



*Программирование
и администрирование
СУБД*

Microsoft®
SQL Server

Урок №8

Содержание

| | |
|--|-----------|
| 1. Пользовательские функции | 3 |
| 2. Курсоры в SQL Server | 15 |
| 2.1. Характеристики и типы курсоров | 15 |
| 2.2. Создание курсоров..... | 21 |
| 2.3. Манипулирование записями с помощью курсоров | 27 |
| 2.4. Модификация и удаление данных | 35 |
| 3. Использование XML в SQL Server 2008 | 39 |
| 3.1. Тип данных XML. Коллекция схем XML | 39 |
| 3.2. Выборка и изменение XML-данных..... | 48 |
| 3.3. Выборка реляционных данных в формате XML. Конструкция FOR XML..... | 55 |
| 4. Домашнее задание | 68 |

1. Пользовательские функции

В SQL Server существует большой набор стандартных функций, которыми вы уже не раз пользовались. И этого набора для обеспечения функциональности базы данных иногда может быть недостаточно. Поэтому SQL Server предоставляет возможность создавать свои собственные пользовательские функции (user-defined functions).

Типы пользовательских функций:

1. скалярные (scalar functions) – это функции, которые возвращают одно скалярное значение, то есть число, строка и т.п.
2. встроенные однотабличные или подставляемые табличные (inline table-valued functions) – это функции, которые возвращают результат в виде таблицы. Возвращаются они одним оператором SELECT. Причем, если в результате создается таблица, то имена ее полей являются псевдонимами полей при выборке данных.
3. многооператорные функции (multistatement table-valued functions) – это функции, при определении которой задаются новые имена полей и типы.

Кроме того функции разделяют по детерминизму. Детерминизм функции определяется постоянством ее результатов. Функция является детерминированной (deterministic function), если при одном и том же заданном входном значении она всегда возвращает один

и тот же результат. Например, встроенная функция DATEADD () является детерминированной, поскольку добавление трех дней к дате 5 мая 2010 г. Всегда дает дату 8 мая 2010, или функция COS(), которая возвращает косинус указанного угла.

Функция является недетерминированной (non-deterministic function), если она может возвращать разные значения при одном и том же заданном входном значении. Например, встроенная функция GETDATE() является недетерминированной, поскольку при каждом вызове она возвращает разные значения.

Детерминизм пользовательской функции не зависит от того, является она скалярной или табличной – функции обоих этих типов могут быть как детерминированными, так и недетерминированными.

От детерминированности функции зависит, можно ли проиндексировать ее результат, а также можно ли определить кластеризованный индекс на представление, которое ссылается на эту функцию. Например, недетерминированные функции не могут быть использованы для создания индексов или расчетных полей. Что касается кластеризованного индекса, то он не может быть создан для представления, если оно обращается к недетерминированной функции (независимо от того, используется она в индексе или нет).

Пользовательские функции имеют ряд преимуществ, среди которых основным является повышение производительности выполнения, поскольку функции, как и хранимые процедуры, кэшируют код и повторно используют план выполнения.

Итак, рассмотрим типы функций по порядку. Начнем со скалярных. Синтаксис их объявления следующий:

```
CREATE FUNCTION [ схема. ] имя_функции
    ( [ @параметр [ AS ] [ схема. ] тип
      [ = значение_по_умолчанию | default ]
      [ READONLY ]
      ] [, ... n]
    )
RETURNS тип_возвращаемого_значения
[ WITH { [ ENCRYPTION ]
        | [ SCHEMABINDING ]
        | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
        | [ EXECUTE AS контекст_безопасности]
      }
  [ , ...n ]
]
[ AS ]
BEGIN
    тело_функции
RETURN (возвращаемое_скалярное_значение)
END
```

Как видно из синтаксиса, после имени функции, необходимо в скобках перечислить необходимые входные параметры, если они предусмотрены. Поскольку все параметры являются локальными переменными, то перед именем необходимо ставить символ @, после чего указывается его тип (*допускаются все типы данных, включая типы данных CLR, кроме timestamp (!)*). В случае необходимости можно задать значение по умолчанию для каждого из входных параметров или указать ключевое слово default. Если определено значение default, тогда параметру присваивается значение по умолчанию для данного типа.

Ключевое слово *READONLY* указывает на то, что параметр не может быть обновлен или изменен при определении функции. Заметим, что если тип параметра является пользовательским табличным типом, тогда обязательно указать ключевое слово *READONLY*.

Далее следует указать тип возвращаемого скалярного значения для функции (*returns*). Допускаются все типы данных, кроме нескаларных типов *cursor* и *table*, а также типы *rowversion* (*timestamp*), *text*, *ntext* или *image*. Их лучше заменить типами *uniqueidentifier* и *binary* (8).

Параметр *WITH* задает дополнительные характеристики для входных аргументов. С ключевым словом *ENCRYPTION*, *SCHEMABINDING* и *EXECUTE AS* вы уже знакомы, поэтому на них останавливаться не будем. Хотя здесь есть несколько замечаний:

- параметр *SCHEMABINDING* нельзя указывать для функций *CLR* и функций, которые ссылаются на псевдонимы типов данных;
- параметр *EXECUTE AS* нельзя указывать для встроенных пользовательских функций.

Параметр *RETURNS / CALLED* может быть представлен одним из двух значений:

- *CALLED ON NULL INPUT* (по умолчанию) означает, что функция выполняется и в случаях, если в качестве аргумента передано значение *NULL*.
- *RETURNS NULL ON NULL INPUT* указанный для функций *CLR* и означает, что функция может вернуть *NULL* значения, если один из аргументов равен *NULL*. При этом код самой функции *SQL Server* не вызывает.

Тело функции размещается в середине блоков BEGIN..END, который обязательно должен содержать оператор RETURN для возврата результата. При написании кода тела функции следует помнить, что здесь существует ряд ограничений, основным из которых является запрет изменять состояние произвольного объекта базы данных или же базу данных.

Кстати, рекурсивные функции также поддерживаются. Допускается до 32 уровней вложенности.

Вызвать скалярную функцию можно одним из двух способов: с помощью оператора *select* или *execute*.

```
SELECT имя_функции (параметр1 [, ... n])
EXEC[UTE] @переменная = имя_функции параметр1 [, ... n]
```

Напишем функцию, которая возвращает день недели по указанной в качестве параметра дате.

```
create function DayOfWeek (@day datetime)
returns nvarchar(15)
as
begin
    declare @wday nvarchar(15)
    if (datename(dw, @day) = 'Monday')
        set @wday = 'понедельник'
    if (datename(dw, @day) = 'Friday')
        set @wday = 'пятница'
    else
        set @wday = 'другой'
    return @wday
end;

-- ВЫЗОВ
select DayOfWeek(GETDATE()) as 'День недели';
```

Результат:

| Results | | Messages | |
|---------|--|------------|--|
| | | День тижня | |
| 1 | | пятница | |

Встроенные табличные функции подчиняются тем же правилам, что и скалярные. Их отличие от последних заключается в том, что они возвращают результат в виде таблицы. Табличные функции являются неплохой альтернативой представлениям и хранимым процедурам. Например, недостатком представлений является то, что они не могут принимать параметры, которые иногда необходимо передать. Хранимые процедуры в свою очередь могут принимать параметры, но не могут быть использованы в выражении *FROM* оператора *SELECT*, что несколько усложняет обработку результатов. Табличные функции решают вышеописанные проблемы.

Итак, синтаксис однотабличной функции выглядит так:

```
CREATE FUNCTION [ схема. ] имя_функции
    ( [ @параметр [ AS ] [ схема. ] тип
      [ = значение_по_умолчанию | default ]
      [ READONLY ]
    ] [, ... n]
  )
  RETURNS TABLE
  [ WITH { [ ENCRYPTION ]
    | [ SCHEMABINDING ]
    | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
    | [ EXECUTE AS контекст_безопасности]
  }
  [ , ...n ]
]
```



```
[ AS ]
BEGIN
    тело_функции
RETURN [ ( ] оператор SELECT[ ) ]
END
```

Вызывается табличная функция следующим образом:

```
select * from имя_функции (параметр1 [, ... n])
```

Например, напомним функцию, которая выводит названия книг и количество магазинов, которые их продают.

```
create function countShops ()
returns table
as
return (select b.NameBook as 'Book Title',
        count(sh.id) as 'Number of Shops'
        from books b, shops sh, sales s
        where s.id_book = b.id and s.id_shop = sh.id
        group by b.NameBook);
go
-- call the function
select * from countShops();
```

Результат:

| | Book Title | Number Of Shops |
|----|---|-----------------|
| 1 | Applied Microsoft .NET Framework Programming | 1 |
| 2 | ASP.Net: The Complete Reference | 1 |
| 3 | CLR via C# | 1 |
| 4 | Java: A Beginner Guide | 1 |
| 5 | Java: The Complete Reference | 2 |
| 6 | Pro Wpf 4.5 in C#: Windows Presentation Foundati... | 1 |
| 7 | Programming ASP.NET Core (Developer Reference) | 1 |
| 8 | Ring Around the Sun | 1 |
| 9 | Swing: A Beginner Guide | 1 |
| 10 | The Art of Computer Programming, vol.1 | 1 |
| 11 | Time is the Simplest Thing | 2 |
| 12 | Windows Runtime via C# | 1 |

Синтаксис многооператорной функции:

```
CREATE FUNCTION [ схема. ] имя_функции
    ( [ @параметр [ AS ] [ схема. ] тип
      [ = значение_по_умолчанию
    | default ]
      [ READONLY ]
    ) [ , ... n ]
    )
RETURNS @возвращаемая_таблица
    TABLE структура_таблицы
[ WITH { [ ENCRYPTION ]
    | [ SCHEMABINDING ]
    | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
    | [ EXECUTE AS контекст_безопасности]
  }
  [ , ...n ]
]
[ AS ]
BEGIN
    тело_функции
RETURN
END
```

Отметим, что в многооператорной функции возвращаемая таблица не обязательно создается оператором `select`. Отсюда происходит и название функции. Здесь можно, например, выполнять предварительную обработку данных и создавать временную таблицу, после чего доработать ее и вернуть новую таблицу в вызывающую программу.

Следует также быть внимательным, поскольку блок тела функции может содержать несколько операторов `SELECT`. В таком случае, в выражении `RETURNS` нужно явно определить таблицу, которая будет возвращаться. Кроме того, поскольку оператор `RETURN`

в многооператорной табличной функции всегда возвращает таблицу, которая задана в RETURNS, то он должен выполняться без аргументов. Например, **RETURN**, а не **RETURN @myTable**.

Вызывается многооператорная табличная функция подобно встроенной табличной, то есть с помощью оператора SELECT:

```
select * from имя_функции (параметр1 [... n])
```

В качестве примера напомним многооператорную табличную функцию, которая возвращает название магазина (-ов), который продал наибольшее количество книг.

Данный процесс можно разделить на **два этапа**:

- **первый** – создадим временную таблицу, которая возвращает название книги и количество магазинов, которые их продают;
- **второй** – получаем магазин, продавший максимальное количество книг.

```
create function bestShops()
returns @tableBestShops table (ShopName varchar(30)
not null, BookCount int not null)
as
begin
-- creating a temporary table that returns the shop name
-- and number of books in the shop
declare @tmpTable table (id_book int not null,
bookNumber int not null)
insert @tmpTable
select b.id as 'Identifier', count(s.id_shop) as 'Number
Of Shops'
from books b, sales s
```

```

where s.id_book=b.id
group by b.id
insert @tableBestShops
select sh.Shop as 'Shop Name',
       'Number Of Books' = max(tt.bookNumber)
from @tmpTable tt, sales s, Shops sh
where tt.id_book=s.id_book and
      s.id_shop = sh.id
group by sh.Shop
return
end;
go
-- call the function
select * from bestShops();

```

Результат:

| | ShopName | BookCount |
|---|-------------|-----------|
| 1 | HashTag | 2 |
| 2 | Rare Books | 2 |
| 3 | Smith&Brown | 2 |

Изменить существующую функцию пользователя можно с помощью оператора **ALTER FUNCTION**:

```

-- скалярная функция
ALTER FUNCTION [ схема. ] имя_функции
    ( [ @параметр [ AS ] [ схема. ] тип
      [ = значение_по_умолчанию | default ]
      [ READONLY ]
    ] [, ... n]
    )

```

```

RETURNS тип_возвращаемого_значения
[ WITH { [ ENCRYPTION ]
        | [ SCHEMABINDING ]
        | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
        | [ EXECUTE AS контекст_безопасности]
      }
  [ ,...n ]
]
[ AS ]
BEGIN
  тело_функции
  RETURN (возвращаемое_скалярное_значение)
END
-- встроенная однотабличная функция
ALTER FUNCTION [ схема. ] имя_функции
              ( [ @параметр [ AS ] [ схема. ] тип
                [ = значение_по_умолчанию | default ]
                [ READONLY ]
              ][ ,... n]
              )
RETURNS TABLE
[ WITH { [ ENCRYPTION ]
        | [ SCHEMABINDING ]
        | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
        | [ EXECUTE AS контекст_безопасности]
      }
  [ ,...n ]
]
[ AS ]
BEGIN
  тело_функции
  RETURN [ ( ] оператор SELECT[ ) ]
END
-- многооператорная функция
ALTER FUNCTION [ схема. ] имя_функции
              ( [ @параметр [ AS ] [ схема. ] тип
                [ = значение_по_умолчанию
| default ]

```

```

        [ READONLY ]
    ] [, ... n]
)
RETURNS @возвращаемая_таблица
    TABLE структура_таблицы
[ WITH { [ ENCRYPTION ]
        | [ SCHEMABINDING ]
        | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
        | [ EXECUTE AS контекст_безопасности]
      }
  [ , ...n ]
]
[ AS ]
BEGIN
    тело_функции
RETURN
END

```

Несмотря на то, что с помощью оператора ALTER FUNCTION можно изменить функцию практически полностью, использовать данный оператор для преобразования скалярной функции в табличную или наоборот запрещено. С помощью ALTER FUNCTION также нельзя превращать функцию T-SQL в функцию среды CLR или наоборот.

Удаление пользовательской функции осуществляется оператором **DROP FUNCTION**:

```

DROP FUNCTION [ схема. ] имя_функции [ , ...n ]

```

Получить список пользовательских функций можно из системного представления sys.sql_modules, а список параметров каждой из них размещается в представлении **sys.parameters**. Список пользовательских функций CLR размещается по другому адресу, а именно в системном представлении **sys.assembly_modules**.

2. Курсоры в SQL Server

2.1. Характеристики и типы курсоров

Одной из характерных особенностей операторов языка SQL как SELECT, UPDATE и DELETE является то, что они работают сразу с множеством записей. С одной стороны, такая обработка полезна, например, при изменении цены товара и позволяет обойтись без многократного вызова одного запроса. Но она не всегда удобна для прикладных клиентских приложений, которые выполняют порядковую обработку данных, а также в ряде случаев, когда необходимо обрабатывать результаты построчно. Для решения таких задач в SQL предусмотрены курсоры.

Курсоры – это временные объекты, которые позволяют организовать построчную обработку набора данных. С их помощью можно в цикле пройти по результирующему набору, отдельно считывая и обрабатывая каждую его строку. В зависимости от назначения создаваемого курсора, вы можете перемещать курсор в середине множества, модифицируя или уничтожая данные. Следует отметить, что необходимость в использовании курсора очень низкая, они ресурсоемкие и поэтому перед их использованием следует убедиться, что курсор действительно обеспечит повышение производительности запроса.

Работа с результирующим множеством, которое помещается в курсор, осуществляется по следующей схеме:

1. Сначала следует объявить курсор. Объявления курсора осуществляется с помощью оператора `DECLARE CURSOR`. Следует отметить, что курсор и связанный с ним результирующий набор должны иметь одинаковые имена. Это имя указывается при объявлении курсора.
2. Открытие курсора для работы, то есть заполнение его данными – оператор `OPEN`. Результирующий набор в курсоре после его открытия остается открытым до тех пор, пока он не будет закрыт явно.
3. Для основной работы с курсором, то есть для перемещения в наборе с одной записи на другой, используются специальные команды:

3.1. Выборка записей с помощью курсора – оператор `FETCH`.

3.2. Для позиционированного обновления используется инструкция `WHERE CURRENT OF` для `UPDATE` или `DELETE`.

4. Закрытие курсора и очистка текущего результирующего набора – оператор `CLOSE`.
5. Высвобождение ресурсов, используемых курсор, и удаление его как объекта – оператор `DEALLOCATE`. Курсоры обладают рядом *характеристик*, к которым можно отнести:
 - область видения – в которых соединениях и процессах может быть получен доступ к курсору.
 - чувствительность курсора – способность отражать изменения в исходных данных.

- прокрутка – способность осуществлять прокрутку как вперед, так и назад во множестве записей. При этом последовательные курсоры (forward only) работают намного быстрее, но имеют меньшую гибкость.
- обновление – способность модифицировать (обновлять) множество записей. Такие курсоры используются только для чтения (read only), они, как правило, более производительные, но менее гибкие.

MS SQL Server поддерживает три *вида курсоров*:

1. Курсоры T-SQL, которые используются внутри триггеров, хранимых процедур и сценариев. Их мы рассмотрим подробнее.
2. Серверные курсоры **API**, которые действуют на сервере и реализуют программный интерфейс прикладных приложений для ODBC, ADO и ADO.NET, OLE DB и DB-Library. Каждое из этих API использует разный синтаксис и отличается по функциональным характеристикам.
3. Клиентские курсоры реализуются на самом клиенте. Они выбирают весь результирующий набор записей с сервера и сохраняют его локально, что позволяет ускорить операции обработки данных за счет снижения затрат времени на выполнение сетевых операций.

Все курсоры можно разделить на **4 типа**:

- а) статические (static cursors) – при его создании делается "снимок" помещаемых в курсор данных. Он не чувствителен к изменениям в структуре или в значениях данных, то есть любые изменения во

входных данных в курсоре отражены не будут. В связи с этим такие типы курсоров используются очень редко, а на уровне клиент-серверных приложений они вообще неуместны. Вместо них очень часто используют таблицы базы данных **tempdb**, в которых хранятся данные курсора, или же создают временные таблицы с помощью оператора **SELECT INTO**. Если все же статический курсор необходим, тогда его открывают только для чтения.

Статические курсоры могут быть последовательными и прокручиваемые.

б) динамические (**dynamic cursors**) – это самый мощный и гибкий тип курсора, но и наиболее ресурсоемкий. Он отражает все изменения, которые вносятся пользователями в базовые таблицы, то есть поддерживает данные в "живом" состоянии.

с) ключевые (**keyset-driven cursors**) – основаны на использовании набора ключей, который определяется данными, которые уникально идентифицируют каждую запись в таблице базы данных. Поэтому в таблице **keyset** базы данных **tempdb** хранятся только наборы ключей, а не весь набор данных. Чтобы объявить ключевой курсор, каждая таблица, которая входит в множество данных курсора, должна иметь уникальный индекс (как правило, это индекс первичного ключа), который задает набор для копирования, – ключ.

Такие курсоры не чувствительны к вставке или удалению данных, которые возникают после создания курсора, только к изменениям. Поэтому ключевые курсоры

в основном используются в качестве основы для создания курсоров на обновление данных.

Ключевые курсоры могут быть модифицированными, только для чтения (read only), последовательными и с прокруткой.

- последовательные (fast forward cursors; однонаправленные, "пожарные" (firehouse)) – представляет собой быстрый однонаправленный курсор. Однонаправленными их называют потому, что после получения данных с помощью такого курсора отсутствует возможность вернуть откорректированные данные в ту же таблицу. Он открывается только для чтения, не позволяет выполнять выборку данных в обратном направлении, не поддерживает прокрутку и приводится автоматически к курсору любого другого типа в следующих случаях:
 - если в базовом запросе используются поля типа text, ntext, image или оператор TOP, SQL Server превращает курсор в ключевой;
 - если последовательный курсор объявлен как курсор FOR UPDATE, тогда он превращается в ключевой;
 - если последовательный курсор объявлен как курсор FOR UPDATE, но хотя бы одна базовая таблица не имеет уникального индекса, тогда курсор превращается в статический;
 - если базовый запрос создает временную таблицу курсор также превращается в статический.

Среди такого разнообразия курсоров поневоле сталкиваешься с вопросом: а какой же тип курсора выбрать?

Статический курсор из-за его чрезмерной ресурсоемкости, особенно в клиент-серверных приложениях лучше не использовать. Если необходимо только просматривать данные и достаточно однонаправленного просмотра, тогда подойдет последовательный курсор. Оптимальным выбором и в локальном, и в клиент-серверном приложении будет динамический курсор. Но, если в последнем случае (клиент-сервер) ресурсы ограничены, тогда лучше остановить свой выбор на ключевом курсоре.

Использование курсоров схоже с использованием обычных локальных переменных – их нужно сначала объявить, установить значение, а затем можно использовать. Но в отличие от локальных переменных, которые автоматически уничтожаются при выходе из области видения, курсоры необходимо закрыть, освободив при этом используемые данные, а затем уничтожить.

Для получения информации о характеристиках курсора используют следующие системные хранимые процедуры:

- **sp_cursor_list** – список курсоров вместе с атрибутами, доступных для соединения в текущий момент времени;
- **sp_describe_cursor** – описывает атрибуты курсора (тип курсора, прокрутка и т.п.);
- **sp_describe_cursor_columns** – описывает атрибуты полей результирующего набора;
- **sp_describe_cursor_tables** – описывает базовые таблицы, к которым имеет доступ курсор.

2.2. Создание курсоров

Для того, чтобы создать курсор используется оператор `DECLARE CURSOR`. SQL Server 2008 поддерживает два формата объявления курсора:

```
-- синтаксис ISO
DECLARE имя_курсора
    [ INSENSITIVE ]
    [ SCROLL ]           /* прокрутка */
    CURSOR
FOR оператор_select    /* записи, которые будут включены
в множество курсора */
[ FOR { READ ONLY | UPDATE [ OF имя_поля [ ,...n ] ] } ]

-- синтаксис Transact-SQL
DECLARE имя_курсора CURSOR
[ LOCAL | GLOBAL ]     /* область видения */
[ FORWARD_ONLY | SCROLL ] /* прокрутка */
[ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ] /* тип
курсора */
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ] /* блокирование
*/
[ TYPE_WARNING ]
FOR оператор_select    /* записи, которые будут включены
в множество курсора */
[ FOR UPDATE [ OF имя_поля [ ,...n ] ] ]
```

Разберем кратко каждый из параметров данного оператора:

- Область видения курсора определяется с помощью ключевых слов **LOCAL** или **GLOBAL**, которые равносильны переменным **@локальная_таблица** и **@@глобальная_таблица**, или префиксам **#** и **##** при объявлении временных таблиц. По умолчанию область видения курсора определяется параметром **default to local cursor** базы данных.

Что касается глобального курсора, то в текущем соединении, кроме текущего процесса, к нему имеют доступ и другие процессы, такие как пакеты, триггеры, хранимые процедуры. Простыми словами, курсор можно создать с помощью одной хранимой процедуры и обратиться к нему с другой без передачи на него ссылки. Локальный курсор указывает на доступ к нему только с текущего процесса текущего соединения.

SQL Server закрывает и высвобождает локальный курсор при выходе за пределы его области действия, но лучше сделать это самостоятельно.

- прокрутка задает возможность перемещения курсора: только с начала в конец (**FORWARD_ONLY**) или в произвольном направлении (**SCROLL**). По умолчанию допускается только последовательное перемещение курсора, поэтому в случае необходимости перемещения на предыдущую запись следует переоткрыть курсор. При установлении параметра прокрутки не следует забывать о том, какой тип курсора вы создаете;
- тип курсора по умолчанию динамический (**DYNAMIC**). Напомним, что параметр **FAST_FORWARD** (для создания последовательного курсора) не может быть указан вместе с параметрами **SCROLL** или **FOR_UPDATE**;
- блокировка определяет, могут ли записи модифицироваться курсором, и если да, то могут ли их модифицировать другие пользователи. Блокировка позволяет минимизировать проблемы, связанные с организацией параллельной работы на сервере

при наличии курсора. Для этого предназначены три *опции*:

- **READ_ONLY** – курсор только для чтения;
- **SCROLL_LOCKS** – осуществляет блокирование прокрутки (иногда называют "пессимистической блокировкой"). Данный параметр запрещает любому пользователю вносить изменения в запись, если он в этот момент модифицируется курсором. SQL Server будет блокировать записи по мере считывания их в курсор, для обеспечения их доступности для дальнейших изменений. Не следует также забывать, что параметр SCROLL_LOCKS не поддерживается последовательными и статическими курсорами;
- **OPTIMISTIC** – не осуществляет блокирование прокрутки, так называемое "оптимистическое блокирование". SQL Server не блокирует записи во время их считывания в курсор, считая, что в это время никто не работает с этими данными (не обновляет их и не удаляет). Такая ситуация довольно оптимистичная, но вполне реальная, когда база данных большая, а пользователей мало. При этом, если данные все же кто-то изменил, то операции обновления и удаления, которые осуществляются через курсор, работать не будут и для осуществления дальнейших действий необходимо перезаполнять курсор. Для определения того, изменились данные записи после считывания их в курсор, SQL Server выполняет сравнение значений поля timestamp (если оно

существует) или контрольных сумм. Параметр **OPTIMISTIC** нельзя указывать для последовательных (**FAST_FORWARD**) курсоров;

- параметр **TYPE_WARNING** предусматривает от SQL Server'ом предупреждения, если тип курсора приводится от заданного к другому типу, то есть в случае неявного приведения к типу;
- **FOR UPDATE** определяет поля в курсоре, которые будут обновляться;
- В разделе **OF**, если он указан, содержится перечень полей, которых будут касаться изменения, все остальные поля будут рассмотрены как предназначенные только для чтения. В случае отсутствия списка, обновление коснется всех полей, которые указаны в списке **SELECT** при создании курсора.

При использовании синтаксиса **ISO** используется параметр **INSENSITIVE**, который отсутствует в синтаксисе создания курсора стандарта **T-SQL**. Он позволяет создать курсор, в котором не будут отображаться изменения (обновление или удаление), осуществленные в базовых таблицах. Такой курсор работает только с данными временной таблицы базы данных **tempdb**, которая была заполнена при создании курсора. В связи с этим, курсор, созданный с параметром **INSENSITIVE**, не может использоваться для изменения данных в базовых таблицах.

Множество записей, на которые указывает курсор, определяется с помощью оператора **SELECT**. При этом инструкция **SELECT** не может:

- возвращать несколько результирующих множеств;
- содержать ключевое слово **INTO** для создания новой таблицы;

- содержать ключевые слова COMPUTE, COMPUTE BY и FOR BROWSE, но может содержать функции агрегирования.

ПРИМЕЧАНИЕ!

Если при создании курсора не указываются параметры блокировки, тогда принимаются следующие значения по умолчанию:

- если оператор SELECT не поддерживает обновление (например, недостаточно привилегий), тогда курсору присваивается параметр READ_ONLY;
- статические и последовательные курсоры по умолчанию имеют значение READ_ONLY;
- динамические и ключевые курсоры по умолчанию имеют значение OPTIMISTIC.

Итак, курсор объявили, но декларация курсора не создает набор записей, которыми курсор будет манипулировать. Для того, чтобы создать собственное множество записей курсора, необходимо его открыть с помощью оператора **OPEN**:

```
OPEN { [ GLOBAL ] курсор | переменная_курсора }
```

После открытия курсора выполняется оператор SELECT и осуществляется выборка данных из указанных таблиц в курсор, после чего можно приступить к основной работе.

Ключевое слово GLOBAL в операторе OPEN помогает избежать конфликтов имен, поскольку в случае наличия двух курсоров с одинаковыми идентификаторами по умолчанию все ссылки будут касаться локального курсора.

ПРИМЕЧАНИЕ!

*Для получения количества выбранных записей в последнем открытом курсоре, следует воспользоваться глобальной переменной @@ **CURSOR_ROWS**.*

Закрывает открытый курсор и освобождает текущий результирующий набор оператор **CLOSE**:

```
CLOSE { [ GLOBAL ] курсор | переменная_курсора }
```

Оператор **CLOSE** оставляет структуры данных доступными для повторного открытия, но выборка и обновление данных запрещены.

Удаление ссылки курсора осуществляется с помощью оператора **DEALLOCATE**. При этом, когда уничтожается последняя ссылка курсора, SQL Server высвобождает структуры данных, которые входили в курсор. Синтаксис данного оператора следующий:

```
DEALLOCATE { [ GLOBAL ] курсор | @ переменная_курсора }
```

Приведем небольшой пример создания курсора и заполнение его данными.

```
-- создаем курсор
declare myCursor cursor
for select *
from book.Books
-- открываем курсор, то есть создаем множество записей
курсора
open myCursor
/ * Манипулирования записями с помощью курсора * /
- Закрываем курсор
close myCursor
-- освобождаем курсор
deallocate myCursor
```

Кстати, T-SQL позволяет объявлять переменные типа **CURSOR**. Переменная такого типа создается привычным образом, а значение ей присваивается с помощью оператора **SET**.

К примеру:

```
declare myCursor cursor local fast_forward
for
select NameBook
from book.Books
declare @myCursorVariable cursor /* создаем переменную
курсора */
open myCursor
set @myCursorVariable = myCursor /* Назначаем переменную
курсора манипулирования записями с помощью курсора или
через ассоциированную переменную */
close myCursor
deallocate myCursor
```

Следует отметить, что после освобождения курсора, идентификатор `myCursor` больше не ассоциируется с набором записей курсора. Но на множество курсора еще ссылается переменная `@cursorVariable`, поэтому курсор и его множество не освободятся, пока явно не освободить и курсорную переменную. Фактически курсор и его набор записей будут существовать до тех пор, пока переменная не потеряет свое действие.

2.3. Манипулирование записями с помощью курсоров

После создания и открытия курсора можно приступать к основной работе с ним. Для этого необходимо сделать следующие действия:

- Получить первую запись с помощью оператора **FETCH**.
- По мере необходимости провести обработку в цикле других записей с помощью оператора **FETCH**.

Синтаксис оператора **FETCH** следующий:

```

FETCH
[ [ NEXT | PRIOR | FIRST | LAST
    | ABSOLUTE { n | @nvar }
    | RELATIVE { n | @nvar }
]
FROM
]
{ [ GLOBAL ] курсор | @переменная_курсора }
[ INTO @ переменная [ ,...n ] ]

```

Расшифруем ключевые слова, которые отвечают за *направление курсора*:

- **NEXT** – возвращает следующую запись в курсоре после текущей. Если выборка осуществляется впервые, тогда возвращается первая запись результирующего набора.
- **PRIOR** – возвращает запись, которая находится в курсоре перед текущей (предварительная запись). Если выборка из курсора осуществляется впервые, тогда никакая запись не возвращается и положение курсора остается перед первой записью.
- **FIRST** – возвращает первую запись в курсоре и делает ее текущей.
- **LAST** – возвращает последнюю запись в курсоре и делает ее текущей.
- **ABSOLUTE** – варианты действий следующие:
 - если значение параметра (n или @nvar) имеет

- положительное значение, тогда возвращается запись, удаленная на *n* записей от начала курсора;
- если значение отрицательное – возвращается запись, удаленная на *n* записей от конца курсора;
- если *n* или *@nvar* равны 0, тогда записи не возвращаются.
- **RELATIVE** – варианты действий следующие:
 - если значение параметра (*n* или *@nvar*) имеет положительное значение, тогда возвращается запись, удаленная на *n* записей от текущей;
 - если значение отрицательное – возвращается запись, которая предшествует на *n* записей текущей;
 - если *n* или *@nvar* равны 0, тогда возвращается текущая запись. Но, если при первой выборке указывается отрицательное или нулевое значение, тогда записи не возвращаются.

В параметрах **ABSOLUTE** и **RELATIVE** значение *n* должно быть целочисленной константой, а *@nvar* должно иметь тип `smallint`, `tinyint` или `int`.

Параметр **INTO** позволяет поместить данные из полей выборки в локальные переменные. Каждая переменная из списка связывается с соответствующим полем в результирующем наборе курсора. При этом типы данных должны совпадать или поддерживать неявное приведение к типу.

ПРИМЕЧАНИЯ!

1. Если в инструкции **DECLARE CURSOR** стандарта ISO не указан параметр **SCROLL**, то единственным параметром инструкции **FETCH** может быть **NEXT**.

2. В стандарте Transact-SQL при объявлении курсора действуют следующие правила:
- если указанный FORWARD_ONLY или FAST_FORWARD, тогда оператор FETCH поддерживает только NEXT;
 - если не указаны DYNAMIC, FORWARD_ONLY или FAST_FORWARD и указан один из параметров KEYSET, STATIC или SCROLL, тогда поддерживаются все параметры оператора FETCH;
 - курсоры DYNAMIC SCROLL поддерживают все параметры оператора FETCH, кроме ABSOLUTE.

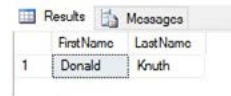
Итак, рассмотрим ряд примеров на использование курсора. Для начала рассмотрим простой пример, в котором с помощью динамического курсора возвращается запись в его текущей позиции.

```
-- declare cursor
declare auth_cursor cursor
for select FirstName, LastName
from Authors;
-- open cursor
open auth_cursor;
-- get first record from cursor
fetch next from auth_cursor;
-- close cursor
close auth_cursor;
-- deallocate resources
deallocate auth_cursor;
go
```

Результат:

| | id | FirstName | LastName | id_country |
|---|------|-----------|-----------|------------|
| ▶ | 1 | Donald | Knuth | 5 |
| | 2 | Jeffrey | Richter | 5 |
| | 3 | Herbert | Schildt | 5 |
| | 4 | Dino | Esposito | 9 |
| | 5 | Matthew | MacDonald | 5 |
| | 6 | Clifford | Simak | 5 |
| | 7 | Herbert | Wells | 4 |
| | 8 | Richard | Waymire | 4 |
| * | NULL | NULL | NULL | NULL |

Входная таблица



| | FirstName | LastName |
|---|-----------|----------|
| 1 | Donald | Knuth |

From cursor

Если вы обратите свое внимание на быстрое действие сценария, то заметите, что он выполняется дольше, чем соответствующий оператор SELECT. Это происходит потому, что операции создания и открытия курсора потребуют дополнительного времени. В связи с этим повторимся, что предпочесть курсор следует в самую последнюю очередь, если без них действительно не обойтись.

Расширим наш пример и выведем на экран все множество данных курсора. Для этого воспользуемся циклом:

```
-- declare cursor
declare auth_cursor cursor
for
select FirstName, LastName
from Authors;
-- open cursor
open auth_cursor;
-- get first record from cursor
fetch next from auth_cursor;
while @@FETCH_STATUS = 0
begin
-- get the next record
fetch next from auth_cursor;
end
```

```
-- close cursor
close auth_cursor;
-- deallocate resources
deallocate auth_cursor;
go
```

Результат:

| Results | | Messages | |
|---------|-----------|-----------|--|
| | FirstName | Last Name | |
| 1 | Donald | Knuth | |
| | FirstName | Last Name | |
| 1 | Jeffrey | Richter | |
| | FirstName | Last Name | |
| 1 | Herbert | Schildt | |
| | FirstName | Last Name | |
| 1 | Dino | Esposito | |
| | FirstName | Last Name | |
| 1 | Matthew | MacDonald | |
| | FirstName | Last Name | |
| 1 | Clifford | Simak | |
| | FirstName | Last Name | |
| 1 | Herbert | Wells | |
| | FirstName | Last Name | |
| 1 | Richard | Waymire | |
| | FirstName | Last Name | |

В нашем примере мы использовали глобальную переменную **@@ FETCH_STATUS**. Данная переменная возвращает информацию о состоянии выполнения последней команды **FETCH**, то есть позволяет определить, чем закончилась операция получения данных. Переменная **@@ FETCH_STATUS** может иметь следующие значения:

- 0 – выборка выполнена успешно;
- 1 – выборка завершилась неудачей;
- 2 – ошибка, связанная с отсутствием необходимой записи (попытка считать запись после последней или перед первой).

Оператор FETCH может также сохранять значения с возвращаемого поля в переменные. Продемонстрируем это на уже рассмотренном примере:

```
-- declare cursor
declare auth_cursor cursor
for
select FirstName, LastName
from Authors;
-- declare variables for storing data returned by FETCH
instruction
declare @fname varchar(50), @lname varchar(50);
-- open cursor
open auth_cursor;
-- get first record from cursor and save values into
variables
fetch next from auth_cursor into @fname, @lname;
-- display the variables values
print 'Name: ' + @fname + ' ' + @lname
while @@FETCH_STATUS = 0
begin
-- get the next record
fetch next from auth_cursor into @fname, @lname;
print 'Name: ' + @fname + ' ' + @lname
end
-- close cursor
close auth_cursor;
-- deallocate resources
deallocate auth_cursor;
go
```

Результат:

| Messages | |
|----------|-------------------|
| Name: | Donald Knuth |
| Name: | Jeffrey Richter |
| Name: | Herbert Schildt |
| Name: | Dino Esposito |
| Name: | Matthew MacDonald |
| Name: | Clifford Simak |
| Name: | Herbert Wells |
| Name: | Richard Waymire |
| Name: | Richard Waymire |

В предыдущих примерах оператор FETCH использовался для возвращения текущей записи, но данный оператор позволяет просматривать и другие записи, делая их текущими. Например, создадим локальный динамический курсор с прокруткой, содержащий набор данных о магазинах издательства.

```
-- declare cursor
declare shop_cursor cursor local scroll dynamic
for
select sh.Shop, c.Country
from Shops sh, Countries c
where sh.id_country=c.id
order by 1;
-- open cursor
open shop_cursor;
-- get the last record from the cursor
fetch last from shop_cursor;
-- get the record preceding the current cursor
position
fetch prior from shop_cursor;
-- get the second record from the cursor
fetch absolute 2 from shop_cursor;
```

```
-- get the third record from the cursor
fetch absolute 3 from shop_cursor;
-- get the record two rows upper from the current cursor
position
fetch relative - 2 from shop_cursor;
-- close cursor
close shop_cursor;
-- deallocate resources
deallocate shop_cursor;
go
```

Результат:

Множество данных курсора

| | Shop | Country |
|----|-------------|---------------|
| 1 | Booker | Canada |
| 2 | Booker | France |
| 3 | Booker | Great Britain |
| 4 | Booker | USA |
| 5 | Booker | Ukraine |
| 6 | Booker | Belgium |
| 7 | Booker | Germany |
| 8 | Booker | Sweden |
| 9 | BrightIdea | Belgium |
| 10 | BrightIdea | France |
| 11 | BrightIdea | Great Britain |
| 12 | BrightIdea | Germany |
| 17 | Rainbow | Canada |
| 18 | Rainbow | Great Britain |
| 19 | Rainbow | USA |
| 20 | Rare Books | USA |
| 21 | Rare Books | Ukraine |
| 22 | Smith&Brown | Great Britain |
| 23 | Smith&Brown | Germany |

Результаты действий над курсором

| | Shop | Country | |
|---|-------------|---------------|-----------|
| 1 | Smith&Brown | Germany | Last |
| 1 | Smith&Brown | Great Britain | Prior |
| 1 | Booker | France | Second |
| 1 | Booker | Great Britain | Third |
| 1 | Booker | Canada | 2 rows up |

2.4. Модификация и удаление данных

Если ваш курсор является модифицированным, тогда вы можете изменять входные данные в множестве курсора.

Для этого в T-SQL используется специальная форма инструкции WHERE для операторов UPDATE или DELETE – **WHERE CURRENT OF**.

С ее использованием синтаксис операторов UPDATE и DELETE приобретет следующий вид:

```
-- оператор UPDATE для позиционированного обновления
данных
UPDATE { таблица | представление }
SET название_поля = значение
WHERE CURRENT OF { [ GLOBAL ] имя_курсора | переменная_
курсора }
-- оператор DELETE для позиционированного удаления данных
DELETE
FROM { таблица | представление }
WHERE CURRENT OF { [ GLOBAL ] имя_курсора | переменная_
курсора }
```

Инструкция CURRENT OF в позиционированных обновлениях или удалениях используется для указания используемого курсора. При этом операция будет выполняться в текущем положении курсора, а ключевое слово GLOBAL указывает на то, что операция касается глобального курсора.

Например, создадим курсор, с помощью которого осуществим позиционированное обновление данных таблицы "Country", а именно: переименуем название другой страны в списке опция "USA".

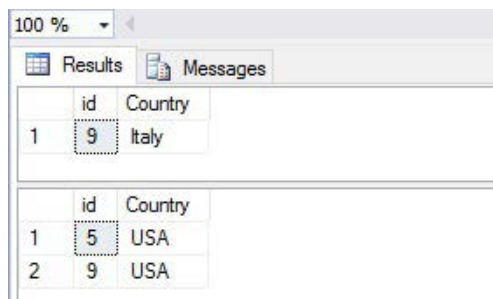
```
-- declare cursor
declare cnt_cursor cursor local keyset
for
select id, Country
from Countries
for update;
```

```

open cnt_cursor;
--get the 9th record
fetch absolute 9 from cnt_cursor;
--update the current record
update Countries
set Country = 'USA'
where current of cnt_cursor;
-- check the updated data
select *
from Countries
where Country = 'USA';
-- close cursor
close cnt_cursor;
-- deallocate resources
deallocate cnt_cursor;
go

```

Результат:



| | id | Country |
|---|----|---------|
| 1 | 9 | Italy |

| | id | Country |
|---|----|---------|
| 1 | 5 | USA |
| 2 | 9 | USA |

В результате работы сценария отображаются две панели сетки. В первой панели отображаются начальные значения полей, а во второй содержатся значения после изменения.

А теперь рассмотрим пример позиционированного удаления для последней записи курсора. В курсор

поместим набор данных о книгах, цена которых выше средней цены продажи книг издательства за текущий год.

```
-- объявляем курсор
declare qbook_cursor cursor
for
select b.ID_BOOK, b.NameBook
from book.Books b
where b.Price >
(select AVG(s.Price)
from sale.Sales s
where s.ID_BOOK = b.ID_BOOK
and DATEPART(YEAR, s.DateOfSale) = DATEPART(YEAR,
GETDATE()))
);
-- открываем курсор
open qbook_cursor;
-- перемещаемся на первую запись
fetch from qbook_cursor;
-- удаляем запись, на которую указывает курсор
delete
from book.Books
where current of qbook_cursor;
-- закрываем и высвобождаем курсор
close qbook_cursor;
deallocate qbook_cursor;
```

3. Использование XML в SQL Server 2008

3.1. Тип данных XML. Коллекция схем XML

Поддержка реляционными базами данных обработки XML-данных представляет собой мощный механизм по организации хранения и работы с неструктурированными данными. Начиная с версии SQL Server 2000, в ядро SQL Server была интегрировано большое количество встроенных средств поддержки языка XML. А после того, как в 2003 году Международная организация по стандартизации (ISO) и Американский национальный институт стандартов (ANSI) утвердили основные правила обработки XML-данных реляционными базами данных, в SQL Server 2005 был включен тип данных XML, который поддерживается и сейчас.

SQL Server имеет ряд **преимуществ** по сохранению данных в формате XML, среди которых можно выделить следующие:

- XML-данные имеют вид иерархической древовидной структуры, которая всегда должна содержать корневой элемент, называемый **XML-документом (XML document)**. Если XML-данные организованы без наличия корневого узла, тогда он называется фрагментом XML (XML fragment).

- SQL Server поддерживает коллекцию схем XML, которые содержат набор правил по структуре XML документов, которые хранятся в базе данных.
- По XML-данным можно выполнять поиск. Для этого в SQL Server существует поддержка языков XPath и XQuery.
- Вы можете обрабатывать XML-данные, вставлять, модифицировать или удалять необходимые узлы.

SQL Server 2008 позволяет *хранить XML-данные двумя способами*, каждый из которых имеет свои преимущества и недостатки:

1. Сохранение в текстовых полях одного из типов (n) char, (n) varchar или varbinary.

Основные преимущества хранения данных таким способом:

- XML сохраняет точность передачи текста, включая комментарии;
- не зависит от возможностей базы данных в обработке XML-данных;
- при изменении данных обеспечивается высокая производительность, поскольку вся работа сводится к действиям с обычными текстовыми данными.

Основные недостатки:

- отсутствует возможность работать на уровне узла;
- низкая производительность при поиске данных, поскольку придется считывать все данные.

2. Сохранение в полях с типом данных XML.

Основные преимущества хранения данных таким способом:

- данные хранятся и обрабатываются как XML;
- сохраняется порядок и структура XML-документа;
- поддерживается работа с данными на уровне узла (изменение, вставка, удаление);
- повышение производительности операций извлечения данных, поскольку для типа данных XML можно создавать несколько индексов;
- быстрая производительность поиска, за счет поддержки механизмов поиска XML-данных: как XPath и XQuery.

Основные недостатки:

- не сохраняется точность передачи текста, например, комментарии и XML-декларация и тому подобное;
- существует ограничение на вложенность узлов – максимально 128 уровней.

Следует отметить, что при сохранении XML-данных любым из двух способов, максимальный допустимый размер данных – 2 Гб. Но следует различать, что в первом случае – это 2 Гб обычных текстовых данных, а во втором непосредственно XML-данных.

В SQL Server 2008 разрешается также хранить XML-данные в переменных всех вышеперечисленных типов.

Приведем несколько примеров на создание источников сохранения XML-данных и их заполнения. Предположим, что необходимо для каждого магазина в XML формате сохранять каталоги книг издательства, которые находятся у них на реализации.

```

create table BooksCatalog
(
  id_catalog int identity not null,
  id_shop int not null constraint fkShops references
  Shops(id), BCatalog xml
);
insert into BooksCatalog (id_shop, BCatalog)
values (1, '<catalog><book>
          <name>C# 5.0: The Complete Reference</name>
          <author>>Herbert Schildt</author>
          <dateOfPublish>01/01/2012</dateOfPublish>
          <price>50</price>
          </book></catalog>');
insert into BooksCatalog (id_shop, BCatalog)
values (2, '<catalog><book>
          <name>C# 5.0: The Complete Reference</name>
          <author>>Herbert Schildt</author>
          <dateOfPublish>01/01/2012</dateOfPublish>
          <price>50</price>
          </book></catalog>');
insert into BooksCatalog (id_shop, BCatalog)
values (8, '<catalog><book>
          <name>C# 5.0: The Complete Reference</name>
          <author>>Herbert Schildt</author>
          <dateOfPublish>01/01/2012</dateOfPublish>
          <price>50</price>
          </book></catalog>')

```

SQL Server позволяет проверять XML-документы на корректность. Для этого используется коллекция XML-схем, которая представляет собой обычный набор документов схемы, которые хранятся в базе данных под одним именем. Все XML-схемы объявляются на уровне базы данных и разворачиваются на SQL Server. После их создания можно типизировать и проверять на корректность произвольное поле таблицы, содержаемое переменной

или параметра типа XML в соответствии с коллекцией XML-схем. Кстати, SQL Server поддерживает как типизированные (typed), так и нетипизированные (untyped) XML-данные.

Для создания XML-схемы используется оператор **CREATE XML SCHEMA COLLECTION**, синтаксис которого следующий:

```
CREATE XML SCHEMA COLLECTION [ схема. ] имя_ XML_схемы
AS
{ 'строочная_константа' | @переменная }
```

Строчная константа или скалярная переменная должны иметь тип varchar, varbinary, nvarchar или xml.

Например, создадим коллекцию XML схем для разработанного нами каталога книг:

```
create xml schema collection BooksCatalogSchema
as
'<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="catalog">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="book" minOccurs="0"
maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string" />
            <xs:element name="author" type="xs:string" />
            <xs:element name="DateOfPublish"
              type="xs:dateTime" />
            <xs:element name="Price" type="xs:double" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:element>
</xs:sequence>
```

```

</xs:complexType>
</xs:element>
</xs:schema>;

```

Но в большинстве случаев XML схему создают отдельно в файле с расширением .xsd, после чего загружают ее в переменную типа XML с помощью команды **OPENROWSET**.

```

-- синтаксис оператора OPENROWSET
OPENROWSET
( 'поставщик_данных_OLEDB', { 'имя_БД'; 'логин'; 'пароль'
| 'строка_доступа' },
                                { [ каталог. ] [ схема. ] название_объекта
| 'запрос' }
| BULK 'путь_к_файлу_данных',
{ FORMATFILE = 'путь_к_файлу_форматирования' [ bulk-опции ]
| SINGLE_BLOB -- содержание файла данных возвращается
в виде набора записей.
- Рекомендуется (!)
| SINGLE_CLOB-- считывает файл данных как ASCII-файл
| SINGLE_NCLOB -- считывает файл данных как Unicode
}
)
- Bulk-опции:
    - Кодовая страница данных в файле данных
[, CODEPAGE = { 'ACP' | 'OEM' | 'RAW' | 'кодовая_страница'
} ]
-- файл, который используется для выбора записей
с ошибками форматирования
[, ERRORFILE = 'название_файла' ]
[, FIRSTROW = номер ] -- номер первой записи для загрузки
[, LASTROW = номер ] -- номер последней записи для загрузки
[, MAXERRORS = число ] -- максимальное количество ошибок
в файле форматирования
[, ROWS_PER_BATCH = количество_запросов ] --
приблизительное количество записей в файле данных
[, ORDER ( { поле [ ASC | DESC ] } [ ,...n ] ) [ UNIQUE ]

```

Например:

```
declare @schema XML
select @schema = c
from OPENROWSET (BULK 'MySchema.xsd', SINGLE_BLOB) as
tmp (c)
create xml schema collection MySchema as @schema
```

После того, как XML схема подбавляется в базу данных (импортируется или создается локально), она может лексически отличаться от своего первоначального вида. Это связано с тем, что для повышения эффективности сохранения схемы SQL Server удаляет из нее ряд компонентов различных типов. Например, удаляются комментарии, пробелы и другие компоненты, приставки пространств имен также не сохраняются, а данные неявных типов приводятся к явным. Например, `<xs: element name = "author" />` конвертируется в `<xs: element name = "author" type = "xs: anyType" />`. В связи с этим рекомендуется хранить копию каждой схемы.

Для просмотра существующей коллекции XML схем используется встроенная функция **xml_schema_namespace()**. Функция возвращает данные типа xml.

```
xml_schema_namespace ( 'схема_БД', 'имя_коллекции_XML_схем', [ 'пространство_имен' ] )
```

В первом параметре указывается название реляционной схемы, в которой размещается коллекция XML схем. Во втором – ее имя, которое имеет тип **sysname**. Необязательный параметр "пространство имен" используется для обозначения конкретного URI пространства имен, которое относится к XML-схемы. Фактически, данный параметр позволяет выбрать из коллекции XML схему,

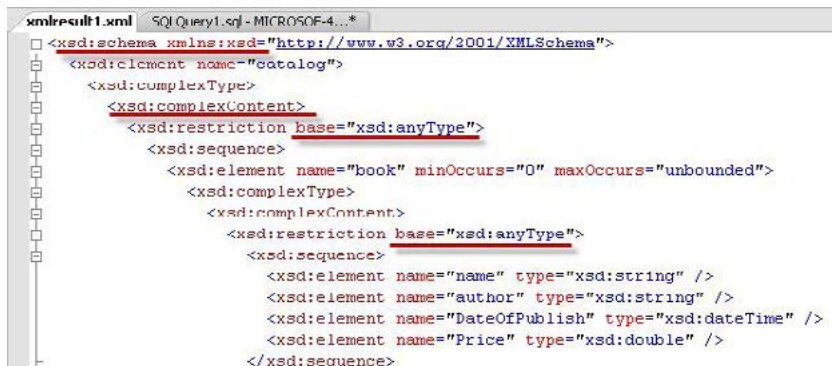
которая относится к указанному пространству имен. Если же URI пространства имен отсутствует, тогда перестраивается вся коллекция XML-схем.

Аргумент пространства имен имеет тип `nvarchar (4000)`, то есть его максимальная длина равна 1000 символов.

Итак, воспользуемся функцией `xml_schema_namespace ()` для получения информации о коллекции XML схем:

```
select xml_schema_namespace ( N'dbo',
N'BooksCatalogSchema')
```

Результат:



Как видно из результата, после вставки XML-схема изменила свой первоначальный вид, как и отмечалось выше.

Если XML-данные являются типизированными, тогда после создания коллекции XML-схем, ее необходимо с ними связать. Это можно сделать одним из следующих **способов**.

- При создании таблицы:

```
create table BooksCatalog ( ID_CATALOG int identity not
null,
BCatalog xml (BooksCatalogSchema) );
```

- Если таблица уже создана:

```
alter table BooksCatalog
alter column BCatalog xml (BooksCatalogSchema);
```

- При объявлении переменной типа XML:

```
declare @xmldoc as xml (BooksCatalogSchema);
```

Для манипулирования созданными XML-схемами используются операторы **ALTER / DROP XML SCHEMA COLLECTION**.

Инструкция **ALTER XML SCHEMA COLLECTION** позволяет только добавлять новые XML-схемы с пространствами имен в коллекцию или новые компоненты к существующим пространствам имен. Синтаксис данного оператора следующий:

```
ALTER XML SCHEMA COLLECTION [ схема. ] имя_XML_схемы
ADD
'компонент_схемы_для_вставки'
```

Например, необходимо добавить новый элемент **<Annotation>** к существующему пространству имен в коллекции XML-схем BooksCatalogSchema.

```
alter xml schema collection BooksCatalogSchema
add
'<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="Annotation" type="xs:string" />
</xs:schema>'
Go
select xml_schema_namespace ( N'dbo',
N'BooksCatalogSchema')
```

Результат:



```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Annotation" type="xsd:string" />
  <xsd:element name="catalog">
    <xsd:complexType>
      <xsd:complexContent>
        <xsd:restriction base="xsd:anyType">
```

Заметим, что в случае добавления новых компонентов к коллекции, которые ссылаются на уже существующие, необходимо воспользоваться командой `<import namespace = "referenced_component_namespace" />`.

Для удаления существующей коллекции схем, используется оператор **DROP XML SCHEMA COLLECTION**.

```
DROP XML SCHEMA COLLECTION [ схема. ] имя_XML_схемы
```

3.2. Выборка и изменение XML-данных

В SQL Server 2008 реализовано много средств получения XML-данных, независимо от того, в каком виде они хранятся: в полях типа XML или в текстовых полях. Рассмотрим методы получения XML-данных для обоих вариантов сохранения.

Итак, если XML-данные хранятся в полях, параметрах или переменных типа XML, тогда для работы с фрагментами XML используют **5 методов**, которые для поиска используют выражения XPath или XQuery:

- **query ("запрос")** – возвращает искомый нетипизированный (который не соответствует XML-схеме) фрагмент данных XML. Приравнивается к выполнению обычных SQL-запросов.
- **value ("выражение", "тип_результата")** – возвращает скалярное типизированное значение, которое затем превращается в тип T-SQL. В связи с тем, что метод

возвращает скалярное значение, выражение XPath или XQuery нужно писать так, чтобы результатом было единственное значение. Например, в XPath существует функция `count ()`, которая возвращает количество выделенных элементов, или `position ()`, что указывает позицию текущего узла.

- **exist ("выражение")** – осуществляет проверку узлов на существование. Данный метод возвращает `true`, если запрос найдет хотя бы один узел. Обычно метод `exist ()` используется в приложении WHERE оператора SELECT для проверки наличия определенного узла.
- **modify ("выражение XML DML")** – служит для манипулирования (вставки, удаления или обновления) XML-данными. При этом он позволяет изменять как значение в XML-узлах, так и структуру фрагмента XML.
- **nodes ("выражение")** – возвращает фрагмент XML, разбитый на набор записей, то есть в виде реляционной таблицы.

Приведем несколько примеров по работе с XML-данными с помощью вышерассмотренных методов. В качестве источника данных возьмем созданную в предыдущем подразделе таблицу `BooksCatalog`, которая содержит каталоги книг для магазинов. Для начала попробуем получить XML-данные о магазинах, которые реализуют книги Герберта Шилдта.

```

select bc.id_shop as 'Identifier',
       sh.Shop as 'Shop',
       bc.BCatalog.query('/catalog/book[author = "Herbert
                           Schildt"]') as 'Catalog'
from BooksCatalog bc, Shops sh
where bc.id_shop = sh.id
and bc.BCatalog.exist('/catalog/book[author = "Herbert
                        Schildt"]') = 1

```

Результат:

| Results | | | |
|---------|------------|---------|---|
| | Identifier | Shop | Catalog |
| 1 | 1 | Rainbow | <book><name>C# 5.0. The Complete Reference</name><author>Herbert Schildt</author><dateOfPublish>01/01/2012... |
| 2 | 2 | Rainbow | <book><name>C# 5.0. The Complete Reference</name><author>Herbert Schildt</author><dateOfPublish>01/01/2012... |
| 3 | 8 | HaehTag | <book><name>C# 5.0. The Complete Reference</name><author>Herbert Schildt</author><dateOfPublish>01/01/2012... |

Для получения информации о книге, которая первая представлена в каталоге магазина "Rainbow" следует написать следующий запрос:

```

select bc.BCatalog.value('(/catalog/book)[1]',
                          'nvarchar(50)')
from BooksCatalog bc, Shops sh
where bc.id_shop = sh.id and sh.Shop = 'Rainbow'

```

Метод **modify()** сложнее предыдущих методов и требует короткого разъяснения. В качестве параметра метод принимает выражение XML DML, который выполняется над фрагментом XML.

В языке XML DML используются следующие регистрозависимые *ключевые слова*:

- **insert** – добавляет один или несколько узлов. Для более точной вставки узлов используются операторы:
 - **after** – вставить после узла, указанного в качестве второго параметра;

- **before** – вставить перед узлом, указанного в качестве второго параметра;
- **into** – узлы будут добавлены как дочерние по отношению к узлу, указанного в качестве второго параметра. Если узел уже имеет дочерние элементы, тогда можно указать их точное место расположения: на начало (as first into) или в конец (as last into).
- **delete** – удаляет один или несколько узлов;
- **replace value of** – обновляет значение заданного узла и используется в операторе UPDATE.

Стоит отметить, что метод `modify()` вызывается с помощью инструкции SET, которая может использоваться отдельно или в составе оператора UPDATE.

Допустим, в магазин "Rainbow" отправлены на реализацию ряд новых книг. Итак, каталог данного магазина следует подправить. Сначала необходимо добавить информацию о новых книгах в XML-документ, а затем добавить сами данные. Для этого следует воспользоваться методом **modify()**, а в выражениях XML DML будет использовано ключевое слово `insert` с необходимыми опциями:

```
update BooksCatalog
set BCatalog.modify(
'insert <book>
<name>CLR via C#</name>
<author>Jeffrey Richter</author>
<dateOfPublish>11/25/2012</dateOfPublish>
<price>43</price>
</book> before (/catalog/book)[1]')
where id_shop in
```

```

(select id from Shops where Shop = 'Rainbow' and id_
country=2)
--add discount attribute price greater than 200
update BooksCatalog
set BCatalog.modify(
'insert attribute discount {"true"} into (/catalog/
book[price>20])[1]')
--disable discount in shops "Rainbow" in Great Britain
update BooksCatalog
set BCatalog.modify(
'replace value of (/catalog/book/@discount)[1] with
"false"')
where id_shop in
(select id from Shops where Shop = 'Rainbow' and id_
country=4)
--remove all books with price > 500 in shops "Rainbow" in
Great Britain
update BooksCatalog
set BCatalog.modify(
'delete catalog /catalog/book[price>500]')
where id_shop in
(select id from Shops where Shop = 'Rainbow' and id_
country=4)

```

Результат:

Rainbow shop in Canada (id_country=2)

```

<book discount="true">
  <name>C# 5.0: The Complete Reference</name>
  <author>Herbert Schildt</author>
  <dateOfPublish>01/01/2012</dateOfPublish>
  <price>50</price>
</book>

```

Rainbow shop in Great Britain (id_country=4)

```

<book discount="false">
  <name>C# 5.0: The Complete Reference</name>
  <author>Herbert Schildt</author>
  <dateOfPublish>01/01/2012</dateOfPublish>
  <price>50</price>
</book>

```

Последний метод **nodes ()** используется для отображения данных в табличном виде. Для каждого узла входного XML-документа, который соответствует выражению, формируется отдельная запись. Результирующая таблица содержит единственное поле типа XML. В связи с тем, что оно только одно, для получения данных каждой записи, как правило, используют методы **value ()**, **query ()** или **exist ()**. Общий синтаксис использования данного метода следующий:

```
nodes ('выражение') as результирующая_таблица (поле)
```

Например:

```
declare @vcatalog as xml;
set @vcatalog = '<catalog><book discount="false">
<name>CLR via C#</name>
<author>Jeffrey Richter</author>
<dateOfPublish>11/25/2012</dateOfPublish>
<price>43</price>
</book>
<book>
<name>Swing: A Beginner Guide</name>
<author>Herbert Schildt</author>
<dateOfPublish>08/09/2006</dateOfPublish>
<price>34</price>
</book>
</catalog>';
select c.value('name[1]', 'nvarchar(max)') as 'Book Title',
       c.query('.') as 'Full Description'
from @vcatalog.nodes('/catalog/book') as tmp(c);
```

Результат:

| | Book Title | Full Description |
|---|-------------------------|---|
| 1 | CLR via C# | <book discount="false"><name>CLR via C#</name><author>Jeffrey Richter</author><dateOfPublish>11/25/2012</dateOfPublish><price>43</price></book> |
| 2 | Swing: A Beginner Guide | <book><name>Swing: A Beginner Guide</name><author>Herbert Schildt</author><dateOfPublish>08/09/2006</dateOfPublish><price>34</price></book> |

К сожалению, если необходимо вывести данные, хранящиеся в поле типа XML некоторой таблицы, такой синтаксис использовать нельзя. Причина заключается в том, что метод `nodes()` возвращает результирующий набор, а оператор `SELECT` требует возврата единого значения. Чтобы постричь данную ситуацию, следует воспользоваться оператором **APPLY**, который позволяет вызвать метод для каждой записи результирующей множества.

Этот оператор имеет две формы:

- **CROSS APPLY** – возвращает только НЕ NULL-значения;
- **OUTER APPLY** – отображает записи, даже если они равны NULL.

Обобщенный синтаксис использования оператора `APPLY` с методом `nodes()` имеет следующий вид:

```
SELECT список_полей
FROM список_таблиц
{ CROSS | OUTER } APPLY поле.nodes('выражение') as
результующая_таблица (поле)
```

Например:

```
select bc.ID_CATALOG,
       tmp.c.value('name[1]', 'nvarchar(max)') as 'Book
       title',
       tmp.c.query('.') as 'Full description'
from BooksCatalog bc
CROSS APPLY bc.BCatalog.nodes('/catalog/book') as tmp(c)
```

Результат:

| Results | | Messages | |
|---------|------------|--------------------------------|--|
| | id_catalog | Book title | Full Description |
| 1 | 2 | CLR via C# | <book discount="true"><name>CLR via C#</name><author>Jeffrey Richter</author></book> |
| 2 | 2 | C# 5.0: The Complete Reference | <book><name>C# 5.0: The Complete Reference</name><author>Herbert Sc... |
| 3 | 3 | C# 5.0: The Complete Reference | <book discount="true"><name>C# 5.0: The Complete Reference</name><aut... |
| 4 | 4 | C# 5.0: The Complete Reference | <book discount="true"><name>C# 5.0: The Complete Reference</name><aut... |

3.3. Выборка реляционных данных в формате XML. Конструкция FOR XML

Для того, чтобы получить результаты SELECT запроса в виде XML, то есть для конвертирования данных текстовых полей, параметров или переменных в XML-структуру используется приложение **FOR XML** того же оператора SELECT. Инструкция FOR XML может использоваться как в запросах верхнего уровня (только в операторе SELECT), так и в подзапросах (в операторах INSERT, DELETE, UPDATE).

```
SELECT список_выборки [ INTO имя_новой_таблицы]
[ FROM список_таблиц ]
[ WHERE условие ]
[ GROUP BY выражение_группирования ]
[ HAVING условие_на_группу]
[ ORDER BY направление_сортирования ]
[ COMPUTE { {AVG | COUNT | MAX | MIN | SUM} (выражение) }
[ BY выражение] ]
[ FOR XML
{
{ RAW [ ( 'название_элемента' ) ] | AUTO }
[ общие_характеристики
[ , { XMLDATA | XMLSCHEMA [ ( 'URI_пространства_имен' )
] } ]
[ , ELEMENTS [ XSINIL | ABSENT ]
]
| EXPLICIT
[ общие_характеристики [ , XMLDATA ] ]
| PATH [ ( 'название_элемента' ) ]
[ общие_характеристики
[ , ELEMENTS [ XSINIL | ABSENT ] ] ]
}]
-- общие_характеристики
[ , BINARY BASE64 ] -- возвращает двоичные данные в
```

```

зашифрованном двоичном формате base64
[, TYPE ]          -- вернуть данные в виде типа данных
XML
[, ROOT [ ( 'корневой_элемент' ) ] ] -- создать корневой
элемент

```

Как видно из синтаксиса использования, приложение FOR XML превращает результирующий набор запроса в XML-структуру и поддерживает следующие **4 режима форматирования**:

- RAW;
- AUTO;
- EXPLICIT;
- PATH.

Рассмотрим кратко каждый из этих режимов.

Режим **RAW** является самым простым и ограниченным способом преобразования данных в формат XML. Он превращает каждую запись результирующего набора в XML элемент под названием <row />. Элементу <row /> можно задать собственное имя, указав его в скобках после ключевого слова RAW.

Все поля в результирующем XML фрагменте будут представлены *в виде атрибутов элемента* <row /> с соответствующими названиями полей. Чтобы переименовать атрибуты, следует в списке выборки SELECT задать каждому полю необходимый псевдоним.

К примеру,

```

select b.NameBook as 'name',
       a.FirstName+' '+a.LastName as 'author',
       b.DateOfPublish,
       b.Price
from book.Books b, book.Authors a

```



```
where b.ID_AUTHOR=a.ID_AUTHOR
order by 1
for xml raw('book')
```

Результат:



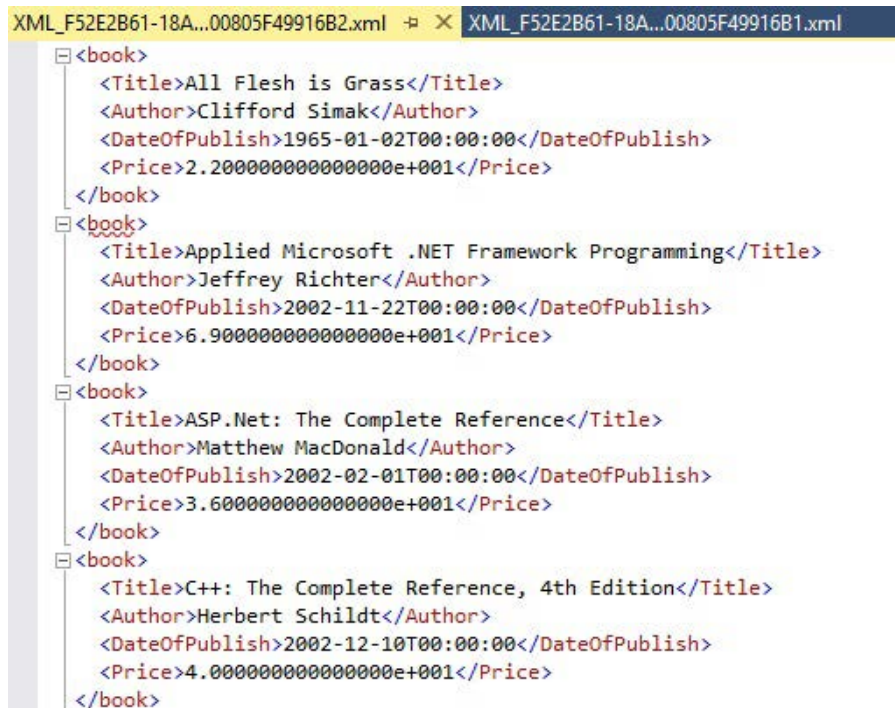
ПРИМЕЧАНИЕ!

Для корректного отображения результата работы запроса следует выставить режим отображения данных "Results to grid".

Для представления данных полей в виде элементов необходимо указать ключевое слово **ELEMENTS** после приложения **FOR XML RAW**. Данный параметр поддерживается также в режимах **AUTO** и **PATH**.

```
select b.NameBook as 'name',
       a.FirstName+' '+a.LastName as 'author',
       b.DateOfPublish,
       b.Price
from book.Books b, book.Authors a
where b.ID_AUTHOR=a.ID_AUTHOR
order by 1
for xml raw('book'), elements
```

Результат:



```
XML_F52E2B61-18A...00805F49916B2.xml  XML_F52E2B61-18A...00805F49916B1.xml
<book>
  <Title>All Flesh is Grass</Title>
  <Author>Clifford Simak</Author>
  <DateOfPublish>1965-01-02T00:00:00</DateOfPublish>
  <Price>2.2000000000000000e+001</Price>
</book>
<book>
  <Title>Applied Microsoft .NET Framework Programming</Title>
  <Author>Jeffrey Richter</Author>
  <DateOfPublish>2002-11-22T00:00:00</DateOfPublish>
  <Price>6.9000000000000000e+001</Price>
</book>
<book>
  <Title>ASP.Net: The Complete Reference</Title>
  <Author>Matthew MacDonald</Author>
  <DateOfPublish>2002-02-01T00:00:00</DateOfPublish>
  <Price>3.6000000000000000e+001</Price>
</book>
<book>
  <Title>C++: The Complete Reference, 4th Edition</Title>
  <Author>Herbert Schildt</Author>
  <DateOfPublish>2002-12-10T00:00:00</DateOfPublish>
  <Price>4.0000000000000000e+001</Price>
</book>
```

Дополнительные опции XSINIL и ABSENT параметра ELEMENTS используются для манипулирования отображением данных полей с NULL-значениями:

- **XSINIL** – создает элемент с атрибутом xsi: nil, который содержит значение true для полей с NULL-значениями;
- **ABSENT** – поля, содержащие NULL-значения в результирующий фрагмента XML не добавляются.

Можно также построить XML-схему для текущего XML-документа с помощью одной из следующих *опций*:

- **XMLDATA** возвращает встроенную XDR-схему, не добавляя корневой элемент к результату. Директива XMLDATA для параметра XML FOR является устаревшее и в следующей версии MS SQL Server планируется ее извлечение. В связи с этим следует избегать ее использования.
- **XMLSCHEMA** возвращает встроенную XSD-схему.

Результатом работы предыдущего запроса является фрагмент XML, поскольку в нем отсутствует корневой элемент. Для того, чтобы получить на выходе XML-документ, необходимо добавить инструкцию **ROOT** после приложения FOR XML RAW. Корневым элементом по умолчанию будет элемент с именем <root>, но вы можете задать собственное имя.

Итак, расширим наш пример:

```
select b.NameBook as 'Title',
       a.FirstName+' '+a.LastName as 'Author',
       b.DateOfPublish,
       b.Price
from Books b, Authors a
where b.id_author = a.id
order by 1
for xml raw('book'), root('catalog'), elements,
xmlschema('BooksCatalogSchema')
```

Частичный результат:

```

XML_F52E2B61-18A...00805F49916B3.xml  XML_F52E2B61-18A...00805F49916B1.xml  SQLQuery1.sql - (L:idea-PC\admin (53))
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
  <xsd:schema targetNamespace="BooksCatalogSchema" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:sqltypes="http://schemas.microsoft.com/sqlserver/2004/sqltypes" schemaLocation="http://schemas.microsoft.com/sqlserver/2004/sqltypes/BooksCatalogSchema.xsd">
    <xsd:element name="book">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="Title">
            <xsd:simpleType>
              <xsd:restriction base="sqltypes:varchar" sqltypes:localId="1049" sqltypes:sqlCompareOptions="IgnoreCase" >
                <xsd:maxLength value="128" />
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="Author">
            <xsd:simpleType>
              <xsd:restriction base="sqltypes:varchar" sqltypes:localId="1049" sqltypes:sqlCompareOptions="IgnoreCase" >
                <xsd:maxLength value="257" />
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="DateOfPublish" type="sqltypes:datetime" minOccurs="0" />
          <xsd:element name="Price" type="sqltypes:float" minOccurs="0" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
  <book xmlns="BooksCatalogSchema">
    <Title>All Flesh is Grass</Title>
    <Author>Clifford Simak</Author>
    <DateOfPublish>1965-01-02T00:00:00</DateOfPublish>
    <Price>2.2000000000000000e+001</Price>
  </book>

```

XSD-схема

XML-документ

Идем дальше. По умолчанию инструкция FOR XML возвращает XML как текст, поэтому результат ее работы может быть присвоен как строчной переменной, так и переменной типа XML. В последнем случае осуществляется неявное приведение к типу. Но приложение FOR XML поддерживает также явное преобразование с помощью инструкции **TYPE**.

В связи с тем, что инструкция TYPE возвращает данные типа XML, к исходному набору данных можно применить все методы для работы с XML. Это значительно расширяет возможности по обработке данных.

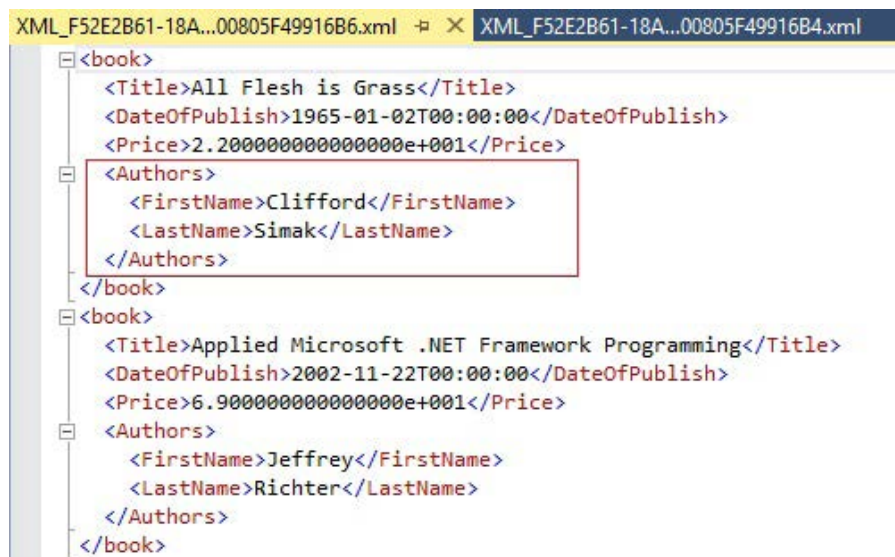
В режиме **AUTO** результаты запроса возвращаются в виде простого вложенного дерева XML. Для каждой таблицы, указанной в запросе SELECT, создается новый уровень в XML-структуре. Список SELECT задает порядок

вложенности XML-данных. Но, если поля в запросе смешанные, тогда XML-узлы переупорядковываются таким образом, чтобы все узлы, которые относятся к одному уровню, были сгруппированы под одним и тем же родительским элементом.

К примеру,

```
select b.NameBook as 'Title',
       a.FirstName, a.LastName,
       b.Price
from Books b, Authors a
where b.id_author = a.id
order by 1
for xml auto, elements
```

Результат:



Режим **EXPLICIT** позволяет более гибко определить исходную XML-структуру. Каждое поле настраивается отдельно, то есть разрешается смешанное использование

элементов и атрибутов. При этом результат запроса сравнивается с шаблоном, так называемой универсальной таблицей. Универсальная таблица должна содержать несколько **обязательных полей**:

- **Tag** – первое поле результирующей множества, которое указывает глубину XML-структуры, начиная с 1;
- **Parent** – второе поле, указывающее на родительский узел.

Псевдонимы полей формируются по следующему шаблону:

```
[директива] имя_XML_элемента!уровень_ вложенности!имя_
атрибута_XML
```

Директива является необязательным параметром и предоставляет дополнительную информацию для форматирования XML. Кстати, псевдонимы таблиц в режиме EXPLICIT игнорируются (!).

Для объединения нескольких запросов в одну полную универсальную таблицу можно использовать оператор UNION или UNION ALL. В таком случае каждый запрос представляет отдельный уровень иерархии.

Например, сформируем XML-документ, содержащий список книг, в разрезе тематик.

```
select 1 as Tag, NULL as Parent,
       Themes.ID_THEME as [Themes!1!id],
       Themes.NameTheme as [Themes!1!name],
       NULL as [Books!2!name]
from book.Themes
UNION ALL
```

```

select 2 as Tag, 1 as Parent,
       Themes.ID_THEME,
       NULL,
       Books.NameBook
from book.Themes, book.Books
where Books.ID_THEME = Themes.ID_THEME
order by [Themes!1!id], [Books!2!name]
FOR XML EXPLICIT, ROOT('BooksByThemes');

```

Результат:



```

XML_F52E2B61-18A...00805F49916B7.xml XML_F52E2B61-18A...00805F49916B6.xml XML_F52E2B61-18A...
BooksByThemes
<Themes id="1" name="Computer Science">
  <Books name="Applied Microsoft .NET Framework Programming" />
  <Books name="ASP.Net: The Complete Reference" />
  <Books name="C++: The Complete Reference, 4th Edition" />
  <Books name="CLR via C#" />
  <Books name="How to become a Hacker" />
  <Books name="Java: A Beginner Guide" />
  <Books name="Java: The Complete Reference" />
  <Books name="Microsoft SQL Server 2005 Complete Handbook" />
  <Books name="Pro Wpf 4.5 in C#: Windows Presentation Foundation in .Net 4.5" />
  <Books name="Pro WPF in C# 2010: Windows Presentation Foundation in .Net 4" />
  <Books name="Swing: A Beginner Guide" />
  <Books name="The Art of Computer Programming, vol.1 " />
  <Books name="The Art of Computer Programming, vol.2 " />
  <Books name="The Art of Computer Programming, vol.3 " />
  <Books name="Windows Runtime via C#" />
</Themes>
<Themes id="2" name="Science Fiction">
  <Books name="All Flesh is Grass" />
  <Books name="Ring Around the Sun" />
  <Books name="The Invisible Man" />
  <Books name="Time is the Simplest Thing" />
  <Books name="Way Station" />
</Themes>
<Themes id="3" name="Web Technologies">
  <Books name="Creating a Web Site: The Missing Manual" />
  <Books name="Programming ASP.NET Core (Developer Reference)" />
  <Books name="Programming Microsoft ASP.NET MVC (3rd Edition)" />
</Themes>

```

Без использования сортировки, результат будет следующим:

Как видно из примера, написание запросов в режиме EXPLICIT довольно сложно. В качестве альтернатив для достижения аналогичного результата (создание XML-иерархий) SQL Server предлагает использовать вложенные запросы FOR XML в режиме RAW, AUTO, PATH. Например, построим XML-документ с данными об авторах в разрезе стран их проживания:

```
select NameCountry,
(
    select FirstName as 'fname', LastName as 'lname'
    from book.Authors a
    where c.ID_COUNTRY = a.ID_COUNTRY
    for xml raw('author'), type, root('authors')
)
from global.Country c
for xml raw('country'), elements,
root('ListAuthorsByCountry')
```


Результат:

```
<country>
  <Country>Great Britain</Country>
  <authors>
    <author fname="Herbert" lname="Wells" />
    <author fname="Richard" lname="Waymire" />
  </authors>
</country>
<country>
  <Country>USA</Country>
  <authors>
    <author fname="Donald" lname="Knuth" />
    <author fname="Jeffrey" lname="Richter" />
    <author fname="Herbert" lname="Schildt" />
    <author fname="Matthew" lname="MacDonald" />
    <author fname="Clifford" lname="Simak" />
  </authors>
</country>
```

Режим **PATH** является упрощенной формой режима **EXPLICIT**. В этом режиме с помощью обычных XPath-выражений полям можно задавать псевдонимы, которые определяют месторасположение их данных в исходном фрагменте XML. По умолчанию все поля добавляются в элемент `<row />`, как и в режиме **RAW**.

```
declare @shops xml;
set @shops = (
  select sh.NameShop '@nameShop',
         c.NameCountry 'comment()',
         b.NameBook 'book/@name',
         b.Price 'book/price'
  from sale.Shops sh, sale.Sales s, global.Country c, book.
Books b
  where sh.ID_SHOP=s.ID_SHOP and b.ID_BOOK=s.ID_BOOK and
sh.ID_COUNTRY=c.ID_COUNTRY
  order by sh.NameShop
  FOR XML PATH('shop'), ROOT('shops')
);
select @shops;
```

Результат:

```

<shop nameShop="Rare Books">
  <!--Ukraine-->
  <book name="CLR via C#">
    <price>4.300000000000000e+001</price>
  </book>
</shop>
<shop nameShop="Rare Books">
  <!--USA-->
  <book name="The Art of Computer Programming, vol.1 ">
    <price>4.500000000000000e+001</price>
  </book>
</shop>
<shop nameShop="Rare Books">
  <!--USA-->
  <book name="Applied Microsoft .NET Framework Programming">
    <price>6.900000000000000e+001</price>
  </book>
</shop>
<shop nameShop="Rare Books">
  <!--USA-->
  <book name="Java: The Complete Reference">
    <price>3.700000000000000e+001</price>
  </book>
</shop>
<shop nameShop="Smith&Brown">
  <!--Germany-->
  <book name="Ring Around the Sun">
    <price>2.500000000000000e+001</price>
  </book>
</shop>

```

Как видно из результата, объявленная XML-структура повторяется для каждой записи. Для создания сложных вложений XML-структур, используются вложенные запросы FOR XML.

Кстати, если указать пустую запись после названия режима (например, FOR XML PATH ()), тогда упаковщик элементов не создается.

В ходе своего рассказа, мы не затрагивали еще один механизм работы с XML-данными в SQL Server 2008 – технологию для манипулирования XML-данными **SQLXML**. Текущая версия которой 4.0. SQLXML является API-интерфейсом среднего уровня, построенного на базе COM,

которая позволяет работать с реляционными данными без использования инструкций T-SQL. Более подробно о данной технологии можно прочитать в разделе "Основные понятия о программировании для SQLXML 4.0" электронной документации по SQL Server 2008.

4. Домашнее задание

Написать следующие пользовательские функции.

1. Функцию, которая возвращает количество магазинов, которые не продали ни одной книги издательства.
2. Функцию, которая возвращает минимальный из трех параметров.
3. Многооператорную функцию, которая возвращает количество проданных книг по каждой из тематик и в разрезе каждого магазина.
4. Функцию, которая возвращает список книг, которые соответствуют набору критериев (имя и фамилия автора, тематика), и отсортированы по фамилии автора в указанном в 4-м параметре направлении.
5. Функцию, которая возвращает среднее арифметическое цен всех книг, проданных до указанной даты.
6. Функцию, которая возвращает самую дорогую книгу издательства указанной тематики.
7. Функцию, которая по ID магазина возвращает информацию о нем (ID, название, местоположение, средняя стоимость продаж за последний год книг вашего издательства) в табличном виде.

С помощью курсоров.

1. Создайте последовательный курсор, который указывает на то, сколько каждый канадский магазин

продал книг за прошлый год. Выведите данные курсора ориентировочно в следующем виде:

| Shop | | | |
|------|-------------|------------|-------|
| Shop | HashTag | has sold 5 | books |
| Shop | Rare Books | has sold 7 | books |
| Shop | Smith&Brown | has sold 2 | books |

- Создайте локальный ключевой курсор, который содержит список тематик и информацию о том, сколько авторов пишут в каждом отдельном жанре. Сохраните значения полей в переменных. С помощью курсора и связанных переменных выполните следующие действия:
 - выведите в цикле все множество данных курсора;
 - выведите отдельно последнюю запись;
 - выведите отдельно 5-ю запись с конца;
 - выведите отдельно 3-ю запись с начала.
- С помощью локального ключевое курсора выполните позиционированное обновление данных в таблице "Themes", а именно: измените название тематики, которая стоит на 3-й позиции с самого начала.
- С помощью глобального курсора выполните позиционированное удаление 2-й с конца записи. В курсоре содержится информация об авторах, книги которых за два последних года еще ни разу не продались.

Использовать технологию XML для выполнения следующих задач.

- Создать переменную типа XML, которая содержит в иерархическом виде данные о тематике и авторах, книги которых в них представлены. Данные должны

быть отсортированы по авторам, книги которых были первые изданные издательством. От старых авторов к новым. Вывести на экран данные переменной.

2. Отфильтровать и вывести с помощью метода `query()` и XPath значение переменной с задания (1), чтобы получить список авторов только определенной тематики, например, только тематики "HTML / XHTML / XML".
3. С помощью метода `modify()` и XPath удалить все тематики с переменной задачи (1), в которых не представлено ни одного автора.
4. Необходимо разработать таблицу Order, которая будет содержать отчеты о продаже книг магазинами в следующем виде:
 - магазин, который представляет отчет (для обеспечения целостности данных связывается с таблицей Shops);
 - дата формирования отчета (по умолчанию принимается текущая дата);
 - отчет в виде XML-документа.

Форма отчета должна соответствовать определенной XML-схеме, которая содержит краткую аннотацию. Примерная структура отчета должна быть следующая:



5. По максимуму попробовать автоматизировать процесс внесения данных в таблицу.
6. Выбрать все записи из таблицы Order с использованием метода query () и XPath. Ориентировочный вид представлен в задании (4).
7. Отфильтровать результаты таблицы Order с использованием метода query () и XPath, чтобы получить только списки уцененных проданных книг в течение последнего месяца.
8. Выбрать все записи из таблицы Order с использованием метода query () и XPath, но включить при этом в XML-структуру данных, которые хранятся в реляционной инфраструктуре, например, поле ID магазина.
9. Выбрать независимые значения из таблицы Order с помощью метода value () и XPath. Верните табличную структуру, содержащую информацию обо всех

проданных книгах, цена которых более \$30, по тема-
тикам "Computer Science" и "Web Technologies".

10. С помощью метода `modify()` и XPath добавить в отчет магазина, например, "Rainbow", еще одну книгу. После вставки данных, добавьте ей отметку об уценке товара. Выполните необходимые изменения в элементе "Results".

ПРИМЕЧАНИЕ!

За каждый блок задач выставляется отдельная оценка.



Урок №8

Программирование и администрирование СУБД MS SQL Server

© Компьютерная Академия «Шаг»

www.itstep.org

Все права на охраняемые авторским правом фото-, аудио- и видеопроизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объеме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объем и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.