1. **Основные понятия: база данных; СУБД; принципы и этапы построения, основные модели данных, основные понятия теории реляционных баз данных**.

База данных - совокупность связанных данных, организованных по определенным правилам, предусматривающим общие принципы описания, хранения и манипулирования, независимая от прикладных программ.

Систе́ма управле́ния ба́зами да́нных (СУБД) — совокупность программных и лингвистических средств общего или специального назначения, обеспечивающих управление созданием и использованием баз данных.

Принципы построения баз данных

· Высокое быстродействие (малое время отклика на запрос).

· Простота обновления данных.

· Независимость данных - возможность изменения логической и физической структуры БД без изменения представлений пользователей.

· Совместное использование данных многими пользователями.

· Безопасность данных - защита данных от преднамеренного или непреднамеренного нарушения секретности, искажения или разрушения.

Этапы разработки БД:

концептуальное проектирование — сбор, анализ и редактирование требований к данным;

логическое проектирование — преобразование требований к данным в структуры данных;

физическое проектирование — определение особенностей хранения данных, методов доступа и т. д.

*Иерархическая модель данных.* Иерархическая модель данных, имеет иерархическую структуру, т.е. каж­дый из элементов связан только с одним стоящим выше элементом, но в то же время на него могут ссылаться один или несколь­ко стоящих ниже элементов.

*Сетевая модель данных*. Единственное отличие между иерархической и сетевой моделями данных заключается в том, что в последней каждый элемент дан­ных (узел) может быть связан с любым другим элементом (узлом).

*Реляционная модель данных.* Основная идея реляционной модели данных заключается в том, чтобы представить любой набор данных в виде двумерного массива — таблицы. В простейшем слу­чае реляционная модель описывает единственную двумерную таб­лицу, но чаще всего эта модель описывает структуру и взаимоотношения между несколькими различными таблицами.

Выделим следующие основные понятия реляционных баз данных: тип данных, домен, атрибут, кортеж, отношение, первичный ключ.

Значения данных, хранимые в реляционной базе данных, являются типизированными, т. е. известен тип каждого хранимого значения.

Наиболее правильной интуитивной трактовкой понятия домена является его восприятие как допустимого потенциального, ограниченного подмножества значений данного типа.

*Атрибут* - это свойство, характеризующее объект.

*Кортеж,* соответствующий данной схеме отношения, - это множество пар {имя атрибута, значение}, которое содержит одно вхождение каждого имени атрибута, принадлежащего схеме отношения.

*Отношение* - это множество кортежей, соответствующих одной схеме отношения.

1. **Проектирование баз данных. Модели данных.**

Основные этапы, на которые разбивается процесс проектирования базы данных информационной системы:

1. **Концептуальное проектирование** - сбор, анализ и редактирование требований к данным. Для этого осуществляются следующие мероприятия:
   * обследование предметной области, изучение ее информационной структуры
   * выявление всех фрагментов, каждый из которых харакетризуется пользовательским представлением, информационными объектами и связями между ними, процессами над информационными объектами
   * моделирование и интеграция всех представлений
2. **Логическое проектирование** - преобразование требований к данным в структуры данных. На выходе получаем СУБД-ориентированную структуру базы данных и спецификации прикладных программ. На этом этапе часто моделируют базы данных применительно к различным СУБД и проводят сравнительный анализ моделей.
3. **Физическое проектирование** - определение особенностей хранения данных, методов доступа и т.д.

**Модели данных**:

Иерархическая модель данных. Иерархическая модель данных, имеет иерархическую структуру, т.е. каж­дый из элементов связан только с одним стоящим выше элементом, но в то же время на него могут ссылаться один или несколь­ко стоящих ниже элементов.

Сетевая модель данных. Единственное отличие между иерархической и сетевой моделями данных заключается в том, что в последней каждый элемент дан­ных (узел) может быть связан с любым другим элементом (узлом).

Реляционная модель данных. Основная идея реляционной модели данных заключается в том, чтобы представить любой набор данных в виде двумерного массива — таблицы. В простейшем слу­чае реляционная модель описывает единственную двумерную таб­лицу, но чаще всего эта модель описывает структуру и взаимоотношения между несколькими различными таблицами.

1. **Нормализация реляционной базы данных (1, 2, 3 формы).**

**Нормализация** данных − это процесс, в результате выполнения которого таблицы базы данных проверяются на наличие зависимостей между столбцами таблицы.

Первая и главная нормальная форма требует от таблицы (а точнее, от ее проектировщика) следования следующим правилам:

* Каждый столбец в строке должен быть атомарным, т.е. столбец может содержать одно и только одно значение для заданной строки.
* Каждая строка в таблице обязана содержать одинаковое количество столбцов. Учитывая обязательную атомарность столбцов, следует, что все строки в таблице должны иметь одинаковое количество значений.
* Все строки в таблице, в общем, должны быть уникальны. Значения в столбцах могут дублироваться, но строки, взятые целиком — не могут.

Два правила второй нормальной формы говорят о том, что:

* Таблица обязана соответствовать первой нормальной форме.
* Все столбцы, не входящие в полный первичный ключ, должны зависеть от полного первичного ключа

Третья норма данных расширяет две предыдущие, неся в себе два правила:

* Таблица должна соответствовать второй нормальной форме.
* Все столбцы, не входящие в полный первичный ключ, должны зависеть от него и не должны зависеть друг от друга.

1. **Основы SQL: стандарты, группы операторов и их назначение.**

Операторы SQL делятся на:

Операторы определения данных (Data Definition Language, DDL):

CREATE создает объект БД (саму базу, таблицу, представление, пользователя и т. д.),

ALTER изменяет объект,

DROP удаляет объект;

Операторы манипуляции данными (Data Manipulation Language, DML):

SELECT считывает данные, удовлетворяющие заданным условиям,

INSERT добавляет новые данные,

UPDATE изменяет существующие данные,

DELETE удаляет данные;

Операторы определения доступа к данным (Data Control Language, DCL):

GRANT предоставляет пользователю (группе) разрешения на определенные операции с объектом,

REVOKE отзывает ранее выданные разрешения,

DENY задает запрет, имеющий приоритет над разрешением;

Операторы управления транзакциями (Transaction Control Language, TCL):

COMMIT применяет транзакцию,

ROLLBACK откатывает все изменения, сделанные в контексте текущей транзакции,

SAVEPOINT делит транзакцию на более мелкие участки.

**5.Основные принципы работы SQL Server: клиент, сервер, системные базы данных.**

**6.Создание базы данных: оператор создания, файловые группы, журнал транзакций, основные опции.**

Все БД, управляемые сервером MicrosoftSQLServer 2008, можно разбить на две группы: системные и пользовательские. Системные БД создаются при инсталляции сервера и используются сервером в процессе его работы. Особую роль среди них играет БД MASTER, хранящая информацию обо всех остальных БД, управляемых сервером.

Для создания пользовательской БД применяется оператор CREATEDATABASE, который должен выполняться в контексте системной БД MASTER.

Все файлы БД, кроме файлов журнала транзакций, распределены по файловым группам. Файловые группы – это поименованный набор файлов БД. Журнал транзакций в операторе CREATEDATABASE описывается отдельно в секции, обозначенной ключевыми словами LOGON. Обязательной является файловая группа, называемая первичной. Для ее обозначения используются ключевые слова ONPRIMARY.

Описание каждого файла БД включает несколько атрибутов: логическое имя (заданное ключевым словом name), файл операционной системы (filename), а также первоначальный размер (size), максимальный размер (maxsize), приращение размера (filegrowth), заданное абсолютной величиной или в процентах по отношению к первоначальному размеру. Обязательными являются только первичный файл БД, всегда расположенный в первичной файловой группе, и файл журнала транзакций.

Первичный файл содержит всю информацию, необходимую для нормального функционирования БД, а также данные (например, строки таблиц).

Файл журнала транзакций используется для восстановления данных БД в случае аварийного завершения работы сервера.

Разработчик БД помимо первичной файловой группы может создавать дополнительные файловые группы, называемые вторичными. Во вторичных файловых группах могут располагаться только вторичные файлы. В первичной файловой группе помимо обязательного первичного файла тоже могут быть расположены вторичные файлы.

Файловые группы используются в основном для упрощения администрирования БД. Существуют операторы (например, копирования и восстановления БД), позволяющие рассматривать файловую группу как единое целое и выполнять операции не для каждого файла по отдельности, а сразу для файловой группы. Кроме того, файловые группы применяются при секционировании таблиц – технологии, позволяющей разместить фрагменты таблицы в разных файловых группах, что при некоторых дополнительных условиях позволяет повысить производительность запросов к этой таблице.

При создании таблиц и индексов дисковая память для них автоматически отводится в файловой группе по умолчанию. Для размещения этих объектов БД в другой файловой группе следует явно указывать ее имя в операторе CREATE, создающем таблицу или индекс. Кроме того, с помощью оператора ALTERDATABASE можно сделать другую файловую группу, которая будет использоваться по умолчанию.

**7. SQL DDL: операторы, назначение, применение.**

Операторы определения данных (язык DDL).

Соответствующие операторы предназначены для создания, удаления, изменения основных объектов модели данных реляционных БД: таблиц, представлений, индексов.

CREATE TABLE <имя> - создание новой таблицы в базе данных.

**CREATE TABLE Товары**

**( Наименование nvarchar(50)**

**constraint PK\_Товары primary key,**

**Цена real,**

**Количество int -- количество товара**

**)**

DROP TABLE <имя> - удаление таблицы из базы данных.

**DROP ТОВАРЫ**

ALTER TABLE <имя> - изменение структуры существующей таблицы или ограничений целостности, задаваемых для данной таблицы.

**ALTER Table Товары DROP Column Дата\_поступления;**

При выполнении аналогичных операций с представлениями или индексами в указанных операторах вместо служебного слова TABLE записывается слово VIEW (представление) или слово INDEX (индекс)

**8.SQL DML: операторы, назначение, применение.**

Операторы манипулирования данными (язык DML).

Операторы DML работают с базой данных и используются для изменения данных и получения необходимых сведений.

SELECT – выборка строк, удовлетворяющих заданным условиям. Оператор реализует, в частности, такие операции реляционной алгебры как "селекция" и "проекция".

**SELECT Наименование, Цена From Товары;**

UPDATE – изменение значений определенных полей в строках таблицы, удовлетворяющих заданным условиям.

**UPDATE ТОВАРЫ set Количество = 1;**

INSERT – вставка новых строк в таблицу.

**INSERT into Товары (Наименование, Цена,Количество)**

**values ('Полка', 20.0, 10)**

DELETE – удаление строк таблицы, удовлетворяющих заданным условиям.

**DELETE from ТОВАРЫ where NZach like '13';**

Применение этого оператора учитывает принципы поддержки целостности, поэтому он не всегда может быть выполнен корректно.

**9.SQL DCL: операторы, назначение, применение.**

Операторы разграничения доступа пользователей к объектам базы данных (DCL).

GRANT – создание в системе безопасности записи, разрешающей пользователю работать с данными или выполнять определенные операции SQL.

Предположим, пользователь Р1 является владельцем таблицы Student и нужно передать пользователю Р2 право на формулирование запросов к этой таблице:

**GRANT SELECT ON**Student**TO**P2**;**

DENY - создание в системе безопасности записи, запрещающей доступ для определенной учетной записи.

Выдаёт разрешение пользователю peter на создание таблиц и процедур.

**DENY CREATE TABLE, CREATE PROCEDURE** TO peter

REVOKE - отзывает ранее выданные разрешения,

Отзывает разрешение для создания таблиц пользователю test

**REVOKE CREATE TABLE FROM** test;

**10.SQLTCL: операторы, назначение, применение.**

Операторы TCL предназначены для создания транзакций (транзакция – это несколько DML-операторов, которые либо все успешно выполнятся, либо все не выполняются). TCL SQL включает четыре оператора: BEGIN TRAN, SAVE TRAN, COMMIT TRAN и ROLLBACK TRAN.

Оператор BEGIN TRAN указывает на начало транзакции. Структура оператора:

Предполагается, что за оператором BEGIN TRAN будут следовать один или несколько DML-операторов, входящих в транзакцию.

Снизу группу DML-операторов, входящую в транзакцию, ограничивает один из операторов: COMMIT TRAN или ROLLBACK TRAN.

Оператор COMMIT TRAN фиксирует транзакцию – подтверждает ее успешное завершение.

Оператор ROLLBACK TRAN откатывает транзакцию – отменяет все изменения, произведенные DML-операторами в таблицах БД в рамках данной транзакции.

Оператор SAVE TRAN применяется для формирования контрольной точки – промежуточного состояния транзакции, к которому может быть осуществлен откат. В тексте SQL-скрипта оператор SAVE TRAN должен находиться между операторами BEGIN TRAN и ROLLBACK TRAN.

**11.Типы ограничений целостности.**

Ограничения, накладываемые на столбцы таблицы, предотвращающие появление в БД данных, не соответствующих предварительно заданным свойствам таблицы, называются ограничениями целостности.

В простейшем случае ограничения целостности не допускают появление в БД данных, не соответствующих типу.

Помимо проверки типа данных при создании таблицы БД имеется возможность описать более сложные ограничения. Все типы таких ограничений целостности перечислены в табл..

Ограничения целостности

Условное обозначение

ограничения целостности Действие ограничения

целостности

datatype

тип данных Предотвращает появление в столбце значений, не соответствующих типу данных

notnull

запрет значений null Предотвращает появление в столбце значений null

default

значение по умолчанию Устанавливает значение в столбце по умолчанию при выполнении операции INSERT

primary key

первичный ключ Предотвращает появление в столбце (группе столбцов)

повторяющихся значений (комбинации значений) и пустого значения (комбинации пустых значений)

foreign key

внешний ключ Устанавливает связь между таблицей со столбцом,

имеющим свойство foreignkey (FK) и таблицей,

имеющей столбец со свойством primarykey (PK);

предотвращает не согласованные операции

между PK и FK

unique

уникальное значение Аналогично primarykey, но допускает пустые значения и

не может быть использован для связи с foreignkey

check

проверка значений Предотвращается появление в столбце значения, не удовлетворяющего логическому условию

Для ограничений целостности PRIMARYKEY, FOREIGNKEY, UNIQUE и CHECK может быть задано имя, которое при возникновении ошибки, связанной с этим ограничением, будет указано в сообщении сервера. В том случае, если это имя не задано, при создании таблицы сервер назначает ограничениям этих типов собственные имена.

**12.Оператор SELECT.**

Среди DML-операторов SELECT является наиболее сложным и важным. Умение составлять SELECT-запросы – необходимое условие разработки приложений БД. В общем случае, SELECT-запрос может быть частью запроса к БД, выполненного с помощью любого другого DML-оператора.

Структура оператора SELECT

Оператор SELECT состоит из нескольких секций: SELECT, INTO, FROM, WHERE, GROUP BY, HAVING, ORDER BY. Кроме того, в секции SELECT могут использоваться опции TOP и/или DISTINCT, а в секции ORDERBY опции ASC или DESC. Обязательной является только секция SELECT.

Результатом работы SELECT-запроса являются данные, организованные в виде набора однородных строк, называемого результирующим набором данных.

Основное назначение секции SELECT – указать порядок и имена столбцов результирующего набора данных. Основой для содержимого SELECT-секции является набор однородных строк, описанный во FROM. В простейшем случае SELECT без изменения выводит все столбцы однородного набора строк, сформированного в секции FROM. Опция DISTINCT позволяет избавиться от повторяющихся строк в результирующем наборе, а TOP – ограничить в нем их количество. В общем случае в этой секции указывается список выражений, содержащих имена столбцов набора строк из FROM, константы, а также скалярные результаты SELECT-запросов. Такой список будем называть SELECT-списком. Часто говорят, что SELECT-список определяет вертикаль результирующего набора.

Секция INTO позволяет на основе структуры результирующего набора (заданную SELECT-списком) создать новую таблицу и заполнить ее строками результирующего набора.

Секция FROM в общем случае содержит выражение, результатом которого является набор однородных строк с явно поименованными столбцами. В простейшем случае – это таблица БД, а в общем случае, может быть достаточно сложное выражение, содержащее несколько таблиц, SELECT-запросов, представлений и табличных функций (представления и табличные функции рассматриваются позже).

Секция WHERE применяется для отбора (фильтрации) строк, сформированных в секции FROM. Как правило, секция содержит логическое выражение, которое вычисляется для каждой строки. В том случае, если выражение принимает истинное значение – строка отбирается. Логическое выражение может быть достаточно сложным и содержать в себе другой SELECT-запрос. Говорят, что секция WHERE определяет горизонталь результирующего набора.

GROUPBY – секция, позволяющая сгруппировать данные по значениям, хранящимся в столбцах однородного набора строк, сформированного в секции FROM. Секция GROUPBY ограничивает содержимое SELECT-списка.

Секция HAVING имеет такое же назначение, что и WHERE, но применяется после выполнения секции GROUPBY.

Для сортировки результирующего набора применяется секция ORDERBY. Выражения дают возможность задать порядок строк в результирующем наборе. Опции ASC и DESC позволяют указать порядок сортировки строк (по возрастанию или по убыванию). Применение опции DISTINCT в секции SELECT ограничивает возможности сортировки.

**13.Операторы INSERT, DELETE, UPDATE.**

Оператор INSERT относится к группе операторов DML SQL и предназначен для добавления строк в таблицу.

Оператор INSERT может добавить ноль, одну или более строк в заданную (таблица-приемник) таблицу БД. Источником строк для добавления служит тоже таблица (таблица-источник), содержащая ноль, одну или более строк. Причем таблица-приемник – это таблица БД (временная или постоянная) или таблица-переменная (этот тип таблиц рассматривается позже), а таблица-источник – это результирующий набор данных (строки, полученные в результате SELECT-запроса) или просто заданный список строк.

В зависимости от того, каким образом описывается таблица-приемник и таблица источник, различают несколько форм оператора INSERT.

Простейшая форма:

Простейшая форма оператора INSERT подразумевает ввод данных во все столбцы таблицы-приемника, при этом считается, что столбцы упорядочены в той последовательности, в которой они заданы оператором CREATETABLE при создании таблицы-приемника.

Форма оператора INSERT с применением списка столбцов:

В том случае, если требуется обозначить порядок значений в списке VALUES оператора INSERT, применяется список столбцов. Список с именами столбцов указывается в скобках сразу после имени таблицы в секции.

Конструкция INSERT … SELECT

В качестве таблицы-источника может выступать результирующий набор, сформированный оператором SELECT.

Конструкция INSERT … EXECUTE

SELECT-запросы, используемые в операторе INSERT, могут быть достаточно сложными и, кроме того, могут применяться и в других местах.

Конструкция INSERT … EXECUTE очень похожа на конструкцию INSERT … SELECT и таблица-источник тоже является результирующим набором, сформированным в результате SELECT-запроса. Отличие в том, что SELECT-запрос выполняется в рамках хранимой процедуры.

В первом приближении хранимая процедура – объект БД, представляющий собой поименованный код, написанный на T-SQL, который может принимать параметры, возвращать параметры и формировать результирующие наборы.

СекцияOUTPUT

Секция OUTPUT может использоваться во всех трех DML-операторах, изменяющих БД: INSERT, DELETE, UPDATE. Главной особенностью этой секции является то, что на ее работу не распространяется действие процедуры отката (ROLLBACK), устраняющей все изменения БД, выполненные DML-операторами в рамках транзакции в случае ее неуспешного завершения.

В рамках оператора INSERT секция OUTPUT позволяет получить и записать данные из INSERT-оператора в один из двух источников: в таблицу БД или в результирующий набор.

Оператор DELETE относится к группе операторов DML SQL и предназначен для удаления строк из таблицы.

Структура оператора DELETE.

Синтаксис DELETE предусматривает две секции FROM (обязательной является только первая) и необязательную секцию WHERE. Кроме того, допускается применять секцию OUTPUT и опцию TOP.

Во второй секции FROM может быть указано JOIN-выражение для внутреннего соединения целевой таблицы с другими таблицами. Удалению подлежат только те строки целевой таблицы, для которых выполнилось соединение.

В секции WHERE может быть указано логическое выражение, фильтрующее строки, выбранные второй секцией FROM, подлежащие удалению.

Оператор UPDATE относится к группе операторов DML SQL и предназначен для изменения строк таблицы. Структуру оператора UPDATE.

Синтаксис UPDATE предусматривает обязательную секцию SET и две необязательные секции FROM и WHERE. Кроме того, допускается применять секцию OUTPUT и опцию TOP.

Логика действия оператора UPDATE похожа на логику оператора DELETE. Оператор UPDATE может изменить ноль, одну или более строк в одной таблице (в целевой таблице), имя которой указывается сразу за ключевым словом UPDATE.

В секции FROM может быть указано JOIN-выражение для внутреннего соединения целевой таблицы с другими таблицами. Изменению подлежат только те строки целевой таблицы, для которых выполнилось соединение. Если секции FROM нет, то предполагается, что в этой секции выбираются все строки таблицы.

В секции WHERE может быть указано логическое выражение, фильтрующее строки, выбранные секцией FROM и подлежащие изменению.

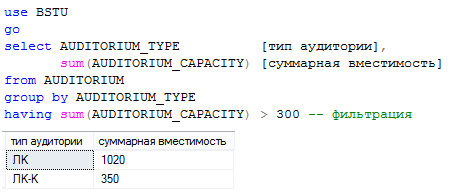
**14.Группировка в операторе SELECT, применение секции HAVING**

Основное назначение секции SELECT – указать порядок и имена столбцов результирующего набора данных.

Секция WHERE применяется для отбора (фильтрации) строк, сформированных в секции FROM. Как правило, секция содержит логическое выражение, которое вычисляется для каждой строки. В том случае, если выражение принимает истинное значение – строка отбирается. Логическое выражение может быть достаточно сложным и содержать в себе другой SELECT-запрос. Говорят, что секция WHERE определяет горизонталь результирующего набора.

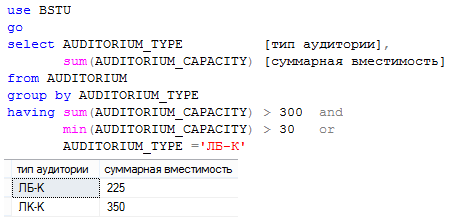
Секция HAVING имеет такое же назначение, что и WHERE, но применяется после выполнения секции GROUP BY.

Для фильтрации результирующего набора строк, сформированного секцией GROUP BY, служит секция HAVING. Представлен SELECT-запрос применяющий секцию GROUP BY и дополненный секцией HAVING.



Смысл секции HAVING такой же, как и у секции WHERE. Основное отличие – действие секции HAVING распространяется на результат выполнения секции GROUP BY, а действие секции WHERE – на результат секции FROM.

Логическое выражение, указанное в HAVING, вычисляется для каждой строки результирующего набора, сформированного секцией GROUP BY. Строка отбирается в результирующий набор секции, если логическое выражение принимает значение «истина». При этом следует помнить, что в логическом выражении могут применяться лишь такие компоненты, которые основываются на выражениях из секции GROUP BY и/или значениях, которые могут быть вычислены. В HAVING можно использовать выражения, построенные над значением из столбца AUDITORIUM\_TYPE, и/или агрегатные функции. В случае, отфильтровываются такие группы строк, в которых сумма значений в столбце AUDITORIUM\_CAPACITY превышает 300.

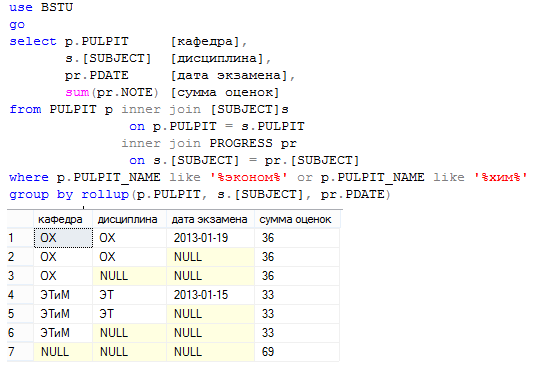


Следует обратить внимание на то, что в логическом выражении HAVING могут использоваться выражения, которых нет в SELECT-списке, как, например, это сделано в запросе: применяется агрегатная функция MIN.

**15.Использование в запросах GROUP WITH CUBE, GROUP WITH ROLLUP**

Конструкции ROLLUP и CUBE применяются в секции GROUP BY и служат для вычисления агрегатных значений над подмножествами строк.

Представлен пример использования конструкции ROLLUP.



Сформулируем правило формирования результирующего набора SELECT-запроса, применяющего в секции GROUP BY конструкцию ROLLUP.

1. Результирующий набор содержит *n* + 1 групп строк, где *n* – количество выражений для группировки, указанных в скобках за ключевым словом ROLLUP.
2. Первая группа строк является результатом выполнения группировки по всем *n* выражениям.
3. Вторая группа строк является результатом группировки первой группы строк по *n* – 1 первым выражениям. Причем столбцы, по которым не выполнялась группировка, заполняются значениями NULL.
4. Группа строк k является группировкой группы строк, полученной на предыдущем этапе по *n* – *k* + 1 первым выражениям.
5. Последняя (*n* + 1)-я группа содержит одну строку. Значения во всех столбцах, соответствующих выражениям в ROLLUP, равны NULL. Значение в столбце, соответствующем агрегатной функции, вычисляется по всем строкам исходного для группировки набора строк.

Представлен запрос и сформированный им результирующий набор строк. В секции GROUP BY запроса применяется конструкция CUBE.

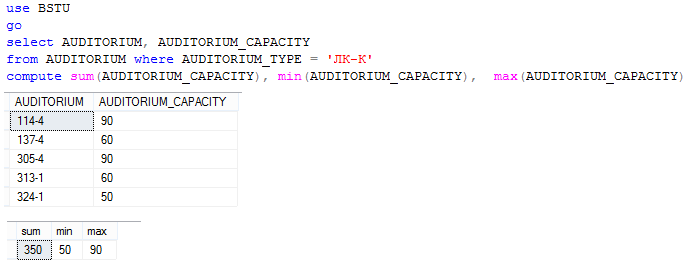
Сформулируем правило формирования результирующего набора SELECT-запроса, применяющего в секции GROUP BY конструкцию CUBE.

1. Формируется множество всех подмножеств выражений, указанных в CUBE-списке.
2. Для каждого непустого подмножества, сформированного в п. 1 выполняется группировка. Если количество элементов подмножества меньше количества элементов CUBE-списка, то соответствующие значения в строках заполняются NULL (как это делалось конструкцией ROLLUP). Сформированные строки помещаются в результирующий набор секции GROUP BY.
3. Для пустого подмножества, сформированного в п. 1 выполняется группировка аналогичная той, что выполнялась в п. 5 правил для ROLLUP.

**16. Применение в запросах COMPUTE, COMPUTE BY.**

Секции COMPUTE и COMPUTE BY предназначены для формирования дополнительных результирующих наборов с итоговыми строками. Применение этих секций возможно только для БД, управляемых Microsoft SQL Server, так как они предусмотрены стандартами SQL.

Представлен пример использования секции COMPUTE. Исходным является результирующий набор, сформированный в любой завершающей секции SELECT-запроса. Секция COMPUTE формирует дополнительный результирующий набор строк, состоящий только из одной строки, столбцы которой содержат итоговые значения. Для вычисления итоговых значений допускается применение только агрегатных функций и не допускается использование псевдонимов столбцов. При выполнении секции формируются столбцы, имена которых задаются автоматически и соответствуют агрегатным функциям, используемым в секции COMPUTE.



Допускается использование нескольких секций COMPUTE для одного SELECT-запроса. Каждой секции COMPUTE соответствует свой отдельный результирующий набор строк.

Применение секции COMPUTE BY несколько отличается от применения COMPUTE. Назначение этой секции – вычисление промежуточных итогов. Секция может быть применена только совместно с секцией ORDER BY. Причем за ключевым словом BY секции COMPUTE BY может быть записана такая же последовательность выражений (ASC и DESC не указывается), как в секции ORDER BY. Если в ORDER BY через запятую указано несколько выражений, то в COMPUTE BY допускается указывать только первые слева, без пропуска.

На рис. 7.115 приведен пример SELECT-запроса с применением секции COMPUTE BY и первые четыре результирующих набора, сформированных этим запросом. При использовании COMPUTE BY формируется несколько результирующих наборов, каждый из которых можно отнести к одному из двух типов.

Первый тип – это набор строк, структура которых формируется на основе SELECT-списка. В набор первого типа включаются строки, имеющие одно и то же значение выражений, указанных за ключевым словом BY секции COMPUTE BY.

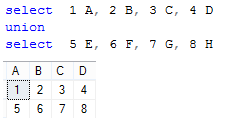
Второй тип – это набор, состоящий из одной строки, сформированной по тому же принципу, что и при выполнении секции COMPUTE без BY. Агрегатные функции вычисляются только над строками предшествующего набора первого типа.

В целом результат выполнения COMPUTE BY формирует последовательность чередующихся результирующих наборов первого и второго типов.

**17.SQL-команды: UNION, INTERSECT, EXCEPT.**

Оператор UNION позволяет объединить результирующие наборы, сформированные двумя SELECT-запросами в общий результирующий набор. Единственным требованием, предъявляемым к SELECT-запросам, является совместимость типов соответствующих столбцов сформированных результирующих наборов.

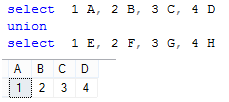
Приведен простейший пример использования оператора UNION для объединения результирующих наборов двух SELECT-запросов.



Каждый оператор SELECT формирует результирующий набор, состоящий из одной строки. Заметим следующее.

1. Результирующий набор оператора UNION содержит объединение строк результирующих наборов первого и второго SELECT-запроса.
2. Имена столбцов результирующего набора оператора UNION совпадают с именами столбцов результирующего набора, сформированного первым SELECT-запросом.
3. Количество столбцов результирующих наборов первого и второго SELECT-запросов должно быть одинаковое.
4. Типы соответствующих столбцов должны быть совместимы.
5. Оператор UNION является коммутативным: результат не зависит от порядка объединяемых наборов.

Оператор UNION выполняет теоретико-множественную операцию объединения, т. е. результатом является множество строк, в котором строки не могут повторяться. Обратите внимание на то, что результатом объединения является только одна строка.

****

Если требуется механическое объединение строк, можно применить оператор UNION ALL.

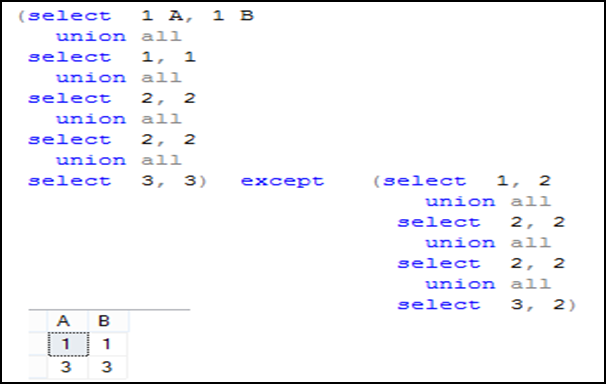
Результатом оператора INTERSECT является набор строк, представляющий собой пересечение двух исходных наборов строк. Как и для оператора UNION, к SELECT-запросам предъявляется требование совместимости типов соответствующих столбцов.

Заметим следующее.

1. В исходных наборах строк должно быть одинаковое количество столбцов и совместимый тип соответствующих столбцов.
2. Результирующий набор оператора INTERSECT содержит множество строк, являющееся пересечением двух исходных множеств строк. Другими словами, результирующий набор оператора INTERSECT содержит строки входящие одновременно и в первый во второй исходные наборы данных.
3. Результатом оператораINTERSECTявляется множество неповторяющихся строк. В этом смысле, оператором выполняется теоретико-множественная операция пересечения множеств.
4. Имена и типы столбцов результирующего набора оператора INTERSECTсовпадают с именами и типами столбцов первого результирующего набора.
5. Оператор INTERSECT является коммутативным: результат не зависит от порядка исходных наборов.

Оператор INTERSECT имеет более высокий приоритет, чем оператор UNION.

Результатом оператора EXCEPT является набор строк, представляющий собой разность двух исходных наборов строк. Как и для операторов UNION и INTERSECT к SELECT-запросам предъявляется требование совместимости типов соответствующих столбцов.



Заметим следующее.

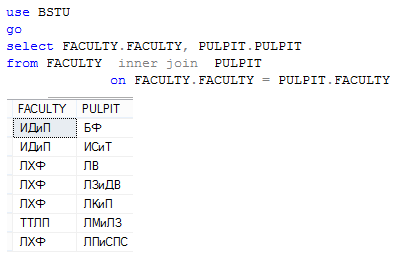
1. В исходных наборах строк должно быть одинаковое количество столбцов и совместимый тип соответствующих столбцов.
2. Результирующий набор оператора EXCEPT содержит множество строк, являющееся разностью двух исходных множеств строк. Другими словами, результирующий набор оператора EXCEPT содержит строки, которые входят в первый набор строк, но не входят во второй.
3. Результатом оператора EXCEPT является множество неповторяющихся строк. В этом смысле оператором выполняется теоретико-множественная операция вычитания множеств.
4. Имена и типы столбцов результирующего набора оператора EXCEPT совпадают с именами и типами столбцов первого результирующего набора.
5. Оператор EXCEPT не является коммутативным: результат зависит от порядка исходных наборов.

Изменение порядка исходных наборов строк приводит к другому результату оператора EXCEPT.

Наивысшим приоритетом обладает оператор INTERSECT, а операторы UNION и EXCEPT имеют одинаковый приоритет.

**18. Внутреннее строение таблиц INNER JOIN, многотабличные запросы.**

Соединение INNER JOIN наиболее часто используемый вид соединения реляционных таблиц.

выражение INNER JOIN, указываемое между двумя соединяемыми таблицами; логическое выражение, записываемое после ключевого слова ON.

операция INNER JOIN является коммутативной – формируемый результирующий набор не зависит от порядка, в котором указаны таблицы.

Внутреннее соединение можно выполнить для двух и более таблиц.

Допускается применение подзапросов.

С помощью INNER JOIN можно соединять таблицу саму с собой. Обычно такие запросы применяются для получения информации об иерархических связях между строками одной таблицы.

Логическое условие, указываемое после ключевого слова ON, не обязательно должно содержать условие, записываемое в виде равенства – это может любое логическое выражение.

**19. Внешнее соединение таблиц OUTER JOUN (LEFT, RIGHT, FULL, CROSS).**

Внешнее соединение двух таблиц формирует набор строк, состоящий из двух частей:

1)результат внутреннего соединения двух таблиц;

2)строки из двух таблиц, которые не смогли соединиться; если в SELECT-списке указаны столбцы двух таблиц, то значения в столбцах, соответствующих незаполненной (несоединенной) части строки, будут NULL.

Имеется два вида внешнего соединения: **LEFT OUTER JOIN** – левое внешнее соединение и RIGHT **OUTER JOIN** – правое внешнее соединение.

Левое внешнее соединение, в соответствии записано слева от ключевых слов LEFT OUTER JOIN, а правое внешнее соединение – несоединенные строки таблицы, имя которой записано справа от RIGHT OUTER JOIN.

**FULL OUTER JOIN** определяет объединение правого и левого соединения

Рассмотрим запрос для таблиц **Товары** и **Заказы**.

**SELECT \* from dbo.Товары at**

**FULL OUTER JOIN dbo.Заказы aa**

**on aa.Наименование\_товара = at.Наименование**

**Order by aa.Наименование\_товара, at.Наименование**

При использовании соединения **CROSS JOIN** каждая строка одной таблицы соединяется с каждой строкой другой таблицы.

Например, на основании таблиц **Товары** и **Заказы** сформировать перечень товаров с ценой исходной и ценой продажи можно с помощью запроса:

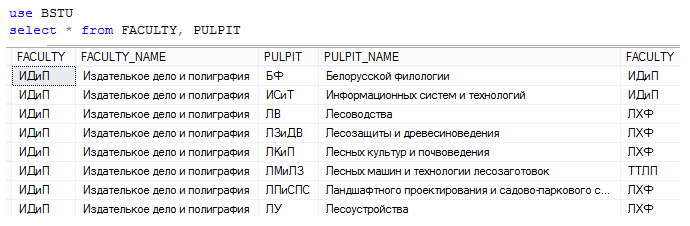
**SELECT Товары.Наименование, Товары.Цена,**

**Заказы.Цена\_продажи From Заказы Cross Join Товары**

**Where Заказы.Наименование\_товара = Товары.Наименование**

***Многотабличные запросы*.** Если секция FROM содержит выражение, включающее несколько таблиц, то говорят о многотабличных запросах.

Простейшего многотабличного запроса и фрагмент сформированного им результирующего набора.



**20. Подзапросы. (NOT) IN, (NOT) EXISTS, ALL, ANY, SOME**

**Подзапрос** – это SELECT-запрос, который выполняется в рамках другого запроса. Подзапросы могут применяться в секции WHERE. Подзапросы бывают двух видов: коррелируемые и независимые. *Коррелируемый* подзапрос зависит от внешнего запроса и выполняется для каждой строки результирующего набора. *Независимый* подзапрос не зависит от внешнего запроса и выполняется только один раз, но результат его выполнения подставляется в каждую строку результирующего набора. В SELECT-списке допускается применять только такие подзапросы, которые формируют скалярный результирующий набор (набор, состоящий из одной строки и одного столбца).

NOT – это отрицание. Операция **IN** формирует логическое значение «истина» в том случае, если значение, указанное слева от ключевого слова IN равно хотя бы одному из значений списка, указанного справа. Операция IN может применяться совместно с другими логическими операциями.

пример SELECT-запроса, в котором список для операции IN формируется как результат выполнения некоррелированного подзапроса. Обратите внимание на то, что в этом случае заменить операцию IN несколькими сравнениями не получится.

Операция **EXISTS** формирует значение «истина», если результирующий набор подзапроса содержит хотя бы одну строку, в противоположном случае − значение «ложь».

На основании таблиц **Товары** и **Заказы** сформировать перечень товаров из таблицы **Товары**, которые не заказаны покупателями

**SELECT Наименование from Товары**

**Where not exists (select \* from Заказы**

**Where Заказы.Наименование\_товара = Товары.Наименование)**

Операция **ALL** используется совместно с операциями сравнения и тоже предназначена для сравнения со списком значений, который задается как результат выполнения подзапроса

Операцию **ALL** можно применять со всеми операциями сравнения и в сочетании с другими логическими операциями

**ALL** применяется совместно с операцией сравнения «больше».

Операция **>=ALL** формирует истинное значение в том случае, если значение стоящее слева больше или равно каждому значению в списке, указанном справа.

**SELECT Наименование\_товара, Цена\_продажи from Заказы**

**Where Цена\_продажи >=all (select Цена\_продажи from Заказы**

**where Наименование\_товара like 'с%')**

**ANY и SOME** название одной и той же функции.

Операция **>=ANY** формирует истинное значение в том случае, если значение стоящее слева, больше или равно хотя бы одному значению в списке, указанном справа.

**21. Временные таблицы: локальная, глобальная, работа с временными таблицами. Пример.**

**Временный табл** хранять во временной системе данных TEMPDB. Не могут иметь внешнего ключа. Создаются каждый раз заново, хранятся во время сеанса.

**Локальные временные** таблицы имеют имена, начинающиеся с символа #, доступны только создавшему ее пользователю и могут быть удалены с помощью оператора DROP TABLE. Если пользователь временную таблицу не удалил сам, то она удалится автоматически при его отключении.

use P\_Bstu; create table #VRTABL( NAME nvarchar(100), BDAY date);insert into #VRTABL(NAME, BDAY)values ('Петров Игорь Васильевич,'11.03.1995') select\*from #VRTABL;

drop table #VRTABL;

Следует обратить внимание: несмотря на то, что временная таблица **#VRTABL** создается в контексте БД **P\_Bstu** (оператор USE), размещена она будет в системной базе данных **TEMPDB**. Как правило, локальные временные таблицы применяются для временного хранения результатов трудоемких SELECT-запросов. **Глобальные временные** таблицы имеют имена, начинающиеся с символа ##, доступны всем пользователям, подключенным к серверу, и могут быть удалены с помощью оператора DROP TABLE. Если глобальная временная таблица не удалена одним из пользователей, то она удалится автоматически при отключении всех пользователей, которые работали с этой таблицей. Если таблица использовалась только создавшим ее пользователем, то она будет удалена сразу после его отключения. Обычно глобальные временные таблицы применяются для обмена данными между несколькими сеансами.

create table ##VR ( Adress nvarchar(100), telefon nvarchar(100));insert into ##VR (Adress, telefon)values (“Ершова 15”,'25658974) select\*from #VR; drop table ##VR;

**22. Представления: создание, применение, WITH CHECK OPTION, DML-операция, ORDER BY.**

Представление– это объект БД, представляющий собой поименованный SELECT-запрос. При этом следует подчеркнуть: в БД хранится именно SELECT-запрос, а не результат его выполнения. Как и любой объект БД, представление создается с помощью оператора CREATE, удаляется с помощью оператора DROP и изменяется с помощью ALTER. При создании представления к SELECT-запросу предъявляются следующие требования: секцию ORDER BY можно использовать только совместно с опцией TOP; не допускается применение секции INTO, COMPUTE и COMPUTE BY; все столбцы результирующего набора должны быть поименованы.

CREATE VIEW [аудитории]

AS SELECT AUDITORIUM [номер],

AUDITORIUM\_NAME [наименование] FROM AUDITORIUM

WHERE AUDITORIUM\_TYPE='ËÊ'

SELECT \* FROM [аудитории]; DROP VIEW [аудитории];

При создании представлений, позволяющих выполнять операции INSERT, DELETE и UPDATE, базовый SELECT-запрос должен удовлетворять правилам:

* запрос не должен содержать секцию группировки GROUP BY;
* запрос не должен применять агрегатные функции, опции DISTINCT и TOP, операторы UNION, INTERSECT и EXCEPT;
* в SELECT-списке запроса не должно быть вычисляемых значений;
* в секции FROM запроса должна указываться только одна таблица.

В приведенном ниже примере представление не удовлетворяет одному из этих правил.

**CREATE VIEW [Сравнение цен] as SELECT zk.Наименование\_товара [Товар],**

**tv.Цена [Исходная цена], zk.Цена\_продажи [Цена продажи] from Заказы zk join Товары tv on zk.Наименование\_товара = tv.Наименование;**

Имя представления может содержать параметры, которые заключаются в скобки и отображаются в первой строке результирующего набора.

**CREATE VIEW Дорогие\_товары (Товар, Цена, Количество)as select Наименование, Цена, Количество from Товары where Цена>200;**

**go SELECT \* from Дорогие\_товары**

Операторы INSERT осуществят вставку строк с новой информацией:

**INSERT Дорогие\_товары values('Диван', 300, 3) INSERT Дорогие\_товары values('Шкаф', 150, 7)**

Чтобы операция вставки не могла осуществиться в том случае, когда информация не удовлетворяет условию, записанному в секции Where, то следует создавать представление с опцией WITH CHECK OPTION.

Например, можно изменить представление **Дорогие\_товары**:

**ALTER VIEW Дорогие\_товары (Товар, Цена, Количество) as select Наименование, Цена, Количество from Товары where Цена>200 WITH CHECK OPTION; go**

Тогда оператор INSERT не выполнится, поскольку цена не удовлетворяет нужному условию секции Where.  **INSERT Дорогие\_товары values('Стол', 80, 9)**

Поскольку секцию ORDER BY можно использовать только совместно с опцией TOP, то представление может выглядеть следующим образом: **CREATE VIEW Дорогие\_товары (Товар, Цена, Количество) as select TOP 150 Наименование, Цена, Количество from Товары ORDER BY Наименование;**

**23. Агрегатные функции: MIN, MAX, SUM, COUNT, AVG.**

**AVG** (вычисление среднего значения), **COUNT** (вычисление количества строк), **MAX** (вычисление максимального значения), **MIN** (вычисление минимального значения), **SUM** (вычисление суммы значений). Определить минимальную и максимальную цены продажи товаров, количество заказанных товаров и общую сумму заказов:

**SELECT min(Цена\_продажи) [Минимальная цена], max(Цена\_продажи) [Максимальная цена], count(\*) [Количество товаров],**

**sum(Цена\_продажи) [Заказы на общую сумму] From Заказы**

**24. Transact-SQL: DECLARE, локальные переменные, инициализация, глобальные переменные, оператор PRINT.**

Оператор DECLARE предназначен для объявления используемых в программе переменных. Для каждой переменной указывается имя и тип. Имя должно обязательно начинаться с символа **@**, но не c **@@** , так как это сочетание символов зарезервировано для системных переменных и функций/ Переменная может быть проинициализирована в операторе DECLARE с помощью символа **=**. В одном сценарии допускается несколько операторов DECLARE.

Жизненный цикл всех объявленных переменных заканчивается вместе с завершением пакета, в котором находится соответствующий оператор DECLARE. *Вывод данных* в T-SQL возможен двумя способами: оператором SELECT можно сформировать выходной результирующий набор и с помощью оператора PRINT можно вывести строку в стандартный выходной поток. Если одновременно выводятся данные, сформированные оператором SELECT и данные, сформированные оператором PRINT, то посмотреть последние можно на вкладке Messages.

**DECLARE @d numeric(5,2) = 4.7, @a char(2), @f float(4)=1;**

**SET @a = 'РБ'; SET @f = 11.4+@f;print 'd= ' +cast(@d as varchar(10));**

**print 'a= ' +cast(@a as varchar(10));print 'f= ' +cast(@f as varchar(10));**

Функция CAST используется для преобразования типов.

**DECLARE @Количество int = (select count(\*) from Заказы) print 'Количество : ' + cast (@Количество as varchar(10));**

25. **Transact-SQL: оператор IF-ELSE, операторные скобки BEGIN/END, оператор RETURN, конкатенация строк, преобразование типов CAST**.

Язык SQL – является декларативным языком. В нем отсутствуют инструкции, позволяющие управлять последовательностью выполнения операторов.

Оператор ветвления **IF** позволяет в зависимости от значения логического выражения выполнять или не выполнять оператор или блок0 операторов сценария T-SQL. За логическим выражением оператора IF располагается оператор, выполняющийся в случае истинности значения логического выражения. Если при этом требуется выполнить оператор при альтернативном значении логического выражения, следует применить конструкцию IF ELSE. Сократить количество вычислений значений логических выражений позволяет конструкция IF ELSE IF, в которой проверка осуществляется только в ветке ELSE оператора IF. Заметим, что при истинном или альтернативном значении логического выражения оператора IF выполняется только один оператор. Для выполнения нескольких операторов применяются операторные скобки.

С помощью операторных скобок **BEGIN END** можно объединять операторы в группы.

Оператор **RETURN** служит для немедленного завершения работы пакета.

При формировании строк, выводимых оператором PRINT, применяется операция конкатенации (объединения) строк, для обозначения которой используется символ +.

**print 'd= ' +cast(@d as varchar(10));**

Функция CAST используется для преобразования типов.

**DECLARE @Количество int = (select count(\*) from Заказы)**

**print 'Количество : ' + cast (@Количество as varchar(10));**

26. **Transact-SQL: конструкции CASE, WHILE, WAITFOR**.

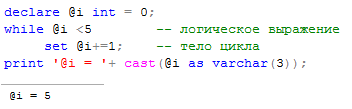
Язык SQL – является декларативным языком. В нем отсутствуют инструкции, позволяющие управлять последовательностью выполнения операторов.

Выражение **CASE**, как правило, применяется в операторе SELECT (в SELECT-списке, секциях WHERE, ORDER BY, HAVING) и служит для формирования одного из нескольких возможных значений. Может применяться в операторах UPDATE (в секциях SET и WHERE), DELETE (в секции WHERE). Выражение CASE имеет два формата: первый сравнивает результат одного выражения со списком заданных значений, второй – вычисляет набор логических выражений.

Сразу за ключевым словом CASE следует вычисляемое выражение, а далее одно или более предложений WHEN, содержащих значения для сравнений. Ключевое слово THEN, присутствующее в каждом предложении WHEN, указывает на один из возможных результатов, формируемых CASE. Список WHEN-предложений завершается предложением ELSE, содержащим результат выражения CASE, формируемого в случае безуспешного сравнения значения исходного выражения со всеми предшествующими WHEN-значениями.

Оператор **WHILE** предназначен для организации программного цикла. Принцип работы оператора WHILE такой же, как в большинстве алгоритмических языков.

Оператор WHILE содержит две составляющие: логическое выражение и тело цикла. Логическое выражение определяет условие выполнения тела цикла. Тело цикла содержит один или более операторов, которые выполняются в том случае и до тех пор, пока логическое выражение принимает значение «истина».

 В том случае, если тело цикла содержит более, чем один оператор, следует применять операторные скобки BEGIN END.

С помощью оператора **WAITFOR** можно приостановить выполнение пакета. Оператор работает в двух режимах: DELAY и TIME.

В режиме DELAY оператор WAITFOR приостанавливает выполнение пакета на заданный интервал времени.

Оператор WAIFOR в режиме TIME. В этом режиме можно приостановить выполнение потока до наступления заданного значения системного времени на сервере.

**27. Transact-SQL: TRY/CATCH, процедура RAISEERROR**.

Язык SQL – является декларативным языком. В нем отсутствуют инструкции, позволяющие управлять последовательностью выполнения операторов.

Для обработки ошибок выполнения в сценарии T-SQL предусмотрена конструкция, состоящая из двух блоков: TRY и CATCH. Блок TRY содержит код T-SQL, в котором могут возникнуть ошибки, а блок CATCH – код, предназначенный для обработки ошибок.

Ошибка, возникающая в охраняемом коде, приводит к передаче управления в блок обработки ошибок. В блоке CATCH можно использовать несколько системных функций, позволяющих диагностировать возникшую в cохраняемом коде ошибку.

|  |  |
| --- | --- |
| Наименование функции | Функция возвращает |
| ERROR\_NUMBER | Код (целочисленное значение) последней ошибки |
| ERROR\_MESSAGE | Сообщение (символьная строка) об ошибке |
| ERROR\_SEVERITY | Уровень серьезности (целочисленное значение) ошибки |
| ERROR\_STATE | Метку (целочисленное значение) ошибки |
| ERROR\_LINE | Код (целочисленное значение) последней ошибки |
| ERROR\_PROCEDURE | Имя (символьная строка) процедуры или NULL |

Разработчик сценария на языке T-SQL может сам сгенерировать ошибку с помощью специальной инструкции **RAISERROR**. В простейшем случае, при вызове инструкции можно передать три параметра: текстовое сообщение об ошибке, уровень серьезности ошибки и метку.

Инструкция RAISERROR применяется дважды: в первом случае для предупреждения об ошибке при делении на нуль, во втором – об ошибке нарушения ограничения PRIMARY KEY. Обратите внимание: инструкция ведет себя по-разному в зависимости от заданного параметром уровня серьезности ошибки. В первом случае (уровень серьезности 10) выполнение инструкции свелось к выводу в стандартный поток сообщения и выполнение охраняемого кода продолжилось. Во втором (уровень серьезности 11) – управление было передано в блок обработки ошибок.

**28. Индексы: определение, типы, назначение. Кластеризованные и некластеризованные индексы**.

**Индекс** – это объект базы данных, позволяющий **ускорить поиск** в определенной таблице. Как и любой другой объект базы данных, индекс может быть создан с помощью оператора CREATE, модифицирован с помощью ALTER и удален с помощью оператора DROP. Для одной таблицы возможно построение нескольких индексов. Индексы бывают кластеризованные, некластеризованные, уникальные, неуникальные и др.

Обычно **кластеризованные** индексы создаются автоматически при создании таб-лицы. Наличие первичного ключа (ограничение PRIMARY KEY) влечет создание индекса при создании таблицы. Если при этом в свойствах столбца первичного ключа специально не указано ключевое слово NONCLUSTERED, то создается кластеризованный индекс. Такие индексы физически упорядочены в соответствии со значениями индексируемых столбцов. В таблице может быть только один кластеризованный индекс.

С помощью системной процедуры SP\_HELPINDEX можно получить перечень индексов, связанных с заданной таблицей:

use BSTU

exec SP\_HELPINDEX 'AUDITORIUM\_TYPE'

Основное отличие *временных* таблиц от постоянных в том, что они хранятся в системной базе данных TEMPDB и не могут иметь внешние ключи. Локальные временные таблицы имеют имена, начинающиеся с символа # и доступны только пользователю, ее создавшему.

**Некластеризованные** индексы в отличие от кластеризованных никак не влияют на физический порядок строк в таблице. Как и в случае кластеризованных индексов, данные индекса организованы в виде сбалансированного дерева. С таблицей некластеризованный индекс связан указателями на ее строки. Для одной таблицы БД допускается создавать до 1023 некластеризованных индексов.

Индексы, соответствующие ограничению PRIMARY KEY, не обязательно должны быть кластеризованными.

Индексы, созданные по одному столбцу таблицы. Такие индексы называются простыми. MSS допускает создавать индексы по нескольким столбцам – такие индексы называются составными. Составными могут быть кластеризованные и некластеризованные индексы. Например, составному первичному ключу по умолчанию соответствует составной кластерный индекс.

**29. Индексы покрытия, уникальные некластеризованные индексы, фильтруемые некластеризованные индексы**.

**Индекс** – это объект базы данных, позволяющий **ускорить поиск** в определенной таблице. Как и любой другой объект базы данных, индекс может быть создан с помощью оператора CREATE, модифицирован с помощью ALTER и удален с помощью оператора DROP. Для одной таблицы возможно построение нескольких индексов. Индексы бывают кластеризованные, некластеризованные, уникальные, неуникальные и др.

**Индекс покрытия** запроса позволяет включить в состав индексной строки значения одного или нескольких неиндексируемых столбцов. Например, индекс покрытия #EX\_TKEY\_X включает значения столбца CC (ключевое слово INCLUDE):

CREATE index #EX\_TKEY\_X on #EX(TKEY) INCLUDE (CC)

Чтобы оценить процедуры поиска можно посмотреть планы выполнения запроса без применения индексов и с использованием индекса покрытия.

SELECT CC from #EX where TKEY>15000

**Некластеризованные индексы**, как и кластеризованные, могут быть **уникальными**. Например, уникальные индексы создаются автоматически при применении ограничения целостности UNIQUE. Установка такого свойства влечет за собой автоматическое создание некластеризованного уникального индекса с именем, заданным с ключевым словом CONSTRAINT.

В некоторых случаях бывает важным, чтобы часть запросов выполнялась особенно быстро. Если эти запросы основаны на WHERE-фильтрации строк, то может быть эффективным применение **фильтруемых некластеризованных индексов**.

Обратите внимание на ключевое слово WHERE и следующее за ним логическое условие в операторе CREATE INDEX: фильтруемый индекс создается только для строк таблицы #EXPLORE, которые удовлетворяют этому логическому условию. Кроме того, заметим, что WHERE-выражение при создании индекса описывает подмножество строк, являющееся надмножеством для всех подмножеств строк, выбираемых запросами.

**30. Перестройка и реорганизация индексов**.

Индексы таблиц, часто подвергающихся операциям изменения, становятся пористыми, или еще говорят – фрагментированными. Процесс образования неиспользуемых фрагментов памяти называют фрагментацией. Фрагментация индексов снижает эффект от их применения и в некоторых случаях может привести к значительным потерям производительности сервера БД.

Степень фрагментации индекса обычно измеряется в процентах. От фрагментации индекса можно избавиться с помощью простого пересоздания индекса.

В больших БД пересоздание индекса может выполняться очень долго. Кроме того, эта операция требует эксклюзивного использования индексируемой таблицы, что приводит к блокировке выполнения всех остальных запросов к таблице. В реальных условиях это непозволительно.

Для избавления от фрагментации индекса предусмотрены две специальные операции: реорганизация и перестройка индекса. Обе операции могут выполняться в режиме ONLINE, позволяющем не блокировать таблицу на время построения индекса и допускающем реализацию других запросов к этой таблице. Главное отличие между реорганизацией и перестройкой индекса в глубине перестройки индекса.

Реорганизация (REORGANIZE) выполняется значительно быстрее, но после этой операции фрагментация будет убрана только из листовых узлов индексного дерева (на самом нижнем уровне). Поэтому после реорганизации может оставаться некоторая степень фрагментации индекса. Операцию реорганизации индексов рекомендуется выполнять при уровне фрагментации 10–20%. Реорганизация индекса всегда выполняется в режиме ONLINE.

Операция перестройки (REBUILD) затрагивает все узлы дерева, поэтому после ее выполнения степень фрагментации равна нулю. Перестройку индексов рекомендуется выполнять при уровне фрагментации, превышающем 20%. По умолчанию операция перестройки индекса выполняется в режиме OFFLINE и полностью блокирует индексируемую таблицу для доступа других запросов, но с помощью специального параметра может быть установлен режим ONLINE. Исключением являются индексы локальных временных таблиц, их перестройка всегда выполняется в режиме OFFLINE.

**31. Курсоры: определение, общая схема работы, локальные/глобальные, динамические/статические, только для чтения**.

**Курсор** является программной конструкцией, которая служит для хранения результата запроса и дает возможность пользователю обрабатывать строки результирующего набора запись за записью. Курсоры бывают локальные и глобальные (по умолчанию), статические и динамические (по умолчанию). Работа с курсором осуществляется в следующей последовательности: курсор объявляется в операторе DECLARE, открывается с помощью оператора OPEN, с помощью оператора FETCH считывается одна или несколько строк результирующего набора. Результат каждого считывания проверяется с помощью системной функции @@FETCH\_STATUS. Затем курсор закрывается с помощью оператора CLOSE. Если курсор глобальный, то он должен быть освобожден с помощью оператора DEALLOCATE.

*Локальный* курсор может применяться в рамках одного пакета и ресурсы, выделенные ему при объявлении, освобождаются сразу после завершения работы пакета.

*Глобальный* курсор может быть объявлен, открыт и использован в разных пакетах. Выделенные ему при объявлении ресурсы освобождаются только после выполнения оператора DEALLOCATE или при завершении сеанса пользователя.

Открытие *статического* курсора приводит к выгрузке результирующего набора во временную таблицу системной БД **TEMPDB**, и все дальнейшие операции осуществляются с этой таблицей, т. е. после открытия курсора все текущие изменения в исходных таблицах не будут отражаться в курсорном результирующем наборе.

Динамический курсор поддерживает данные в "живом" состоянии, но это требует затрат сетевых и программных ресурсов. При использовании динамических курсоров не создается полная копия исходных данных, а выполняется динамическая выборка из исходных таблиц только при обращении пользователя к тем или иным данным. На время выборки сервер блокирует строки, а все изменения, вносимые пользователем в полный результирующий набор курсора, будут видны в курсоре. Однако если другой пользователь внес изменения уже после выборки данных курсором, то они не отразятся в курсоре.

Курсору может быть установлено свойство READ\_ONLY, запрещающее применение операции CURRENT OF в секции WHERE операторов DELETE и UPDATE. Однако свойство READ\_ONLY не мешает применять эти операторы без использования CURRENT OF.

Следует отметить, что атрибут READ\_ONLY имеет смысл только для динамических и ключевых курсоров, статические курсоры по определению применяются только для чтения.

**32. SCROLL-курсоры: дополнительная навигация со SCROLL-курсорами, конструкция WHERE CURRENT OF**.

По умолчанию для курсора установлен атрибут SCROLL, позволяющий применять оператор FETCH с дополнительными опциями позиционирования.

**Scroll**– тип курсора, в котором мы можем двигаться в любом направлении. Работа со скролл курсорами:

* last-позиция переходит на последнюю строку;
* first-позиция переходит на первую строку;
* absolute 5 – на 5 строку относительно начала;
* absolute -5 – на 5 строку относительно конца;
* relative 5 – на строку, от текущей +5 (вниз на 5);
* relative -5 на строку, от текущей -5 (вверх на 5)
* next- позиция переходит на следующую строку;
* prior-позиция переходит на предыдущую строку.

**Where current ofcursor** — изменение текущей строки, очень быстрая операция. работает быстрее обычного where.

**33. Транзакции: определение, назначение, свойства ACID**.

Транзакция - это механизм базы данных, позволяющий таким образом объединять несколько операторов, изменяющих базу данных, чтобы при выполнении этой совокупности операторов они или все выполнились или все не выполнились.

Основные свойства транзакции: атомарность (операторы изменения БД, включенные в транзакцию, либо выполнятся все, либо не выполнится ни один); согласованность (транзакция должна фиксировать новое согласованное состояние БД); изолированность (отсутствие взаимного влияния параллельных транзакций на результаты их выполнения); долговечность (изменения в БД, выполненные и зафиксированные транзакцией, могут быть отменены только с помощью новой транзакции).

ACID: Atomicity – атомарность, Consistency – согласованность, Isolation – изолированность, Durability – долговечность.

**34. Уровни изоляции транзакций, операторы TCL и их применение**.

Одним из основных свойств транзакции является изолированность. Под изолированностью подразумеваются ограничения на одновременное чтение или изменение данных в параллельно работающих транзакциях.

MSS поддерживает пять уровней изолированности транзакций: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SNAPSHOT и SERIALIZABLE. Для установки уровня изолированности текущей для транзакций в текущем подключении служит инструкция SET TRANSACTION ISOLATION LEVEL.

Обычно уровни изолированности транзакций рассматриваются относительно трех проблем параллельного доступа: неподтвержденное чтение, неповторяющееся чтение, фантомное чтение.

Неподтвержденное чтение. До момента t1 транзакцией B выполняются два оператора: INSERT и UPDATE. Эти операторы изменяют таблицы БД, но до момента времени t2 не фиксируют и не откатывают эти изменения. После момента t1 транзакция A считывает содержимое таблиц, измененных транзакцией B и «видит» измененные или добавленные строки. При этом изменения остаются до момента t2 в неподтвержденном состоянии, т. е. могут быть как зафиксированными, так и отмененными. Другими словами: неподтвержденное чтение фактически свидетельствует об отсутствии изолированности транзакций.

Неповторяющееся чтение. Одна транзакция читает данные несколько раз, а другая изменяет те же данные между двумя операциями чтения в первом процессе. По этой причине данные, прочитанные в различных операциях, будут разными.

Фантомное чтение. Две последовательные операции чтения могут получать различные значения, т. к. дополнительные строки, называемые фантомными, могут добавляться другими транзакциями.

Операторы TCL предназначены для создания транзакций.

TCL SQL включает четыре оператора: BEGIN TRAN, SAVE TRAN, COMMIT TRAN и ROLLBACK TRAN.

Оператор BEGIN TRAN указывает на начало транзакции.

Предполагается, что за оператором BEGIN TRAN будут следовать один или несколько DML-операторов, входящих в транзакцию.

Снизу группу DML-операторов, входящую в транзакцию, ограничивает один из операторов: COMMIT TRAN или ROLLBACK TRAN.

Оператор COMMIT TRAN фиксирует транзакцию – подтверждает ее успешное завершение.

Оператор ROLLBACK TRAN откатывает транзакцию – отменяет все изменения, произведенные DML-операторами в таблицах БД в рамках данной транзакции.

Оператор SAVE TRAN применяется для формирования контрольной точки – промежуточного состояния транзакции, к которому может быть осуществлен откат. В тексте SQL-скрипта оператор SAVE TRAN должен находиться между операторами BEGIN TRAN и ROLLBACK TRAN.

**35.Хранимые процедуры T-SQL: параметры, создание, вызов, возврат значения, системные процедуры.**

**Хранимая процедура** – это объект базы данных, представляющий собой поименованный код T-SQL. Хранимая процедура может быть создана с помощью CREATE, изменена с помощью ALTER и удалена с помощью оператора DROP. Процедура может принимать входные и формировать выходные параметры. Результатом ее выполнения может быть целочисленное значение, которое возвращается к точке вызова оператором RETURN, либо один или более результирующих наборов, сформированных операторами SELECT, либо содержимое стандартного выходного потока, полученного при выполнении операторов PRINT. Вызов процедуры осуществляется оператором EXECUTE (EXEC).

В хранимых процедурах допускается применение основных DDL и всех DML и TCL-операторов, конструкций TRY/CATCH, курсоров, временных таблиц.

CREATE PROCEDURE PSUBJECT -- Создаём процедуру

@p varchar(20), -- С 2 параметрами: р и с

@c int output

AS BEGIN

DECLARE @k int = (SELECT count(\*) FROM SUBJECT1 );

PRINT 'Параметры: @p='+@p+', @c='+cast(@c as varchar(3));

SELECT \* FROM SUBJECT1 WHERE SUBJECT=@p; -- находим кол строк

SET @c = @@rowcount;

RETURN @k;

END;

DECLARE @k INT =0, @r INT =0, @p VARCHAR(20);

EXEC @k= PSUBJECT @p='ОАиП', @c=@r output; -- вызов процедуры

DROP PROCEDURE PSUBJECT -- Удаление процедуры

Возвращает полное название предмета, кафедру на которой он читается и количество строк в результирующем наборе.

Системная хранимая процедура - Описание

sp\_configure - Используется для вывода или изменения глобальных установок конфигурации сервера

sp\_databases - Используется для получения списка баз данных, имеющихся на сервере или доступных с его помощью

sp\_datatype\_info - Используется для получения сведений о поддерживаемых типах данных

sp\_help - Используется для получения сведений об объекте в базе данных

sp\_helpdb - Используется для получения сведений об указанной базе данных или о всех базах данных, доступных с помощью данного сервера

sp\_helpfile - Используется для получения сведений о физических именах и атрибутах файлов, в которых содержится текущая база данных

sp\_monitor - Выводит статистику Microsoft SQL Server

xp\_msver - Используется для получения сведений о версии сервера и связанных с этим данных.

**36.Функции: типы, параметры, создание, вызов, возврат значения, принципы применения**.

**Функция** – это объект БД, представляющий собой поименованный код T-SQL. Для создания, удаления и изменения функций надо использовать операторы CREATE, DROP и ALTER соответственно. Отличие функций от хранимых процедур в ограничениях, накладываемых на код функции, в форме представления результата работы, а также в способе вызова. В функции не допускается применение DDL-операторов, DML-операторов, изменяющих БД (INSERT, DELETE, UPDATE), конструкций TRY/CATCH, а также использование транзакций.

Результатом выполнения функции является возвращаемое к точке вызова значение. Если функция возвращает единственное значение (число, строка, дата, время и пр.), то она называется *скалярной*. Функция, возвращающая таблицу, называется *табличной*. В зависимости от структуры кода, различают *встроенные* функции и *многооператорные* табличные функции.

**Скалярные функции**

Скалярная функция возвращает единственное значение и принимает любое количество параметров. Допускается возвращать и принимать все типы данных, за исключением TIMESTAMP и устаревших IMAGE, TEXT и NTEXT.

DROP FUNCTION COUNT\_STUDENT;

create function COUNT\_STUDENT(@faculty varchar(20)) returns int

as begin

declare @rc int = 0;

set @rc = (select count(IDSTUDENT)

from STUDENT s inner join GROUPS g

on s.IDGROUP=g.IDGROUP INNER JOIN FACULTY f

ON g.FACULTY =f.FACULTY

where s.IDGROUP = @faculty) ;

return @rc;

end;

declare @f int= dbo.COUNT\_STUDENT('3'); -- вызов функции

print 'Количество студентов= '+cast(@f as varchar(4));

**Встроенные табличные функции**

Встроенные табличные функции возвращают к точке вызова результирующий набор SELECT-запроса и по своим возможностям очень напоминают представления.

create function FFACPUL()

returns table

as return select FACULTY, FACULTY\_NAME from FACULTY

select \* from dbo.FFACPUL();

**Многооператорные табличные функции**

Многооператорные табличные функции возвращают к точке вызова результирующий набор и допускают более сложную логику работы.

**37.Триггеры: типы триггеров, создание, назначение, применение, вложенные и рекурсивные триггеры.**

*Триггер* – это особый вид хранимой процедуры, предназначенной для обработки событий в БД. Поддерживается два типа триггеров: DDL-триггеры и DML-триггеры. Для каждого типа определено свое семейство событий, обработку которых триггер этого типа может выполнять.

*DDL-триггеры уровня сервера*.Триггеры уровня сервера предназначаются, прежде всего, для отслеживания настроек сервера, а также для обеспечения его безопасности. Условно триггеры уровня сервера можно разбить на две группы: триггеры для обработки событий, связанных с созданием, изменением или удалением объектов сервера, и триггеры, обрабатывающие событие подключения к серверу.

Каждый DML-триггер связан с таблицей или представлением и предназначен для обработки одного или нескольких событий, соответствующих трем операторам, изменяющим содержимое таблицы: INSERT, UPDATE и DELETE. MSS поддерживает два типа DML-триггеров: AFTER и INSTEAD OF.

*AFTER-триггеры*.Триггеры типа AFTER исполняются после выполнения оператора, вызвавшего соответствующее событие.

create trigger TR\_TEACHER

on TEACHER after INSERT, DELETE, UPDATE

*INSTEAD OF-триггеры*.Триггеры типа INSTEAD OF исполняются вместо оператора, вызвавшего соответствующее событие.

create trigger TEACH\_INSTEAD\_OF

on TEACHER instead of DELETE

*Удаление DML-триггеров*.Как и DDL-триггеры, DML- триггеры могут быть удалены с помощью оператора DROP TRIGGER

DROP TRIGGER TR\_TEACHER\_INS

Триггер AFTER не вызывает самого себя рекурсивно, если только не установлен параметр базы данных RECURSIVE\_TRIGGERS.

Существует два типа рекурсии.

Прямая рекурсия

Такая рекурсия происходит, когда триггер срабатывает и выполняет действие, вызывающее повторное срабатывание того же триггера.

Косвенная рекурсия

Косвенная рекурсия возникает, когда триггер срабатывает и выполняет действие, которое вызывает срабатывание другого триггера того же типа (AFTER или INSTEAD OF). Второй триггер выполняет действие, вызывающее повторное срабатывание исходного триггера. Другими словами, косвенная рекурсия может возникать, когда триггер INSTEAD OF вызывается второй раз, но лишь после того, как между этими двумя вызовами вызывается другой триггер того же типа INSTEAD OF. Аналогичным образом косвенная рекурсия может возникать, когда триггер AFTER вызывается второй раз, но лишь после того, как между этими двумя вызовами вызывается другой триггер того же типа AFTER.

**38. Использование таблиц INSERTED, DELETED и их применение, последовательность выполнения триггеров.**

Один AFTER-триггер может реагировать на несколько событий. В этом случае после ключевого слова AFTER должны быть перечислены все необходимые события.

Nриггер будет вызван после выполнения одного из операторов INSERT, UPDATE или DELETE.

В рамках выполнения триггера разработчику доступны две псевдотаблицы с именами INSERTED и DELETED. В зависимости от типа события, активизировавшего триггер, содержимое таблиц разное.

Событие INSERT приводит к тому, что в таблицу INSERTED помещаются строки, добавленные оператором INSERT, вызвавшим это событие. При этом таблица DELETED остается пустой.

При событии DELETE в таблицу DELETED копируются удаленные строки, а таблица INSERTED остается пустой.

При изменении строк таблицы с помощью оператора UPDATE заполняются обе псевдотаблицы. При этом INSERTED содержит обновленные версии строк, а таблица DELETED версию строк до их изменения.

*Порядок выполнения DML-триггеров*.Если для таблицы или представления созданы INSTEADOF и AFTER-триггеры, реагирующие на одно и то же событие, то выполниться только INSTEAD OF- триггер.

Для нескольких AFTER-триггеров, реагирующих на одно и то же событие, имеется возможность частично упорядочить их выполнение с помощью системной процедуры SP\_SETTRIGGERORDER.

exec SP\_SETTRIGGERORDER @triggername = 'TR\_TEACHER\_DEL3',

@order='First', @stmttype = 'DELETE';

exec SP\_SETTRIGGERORDER @triggername = 'TR\_TEACHER\_DEL2',

@order='Last', @stmttype = 'DELETE';

**39.Применение XML: конструкции FOR XML AUTO, FOR XML PATH.**

Для преобразования результата SELECT-запроса в формат XML в операторе SELECT применяется секция FOR XML. При этом можно использовать один из четырех режимов: RAW, AUTO, PATH и EXPLICIT.

**Режим RAW**.По умолчанию в режиме RAW в результате SELECT-запроса создается XML-фрагмент, состоящий из последовательности элементов с именем **row**. Каждый элемент **row** соответствует строке результирующего набора, имена его атрибутов совпадают с именами столбцов результирующего набора (из списка SELECT), а значения атрибутов равны их значениям.

select TEACHER.TEACHER\_NAME 'преподоватль'

from TEACHER

where TEACHER.GENDER = 'ж' for xml RAW('наименование'),

root('Список\_преподоватлей'), elements;

**Режим AUTO.** Результат, полученный в режиме AUTO для простых SELECT-запросов, похож на результат, полученный в режиме RAW. Основное отличие – в качестве имени элемента, соответствующего строке исходной таблицы, используется ее имя (рис. 14.9).

Особенность режима AUTO проявляется в многотабличных запросах. В этом случае режим AUTO позволяет построить XML-фрагмент с применением вложенных элементов. Причем порядок вложенности основывается на порядке столбцов (слева направо), указанных в SELECT-списке.

select [студент].IDSTUDENT [номер],

[студент].NAME [имя],

[группа].IDGROUP [номер\_группы]

from STUDENT [студент] join GROUPS [группа]

on [студент].IDGROUP=[группа].IDGROUP

where [группа].YEAR\_FIRST in (2010, 2012)

order by [номер\_группы] for xml AUTO,

root('Список\_студентов'), elements;

**Режим PATH**. Режим PATH позволяет разработчику наиболее полным образом управлять процессом формирования XML-структуры. Каждый столбец конфигурируется независимо с помощью заданного в формате XPATH [7] имени псевдонима этого столбца.

Если псевдонимы задавать традиционным способом, то сформируется XML-фрагмент, состоящий из элементов с именем row, соответствующих строкам результирующего набора. Каждый row-элемент включает в себя все значения столбцов одной строки в элементной форме. При этом элементы будут иметь имена, совпадающие с заданными в SELECT-списке псевдонимами

select [преподователь].TEACHER\_NAME [имя],

[кафедра].PULPIT\_NAME [кафедра],

[факультет].FACULTY\_NAME [факультет]

from TEACHER [преподователь] join PULPIT [кафедра]

on [преподователь].PULPIT = [кафедра].PULPIT join FACULTY [факультет]

on [факультет].FACULTY = [кафедра].FACULTY

where [факультет].FACULTY in('ТТЛП','ЛХФ')

order by [факультет] for xml PATH('преподоатель'),

root('Список\_преподавателей'), elements;

**40.Функция OPENXML, тип XML, коллекции XML-схем и их применение, типизированные XML-столбцы.**

Для преобразования XML-данных в строки таблицы предназначена функция OPENXML. Она принимает три входных параметра: дескриптор, сформированный системной хранимой процедурой с именем SP\_XML\_PREPAREDOCUMENT; выражение XPATH [7] и целое положительное число, определяющее режим работы функции.

Процедура SP\_XML\_PREPAREDOCUMENT должна быть выполнена до SELECT-запроса, применяющего OPENXML. Процедура принимает в качестве входного параметра XML-документ (в формате строки) и возвращает дескриптор, который впоследствии применяется функцией OPENXML.

Выражение XPATH, принимаемое функцией OPENXML в качестве второго параметра, предназначено для выбора требуемых данных из исходного (введенного процедурой SP\_XML\_PREPAREDOCUMENT) XML-документа.

Последний, третий, параметр функции OPENXML указывает на тип преобразования (режим).

В большинстве своем данные, хранимые в таблицах БД, имеют фиксированный формат. Например, дата рождения. Очевидно, что для хранения этой информации можно воспользоваться типом DATE.

В тех случаях, если данные являются составными (состоящими из нескольких компонент), и при этом не имеют фиксированный формат, удобно использовать XML для их структуризации. Например, если в таблице хранится информация о гражданах разных стран, то структура их почтового адреса может значительно отличаться в зависимости от страны проживания. Или, например, информация об образовании людей (учебные заведения, даты начала и завершения обучения, специальности, квалификации и пр.) будут различными у разных людей.

Часто требуется извлечь не полностью хранящуюся в БД XML-структуру, а лишь ее подмножество. Для этого можно воспользоваться методами XML-типа В качестве параметра методы принимают строку, содержащую выражение на языках XPATH и XQUERY, позволяющие либо выделить, либо изменить фрагменты хранящегося в столбце XML-документа.

Сложность XML-данных требует иного подхода к механизму ограничения целостности для этого типа данных. В семействе XML-технологий существует и активно используется технология, основанная на языке XML-Schema.

XML-Schema – это одна из реализаций языка XML, поддерживаемая консорциумом W3C и предназначенная для описания структуры XML-документа. С помощью языка XML-Schema можно описать правила, которым должен подчиниться XML-документ. Файл, содержащий XML-Schema, обычно имеет расширение XSD (XML Schema definition). Большинство современных систем программирования предусматривают встроенные механизмы, позволяющие с помощью заданного XSD-файла проверять на корректность XML-документы. Для знакомства с языком XML-Schema рекомендуются источники [6, 7].

Для хранения документов XML-Schema в БД MSS предусмотрен специальный объект – XML SCHEMA COLLECTION. Каждый такой объект может содержать один или более XML-SCHEMA-документов.

**41.Настройка производительности базы данных**

Улучшение производительности базы данных требует принятия множества решений. Основные из них: где хранить данные и как осуществить доступ к данным.

Производительность определяется по двум критериям

- время реакции

- пропускная способность

Время реакции трактуется как длительность во времени от момента, когда пользователь вводит команду.

Время реакции определяет производительность отдельной транзакции или программы.

Пропускная способность измеряет общую работу системы, вычисляя количество транзакций, которые могут быть обработаны Database Engine в заданный период времени. Обычно пропускная способность измеряется в количестве транзакций в секунду.

Факторы влияющие на производительность

Факторы подразделяются на три критерия:

- приложение БД

- система БД

- системные ресурсы

На производительность приложений БД влияют следующие факторы:

- эффективность кода приложений

- физическое проектирование

Для улучшения общей производительности следует использовать индексы и исключить NOT IN ( плохо взаимодействуют с оптимизаторами, сканируется вся таблица) .

Физическое проектирование. В процессе проектирования БД выбирается конкретные структуры хранения и пути доступа к файлам БД.

Иногда рекомендуют выполнять денормализацию .

Database Engine может значительно влиять на производительность всей системы. Двумя наиболее важными компонентами (механизации) Database Engine которые влияют на производительность является оптимизатор и блокировки.

**42. Основы безопасности: аутентификация; шифрование; авторизация; отслеживание изменений.**

Аутентификация эта концепция безопасности задает процесс проверки полномочий для предотвращения использования системы неавторизованными пользователями. Аутентификация может быть проверена посредством запроса пользователя предоставить, например, следующее:

что-то, что знает пользователь (обычно пароль)

что-нибудь, чем пользователь владеет – магнитная карта или жетон

физические характеристики пользователя, такие как подпись или отпечатки пальцев

Наиболее общий способ подтверждения аутентификации – использование имени и пароля. Эта информация оценивается системой для определения, является ли субъект допустимым пользователем. Этот процесс может быть усилен при использовании шифрования.

Шифрование данных – это процесс кодирования информации таким образом, что она становится непонятной, пока не будет дешифрована для конкретного пользователя. Для шифрования данных могут быть использованы различные методы.

Авторизация – это процесс, который применяется после проверки того, что пользователь соответствует аутентификации. В течение этого процесса система определяет, какие ресурсы может использовать данный пользователь. Иными словами, информация из структурного и системного каталога о конкретной сущности теперь доступна лишь доверенным лицам – тем людям, которые имеют полномочия доступа к данной сущности.

Отслеживание изменений означает, что действия неавторизованных пользователей отслеживаются и документируются на пользовательском компьютере. Этот процесс полезен для защиты системы от пользователей, которые имеют повышенные привилегии. Рассмотрим подробней основные аспекты безопасности данных.

**43. Копирование и восстановление базы данных: причины потерь данных, типы резервного копирования.**

1. Причины потери данных

Можно выделить 5 основных причин потери данных:

Программные ошибки — Возникновение условий, приводящих к аварийному завершению системы. Поскольку такие ошибки основываются на дефектах программной логики, система баз данных не может выполнить восстановление в подобных ситуациях. Поэтому восстановление должен проводить сам программист, выполнив обработку таких исключений.

Ошибки администратора (человеческий фактор) — Случаи, в которых пользователь с большими полномочиями может неумышленно (или умышленно) разрушить данные. Необходимо постараться создать такой режим работы, который сделает подобную ситуацию маловероятной, однако совсем исключать такую возможность нельзя.

Выход из строя компьютера (сбой системы) — Возникает в результате ошибок в оборудовании и программном обеспечении. В этом случае содержимое оперативной памяти компьютера может быть потерянно. В качестве защиты, можно рекомендовать использование резервного сервера, зеркальное отображение баз данных и пр.

Отказ дискового накопителя — Физическое разрушение жесткого диска. Рекомендуется использование технологий RAID для хранения файлов баз данных, кроме того необходимо, чтобы файлы резервных копий хранились на дисковом носителе, отличным от устройства, на котором располагаются файлы баз данных.

Катастрофы (пожар, наводнение, землетрясение) или кража — Обойти эту ситуацию станет возможным, если устройство, содержащее необходимую для восстановления данных информацию, будет храниться отдельно от основного оборудования и не будет потеряно в результате катастроф или краж.

Необходимо постараться создать систему резервного копирования, позволяющую восстановить данные в любой из описанных выше ситуаций.

2. Типы резервного копирования

Существует 2 режима создания резервных копий:

Статическое резервное копирование — режим, при котором в процессе создания копии только одна активная сессия, поддерживаемая системой, является той сессией, которая создает резервную копию. Другие пользовательские процессы во время выполнения копирования недопустимы.

Динамическое резервное копирование — режим, при котором копирование баз данных может выполняться без остановки сервера баз данных, удаления пользователей или даже закрытия файлов. Пользователи могут и не знать, что выполняется процесс резервного копирования.

MS SQL Server поддерживает оба режима создания резервных копий.