**Лабораторная работа 6**

**Индексы. Представления. Триггеры. Процедуры и функции.**

**Цель работы**

Используя данные базы данных из пятой лабораторной работы, выполнить требуемые действия.

**Содержание работы и методические указания к ее выполнению**

1. **Создать индексы таблиц по следующей спецификации:** таблица **Cources** должна быть преобразована в кластеризованную таблицу, строки данных должны храниться по порядку возрастания значений столбца **Id**. Поскольку на сайте организации часто выводятся результаты запросов, отображающих название и краткое описание по коду курса, необходимо оптимизировать поиск курса по коду, для этого требуется создать некластеризованный индекстаблицы **Cources** по столбцу **Code** и добавить в качестве неключевых два столбца **Name**, **ShortDescription.**
2. **Создать представление**, которое будет выводить Фио студента, количество его контрактов и общую сумму заключенных контрактов.
3. **Создать запрос для вывода** первых трех студентов с наибольшим количеством контрактов **(top).**
4. **Создать индексированное представление** StudentCources, включающее фамилию и имя студентов, прослушанные курсы и сумму контракта по каждому из этих курсов.
5. Добавьте скрипт, **создающий пользовательскую функцию**, рассчитывающую суммарную стоимость заключенных студентом контрактов на посещение курсов.
6. **Cоздать триггер**, препятствующий удалению данных из таблицы Contracts, который должен создавать понятную пользователю ошибку, номер удаляемой записи (**ContractId), идентификатор студента ( StudentId)** и откатывать транзакцию.
7. **Оформить в виде процедуры:** Определите квартал (1,2,3,4) для заданной даты.
8. Определите и запишите в отчет **корневой каталог сервера** (root directory), **количество процессоров в системе, тип аутентификации пользователей и максимальное количество пользователей**, поддерживаемых сервером. Изучите остальные свойства MS SQL Server, доступные в этом диалоге (*для сервера в Object Explorer в контекстном меню команда Properties*)
9. **Вывести информацию о текущих именах сервера, учетной записи, имя компьютера и пользователя ба**зы данных в следующем виде:  *Вы вошли на сервер User400-01\SQL2008 как User400-01\User with dbo permissions на компьютере COMP\_01*
10. Определите **количество системных хранимых процедур ( использовать** @@ROWCOUNT)
11. **Определите список ролей сервера (sp\_helpsrvrole).** Используйте способ отображения информации в виде таблицы
12. Использовать системные вьюки **sys.dm\_db\_index\_physical\_stats, sys.indexes, sys.dm\_db\_index\_usage\_stats** и другие, и проанализируйте на сколько удачно вы создали индексы из пункта 1.
13. С помощью wizard-а создайте **Job** для заполнения таблицы **Courses** случайными записями каждые n-минут где **n-номер в журнале**.

*Замечание.* Для управления форматом даты используется следующая команда:

SET DATEFORMAT *Формат*.

Опция *Формат* указывает формат ввода даты и может принимать следующие значения – *mdy*, *dmy*, *ymd*, *myd* и *dym* (где *m*, *d* и *y* – день, месяц и год соответственно).

**Методические указания**

**Представления**

**Представление (View**) является объектом, составляющим логическую структуру любой базы данных. Представление для конечных пользователей выглядит как таблица, однако **при этом не содержит данных,** а лишь **представляет их.** Физически же представляемые данные расположены в различных таблицах базы данных. Представления чаще всего используются полностью аналогично таблицам.

Представление **реализуется в виде сохраненного запроса,** на основе которого и производится выборка из различных таблиц базы данных. Этот запрос запрашивает данные из некоторых, но не обязательно всех, столбцов, выборка которых осуществляется из одной или нескольких таблиц. При выборке данных могут применяться или не применяться (в зависимости от определения представления) критерии, которым должны соответствовать данные, содержащиеся в представлении.

Синтаксическая структура оператора создания представления выглядит следующим образом:

**CREATE VIEW [<schema name>].<view name> [(<column name list>)]**

**[WITH [ENCRYPTION] [, SCHEMABINDING] [, VIEW METADATA]]**

**AS**

**<SELECT statement> WITH CHECK OPTION**

При указании **ИмяПредставления н**еобходимо придерживаться ранее определенных правил именования объектов, также это имя не должно совпадать с именем уже существующей таблицы в базе данных. Параметр **WITH ENCRYPTION**определяет **шифрование кода запроса** и гарантирует, что пользователи не смогут просмотреть и использовать его.

Параметры оператора **SELECT** определяют содержимое представления. Имена полей представления задаются либо с помощью псевдонимов в операторе выборки, либо указываются в параметре **поле.**

Выборка результатов запросов по критериям осуществляется с помощью конструкции **WHERE**. С этой точки зрения представления не отличаются от обычных операторов выборки данных. **В представлении столбец (столбцы), содержащийся в конструкции WHERE, не обязательно должен быть включен в список выборки.**

Пример .

Создадим представление viewname. Выполним представление для вывода только некоторых столбцов.

**CREATE VIEW viewname AS**

**SELECT field1,Field2,Field3,Field4,Field5**

**FROM tablenameWHERE field6 US NULL;**

Вызовем на выполнение представление:

**SELECT field1,Field2,Field3 FROM viewname;**

В запросе выборки может быть указана команда SELECT любой сложности, однако при этом **запрещается использовать раздела** **ORDER BY,** который в дальнейшем можно применить при выборке данных из созданного представления.

**Триггер.**

**Триггер - это хранимая процедура особого типа, вызываемая на выполнение в ответ на определенные события.**

Синтаксис операторов создания триггеров во многом напоминает синтаксис всех прочих операторов CREATE, за одним исключением - **в этом операторе должна быть указана таблица, за которой закрепляется триггер**, поскольку триггер не может применяться отдельно от таблицы. Оператор создания триггера имеет знакомую структуру **CREATE <object type> <object name>,** a также включает в себя такое же определение исполняемого кода, как и во многих других объектах. Введена лишь дополнительная конструкция **ON,** позволяющая указать таблицу, за которой должен быть закреплен триггер, а также задано, когда и при каких условиях должен происходить запуск триггера.

Синтаксис оператора CREATE TRIGGER:

**CREATE TRIGGER <trigger name>**

**ON [<schema name>.]<table or view name>**

**[WITH ENCRYPTION EXECUTE AS <CALLER SELF <user> >]**

**{{**

**{F0RAFTER} <[DELETE] [J [INSERT] [J [UPDATE]>**

**}**

**INSTEAD OF}**

**[WITH APPEND]**

**[NOT FOR REPLICATION]**

**AS**

**< <sql statements> EXTERNAL NAME <assembly method specifier> >**

**Конструкция ON** оператора создания триггера предназначена для указания объекта, к которому применяется создаваемый триггер. Но необходимо учитывать, что если триггер относится к типу AFTER (т.е. для объявления триггера используется ключевое слово FOR или AFTER), то назначением конструкции ON должна быть таблица, поскольку **триггеры AFTER не предназначены для применения с представлениями.**

Триггеры **подразделяются на два основных типа:**

* триггеры языка определения данных (Data Definition Language - DDL) и
* триггеры языка манипулирования данными (Data Manipulation Language - DML).

**Триггеры DDL активизируются в ответ на внесение каких-либо изменений в структуру базы данных теми или другими пользователями (под этим подразумеваются изменения, осуществляемые с помощью операторов CREATE, ALTER, DROP и т.п.).** Триггеры этого типа были впервые введены в версии SQL Server 2005. Чаще всего триггеры DDL используются, если есть необходимость чрезвычайно строго следить за внесением изменений в структуру базы данных и регистрировать хронологию этих изменений.

**Триггеры DML представляют собой фрагменты кода, которые закрепляются за конкретной таблицей или представлением.**

**В отличие от хранимых процедур, при использовании которых необходимо явно вызывать на выполнение определенный код, триггеры вызываются на выполнение автоматически при обнаружении события (событий), связанного с выполнением операций над таблицей, за которой закреплен триггер**. Единственный способ обеспечения вызова триггера состоит в выполнении требуемого действия над таблицей, за которой он закреплен.

**Хранимые процедуры не только вызываются явно, но и отличаются двумя особенностями, которыми не обладают триггеры: принимают параметры и возвращают коды завершения**.

Триггеры не имеют параметров, но в коде триггера может использоваться определенный механизм, позволяющий выяснить, на какие строки должно распространяться действие триггера. Еще одна особенность триггеров состоит в том, что в них допускается применение ключевого слова RETURN, но они не позволяют возвращать конкретные значения кода завершения (дело в том, что явный вызов триггера не предусмотрен, поэтому не определена точка вызова, в которую можно было бы возвратить код завершения).

В языке SQL предусмотрены три типа запросов, предназначенных для внесения изменений в данные. В связи с этим предусмотрены три основных типа триггеров, а также дополнительные типы, создаваемые с учетом одновременного возникновения и совпадения событий, обусловленных выполнением запросов этих типов:

1. **Триггеры INSERT,**
2. **Триггеры DELETE,**
3. **Триггеры UPDATE**
4. **Триггеры, создаваемые с учетом одновременного возникновения и совпадения событий.**

Синтаксис операторов создания триггеров во многом напоминает синтаксис всех прочих операторов CREATE, за одним исключением - **в этом операторе должна быть указана таблица, за которой закрепляется триггер**, поскольку триггер не может применяться отдельно от таблицы. Оператор создания триггера имеет знакомую структуру **CREATE <object type> <object name>,** a также включает в себя такое же определение исполняемого кода, как и во многих других объектах. Введена лишь дополнительная конструкция **ON,** позволяющая указать таблицу, за которой должен быть закреплен триггер, а также задано, когда и при каких условиях должен происходить запуск триггера.

Важно отметить, что **независимо от используемого типа триггера для реализации действий, предусмотренных этим триггером в СУБД SQL Server, используются две рабочие таблицы.**

В одной из этих таблиц хранятся копии всех вставляемых строк (в связи с этим данная таблица именуется **INSERTED**), а во второй хранятся копии всех удаляемых строк (эта таблица именуется **DELETED**). Важно отметить, что **при выполнении действий, предусмотренных в коде триггера INSTEAD OF, создание этих рабочих таблиц происходит до проверки каких-либо ограничений, а при обслуживании триггера FOR эти таблицы создаются после проверки ограничений.**

**Независимо от используемого типа триггера для реализации действий, предусмотренных этим триггером в СУБД SQL Server, используются две рабочие таблицы.**

В одной из этих таблиц хранятся копии всех вставляемых строк (в связи с этим данная таблица именуется **INSERTED**), а во второй хранятся копии всех удаляемых строк (эта таблица именуется **DELETED**). Важно отметить, что **при выполнении действий, предусмотренных в коде триггера INSTEAD OF, создание этих рабочих таблиц происходит до проверки каких-либо ограничений, а при обслуживании триггера FOR эти таблицы создаются после проверки ограничений.**

**Триггер INSERT**

**Код любого триггера, который объявлен с ключевыми словами FOR INSERT, вызывается на выполнение каждый раз, когда кто-либо вставляет новую строку в таблицу, за которой закреплен триггер.**

Для каждой вставляемой строки СУБД SQL Server создает копию этой новой строки и вставляет ее в специальную таблицу, существующую только в области определения данного триггера. Эта таблица называется INSERTED. Наиболее важная особенность таблицы INSERTED состоит в том, что существует только с момента запуска триггера и до момента завершения его выполнения.

**Триггер DELETE**

**Триггеры DELETE действуют во многом аналогично триггерам INSERT**, не считая того, что **связанная с ними таблица INSERTED пуста** (и это очевидно, поскольку в триггерах DELETE осуществляется удаление, а не вставка строк, поэтому отсутствуют строки, копия которых должна находиться в таблице INSERTED). Вместо этого **копия каждой удаленной строки вставляется в другую таблицу, называемую DELETED**. Эта таблица, как и таблица INSERTED, имеет область определения, которая ограничивается исключительно продолжительностью существования триггера.

**Триггер UPDATE**

Триггеры UPDATE имеют свою специфику**. Запуск кода триггера, объявленного с ключевыми словами FOR UPDATE, происходит при каждом внесении изменения в строку, существующую в таблице.**

**А особенность триггера UPDATE состоит в том, что отсутствует такая таблица, как UPDATED.** Вместо этого в СУБД SQL Server операция модификации каждой строки трактуется так, как будто удаляется существующая строка и вставляется полностью новая. Триггер, объявленный с ключевыми словами FOR UPDATE, обслуживается с помощью не одной, а двух специальных таблиц**, INSERTED и DELETED**. Безусловно, в процессе работы количество строк в этих двух таблицах полностью совпадает.

С помощью **ключевого слова AS** в СУБД SQL Server передаются сведения о том, запуск какого кода должен быть осуществлен при вызове триггера

Триггеры **INSTEAD OF, как и триггеры FOR/AFTER, подразделяются на три основные разновидности- INSERT, UPDATE и DELETE**. **Но в отличие от триггеров FOR/AFTER**, предусмотрена возможность **задавать только по одному триггеру INSTEAD OF** каждого типа в расчете на одну таблицу или представление (т.е. по одному триггеру INSTEAD OF для выполнения операций вставки, обновления и удаления.

Пример:

Создать триггер, препятствующий удалению данных из таблицы Contracts, который должен создавать понятную пользователю ошибку и откатывать транзакцию.

CREATE TRIGGER DisableContractDelete ON Contracts

INSTEAD OF DELETE

AS

DECLARE @contractId int

SELECT @contractId = Id FROM deleted

RAISERROR (N'Попыткаудалить Contract сидентификатором %d', 16, 1, @contractId);

ROLLBACK TRANSACTION

GO

**Процедуры**

**Хранимая процедура (Stored procedure)** – это именованный набор операторов Transact-SQL, хранящийся на сервере.

Создание хранимой процедуры заключается в выполнении следующей команды:

**CREATE PROCEDURE|PROC <sproc name>**

**[<parameter name> [<schema>.]<data type> [VARYING]**

**[= <default value>] [OUT[PUT]] [READONLY]**

**[, n...]**

**[WITH**

**RECOMPILE| ENCRYPTION | [EXECUTE AS { CALLER|SELF|OWNER|’<user name>’}]**

**[FOR REPLICATION]**

**AS**

**<code> | EXTERNAL NAME <assembly name>.<assembly class>.<method>**

Имя процедуры должно удовлетворять правилам именования объектов MS SQL Server;

**parameter name** определяет имя параметра (должно начинаться с символа @), который будет использоваться для передачи входных или выходных данных (при указании ключевого слова OUTPUT);

**data type** указывает, к какому типу данных должны относиться значения параметра;

**default value** – позволяет определить значение по умолчанию, если при вызове процедуры параметр был не указан;

опция **READONLY** создает параметр доступный только для чтения, если параметр имеет тип table, то указание READONLY обязательно;

режим **WITH ENCRYPTION** запрещает дальнейший просмотр кода создаваемой хранимой процедуры, шифруя его;

режим **RECOMPILE** указывает, что сервер не кэширует план исполнения процедуры, и процедура компилируется только во время выполнения.

После ключевого слова **AS** следуют или команды Transact-SQL, которые и составляют тело процедуры, или прописывается метод из указанной сборки .Net Framework.

При вызове хранимой процедуры параметры могут быть заданы либо с учетом позиции, либо по имени. В самой вызываемой хранимой процедуре способ, применяемый для передачи параметров, не играет особой роли, поскольку для всех параметров, независимо от способа их передачи в процедуру, используется одинаковый формат объявления.

**Объявление формальных параметров**

Для объявления параметра необходимо задать от двух до четырех фрагментов информации:

* имя;
* тип данных.
* заданное по умолчанию значение;
* обозначение выходного параметра.

При этом применяется следующий синтаксис: @parameter name [AS] data type [= defaultNULL] [VARYING] [OUTPUTOUT]

Правила именования параметров по существу совпадают с правилами именования переменных. На типы данных распространяются также правила, применяемые к переменным.

Одно особое требование, которое следует учитывать при объявлении типа данных, состоит в том, что при объявлении параметра типа **CURSOR** необходимо использовать опции **VARYING и OUTPUT**. Формальный параметр этого типа используется довольно редко, но следует помнить, каковы условия вызова хранимой процедуры с этим параметром.

Различия между объявлениями параметров хранимых процедур и объявлениями переменных обнаруживаются, когда дело касается значений, заданных по умолчанию. Предусмотрено, что **переменные в результате инициализации всегда принимают NULL-значение**, но по отношению к формальным параметрам это правило не применяется. В действительности, **если значение, применяемое по умолчанию, не задается**, то **формальный параметр считается обязательным**, поэтому при вызове хранимой процедуры должно быть задано начальное значение, так как в противном случае активизируется ошибка. Чтобы задать предусмотренное по умолчанию значение, необходимо добавить знак равенства (=) после обозначения типа данных, а затем установить значение, применяемое по умолчанию. Благодаря этому пользователи получают возможность при вызове хранимой процедуры принимать решение о том, следует ли задать другое значение параметра или воспользоваться значением, предусмотренным по умолчанию.

Пример:

В следующем примере показано создание хранимой процедуры, которая возвращает сведения об определенном сотруднике по переданным значениям фамилии и имени сотрудника. Эта процедура принимает только полные соответствия передаваемым параметрам.

**CREATE PROCEDURE HumanResources.uspGetEmployees**

**@LastName nvarchar(50),**

**@FirstName nvarchar(50)**

**AS**

**SET NOCOUNT ON;**

**SELECT FirstName, LastName, Department**

**FROM HumanResources.vEmployeeDepartmentHistory**

**WHERE FirstName = @FirstName AND LastName = @LastName;**

Вызов:

**EXECUTE HumanResources.uspGetEmployees N'Ackerman', N'Pilar';**

**или**

**EXEC HumanResources.uspGetEmployees @LastName = N'Ackerman', @FirstName = N'Pilar';**

**GO**

**-- или**

**EXECUTE HumanResources.uspGetEmployees @FirstName = N'Pilar', @LastName = N'Ackerman';**

**GO**

**Выходные параметры**

Ключевое слово **OUTPUT** для выходного параметра в объявлении хранимой процедуры является обязательным.

При вызове хранимой процедуры ключевое слово **OUTPUT** должно использоваться так же, как и при объявлении этой хранимой процедуры. Благодаря этому СУБД SQL Server получает заблаговременное предупреждение о том, что соответствующий параметр потребует специальной обработки.

**Но следует учитывать, что упущение, в результате которого необходимо ключевое слово OUTPUT не будет задано, не приведет к возникновению ошибки во время выполнения сценария**. Об этом не будет сказано в каких-либо сообщениях, **но значение, предназначенное для формирования выходного параметра, не будет передано в переменную**. В конечном итоге окажется, что получено значение, которое было присвоено выходному параметру по умолчанию, причем с наибольшей вероятностью таковым станет NULL-значение. Таким образом, указанное упущение приводит к тому нарушению в работе программного обеспечения, которое автор считает наиболее опасным, - к получению непредсказуемых результатов.

Переменная, которой присваивается результат вычисления значения выходного параметра, не обязательно должна иметь то же имя, что и внутренний параметр хранимой процедуры.

**Способ использования оператора return**

Любая процедура, вызываемая на выполнение из программы, возвращает значение, независимо от того, предусмотрен ли в ней возврат значения или нет. По умолчанию после успешного завершения процедуры СУБД SQL Server автоматически возвращает значение, равное нулю.

Чтобы передать некоторое возвращаемое значение из хранимой процедуры обратно в вызывающий код, достаточно применить оператор RETURN:

**RETURN [<integer value to return>]**

Обратите внимание на то, что **возвращаемое значение должно быть целочисленным**.

Одной из наиболее важных особенностей оператора RETURN является то, что его **выполнение приводит к безусловному завершению работы** и выходу из хранимой процедуры. Это означает, что, независимо от местонахождения оператора RETURN в коде хранимой процедуре, после его выполнения больше не будет выполнена ни одна строка кода.

**Выполнение процедуры**

Хранимая процедура может быть выполнена с помощью оператора **EXECUTE**:

**EXEC[UTE]**

**[@СтатусВозврата =] ИмяПроцедуры**

**[ [@параметр=] {Значение | Выражение} [OUTPUT] ] [,...]**

При использовании выходного параметра, его необходимо описать с использование ключевого слова OUTPUT при создании процедуры, а также использовать это слово и при указании соответствующего параметра при вызове процедуры. В противном случае процедура не передает выходное значение.

Параметр @СтатусВозврата используется для получения значения кода возврата из хранимой процедуры, выполненного с помощью оператора RETURN Статус. Пользователю рекомендовано использовать положительные значения статуса.

Пример:

В следующем примере выходной параметр используется для возвращения уникального идентификатора вновь добавленного товара.

CREATE PROCEDURE spr\_addProduct

@Description nvarchar(100),@InStock int = 0**,@IdProd int OUT**

AS

INSERT Product([Description], InStock)

VALUES (@Description, @InStock)

SET @IdProd = @@IDENTITY

RETURN

**Пример вызова:**

DECLARE @IdProd int

EXEC spr\_addProduct

@Description = N'Новый товар',

@IdProd = @IdProd OUTPUT

SELECT @IdProd as N'@IdProd'

Обратите внимание на то, что при вызове процедуры переданы значения не для всех параметров. Параметр InStock являются необязательным, поскольку для него указано значение по умолчанию в виде нуля, которое и будет использовано, в случае если для него не будет явно предоставлено другое значение. При этом если бы вызов хранимой процедуры и передача значений происходили с использованием позиционных параметров, то пришлось бы заполнять каждую позицию в списке параметров, по меньшей мере, до того, как встретился бы последний параметр, для которого должно быть предусмотрено значение.

Пример 1:

Создать процедуру для получения всех данных из всех строк таблицы tablename и вызвать ее на выполнение.

**CREATE PROC allstring**

**AS**

**SELECT \* FROM tablename;**

Вызовем на выполнение:

**EXEC allstring;**

Пример 2

Создать хранимую процедуру с шифрованием:

**CREATE PROCEDURE HumanResources.uspEncryptThis**

**WITH ENCRYPTION**

**AS**

**SELECT BusinessEntityID, JobTitle, NationalIDNumber,**

**VacationHours, SickLeaveHours**

**FROM HumanResources.Employee;**

Чтобы убедиться, что исходный текст процедуры недоступен, можно выполнить системную процедуру, выводящую код хранимой процедуры:

**EXEC sp\_helptext 'HumanResources.uspEncryptThis';**

Результатом выполнения будет сообщение:

**The text for object 'HumanResources.uspEncryptThis' is encrypted**

(Текст объекта 'HumanResources.uspEncryptThis' зашифрован).

**Индексы**

Одними из основных требований, предъявляемым к системам управления базами данных, являются возможность представления данных в определенном, отличном от физического, порядке и возможность быстрого поиска определенной записи. Эффективным средством решения этих задач является использование **индексов.**

**Индекс** **представляет собой таблицу**, которая содержит ключевые значения для каждой записи в таблице данных, записанные в порядке, требуемом для пользователя. Ключевые значения определяются **на основе одного или нескольких полей** таблицы. Кроме того, индекс содержит уникальные ссылки на соответствующие записи в таблице.

Каждая таблица может иметь несколько различных индексов, каждый из которых определяет свой собственный порядок следования записей

**Индекс** – **это объект, который существует только в пределах инфраструктуры конкретной таблицы или представления**

Формально считается, что в СУБД SQL Server предусмотрены **два типа индексов** (**кластеризованный и некластеризованный**), но по своему внутреннему устройству индексы SQL Server подразделяются на три типа, указанных ниже.

**Кластеризованные индексы.**

**Некластеризованные индексы**, которые подразделяются на следующие типы:

□ некластеризованные индексы, заданные на неупорядоченной таблице;

□ некластеризованные индексы, заданные на кластеризованном индексе.

В кластеризованных и некластеризованных индексах применяются разные способы физического хранения данных. Кроме того, все три типа индексов отличаются друг от друга тем, что в них СУБД SQL Server по-разному осуществляет переход по В дереву для достижения конечных данных

**Контрольные вопросы:**

1. Что представляет собой индекс?
2. Можно ли создать индекс на основе нескольких полей таблицы?
3. Может ли таблица иметь несколько различных индексов?
4. Чем различаются простой и составной индекс?
5. Может ли индекс существовать без таблицы?
6. Какие типы индексов предусмотрены в СУБД SQL Server?
7. Сколько кластеризованных индексов может быть задано на одной таблице?
8. Сколько некластеризованных индексов может быть задано на одной таблице
9. Чем отличаются кластеризованный и некластеризованный индексы?
10. Что представляет собой кластеризированный ключ?
11. Какая таблица является неупорядоченной?
12. Что такое представление?
13. Какие типы триггеров Вы знаете?
14. Чем отличается триггер от процедуоы?