dekan.ufn\_SalesByStore(@storeid int)**

**RETURNS TABLE**

**AS**

**RETURN**

**(**

**SELECT d.Name\_kafedru AS "Кафедра",**

**t.Dolgnost AS "Должность",** 164

**SUM(t.Salary + t.RISE) AS "Сумма зарплаты"**

**FROM KAFEDRA d, TEACHER t**

**WHERE d.KOD\_kafedru =t.KOD\_kafedru**

**and t.salary>@storeid**

**GROUP BY d.Name\_kafedru, t.Dolgnost**

**);**

Для вызова этой функции нужно выполнить следующий запрос:

**GO**

**SELECT \* from dekan.ufn\_SalesByStore(100);**

Таблица Orders представляет заказы, а столбец CreatedAt - дату заказов. *Узнаем, были ли заказы за последние 10 дней:*

DECLARE @lastDate DATE

SELECT @lastDate = MAX(CreatedAt) FROM Orders

IF DATEDIFF(day, @lastDate, GETDATE()) > 10

PRINT 'За последние десять дней не было заказов'

Добавим выражение ELSE:

DECLARE @lastDate DATE

SELECT @lastDate = MAX(CreatedAt) FROM Orders

IF DATEDIFF(day, @lastDate, GETDATE()) > 10

PRINT 'За последние десять дней не было заказов'

ELSE

PRINT 'За последние десять дней были заказы'

Если после IF или ELSE идут две и более инструкций, то они заключаются в блок BEGIN...END:

DECLARE @lastDate DATE, @count INT, @sum MONEY

SELECT @lastDate = MAX(CreatedAt),

@count = SUM(ProductCount) ,

@sum = SUM(ProductCount \* Price)

FROM Orders

IF @count > 0

BEGIN

PRINT 'Дата последнего заказа: ' + CONVERT(NVARCHAR, @lastDate)

PRINT 'Продано ' + CONVERT(NVARCHAR, @count) + ' единиц(ы)'

PRINT 'На общую сумму ' + CONVERT(NVARCHAR, @sum)

END;

ELSE

PRINT 'Заказы в базе данных отсутствуют'

Для выполнения повторяющихся операций в T-SQL применяются циклы. В частности, в T-SQL есть цикл WHILE. Этот цикл выполняет определенные действия, пока некоторое условие истинно.

WHILE условие

{инструкция|BEGIN...END}

Если в блоке WHILE необходимо разместить несколько инструкций, то все они помещаются в блок BEGIN...END.

Например, вычислим факториал числа:

DECLARE @number INT, @factorial INT

SET @factorial = 1;

SET @number = 5;

WHILE @number > 0

BEGIN

SET @factorial = @factorial \* @number

SET @number = @number - 1

END;

PRINT @factorial

То есть в данном случае пока переменная @number не будет равна 0, будет продолжаться цикл WHILE. Так как @number равна 5, то цикл сделает пять проходов. Каждый проход цикла называется итерацией. В каждой итерации будет переустанавливаться значение переменных @factorial и @number.

Другой пример - рассчитаем баланс счета через несколько лет с учетом процентной ставки:

USE productsdb;

CREATE TABLE #Accounts ( CreatedAt DATE, Balance MONEY)

DECLARE @rate FLOAT, @period INT, @sum MONEY, @date DATE

SET @date = GETDATE()

SET @rate = 0.065;

SET @period = 5;

SET @sum = 10000;

WHILE @period > 0

BEGIN

INSERT INTO #Accounts VALUES(@date, @sum)

SET @period = @period - 1

SET @date = DATEADD(year, 1, @date)

SET @sum = @sum + @sum \* @rate

END;

SELECT \* FROM #Accounts

Здесь создается временная таблица #Accounts, в которую добавляется в цикле пять строк с данными.

Операторы BREAK и CONTINUE

Оператор BREAK позволяет завершить цикл, а оператор CONTINUE - перейти к новой итерации.

DECLARE @number INT

SET @number = 1

WHILE @number < 10

BEGIN

PRINT CONVERT(NVARCHAR, @number)

SET @number = @number + 1

IF @number = 7

BREAK;

IF @number = 4

CONTINUE;

PRINT 'Конец итерации'

END;

Когда переменная @number станет равна 4, то с помощью оператора CONTINUE произойдет переход к новой итерации, поэтому последующая строка PRINT 'Конец итерации' не будет выполняться, хотя цикл продолжится.

Когда переменная @number станет равна 7, то оператор BREAK произведет выход из цикла, и он завершится.

**Как работает запрос SQL со связанным подзапросом**

SELECT \*

FROM EXAM\_MARKS

WHERE STUDENT\_ID =

(SELECT STUDENT\_ID

FROM STUDENT

WHERE SURNAME = 'Петров');

Как работает запрос SQL со связанным подзапросом?

• Выбирается строка из таблицы, имя которой указано во внешнем запросе.

• Выполняется подзапрос и полученное значение применяется для анализа этой строки в условии предложения WHERE внешнего запроса.

• По результату оценки этого условия принимается решение о включении или не включении строки в состав выходных данных

• Процедура повторяется для следующей строки таблицы

внешнего запроса.

Следует обратить внимание, что приведенный выше запрос корректен только в том случае, если в результате выполнения указанного в скобках подзапроса возвращается единственное значение. Если в результате выполнения подзапроса будет возвращено несколько значений, то этот подзапрос будет

ошибочным. В данном примере это произойдет, если в таблице STUDENT будет несколько записей со значениями поля

SURNAME = 'Петров'.

В некоторых случаях для гарантии получения единственного значения в результате выполнения подзапроса используется

DISTINCT.

Одним из видов функций, которые автоматически

всегда выдают в результате единственное значение для любого

количества строк, являются агрегирующие функции.

Оператор IN также широко применяется в подзапросах. Он

задает список значений, с которыми сравниваются другие значения для определения истинности задаваемого этим оператором предиката.

Данные обо всех оценках (таблица EXAM\_MARKS) студентов

из Воронежа можно выбрать с помощью следующего запроса:

SELECT \*

FROM EXAM\_MARKS

WHERE STUDENT\_ID IN

(SELECT STUDENT\_ID

FROM STUDENT

WHERE CITY = 'Воронеж');

Подзапросы можно применять внутри предложения HAVING.

Пусть требуется определить количество предметов обучения

с оценкой, превышающей среднее значение оценки студента

с идентификатором 301:

SELECT COUNT(DISTINCT SUBJ\_ID), MARK

FROM EXAM\_MARKS

GROUP BY MARK

HAVING MARK >

(SELECT AVG(MARK)

FROM EXAM\_MARKS

WHERE STUDENT\_ID = 301);

При использовании подзапросов во внутреннем запросе

можно ссылаться на таблицу, имя которой указано в предложении FROM внешнего запроса. В этом случае такой связанный

подзапрос выполняется по одному разу для каждой строки таблицы основного запроса.

Пример: выбрать сведения обо всех предметах обучения, по

которым проводился экзамен 20 января 1999 г.

SELECT \*

FROM SUBJECT SU

WHERE '20/01/1999' IN

(SELECT EXAM\_DATE

PROM EXAM\_MARKS EX

WHERE SU.SUBJ\_ID = EX.SUBJ\_ID);

В некоторых СУБД для выполнения этого запроса может

потребоваться преобразование значения даты в символьный

тип. В приведенном запросе su и ЕХ являются псевдонимами

(алиасами), то есть специально вводимыми именами, которые

могут быть использованы в данном запросе вместо настоящих

имен. В приведенном примере они используются вместо имен

таблиц SUBJECT и EXAM\_MARKS.

Эту же задачу можно решить с помощью операции соединения таблиц:

SELECT DISTINCT SU.SUBJ\_ID, SUBJ\_NAME, HOUR, SEMESTER

FROM SUBJECT FIRST, EXAM\_MARKS SECOND

WHERE FIRST.SUBJ\_ID = SECOND.SUBJ\_ID

AND SECOND.EXAM\_DATE = '20/01/1999';

В этом выражении алиасами таблиц являются имена FIRST

И SECOND.

Можно использовать подзапросы, связывающие таблицу со

своей собственной копией.

Например, надо найти идентификаторы, фамилии и стипендии студентов, получающих стипендию выше средней на курсе, на котором они учатся.

SELECT DISTINCT STUDENT\_ID, SURNAME, STIPEND

FROM STUDENT El

WHERE STIPEND >

(SELECT AVG(STIPEND)

FROM STUDENT E2

WHERE El.KURS = E2.KURS);

Тот же результат можно получить с помощью следующего

запроса:

SELECT DISTINCT STUDENT\_ID, SURNAME, STIPEND

FROM STUDENT El,

(SELECT KURS, AVG (STIPEND) AS AVG\_STIPEND

FROM STUDENT E2

GROUP BY E2.KURS) E3

WHERE El.STIPEND > AVG\_STIPEND AND El.KURS=E3.KURS;

Обратите внимание — второй запрос будет выполнен гораздо

быстрее. Дело в том, что в первом варианте запроса агрегирующая функция AVG выполняется над таблицей, указанной в подзапросе, для каждой строки внешнего запроса.

В другом варианте вторая таблица (алиас Е2) обрабатывается агрегирующей

функцией один раз, в результате чего формируется вспомогательная таблица (в запросе она имеет алиас ЕЗ), со строками которой

затем соединяются строки первой таблицы (алиас Е1). Следует

иметь в виду, что реальное время выполнения запроса в большой

степени зависит от оптимизатора запросов конкретной СУБД.

3. Связанные подзапросы в HAVING

Предложение GROUP BY позволяет группировать выводимые SELECT-запросом записи по значению некоторого поля. Использование предложения HAVING позволяет при выводе осуществлять фильтрацию таких групп.

Предикат предложения HAVING оценивается не для каждой

строки результата, а для каждой группы выходных записей,

сформированной предложением GROUP BY внешнего запроса.

Пусть, например, необходимо по данным из таблицы

EXAM\_MARKS определить сумму полученных студентами оценок

(значений поля MARK), сгруппировав значения оценок по датам

экзаменов и исключив те дни, когда число студентов, сдававших в течение дня экзамены, было меньше 10.

SELECT EXAM\_DATE, SUM(MARK)

FROM EXAM\_MARKS A

GROUP BY EXAM\_DATE

HAVING 10 <

(SELECT COUNT(MARK)

FROM EXAM\_MARKS В

WHERE A.EXAM\_DATE = В.EXAM\_DATE);

Подзапрос вычисляет количество строк с одной и той же

датой, совпадающей с датой, для которой сформирована очередная группа основного запроса.

**Что такое MERGE в T-SQL?**

**MERGE** – операция в языке T-SQL, при которой происходит обновление, вставка или удаление данных в таблице на основе результатов соединения с данными другой таблицы или SQL запроса. Другими словами, с помощью MERGE можно осуществить слияние двух таблиц, т.е. синхронизировать их.

В операции MERGE происходит объединение по ключевому полю или полям основной таблицы **(в которой и будут происходить все изменения**) с соответствующими полями другой таблицы или результата запроса. В итоге если условие, по которому происходит объединение, истина

(**WHEN MATCHED**),

то мы можем выполнить операции обновления или удаления, если условие не истина, т.е. отсутствуют данные

(**WHEN NOT MATCHED**),

то мы можем выполнить операцию вставки

(**INSERT добавление данных**),

также если в основной таблице присутствуют данные, которое отсутствуют в таблице (*или результате запрос*а) источника

(**WHEN NOT MATCHED BY SOURCE**), то мы можем выполнить обновление или удаление таких данных.

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

**Упрощённый синтаксис MERGE**

MERGE <Основная таблица>

USING <Таблица или запрос источника>

ON <Условия объединения>

[ WHEN MATCHED [ AND <Доп. условие> ]

THEN <UPDATE или DELETE>

[ WHEN NOT MATCHED [ AND Доп. условие> ]

THEN <INSERT> ]

[ WHEN NOT MATCHED BY SOURCE [ AND <Доп. условие> ]

THEN <UPDATE или DELETE> ] [ ...n ]

[ OUTPUT ]

;

**Важные моменты при использовании MERGE:**

* В конце инструкции MERGE обязательно должна идти точка с запятой (;) иначе возникнет ошибка;
* Должно быть, по крайней мере, одно условие MATCHED;
* Операцию MERGE можно использовать совместно с [CTE (обобщенным табличным выражением)](https://info-comp.ru/obucheniest/495-the-with-in-t-sql-or-common-table-expression.html);
* В инструкции MERGE можно использовать ключевое слово OUTPUT, для того чтобы посмотреть какие изменения были внесены. Для идентификации операции здесь в OUTPUT можно использовать переменную $action;
* На все операции к основной таблице, которые предусмотрены в MERGE (удаления, вставки или обновления), действуют все ограничения, определенные для этой таблицы;
* Функция @@ROWCOUNT, если ее использовать после инструкции MERGE, будет возвращать общее количество вставленных, обновленных и удаленных строк;
* Для того чтобы использовать MERGE необходимо разрешение на INSERT, UPDATE или DELETE в основной таблице, и разрешение SELECT для таблицы источника;
* При использовании MERGE необходимо учитывать, что все триггеры AFTER на INSERT, UPDATE или DELETE, определенные для целевой таблицы, будут запускаться.

А теперь переходим к практике. И для начала давайте определимся с исходными данными.

## Исходные данные для примеров операции MERGE

У меня в качестве SQL сервера будет выступать [Microsoft SQL Server 2016 Express](https://info-comp.ru/sisadminst/535-install-microsoft-sql-server-2016-express.html). На нем есть тестовая база данных, в которой я создаю тестовые таблицы, например, с товарами: **TestTable** – это у нас будет целевая таблица, т.е. та над которой мы будем производить все изменения, и **TestTableDop** – это таблица источник, т.е. данные в соответствии с чем, мы будем производить изменения.

***Запрос для создания таблиц.***

--Целевая таблица

CREATE TABLE dbo.TestTable(

ProductId INT NOT NULL,

ProductName VARCHAR(50) NULL,

Summa MONEY NULL,

CONSTRAINT PK\_TestTable PRIMARY KEY CLUSTERED (ProductId ASC)

)

--Таблица источник

CREATE TABLE dbo.TestTableDop(

ProductId INT NOT NULL,

ProductName VARCHAR(50) NULL,

Summa MONEY NULL,

CONSTRAINT PK\_TestTableDop PRIMARY KEY CLUSTERED (ProductId ASC)

)

***Далее я их наполняю тестовыми данными.***

--Добавляем данные в основную таблицу

INSERT INTO dbo.TestTable

(ProductId,ProductName,Summa)

VALUES

(1, 'Компьютер', 0)

GO

INSERT INTO dbo.TestTable

(ProductId,ProductName,Summa)

VALUES

(2, 'Принтер', 0)

GO

INSERT INTO dbo.TestTable

(ProductId,ProductName,Summa)

VALUES

(3, 'Монитор', 0)

GO

--Добавляем данные в таблицу источника

INSERT INTO dbo.TestTableDop

(ProductId,ProductName,Summa)

VALUES

(1, 'Компьютер', 500)

GO

INSERT INTO dbo.TestTableDop

(ProductId,ProductName,Summa)

VALUES

(2, 'Принтер', 300)

GO

INSERT INTO dbo.TestTableDop

(ProductId,ProductName,Summa)

VALUES

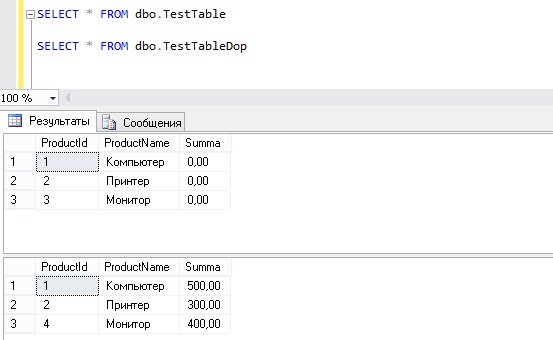
(4, 'Монитор', 400)

GO

***Посмотрим на эти данные.***

SELECT \* FROM dbo.TestTable

SELECT \* FROM dbo.TestTableDop



Видно, что в целевой таблице значение поля Summa = 0, а также есть несоответствие некоторых идентификаторов, т.е. у нас есть товары, которые есть в одной таблице, при этом они отсутствуют в другой.

## Пример 1 – обновление и добавление данных с помощью MERGE

Это, наверное, классический вариант использования MERGE, когда мы по условию объединения обновляем данные, а если таких данных нет, то добавляем их. Для наглядности в конце инструкции MERGE я укажу ключевое слово OUTPUT, для того чтобы посмотреть какие именно изменения мы произвели, а также сделаю выборку итоговых данных.

MERGE dbo.TestTable AS T\_Base --Целевая таблица

USING dbo.TestTableDop AS T\_Source --Таблица источник

ON (T\_Base.ProductId = T\_Source.ProductId) --Условие объединения

WHEN MATCHED THEN --Если истина (UPDATE)

UPDATE SET ProductName = T\_Source.ProductName, Summa = T\_Source.Summa

WHEN NOT MATCHED THEN --Если НЕ истина (INSERT)

INSERT (ProductId, ProductName, Summa)

VALUES (T\_Source.ProductId, T\_Source.ProductName, T\_Source.Summa)

--Посмотрим, что мы сделали

OUTPUT $action AS [Операция], Inserted.ProductId,

Inserted.ProductName AS ProductNameNEW,

Inserted.Summa AS SummaNEW,

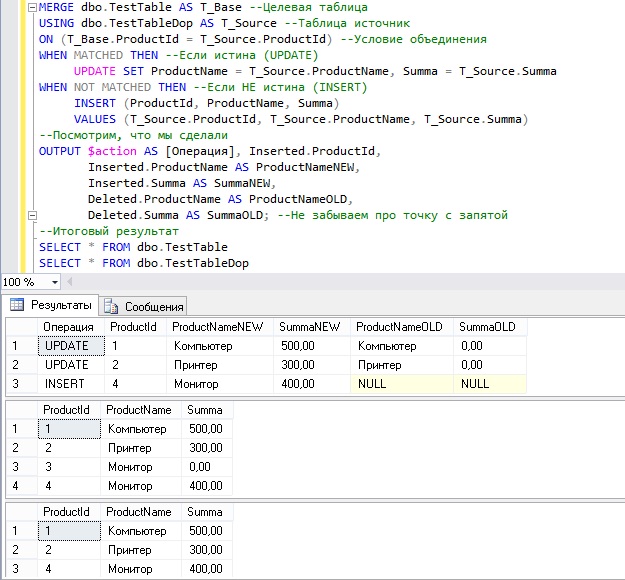
Deleted.ProductName AS ProductNameOLD,

Deleted.Summa AS SummaOLD; --Не забываем про точку с запятой

--Итоговый результат

SELECT \* FROM dbo.TestTable

SELECT \* FROM dbo.TestTableDop



Мы видим, что у нас было две операции UPDATE и одна INSERT. Так оно и есть, две строки из таблицы TestTable соответствуют двум строкам в таблице TestTableDop, т.е. у них один и тот же ProductId, у данных строк в таблице TestTable мы обновили поля ProductName и Summa. При этом в таблице TestTableDop есть строка, которая отсутствует в TestTable, поэтому мы ее и добавили через INSERT.

## Пример 2 – синхронизация таблиц с помощью MERGE

Теперь, допустим, нам нужно синхронизировать таблицу TestTable с таблицей TestTableDop, для этого мы добавим еще одно условие WHEN NOT MATCHED BY SOURCE, суть его в том, что мы удалим строки, которые есть в TestTable, но нет в TestTableDOP. Но для начала, для того чтобы у нас все три условия отработали (в частности WHEN NOT MATCHED) давайте в таблице TestTable удалим строку, которую мы добавили в предыдущем примере. Также здесь я в качестве источника укажу запрос, чтобы Вы видели, как можно использовать запросы в качестве источника.

--Удаление строки с ProductId = 4

--для того чтобы отработало условие WHEN NOT MATCHED

DELETE dbo.TestTable WHERE ProductId = 4

--Запрос MERGE для синхронизации таблиц

MERGE dbo.TestTable AS T\_Base --Целевая таблица

--Запрос в качестве источника

USING (SELECT ProductId, ProductName, Summa

FROM dbo.TestTableDop) AS T\_Source (ProductId, ProductName, Summa)

ON (T\_Base.ProductId = T\_Source.ProductId) --Условие объединения

WHEN MATCHED THEN --Если истина (UPDATE)

UPDATE SET ProductName = T\_Source.ProductName, Summa = T\_Source.Summa

WHEN NOT MATCHED THEN --Если НЕ истина (INSERT)

INSERT (ProductId, ProductName, Summa)

VALUES (T\_Source.ProductId, T\_Source.ProductName, T\_Source.Summa)

--Удаляем строки, если их нет в TestTableDOP

WHEN NOT MATCHED BY SOURCE THEN

DELETE

--Посмотрим, что мы сделали

OUTPUT $action AS [Операция], Inserted.ProductId, Inserted.ProductName AS ProductNameNEW,

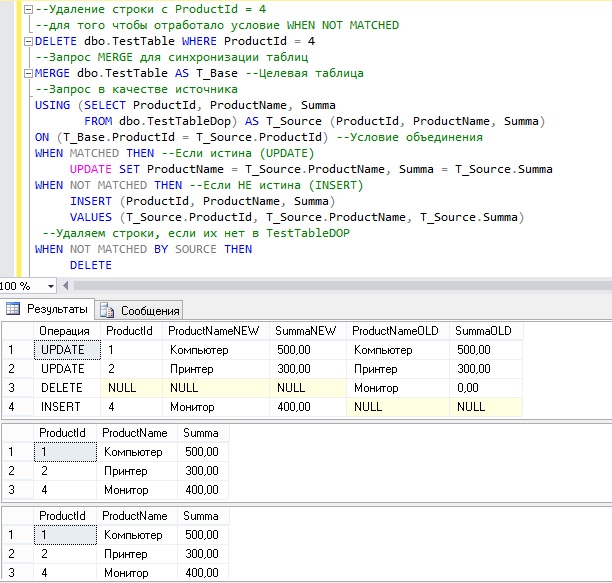
Inserted.Summa AS SummaNEW,Deleted.ProductName AS ProductNameOLD,

Deleted.Summa AS SummaOLD; --Не забываем про точку с запятой

--Итоговый результат

SELECT \* FROM dbo.TestTable

SELECT \* FROM dbo.TestTableDop



В итоге мы видим, что у нас таблицы содержат одинаковые данные. Для этого мы выполнили две операции UPDATE, одну INSERT и одну DELETE. При этом мы использовали всего одну инструкцию MERGE.

## Пример 3 – операция MERGE с дополнительным условием

Сейчас давайте выполним запрос похожий на запрос, который мы использовали в примере 1, только добавим дополнительное условие на обновление данных, например, мы будем обновлять TestTable только в том случае, если поле Summa, в TestTableDop, содержит какие-нибудь данные (например, мы не хотим использовать некорректные значения для обновления). Для того чтобы было видно, как отработало это условие, давайте предварительно очистим у одной строки в таблице TestTableDop поле Summa (поставим NULL).

--Очищаем поле сумма у одной строки в TestTableDop

UPDATE dbo. TestTableDop SET Summa = NULL

WHERE ProductId = 2

--Запрос MERGE

MERGE dbo.TestTable AS T\_Base --Целевая таблица

USING dbo.TestTableDop AS T\_Source --Таблица источник

ON (T\_Base.ProductId = T\_Source.ProductId) --Условие объединения

--Если истина + доп. условие отработало (UPDATE)

WHEN MATCHED AND T\_Source.Summa IS NOT NULL THEN

UPDATE SET ProductName = T\_Source.ProductName, Summa = T\_Source.Summa

WHEN NOT MATCHED THEN --Если НЕ истина (INSERT)

INSERT (ProductId, ProductName, Summa)

VALUES (T\_Source.ProductId, T\_Source.ProductName, T\_Source.Summa)

--Посмотрим, что мы сделали

OUTPUT $action AS [Операция], Inserted.ProductId,

Inserted.ProductName AS ProductNameNEW,

Inserted.Summa AS SummaNEW,

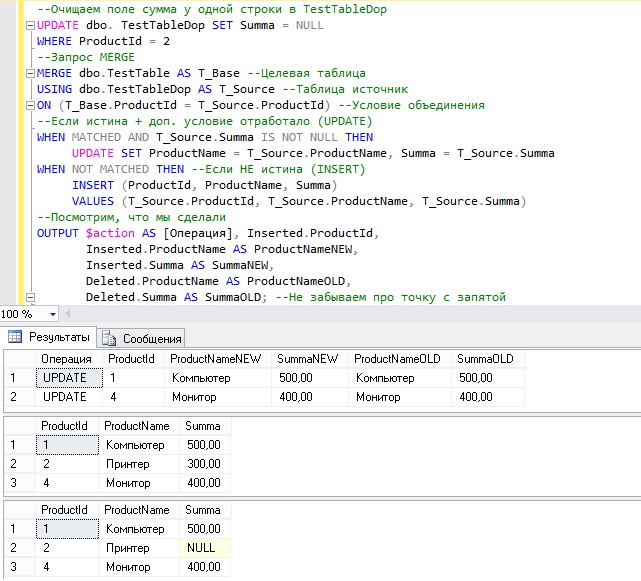
Deleted.ProductName AS ProductNameOLD,

Deleted.Summa AS SummaOLD; --Не забываем про точку с запятой

--Итоговый результат

SELECT \* FROM dbo.TestTable

SELECT \* FROM dbo.TestTableDop



В итоге у меня обновилось всего две строки, притом, что все три строки успешно выполнили условие объединения, но одна строка не обновилась, так как сработало дополнительное условие Summa IS NOT NULL, потому что поле Summa у строки с ProductId = 2, в таблице TestTableDop, не содержит никаких данных, т.е. NULL.

<https://info-comp.ru/obucheniest/561-merge-in-t-sql.html>

**Лекция 21**

**1.Внешнее объединение**

Часто бывает полезна операция объединения двух запросов,

в которой второй запрос выбирает строки, исключенные первым. Такая операция называется внешним объединением.

Рассмотрим пример.

Пусть в таблице STUDENT имеются записи о студентах, в которых не указан идентификатор университета.

Требуется составить список студентов с указанием наименований университетов для тех студентов, у которых эти данные есть, не исключая тех, у которых UNIV\_ID не указан.

Формируем объединение из двух запросов, один из которых выполняет выборку студентов с названиями UNIV\_ID , а второй с NULL-значениями в поле UNIV\_ID.

SELECT SURNAME, NAME, UNIV\_NAME

FROM STUDENT, UNIVERSITY

WHERE STUDENT.UNIV\_ID=UNIVERSITY.UNIV\_ID

UNION

SELECT SURNAME, NAME, ‘Неизвестен’

FROM STUDENT

WHERE UNIV\_ID IS NULL

ORDER BY 1;

Для совместимости столбцов объединяемых запросов константу ‘Неизвестен’ во втором запросе следует дополнить пробелами так, чтобы ее длина соответствовала длине поля UNIV\_NAME, или использовать функцию CAST. В некоторых СУБД согласование типов поля и замещающей его текстовой константы осуществляется автоматически.

**2.Соединение таблиц с использованием JOIN**

Если в операторе SELECT после ключевого слова FROM указывается не одна, а две таблицы, то в результате выполнения запроса, в котором отсутствует предложение WHERE каждая строка одной таблицы будет соединена с каждой строкой второй.

Такая операция называется *декартовым произведением или полным (CROSS) соединением таблиц.*

В случае использования этой операции в ответе на запрос количество записей будет равно произведению числа записей в соединяемых таблицах, т.е. может оказаться чрезвычайно большим.

Соединение таблиц имеет смысл в том случае, если соединяются не все строки исходных таблиц, а только интересующие пользователя.

Ограничение осуществляется предложением WHERE. Таким образом SQL позволяет выводить информацию из нескольких таблиц, связывая их по значениям определенных полей.

Например, если необходимо получить фамилии студентов и для каждого студента названия университетов, расположенных в городе, где живет студент:

SELECT STUDENT.SURNAME, UNIVERSITY.UNIV\_NAME, STUDENT.CITY

FROM STUDENT, UNIVERSITY

WHERE STUDENT.CITY=UNIVERSITY.CITY;

Соединение, использующее предикаты, основанные на равенствах, называется *эквисоединением*.

Рассмотренный пример соединения таблиц относится к виду так называемого внутреннего (INNER) соединения. При этом соединяются только те строки таблиц, для которых истинным является предикат, задаваемый в предложении ON выполняемого запроса.

Приведенный выше запрос может быть записан иначе, с использованием ключевого слова JOIN.

SELECT STUDENT.SURNAME, UNIVERSITY.UNIV\_NAME,

STUDENT.CITY

FROM STUDENT INNER JOIN UNIVERSITY

ON STUDENT.CITY = UNIVERSITY.CITY;

Ключевое слово INNER в запросе может быть опущено, так как эта опция в операторе JOIN действует по умолчанию.

Рассмотренный выше случай полного соединения (декартова произведения) таблиц с использованием ключевого слова JOIN будет выглядеть следующим образом:

SELECT \* FROM STUDENT JOIN UNIVERSITY;

что эквивалентно

SELECT \* FROM STUDENT, UNIVERSITY;

Заметим, что в СУБД Oracle задаваемый стандартом языка SQL оператор JOIN не поддерживается.

**3.Операции соединения таблиц посредством ссылочной целостности**

Информация в таблицах STUDENT и EXAM\_MARKS уже связана посредством поля STUDENT\_ID. В таблице STUDENT поле STUDENT\_ID является первичным ключом, а в таблице EXAM\_MARKS — ссылающимся на него внешним ключом.

Состояние связанных таким образом таблиц называется *состоянием ссылочной целостности.*

В данном случае ссылочная целостность этих таблиц подразумевает, что каждому значению поля STUDENT\_ID в таблице EXAM\_MARKS обязательно соответствует такое же значение поля STUDENT ID в таблице STUDENT.

Другими словами, в таблице EXAM\_MARKS не может быть записей, имеющих идентификаторы студентов, которых нет в таблице STUDENT. Стандартное применение операции соединения состоит в извлечении данных в терминах этой связи.

Чтобы получить список фамилий студентов с полученными ими оценками и идентификаторами предметов, можно использовать следующий запрос:

SELECT SURNAME, MARK, SUBJ\_ID

FROM STUDENT, EXAM\_MARKS

WHERE STUDENT.STUDENT\_ID = EXAM\_MARKS.STUDENT\_ID;

Тот же результат может быть получен при использовании в запросе для задания операции соединения таблиц ключевого слова JOIN. Запрос с оператором JOIN выглядит следующим образом:

SELECT SURNAME, MARK

FROM STUDENT JOIN EXAM\_MARKS

ON STUDENT.STUDENT\_ID = EXAM\_MARKS.STUDENT\_ID;

Хотя выше речь шла о соединении двух таблиц, можно сформировать запросы путем соединения более чем двух таблиц.

Пусть требуется найти фамилии всех студентов, получивших неудовлетворительную оценку, вместе с названиями предметов обучения, по которым получена эта оценка.

SELECT SUBJ\_NAME, SURNAME, MARK

FROM STUDENT, SUBJECT, EXAM\_MARKS

WHERE STUDENT.STUDENT\_ID = EXAM\_MARKS.STUDENT\_ID

AND SUBJECT.SUBJ\_ID = EXAM\_MARKS.SUBJ\_ID

AND EXAM\_MARKS.MARK = 2;

Toже самое с использованием оператора JOIN:

SELECT SUBJ\_NAME, SURNAME, MARK

FROM STUDENT JOIN SUBJECT JOIN EXAM\_MARKS

ON STUDENT.STUDENT\_ID = EXAM\_MARKS.STUDENT\_ID

AND SUBJECT.SUBJ\_ID = EXAM\_MARKS.SUBJ\_ID

AND EXAM\_MARKS.MARK = 2;

Как отмечалось ранее, при использовании внутреннего (INNER) соединения таблиц соединяются только те их строки, в которых совпадают значения полей, задаваемые в запросе предложением WHERE.

Однако во многих случаях это может привести к нежелательной потере информации. Рассмотрим еще раз приведенный выше пример запроса на выборку списка фамилий студентов с полученными ими оценками и идентификаторами предметов. При использовании, как это было сделано в рассматриваемом примере, внутреннего соединения в результат запроса не попадут студенты, которые еще не сдавали экзамены, и которые, следовательно, отсутствуют в таблице EXAM\_MARKS.

Если же необходимо иметь записи об этих студентах в выдаваемом запросом списке, то можно присоединить сведения о студентах, не сдававших экзамен, путем использования оператора UNION с соответствующим запросом. Например, следующим образом:

SELECT SURNAME, CAST MARK AS CHAR(1), CAST SUBJ\_ID AS CHAR (10)

FROM STUDENT, EXAM\_MARKS

WHERE STUDENT.STUDENT\_ID = EXAM\_MARKS.STUDENT\_ID

UNION

SELECT SURNAME, CAST NULL AS CHAR (1), CAST NULL AS CHAR (10)

FROM STUDENT

WHERE NOT EXISTS

(SELECT \*

FROM EXAM\_MARKS

WHERE STUDENT.STUDENT\_ID = EXAM\_MARKS.STUDENT\_ID);

(здесь функция преобразования типов CAST используется для обеспечения совместимости типов полей объединяемых запросов).

Нужный результат может быть получен и путем использования внешнего соединения, точнее, одной из его разновидностей —левого внешнего соединения, с применением которого запрос будет выглядеть следующим образом:

SELECT SURNAME, MARK

FROM STUDENT LEFT OUTER JOIN EXAM\_MARKS

ON STUDENT.STUDENT\_ID = EXAM\_

MARKS.STUDENT ID;

При использовании левого соединения расширение выводимой таблицы осуществляется за счет записей входной таблицы, имя которой указано слева от оператора JOIN.

Приведенный выше запрос может быть реализован и с применением правого внешнего соединения. Он будет иметь следующий вид:

SELECT SURNAME, MARK

FROM EXAM\_MARKS RIGHT OUTER JOIN STUDENT

ON EXAM\_MARKS.STUDENT\_ID = STUDENT.STUDENT\_ID;

Здесь таблица STUDENT, за счет записей которой осуществляется расширение выводимой таблицы, указана справа от оператора JOIN.

Иногда возникает необходимость включения в результат запроса записей из обеих (правой и левой) соединяемых таблиц, для которых не удовлетворяется условие соединения. Такое соединение называется *полным внешним соединением* и осуществляется указанием в запросе ключевых слов FULL OUTER JOIN или UNION JOIN.

**3.Использование псевдонимов при соединении таблиц**

Часто при запросе информации необходимо осуществлять соединение таблицы с ее же копией.

Например, это требуется в случае, когда нужно найти фамилии студентов, имеющих одинаковые имена. При соединении таблицы с ее же копией вводят псевдонимы (алиасы) таблицы. Запрос для поиска фамилий студентов, имеющих одинаковые имена, выглядит следующим

образом:

SELECT FIRST.SURNAME, SECOND.NAME

FROM STUDENT FIRST, STUDENT SECOND

WHERE FIRST.NAME = SECOND.NAME;

В этом запросе введены два псевдонима для одной таблицы STUDENT, что позволяет корректно задать выражение, связывающее две копии таблицы.

Чтобы исключить повторения строк в выводимом результате запроса из-за повторного сравнения одной и той же пары студентов, необходимо задать порядок следования для двух значений так, чтобы одно значение было меньше, чем другое, что делает предикат асимметричным.

SELECT FIRST.SURNAME, SECOND.NAME

FROM STUDENT FIRST, STUDENT SECOND

WHERE FIRST.NAME = SECOND.NAME

AND FIRST.SURNAME < SECOND.SURNAME;

Тема 22

1. **Команды манипулирования данными**

В SQL для выполнения операций ввода данных в таблицу, их изменения и удаления предназначены три команды языка манипулирования данными (DML).

Это команды **INSERT** (вставить), **UPDATE** (обновить), **DELETE** (удалить).

Команда **INSERT** осуществляет вставку в таблицу новой строки. В простейшем случае она имеет вид:

INSERT INTO <имя таблицы> VALUES (<значение>, <значение>,);

При такой записи указанные в скобках после ключевого слова VALUES значения вводятся в поля добавленной в таблицу новой строки в том порядке, в котором соответствующие столбцы указаны при создании таблицы, то есть в операторе CREATE TABLE.

Например, ввод новой строки в таблицу STUDENT может быть осуществлен следующим образом:

INSERT INTO STUDENT

VALUES (101,'Иванов','Александр', 2000, 3,'Москва’, '6/10/2004', 15);

Чтобы такая команда могла быть выполнена, таблица с указанным в ней именем (STUDENT) должна быть предварительно определена (создана) командой CREATE TABLE.

Если в какое-либо поле необходимо вставить NULL-значение, то оно вводится как обычное значение:

INSERT INTO STUDENT

VALUES (101, 'Иванов', NULL, 2000, 3, 'Москва','6/10/2004', 15);

В случаях, когда необходимо ввести значения полей в порядке, отличном от порядка столбцов, заданного командой CREATE TABLE, или требуется ввести значения не во все столбцы, следует использовать следующую форму команды INSERT:

INSERT INTO STUDENT (STUDENT\_ID, CITY, SURNAME, NAME)

VALUES (101, 'Москва', 'Иванов', 'Саша');

Столбцам, наименования которых не указаны в приведенном в скобках списке, автоматически присваивается значение по умолчанию, если оно назначено при описании таблицы (команда CREATE TABLE), либо значение NULL.

С помощью команды INSERT можно извлечь значение из одной таблицы и разместить его в другой, например, запросом следующего вида:

INSERT INTO STUDENT1

SELECT \*

FROM STUDENT

WHERE CITY = 'Москва';

При этом таблица STUDENTI должна быть предварительно создана командой CREATE TABLE и иметь структуру, идентичную таблице STUDENT.

Удаление строк из таблицы осуществляется с помощью команды **DELETE.**

Следующее выражение удаляет все строки таблицы EXAM\_MARKS1.

DELETE FROM EXAM\_MARKS1;

В результате таблица становится пустой (после этого она может быть удалена командой DROP TABLE).

Для удаления из таблицы сразу нескольких строк, удовлетворяющих некоторому условию, можно воспользоваться предложением WHERE:

DELETE FROM EXAM\_MARKS1

WHERE STUDENT\_ID = 103;

Можно удалить группу строк:

DELETE FROM STUDENT1

WHERE CITY = 'Москва';

Команда **UPDATE** позволяет изменять, то есть обновлять значения некоторых или всех полей в существующей строке или строках таблицы.

Например, чтобы для всех университетов, сведения о которых находятся в таблице UNIVERSITYI, изменить рейтинг на значение 200, можно использовать конструкцию:

UPDATE UNIVERSITY1

SET RATING = 200;

Для указания конкретных строк таблицы, значения полей которых должны быть изменены, в команде UPDATE можно использовать предикат, указываемый в предложении WHERE.

UPDATE UNIVERSITY1

SET RATING = 200

WHERE CITY = 'Москва';

В результате выполнения этого запроса будет изменен рейтинг только у университетов, расположенных в Москве.

Команда UPDATE позволяет изменять не только один, но и множество столбцов. Для указания конкретных столбцов, значения которых должны быть модифицированы, используется предложение SET.

Например, наименование предмета обучения 'Математика' (для него SUBJ\_ID = 43) должно быть заменено на название 'Высшая математика', при этом идентификационный номер необходимо сохранить, но в соответствующие поля строки таблицы ввести новые данные об этом предмете обучения. Запрос будет выглядеть следующим образом:

UPDATE SUBJECT1

SET SUBJ\_NAME = ‘Высшая математика', HOUR = 36, SEMESTER = 1

WHERE SUBJ\_ID = 43;

В предложении SET команды UPDATE можно использовать скалярные выражения, указывающие способ изменения значений поля, в которые могут входить значения изменяемого и других полей.

UPDATE UNIVERSITY1

SET RATING = RATING\*2;

Например, для увеличения в таблице STUDENT1 значения поля STIPEND в два раза для студентов из Москвы можно использовать запрос:

UPDATE STUDENT 1

SET STIPEND = STIPEND\*2

WHERE CITY = 'Москва';

Предложение SET не является предикатом, поэтому в нем можно указать значение NULL следующим образом:

UPDATE UNIVERSITYl

SET RATING = NULL

WHERE CITY = 'Москва';

Применение оператора INSERT с подзапросом позволяет загружать сразу несколько строк в одну таблицу, используя информацию из другой таблицы.

В то время как оператор INSERT, использующий VALUES, добавляет только одну строку, INSERT с подзапросом добавляет в таблицу столько строк, сколько подзапрос извлекает из другой таблицы.

При этом количество и тип возвращаемых подзапросом столбцов должен соответствовать количеству и типу столбцов таблицы, в которую вставляются данные.

Например, пусть таблица STUDENT1 имеет структуру, полностью совпадающую со структурой таблицы STUDENT.

Запрос, позволяющий заполнить таблицу STUDENT1 записями обо всех

студентах из Москвы из таблицы STUDENT, выглядит следующим образом:

INSERT INTO STUDENT1

SELECT \*

FROM STUDENT

WHERE CITY = 'Москва';

Чтобы добавить в таблицу STUDENT1 сведения обо всех студентах, которые учатся в Москве, можно использовать в предложении WHERE соответствующий подзапрос.

INSERT INTO STUDENT1

SELECT \*

FROM STUDENT

WHERE UNIV\_ID IN

(SELECT UNIV\_ID

FROM UNIVERSITY

WHERE CITY = 'Москва');

Предположим, существует таблица SSTUD, в которой хранятся сведения о студентах, обучающихся в том же городе, в котором они живут. Можно заполнить эту таблицу данными из таблицы STUDENT, используя связанные подзапросы, следующим образом:

INSERT INTO SSTUD

SELECT \*

FROM STUDENT A

WHERE CITY IN

(SELECT CITY

FROM UNIVERSITY В

WHERE A.UNIV\_ID = B.UNIV\_ID);

Предположим, требуется выбрать список студентов, имеющих максимальный балл на каждый день сдачи экзаменов, и разместить его в другой таблице с именем EXAM. Это можно осуществить с помощью запроса

INSERT INTO EXAM

SELECT EXAM\_ID, STUDENT\_ID, SUBJ\_ID, MARK, EXAM\_DATE

FROM EXAM\_MARKS A

WHERE MARK =

(SELECT MAX (MARK)

FROM EXAM\_MARKS В

WHERE A.EXAM\_DATE = В.EXAM\_DATE);

*Использование подзапросов с DELETE*

Пусть филиал университета в Нью-Васюках ликвидирован и требуется удалить из таблицы STUDENT записи о студентах, которые там учились. Эту операцию можно выполнить с помощью запроса

DELETE

FROM STUDENT

WHERE UNIV\_ID IN

(SELECT UNIV\_ID

FROM UNIVERSITY

WHERE CITY = 'Нью-Васюки');

В предикате предложения FROM (подзапроса) нельзя ссылаться на таблицу, из которой осуществляется удаление. Однако можно ссылаться на текущую строку из таблицы, являющуюся кандидатом на удаление, то есть на строку, которая в настоящее время проверяется в основном предикате.

DELETE

FROM STUDENT

WHERE EXISTS

(SELECT \*

FROM UNIVERSITY

WHERE RATING = 401

AND STUDENT.UNIV\_ID = UNIVERSITY. UNIV\_ID);

Часть AND предиката внутреннего запроса ссылается на таблицу STUDENT. Команда удаляет данные о студентах, которые учатся в университетах, имеющих рейтинг, равный 401.

Существуют и другие способы решения этой задачи.

DELETE

FROM STUDENT

WHERE 401 IN

(SELECT RATING

FROM UNIVERSITY

WHERE STUDENT. UNIV\_ID = UNIVERSITY. UNIV\_\_ID);

Пусть нужно найти наименьшее значение оценки, полученной в каждый день сдачи экзаменов, и удалить из таблицы сведения о студенте, который получил эту оценку. Запрос будет иметь вид:

DELETE

FROM STUDENT

WHERE STUDENT\_ID IN

(SELECT STUDENT\_ID

FROM EXAM\_MARKS A

WHERE MARK=

(SELECT MIN (MARK)

FROM EXAM\_MARKS В

WHERE A.EXAM\_DATE = В.EXAMJDATE));

Так как столбец STUDENT\_ID является первичным ключом,

то удаляется единственная строка.

Если в какой-то день сдавался только один экзамен (то есть получена только одна минимальная оценка) и по какой-либо причине запись, в которой находится эта оценка, требуется оставить, то решение будет иметь вид:

DELETE

FROM STUDENT

WHERE STUDENT\_ID IN

(SELECT STUDENT\_ID

FROM EXAM\_MARKS A

WHERE MARK =

(SELECT MIN (MARK)

FROM EXAM\_MARKS В

WHERE A.EXAM\_DATE = B.EXAM\_DATE

AND 1 <

(SELECT COUNT (SUBJ\_ID)

FROM EXAM\_MARKS В

WHERE A.EXAM\_DATE = В.EXAM\_DATE)));

С помощью команды UPDATE можно применять подзапросы в любой форме, приемлемой для команды DELETE.

Например, используя связанные подзапросы, можно увеличить значение размера стипендии на 2000 в записях студентов, сдавших экзамены на 4 и 5.

UPDATE STUDENT1

SET STIPEND = STIPEND + 2000

WHERE 4 <=

(SELECT MIN (MARK)

FROM EXAM\_MARKS

WHERE EXAM\_MARKS.STUDENT\_ID = STUDENT1.STUDENT\_ID);

Другой запрос: «Уменьшить величину стипендии на 20 всем

студентам, получившим на экзамене минимальную оценку».

UPDATE STUDENT1

SET STIPEND = STIPEND - 20

WHERE STUDENT\_ID IN

(SELECT STUDENT\_ID

FROM EXAM\_MARKS A

WHERE MARK =

(SELECT MIN (MARK)

FROM EXAM\_MARKS В

WHERE A.EXAM DATE = B.EXAM DATE));

**Тема Определение прав доступа пользователей к данным**

1. **Пользователи и привилегии. Стандартные привилегии**

SQL обычно применяется в многопользовательских средах, которые требуют разграничения прав пользователей с точки зрения доступа к данным и прав на выполнение с ними тех или иных манипуляций. Для этих целей в SQL реализованы средства, позволяющие устанавливать и контролировать привилегии пользователей баз данных.

Каждый пользователь в среде SQL имеет специальное имя (идентификатор), с помощью которого осуществляется идентификация пользователя с целью установки и определения его прав с точки зрения доступа к данным. Каждая посланная к СУБД команда SQL-запроса ассоциируется СУБД с идентификатором доступа к данным конкретного пользователя.

Пользователь определяется с помощью следующей команды:

CREATE USER <username> IDENTIFIED BY <password>

Пользователь становится известен БД, но не может выполнять никаких операций.

Удаление пользователя производится командой:

DROUP USER<username>

Назначаемые привилегии –это разрешение на выполнение указанным пользователем данной команды над определенным объектом БД.

Имеется несколько типов привилегий, соответствующих нескольким типам операций.

Привилегии даются и отменяются двумя командами SQL:

GRANT –установка привилегий;

REVOKE-отмена привилегий.

Привилегии над объектом означают, что пользователь имеет право (привилегию) на выполнение данной команды только на определенном объекте в БД.

Привилегии объекта связаны одновременно и с пользователями, и с таблицами БД. То есть привилегия дается определенному пользователю в определенной таблице.

Это может быть, как базовая таблица, так и представление (т.е. таблица, полученная в результате запроса).

Пользователь, создавший таблицу любого вида, является **владельцем** этой таблицы.

То есть имеет все привилегии, относящиеся к таблице, может передавать привилегии на работу с этой таблицей другим пользователям.

Могут быть назначены следующие привилегии:

SELECT, INSERT, UPDATE, DELETE и другие.

Могут быть нестандартные привилегии объекта (INDEX, SYNONIM, ALTER, EXECUTE).

Назначение пользователям этих привилегий осуществляется с помощью команды GRANT.

1. **Команда GRANT, REVOKE**

Пользователь, являющийся владельцем таблицы STUDENT, может передать другому пользователю (IVANOV) привилегию SELECT с помощью следующей команды:

GRANT SELECT ON STUDENT TO IVANOV;

Без наличия других привилегий IVANOV может только выбирать (SELECT) значения, но не может выполнять любые действия, которые бы воздействовали на значения в таблице STUDENT, включая использование таблицы в качестве родительской таблицы внешнего ключа.

Для команды GRANT сначала проверяется допустимость этой команды для давшего ее пользователя. IVANOV дать этой команды не может.

Он также не может предоставить право SELECT другому пользователю, так как таблица принадлежит не ему. Команда:

GRANT INSERT ON EXAM\_MARKS TO IVANOV;

позволяет вводить в таблицу новые строки.

Можно передавать несколько привилегий нескольким пользователям:

GRANT SELECT, INSERT ON SUBJECT

TO IVANOV, PETROV;

Однако, в ANSI **невозможно предоставить привилегии для нескольких таблиц** одной командой GRANT.

Чтобы предоставить все привилегии в таблице используют команду ALL PRIVILEGES или ALL.

Например,

GRANT ALL PRIVILEGES ON STUDENT TO IVANOV;

Или:

GRANT ALL ON STUDENT TO IVANOV;

Передает пользователю Иванов все привилегии в таблице Студент.

Аргумент PUBLIC используется для передачи указанных в команде привилегий всем остальным пользователям. Чаще применяется для привилегии SELECT в базовых таблицах или представлениях, которые необходимо сделать доступными пользователям:

GRANT SELECT ON EXAM\_MARKS TO PUBLIC;

Позволяет любому пользователю получить информацию из таблицы EXAM\_MARKS.

Предоставление **всех** привилегий в таблице **всем** пользователям обычно является нежелательным. Так как это может вызвать определенные проблемы обеспечения безопасности и защиты данных. Поэтому для ограничения доступа к таблице лучше всего предоставить привилегии, отличные от SELECT, только индивидуальным пользователям.

Отмена привилегии осуществляется с помощью команды REVOKE, синтаксис аналогичный GRANT.

1. **Другие типы привилегий**

Теперь следует рассмотреть следующие вопросы:

-кто имеет право создавать таблицы;

-изменять, удалять, ограничивать таблицы;

-должны ли права создания базовых таблиц отличаться от прав создания представлений;

-должен ли существовать суперпользователь, т.е. пользователь, отвечающий за поддержку БД, имеющий наибольшие (полные привилегии), которые не предоставляются обычному пользователю.

Привилегии, которые не определяются в терминах специальных объектов данных, называются *привилегиями системы или правами БД.*

Эти привилегии включают в себя право создавать объекты данных, отличающиеся от базовых таблиц (обычно создаваемых несколькими пользователями) и представлений (создаваемых большинством пользователей).

Привилегии системы должны дополнять, а не заменять привилегии объекта, которые стандарт требует от создателей представлений.

В любой системе всегда имеются некоторые типы суперпользователей. То есть пользователей, которые имеют большинство или все привилегии и могут передать их (администратор БД).

Общий подход определяет три базовые привилегии системы:

CONNECT-подключить;

RESOURCE-ресурс;

DBA-администратор БД.

CONNECT-право зарегистрироваться и права создать представления и синонимы;

RESOURCE-создать базовые таблицы;

DBA-привилегия администратора, может иметь одного или более пользователей.

В большинстве реализаций SQL нового пользователя создает DBA, т.е. администратор БД, автоматически предоставляющий новому пользователю привилегию CONNECT.

В этом случае добавляется IDENTIFIED BY ‘PASSWORD’.

Например,

GRANT CONNECT TO PETROV IDENTIFIED BY ‘PASSWORD’;

**Тема 29. Хранимые процедуры**

1. **Необходимость применения хранимых процедур**

Если необходимо выполнить несколько инструкций для сложного запроса.

Допустим,

1. выяснить, есть ли товары на складе;
2. если они есть, зарезервировать их и количество, доступное другим пользователям на заказ;
3. запросить отсутствующие товары;
4. сообщить клиенту, какие товары будут отгружены, а какие запрашиваются под заказ.

Чтобы не писать по каждой операции отдельную инструкцию и выполнять ее каждый раз по ситуации необходимо создать **хранимую процедуру** (ХП)(набор из нескольких инструкций, сохраненный для последующего выполнения). Не поддерживаются в Access и SQLite.

Применяют ХП:

1. для упрощения запросов;
2. для обеспечения непротиворечивости данных за счет выполнения единого кода (повышает целостность данных);
3. при обновлении данных изменяется только код хранимой процедуры (повышает безопасность);
4. СУБД обрабатывает уже скомпилированные процедуры (повышение производительности);
5. хранимые процедуры можно применять для написания более гибкого и мощного кода.

Достоинства: простота, безопасность, производительность.

Проблемы:

1. синтаксис сильно зависит от СУБД. Вызовы хранимых процедур (имена и способы передачи аргументов могут быть переносимыми).
2. сложные в написании, поэтому многие администраторы БД ограничивают права на создание ХП в качестве меры безопасности.

Если не получается написать, используйте готовые.

1. **Выполнение хранимых процедур**

Команда EXECUTE имя процедуры (аргументы).

EXECUTE AddNewProduct (‘JTS01’,’Stuffed Eiffel Tower’, 6.49,’Plush staffed toy with the text La Tour Eiffel in red, white and blue’)

Добавляем новый товар в таблицу, у функции четыре аргумента: идентификатор поставщика, название товара, цена, описание.

ХП выполняет следующие действия:

1. подтверждает правильность передаваемых данных всем четырем аргументам;
2. генерирует у них идентификатор (первичный ключ);
3. добавляет данные о новом товаре в таблицу Products с первичным ключом в соответствующие столбцы.

Могут быть и другие варианты выполнения: опции по умолчанию, нестандартные параметры, входные параметры обновления переменной, данные, извлекаемые Select, возвращаемые коды.

1. **Создание хранимых процедур**

Создадим ХП, которая подсчитывает число клиентов, имеющих адрес почты.

CREATE PROCEDURE MailListCount AS

DECLARE @cnt INT

SELECT @cnt= COUNT (\*)

FROM Customers

WHERE NOT Cust\_email IS NULL;

RETURN @cnt;

Эта хранимая процедура не принимает аргументов.

DECLARE объявляет локальную переменную @cnt.

Затем она используется в инструкции SELECT, принимая значение, возвращаемое функцией COUNT (\*).

RETURN передает результат подсчета в вызываемое приложение.

Для запуска примера

DECLARE @ReturnValue INTEGER

EXECUTE

@ReturnValue=MailListCount

SELECT @ReturnValue;

Например,

CREATE PROCEDURE NewOrder

@Cust\_id CHAR (10) AS

--объявление переменной для номера заказа

DECLARE @Order\_num INTEGER

--получение текущего наибольшего номера заказа

SELECT @Order\_num=MAX (Order\_num)

FROM Orders

--определение следующего номера заказа

SELECT @ Order\_num= @Order\_num+1

--добавление нового заказа

INSERT INTO Orders (Order\_num, Order\_date, Cust\_id)

VALUES (@Order\_num, GetDATE (), @Cust\_id)

--возвращение номера заказа

RETURN @ Order\_num;

*--* Это универсальный синтаксис комментария. *ХП рекомендуется комментировать!*

Хранимая процедура создает новый заказ в таблице Orders и принимает один аргумент: классификатор клиента, сделавшего заказ. Номер и дата заказа генерируются автоматически в самой хранимой процедуре. Вначале объявляется локальная переменная для хранения номера заказа, затем запрашивается текущий максимальный номер заказа, увеличивающийся на один с помощью SELECT.

Затем с помощью INSERT добавляется новый заказ с использованием сгенерированного номера заказа, текущей системной даты, которая определяется с помощью RETURN @Order\_num.

В SQL Server можно реализовать эту хранимую процедуру и таким образом:

CREATE PROCEDURE New\_Order @cust\_id CHAR (10)

AS

--Добавление нового заказа

INSERT INTO Orders (Cust\_id)

VALUES (@Cust\_id)

--Возвращение номера заказа

SELECT Order\_num=@@IDENTITY;

Данная хранимая процедура создает новый заказ в таблице Заказы (Orders). СУБД сама генерирует №заказа (поле автогенерации). Процедуре передается только id клиента. № и дата\_заказа не указываются вообще, так как СУБД использует значение по умолчанию, для даты и № оно генерируется автоматически.

Какой же id был сгенерирован? В SQL Server для этого есть глобальная переменная @@IDENTTY, возвращаемая в вызывающее приложение с помощью SELECT.

Выходные параметры позволяют возвратить из процедуры некоторый результат. Выходные параметры определяются с помощью ключевого слова OUTPUT. Например, определим еще одну процедуру:

USE productsdb;

GO

CREATE PROCEDURE GetPriceStats

@minPrice MONEY OUTPUT,

@maxPrice MONEY OUTPUT

AS

SELECT @minPrice = MIN(Price), @maxPrice = MAX(Price)

FROM Products

При вызове процедуры для выходных параметров передаются переменные с ключевым словом OUTPUT:

USE productsdb;

DECLARE @minPrice MONEY, @maxPrice MONEY

EXEC GetPriceStats @minPrice OUTPUT, @maxPrice OUTPUT

PRINT 'Минимальная цена ' + CONVERT(VARCHAR, @minPrice)

PRINT 'Максимальная цена ' + CONVERT(VARCHAR, @maxPrice)

Также можно сочетать входные и выходные параметры. Например, определим процедуру, которая добавляет новую строку в таблицу и возвращает ее id:

USE productsdb;

GO

CREATE PROCEDURE CreateProduct

@name NVARCHAR(20),

@manufacturer NVARCHAR(20),

@count INT,

@price MONEY,

@id INT OUTPUT

AS

INSERT INTO Products(ProductName, Manufacturer, ProductCount, Price)

VALUES(@name, @manufacturer, @count, @price)

SET @id = @@IDENTITY

С помощью глобальной переменной @@IDENTITY можно получить идентификатор добавленной записи.

При вызове этой процедуры ей также по позиции передаются все входные и выходные параметры:

USE productsdb;

DECLARE @id INT

EXEC CreateProduct 'LG V30', 'LG', 3, 28000, @id OUTPUT

PRINT @id

Возвращение значения

Кроме передачи результата выполнения через выходные параметры хранимая процедура также может возвращать какое-либо значение типа INT с помощью оператора RETURN. Хотя данная возможность во многом нивелирована использованием выходных параметров, через которые можно возвращать результат, тем не менее, если надо возвратить из процедуры одно значение, то вполне можно использовать оператор RETURN.

Например, возвратим среднюю цену на товары:

USE productsdb;

GO

CREATE PROCEDURE GetAvgPrice AS

DECLARE @avgPrice MONEY

SELECT @avgPrice = AVG(Price)

FROM Products

RETURN @avgPrice;

После оператора RETURN указывается возвращаемое значение. В данном случае это значение переменной @avgPrice.

Вызовем данную процедуру:

USE productsdb;

DECLARE @result MONEY

EXEC @result = GetAvgPrice

PRINT @result

Для получения результата процедуры ее значение сохраняется в переменную (в данном случае в переменную @result):

Стоит отметить, что RETURN возвращает только целочисленные значения.

**Тема 31. Программирование**

**1.Применение переменных.**

**2.Выполнение условных вычислений.**

**3.Циклическое повторение процессов.**

1. Иногда задачи оказываются более сложными и требуют применения нескольких операторов, условных вычислений и промежуточного манипулирования данными.

В T-SQL поддерживаются некоторые из основных принципов программирования и программных конструкций, чаще они применяются с хранимыми процедурами, триггерами и другими средствами БД.

Это применение переменных, выполнение условных вычислений, циклическое повторение процессов.

Правила и требования к переменным:

1. Имена всех переменных должны начинаться со знака @, глобальных -@@.
2. Перед тем, как пользоваться переменная должна быть объявлена с помощью DECLARE.
3. При объявлении указывается тип.
4. Несколько переменных можно объявить в отдельных операторах DECLARE или через запятую в одном.
5. Сделать переменные «необъявленными» нельзя. Локальные переменные существуют пока не завершится процесс.

Например

DECLARE @age int;

DECLARE @firstName char(20);

DECLARE @lastName char(20);

Когда переменные объявляются, то они не содержат никаких значений (т.е. NULL).

Чтобы присвоить им значения используют SET.

SET @lastName=’\_\_\_\_\_\_\_\_\_’;

SET @firstName=’\_\_\_\_\_\_\_\_\_’;

SET @age=\_\_\_;

Необходимо при инициализации установить либо значения по умолчанию, либо первоначальные значения.

Можно присваивать и значения с помощью SELECT, например

SELECT @age=\_\_\_;

SET или SELECT?

С помощью SET можно присвоить значение только одной переменной. SET записывают несколько раз для каждой переменной. А в одном SELECT можно присвоить несколько значений.

Например, выведем содержимое переменных для проверки:

SELECT @lastName, @firstName, @age;

SURNAME NAME 20

SELECT возвращает содержимое переменных, а не их имена. Можно в SELECT использовать AS, также оператор PRINT.

Например

PRINT @lastName+ ‘,’+@firstName,

PRINT @age;

SURNAME ,NAME

20

Чтобы вывести в одну строку:

PRINT ‘Age:’+Convert(char, @age);

Проверить содержимое переменных можно с помощью отладчика, в состав входит доступная для изучения документация на отладчик.

Допустим требуется выполнить два запроса:

1-получить сведения о конкретном заказчике,

2-возврат заказов по датам

SELECT cust\_name, cust\_email

FROM customers

WHERE cust\_id=10001;

SELECT order\_num,order\_date

FROM orders

WHERE cust\_id=10001

ORDER BY order\_date;

В таком случае для другого заказчика необходимо обновлять условие!

--объявим переменную @cust\_id

DECLARE @cust\_id INT;

SET @cust\_id=10001;

--Получить имя заказчика и адрес его почты

SELECT cust\_name, cust\_email

FROM customers

WHERE cust\_id=@cust\_id

--Получить историю его заказов

SELECT order\_num,order\_date

FROM orders

WHERE cust\_id=@cust\_id

ORDER BY order\_date;

Имена переменных не следует заключать в одиночные кавычки!

2. В T-SQL предоставляется возможность писать код с вариантами выбора дальнейших действий во время выполнения (IF).

Например, необходимо обновить данные заказа, если день рабочий и не обновлять , если воскресенье.

Функция

GetDate ()-текущая дата, время

DatePart (dw, GetDate());-текущий день недели

--Объявить переменные

DECLARE @open BIT;

--Сегодня рабочий день?

IF DatePart(dw, GetDate())=7

SET @Open=0;

ELSE

SET @Open=1;

--Вывести результат

SELECT @Open AS OpenForBusiness

OpenForBusiness

1

3. В SQL поддерживается организация циклов, способных неоднократно повторять блок кода по мере надобности.

Циклы организуются с помощью оператора WHILE. Часто оператор применяется в сочетании с курсорами.

Например

DECLARE @counter INT;

SET @counter =1;

WHILE @counter<=10

BEGIN

PRINT @counter;

SET @counter=@counter+1;

END

Результат:

1

2

3

4

5

6

7

8

9

10

BREAK –прерывание цикла, CONTINUE-перезапуск обработки с начала цикла.

**1.Использование COUNT вместо EXISTS**

При отсутствии NULL-значений оператор EXISTS может быть использован вместо ANY и ALL.

Также вместо EXISTS и NOT EXISTS могут быть использованы те же самые подзапросы, но с использованием COUNT (\*) в предложении SELECT. Например, запрос

SELECT \* FROM UNIVERSITY A

WHERE NOT EXISTS

(SELECT \* FROM UNIVERSITY В

WHERE A.RATING > = B.RATING AND B.CITY = 'Москва');

может быть представлен и в следующем виде:

SELECT \* FROM UNIVERSITY A

WHERE 1 >

(SELECT COUNT (\*) FROM UNIVERSITY В

WHERE A.RATING > = B.RATING AND B.CITY = 'Москва');

**2.Оператор объединения UNION**

Оператор UNION используется для объединения выходных данных двух или более SQL-запросов в единое множество строк и столбцов.

Например, для того чтобы получить в одной таблице фамилии и идентификаторы студентов и преподавателей из Москвы, можно использовать следующий запрос:

SELECT 'Студент ', SURNAME, STUDENT\_ID

FROM STUDENT

WHERE CITY = 'Москва'

UNION

SELECT 'Преподаватель', SURNAME, LECTURER\_ID

FROM LECTURER

WHERE CITY = 'Москва';

Обратите внимание на то, что символом «;» оканчивается только последний запрос. Отсутствие этого символа в конце SELECT-запроса означает, что следующий за ним запрос так же, как и он сам, является частью общего запроса с UNION.

Использование оператора UNION возможно только при объединении запросов, соответствующие столбцы которых *совместимы по объединению*, то есть соответствующие числовые поля должны *иметь полностью совпадающие тип и размер,*

символьные поля должны иметь точно *совпадающее количество символов*. Если NULL-значения запрещены для столбца хотя бы одного любого подзапроса объединения, то они должны быть запрещены и для всех соответствующих столбцов в других подзапросах объединения.

**3.Устранение дублирования в UNION**

В отличие от обычных запросов UNION автоматически исключает из выходных данных дубликаты строк, например, в запросе

SELECT CITY

FROM STUDENT

UNION

SELECT CITY

FROM LECTURER;

совпадающие наименования городов будут исключены.

Если все же необходимо в каждом запросе вывести все строки независимо от того, имеются ли такие же строки в других объединяемых запросах, то следует использовать во множественном запросе конструкцию с оператором UNION ALL.

Так, в запросе

SELECT CITY

FROM STUDENT

UNION ALL

SELECT CITY

FROM LECTURER;

дубликаты значений городов, выводимые второй частью запроса, не будут исключаться.

Приведем еще один пример использования оператора UNION.

Пусть необходимо составить отчет, содержащий для каждой даты сдачи экзаменов сведения по каждому студенту, получившему

максимальную или минимальную оценки.

SELECT 'МАКСОЦ', A.STUDENT\_ID, SURNAME, MARK, EXAM\_DATE

FROM STUDENT A, EXAM\_MARKS В

WHERE (A.STUDENT\_ID = B.STUDENT\_ID

AND B.MARK =

(SELECT MAX (MARK)

FROM EXAM\_MARKS С

WHERE C.EXAM\_DATE = B.EXAM\_DATE))

UNION ALL

SELECT 'МИНОЦ’, A.STUDENT\_ID, SURNAME, MARK, EXAM\_DATE

FROM STUDENT A, EXAM\_MARKS В

WHERE (A.STUDENT\_ID = B.STUDENT\_ID

AND B.MARK =

(SELECT MIN (MARK)

FROM EXAM\_MARKS С

WHERE C.EXAM\_DATE = В.EXAM\_DATE));

Для отличия строк, выводимых первой и второй частями запроса, в них вставлены текстовые константы 'Макс оц' и 'Мин оц'.

В приведенном запросе агрегирующие функции используются в подзапросах. Это является нерациональным с точки зрения времени, затрачиваемого на выполнение запроса. Более эффективна форма запроса, возвращающего аналогичный результат:

SELECT 'МаКСОЦ', A.STUDENT\_ID, SURNAME, E.MARK, E.EXAM\_DATE

FROM STUDENT A,

(SELECT B.STUDENT\_ID, B.MARK, B.EXAM\_DATE

FROM EXAM\_MARKS B,

(SELECT MAX (MARK) AS MAX\_MARK, C.EXAM\_DATE

FROM EXAM\_\_MARKS С

GROUP BY C.EXAM\_DATE) D

WHERE В.EXAM\_DATE=D.EXAM\_DATE

AND B.MARK=MAX\_MARK) E

WHERE A.STUDENT\_ID=E.STUDENT\_ID

UNION ALL

SELECT 'МИНОЦ', A.STUDENT\_\_ID, SURNAME, E.MARK, E .EXAM\_DATE

FROM STUDENT A,

(SELECT B.STUDENT\_ID, B.MARK, B.EXAM\_DATE

FROM EXAM\_MARKS B,

(SELECT MIN (MARK) AS MIN\_MARK, C.EXAM\_DATE

FROM EXAM\_MARKS С

GROUP BY C.EXAM\_DATE) D

WHERE B.EXAM\_DATE=D.EXAM\_DATE

AND B.MARK=MIN\_MARK) E

WHERE A.STUDENT\_ID=E.STUDENT\_ID;

**3.Использование UNION с ORDER BY**

Предложение ORDER BY применяется для упорядочения выходных данных объединения запросов так же, как и для отдельных запросов. Последний пример, при необходимости упорядочения выходных данных запроса по фамилиям студентов и датам

экзаменов, может выглядеть следующим образом:

SELECT 'МАКСОЦ', A.STUDENT\_ID, SURNAME, E.MARK, E.EXAM\_DATE

FROM STUDENT A,

(SELECT B.STUDENT\_ID, B.MARK, B.EXAM\_DATE

FROM EXAM\_MARKS B,

(SELECT MAX (MARK) AS MAX\_MARK, C.EXAM\_DATE

FROM EXAM\_MARKS С

GROUP BY C.EXAM\_DATE) D

WHERE B.EXAM\_DATE=D.EXAM\_DATE

AND B.MARK=MAX\_MARK) E

WHERE A.STUDENT\_ID=E.STUDENT\_ID

UNION ALL

SELECT 'МИНОЦ', A.STUDENT\_ID, SURNAME, E.MARK, E.EXAM\_DATE

FROM STUDENT A,

(SELECT B.STUDENT\_ID, B.MARK, B.EXAM\_DATE

FROM EXAM\_MARKS B,

(SELECT MIN (MARK) AS MIN\_MARK, C.EXAM\_DATE

FROM EXAM\_MARKS С

GROUP BY C.EXAM\_DATE) D

WHERE В.EXAM\_DATE=D.EXAM\_DATE

AND B.MARK=MIN\_MARK) E

WHERE A.STUDENT\_ID=E.STUDENT\_ID

ORDER BY SURNAME, E.EXAM\_DATE;

1. **Использование оператора EXISTS**

Нужно извлечь из таблицы EXAM\_MARKS данные о студентах, получивших хотя бы одну неудовлетворительную оценку:

SELECT DISTINCT STUDENT\_ID

FROM EXAM\_MARKS A

WHERE EXISTS

(SELECT \* FROM EXAM\_MARKS В

WHERE MARK < 3

AND B.STUDENT\_ID = A.STUDENT\_ID);

Получить идентификаторы предметов обучения, экзамены по которым сдавались не одним, а несколькими студентами:

SELECT DISTINCT SUBJ\_ID

FROM EXAM\_MARKS A

WHERE EXISTS

(SELECT \*

FROM EXAM\_MARKS В

WHERE A.SUBJ\_\_ID = B.SUBJ\_ID

AND A.STUDENT\_ID < > В.STUDENT\_ID);

Из таблицы STUDENT требуется извлечь строки для каждого студента, сдавшего более одного предмета.

SELECT \*

FROM STUDENT FIRST

WHERE EXISTS

(SELECT SUBJ\_ID

FROM EXAM\_MARKS SECOND

GROUP BY SUBJ\_ID

HAVING COUNT (SUBJ\_ID) > 1

WHERE FIRST.STUDENT ID = SECOND.STUDENT ID);

**2.Операторы сравнения с множеством значений**

**IN, ANY, All**

Операторы сравнения с множеством значений имеют следующий смысл.

|  |  |
| --- | --- |
| IN | Равно любому из значений, полученных во внутреннем запросе. |
| NOT IN | Не равно ни одному из значений, полученных во внутреннем запросе. |
| = ANY | То же, что и IN, соответствует логическому оператору OR. |
| > ANY, > = ANY | Больше, чем (либо больше или равно) любое полученное число. Эквивалентно > или > = для самого меньшего полученного числа. |
| < ANY, < = ANY | Меньше, чем (либо меньше или равно) любое полученное число. |
| =ALL | Равно всем полученным значениям. Эквивалентно логическому оператору AND |
| >ALL, >=ALL | Больше, чем (либо больше или равно) все полученные числа. Эквивалент > или > = для самого  большего полученного числа. |
| <ALL, <=ALL | Меньше, чем (либо меньше или равно) все полученные числа. Эквивалентно < или < = самого  меньшего полученного числа. |

*Следует иметь в виду, что в некоторых СУБД поддерживаются не все из этих операторов!*

Выбрать сведения о студентах, проживающих в городе, где

расположен университет, в котором они учатся.

SELECT \*

FROM STUDENT S

WHERE CITY = ANY

(SELECT CITY

FROM UNIVERSITY U

WHERE U.UNIV\_ID = S.UNIV\_ID);

Другой вариант этого запроса:

SELECT \*

FROM STUDENT S

WHERE CITY IN

(SELECT CITY

FROM UNIVERSITY U

WHERE U.UNIV\_ID = S. UNIV\_ID);

Оператор ALL, как правило, эффективно используется с неравенствами, а не с равенствами, поскольку значение равно всем, которое должно получиться в этом случае в результате выполнения подзапроса, может иметь место, только если все результаты идентичны.

Такая ситуация практически не может быть реализована,

*так как если подзапрос генерирует множество различных значений, то никакое одно значение не может быть равно сразу всем значениям в обычном смысле.*

В SQL выражение < > ALL реально означает не равно ни одному из результатов подзапроса.

Подзапрос, выбирающий данные о названиях всех университетов с рейтингом более высоким, чем рейтинг любого университета Воронежа:

SELECT \*

FROM UNIVERSITY

WHERE RATING > ALL

(SELECT RATING

FROM UNIVERSITY

WHERE CITY = 'Воронеж');

В этом запросе вместо ALL можно использовать ANY (проанализируйте, как в этом случае изменится смысл приведенного запроса):

SELECT \*

FROM UNIVERSITY

WHERE NOT RATING <= ANY

(SELECT RATING

FROM UNIVERSITY

WHERE CITY = 'Воронеж');

**3.Особенности применения операторов ANY, ALL, EXISTS для NULL-значений**

Необходимо иметь в виду, что при обработке NULL-значений следует учитывать различие реакции на них операторов EXISTS, ANY и ALL.

Когда правильный подзапрос не генерирует никаких выходных данных, оператор ALL автоматически принимает значение истина, а оператор ANY — значение ложь.

Запрос

SELECT \*

FROM UNIVERSITY

WHERE RATING > ANY

(SELECT RATING

FROM UNIVERSITY

WHERE CITY = 'New York');

не генерирует выходных данных, в то время как запрос

SELECT \*

FROM UNIVERSITY

WHERE RATING > ALL

(SELECT RATING

FROM UNIVERSITY

WHERE CITY = 'New York');

полностью воспроизведет таблицу UNIVERSITY.

Использование NULL-значений создает определенные проблемы для рассматриваемых операторов. Когда в SQL сравниваются два значения, одно из которых NULL-значение, результат принимает значение UNKNOWN (неизвестно).

Предикат UNKNOWN, так же, как и FALSE-предикат, создает ситуацию, когда строка не включается в состав выходных данных, но результат при этом будет различен для разных типов запросов, в зависимости от использования в них ALL или ANY вместо EXISTS. Рассмотрим в качестве примера две реализации запроса:

Найти все данные об университетах, рейтинг которых меньше рейтинга любого университета в Москве.

1) SELECT \*

FROM UNIVERSITY

WHERE RATING < ANY

(SELECT RATING

FROM UNIVERSITY

WHERE CITY = 'Москва');

2) SELECT \*

FROM UNIVERSITY A

WHERE NOT EXISTS

(SELECT \*

FROM UNIVERSITY В

WHERE A.RATING >= B.RATING

AND B.CITY = 'Москва');

При отсутствии в таблицах NULL оба эти запроса ведут себя совершенно одинаково. Пусть теперь в таблице UNIVERSITY есть строка с NULL-значениями в столбце RATING. В версии запроса с ANY в основном запросе, когда выбирается поле RATING с NULL, предикат принимает значение UNKNOWN и строка не включается в состав выходных данных. Во втором же варианте запроса, когда NOT EXISTS выбирает эту строку в основном запросе, NULL-значение используется в предикате подзапроса, присваивая ему значение UNKNOWN. Поэтому в результате выполнения подзапроса не будет получено ни одного значения, и подзапрос примет значение ложь. Это в свою очередь сделает NOT EXISTS истинным, и, следовательно, строка с NULL-значением в поле RATING попадет в выходные данные. По смыслу запроса такой результат является неправильным, так как на самом деле рейтинг университета, описываемого данной строкой, может

быть и больше рейтинга какого-либо московского университета (он просто неизвестен). Указанная проблема связана с тем, что значение EXISTS всегда принимает значения истина или ложь, и никогда — UNKNOWN .

Это является доводом для использования в таких случаях оператора ANY вместо EXISTS.

Лекция

Представления

Это виртуальные таблицы, содержащие данные извлекаемые запросом, когда это необходимо.

Например, выбрать данные о преподавателях, которые вели заданный предмет:

Select lector.surname, lector.name, subject.name, subject.id

From lecturer, subject, lecturer-subject

Where lecturer.lecturerid=subj.lecturer\_id

And subject.subj\_id=subj.subj\_id

And subj\_id=24;

Чтобы извлечь данные по другому предмету условия WHERE необходимо модифицировать.

Сохраним запрос в виде виртуальной таблицы, назовем lecturer\_subject.

Теперь можно записать следующий запрос

Select lecturer.surname, lector.name, subject.name, subject.id

From lecturer\_subject

Where subj\_id=24;

Таблица lecturer-subject является представлением, поскольку не содержит каких-либо столбцов данных, в ней хранится запрос.

Синтаксис создания представлений одинаков во всех основных СУБД.

Часто представления применяют для выполнения следующих операций:

* Повторное использование инструкций SQL
* Упрощение сложных запросов
* Вывод фрагментов таблицы
* Защита данных (можно предоставить доступ только к определенному подмножеству таблиц)
* Изменение формата и способа отображения данных (не так как в основных таблицах)

Использовать представления можно также, как и таблицы (выборка, фильтр, сортировка, объединение с другими представлениями или таблицами, обновление/удаления).

Представление — это виртуальная таблица данные о которой хранятся в других таблицах(основных), если данные в таблицах изменятся, представления обновятся автоматически.

Если создать сложное представление с объединениями и фильтрами, с вложенными представлениями, производительность СУБД резко снизится.

При создании представлений необходимо обращаться к документации СУБД.

Общие правила и ограничения:

* Представления должны иметь уникальные имена
* Не существует ограничений на количество представлений, которые могут быть созданы
* Для того чтобы создать представление необходимо иметь права доступа
* Представления могут быть вложенными, уровень вложения зависит от СУБД
* Во многих СУБД запрещается использовать ORDER BY в запросах-представлениях
* В некоторых СУБД требуется, чтобы каждый возвращаемый столбец имел имя (псевдонимы для вычисляемых полей)
* Нельзя индексировать представления, не могут иметь триггеров или связанных с ними значений по умолчанию

Создание представлений

Команда создания CREATE VIEW \_имя\_представления;

Удаление DROP VIEW \_имя\_представления;

Чтобы перезаписать (обновить представление) нужно выполнить команду DROP VIEW, а затем заново создать.

Примеры

Использование представлений для упрощения объединения таблиц

Create view mark\_student

As

Select student.surname, exam\_marks.mark, subject.subj\_name

From student, exam\_marks, subject

Where student.student\_id=exam\_marks.student\_id

And

Exam\_marks.subj\_id=subject.subj\_id;

Select \* from mark\_student; –чтобы увидеть что сделали

Использование

Select student.surname, exam\_marks.mark, subject.subj\_name

From mark\_student

Where subj\_name = ‘Информатика’;

Когда СУБД обрабатывает запрос она добавляет указанное условие (Where subj\_name = ‘Информатика’) к уже существующему предложению Where в запросе самого представления.

Расширение диапазона представления позволяет его использовать многократно и не хранить похожие.

Представление для переформатирования данных

Создадим представление используя запрос с конкатенацией строк (их слиянием)

Create view lecturer\_location

As

Select lecturer.surname || ‘(‘ || lecturer.city || ‘)’ as lecturer.city

From lecturer

Order by lecturer.surname;

Select \* from lecturer\_location;

Для фильтрации нежелательных данных

Create View Bir\_stud as

Select bir, surname, name

From student

Where bir is not null;

Select \* from Bir\_stud;

Если условие используется для извлечения данных из самого представления два условия комбинируются автоматически.

Представление с вычисляемыми полями

Create View цена\_товара as

Select номер\_товара, количество, цена, количество\*цена

As общая\_цена

From товары;

Select \* from цена\_товара

Операции модификации представлений

Создадим представление, выбирающее всех студентов, живущих в Москве

Create view mosk\_stud as

Select \* from student

Where сity=’Москва’;

К таблице mosk\_stud можно обращаться с помощью запроса также, как и к любой другой.

Рассмотреть представление

Select \* from mosk\_stud

Могут быть представления таблиц и столбцов.

Если выбираются все строки и столбцы базовой таблицы, то представление эквивалентно синониму – неэффективно и применяется редко.

Представления столбцов, выбирает все строки и столбцы. В качестве имен столбцов используются псевдонимы.

Create view new\_student (new\_id, new\_name, new\_surname) as

Select student\_id, name, surname…

From Student;

Данные которые передаются пользователю через представление могут быть изменены, при этом фактические изменения произойдут в базовой таблице.

Update new\_student

Set city = ‘Москва’

Where student\_id = 104;

Аналогично update базовой таблицы.

Могут быть маскирующие представления, ограничивающие число столбцов, к которым возможен доступ.

CREATE VIEW STUD AS

SELECT STUDENT\_ID, NAME, SURNAME

FROM STUDET;

Пользователь не видит ни содержимое, ни фактическое наличие остальных полей.

Представления могут изменяться с помощью команд-модификаций. Модификации будет осуществляться в базовой таблице. Поэтому в представлениях функции вставки, удаления, изменения работают иначе чем в таблицах.

Insert - осуществляет вставку строки во все столбцы базовой таблицы, независимо от того, видны они пользователю или нет.

В столбцах, не включенных в представления устанавливаются null-значения или значения по умолчанию.

Если не включённый в представление столбец имеет опцию not-null, то будет сообщение об ошибке.

Delete - удаляет строки базовой таблицы независимо от значений. Можно ограничить доступ и к строкам.

Каждая включенная в представление строка доступна для вывода, обновления и удаления.

Create view hight\_rating as

Select \* from university

Where rating = 300;

Представление ограничивает доступ пользователя к определенным столбцам и строкам в таблице university, оно является обновляемым.

Нужно вставить с помощью insert следующую строку:

Insert into hight\_rating

Values (180, ‘новый универ’,200, ‘Воронеж’);

С помощью представления строка вставится в базовую таблицу, однако в самом представлении она исчезнет, так как не соответствует условию where. То есть пользователь эту строку не видит, не знает с таблицей он работает или с представлением.

Такие проблемы можно устранить путем включения в представлении опции with check option.

Например,

Create view hight\_rating as

Select \* from university

Where rating = 300

with check option;

Операции вставки в строку или коррекция поля rating будут отклонены.

Опция вставляется в определенное представление, а не в команду DMA.

Поэтому все команды модификации представлений будут проверяться.

Рекомендуется использовать опцию во всех случаях, когда нет причины разрешать представлению помещать в таблицу значение, которое в нем самом не видны.

Например,

Create view mosk\_stud as

Select student\_id, surname////

Where city = ‘Москва’;

Данное представление не включено в поле city, при попытке вставки строки в представление будет вставляться null в базовую таблицу, где city не равно Москва. Вставляемая строка будет исключена из представления и не видна пользователю. Пользователь не сможет видеть строки, даже при использовании with check option!!!!!!0\_0!!! То есть в (этом) представлении можно можно модифицировать значения полей или удалять строки, но нельзя вставлять строки.

Поэтому рекомендуется при определении представления включать в него все поля на которые имеется ссылка в предикате (условии).

Если эти поля не должны отображаться в выводе таблицы, из можно исключить запросом.

То есть,

Create view mosk\_stud as

Select \* from student

Where city = ‘Москва’

with check option;

Команда заполнит city одинаковыми значениями, которые можно исключить из вывода с помощью другого запроса.

Select student\_id, surname, stipend

From mosk\_stud;

Допустим необходимо следить за количеством студентов, сдающих экзамены, количеством сданных предметов и средним баллом по каждому предмету.

Create view total\_day as

Select exam\_date, count(distinct subj\_id) as subj\_count, count(student\_id) as stud\_count, count(mark) as mark\_count, AVG(mark) as avg\_mark, sum(mark) as mark\_sum

From exam\_marks

Group exam\_date;

Select \* from total\_day

**Лаборат****орная работа №1**

**Установка соединения с сервером Microsoft SQL Server.**

**Создание, копирование и удаление базы данных**

Цель: Познакомиться с основными принципами создания базы данных в MS SQL Server. Получить навыки для создания, удаления, резервного копирования и восстановления базы данных. Изучить SQL-операторы для создания, подключения и удаления базы данных.

Методические указания.

Microsoft SQL Server — система управления реляционными базами данных (РСУБД), разработанная корпорацией Microsoft. Основной используемый язык запросов — Transact-SQL. Для выполнения работ из практикума рекомендуется использовать версию 2012 или 2019.

Для начала давайте вспомним, что такое база данных, и из каких объектов она состоит. Как правило БД – это не только таблицы и связи между ними, но также представления, процедуры и функции. С ними мы будем работать в следующих лабораторных работах. Такая система хранится в двух и более файлах.

Давайте рассмотрим из каких файлов состоит БД.

У файлов существуют два имени: логическое и физическое.

Логическое имя подчиняется стандартным правилам выбора имен объектов SQL Server.

Физическое имя представляет собой полное имя любого локального или сетевого файла. Максимальное число файлов в базе данных — 32 768.

Файлы делятся на три типа:

• Первичные файлы. Используются для хранения данных и информации, определяющих начальные действия с базой. База данных содержит лишь один первичный файл. Стандартное расширение — .mdf.

• Вторичные файлы. Одна или несколько вспомогательных областей для хранения данных. Могут использоваться для распределения операций чтения/записи по нескольким дискам. Стандартное расширение — .ndf.

• Файлы журналов. Содержат журналы транзакций базы данных. База данных содержит по крайней мере один файл журнала. Стандартное расширение — .ldf. Перед непосредственной записью транзакций в файл данных все вносимые изменения записываются в журнал.

Каждый файл может использоваться лишь одной базой данных.

Также существует такое понятие как группы файлов. Группы файлов предназначены для объединения нескольких файлов. Каждый файл может входить не более чем в одну группу. Файлы журналов не могут принадлежать никаким группам. Группы файлов используются для распределения операций чтения/записи по нескольким дискам. Если группа содержит более одного файла, операции записи распределяются между файлами группы. Базы данных могут содержать до 32 768 групп файлов.

У каждой базы данных имеется первичная группа файлов. Она содержит первичный файл данных и все файлы, которые не были явно назначены в другую группу файлов. Имя первичной группы файлов — PRIMARY.

В SQL Server используется два типа баз данных: системные и пользовательские. Системные базы данных необходимы серверу SQL для корректной работы. А пользовательские базы данных создаются пользователями сервера и могут хранить любую произвольную информацию. Их можно изменять и удалять, создавать заново. Собственно, это те базы данных, которые мы будем создавать и с которыми мы будем работать.

К системным базам данных относятся: master, model, msdb, tempbd. База данных master является самой главной базой данных сервера, в случае ее отсутствия или повреждения сервер не сможет работать. Она хранит все используемые логины пользователей сервера, их роли, различные конфигурационные настройки, имена и информацию о базах данных и другую информацию. Также, если не было дополнительных настроек, она является базой данных по умолчанию, то есть при подключении к серверу, вы автоматически начинаете работу в БД master.

Самостоятельно прочитайте про назначение и размещение системных БД.

Теперь можем приступать к работе с сервером. Для начала убедитесь, что он запущен.

Для этого запустите SQL Server Configuration Manager. В диспетчере конфигурации найдите «Службы SQL Server» и убедитесь, что SQL Server находится в состоянии «Работает». Если нет, нажмите правой кнопкой мыши на состояние и выберите «Запустить».

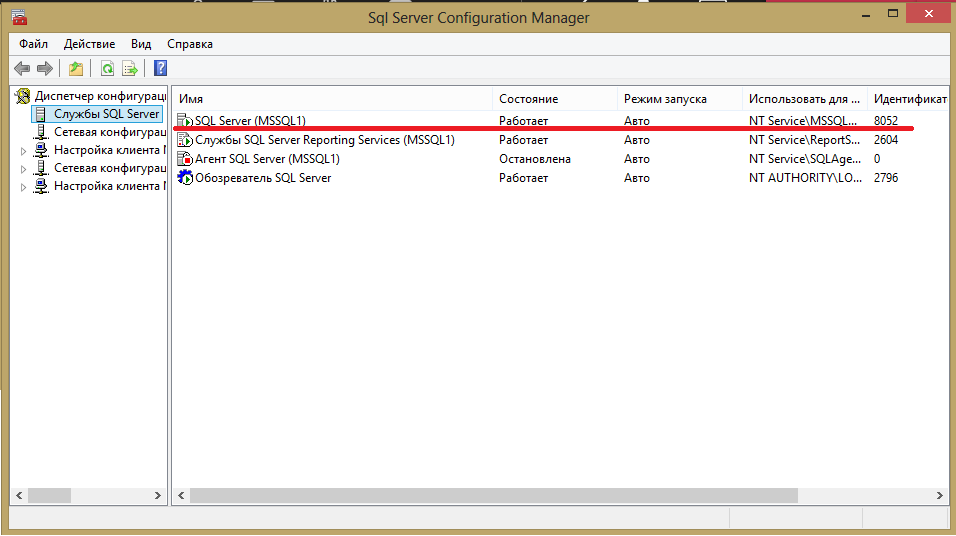


Рис.1. Правильное состояние SQL Server

После того, как вы убедились, что сервер запущен, можете запускать Microsoft SQL Management Studio.

У вас появится окно «Соединение с сервером». Давайте его рассмотрим подробнее.

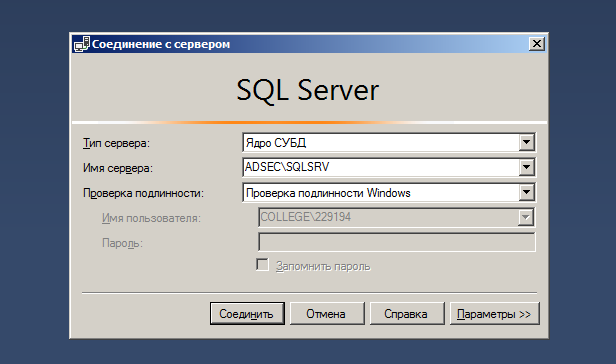


Рис.2 Окно «Соединение с сервером»

Выберите в качестве типа сервера «Компонент Database Engine» и введите имя сервера (если оно вам неизвестно, спросите у преподавателя).

Обратите внимание, что в MS SQL Server поддерживается два варианта проверки подлинности: Windows и SQL Server.

После ввода данных нажмите кнопку «Соединить». Вам откроется следующий интерфейс.

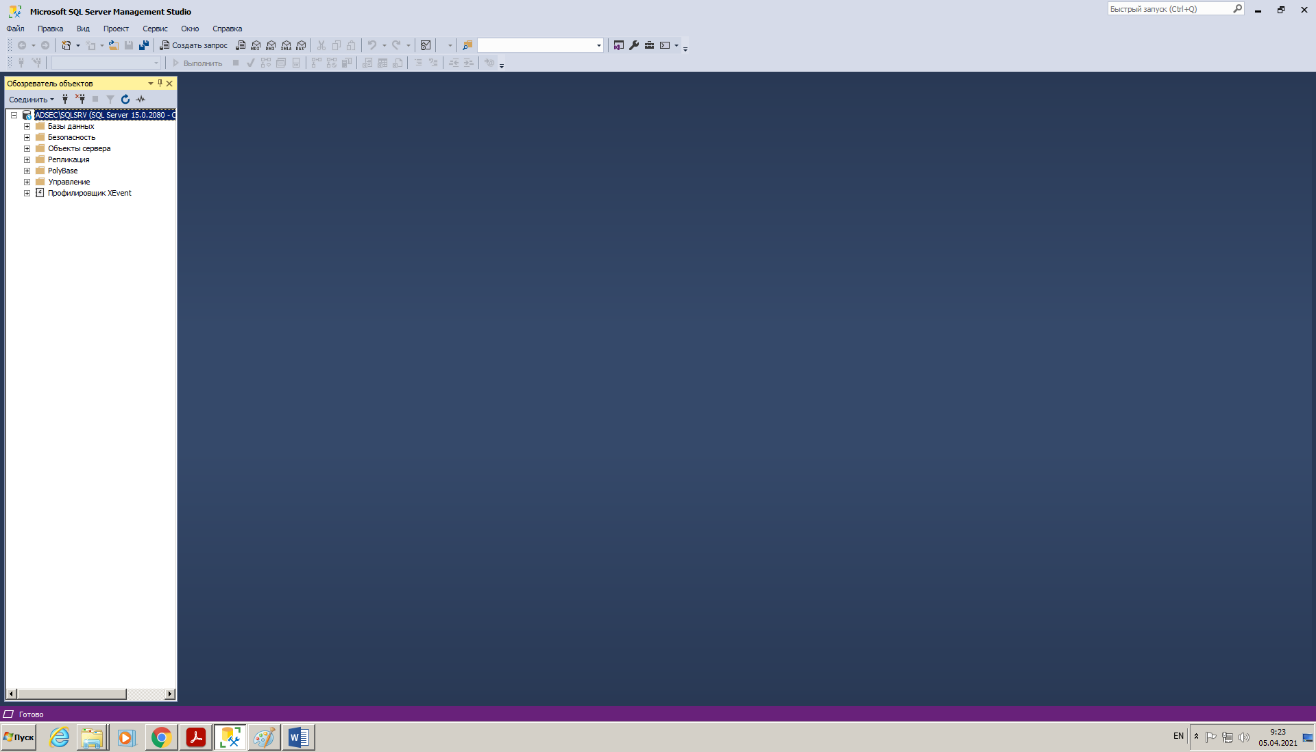


Рис. 3 Окно работы с MS SQL Server Management Studio

Создать базу данных можно двумя способами. Давайте рассмотрим каждый из них.

Первый способ – через Обозреватель объектов. На вашем сервере нажмите на папку «Базы данных» и в выпадающем списке выберите «Создать базу данных…» (см. Рис. 4).

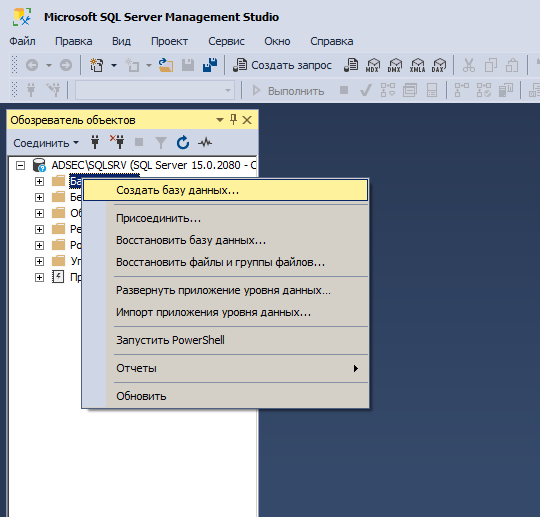


Рис. 4 «Создать базу данных…»

Вам откроется окно (см. Рис. 5). Для начала работы вам достаточно ввести имя базы данных и нажать кнопку «ОК». Но давайте рассмотрим, какие еще опции нам доступны при создании.

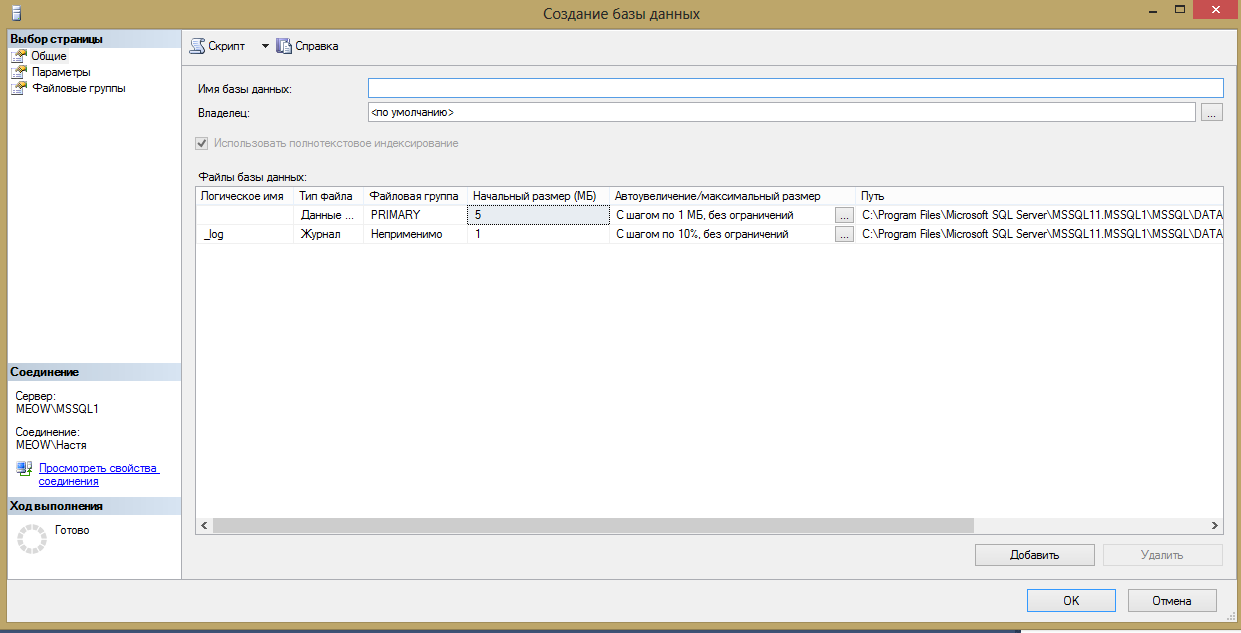


Рис. 5. Окно «Создание базы данных»

На первой вкладке мы можем задать начальный размер нашей БД, максимальный размер, а также шаг автоувеличения. По умолчанию они соответственно равны 5МБ и 1МБ.

На вкладке «Параметры» можно установить особые настройки. Такие как язык БД, уровень совместимости, параметры восстановления и другие.

После введения всех необходимых данных нажмите “ok”. Слева в обозревателе объектов появится ваша новая БД. Помимо БД создается sql-скрипт для ее создания. Выберите вашу БД, откройте контекстное меню. Далее Создать скрипт для БД -> Используя CREATE -> Новое окно редактора запросов.

Второй способ – с помощью SQL-скрипта.

Для этого на панели инструментов выберите команду «Создать запрос» (либо используйте сочетание клавиш Ctrl+N).

Команда для создания БД – CREATE DATABASE.

Синтаксис команды выглядит так:

CREATE DATABASE имя\_бд

ON

( NAME = логическое\_имя\_файла,

FILENAME = 'имя\_файа\_ос',

SIZE = размер, --пример 10

MAXSIZE = максимальный\_размер, --пример 30 или UNLIMITED

FILEGROWTH = шаг\_автоувеличения)

LOG ON --далее данные файла журнала

(NAME = логическое\_имя\_файла,

FILENAME = 'имя\_файа\_ос',

SIZE = размер, --пример 10

MAXSIZE = максимальный\_размер, --пример 30 или UNLIMITED

FILEGROWTH = шаг\_автоувеличения)

) ;

При создании БД можно не указывать дополнительные данные, а просто использовать команду CREATE DATABASE имя\_бд. В таком случае будет создана БД со стандартным размером и именами файлов.

После выполнения кода, ваша БД будет создана.

Как уже говорилось выше после подключения к серверу, если не было дополнительных установок, вы начинаете работу с системной БД master. Чтобы подключиться к другой БД, достаточно сделать двойной щелчок мыши по нужной БД. Если нет ограничений доступа, после этих действий вам будут доступны ее элементы в Обозревателе объектов.

Удалить базу данных можно тремя способами.

Через контекстное меню «Удалить».

Через sql-скрипт c помощью команды DROP DATABASE. Полный синтаксис выглядит так: DROP DATABASE имя\_БД.

Просто удалив файл БД.

Любая современная СУБД имеет систему резервного копирования и восстановления.

Резервное копирование (backup) базы данных и восстановление из резервной копии (restore) – два важнейших и наиболее частых процесса, осуществляемых администраторами баз данных.

Резервное копирование базы данных – единственный надежный способ предохранить данные от потери в результате поломки диска, сбоев электропитания, действий злоумышленников и ошибок в программах. В процессе резервного копирования создается независимый от платформы "снимок" базы данных, с помощью которого можно перенести данные на другую операционную систему или даже другую платформу.

Для создания резервной копии базы данных с помощью программы "SQL Server Management Studio" необходимо подключиться к базе данных, выбрать из контекстного меню базы данных Задачи-> Создать резервную копию. В открывшемся диалоговом окне "Мастер резервного копирования" задать несколько параметров и нажать кнопку Выполнить.

После выбора пути и файла для резервной копии в окне Back Up Database нажатием на OK запускаем процесс создания резервной копии. В случае успешной работы появится сообщение.

В результате будет создан файл с резервной копией. Стандартным расширением таких файлов для "SQL Server Management Studio" является "\*.bak". Файл с резервной копией базы данных обычно на порядок меньше оригинала.

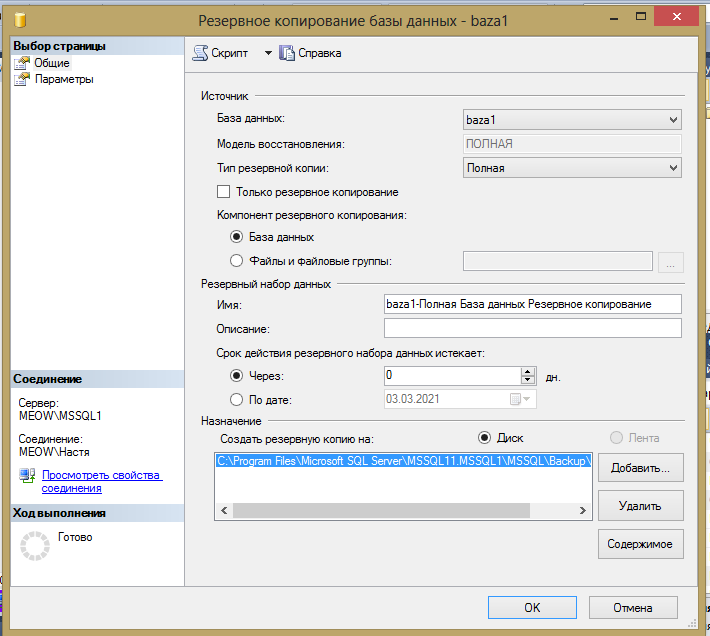


Рис. 6. Окно "Мастер резервного копирования"

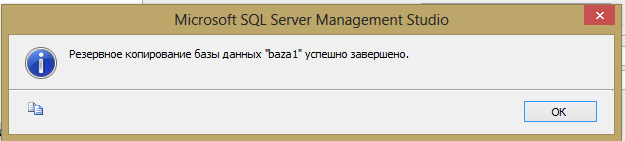


Рис. 7. Сообщение об успешном резервном копировании

Для восстановления базы данных из резервной копии используется команда База данных -> Восстановление базы данных. В результате откроется диалоговое окно "Мастер восстановления баз данных", в котором надо выбрать имя БД, место, куда будет восстанавливаться база данных, в которую будет помещен результат, способ восстановления, файл, из которого будет восстанавливаться база данных. Отмечаем выбранную резервную копию, и нажимаем кнопку Восстановить. Запускаем процесс восстановления. В случае успешного выполнения получим сообщение.

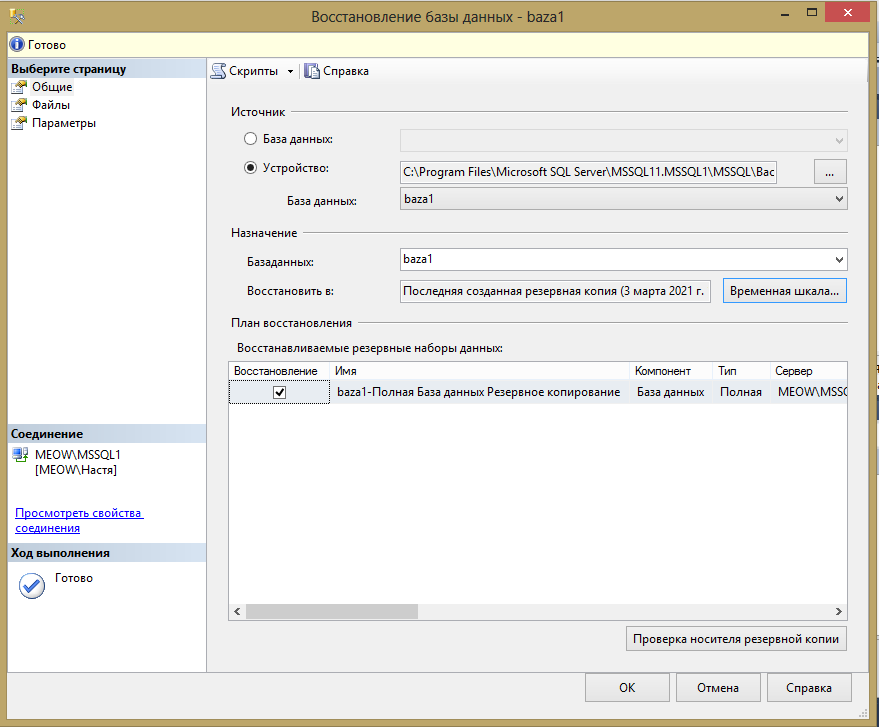


Рис. 8. Окно "Мастер восстановления баз данных"

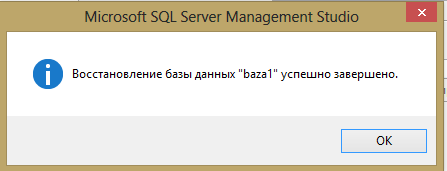


Рис. 9. Сообщение об успешном восстановлении

Задание на лабораторную работу:

1. Откройте SQL Server Management Studio.
2. Подключитесь к серверу.
3. Создайте базу данных с помощью контекстного меню с именем Ваша\_Фамилия\_Номер\_группы\_номер\_в\_журнале.
4. Извлеките команду для создания БД.
5. Удалите БД с помощью контекстного меню.
6. Создайте БД с помощью скрипта с именем ВашаФамилия\_ПредметнаяОбластьИзЛР1.
7. Создайте резервную копию этой БД.
8. Удалите БД с помощью SQL-кода.
9. Восстановите БД из файла с резервной копией.
10. Сохраните файл сценария.

Отчет должен содержать:

Титульный лист с номером и названием работы, фамилией студента и номером группы.

Цель работы.

Подписанные скриншоты выполненных заданий.

Коды всех выполненных команд.

Код из пункта 4.

**Лабораторная работа № 2**

**Импорт данных. Работа с csv и txt файлами**

Цель: Научиться правильно импортировать данные разных форматов, а также экспортировать данные.

Методические указания.

Данные в csv и txt файлах, хранятся в виде строк, где атрибуты разделены между собой запятыми. В MS SQL Server работать с такими файлами довольно просто, благодаря Мастеру импорта и экспорта. Однако, надо работать внимательно. Чтобы импортировать данные из csv или txt файла, зайдите в Мастер импорта и экспорта: нажмите правой кнопкой мыши на вашу БД в обозревателе объектов, затем выберите пункт «Задачи», а далее «Импорт

данных».

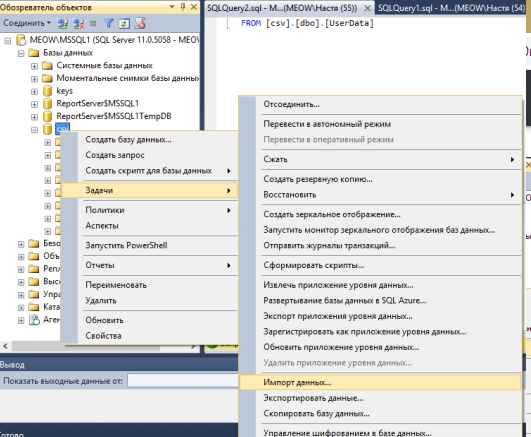


Рис. 10. Импорт данных

В качестве источника данных выберите «Неструктурированный файл» (при работе с SQL-сервер в англоязычной версии FLAT-файл). В обзоре файлов выберите необходимый файл.

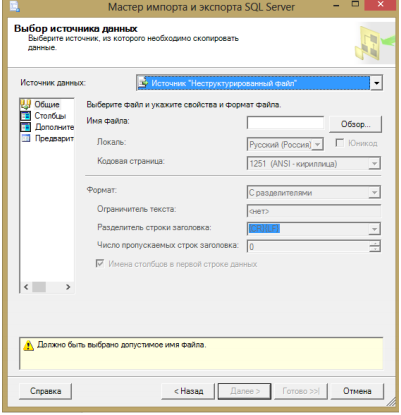


Рис. 11. Выбор источника данных

Если его нет в предложенных убедитесь, что указан нужный формат.



Рис. 12. Убедитесь, что видны все файлы, а не только txt

Теперь давайте посмотрим на доступные функции «Неструктурированного файла».

Локаль используется для интерпретации данных, зависящих от локаля, например, дат, а кодовая страница используется для конвертации строковых данных в формат Юникод.

Текстовый файл может иметь разделители, фиксированную ширину полей или смешанный формат.

• В формате с разделителями для выделения столбцов и строк используются разделители столбцов и строк.

• В формате с фиксированной шириной для определения столбцов и строк используются параметры ширины. Этот формат также включает символ для заполнения поля до его максимальной ширины.

• В формате без выравнивания справа для выделения всех столбцов, кроме последнего, используются параметры ширины; для последнего столбца используется разделитель строк.

Ограничитель текста необязательная характеристика. Например, можно указать, что текстовые поля должны быть заключены в кавычки.

Ниже приведены объяснения возможных разделителей строки заголовка:

{CR}{LF} В качестве разделителей для строки заголовка используются сочетания символов возврата каретки и перевода строки.

{CR} В качестве разделителей для строки заголовка используются символы возврата каретки.

{LF} В качестве разделителей для строки заголовка используются символы перевода строки.

Точка с запятой {;} В качестве разделителя для строки заголовка используется точка с запятой.

Двоеточие {:} В качестве разделителя для строки заголовка используется двоеточие.

Запятая {,} В качестве разделителя для строки заголовка используется запятая.

Табуляция {t} В качестве разделителя для строки заголовка используется символ табуляции.

Вертикальная черта {|} В качестве разделителя для строки заголовка используется вертикальная черта.

Количество строк заголовка или строк начальных данных – это данные, которые следует пропускать при необходимости. Не забудьте указать ожидать или задать имена столбцов в первой строке данных. Далее укажите разделители строк и столбцов. Вам будет доступен предварительный просмотр. Выберите его и посмотрите, как будет выглядеть ваше отношение после импорта.

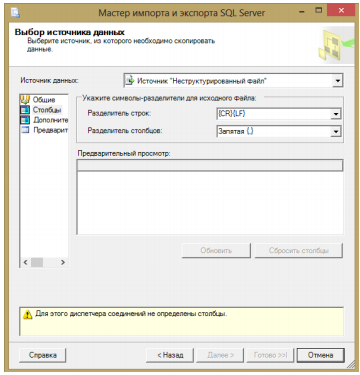


Рис. 12. Предварительный просмотр

Как правило, при импорте все данные будут иметь тип varchar. Это не очень удобно, например, если вы храните Даты, Цены или Числа. Конечно, это можно изменить уже при импорте.

Нажимайте далее, пока не дойдете до окна «Выбор исходных таблиц и представлений».

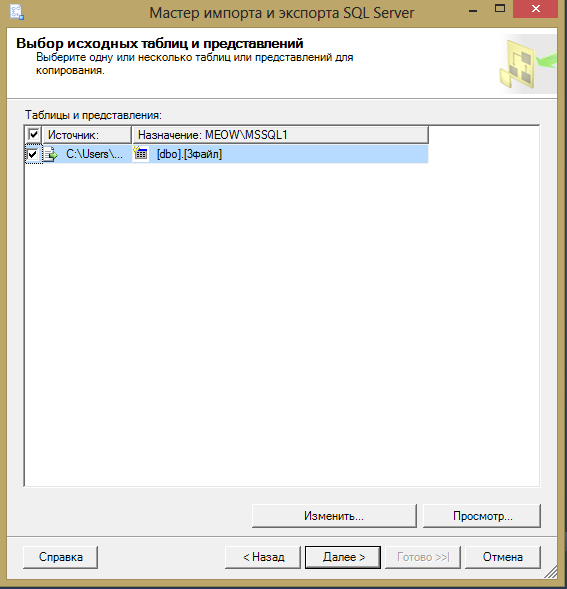


Рис. 13. «Выбор исходных таблиц и представлений»

Внизу окна есть кнопка «Изменить», после нажатия вам будет доступно окно настроек отношений. Вы сможете разрешить или изменить значение NULL, изменить тип или длину атрибута.

После завершения работы, выберите «Готово».

Задание на лабораторную работу:

1. Спросите у преподавателя, где находятся файлы для импорта.
2. Импортируйте каждый и них в вашу БД.
3. Следуя указаниям «мастера импорта и экспорта», экспортируйте любую таблицу в csv или txt файл.
4. Вставьте в отчет выбранные вами свойства и результат экспорта.

**Лабораторная работа №3.**

**Создание и заполнение таблиц**

Цель: Научиться создавать таблицы (отношения) и заполнять их. Получить навык создания БД с помощью даталогической модели.

Методические указания.

Невозможно представить себе базу данных без отношений. Создать таблицу можно двумя способами: с помощью конструктора или скрипта. Первый способ мы рассмотрим с вами позже, а в этой работе будем использовать скрипты.

Перед тем, как мы рассмотрим непосредственно само создание, давайте вспомним основные типы данных в MS SQL Server Management.

Числовые типы данных.

BIT: хранит значение 0 или 1. Фактически является аналогом булевого типа в языках программирования. Занимает 1 байт.

INT: хранит числа от –2 147 483 648 до 2 147 483 647. Занимает 4 байта. Наиболее используемый тип для хранения чисел.

DECIMAL: хранит числа c фиксированной точностью. Занимает от 5 до 17 байт в зависимости от количества чисел после запятой.

MONEY: хранит дробные значения от -922 337 203 685 477.5808 до 922 337 203 685 477.5807. Представляет денежные величины и занимает 8 байт. Эквивалентен типу DECIMAL(19,4).

FLOAT: хранит числа от –1.79E+308 до 1.79E+308. Занимает от 4 до 8 байт в зависимости от дробной части.

REAL: хранит числа от –340E+38 to 3.40E+38. Занимает 4 байта. Эквивалентен типу FLOAT(24).

Типы данных, представляющие дату и время:

DATE: хранит даты от 0001-01-01 (1 января 0001 года) до 9999-12-31 (31 декабря 9999 года). Занимает 3 байта.

TIME: хранит время в диапазоне от 00:00:00.0000000 до 23:59:59.9999999. Занимает от 3 до 5 байт.

Может иметь форму TIME(n), где n представляет количество цифр от 0 до 7 в дробной части секунд.

DATETIME: хранит даты и время от 01/01/1753 до 31/12/9999. Занимает 8 байт. Есть несколько способов записать дату, например: ‘гггг-мм-дд’ или ‘дд/мм/гггг ‘.

Время можно записывать, например так:’чч:мм’, ‘чч:мм:сс’.

Строковые типы данных.

CHAR: хранит строку длиной от 1 до 8 000 символов. На каждый символ выделяет по 1 байту. Не подходит для многих языков, так как хранит символы не в кодировке Unicode.

Количество символов, которое может хранить столбец, передается в скобках. Например, для столбца с типом CHAR(10) будет выделено 10 байт. И если мы сохраним в столбце строку менее 10 символов, то она будет дополнена пробелами.

VARCHAR: хранит строку. На каждый символ выделяется 1 байт. Можно указать конкретную длину для столбца - от 1 до 8 000 символов, например, VARCHAR(10). Если строка должна иметь больше 8000 символов, то задается размер MAX, а на хранение строки может выделяться до 2 Гб: VARCHAR(MAX).

Не подходит для многих языков, так как хранит символы не в кодировке Unicode.

В отличие от типа CHAR, если в столбец с типом VARCHAR(10) будет сохранена строка в 5 символов, то в столбце будет сохранено именно пять символов.

NCHAR: хранит строку в кодировке Unicode длиной от 1 до 4 000 символов. На каждый символ выделяется 2 байта. Например, NCHAR(15).

NVARCHAR: хранит строку в кодировке Unicode. На каждый символ выделяется 2 байта. Можно задать конкретный размер от 1 до 4 000 символов. Если строка должна иметь больше 4000 символов, то задается размер MAX, а на хранение строки может выделяться до 2 Гб.

Бинарные типы данных.

BINARY: хранит бинарные данные в виде последовательности от 1 до 8 000 байт.

VARBINARY: хранит бинарные данные в виде последовательности от 1 до 8 000 байт, либо до 2^31–1 байт при использовании значения MAX (VARBINARY(MAX)).

Теперь давайте вспомним об одном ограничении, которое можно накладывать на любой атрибут, а именно возможность оставлять его значения пустым. Как правило, запрет накладывается на атрибуты, которые являются первичными или внешними ключами. По умолчанию все атрибуты НЕОБЯЗАТЕЛЬНЫ к заполнению. А это свойство регулируется с помощью скрипта null/not null.

Вспомнив необходимые параметры, можем создать отношения.

В общем виде он выглядит так:

CREATE TABLE table\_name(

column1 datatype <NOT NULL>,

column2 datatype <NOT NULL>,

column3 datatype <NOT NULL>,

.....

columnN datatype <NOT NULL>)

Например, чтобы создать таблицу с названием Студенты, атрибутами: id, Имя, Фамилия, Адрес, причем id обязателен к заполнению, код будет следующим:

CREATE TABLE Студенты (id int NOT NULL, Имя nvarchar(15), Фамилия nvarchar(25),Адрес nvarchar(50));

Для заполнения таблиц воспользуемся построчным заполнением. Код для построчного заполнения выглядит так:

INSERT INTO TABLE\_NAME [(column1, column2, column3,...columnN)]

VALUES (value1, value2, value3,...,valueN);

Если вы хотите внести данные для всех атрибутов, нужно указывать их список после названия таблицы. Это особенно актуально, если вы вносите только некоторые данные. Например, только id и Фамилию. Давайте рассмотрим пример, для заполнения двух строк из предыдущего примера.

INSERT INTO Студенты

VALUES (1, ‘Карина’, ‘Петрова’,’пр. Художников 20’),

(2, ‘Игнат’, ‘Петров’,’пр. Художников 20’)

Если бы нам было необходимо внести только фамилию и id (без него никак, так как мы уже указали в настройках, что этот атрибут обязателен к заполнению), то код будет следующим:

INSERT INTO Студенты (id, Фамилия)

VALUES (3, ‘Бровина’)

Задание на лабораторную работу.

На основании даталогической модели из лабораторной работы создайте в вашей БД таблицы, а также заполните их минимум пятью строками.

**Лабораторная работа № 4.**

**Разработка ограничений**

Цель: Изучить используемые в SQL Server типы ограничений. Изучить SQL-операторы для работы с ограничениями.

Методические указания.

Ограничение – это очень важный объект БД. Это один из элементов, с помощью которого мы можем обеспечить целостность для нашей БД. Это механизм, обеспечивающий автоматический контроль соответствия данных установленным условиям, или ограничениям целостности.

Имеется пять ограничений целостности, различающихся по функциональности и области применения:

NULL - действует на уровне столбца и пользовательского типа данных и либо разрешает (NULL), либо запрещает (NOT NULL) хранение значений NULL.

CHECK - действует на уровне столбца и ограничивает диапазон значений, которые могут быть сохранены в столбце, путем проверки логического условия для вводимых данных. При вводе или изменении данных вводимое значение подставляется в условие. Если полученный результат TRUE, то изменения данных принимаются, иначе - отвергаются и генерируется сообщение об ошибке. Для одного столбца можно задать несколько ограничений типа CHECK.

UNIQUE - действует на уровне столбца и гарантирует уникальность вводимых значений. В отличии от ограничения PRIMARY KEY, это ограничение допускает хранение значений NULL.

PRIMARY KEY - действует на уровне столбца или таблицы и гарантирует уникальность в пределах таблицы первичного ключа, состоящего из одного или нескольких столбцов. Ни для одного из столбцов ключа не должно быть установлено свойство NULL. Когда используется один столбец, то для него необходимо также задать и свойство UNIQUE. В таблице создается только один первичный ключ.

FOREIGN KEY - действует на уровне таблицы и связывается с одним из кандидатов на первичный ключ в другой таблице. Таблица, в которой определен внешний ключ с помощью этого ограничения, называется зависимой, а таблица с кандидатом на первичный ключ - главной. В зависимую таблицу нельзя вставить строку, если внешний ключ не имеет соответствующего значения в главной таблице. Из главной таблицы нельзя удалить строку, если с ней связана хотя бы одна строка в зависимой таблице.

Формат задания ограничения таков: FOREIGN KEY REFERENCES имя главной таблицы (кандидат на первичный ключ или ее ключ).

В конце используется

ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }

ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }

Эти строки указывают, какое действие совершается над строками в изменяемой таблице, когда эти строки имеют ссылочную связь и строка родительской таблицы, на которую указывает ссылка, обновляется. Параметр по умолчанию — NO ACTION.

NO ACTION Компонент Компонент Database Engine возвращает ошибку, а обновление строки родительской таблицы откатывается.

CASCADE Соответствующие строки обновляются в ссылающейся таблице, если эта строка обновляется в родительской таблице.

SET NULL Всем значениям, составляющим внешний ключ, присваивается значение NULL, когда обновляется соответствующая строка в родительской таблице. Для выполнения этого ограничения внешние ключевые столбцы должны допускать значения NULL.

SET DEFAULT Всем значениям, составляющим внешний ключ, присваивается их значение по умолчанию, когда обновляется соответствующая строка в родительской таблице.

Задание авто инкремента осуществляется следующим образом

client\_id INT IDENTITY (1,1)

Ограничения можно добавлять во время создания отношения или добавить к уже созданным. Рассмотрим общий синтаксис и примеры для каждого из этих случаев.

CREATE TABLE table\_name(

column1 datatype <not null> constraint type,

column2 datatype <not null> constraint type,

column3 datatype <not null> constraint type,

.....

columnN datatype <not null> constraint type)

Например, чтобы создать таблицу с названием Студенты, атрибутами: id, Имя, Фамилия, Адрес, причем id обязателен к заполнению, а также является первичным ключом, код будет следующим:

CREATE TABLE Студенты ( id int NOT NULL primary key,

Имя nvarchar(15),

Фамилия nvarchar(25),

Адрес nvarchar(50))

Чтобы создать таблицу Успеваемость с атрибутами id и средний балл, где id – внешний ключ, связывающий эту таблицу с таблицей Студенты, следует использовать следующий код.

CREATE TABLE Успеваемость ( id int NOT NULL foreign key references Студенты(id) on delete cascade on update cascade,

[Средний балл] float)

Теперь рассмотрим пример создания таблицы Пациенты с атрибутами id, номер медкнижки, ФИО и номером телефона. Причем id – первичный ключ, номер медкнижки – уникальные значения, а номер телефона имеет ограничение ввода данных типа +7(000)000-00-00, где 0 – любая цифра.

CREATE TABLE Пациенты( id int NOT NULL primary key,

[номер медкнижки] int unique,

ФИО nvarchar(50),

[номер телефона] char(16) check ([номер телефона]

like ‘+7([0-9][0-9][0-9])[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9]’))

Далее давайте рассмотрим примеры, когда нам необходимо добавить ограничения к уже созданным таблицам.

Любой такой запрос будет начинаться со слов ALTER TABLE tablename. Позже мы будем отдельно рассматривать, как изменять те или иные объекты в БД. Далее мы пишем ADD CONSTRAINT и тип ограничения.

Приведу код на добавление таких же ограничений, как и в предыдущих примерах, но, если бы мы их добавляли к уже существующим таблицам.

Добавление первичного ключа в таблице Студенты:

ALTER TABLE Студенты.

ADD CONSTRAINT C1 –-добавляем ограничение с именем с1.

PRIMARY KEY (id)—-тип ограничение первичный ключ, накладываем на атрибут id.

Добавление внешнего ключа в таблице Успеваемость:

ALTER TABLE Успеваемость

ADD CONSTRAINT C2 –-добавляем ограничение с именем с2

FOREIGN KEY (id)—-тип ограничение внешнего ключа, накладываем на атрибут id

REFERENCES Студенты (id) –-соединяем с атрибутом id таблицы Студенты

Добавление ограничения уникальности и ввода данных в таблице Пациенты

ALTER TABLE Пациенты

ADD CONSTRAINT C3 –-добавляем ограничение с именем с3

UNIQUE ([номер медкнижки])—-накладываем на атрибут номер медкнижки

ALTER TABLE Пациенты

ADD CONSTRAINT C4 –-добавляем ограничение с именем с4

CHECK ([номер телефона])

like ‘+7([0-9][0-9][0-9])[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9]’)

Задание на лабораторную работу.

На основании даталогической и инфологической моделей, создайте в вашей БД связи между отношениями и другие ограничения.

Создайте таблицу Client, в которой будут храниться данные о клиентах компании. В этом отношении спроектируйте следующие атрибуты и ограничения для них. Id – автоинкримент и первичный ключ, имя, фамилия и отчество имеют только ограничения по длине, возраст - от 14 до 99 лет, номер телефона - начинается с ‘+7’ и содержит еще 10 цифр после этого.

**Лабораторная работа № 5**

**Работа с внешними ключами**

Цель: Изучить синтаксис Foreign key и понятие ссылочной целостности. Изучить разницу между разными параметрами ключевых слов on delete on update.

Методические указания.

Ссылочная целостность—это состояние реляционной базы данных, в которой записи не могут ссылаться на несуществующие записи в этой базе данных.

FOREIGN KEY—особый вид ограничения, которое позволяет предотвратить нарушение ссылочной целостности при удалении/изменении информации в таблицах предках.

Пример нарушения ссылочной целостности:

Пусть существуют две таблицы: Catalogs, являющаяся таблицей-предком, содержащая в себе упоминания о категориях товаров в интернет магазине, и таблица Products, являющаяся таблицей-потомком, со всеми товарами этого магазина.

|  |  |
| --- | --- |
| id\_catalog | name |
| 1 | Процессоры |
| 2 | Материнские платы |
| 3 | Видеоадаптеры |
| 4 | Жёсткие диски |
| 5 | Оперативная память |

Таблица Catalogs

|  |  |  |
| --- | --- | --- |
| Id\_product | Name | Id\_catalog |
| 1 | Celeron 1.8 | 1 |
| 2 | Celeron 2.0GHz | 1 |
| 3 | Celeron 2.4GHz | 1 |
| 4 | Celeron D 320 2.4GHz | 1 |
| 5 | Celeron D 325 2.53GHz | 1 |
| 6 | Celeron D 315 2.26GHz | 1 |
| 7 | Intel Pentium 4 3.2GHz | 1 |
| 8 | Intel Pentium 4 3.0GHz | 1 |
| 9 | Intel Pentium 4 3.0GHz | 1 |
| 10 | Gigabyte GA-8I848P-RS | 2 |
| 11 | Gigabyte GA-8IG1000 | 2 |
| 12 | Gigabyte GA-8IPE1000G | 2 |
| 13 | Asustek P4C800-E Delux | 2 |
| 14 | Asustek P4P800-VM\L i865G | 2 |
| 15 | Epox EP-4PDA3I | 2 |
| 16 | ASUSTEK A9600XT/TD | 3 |
| 17 | ASUSTEK V9520X | 3 |
| 18 | SAPPHIRE 256MB RADEON 9550 | 3 |
| 19 | GIGABYTE AGP GV-N59X128D | 3 |
| 20 | Maxtor 6Y120P0 | 4 |
| 21 | Maxtor 6B200P0 | 4 |
| 22 | Samsung SP0812C | 4 |
| 23 | Seagate Barracuda ST3160023A | 4 |
| 24 | Seagate ST3120026A | 4 |
| 25 | DDR-400 256MB Kingston | 5 |
| 26 | DDR-400 256MB Hynix Original | 5 |
| 27 | DDR-400 256MB PQI | 5 |
| 28 | DDR-400 512MB Kingston | 5 |
| 29 | DDR-400 512MB PQI | 5 |
| 30 | DDR-400 512MB Hynix | 5 |

Таблица Products

При удалении категории из таблицы Сatalogs, в таблице Products останутся товары, которые не привязаны ни к одной из категорий, что может повлечь массу проблем для магазина. В результате следующего запроса получим отношения на рис. 1 и 2.

DELETE \* FROM Catalogs WHERE name = ‘Процессоры’

SELECT \* FROM Catalogs

SELECT \* FROM Products WHERE id\_catalog = 1

|  |  |
| --- | --- |
| id\_catalog | name |
| 2 | Материнские платы |
| 3 | Видеоадаптеры |
| 4 | Жёсткие диски |
| 5 | Оперативная память |

Рис.1 Результаты запроса

|  |  |  |
| --- | --- | --- |
| Id\_product | Name | Id\_catalog |
| 1 | Celeron 1.8 | 1 |
| 2 | Celeron 2.0GHz | 1 |
| 3 | Celeron 2.4GHz | 1 |
| 4 | Celeron D 320 2.4GHz | 1 |
| 5 | Celeron D 325 2.53GHz | 1 |
| 6 | Celeron D 315 2.26GHz | 1 |
| 7 | Intel Pentium 4 3.2GHz | 1 |
| 8 | Intel Pentium 4 3.0GHz | 1 |
| 9 | Intel Pentium 4 3.0GHz | 1 |

Рис. 2. Результаты запроса

Это явление называется нарушением ссылочной целостности.

На ссылочную целостность базы данных как правило, оказывают четыре типа изменений.

Добавление новой записи в таблице-потомке. Например, добавление новой товарной позиции в таблицу Products. Следует заметить, что важную роль играет изменение именно таблицы-потомка, т.к. изменение таблицы-предка (Catalogs) не приведет к нарушению ссылочной целостности, наличие пустой категории товаров допустимо.

Обновление внешнего ключа в таблице-потомке. Эта ситуация похожа на первую и может произойти при изменении у товара ссылки на несуществующий раздел каталога, например, товар с id\_catalog равным 50.

Удаление записи из таблицы-предка. Эта ситуация рассмотрена выше.

Изменение записи в таблице-предке. Эта ситуация отличается от рассмотренной выше тем, что категория каталога не удаляется, а принимает новый id.

Для того, чтобы контролировать ссылочную целостность в базе данных необходимо, чтобы таблицы были связаны при помощи конструкции FOREIGN KEY, которая имеет вид:

FOREIGN KEY [index\_name] (index\_col\_name, …)

REFERENCES tbl\_name (index\_col\_name,…)

[ON DELETE {CASCADE | SET NULL | NO ACTION | SET DEFAULT}]

[ON UPDATE {CASCADE | SET NULL | NO ACTION | SET DEFAULT}]

FOREIGN KEY — используется при создании/изменении таблиц-потомков. В данном примере FOREIGN KEY следует использовать в таблице Products. Данная конструкция позволяет задать в таблице-потомке внешний ключ с именем index\_name на столбцах таблицы, которые перечисляется в круглых скобках. Можно использовать один или несколько столбцов.

Ключевое слово REFERENCES задаёт таблицу-предка tbl\_name на которую будет ссылаться внешний ключ. Поля таблицы-предка задаются в круглых скобках, один или несколько.

Необязательные конструкции ON DELETE и ON UPDATE, определяют поведение SQL при удалении/обновлении записей из таблицы-предка.

Допустимые параметры для ключевых слов ON DELETE и ON UPDATE:

CASCADE — При удалении/обновлении записей в таблице-предке, будут также обновлены/удалены записи из таблицы-потомка с существующим первичным ключом.

SET NULL — При удалении/обновлении записей в таблице-предке, записи из таблицы потомка с существующим первичным ключом будут обновлены на NULL.

NO ACTION — При удалении/обновлении записей в таблице-предке, записи из таблицы-потомка с существующим первичным ключом изменены не будут.

Добавление для таблицы Products из примера:

ALTER TABLE Products ADD CONSTRAINT fk\_catalog

FOREIGN KEY (id\_catalog) REFERENCES Catalogs (id\_catalog)

ON DELETE CASCADE

ON UPDATE CASCADE

Это приведет к тому, что изменения в таблице Catalogs приведут к автоматическому изменению таблицы Products.

Задание на лабораторную работу.

1. Создать четыре отношения и заполнить их любым удобным способом.
2. Должны получиться следующие отношения:

Таблица поставщиков (Shippers)

|  |  |  |  |
| --- | --- | --- | --- |
| Номер поставщика | Фамилия | Рейтинг | Город |
| 1 | Смит | 20 | Лондон |
| 2 | Джонс | 10 | Париж |
| 3 | Блейк | 30 | Париж |
| 4 | Кларк | 20 | Лондон |
| 5 | Адамс | 30 | Афины |

Таблица деталей (Details)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Номер детали | Название | Цвет | Вес | Город |
| 1 | Гайка | Красный | 12 | Лондон |
| 2 | Болт | Зеленый | 17 | Париж |
| 3 | Винт | Голубой | 17 | Рим |
| 4 | Винт | Красный | 14 | Лондон |
| 5 | Кулачок | Голубой | 12 | Париж |
| 6 | Блюм | Красный | 19 | Лондон |

Таблица изделий (Products)

|  |  |  |
| --- | --- | --- |
| Номер изделия | Название | Город |
| 1 | Жесткий диск | Париж |
| 2 | Перфоратор | Рим |
| 3 | Считыватель | Афины |
| 4 | Принтер | Афины |
| 5 | Флоппи-диск | Лондон |
| 6 | Терминал | Осло |
| 7 | Лента | Лондон |

Таблица поставок (Supplies)

|  |  |  |  |
| --- | --- | --- | --- |
| Номер поставщика | Номер детали | Номер изделия | Количество |
| 1 | 1 | 1 | 200 |
| 1 | 1 | 4 | 700 |
| 2 | 3 | 1 | 400 |
| 2 | 3 | 2 | 200 |
| 2 | 3 | 3 | 200 |
| 2 | 3 | 4 | 500 |
| 2 | 3 | 5 | 600 |
| 2 | 3 | 6 | 400 |
| 2 | 3 | 7 | 800 |
| 2 | 5 | 2 | 100 |
| 3 | 3 | 1 | 200 |
| 3 | 4 | 2 | 500 |
| 4 | 6 | 3 | 300 |
| 4 | 6 | 7 | 300 |
| 5 | 2 | 2 | 200 |
| 5 | 2 | 4 | 100 |
| 5 | 5 | 5 | 500 |
| 5 | 5 | 7 | 100 |
| 5 | 6 | 2 | 200 |
| 5 | 1 | 4 | 100 |
| 5 | 3 | 4 | 200 |
| 5 | 4 | 4 | 800 |
| 5 | 5 | 4 | 400 |
| 5 | 6 | 4 | 500 |

1. Установить внешний ключ в последней таблице на соответствующие атрибуты.

Для первой таблицы создать каскадное удаление и обновление, для второй – установку значений null в удаленных строках, а для третьей установить отсутствие действия.

4. Удалить из таблиц Shippers, Details, Products строки с первым номером, вывести таблицу Supplies на экран. Проанализировать изменения.

5. Сделать вывод, который будет содержать ответ на вопрос: «Как вам кажется, какой вариант параметров on delete on update целесообразнее было бы использовать для всех таблиц в этой работе?»

Контрольные вопросы:

1. Какие типы ограничений вы знаете? Какие у них назначения?
2. Зачем нужны ограничения?
3. Можно ли обойтись без ограничений в БД?
4. Существуют ли какие-либо ограничения в таблице по умолчанию?
5. Какие условия необходимо соблюсти, чтобы связать две таблицы связью
6. Какие типы ключей вы знаете?
7. При связи двух таблиц, накладываются ли какие-либо ограничения на дочернюю или родительскую таблицу?
8. Зачем нужны ключевые слова on update on delete?
9. Какие параметры on update on delete существуют?
10. Какой синтаксис у ограничения типа CHECK?

**Лабораторная работа № 6**

**Выборка данных. Выбор строк по условию**

Цель: Изучить запросы на выборку к одной таблице, научиться использовать запросы, сочетание нескольких функций, операторов, кодовых слов.

Методические указания.

В SQL имеется единственный оператор, который предназначен для выборки данных из базы данных – SELECT. Он относится к подмножеству DML.

Ниже приведен почти полный синтаксис оператора SELECT.

SELECT [DISTINCT | ALL]

FROM Имя\_таблицы

[WHERE <условие поиска>]

[GROUP BY Колонка [, Колонка ...]]

[HAVING <условие поиска>]

[ORDER BY <список сортировки>];

После слова SELECT следует указать атрибут или список атрибутов, которые необходимо вывести на экран, а далее после слова FROM название таблицы или нескольких таблиц из которой мы выбираем атрибуты.

Пример. Пусть у нас есть таблица Students с атрибутами id, Name, Surname, Age. Чтобы вывести на экран только имена и фамилии студентов, следует использовать следующий код.

SELECT Name, Surname FROM Students;

Чтобы вывести все столбцы отношения вместо перечисления всех атрибутов можно использовать метасимвол \*.

Запрос SELECT \* FROM Students; выведет все строки и все столбцы отношения Students на экран.

В этой работе мы будем использовать только условия. За условие отбора строк отвечает ключевое слово WHERE. Ниже приведем основные группы операторов, ключевых слов и функций, с помощью которых мы можем фильтровать информацию.

Операторы сравнения.

Особенностью операторов сравнения является то, что независимо от типов операндов их результатом являются логические значения. К ним можно отнести «=», «>», «<». Предположим, у нас есть отношение Сотрудники с атрибутами код, ФИО, должность, стаж. Если мы хотим вывести только строки с сотрудниками, которые занимают должность продавца, то следует использовать следующий код:

SELECT \* FROM Сотрудники

WHERE должность= ‘Продавец’;

А если бы мы хотели вывести только строки с сотрудниками, чей стаж больше 7 лет, то можно использовать следующий код:

SELECT \* FROM Сотрудники

WHERE стаж>7;

Логические операторы.

Операндами и результатом логических операторов являются логические значения. Стандартными логическими операторами являются AND, OR и NOT. Использование операторов сравнения вместе с логическими операторами предоставляет возможность формулировать составные условия для отбора строк таблиц.

Использование логического оператора AND.

Рассмотрим несколько примеров с использованием этого оператора.

Чтобы вывести на экран всех продавцов со стажем больше 2 лет из предыдущего примера, следует использовать следующий код.

SELECT \* FROM Сотрудники

WHERE стаж>2 AND должность= ‘Продавец’;

Логический оператор NOT можно пояснить словами ‘не’ и ‘кроме’.

Например, если мы хотим вывести на экран всех сотрудников, кроме продавцов, необходимо написать следующий код:

SELECT \* FROM Сотрудники

WHERE NOT должность= ‘Продавец’;

Выражения над столбцами.

В выражениях фразы WHERE в качестве значения аргументов могут быть не только имена столбцов, а также выражения над столбцами.

Например, пусть у нас есть таблица ЗП, в которой есть следующие атрибуты ID\_сотрудника, зп, премия.

Чтобы вывести все записи с сотрудниками, у которых ЗП и премия больше 40000, используем следующий запрос:

SELECT \* FROM ЗП

WHERE ЗП+премия>40000;

Проверка на принадлежность множеству

В левой части оператора IN вместо имени столбца можно использовать любое допустимое над столбцами таблицы выражение языка. Синтаксис оператора следующий:

Имя\_Атрибута IN (значение1, …, значениеN)

Например, выведем все строки из таблицы Сотрудники, со стажем 3,4 или 5 лет.

SELECT \* FROM Сотрудники

WHERE стаж IN (3,4,5);

Также оператор IN можно сочетать с отрицанием NOT или с выражениями над множествами.

Проверка на принадлежность диапазону значений.

Еще одной формой проверки вхождения элемента во множество является проверка на его принадлежность диапазону значений. Для этого применяется предикат BETWEEN, который определяет нахождение значения столбца между указанными минимальным и максимальным значениями. Синтаксис следующий:

Имя\_Атрибута BETWEEN минимум AND максимум.

Например, выведем всех сотрудников, со стажем от 2 до 10 лет.

SELECT \* FROM Сотрудники

WHERE стаж BETWEEN 2 AND 10;

Проверка на соответствие шаблону.

Оператор LIKE сравнивает значение столбца со множеством значений, определяемых шаблоном. Он представляет собой строку, в которой помимо обычных символов, составляющих основу поискового выражения, можно использовать так называемые подстановочные символы (иногда они называются групповыми символами). Имеется всего два подстановочных символа, различающихся тем, что именно на их месте может стоять: % — любая последовательность символов, включая их отсутствие; \_ — один любой символ. Подстановочные символы могут находиться в любом месте шаблона в любом наборе. Например, шаблону '%Иван%' соответствуют строки 'Иван', 'Иванов', 'Иванченко', 'Петр Иванович', а шаблону 'л\_с\_' - 'лист', 'леса', 'лоск' (ноне 'лес', 'листок', 'плес'). Общий синтаксис для использования шаблона выглядит так:

Имя\_Атрибута [NOT] LIKE ‘шаблон’.

Проверка на неопределенное значение.

Наличие значения NULL во фразе WHERE приводит к тому, что условие принимает истинностное значение UNKNOWN и соответствующая строка не включается в результат. Чтобы проверить столбец на неопределенное значение, следует применить унарный оператор IS NULL, имеющий такой синтаксис:

Имя\_Атрибута IS [NOT] NULL

Агрегатные функции – встроенные функции T-SQL, обрабатывающие коллекцию значений и возвращающее одно значение. К таким функциям относятся, например, нахождение минимального и максимального числа. Обрабатывая некоторый объем данных, мы получаем лишь одно значение на выходе.

Рассмотрим основные агрегатные функции и их синтаксис.

COUNT (выражение) – возвращает количество значений в группе.

Имеет место следующая запись COUNT (\*), которая позволяет подсчитать количество записей в отношении. Также может использоваться с ключевым словом DISTINCT, которое позволяет посчитать только уникальные значения.

Пример. Пусть у нас есть таблица Guests в базе данных отеля с данными о постояльцах, когда-либо останавливающихся в отеле. Отношение содержит следующие атрибуты: ФИО гостя, даты пребывания, страну и электронную почту.

|  |  |  |  |
| --- | --- | --- | --- |
| ФИО | Дата пребывания | Страна | E-mail |
| Арман Руссо | 17/11/2019-21/11/2019 | Франция | Russeau1986@gmail.com |
| Юджин Земски | 26/08/2020-3/09/2020 | США | Zemskijjj@yahoo.com |
| … | … | … | … |

Рис. 17 Таблица Guests

Для того, чтобы узнать сколько гостей посетило наш отель, мы можем использовать данный код:

SELECT COUNT(\*) FROM Guests;

Но, если мы хотим узнать из скольких разных стран гости приезжали в наш отель, то можно использовать следующий код:

SELECT COUNT DISTINCT (Страна) FROM Guests;

MAX, MIN (выражение) – возвращает максимальное и минимальное значение соответственно.

Синтаксис:

Select Max(имя\_атрибута) from Имя\_таблицы

AVG (выражение) – возвращает среднее арифметическое значение в группе.

Синтаксис:

Select Avg(имя\_атрибута) from Имя\_таблицы

SUM (выражение) – возвращает сумму всех значений в группе.

Синтаксис:

Select Sum(имя\_атрибута) from Имя\_таблицы

Все агрегатные функции могут использоваться в сочетании с оператором where, если вам нужна фильтрация.

Задание на лабораторную работу.

1. Импортируйте данные из файла, в соответствии с вариантом.

2. Для своего отношения создайте следующие запросы:

1. три запроса с использованием оператора сравнения;
2. три запроса с использованием логических операторов and, or, not;
3. один запрос с комбинацией логических операторов;
4. два запроса на использование выражений над столбцами;
5. три запроса с использованием агрегатных функций;
6. один запрос с проверкой на принадлежность множеству;
7. два запроса с проверкой на принадлежность диапазону значений;
8. два запроса с проверкой на соответствие шаблону (разные);
9. один запрос с проверкой на неопределенное значение.

Отчет должен содержать код запроса и результат выборки.

**Лабораторная работа № 7**

**Выборка данных. Многотабличные запросы, запросы на соединения**

Цель: Изучить многотабличные запросы на выборку, а также получить навыки работы с оператором SELECT.

Методические указания.

В информационной системе сущности могут быть взаимосвязанными. Связь между таблицами устанавливается за счет размещения специального столбца первичного ключа одной таблицы, которая называется родительской, в другой таблице, которая называется дочерней. Наличие внешних ключей является основой для инициирования поиска по многим таблицам.

Одна из наиболее важных особенностей предложения SELECT — это способность использования связей между различными таблицами, а также вывода содержащейся в них информации.

Декартово произведение таблиц.

Соединение таблиц - это частный случай операции декартового произведения (или просто произведения). Декартово произведение двух таблиц — это таблица, состоящая из всех возможных пар строк обеих таблиц. Это определение можно естественным образом расширить на любое количество таблиц. В SQL декартово произведение выражается с помощью следующего синтаксиса:

SELECT \* FROM имя\_таблицы1, …, имя\_таблицыN;

Вместо знака \* мы можем использовать имена интересующих нас атрибутов.

Пример. Пусть у нас есть две таблицы: Студенты и Курсы. В таблице Студенты хранится информация о потенциальных посетителях курсов, у нее следующие атрибуты: код, ФИО, телефон. В таблице Курсы список возможных для посещения курсов, у нее следующие атрибуты: Код, Название, количество часов.

Первые несколько строк каждой из таблиц см на рис. 18 и рис. 19.

|  |  |  |
| --- | --- | --- |
| Код | ФИО | Телефон |
| 1 | Иванов Иван Иванович | +79001112233 |
| 2 | Федоров Алексей Иванович | +79013334235 |
| … | … | … |

Рис. 18 Таб. Студенты

|  |  |  |
| --- | --- | --- |
| Код | Название | Количество часов |
| 1 | Программирование для начинающих | 70 |
| 2 | Английский язык | 115 |
| … | … | … |

Рис. 19 Таб. Курсы

Тогда запрос SELECT \* FROM Студенты, Курсы; даст нам следующее отношение:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Код | ФИО | Телефон | Код | Название | Количество часов |
| 1 | Иванов Иван Иванович | +79001112233 | 1 | Программирование для начинающих | 70 |
| 1 | Иванов Иван Иванович | +79001112233 | 2 | Английский язык | 115 |
| 2 | Федоров Алексей Иванович | +79013334235 | 1 | Программирование для начинающих | 70 |
| 2 | Федоров Алексей Иванович | +79013334235 | 2 | Английский язык | 115 |
|  |  |  |  |  |  |

Как мы видим, такие атрибуты как Код теряют свой смысл, к тому же имена данных атрибутов дублируются, что усложняет работу с отношением. Вместо \* мы также можем указать конкретные столбцы. В результате запроса

SELECT Студенты.ФИО, Студенты.Телефон, Курсы.Название FROM Студенты, Курсы;

Получим:

|  |  |  |
| --- | --- | --- |
| ФИО | Телефон | Название |
| Иванов Иван Иванович | +79001112233 | Программирование для начинающих |
| Иванов Иван Иванович | +79001112233 | Английский язык |
| Федоров Алексей Иванович | +79013334235 | Программирование для начинающих |
| Федоров Алексей Иванович | +79013334235 | Английский язык |
| … | … | … |

Обратите внимание, что мы указываем путь до атрибута через таблицу, разделяя их имена точкой.

Условие соединения.

Соединение таблиц может быть указано во фразе WHERE или во фразе FROM. Сначала рассмотрим первый вариант. Большинство запросов, имеющих несколько таблиц во фразе FROM, содержат фразу WHERE, в которой указаны условия, попарно сравнивающие столбцы из различных таблиц. Такое условие называется условием соединения. В этом случае SQL предполагает сцепление только тех пар строк из разных таблиц, для которых условие соединения принимает истинное значение.

Теоретически при соединении сначала выполняется декартово произведение указанных таблиц в одну, а затем из нее отбираются строки согласно условию соединения. Естественно, ни одна СУБД не работает таким образом. Фраза WHERE помимо условия соединения может также содержать другие условия, каждое из которых ссылается на столбцы соединенной таблицы. Эти условия производят отбор строк соединенной таблицы.

Соединение, в котором после WHERE указаны условия соответствия строк из отношений, называется соединением по равенству.

Пусть у нас есть таблицы Кафедры и Факультеты. В таблице Кафедры есть атрибут код\_факультета, который является внешним ключом и определяет к какому факультету принадлежит кафедра. Синтаксис для вывода полной информации из обеих таблиц будет следующий:

SELECT \* FROM Кафедры, Факультеты

WHERE Факультеты.Код=Кафедры.код\_факультета;

Данный пример более реалистичный и осознанный. Здесь мы выводим не все возможные комбинации, а учитываем связь между кафедрами и факультетами.

Также в WHERE мы можем указывать и другие условия для фильтра. Например, вывести только те строки, где код факультета равен 3.

SELECT \* FROM Кафедры, Факультеты

WHERE Факультеты.Код=Кафедры.код\_факультета

AND Факультеты.Код=3;

Используя FROM следует использовать один из видов соединения, левое внешнее, правое внешнее, естественное соединение или полное внешнее.

Сформулируем общую процедуру составления многотабличного запроса и приведем пример ее использования.

1. Определить множество таблиц, необходимых для ответа на запрос. В это множество должны входить таблицы, на столбцах которых сформулированы условия, а также те, столбцы которых необходимо вывести. Это так называемые базовые таблицы запроса.

2. В структуре взаимосвязанных таблиц найти путь, соединяющий базовые таблицы. Это так называемый путь вычисления запроса. В результате вы получите перечень таблиц, необходимых для формулировки запроса. Это так называемые таблицы запроса.

3. Во фразе FROM перечислить необходимые таблицы.

4. Во фразе WHERE соединить таблицы запроса и при необходимости задать условия отбора строк в базовых таблицах запроса.

5. Во фразе SELECT перечислить выводимые столбцы.

Самосоединение таблицы.

Как правило, взаимосвязи существуют и в пределах одной таблицы. В одних случаях эти связи являются явными, например, когда внешний ключ ссылается на первичный ключ той же самой таблицы. В других случаях эта связь присутствует неявно, например, кафедры могут быть связаны между собой на основании того свойства, что располагаются в одном корпусе. Для ответа на такие запросы следует осуществлять соединение таблицы со своей копией. Такое соединение иногда называют самосоединением таблицы. Несмотря на кажущуюся искусственность идеи самосоединения таблиц, существует множество запросов, которые требуют именно такого соединения. На приводимых далее примерах вы убедитесь в этом.

Чтобы произвести соединение таблицы со своей копией, необходимо указать во фразе FROM имя одной и той же таблицы два или большее количество раз, а во фразе WHERE — условие их самосоединения.

Однако в этом случае возникает следующая проблема — как ссылаться на столбцы различных копий таблицы. До сих пор проблема ссылки на столбцы с одинаковыми именами из разных таблиц разрешалась уточнением имени столбца именем таблицы. В нашем же случае соединяемые таблицы имеют одинаковые имена. Для разрешения этой проблемы без синонимов таблиц уже не обойтись. В нашем случае различным вхождениям одной и той же таблицы приписываются различные синонимы и именно по этим синонимам производится обращение к столбцам. Приведем примеры использования самосоединения.

Пример. Вывести фамилии преподавателей, зарплата которых больше, чем у преподавателя Сидорова.

SELECT needed.NAME\_TEACHER

FROM TEACHER needed, TEACHER given

WHERE needed.Salary + needed.Rise > given.Salary + given.Rise

AND given.NAME\_TEACHER = 'Сидоров А.О.';

Самосоединение можно использовать для проверки корректности данных. Например, мы точно знаем, что в нашем вузе нет однофамильцев, занимающих разные должности. С помощью самосоединения таблицы преподавателей мы можем убедиться, что их нет и в базе данных.

Пример. Указать преподавателей-однофамильцев, которые занимают различные должности.

SELECT tch1.NAME\_TEACHER AS ‘Препод. с различ. должностями’

FROM TEACHER tch1, TEACHER tch2

WHERE tch1.NAME\_TEACHER = tch2.NAME\_TEACHER

AND tch1.DOLGNOST <> tch2.DOLGNOST;

Внешнее соединение таблиц.

Внешнее соединение возвращает строки, которые удовлетворяют условию соединения, а также те строки одной из таблиц, для которых в другой не нашлось удовлетворяющих условию соединения строк. Внутренние соединения возвращают результат, когда в обеих таблицах есть хотя бы одна строка, соответствующая условиям соединения.

Левое внешнее соединение.

Общий синтаксис выглядит так:

SELECT имя\_столбца1, …, имя\_столбцаN FROM

Имя\_таблицы1 LEFT JOIN Имя\_таблицы2;

В результате такого запроса будет выведено отношение, в котором будут строки, соответствующие обеим таблицам, а также строки из Таблицы1, для которых не нашлось соответствия в Таблице2.

Правое внешнее соединение.

Общий синтаксис выглядит так:

SELECT имя\_столбца1, …, имя\_столбцаN FROM

Имя\_таблицы1 RIGHT JOIN Имя\_таблицы2;

В результате такого запроса будет выведено отношение, в котором будут строки, соответствующие обеим таблицам, а также строки из Таблицы2, для которых не нашлось соответствия в Таблице1.

Полное внешнее соединение.

Чтобы сохранить в выводе не соответствующие друг другу строки из обеих таблиц, включив их в результаты соединения, используйте полное внешнее соединение. SQL Server предоставляет оператор полного внешнего соединения, FULL OUTER JOIN, включающий все строки из обеих таблиц вне зависимости от того, есть ли в них совпадающие значения.

Синтаксис будет выглядеть так:

SELECT имя\_столбца1, …, имя\_столбцаN FROM

Имя\_таблицы1 FULL OUTER JOIN Имя\_таблицы2;

Перекрестное соединение.

Перекрестное соединение, не имеющее предложения WHERE, выполняет декартово произведение таблиц, вовлеченных в объединение. Размер результирующего набора декартова произведения вычисляется, как произведение количества строк в первой таблице на количество строк во второй таблице.

Синтаксис для создания перекрёстного соединения выглядит так:

SELECT имя\_столбца1, …, имя\_столбцаN FROM

Имя\_таблицы1 CROSS JOIN Имя\_таблицы2;

Внутреннее (естественное) соединение.

Внутреннее соединение возвращает только те строки, которые имеют соответствие во всех таблицах, участвующих в запросе.

Синтаксис выглядит так:

SELECT имя\_столбца1, …, имя\_столбцаN FROM

Имя\_таблицы1 INNER JOIN Имя\_таблицы2;

Примечание. Для всех соединений и запросов мы можем использовать группировку, сортировку, условие для фильтра, а также сочетание нескольких условий с помощью логических операторов AND, OR, NOT.

Задания на лабораторную работу.

Используйте БД, которая была создана в лабораторной работе №3.

Самостоятельно создать следующие запросы:

1. один запрос с использованием декартового произведения двух таблиц;
2. два запроса с использованием соединения двух таблиц по равенству;
3. один запрос с использованием соединения двух таблиц по равенству и условием отбора;
4. один запрос с использованием соединения по трем таблицам с условием;
5. создать копии ранее созданных запросов на соединение по равенству на запросы с использованием внешнего полного соединения таблиц (JOIN).
6. один запрос с использованием левого внешнего соединения;
7. один запрос на использование правого внешнего соединения;
8. один запрос с использованием внутреннего соединения.
9. один запрос на проверку какого-либо условия через самосоединение.

Отчет должен содержать код или скриншот запросов и результат их выполнения. На скриншоте результатов должно быть видно имя вашей БД.

**Лабораторная работа № 8**

**Выборка данных. Группировка и сортировка данных**

Цель: Изучить используемый в реляционных СУБД оператор извлечения данных из таблиц SELECT и выполнение группировки и сортировки данных. Изучить синтаксис языка модификации данных. Научится использовать встроенные функции в запросах.

Методические указания.

Сортировка данных.

Для того, чтобы выводить данные в том или ином порядке, существует специальный оператор ORDER BY. Оператор ORDER BY позволяет отсортировать извлекаемые значения по определенному столбцу.

Синтаксис:

SELECT столбцы FROM Имя\_таблицы

ORDER BY Имя\_столбца

По умолчанию применяется сортировка по возрастанию. С помощью дополнительного оператора DESC можно задать сортировку по убыванию, оператор ASC задает сортировку по возрастанию. если это необходимо. Можно отсортировать сразу по нескольким столбцам, при этом необязательно сортировать все эти столбцы только по убыванию или возрастанию.

Пример. Рассмотрим таблицу Guest. Нам необходимо отсортировать данные в ней по странам, по возрастанию (от А до Я), и именам, для примера отсортируем их по убыванию (от Я до А). Для этого применим следующий код:

SELECT \* FROM Guests

ORDER BY Страна ASC, ФИО DESC;

Группировка.

Для того, чтобы разделить данные в таблице на определенные группы, используется оператор GROUP BY.

Синтаксис:

SELECT столбцы

FROM таблица

[WHERE условие\_фильтрации\_строк]

[GROUP BY столбцы\_для\_группировки]

[HAVING условие\_фильтрации\_групп]

Рассмотрим оператор HAVING. Он используется, если вам нужна фильтрация уже сгруппированных данных.

Примечание. Группировка данных неразрывно связана с агрегатными функциями. Повторите их перед тем как приступать к работе.

Задания на лабораторную работу.

Импортировать данные из ресурса, указанного преподавателем, в соответствии с вариантом.

Самостоятельно создать следующие запросы:

Для 1 варианта:

1. Запрос, который покажет, сколько человек выбрало тот или иной предмет для первого экзамена, сделайте сортировку по убыванию.
2. Запрос, который покажет, какие предметы в качестве первого экзамена выбрало более 50 человек.
3. Запрос, который покажет сколько студентов выбрало в качестве второго экзамена физику, химию или биологию. Создайте два запроса один с помощью WHERE, другой используя HAVING.
4. Запрос, который покажет, сколько студентов выбрало тот или иной предмет в качестве второго экзамена и получивших более 100 баллов за оба экзамена.
5. Запрос, который покажет сколько студентов выбрали Биологию в качестве первого или второго предмета.

Для 2 варианта:

1. Запрос, который покажет, сколько человек выбрало тот или иной предмет в качестве любимого, сделайте сортировку по убыванию.
2. Запрос, который покажет, какие предметы в качестве любимого выбрало более 70 человек.
3. Запрос, который покажет сколько студентов выбрало в качестве любимого физику, русский или английский. Создайте два запроса один с помощью WHERE, другой используя HAVING.
4. Запрос, который покажет, сколько студентов выбрало тот или иной предмет в качестве любимого и имеющих средний балл хотя бы за один год более 4.
5. Запрос, который покажет сколько студентов имею средний бал за 2015 или 2016 год менее 3.

Для 3 варианта:

1. Запрос, который покажет, сколько книг продается у того или иного автора, сделайте сортировку по убыванию.
2. Запрос, который покажет, какие авторы продают более пяти книг.
3. Запрос, который покажет сколько книг продает Автор1, Автор2, Автор3. Создайте два запроса один с помощью WHERE, другой используя HAVING.
4. Запрос, который покажет, сколько книг продает тот или иной автор, считая книги с рейтингом более 3.
5. Запрос, который покажет сколько книг с рейтингом менее 3 и ценой более 500 продается в магазине.

Для 4 варианта:

1. Запрос, который покажет, сколько различных товаров предоставляет тот или иной поставщик, сделайте сортировку по убыванию.
2. Запрос, который покажет, какие поставщики предоставляют более десяти товаров.
3. Запрос, который покажет сколько товаров предоставляет Поставщик1, Поставщик2, Поставщик3. Создайте два запроса один с помощью WHERE, другой используя HAVING.
4. Запрос, который покажет, сколько товаров поставляет тот или иной поставщик, считая товары дороже 1500 рублей.
5. Запрос, который покажет сколько товаров с ценой более 800 рублей и количеством менее 5000 есть на складе.

Отчет должен содержать код или скриншот запросов и результат их выполнения.

Контрольные вопросы:

1. Что такое агрегатная функция?
2. Зачем нужно самосоединение?
3. Какие типы соединений вы знаете?
4. Что такое группировка? Приведите пример использования.
5. Как вывести всю строку, содержащую максимальное значение по тому или иному атрибуту?
6. В чем разница между операторами WHERE и HAVING?
7. В чем разница между функциями COUNT и SUM?
8. Можно ли отсортировать данные таблицы по нескольким атрибутам? Если нет, то почему, если да, то какой будет результат?
9. Можно ли одновременно использовать в одном запросе операторы WHERE и HAVING?
10. В какой последовательности происходит сортировка по умолчанию?
11. Как можно отсортировать уже сгруппированные записи?

**Лабораторная работа № 9**

**Инструкция MERGE**

Цель: Изучить принципы слияния таблиц, настройки и параметры инструкции MERGE.

Методические указания.

MERGE – операция в языке T-SQL, при которой происходит обновление, вставка или удаление данных в таблице на основе результатов соединения с данными другой таблицы или SQL запроса. Другими словами, с помощью MERGE можно осуществить слияние двух таблиц, т.е. синхронизировать их.

В операции MERGE происходит объединение по ключевому полю или полям основной таблицы (в которой и будут происходить все изменения) с соответствующими полями другой таблицы или результата запроса. В итоге если условие, по которому происходит объединение, истина (WHEN MATCHED), то мы можем выполнить операции обновления или удаления, если условие не истина, т.е. отсутствуют данные (WHEN NOT MATCHED), то мы можем выполнить операцию вставки (INSERT добавление данных), также если в основной таблице присутствуют данные, которое отсутствуют в таблице (или результате запроса) источника (WHEN NOT MATCHED BY SOURCE), то мы можем выполнить обновление или удаление таких данных.

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

Синтаксис выглядит так:

Синтаксис выглядит так:

MERGE <Основная таблица> AS TARGET

USING <Таблица или запрос источника>

AS SOURCE

ON <Условия объединения>

[ WHEN MATCHED [ AND <Доп. условие> ]

THEN <UPDATE или DELETE>

[ WHEN NOT MATCHED [ AND Доп. условие> ]

THEN <INSERT> ]

[ WHEN NOT MATCHED BY SOURCE [ AND <Доп. условие> ]

THEN <UPDATE или DELETE> ] [ ...n ] ;

Задания на лабораторную работу:

В соревнованиях по бегу участвовало несколько спортсменов. Для соревнования выделили три бегущих группы. У спортсмена была возможность бежать в любой из этих групп, при этом можно было бежать 2 или 3 раза, в этом случае учитывался бы лучший результат. Каждому участвующему спортсмену дается свой номер. Ниже приведены таблицы с результатами по каждой группе с номером атлета и временем, за которое он пробежал дистанцию.

|  |  |
| --- | --- |
| Номер спортсмена | Время, с |
| 371 | 56 |
| 372 | 57 |
| 373 | 58 |
| 374 | 56 |
| 375 | 51 |
| 376 | 62 |
| 377 | 61 |
| 378 | 67 |
| 379 | 48 |
| 380 | 50 |

Таб. 1. Группа 1.

|  |  |
| --- | --- |
| Номер спортсмена | Время, с |
| 372 | 61 |
| 374 | 58 |
| 376 | 57 |
| 378 | 54 |
| 380 | 53 |

Таб. 2. Группа 2.

|  |  |
| --- | --- |
| Номер спортсмена | Время, с |
| 370 | 58 |
| 371 | 57 |
| 372 | 67 |
| 373 | 62 |
| 374 | 55 |
| 375 | 52 |
| 376 | 60 |
| 377 | 58 |
| 378 | 59 |
| 379 | 53 |
| 380 | 54 |

Таб. 3. Группа 3.

Создайте в своей базе данных три таблицы и заполните их, аналогично таблицам 1, 2 и 3. Номер спортсмена должен заполняться не с клавиатуры, а с помощью свойства identity (x, y), которое устанавливает автоинкремент столбца(значение, шаг).Создайте сводную таблицу, в которой будут храниться все данные из первых трех. Добавьте в нее данные с помощью операции MERGE. Незаполненные ячейки должны быть пустыми (NULL).

|  |  |  |  |
| --- | --- | --- | --- |
| Номер спортсмена | Время 1, с | Время 2, с | Время 3, с |
| 370 | NULL | NULL | 58 |
| 371 | 56 | NULL | 57 |
| 372 | 57 | 61 | 67 |
| 373 | 58 | NULL | 62 |
| 374 | 56 | 58 | 55 |
| 375 | 51 | NULL | 52 |
| 376 | 62 | 57 | 60 |
| 377 | 61 | NULL | 58 |
| 378 | 67 | 54 | 59 |
| 379 | 48 | NULL | 53 |
| 380 | 50 | 53 | 54 |

Таб. 4. Возможный вид сводной таблицы для п.2.

Создайте новую таблицу, в которой будет лучший результат каждого спортсмена. Сортировку выполните с помощью операции MERGE. Обратите внимание, что лучшее время – это минимальное время.

|  |  |
| --- | --- |
| Номер спортсмена | Лучшее время |
| 370 | 58 |
| 371 | 56 |
| 372 | 57 |
| 373 | 58 |
| 374 | 55 |
| 375 | 51 |
| 376 | 57 |
| 377 | 58 |
| 378 | 54 |
| 379 | 48 |
| 380 | 50 |

Таб. 5. Возможный вид таблицы лучших результатов для п.3.

Выведите на экран номер победителя и его результат любым удобным способом.

|  |  |
| --- | --- |
| Победитель | Результат |
| 379 | 48 |

Таб. 6 Возможный вид таблицы победителя и его результат для п.4.

Аналогично выведите на экран худший результат за все соревнование.

Отчет должен содержать все коды и результаты ваших действий.

**Лабораторная работа № 10**

**Функции.**

Цель: Изучить разные типы функций, научиться создавать и вызывать функции разных типов.

Методические указания.

Аналогично функциям в языках программирования, определяемые пользователем функции SQLServer представляют собой подпрограммы, которые принимают параметры, выполняют действие, например, сложные вычисления, и возвращают результат этого действия в виде значения. Возвращаемое значение может быть либо единичным скалярным значением, либо результирующим набором.

Можно, однажды создав функцию, сохранить ее в базе данных, а затем любое число раз вызывать из своей программы. Определяемые пользователем функции могут быть изменены независимо от исходного кода программы.

В MS SQL Server предусмотрены несколько типов функций. В самом общем виде их можно разделить на следующие подгруппы: скалярные функции, функции возвращающие табличное значение и системные функции.

Пользовательские скалярные функции возвращают одно значение типа данных, заданного в предложении RETURNS. Для встроенной скалярной функции возвращаемое скалярное значение является результатом одной инструкции. Скалярная функция из нескольких инструкций может содержать последовательность инструкций Transact-SQL, возвращающих одно значение. Такие функции могут возвращать любые типы данных, кроме text, ntext, image, cursor и timestamp.

Общий синтаксис скалярной функции выглядит следующим образом:

CREATE FUNCTION Имя\_функции (через запятую указываются входные параметры, если есть )

RETURNS тип возвращаемых данных

BEGIN

Тело функции

RETURN выходной параметр

END

Для вызова скалярной функции следует написать SELECT, имя функции и входные параметры.

В качестве примера приведем создание функции, которая возвращает результат умножения цены на количество указанного товара. Товар будет идентифицироваться по названию и дате.

CREATE FUNCTION GetSumm

(@name varchar(50), @date datetime)

RETURNS numeric(10,2)

BEGIN

DECLARE @Summ numeric(10,2)

SELECT @Summ = Цена\*Количество

FROM Товары

WHERE [Название товара]=@name

AND Дата=@date;

RETURN @Summ

END

Чтобы вызвать эту функцию для получения «ручки от 11 ноября 2019 года» следует выполнить следующий запрос:

SELECT GetSumm (‘Ручка’, ‘2019-10-11’).

Определяемые пользователем функции с табличным значением возвращают данные типа table. Встроенная функция с табличным значением не имеет текста, таблица является результирующим набором одной инструкции.

Общий синтаксис выглядит так:

CREATE FUNCTION Имя\_функции

( входные параметры)

RETURNS TABLE

AS

RETURN (select запрос)

Можно использовать и многооператорную функцию, возвращающую табличное значение. Тогда ее синтаксис будет следующим:

CREATE FUNCTION Имя\_функции

( входные параметры )

RETURNS @имя\_переменной TABLE (атрибуты)

AS

BEGIN

Тело функции

RETURN

END

Чтобы получить результат выполнения такой функции, вызывать мы ее должны так же, как и таблицу.

SELECT \* FROM Имя\_функции (входные параметры)

В качестве примера, создадим функцию, которая будет возвращать таблицу товаров, и для каждой строки рассчитаем произведение столбцов количества и цены:

CREATE FUNCTION GetPrice()

RETURNS TABLE

AS

RETURN

(

SELECT Дата, [Название товара], Цена,

Количество, Цена\*Количество AS Сумма

FROM Товары

)

SQL Server предоставляет множество системных функций для выполнения различных операций. Их нельзя изменить. Они включают в себя, например, агрегатные или математические функции.

Задания на лабораторную работу.

1. Создайте функцию, выполняющую роль калькулятора, со следующими возможностями: сложение, вычитание, умножение, деление, sin, cos, tg, ctg, возведение в заданную степень.

2. Протестируйте функцию.

3. Создайте функцию, которая при вводе даты, будет возвращать название сезона (Зима, Весна, Лето, Осень).

4. Протестируйте функцию.

5. Создайте функцию, которая будет разбивать входную строку на отдельные строки. Символ пробела является разделителем строк.

6. Протестируйте функцию.

7. Импортируйте документ Doc.xsl.

8. Работа с таблицей StudList. Создайте функцию, возвращающую список студентов, обучающихся или отчисленных, в зависимости от входного параметра. Предполагается, что, если в столбце Studying стоит No, студент отчислен, в противном случае – обучается.

9. Протестируйте функцию.

Отчет должен содержать код или скриншот запросов для создания функций, результат ваших действий.

**Лабораторная работа № 11**

**Процедуры.**

Цель: Изучить разные типы процедур, научиться создавать и вызывать процедуры без входных и выходных параметров, с входными параметрами.

Методические указания.

Часто операция с данными представляет набор инструкций, которые необходимо выполнить в определенной последовательности или набор действий, которые нужно выполнять с некоторой периодичностью. В этом случае более оптимально будет инкапсулировать все эти действия в один объект - хранимую процедуру.

То есть по сути хранимые процедуры представляют набор инструкций, которые выполняются как единое целое. Тем самым хранимые процедуры позволяют упростить комплексные операции и вынести их в единый объект. Также хранимые процедуры позволяют ограничить доступ к данным в таблицах и тем самым уменьшить вероятность преднамеренных или неосознанных нежелательных действий в отношении этих данных.

И еще один важный аспект - производительность. Хранимые процедуры обычно выполняются быстрее, чем обычные SQL-инструкции. Все потому что код процедур компилируется один раз при первом ее запуске, а затем сохраняется в скомпилированной форме.

Для создания хранимой процедуры применяется команда CREATE PROCEDURE или CREATE PROC.

Общий синтаксис выглядит так:

CREATE PROCEDURE Имя\_процедуры AS

Тело процедуры

Чтобы вызвать процедуры используется команда EXEC или EXECUTE.

EXEC Имя\_процедуры

Процедуры могут принимать параметры. Параметры бывают входными - с их помощью в процедуру можно передать некоторые значения. И также параметры бывают выходными - они позволяют возвратить из процедуры некоторое значение. После названия процедуры идет список входных параметров, которые определяются также, как и переменные - название начинается с символа @, а после названия идет тип переменной. При вызове такой процедуры параметры указываются после имени процедуры.

Пример.

Пусть у нас есть таблица Students с параметрами имя, фамилия, отчество, дата рождения, номер паспорта. Нам необходимо создать процедуру, которая бы вносила данные о новом студенте. Таким образом, входными параметрами будут значения атрибутов Students. Синтаксис создания такой процедуры будет выглядеть следующим образом:

CREATE PROCEDURE New\_student

@name NVARCHAR(20),

@surname NVARCHAR(20),

@patronymic NVARCHAR(20),

@dob DATE,

@docnum CHAR(10)

AS

INSERT INTO Students

VALUES(@name, @surname, @patronymic, @dob, @docnum)

Вызов такой функции может выглядеть так:

EXEC New\_student ‘Александр’, ‘Белов’, ‘Алексеевич’, ‘2003-05-12’,’4017111222’

Изучите правила использования выходных параметров и возвращения результата здесь: https://metanit.com/sql/sqlserver/11.3.php и приступайте к выполнению работы.

Задания на лабораторную работу.

1. Работа с таблицей StudList из предыдущей ЛР. Создать процедуру с входным параметром и return. Процедура должна получать входной параметр – Имя студента, а на выходе выдать 1 или 0, в зависимости от того обучается он или нет.

2. Проверить процедуру.

3. Работа с таблицей Shop. Импортируйте таблицу Shop. В магазине проходит акция. Для всех, кто оставил комментарий на сайте, скидка 5%. Создайте процедуру, которая уменьшит цены в таблице Shop на 5% (атрибут Price), в тех строках, где значение столбца Comment yes.

4. Проверить процедуру.

5. Работа с таблицей Shop. Создать процедуру с входным параметром. Входной параметр – номер клиента. Процедура удаляет все строки с заказами этого клиента.

6. Работа с таблицей Shop. Создать процедуру с входным и выходным параметрами, где входной параметр – номер клиента, а выходной – самый дорогой заказ выбранного клиента.

7. Проверить процедуру.

Контрольные вопросы:

1. Какие типы процедур вы знаете?

2. Где хранятся процедуры?

3. Какой синтаксис у создания процедур?

4. Какой синтаксис у выполнения процедур?

5. В чем разница между функцией и процедурой?

6. Зачем нужны процедуры?

**Лабораторная работа № 12**

**Создание, программирование и управление триггерами**

Цель: Изучить назначения и типы триггеров, условия их активации, синтаксис команд языка T – SQL для их создания, модификации, переименования, программирования и удаления.

Методические указания.

Триггер – это специальный тип хранимых процедур, которые запускаются сервером автоматически при выполнении тех или иных действий с данными таблицы. Триггеры реагируют на попытку вставки ( insert), изменения( update) или удаления( delete) данных.

У триггеров есть параметры FOR, AFTER и INSTEAD OF, указываемые при создании триггера, определяют его поведение следующим образом:

FOR – запуск триггера при выполнении заданной в этом списке команды;

AFTER – запуск триггера после успешного выполнения команд списка;

INSTEAD OF – триггеры вызывается вместо выполнения команд списка.

Можно определить несколько AFTER – триггеров для каждой операции INSERT, UPDATE и DELETE. По умолчанию все триггеры являются AFTER – триггерами.

Триггеры нельзя создавать для временных или системных таблиц, однако можно создавать для представлений.

Синтаксис команды создания триггера следующий:

CREATE TRIGGER Имя\_триггера

ON Имя\_таблицы | Имя\_представления

FOR | AFTER | INSTEAD OF

INSERT | UPDATE | DELETE

AS команда\_SQL

Синтаксис команды удаления триггера:

DROP TRIGGER Имя\_триггера

Синтаксис команды редактирования триггера:

ALTER TRIGGER Имя\_триггера

ON Имя\_таблицы | Имя\_представления

FOR | AFTER | INSTEAD OF

INSERT | UPDATE | DELETE

AS команда\_SQL

Также допускается вызов хранимых процедур, включая системные, но такие задания мы будем выполнять в следующей работе.

Пример триггера, который будет выводить удаленные значения, после удаления строк

Пусть у нас существует следующее отношение BigTab.

|  |  |  |  |
| --- | --- | --- | --- |
| Серийный N | Имя владельца | Дата покупки | Гарантийное обслуживание |
| 111111111 | Перевалов Константин | 10.08.2018 | есть |
| 111111112 | Первакова Марина | 23.11.2013 | нет |
| ….. | ….. | ….. | … |

Чтобы создать триггер, который бы выводил удаленные значения нам потребуется следующий код:

CREATE TRIGGER FirstTrigger

ON BigTab

AFTER DELETE

AS Select \* from deleted;

Задания на лабораторную работу.

1.Импортируйте данные из файла AUTH.xls

2.Создайте триггер Trigger\_1, который после изменения данных будет выводить строки, со старыми и новыми значениями, используйте триггер AFTER UPDATE.

Протестируйте ваш триггер.

3.Создайте триггер Trigger\_2, который вместо удаления выведет следующую информацию: «попытка удаления информации об au\_id». Используйте триггер INSTEAD OF – триггеры вызывается вместо выполнения команд списка..

Протестируйте ваш триггер.

4.Создайте триггер Trigger\_3, который будет считать и выводить количество удаленных строк

Протестируйте ваш триггер.

Контрольные вопросы:

1. Какие типы триггеров вы знаете?

2. Можем ли мы обращаться к журналу транзакций при создании триггеров?

3. Как можно протестировать ваш триггер?

4. Сколько триггеров может иметь одно отношение?

5. Можно ли создать один триггер на несколько операций?

6. Как можно увидеть созданные триггеры?

7. Как можно изменить тело триггера?

**Лабораторная работа № 13**

**Создание и управление транзакциями**

Цель: Изучение способов обеспечения надежной работы SQL Server с помощью механизма транзакций и контрольных точек, приобретение навыков управления транзакциями, а также ознакомление с физической и логической архитектурой журнала транзакций.

Методические указания.

Одним из способов повышения надежности работы системы MS SQL Server является применение встроенного в систему механизма транзакций и контрольных точек и умелое его управление. Транзакция – это одна или несколько последовательных команд языка T – SQL, образующих логически завершенный пакет и выполняемых как единое целое. Если по какой-либо причине хотя бы одна из команд пакета не выполняется, то происходит откат системы к состоянию, в котором она была до начала транзакции, и транзакция считается не выполненной.

По умолчанию каждая команда выполняется как самостоятельная транзакция. При необходимости в пакете можно явно указать начало и конец транзакции. Обработка транзакций в любой системе управления базами данных должна производиться с соблюдением следующих правил – правил ASID (Atomicity, Consistency, Isolation и Durability): Atomicity – атомарность: выполняемые в транзакции изменения либо выполняются все, либо не выполняются вовсе; Consistency – согласованность: все данные после выполнения транзакции должны находиться в согласованном состоянии с соблюдением всех правил и ограничений целостности; Isolation – изолированность: изменения данных, выполняемых Durability – долговечность: после завершения транзакции ничто не может вернуть систему в состояние, в котором она была до начала транзакции (происходит фиксация транзакции).

Для управления явными транзакциями применяют команды:

BEGIN TRANSACTION Имя\_транзакции – начало транзакции;

COMMIT TRANSACTION Имя\_транзакции – фиксирование изменений и конец транзакции;

ROLLBACK TRANSACTION Имя\_транзакции – откат транзакции;

Если в команде ROLLBACK не задано имя транзакции, то откатываются все вложенные транзакции и транзакция самого высокого уровня. Если же имеется необходимость откатить лишь часть транзакций, то предварительно надо создать точку сохранения с помощью команды SAVE TRANSACTION, которую следует указывать при откате.

Функция @@TRANCOUNT предназначена для определения количества активных транзакций, начатых в активном соединении. Во всех транзакциях нельзя использовать следующие команды: ALTER DATABASE, BACKUP LOG, CREATE DATABASE, DISK INIT, DROP DATABASE, DUMP TRANSACTION, LOAD DATABASE, LOAD TRANSACTION, RECONFIGURE, RESTORE DATABASE, RESTORE LOG, UPDATE STATISTICS, а также системной хранимой процедуры sp\_droption и любой другой хранимой процедуры, изменяющей значения в системной базе master. Контрольные точки минимизируют данные в журнале транзакций.

Задания на лабораторную работу.

1.Проверьте работу транзакции следующим образом. Выберите любое отношение в вашей базе данных. Создайте транзакцию, в которой будет содержаться один запрос вставки. Запустите ее дважды. В первом случае завершите транзакцию командой COMMIT, второй раз ROLLBACK. Проанализируйте результаты. Вставьте скриншот результата в отчет.

2.Работа с вложенными транзакциями.

Создать вложенные транзакции, выполнив следующие команды:

CREATE TABLE table\_name (a1 int) -- 0-й уровень

BEGIN TRANSACTION tr1-- 1-й уровень

INSERT INTO table\_name VALUES (11)

BEGIN TRANSACTION tr2-- 2-й уровень

INSERT INTO table\_name VALUES (22)

BEGIN TRANSACTION tr3 -- 3-й уровень

INSERT INTO table\_name VALUES (33)

SELECT \* FROM table\_name

SELECT 'Вложенность транзкций', @@TRANCOUNT

ROLLBACK TRANSACTION

SELECT \* FROM table\_name -- откат на 0-й уровень

SELECT 'Вложенность транзакций', @@TRANCOUNT

Проанализируйте результаты.

3. Работа с данными из файла groups. Импортируйте все листы. В первом листе находятся данные студентов, поступивших в 2018 году. В других листах содержаться списки групп. Задача импортировать данные из списков групп в сводную таблицу всех поступивших. Создайте явную транзакцию, в которой данные о студентах будут вноситься, а, если эти данные уже были внесены раннее, то транзакция будет откатываться. Результаты продемонстрируйте в отчете.

4.Изучите самостоятельно команду SET IMPLICIT\_TRANSACTIONS ON/OFF.

Продемонстрируйте разницу между этими режимами на примере любой таблицы в вашей базе данных.

Контрольные вопросы:

1. Зачем нужны транзакции?
2. С какими режимами транзакции вы работали?
3. Какие команды являются признаком начала неявной транзакции?
4. Зачем нужны контрольные точки транзакций? Продемонстрируйте их работу.
5. Можно ли узнать количество активных транзакций?

**Лабораторная работа № 14**

**Создание и управление курсорами**

Цель: Изучение назначения и типов курсоров, синтаксиса языка T – SQL для создания и открытия курсоров, выборки данных из курсора и изменения строк таблиц с помощью курсоров, удаления данных, закрытия и освобождения курсоров, а также приобретения навыков их применения и управления с помощью команд и системных хранимых процедур SQL Server.

Методические указания

Операции в реляционной базе данных выполняются над множеством строк. Например, набор строк, возвращаемый инструкцией SELECT, содержит все строки, которые удовлетворяют условиям, указанным в предложении WHERE инструкции. Такой полный набор строк, возвращаемых инструкцией, называется результирующим набором. Можно в результирующий набор не включать те или иные столбцы, применяя вертикальную фильтрацию. Результирующие наборы могут содержать сотни тысяч строк, и клиентские приложения не всегда справляются с таким объемом данных.

Приложения, особенно интерактивные, не всегда эффективно работают с результирующим набором как с единым целым. Им нужен механизм, позволяющий обрабатывать одну строку или небольшое их число за один раз. Для решения этой проблемы используются курсоры, которые представляют собой окна, налагаемые на результирующий набор данных и выделяющие требуемую часть данных. Курсоры являются расширением результирующих наборов, которые предоставляют такой механизм. Перемещая созданный курсор, можно получить доступ ко всем результирующим данным. Таким образом, курсоры SQL Server представляют собой механизм обмена данными между сервером и клиентом, который минимизирует ресурсы клиентского приложения.

Курсоры позволяют усовершенствовать обработку результатов:

позиционируясь на отдельные строки результирующего набора;

получая одну или несколько строк от текущей позиции в результирующем наборе;

поддерживая изменение данных в строках в текущей позиции результирующего набора;

поддерживая разные уровни видимости изменений, сделанных другими пользователями для данных, представленных в результирующем наборе;

предоставляя инструкциям Transact-SQL в скриптах, хранимых процедурах и триггерах доступ к данным результирующего набора.

Однако всегда, когда это возможно, следует избегать использования курсоров и применять команды SELECT, UPDATE, DELETE и INSERT.

Один курсор может базироваться на нескольких таблицах, расположенных в разных базах данных. Операция считывания определенных в курсоре данных называется выборкой (fetch). Если за одну операцию курсор позволяет выбрать несколько строк таблицы, то такой курсор называется блочным.

По способу просмотра данных курсоры бывают последовательные (forward only), которые обеспечивают просмотр строк только в одном направлении – от начала к концу, и прокручиваемые, которые допускают просмотр в обоих направлениях и переход к произвольной строке.

По представляемым возможностям курсоры делятся на четыре типа: статические, динамические, последовательные ключевые. Тип курсора определяется на стадии его создания и не может быть изменен.

Статический курсор (static cursor) называют также курсорами моментального снимка (snapshot cursor). При открытии такого курсора сервер выбирает все данные, соответствующие заданным критериям, и сохраняет результирующий набор строк в системной базе данных tempdb без изменения, если даже исходные строки и изменяются. Поэтому статический курсор всегда открывается в режиме “только для чтения”.

Динамические курсоры (dynamic cursor) противоположны статическим. При их использовании не создается полная копия исходных данных, а выполняется динамическая выборка данным из исходных таблиц только при обращении пользователя к тем или иным данным, при этом на время выборки соответствующие строки блокируются сервером. После выборки строк исходные строки могут изменяться пользователями, но эти изменения уже не отражаются в выбранных строках. С другой стороны, изменения в выбранных строках не будут видны другим пользователям, пока они не будут подтверждены (committed).

Последовательный курсор выбирает данные только от начала к его концу. Он не хранит результирующий набор. Строки считываются из базы данных, как только они выбираются в курсоре. Это позволяет отображать все изменения в базе данных. В курсоре видно самое последнее состояние данных.

Курсоры, зависящие от набора ключей (keyset-driven cursor), или ключевые курсоры, построены на основе уникальных идентификаторов. Множество всех уникальных идентификаторов (ключей) строк таблиц базы данных называется набором ключей. Сервер блокирует строки исходных таблиц только на время составления таблицы ключей. Ключевой курсор представляет набор ключей, идентифицирующих строки полного результирующего набора курсора. Набор ключей строится в системной базе данных tempdb.

При работе с курсорами можно выделить пять основных операций: создание курсора, открытие курсора, выборка из курсора и изменение строк данных с помощью курсора, закрытие курсора и освобождение курсора.

Синтаксис команды для создания курсора следующий:

DECLARE Имя курсора CURSOR [LOCAL/GLOBAL] [FORWARD\_ONLY\SCROLL] [STATIC\KEYSET\DINAMIC\FAST\_FORWARD] [READ\_ONLY\SCROLL\_LOCKS\OPTIMISTIC] [TYPE\_WARNING]

FOR select\_statement [FOR UPDATE [OF column\_name [,…n]]].

Для открытия курсора используется команда:

OPEN {{[GLOBAL] Имя курсора}\Имя переменной}.

Для выборки данных необходимо применять команду:

FETCH [[NEXT\PRIOR\FIRST\LAST\ABSOLUTE {n\@nvar}\ RELATIVE {n\@nvar} ] FROM ] {{[GLOBAL] Имя курсора}\Имя переменной} []INTO @ Имя переменной [,…n]]

Команды UPDATE, DELETE, CLOSE и DEALLOCATE позволят соответственно произвести изменение данных, удаление данных, закрытие и освобождение курсора.

Задания на лабораторную работу.

1. Создайте таблицу Clients со следующими атрибутами: Id, Name, Address.
2. С помощью курсора создайте запрос, который будет выводить информацию о всех клиентах c нечетным Id.
3. Создайте таблицу OlympMath с результатами Олимпиады по математике за 2019, 2020 и 2021 года. Таблица OlympMath состоит из следующих атрибутов: id, Name, res19, res20, res21.
4. С помощью курсора создайте запрос, который будет находить и выводить лучший результат для каждого участника. То есть максимальный в строке.
5. Создайте две таблицы со следующей структурой (папки и файлы):

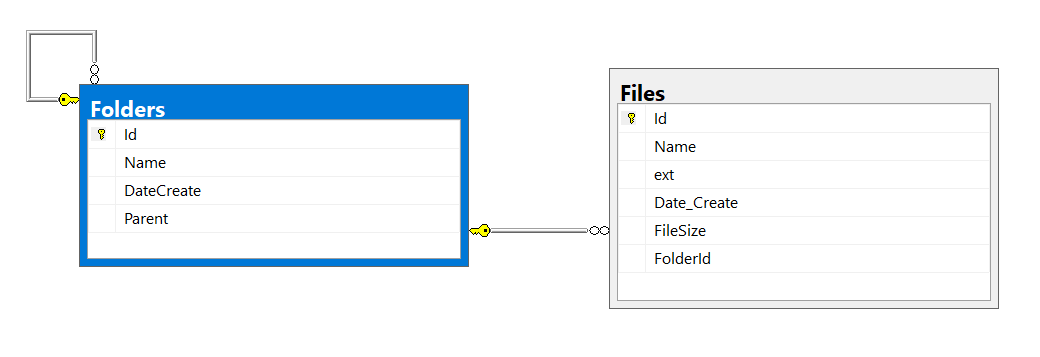


Рис.14 Таблицы Папки и файлы

Реализуйте функцию, которая будет принимать на входе идентификатор каталога и выводить все файлы и каталоги, которые находятся в заданном каталоге и во всех подкаталогах.