



Урок 8

План заняття:

1. Користувальські функції
2. Курсори в SQL Server
 - 2.1. Характеристика та типи курсорів
 - 2.2. Створення курсорів
 - 2.3. Маніпулювання записами за допомогою курсорів
 - 2.4. Модифікація та видалення даних
3. Використання XML в SQL Server 2008
 - 3.1. Тип даних XML. Колекція схем XML
 - 3.2. Вибірка та зміна XML-даних
 - 3.3. Вибірка реляційних даних в форматі XML. Конструкція FOR XML
4. Домашнє завдання

1. Користувальські функції

В SQL Server існує великий набір стандартних функцій, якими ви вже не раз користувались. Та цього набору для забезпечення функціональності бази даних інколи може бути недостатньо. Тому SQL Server надає можливість створювати свої власні користувальські функції (user-defined functions).

Типи користувальських функцій:

1. **скалярні (scalar functions)** - це функції, що повертають одне скалярне значення, тобто число, рядок тощо.
2. **вбудовані однотабличні або підставляємі табличні (inline table-valued functions)** - це функції, які повертають результат у вигляді таблиці. Повертаються вони одним оператором SELECT. Причому, якщо в результаті створюється таблиця, то імена її полів являються псевдонімами полів при вибірці даних.
3. **багатооператорні функції (multistatement table-valued functions)** - це функції, при визначенні якої задаються нові імена полів та типи.

Крім того функції розділяють по **детермінізму**. Детермінізм функції визначається постійністю її результатів. **Функція є детермінованою (deterministic function)**, якщо при одному і тому ж заданому вхідному значенні вона завжди повертає один і той же результат. Наприклад, вбудована функція **DATEADD()** являється детермінованою, оскільки додавання трьох днів до дати 5 травня 2010р. завжди дає дату 8 квітня 2010 р., або ж функція **COS()**, яка повертає косинус вказаного кута.

Функція є недетермінованою (nondeterministic function), якщо вона може повертати різні значення при одному і тому ж заданому вхідному значенні. Наприклад, вбудована функція **GETDATE()** являється недетермінованою, оскільки при кожному виклику вона повертає різні значення.

Детермінізм користувальської функції не залежить від того, являється вона скалярною чи табличною, – функції обох цих типів можуть бути як детермінованими, так і недетермінованими.

Від детермінованості функції залежить чи можна проіндексувати її результат, а також чи можна визначити кластеризований індекс на представлення, яке ссилається на цю функцію. Наприклад, недетерміновані функції не можуть бути використані для створення індексів або розрахункових полів. Щодо кластеризованого індекса, то він не може бути створений для представлення, якщо воно звертається до недетермінованої функції (незалежно від того, використовується вона в індексі чи ні).

Користувальські функції мають ряд **переваг**, серед яких основним є підвищення продуктивності виконання, оскільки функції, як і зберігаємі процедури, кешують код і повторно використовують план виконання.

Отже, розглянемо типи функцій по порядку. Почнемо з **скалярних**. Синтаксис їх оголошення наступний:

```
CREATE FUNCTION [ схема. ] ім'я_функції
    ( [ @параметр [ AS ] [ схема. ] тип
      [ = значення_по_замовчуванню | default ]
      [ READONLY ]
    ] [, ... n]
  )
RETURNS тип_повертаемого_значення
[ WITH { [ ENCRYPTION ]
        | [ SCHEMABINDING ]
        | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
        | [ EXECUTE AS контекст_безпеки]
      }
  ] [, ...n ]
```



```

]
[ AS ]
BEGIN
    тіло_функції
    RETURN (повертаєме_скалярне_значення)
END

```

Як видно з синтаксису, після імені функції, необхідно в дужках перелічити необхідні вхідні параметри, якщо вони передбачаються. Оскільки всі параметри являються локальними змінними, то перед іменем необхідно ставити символ @, після чого вказується його тип (**допускаються всі типи даних, включаючи типи даних CLR, крім timestamp (!)**). У випадку необхідності можна задати значення по замовчуванню для кожного з вхідних параметрів або ж вказати ключове слово default. Якщо визначено значення default, тоді параметру присвоюється значення по замовчуванню для даного типу.

Ключове слово **READONLY** вказує на те, що параметр не може бути оновлений або змінений при визначенні функції. Відмітимо, що якщо тип параметра являється користувацьким табличним типом, тоді обов'язково вказати ключове слово READONLY.

Далі слід вказати тип повертаємого скалярного значення для функції (**returns**). Допускаються всі типи даних, крім нескаларних типів cursor і table, а також типи rowversion (timestamp), text, ntext або image. Їх краще замінити типами uniqueidentifier та binary(8).

Параметр **WITH** задає додаткові характеристики для вхідних аргументів. З ключовим словом ENCRYPTION, SCHEMABINDING та EXECUTE AS ви вже знайомі, тому на них зупинятись не будемо. Хоча тут є кілька **зауважень**:

- параметр SCHEMABINDING не можна вказувати для функцій CLR і функцій, які ссилаються на псевдоніми типів даних;
- параметр EXECUTE AS не можна вказувати для вбудованих користувацьких функцій.

Параметр **RETURNS/CALLED** може бути представлений одним з двох значень:

- CALLED ON NULL INPUT (по замовчуванню) означає, що функція виконується і у випадках, якщо в якості аргумента передано значення NULL.
- RETURNS NULL ON NULL INPUT вказаний для функцій CLR і означає, що функція може повернути NULL значення, якщо один з аргументів рівний NULL. При цьому код самої функції SQL Server не викликає.

Тіло функції розміщується в середині блоків **BEGIN..END**, який обов'язково повинен містити оператор **RETURN** для повернення результату. При написанні коду тіла функції слід пам'ятати, що тут існує ряд обмежень, основним з яких є заборона змінювати стан довільного об'єкта бази даних або саму базу даних.

Доречі, рекурсивні функції також підтримуються. Допускається до 32 рівнів вкладеності.

Викликати скалярну функцію можна одним з двох способів: за допомогою оператора **select** або **execute**.

```

SELECT ім'я_функції (параметр1 [,... n])
EXEC[UTE] @змінна = ім'я_функції параметр1 [,... n]

```

Напишемо функцію, яка повертає день тижня по вказаній в якості параметра даті.

```

create function DayOfWeek (@day datetime)
returns nvarchar (15)
as
begin
    declare @wday nvarchar (15)
    if (datetime(dw, @day)='Monday')
        set @wday = 'понеділок'
    if (datetime(dw, @day)='Friday')
        set @wday = 'п'ятниц'я
    else
        set @wday = 'інший'
    return @wday
end;

-- ВИКЛИК
select DayOfWeek (GETDATE ()) as 'День тижня';

```

Результат:

Results

Messages

	День тижня
1	п'ятниця



Вбудовані табличні функції підпорядковуються тим же правилам, що і скалярні. Їх відмінність від останніх полягає у тому, що вони повертають результат у вигляді таблиці. Табличні функції являються непоганою альтернативою представленням та зберіганням процедур. Наприклад, недоліком представлень є те, що вони не можуть приймати параметри, які часом необхідно передати. Зберігаємі процедури в свою чергу можуть приймати параметри, але не можуть бути використані у виразі **FROM** оператора **SELECT**, що дещо ускладнює обробку результатів. Табличні функції вирішують вищеперелічені проблеми.

Отже, синтаксис однотабличної функції виглядає так:

```
CREATE FUNCTION [ схема. ] ім'я_функції
    ( [ @параметр [ AS ] [ схема. ] тип
      [ = значення_по_замовчуванню | default ]
      [ READONLY ]
    ] [, ... n]
  )

RETURNS TABLE
[ WITH {
    [ ENCRYPTION ]
    | [ SCHEMABINDING ]
    | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
    | [ EXECUTE AS контекст_безпеки ]
  }
  [ , ...n ]
]
[ AS ]
BEGIN
    тіло_функції
    RETURN [ ( ) оператор SELECT [ ) ]
END
```

Викликається таблична функція наступним чином:

```
select * from ім'я_функції (параметр1 [, ... n])
```

Наприклад, напишемо функцію, яка виводить назви книг та кількість магазинів, що їх продають.

```
create function countMagazines ()
returns table
as
return (select b.NameBook as 'Назва книги',
            count(sh.id_shop) as 'Кількість магазинів'
        from book.Books b, sale.Shops sh, sale.Sales s
        where b.id_book=s.id_book and s.id_shop=sh.id_shop
        group by b.NameBook);

-- ВИКЛИК
select * from countMagazines();
```

Результат:

ID_...	NameBook	ID_THEME
16	Windows 2000 Professional. Руководство Питера Нортон	14
17	Максимальная безопасность в Linux	15
18	Путь к LINUX. 2е изд.	15
19	Как программировать на C	16

ID_SALE	ID_BOOK	DateOfSale	Pri
1	7	12.03.2000 0:0...	25,
2	14	25.02.1999 0:0...	110
3	16	17.11.2005 0:0...	90,
4	16	26.12.2005 0:0...	92,
5	8	25.11.2001 0:0...	40,

Назва книги	Кількість магази...
1 JavaScript: сборник рецептов для профессионалов	3
2 Windows 2000 Professional. Руководство Питера Нор...	2
3 Windows NT 5 перспектива	1
4 Как программировать на C	2



Синтаксис багатооператорної функції:

```
CREATE FUNCTION [ схема. ] ім'я_функції
    ( [ @параметр [ AS ] [ схема. ] тип
      [ = значення_по_замовчуванню | default ]
      [ READONLY ]
    ] [, ... n]
  )
  RETURNS @повертаема_таблиця
    TABLE структура_таблиці
  [ WITH { [ ENCRYPTION ]
    | [ SCHEMABINDING ]
    | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
    | [ EXECUTE AS контекст_безпеки]
  }
    [ , ...n ]
  ]
  [ AS ]
BEGIN
    тіло_функції
RETURN
END
```

Відмітимо, що в багатооператорній функції повертаема таблиця не обов'язково створюється оператором select. Звідси походить і назва функції. Тут можна, наприклад, виконувати попередню обробку даних і створювати тимчасову таблицю, після чого доопрацювати її і повернути нову таблицю у викликаючу програму.

Слід також бути уважним, оскільки блок тіла функції може містити кілька операторів SELECT. В такому випадку у виразі RETURNS потрібно явно визначити таблицю, яка буде повертатись. Крім того, оскільки оператор RETURN в багатооператорній табличній функції завжди повертає таблицю, яка задана в RETURNS, то він повинен виконуватись без аргументів. Наприклад, **RETURN**, а не **RETURN @myTable**.

Викликається багатооператорна таблична функція подібно вбудованій табличній, тобто за допомогою оператора SELECT:

```
select * from ім'я_функції (параметр1 [, ... n])
```

В якості прикладу напишемо багатооператорну табличну функцію, яка повертає назву магазину (-ів), який продав найбільшу кількість книг.

Даний процес можна поділити на **два етапи**:

- 1) перший - створимо тимчасову таблицю, що повертає назву книги і кількість магазинів, що їх продають;
- 2) другий - отримуємо магазин, що продав максимальну кількість книг.

```
create function BestMagazine ()
returns @tableBestMagazine table ( nameM varchar(30) not null,
                                  countBooks int not null)
as
begin
-- створюємо тимчасову таблицю, що повертає назву книги і кількість магазинів, що її продають
declare @tmpTable table ( id_book int not null,
                          numBooks int not null)

insert @tmpTable
    select b.id_book as 'Код',
           count(s.id_shop) as 'Кількість книг'
    from book.Books b, sale.Sales s
    where b.id_book=s.id_book
    group by b.id_book

-- заповнюємо повертаему таблицю за допомогою об'єднання тимчасової таблиці з іншою
insert @tableBestMagazine
    select sh.NameShop as 'Назва магазину',
           'Кількість магазинів' = max(tt.numBooks)
    from @tmpTable tt, sale.Sales s, sale.Shops sh
    where tt.id_book=s.id_book and s.id_shop=sh.id_shop
    group by sh.NameShop

return
```



```
end;

-- ВИКЛИК
select * from BestMagazine();
```

Результат:

	nameM	countBooks
1	All about PC	3
2	Book	2
3	Booksworld	3
4	Букинист	3
5	Книга	3
6	Світ книги	3
7	Слово	2

Змінити існуючу функцію користувача можна за допомогою оператора **ALTER FUNCTION**:

```
-- скалярна функція
ALTER FUNCTION [ схема. ] ім'я_функції
    ( [ @параметр [ AS ] [ схема. ] тип
        [ = значення_по_замовчуванню | default ]
        [ READONLY ]
    ][, ... n]
    )
RETURNS тип_повертаемого_значення
[ WITH { [ ENCRYPTION ]
    | [ SCHEMABINDING ]
    | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
    | [ EXECUTE AS контекст_безпеки]
}
[ , ...n ]
]
[ AS ]
BEGIN
    тіло_функції
    RETURN (повертаєме_скалярне_значення)
END

-- вбудована обнотаблична функція
ALTER FUNCTION [ схема. ] ім'я_функції
    ( [ @параметр [ AS ] [ схема. ] тип
        [ = значення_по_замовчуванню | default ]
        [ READONLY ]
    ][, ... n]
    )
RETURNS TABLE
[ WITH { [ ENCRYPTION ]
    | [ SCHEMABINDING ]
    | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
    | [ EXECUTE AS контекст_безпеки]
}
[ , ...n ]
]
[ AS ]
BEGIN
    тіло_функції
    RETURN [ ( ) оператор SELECT [ ) ]
END
```



```
-- багатооператорна функція
ALTER FUNCTION [ схема. ] ім'я_функції
    ( [ @параметр [ AS ] [ схема. ] тип
      [ = значення_по_замовчуванню | default ]
      [ READONLY ]
    ) [ , ... n ]
)

RETURNS @повертаема_таблиця
    TABLE структура_таблиці

[ WITH { [ ENCRYPTION ]
        | [ SCHEMABINDING ]
        | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
        | [ EXECUTE AS контекст_безпеки ]
      }
    [ , ... n ]
]
[ AS ]
BEGIN
    тіло_функції
RETURN
END
```

Не дивлячись на те, що за допомогою оператора ALTER FUNCTION можна змінити функцію практично повністю, використовувати даний оператор для перетворення скалярної функції в табличну чи навпаки заборонено. За допомогою ALTER FUNCTION також не можна перетворювати функцію T-SQL в функцію середовища CLR або навпаки.

Видалення користувацької функції здійснюється оператором **DROP FUNCTION**:

```
DROP FUNCTION [ схема. ] ім'я_функції [ , ... n ]
```

Отримати список користувацьких функцій можна з системного представлення **sys.sql_modules**, а список параметрів кожної з них розміщується в представленні **sys.parameters**. Список користувацьких функцій CLR розміщується по іншій адресі, а саме в системному представленні **sys.assembly_modules**.

2. Курсори в SQL Server

2.1. Характеристики та типи курсорів

Однією з характерних особливостей операторів мови SQL таких як SELECT, UPDATE та DELETE є те, що вони працюють одразу з множиною записів. З однієї сторони, така обробка корисна, наприклад, при зміні ціну товару і дозволяє обійтись без багаторазового виклику одного запиту. Але вона не завжди зручна для прикладних клієнтських додатків, які виконують рядкову обробку даних, а також у ряді випадків, коли необхідно обробляти результати рядково. Для вирішення таких задач в SQL передбачені **курсори**.

Курсори – це тимчасові об'єкти, які дозволяють організувати рядкову обробку набору даних. З їх допомогою можна в циклі пройти по результуючому набору, окремо зчитуючи та обробляючи кожен його рядок. В залежності від призначення створюваного курсора, ви можете переміщувати курсор в середині множини модифікуючи або знищуючи дані. Слід відмітити, що необхідність у використанні курсора дуже низька, вони ресурсоємкі і тому перед їх використанням слід впевнитись, що курсор дійсно забезпечить підвищення продуктивності запиту.

Робота з результуючою множиною, яка поміщається в курсор, здійснюється за наступною схемою:

1. Спочатку слід оголосити курсор. Оголошення курсора здійснюється за допомогою оператора **DECLARE CURSOR**. Слід відмітити, що курсор та зв'язаний з ним результуючий набір повинні мати однакові імена. Це ім'я вказується при оголошенні курсора.
2. Відкриття курсора для роботи, тобто заповнення його даними - оператор **OPEN**. Результуючий набір в курсорі після його відкриття залишається відкритим до тих пір, поки він не буде закритий явно.
3. Для основної роботи з курсором, тобто для переміщення в наборі з одного запису на інший, використовуються спеціальні команди:
 - 3.1. Вибірка записів за допомогою курсора – оператор **FETCH**.
 - 3.2. Для позиціонованого оновлення використовується інструкція **WHERE CURRENT OF** для UPDATE або DELETE.



4. Закриття курсора і очистка поточного результуючого набору – оператор **CLOSE**.
5. Вивільнення ресурсів, які використовує курсор, і видалення його як об'єкта - оператор **DEALLOCATE**.

Курсори володіють рядом характеристик, до яких можна віднести:

- ✓ **Область бачення** – в яких з'єднаннях і процесах може бути отриманий доступ до курсора.
- ✓ **Чутливість курсора** - здатність відображати зміни у вихідних даних.
- ✓ **Прокрутка** - здатність здійснювати прокрутку як вперед, так і назад в множині записів. При цьому послідовні курсори (forward only) працюють значно швидше, але мають меншу гнучкість.
- ✓ **Обновлення** - здатність модифікувати (оновлювати) множину записів. Такі курсори використовуються лише для читання (read only), вони як правило більш продуктивні, але менш гнучкі.

MS SQL Server підтримує три види курсорів:

1. **Курсори T-SQL**, які використовуються всередині тригерів, зберігаємих процедур та сценаріїв. Їх ми розглянемо детальніше.
2. **Серверні курсори API**, які діють на сервері та реалізують програмний інтерфейс прикладних додатків для ODBC, ADO та ADO.NET, OLE DB і DB-Library. Кожне з цих API використовує різний синтаксис та відрізняється по функціональним характеристикам.
3. **Клієнтські курсори** реалізуються на самому клієнті. Вони вибирають весь результуючий набір записів з сервера і зберігають його локально, що дозволяє прискорити операції обробки даних за рахунок зниження затрат часу на виконання мережевих операцій.

Всі курсори можна поділити на 4 типи:

- a) **статичні (static cursors)** – при його створенні робиться «знімок» поміщаємих в курсор даних. Він не чутливий до змін в структурі або в значеннях даних, тобто будь-які зміни у вхідних даних в курсорі відображені не будуть. В зв'язку з цим такі типи курсорів використовуються дуже рідко, а на рівні клієнт-серверних додатків вони взагалі недоречні. Замість них дуже часто використовують таблиці бази даних **tempdb**, в яких зберігаються дані курсора, або ж створюють тимчасові таблиці за допомогою оператора SELECT INTO. Якщо все ж таки статичний курсор необхідний, тоді його відкривають лише для читання.
Статичні курсори можуть бути послідовними і прокручуваними.
- b) **динамічні (dynamic cursors)** – це найпотужніший та гнучкий тип курсора, але і найбільш ресурсоемкий. Він відображає всі зміни, які вносяться користувачами в базові таблиці, іншими словами підтримує дані в «живому» стані.
- c) **ключові (keyset-driven cursors)** – основані на використанні набору ключів, який визначається даними, що унікально ідентифікують кожен запис в таблиці бази даних. Тому в таблиці keyset бази даних tempdb зберігаються лише набори ключів, а не весь набір даних. Щоб оголосити ключовий курсор, кожна таблиця, яка входить в множину даних курсора, повинна мати унікальний індекс (як правило, це індекс первинного ключа), який задає набір для копіювання, – **ключ**.
Такі курсори не чутливі до вставки або видалення даних, які виникають після створення курсора, лише до змін. Тому ключові курсори здебільшого використовуються в якості основи для створення курсорів на оновлення даних.
Ключові курсори можуть бути модифікованими, лише для читання (read only), послідовними та з прокруткою.
- d) **послідовні (fast forward cursors; однонаправлені, “пожежні” (firehose))** – являє собою швидкий однонаправлений курсор. Однонаправленими їх називають тому, що після отримання даних за допомогою такого курсора відсутня можливість повернути відкоректовані дані в ту ж таблицю. Він відкривається лише для читання, не дозволяє виконувати вибірку даних в зворотньому напрямі, не підтримує прокрутку і приводиться автоматично до курсора будь-якого іншого типу у наступних випадках:
 - якщо в базовому запиті використовуються поля типу text, ntext, image або оператор TOP, SQL Server перетворює курсор в ключовий;
 - якщо послідовний курсор оголошений як курсор FOR UPDATE, тоді він перетворюється в ключовий;
 - якщо послідовний курсор оголошений як курсор FOR UPDATE, але хоча б одна базова таблиця не має унікального індекса, тоді курсор перетворюється в статичний;
 - якщо базовий запит створює тимчасову таблицю курсор також перетворюється в статичний.

Серед такого різномайття курсорів по волі стикаєшся з питанням: а який же тип курсора обрати? Статичний курсор через його надмірну ресурсоемкість, особливо у клієнт-серверних додатках краще не використовувати. Якщо необхідно лише переглядати дані і достатньо однонаправленого перегляду, тоді підійде послідовний курсор. Оптимальним вибором і в локальному, і в клієнт-серверному додатку буде динамічний курсор. Але, якщо в останньому випадку (клієнт-сервер) ресурси обмежені, тоді краще зупинити свій вибір на ключовому курсорі.

Використання курсорів схоже з використанням звичайних локальних змінних – їх потрібно спочатку оголосити, встановити значення, а потім можна використовувати. Але на відміну від локальних змінних, які автоматично знищуються при виході з області бачення, курсори необхідно закрити, вивільнивши при цьому використовувані дані, а потім знищити.



Для отримання інформації про **характеристики курсора** використовують наступні **системні зберігаємі процедури**:

- **sp_cursor_list** - список курсорів разом з атрибутами, доступних для з'єднання в поточний момент часу;
- **sp_describe_cursor** - описує атрибути курсора (тип курсора, прокрутка тощо);
- **sp_describe_cursor_columns** – описує атрибути полів результуючого набору;
- **sp_describe_cursor_tables** – описує базові таблиці, до яких має доступ курсор.

2.2. Створення курсорів

Для того, щоб створити курсор використовується оператор **DECLARE CURSOR**. SQL Server 2008 підтримує два формати оголошення курсора:

```
-- синтаксис ISO
DECLARE ім'я_курсора
    [ INSENSITIVE ]
    [ SCROLL ]          /* прокрутка */
    CURSOR
    FOR оператор_select /* записи, які будуть включені в множину курсора */
    [ FOR { READ ONLY | UPDATE [ OF ім'я_поля [ ,...n ] ] } ]

-- синтаксис Transact-SQL
DECLARE ім'я_курсора CURSOR
    [ LOCAL | GLOBAL ]          /* область бачення */
    [ FORWARD_ONLY | SCROLL ]  /* прокрутка */
    [ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ] /* тип курсора */
    [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ] /* блокування */
    [ TYPE_WARNING ]
    FOR оператор_select /* записи, які будуть включені в множину курсора */
    [ FOR UPDATE [ OF ім'я_поля [ ,...n ] ] ]
```

Розберемо коротко кожен з параметрів даного оператора:

- **Область бачення** курсора визначається за допомогою ключових слів **LOCAL** або **GLOBAL**, які рівносильні змінним **@локальна_таблиця** та **@@глобальна_таблиця**, або префіксам **#** і **##** при оголошенні тимчасових таблиць. По замовчуванню область бачення курсора визначається параметром **default to local cursor** бази даних.

Щодо глобального курсора, то в поточному з'єднанні, крім поточного процесу, до нього мають доступ і інші процеси, такі як пакети, тригери, зберігаємі процедури. Простішими словами, курсор можна створити за допомогою однієї зберігаємої процедури і звернутись до нього з іншої без передачі на нього посилання. Локальний курсор вказує на доступ до нього лише з поточного процесу поточного з'єднання.

SQL Server закриває і вивільняє локальний курсор при виході за межі його області дії, але краще зробити це самостійно.

- **Прокрутка** задає можливість переміщення курсора: тільки з початку в кінець (**FORWARD_ONLY**) або в довільному напрямку (**SCROLL**). По замовчуванню допускається лише послідовне переміщення курсора, тому у випадку необхідності переміщення на попередній запис слід перевірити курсор. При встановленні параметра прокрутки не слід забувати про те, який тип курсора ви створюєте.
- **Тип курсора** по замовчуванню динамічний (**DYNAMIC**). Нагадаємо, що параметр **FAST_FORWARD** (для створення послідовного курсора) не може бути вказаний разом з параметрами **SCROLL** або **FOR_UPDATE**.
- **Блокування** визначає чи можуть записи модифікуватись курсором, і якщо так, то чи можуть їх модифікувати інші користувачі. Блокування дозволяє мінімізувати проблеми, пов'язані з організацією паралельної роботи на сервері при наявності курсора. Для цього призначені три **опції**:

- **READ_ONLY** - курсор тільки для читання.
- **SCROLL_LOCKS** – здійснює блокування прокрутки (інколи називають «песимістичним блокуванням»). Даний параметр забороняє будь-якому користувачу вносити зміни в запис, якщо він в цей момент модифікується курсором. SQL Server буде блокувати записи по мірі зчитування їх в курсор, для забезпечення їх доступності для подальших змін.

Не слід також забувати, що параметр **SCROLL_LOCKS** не підтримується послідовними та статичними курсорами.

- **OPTIMISTIC** – не здійснює блокування прокрутки, так зване «оптимістичне блокування». SQL Server не блокує записи під час їх зчитування в курсор, вважаючи, що в цей час ніхто не працює з цими даними (не оновлює їх і не видаляє). Така ситуація доволі оптимістична, але цілком реальна, коли база даних велика, а користувачів мало. При цьому, якщо дані все ж хтось змінив, тоді операції оновлення і видалення, які здійснюються через курсор, працювати не будуть і для здійснення подальших дій необхідно перезаповнювати курсор. Для визначення того, чи



змінилися дані запису після зчитування їх в курсор, SQL Server виконує порівняння значень поля timestamp (якщо воно існує) або контрольних сум.

Параметр OPTIMISTIC не можна вказувати для послідовних (FAST_FORWARD) курсорів.

- Параметр **TYPE_WARNING** передбачає відправку SQL Server'ом попередження, якщо тип курсора приводиться від заданого до іншого типу, тобто у випадку неявного приведення до типу.
- **FOR UPDATE** визначає поля в курсорі, які будуть обновлюватись.
- В розділі **OF**, якщо він вказаний, міститься перелік полів, яких будуть стосуватись зміни, всі інші поля будуть розглянуті як призначені тільки для читання. У випадку відсутності списку, обновлення стосуватиметься всіх полів, які вказані в списку SELECT при створенні курсора.

При використанні синтаксиса ISO використовується параметр **INSENSITIVE**, який відсутній в синтаксисі створення курсора стандарту T-SQL. Він дозволяє створити курсор, в якому не будуть відображатись зміни (обновлення або видалення), здійснені в базових таблицях. Такий курсор працює лише з даними тимчасової таблиці бази даних tempdb, яка була заповнена при створенні курсора. В зв'язку з цим курсор, створений з параметром INSENSITIVE, не може використовуватись для зміни даних в базових таблицях.

Множина записів, на які вказує курсор, визначається за допомогою оператора **SELECT**. При цьому інструкція SELECT не може:

- повертати кілька результуючих множин;
- містити ключове слово INTO для створення нової таблиці;
- містити ключові слова COMPUTE, COMPUTE BY та FOR BROWSE, але може містити функції агрегування.

ПРИМІТКА! Якщо при створенні курсора не вказуються параметри блокування, тоді приймаються наступні значення по замовчуванню:

- якщо оператор SELECT не підтримує обновлення (наприклад, недостатньо привілеїв), тоді курсору присвоюється параметр READ_ONLY;
- статичні і послідовні курсори по замовчуванню мають значення READ_ONLY;
- динамічні та ключові курсори по замовчуванню мають значення OPTIMISTIC.

Отже, курсор оголосили, але декларація курсора не створює набір записів, якими курсор буде маніпулювати. Для того, щоб створити власне множину записів курсора, необхідно його відкрити за допомогою оператора **OPEN**:

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

Після відкриття курсора виконується оператор SELECT і здійснюється вибірка даних з вказаних таблиць в курсор, після чого можна приступати до основної роботи.

Ключове слово GLOBAL в операторі OPEN допомагає уникнути конфліктів імен, оскільки у випадку наявності двох курсорів з однаковими ідентифікаторами по замовчуванню всі посилання будуть стосуватись локального курсора.

ПРИМІТКА! Для отримання кількості вибраних записів в останньому відкритому курсорі, слід скористатись глобальною змінною @@CURSOR_ROWS.

Закриває відкритий курсор і вивільняє поточний результуючий набір оператор **CLOSE**:

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

Оператор CLOSE залишає структури даних доступними для повторного відкриття, але вибірка і обновлення даних заборонені.

Видалення посилання курсора здійснюється за допомогою оператора **DEALLOCATE**. При цьому, коли знищується останнє посилання курсора, SQL Server вивільняє структури даних, які входили в курсор. Синтаксис даного оператора наступний:

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

Наведемо невеличкий приклад створення курсора та заповнення його даними.

```
-- створюємо курсор
declare myCursor cursor
for select *
from book.Books
```



```
-- відкриваємо курсор, тобто створюємо множину записів курсора
open myCursor

/* маніпулювання записами за допомогою курсора */

-- закриваємо курсор
close myCursor

-- вивільняємо курсор
deallocate myCursor
```

Доречі, T-SQL дозволяє оголошувати змінні типу **CURSOR**. Змінна такого типу створюється звичним чином, а значення їй присвоюється за допомогою оператора **SET**. Наприклад:

```
declare myCursor cursor local fast_forward
for
select NameBook
from book.Books

declare @myCursorVariable cursor /* Створюємо змінну курсора */

open myCursor

set @myCursorVariable = myCursor /* Назначаємо змінну курсору */

/* маніпулювання записами за допомогою курсора або через асоційовану змінну */

close myCursor
deallocate myCursor
```

Слід відмітити, що після вивільнення курсора, ідентифікатор myCursor більше не асоціюється з набором записів курсора. Але на множину курсора ще ссилається змінна @cursorVariable, тому курсор і його множина не вивільняються, поки явно не вільнити і курсорну змінну. Фактично курсор і його набір записів будуть існувати до тих пір, поки змінна не втратить свою дію.

2.3. Маніпулювання записами за допомогою курсорів

Після створення та відкриття курсора можна приступати до основної роботи з ним. Для цього необхідно зробити наступні дії:

- Отримати перший запис за допомогою оператора **FETCH**.
- По мірі необхідності провести обробку в циклі інших записів за допомогою оператора **FETCH**.

Синтаксис оператора **FETCH** наступний:

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

Розшифруємо ключові слова, які відповідають за напрямок курсора:

- NEXT** - повертає наступний запис в курсорі після поточного. Якщо вибірка здійснюється вперше, тоді повертається перший запис результуючого набору.
- PRIOR** - повертає запис, який знаходиться в курсорі перед поточним (попередній запис). Якщо вибірка з курсора здійснюється вперше, тоді ніякий запис не повертається і положення курсора залишається перед першим записом.
- FIRST** – повертає перший запис в курсорі і робить його поточним.
- LAST** - повертає останній запис в курсорі і робить його поточним.
- ABSOLUTE** – варіанти дій наступні:
 - якщо значення параметру (n або @nvar) має позитивне значення, тоді повертається запис, який віддалений на n записів від початку курсора;



- якщо значення від'ємне - повертається запис, який віддалений на n записів від кінця курсора;
- якщо n або @nvar рівні 0, тоді записи не повертаються.

• **RELATIVE** - варіанти дій наступні:

- якщо значення параметру (n або @nvar) має позитивне значення, тоді повертається запис, який віддалений на n записів від поточного;
- якщо значення від'ємне - повертається запис, який передус на n записів поточному;
- якщо n або @nvar рівні 0, тоді повертається поточний запис. Але, якщо при першій вибірці вказується негативне або нульове значення, тоді записи не повертаються.

В параметрах ABSOLUTE та RELATIVE значення n повинно бути цілочисельною константою, а @nvar повинно мати тип smallint, tinyint або int.

Параметр INTO дозволяє помістити дані з полів вибірки в локальні змінні. Кожна змінна з списку зв'язується з відповідним полем в результируючому наборі курсора. При цьому типи даних повинні співпадати або підтримувати неявне приведення до типу.

ПРИМІТКИ!

- Якщо в інструкції DECLARE CURSOR стандарту ISO не вказаний параметр SCROLL, то єдиним параметром інструкції FETCH може бути NEXT.
- В стандарті Transact-SQL при оголошенні курсора діють наступні правила:
 - якщо вказаний FORWARD_ONLY або FAST_FORWARD, тоді оператор FETCH підтримує лише NEXT;
 - якщо не вказані DYNAMIC, FORWARD_ONLY або FAST_FORWARD і вказаний один з параметрів KEYSET, STATIC або SCROLL, тоді підтримуються всі параметри оператора FETCH;
 - курси DYNAMIC SCROLL підтримують всі параметри оператора FETCH, крім ABSOLUTE.

Отже, розглянемо ряд прикладів на використання курсора. Для початку розглянемо простий приклад, в якому за допомогою динамічного курсора повертається запис в його поточній позиції.

```
-- оголошуємо курсор
declare auth_cursor cursor
for
select FirstName, LastName
from book.Authors;

-- відкриваємо курсор
open auth_cursor;

-- отримуємо перший запис за допомогою курсора. Всі наступні три запити рівносильні
fetch next from auth_cursor;
-- fetch from auth_cursor;
-- fetch auth_cursor;

-- закриваємо та вивільняємо курсор
close auth_cursor;
deallocate auth_cursor;
go
```

Результат:

вхідна таблиця

	ID	AUTHOR	FirstName	LastName	ID	COUNTRY
1	1		Ричард	Беймаер	4	
2	2		Johnson	White	3	
3	3		Marinria	Green	2	

результируюча множина

	FirstName	LastName
1	Ричард	Беймаер

Якщо ви звернете свою увагу на швидкодію сценарію, то помітите, що він виконується довше, ніж відповідний оператор SELECT. Це відбувається тому, що операції створення і відкриття курсора потребують додаткового часу. В зв'язку з цим повторимось, що віддати перевагу курсорам слід в саму останню чергу, якщо без них дійсно не обійтись.



Розширимо наш приклад та виведемо на екран всю множину даних курсора. Для цього скористаємось циклом:

```
-- оголошуємо курсор
declare auth_cursor cursor
for
select FirstName, LastName
from book.Authors;

-- відкриваємо курсор
open auth_cursor;

-- отримуємо перший запис за допомогою курсора
fetch next from auth_cursor;

-- перебираємо всі записи, поки вони існують
while @@FETCH_STATUS = 0
begin
    -- переходимо на наступний запис
    fetch next from auth_cursor;
end

-- закриваємо та вивільняємо курсор
close auth_cursor;
deallocate auth_cursor;
go
```

Результат:

Results		Messages	
FirstName	LastName		
1	Ричард Веймаер		
FirstName	LastName		
1	Johnson White		
FirstName	LastName		
1	Marjorie Green		
FirstName	LastName		
1	Maender Smith		

В нашому прикладі ми використали глобальну змінну **@@FETCH_STATUS**. Дана змінна повертає інформацію про стан виконання останньої команди FETCH, тобто дозволяє визначити, чим закінчилась операція отримання даних. Змінна **@@FETCH_STATUS** може мати **наступні значення**:

- 0 - вибірка виконана успішно;
- -1 – вибірка завершилась невдачею;
- -2 – помилка, пов'язана з відсутністю необхідного запису (спроба зчитати запис після останнього або перед першим).

Оператор FETCH може також зберігати значення з повертаемого поля в змінні. Продемонструємо це на вже розглянутому прикладі:

```
-- оголошуємо курсор
declare auth_cursor cursor
for
select FirstName, LastName
from book.Authors;

-- оголошуємо змінні для збереження даних, які повертаються оператором FETCH
declare @fname varchar(50), @lname varchar(50);

-- відкриваємо курсор
open auth_cursor;

-- отримуємо перший запис та зберігаємо значення полів в змінних
fetch next from auth_cursor
into @fname, @lname;

while @@FETCH_STATUS = 0
begin
    -- виводимо на екран поточні значення змінних
    print 'Name: ' + @fname + ' ' + @lname
end
```



```

    fetch next from auth_cursor
        into @fname, @lname;
end

-- закриваємо та вивільняємо курсор
close auth_cursor;
deallocate auth_cursor;
go

```

Результат:

```

Messages
Name: Ричард Веймаер
Name: Johnson White
Name: Marjorie Green
Name: Meander Smith
Name: Livia Karsen
Name: Сергей Парижский
Name: Сергей Михайлов
Name: Тарас Тимошок
Name: Андрей Ковязин
Name: Михаил Востриков
Name: Михаил Моченный

```

У попередніх прикладах оператор FETCH використовувався для повернення поточного запису, але даний оператор дозволяє переглядати і інші записи, роблячи їх поточними. Наприклад, створимо локальний динамічний курсор з прокруткою, який містить набір даних про магазини видавництва.

```

-- оголошуємо курсор
declare shop_cursor cursor local scroll dynamic
for
select sh.NameShop, c.NameCountry
from sale.Shops sh, global.Country c
where sh.ID_COUNTRY=c.ID_COUNTRY
order by 1;

-- відкриваємо курсор
open shop_cursor;

-- переходимо на останній запис в курсорі
fetch last from shop_cursor;

-- переходимо на попередній запис від поточної позиції курсора
fetch prior from shop_cursor;

-- переходимо на другий запис в курсорі
fetch absolute 2 from shop_cursor;

-- переходимо на третій запис після поточного
fetch relative 3 from shop_cursor;

-- переміщуємось на два рядки назад відносно поточного
fetch relative -2 from shop_cursor;

-- закриваємо та вивільняємо курсор
close shop_cursor;
deallocate shop_cursor;
go

```



Результат:

Множина даних курсора

	NameShop	NameCountry
1	All about PC	Великобританія
2	Book	США
3	Booksworld	США
4	Букинист	Україна
5	Книга	Росія
6	Світ книги	Україна
7	Слово	Україна

Результати дій над курсором

	NameShop	NameCountry	
1	Слово	Україна	FETCH LAST
1	Світ книги	Україна	FETCH PRIOR
1	Book	США	FETCH ABSOLUTE 2
1	Книга	Росія	FETCH RELATIVE 3
1	Booksworld	США	FETCH RELATIVE -2

2.4. Модифікація та видалення даних

Якщо ваш курсор являється модифікованим, тоді ви можете змінювати вхідні дані в множині курсора. Для цього в T-SQL використовується спеціальна форма інструкції WHERE для операторів UPDATE або DELETE - **WHERE CURRENT OF**.

З її використанням синтаксис операторів UPDATE та DELETE набуде наступного вигляду:

```
-- оператор UPDATE для позиціонованого оновлення даних
UPDATE { таблиця | представлення }
SET назва_поля = значення
WHERE CURRENT OF { [ GLOBAL ] ім'я_курсора | змінна_курсора }

-- оператор DELETE для позиціонованого видалення даних
DELETE
FROM { таблиця | представлення }
WHERE CURRENT OF { [ GLOBAL ] ім'я_курсора | змінна_курсора }
```

Інструкція CURRENT OF в позиціонованих оновленнях або видаленнях використовується для вказання використовуваного курсора. При цьому операція буде виконуватись в поточному положенні курсора, а ключове слово GLOBAL вказує на те, що операція стосується глобального курсора.

Наприклад, створимо курсор, за допомогою якого здійснимо позиціоноване оновлення даних таблиці «Country», а саме: переіменуємо назву другої країни в списку в значення «США».

```
-- оголошуємо та заповнюємо курсор даними
declare cnt_cursor cursor local keyset
for
select ID_COUNTRY, NameCountry
from global.Country
for update;

open cnt_cursor;

-- переміщуємось на 2-й запис
fetch absolute 2 from cnt_cursor;

-- здійснюємо позиціоноване оновлення
update global.Country
set NameCountry = 'США'
where current of cnt_cursor;
```



```
-- перевіряємо дані
select *
from global.Country
where NameCountry = 'США';

-- закриваємо та вивільняємо курсор
close cnt_cursor;
deallocate cnt_cursor;
```

Результат:

Results		
ID	COUNTRY	NameCountry
1	2	Росія

Messages		
ID	COUNTRY	NameCountry
1	2	США
2	3	США

В результаті роботи сценарію, відображаються дві панелі сітки. В першій панелі відображаються початкові значення полів, а в другій містяться значення після зміни.

А тепер розглянемо приклад позиціонованого видалення для останнього запису курсора. В курсор помістимо набір даних про книги, ціна яких вища середньої ціни продажу книг видавництва за поточний рік.

```
-- оголошуємо 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()))
);

-- відкриваємо курсор
open qbook_cursor;

-- переміщуємось на перший запис
fetch from qbook_cursor;

-- видаляємо запис, на який вказує курсор
delete
from book.Books
where current of qbook_cursor;

-- закриваємо та вивільняємо курсор
close qbook_cursor;
deallocate qbook_cursor;
```

3. Використання XML в SQL Server 2008

3.1. Тип даних XML. Колекція схем XML

Підтримка реляційними базами даних обробки XML-даних являє собою потужний механізм по організації збереження та роботи з неструктурованими даними. Починаючи з версії SQL Server 2000, в ядро SQL Server була інтегрована велика кількість вбудованих засобів підтримки мови XML. А після того, як в 2003 році Міжнародна організація по стандартизації (ISO) та Американський національний інститут стандартів (ANSI) затвердили основні правила обробки XML-даних реляційними базами даних, в SQL Server 2005 був включений тип даних XML, який підтримується і зараз.

SQL Server має ряд **переваг** щодо збереження даних в форматі XML, серед яких можна виділити наступні:

- XML-дані мають вигляд ієрархічної деревовидної структури, яка завжди повинна містити кореневий елемент, який називають **XML-документом (XML document)**. Якщо XML-дані організовані без наявності кореневого вузла, тоді він називається **фрагментом XML (XML fragment)**.



- SQL Server підтримує колекцію схем XML, які містять набір правил по структурі XML документів, які зберігаються в базі даних.
- По XML-даним можна виконувати пошук. Для цього в SQL Server існує підтримка мов XPath та XQuery.
- Ви можете обробляти XML-дані, вставляти, модифікувати або видаляти необхідні вузли.

SQL Server 2008 дозволяє **зберігати XML-дані двома способами**, кожен з яких має свої переваги та недоліки:

1. Збереження **в текстових полях** одного з типів (n)char, (n)varchar або varbinary.

Основні переваги збереження даних таким способом:

- XML зберігає точність передачі тексту, включаючи коментарі;
- не залежить від можливостей бази даних в обробці XML-даних;
- при зміні даних забезпечується висока продуктивність, оскільки вся робота зводиться до дій з звичайними текстовими даними.

Основні недоліки:

- відсутня можливість працювати на рівні вузла;
- низька продуктивність при пошуку даних, оскільки доведеться зчитувати всі дані.

2. Збереження **в полях з типом даних XML**.

Основні переваги збереження даних таким способом:

- дані зберігаються і обробляються як XML;
- зберігається порядок і структура XML-документа;
- підтримується робота з даними на рівні вузла (зміна, вставка, видалення);
- підвищення продуктивності операцій вилучення даних, оскільки для типу даних XML можна створювати кілька індексів;
- швидка продуктивність пошуку, за рахунок підтримки механізмів пошуку XML-даних: мов XPath та XQuery.

Основні недоліки:

- не зберігається точність передачі тексту, наприклад, коментарі та XML-декларація тощо;
- існує обмеження на вкладеність вузлів – максимально 128 рівнів.

Слід відмітити, що при збереженні XML-даних будь-яким з двох способів, максимальний допустимий розмір даних – 2 Гб. Але слід розрізняти, що в першому випадку – це 2 Гб звичайних текстових даних, а в другому безпосередньо XML-даних.

В SQL Server 2008 дозволяється також зберігати XML-дані в змінних всіх вищеперерахованих типів.

Наведемо кілька прикладів на створення джерел збереження XML-даних та їх заповнення. Припустимо, що необхідно для кожного магазину в XML форматі зберігати каталоги книг видавництва, які знаходяться у них на реалізації:

```
-- створення таблиці з полем типу XML та заповнення її даними
create table BooksCatalog(
    ID_CATALOG int identity not null,
    ID_SHOP int not null constraint fkShops references sale.Shops (ID_SHOP),
    BCatalog xml
);

INSERT INTO BooksCatalog (ID_SHOP, BCatalog)
VALUES (1, '<catalog><book>
            <name>Основы работы на ПК</name>
            <author>Андрей Ковязин</author>
            <dateOfPublish>11-01-2000</dateOfPublish>
            <price>56.8</price>
        </book></catalog>');

-- створення змінної типу XML та вставка даних
declare @vcatalog as xml;
set @vcatalog = '<catalog><book>
                <name>Основы работы на ПК</name>
                <author>Андрей Ковязин</author>
                <dateOfPublish>11-01-2000</dateOfPublish>
                <price>56.8</price>
            </book></catalog>';
```

SQL Server дозволяє перевіряти XML-документи на коректність. Для цього використовується колекція XML-схем, яка являє собою звичайний набір документів схеми, які зберігаються в базі даних під одним іменем. Всі XML-схеми



оголошуються на рівні бази даних і розгортаються на SQL Server. Після їх створення можна типізувати та перевіряти на коректність довільне поле таблиці, вміст змінної або параметра типу XML у відповідності до колекції XML-схем. Доречі, SQL Server підтримує як **типізовані (typed)**, так і **нетипізовані (untyped) XML-дані**.

Для створення XML-схеми використовується оператор **CREATE XML SCHEMA COLLECTION**, синтаксис якого наступний:

```
CREATE XML SCHEMA COLLECTION [ схема. ] ім'я_ XML_схеми
AS
{ 'рядкова_константа' | @змінна }
```

Рядкова константа або скалярна змінна повинні мати тип varchar, varbinary, nvarchar або xml.

Наприклад, створимо колекцію XML схем для розробленого нами каталогу 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:complexType>
</xs:element>
</xs:schema>';
```

Але в більшості випадків XML схему створюють окремо в файлі з розширенням .xsd, після чого завантажують її в змінну типу XML за допомогою команди **OPENROWSET**.

```
-- синтаксис оператора OPENROWSET
OPENROWSET
( 'постачальник_даних_OLEDB', { 'ім'я_ВД'; 'логін'; 'пароль' | 'рядок_доступу' },
  { [ каталог. ] [ схема. ] назва_об'єкта | 'запит' }
| BULK 'шлях_до_файла_даних',
  { FORMATFILE = 'шлях_до_файла_форматування' [ bulk-опції ]
| SINGLE_BLOB      -- вміст файла даних повертається у вигляді набору записів.
                    -- Рекомендується (!)
| SINGLE_CLOB      -- зчитує файл даних як ASCII-файл
| SINGLE_NCLOB     -- зчитує файл даних як Unicode
}
)

-- bulk-опції:
-- кодова сторінка даних в файлі даних
[, CODEPAGE = { 'ACP' | 'OEM' | 'RAW' | 'кодова_сторінка' } ]

-- файл, який використовується для збору записів з помилками форматування
[, ERRORFILE = 'назва_файла' ]

[, FIRSTROW = номер ]      -- номер першого запису для завантаження
[, LASTROW = номер ]       -- номер останнього запису для завантаження
[, MAXERRORS = число ]     -- максимальна кількість помилок в файлі форматування
[, ROWS_PER_BATCH = кількість_записів ] -- приблизна кількість записів в файлі даних
[, ORDER ( { поле [ ASC | DESC ] } [ ,...n ] ) [ UNIQUE ]
```



Наприклад:

```
declare @schema XML

select @schema = c
from OPENROWSET (BULK 'MySchema.xsd', SINGLE_BLOB) as tmp(c)

create xml schema collection MySchema as @schema
```

Після того, як XML схема додається в базу даних (імпортується чи створюється локально), вона може лексично відрізнитись від свого первинного вигляду. Це пов'язано з тим, що для підвищення ефективності збереження схеми SQL Server видаляє з неї ряд компонентів різних типів. Наприклад, видаляються коментарі, пропуски та інші компоненти, префікси просторів імен також не зберігаються, а дані неявних типів приводяться до явних. Наприклад, `<xs:element name="author" />` конвертується в `<xs:element name="author" type="xs:anyType" />`. В зв'язку з цим рекомендується зберігати копію кожної схеми.

Для перегляду існуючої колекції XML схем використовується вбудована функція `xml_schema_namespace()`. Функція повертає дані типу xml.

```
xml_schema_namespace ( 'схема_БД', 'ім'я_колекції_XML_схем', [ 'простір_імен' ] )
```

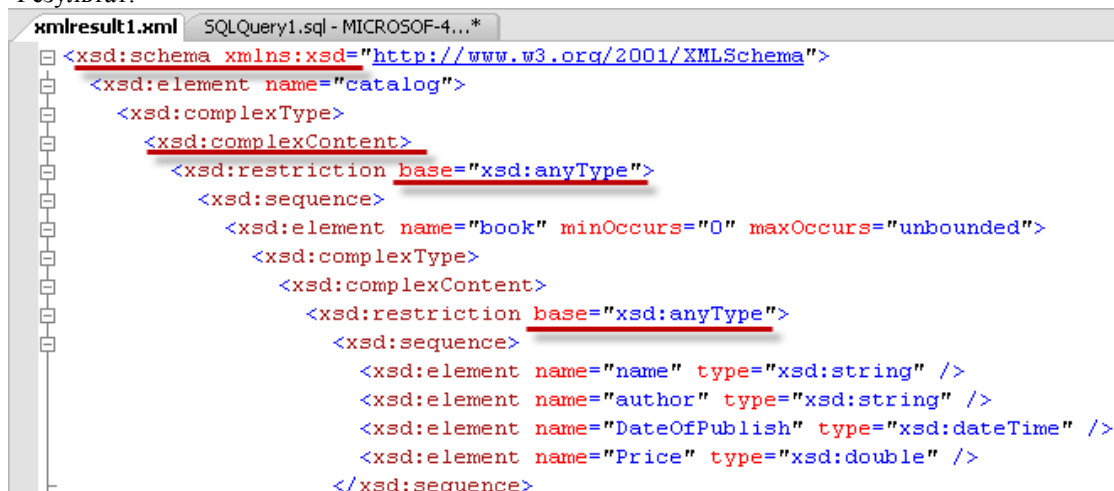
В першому параметрі вказується назва реляційної схеми, в якій розміщується колекція XML схем. В другому - її ім'я, яке має тип **sysname**. Необов'язковий параметр «**простір імен**» використовується для позначення конкретного URI простору імен, яке відноситься до XML-схеми. Фактично, даний параметр дозволяє обрати з колекції XML схему, яка відноситься до вказаного простору імен. Якщо ж URI простору імен відсутнє, тоді перебудовується вся колекція XML-схем.

Аргумент простору імен має тип `nvarchar(4000)`, тобто його максимальна довжина рівна 1000 символів.

Отже, скористаємось функцією `xml_schema_namespace()` для отримання інформації про колекцію XML схем:

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

Результат:



```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="catalog">
    <xsd:complexType>
      <xsd:complexContent>
        <xsd:restriction base="xsd:anyType">
          <xsd:sequence>
            <xsd:element name="book" minOccurs="0" maxOccurs="unbounded">
              <xsd:complexType>
                <xsd:complexContent>
                  <xsd:restriction base="xsd:anyType">
                    <xsd:sequence>
                      <xsd:element name="name" type="xsd:string" />
                      <xsd:element name="author" type="xsd:string" />
                      <xsd:element name="DateOfPublish" type="xsd:dateTime" />
                      <xsd:element name="Price" type="xsd:double" />
                    </xsd:sequence>
                  </xsd:restriction>
                </xsd:complexContent>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:restriction>
      </xsd:complexContent>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Як видно з результату, після вставки XML-схеми змінила свій первинний вигляд, як і відмічалось вище.

Якщо XML-дані являються типізованими, тоді після створення колекції XML-схем, її необхідно з ними зв'язати. Це можна зробити одним з наступних **способів**:

- при створенні таблиці:

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

- якщо таблиця вже створена:

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

- при оголошенні змінної типу XML:

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



Для маніпулювання створеними XML-схемами використовуються оператори **ALTER / DROP XML SCHEMA COLLECTION**.

Інструкція **ALTER XML SCHEMA COLLECTION** дозволяє лише додавати нові XML-схеми з просторами імен до колекції або нові компоненти до існуючих просторів імен. Синтаксис даного оператора наступний:

```
ALTER XML SCHEMA COLLECTION [ схема. ] ім'я_XML_схеми
ADD
'компонент_схеми_для_вставки'
```

Наприклад, необхідно додати новий елемент <Annotation> до існуючого простору імен в колекції 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:

```
<xs: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">
```

Відмітимо, що у випадку додавання нових компонентів до колекції, які ссилаються на вже існуючі, необхідно скористатись командою <import namespace="referenced_component_namespace"/>.

Для видалення існуючої колекції схем, використовується оператор **DROP XML SCHEMA COLLECTION**.

```
DROP XML SCHEMA COLLECTION [ схема. ] ім'я_XML_схеми
```

3.2. Вибірка та зміна XML-даних

В SQL Server 2008 реалізовано багато засобів отримання XML-даних, незалежно від того, в якому вигляді вони зберігаються: в полях типу XML чи в текстових полях. Розглянемо методи отримання XML-даних для обох варіантів збереження.

Отже, якщо XML-дані зберігаються в полях, параметрах або змінних типу XML, тоді для роботи з фрагментами XML використовують **5 методів**, які для пошуку використовуються вирази XPath або XQuery:

Метод	Опис
query('запит')	Повертає шуканий нетипізований (який не відповідає XML-схемі) фрагмент даних XML. Прирівнюється до виконання звичайних SQL-запитів.
value('вираз', 'тип_результату')	Повертає скалярне типізоване значення, яке потім перетворюється в тип T-SQL. В зв'язку з тим, що метод повертає скалярне значення, вираз XPath або XQuery потрібно писати так, щоб результатом було єдине значення. Наприклад, в XPath існує функція count(), яка повертає кількість виділених елементів, або position(), що вказує позицію поточного вузла.
exist('вираз')	Здійснює перевірку вузлів на існування. Даний метод повертає true, якщо запит знайде хоча б один вузол. Зазвичай метод exist() використовується в додатку WHERE оператора SELECT для перевірки наявності певного вузла.
modify('вираз XML DML')	Служить для маніпулювання (вставки, видалення або оновлення) XML-даними. При цьому він дозволяє змінювати як значення в XML-вузлах, так і структуру фрагмента XML.
nodes('вираз')	Повертає фрагмент XML, розбитий на набір записів, тобто у вигляді реляційної таблиці.

Приведемо кілька прикладів по роботі з XML-даними за допомогою вищерозглянутих методів. В якості джерела даних візьмемо створену в попередньому підрозділі таблицю BooksCatalog, яка містить каталоги книг для магазинів. Для початку спробуємо отримати XML-дані про магазини, які реалізують книги Джонсона Вайта.

```
select bc.ID_SHOP as 'Код',
       sh.NameShop as 'Магазин',
```



```
bc.BCatalog.query('/catalog/book[author = "Johnson White"]') as 'Каталог'
from BooksCatalog bc, sale.Shops sh
where bc.ID_SHOP=sh.ID_SHOP
and bc.BCatalog.exist('/catalog/book[author = "Johnson White"]') = 1
```

Результат:

Results			
	Код	Магазин	Каталог
1	1	Букинист	<book><name>Толковый словарь компьютерных технол...
2	4	Booksworld	<book><name>Толковый словарь компьютерных технол...
3	5	All about PC	<book><name>Толковый словарь компьютерных технол...

Для отримання інформації про книгу, яка перша представлена в каталозі магазину «Слово» слід написати наступний запит:

```
select bc.BCatalog.value(' (/catalog/book) [1]', 'nvarchar(50)')
from BooksCatalog bc, sale.Shops sh
where bc.ID_SHOP=sh.ID_SHOP and sh.NameShop='Слово'
```

Метод **modify()** складніший попередніх методів і потребує короткого роз'яснення. В якості параметра метод приймає вираз XML DML, який виконується над фрагментом XML.

В мові XML DML використовуються наступні регістрозалежні **ключові слова**:

- **insert** – додає один або кілька вузлів. Для більш точної вставки вузлів використовуються оператори:
 - after – вставити після вузла, вказаного в якості другого параметра;
 - before – вставити перед вузлом, вказаного в якості другого параметра;
 - into – вузли будуть додані як дочірні по відношенню до вузла, вказаного в якості другого параметра. Якщо вузол вже має дочірні елементи, тоді можна вказати їх точніше місце розташування: на початок (as first into) чи в кінець (as last into).
- **delete** – видаляє один або кілька вузлів.
- **replace value of** – оновлює значення заданого вузла і використовується в операторі UPDATE.

Варто відмітити, що метод **modify()** викликається за допомогою інструкції SET, яка може використовуватись окремо або в складі оператора UPDATE.

Припустимо, в магазин «Букиніст» відправлені на реалізацію ряд нових книг. Отже, каталог даного магазину слід підправити. Спочатку необхідно додати інформацію про нові книги в XML-документ, а потім додати самі дані. Для цього слід скористатись методом **modify()**, а у виразах XML DML буде використане ключове слово insert з необхідними опціями:

```
-- додаємо інформацію про нову книгу в каталог
update BooksCatalog
set BCatalog.modify('
    insert <book><name>Невероятные приключения в Лесной школе. Книга 1</name>
        <author>Всеволод Нестайко</author>
        <DateOfPublish>2010-07-05T00:00:00</DateOfPublish>
        <Price>36.40</Price>
    </book>
    before (/catalog/book) [1]')
where ID_SHOP IN
    ( select sh.ID_SHOP
      from sale.Shops sh
      where NameShop='Букинист')

-- додаємо атрибут discount (знижка), якщо ціна книги більше 200 грн.
update BooksCatalog
set BCatalog.modify('insert attribute discount {"true"}
    into (/catalog/book[Price>200]) [1]')

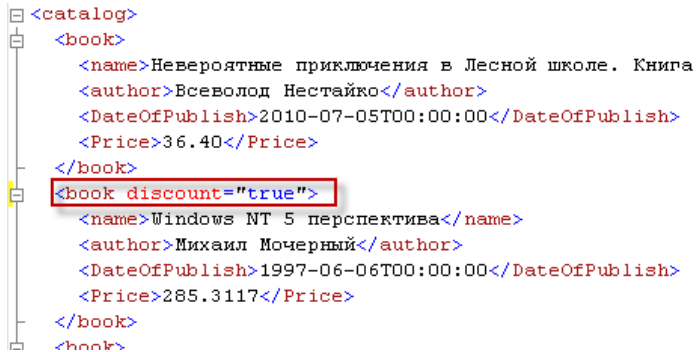
-- відмінити знижку на книги в магазині "Слово"
update BooksCatalog
set BCatalog.modify('replace value of (/catalog/book/@discount) [1]
    with "false"')
where ID_SHOP IN ( select sh.ID_SHOP
    from sale.Shops sh
    where NameShop='Слово')
```



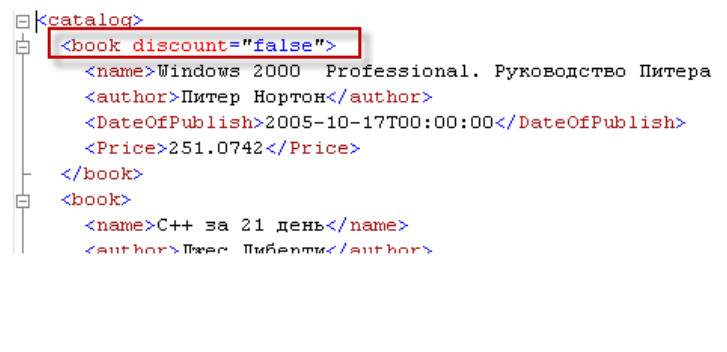
```
-- видалити всі книги в магазині "Слово", ціна яких перевищує 500 грн.
update BooksCatalog
set BCatalog.modify('delete /catalog/book[price>500]')
where ID_SHOP IN
    ( select sh.ID_SHOP
      from sale.Shops sh
      where NameShop='Слово' )
```

Результат:

магазин "Букініст"



магазин "Слово"



Останній метод **nodes()** використовується для відображення даних в табличному вигляді. Для кожного вузла вхідного XML-документа, який відповідає виразу, формується окремий запис. Результуюча таблиця містить єдине поле типу XML. В зв'язку з тим, що воно лише одне, для отримання даних кожного запису, як правило, використовують методи `value()`, `query()` або `exist()`. Загальний синтаксис використання даного метода наступний:

```
nodes ('вираз') as результуюча_таблиця (поле)
```

Наприклад:

```
-- створюємо та заповнюємо змінну
declare @vcatalog as xml;
set @vcatalog = '
<catalog>
  <book discount="false">
    <name>Windows 2000 Professional. Руководство Питера Нортон</name>
    <author>Питер Нортон</author>
    <DateOfPublish>2005</DateOfPublish>
    <Price>251.0742</Price>
  </book>
  <book>
    <name>C++ за 21 день</name>
    <author>Джес Либерти</author>
    <DateOfPublish>2006</DateOfPublish>
    <Price>148.3621</Price>
  </book>
</catalog>';

-- виводимо результат в табличному вигляді за допомогою метода nodes()
select c.value('name[1]', 'nvarchar(max)') as 'Назва книги',
       c.query('.') as 'Повний опис'
from @vcatalog.nodes('/catalog/book') as tmp(c)
```

Результат:

	Results	Messages
	Назва книги	Повний опис
1	Windows 2000 Professional. Руководство Питера Н...	<book discount="false"><name>Windows 2000 Profession...
2	C++ за 21 день	<book><name>C++ за 21 день</name><author>Джес Ли...



На жаль, якщо необхідно вивести дані, що зберігаються в полі типу XML деякої таблиці, такий синтаксис використати неможна. Причина полягає в тому, що метод `nodes()` повертає результуючий набір, а оператор `SELECT` вимагає повернення єдиного значення. Щоб випрати дану ситуацію, слід скористатись оператором **APPLY**, який дозволяє викликати метод для кожного запису результуючої множини.

Цей оператор має **дві форми**:

- **CROSS APPLY** – повертає лише не NULL-значення;
- **OUTER APPLY** – відображає всі записи, навіть якщо вони рівні NULL.

Узальнений синтаксис використання оператора `APPLY` з методом `nodes()` має наступний вигляд:

```
SELECT список_полів
FROM список_таблиць
{ CROSS | OUTER } APPLY поле.nodes('вираз') as результуюча_таблиця(поле)
```

Наприклад:

```
select bc.ID_CATALOG,
       tmp.c.value('name[1]', 'nvarchar(max)') as 'Назва книги',
       tmp.c.query('.') as 'Повний опис'
from BooksCatalog bc
CROSS APPLY bc.BCatalog.nodes('/catalog/book') as tmp(c)
```

Результат:

Results		Messages	
ID_CATALOG	Назва книги	Повний опис	
3	1	Основы работы на ПК	<book><name>Основы работы на ПК</name><author>Ан...
4	1	Windows NT 5 перспектива	<book discount="true"><name>Windows NT 5 перспектива...
5	1	Толковый словарь компьютерных технологий	<book><name>Толковый словарь компьютерных технол...
6	2	Windows 2000 Professional. Руководство Питера Н...	<book discount="false"><name>Windows 2000 Professional...
7	2	C++ за 21 день	<book><name>C++ за 21 день</name><author>Джес Либ...
8	3	Windows 2000 Professional. Руководство Питера Н...	<book discount="true"><name>Windows 2000 Professional...
9	3	JavaScript: справочник рецептов для профессионалов	<book><name>JavaScript: справочник рецептов для професи...

3.3. Вибірка реляційних даних в форматі XML. Конструкція FOR XML

Для того, щоб отримати результати `SELECT` запиту у вигляді XML, тобто для конвертування даних текстових полів, параметрів або змінних в XML-структуру використовується додаток **FOR XML** того ж оператора `SELECT`. Інструкція `FOR XML` може використовуватись як у запитах верхнього рівня (тільки в операторі `SELECT`), так і у підзапитах (в операторах `INSERT`, `DELETE`, `UPDATE`).

```
SELECT список_вибірки [ INTO ім'я_нової_таблиці ]
[ FROM список_таблиць ]
[ WHERE умова ]
[ GROUP BY вираз_групування ]
[ HAVING умова_на_групі ]
[ ORDER BY напрям_сортування ]
[ COMPUTE { { AVG | COUNT | MAX | MIN | SUM } (вираз) } [ BY вираз ] ]

[ FOR XML
{
  { RAW [ ( 'назва_елемента' ) ] | AUTO }
  [ загальні_характеристики
    [, { XMLDATA | XMLSCHEMA [ ( 'URI_простору_імен' ) ] } ]
    [, ELEMENTS [ XSINIL | ABSENT ] ]
  ]
| EXPLICIT
  [ загальні_характеристики [, XMLDATA ] ]
| PATH [ ( 'назва_елемента' ) ]
  [ загальні_характеристики [, ELEMENTS [ XSINIL | ABSENT ] ] ]
}]

-- загальні_характеристики
[, BINARY BASE64 ] -- повертає двійкові дані в зашифрованому двійковому форматі base64
[, TYPE ]          -- повернути дані в вигляді типу даних XML
[, ROOT [ ( 'кореневий_елемент' ) ] ] -- створити кореневий елемент
```




Як видно з синтаксису використання, додаток FOR XML перетворює результуючий набір запити в XML-структуру і підтримує наступні **4 режими форматування**:

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

Розглянемо коротко кожен з цих режимів.

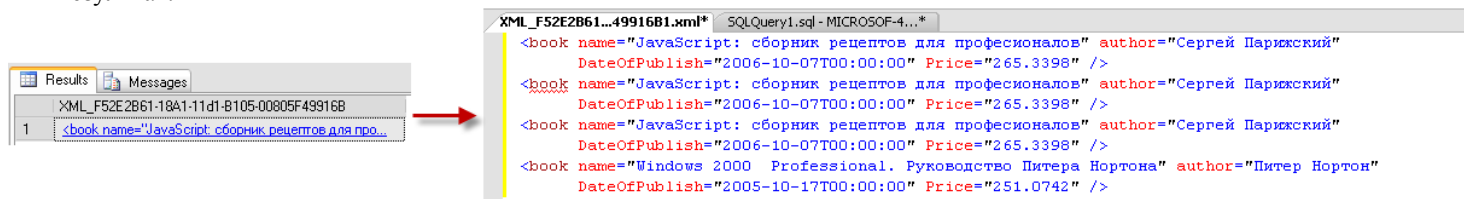
Режим **RAW** являється самим простим і обмеженим способом перетворення даних в формат XML. Він перетворює кожен запис результуючого набору в XML елемент з назвою <row />. Елементу <row /> можна задати власне ім'я, вказавши його в дужках після ключового слова RAW.

Всі поля в результуючому XML фрагменті будуть представлені у вигляді атрибутів елемента <row /> з відповідними назвами полів. Щоб переіменувати атрибути, слід в списку вибірки SELECT задати кожному полю необхідний псевдонім.

Наприклад,

```
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:

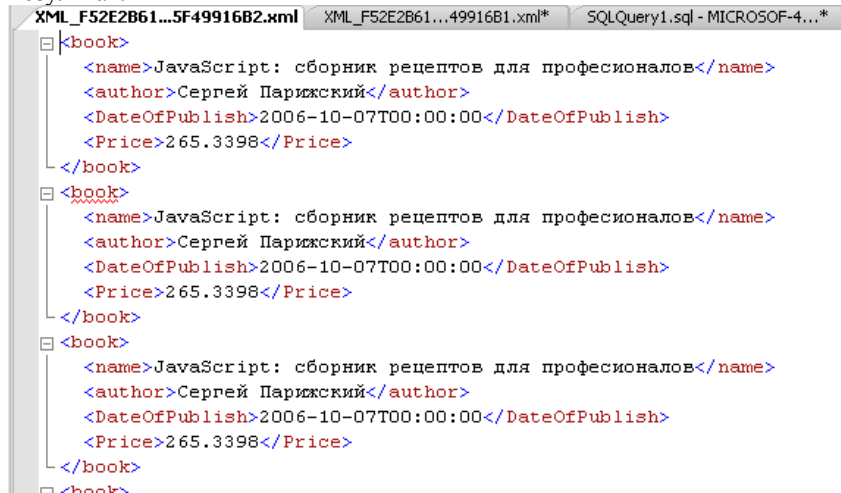


ПРИМІТКА! Для коректного відображення результату роботи запити слід виставити режим відображення даних «Results to grid».

Для представлення даних полів у вигляді елементів необхідно вказати ключове слово **ELEMENTS** після додатку FOR XML RAW. Даний параметр підтримується також в режимах AUTO і 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:





Додаткові опції **XSINIL** та **ABSENT** параметра **ELEMENTS** використовуються для маніпулювання відображенням даних полів з NULL-значеннями:

- **XSINIL** – створює елемент з атрибутом `xsi:nil` , який містить значення `true` для полів з NULL-значеннями;
- **ABSENT** – поля, які містять NULL-значення до результуючого фрагменту XML не додаються.

Можна також побудувати XML-схему для поточного XML-документа за допомогою однієї з наступних **опцій**:

- XMLDATA** повертає вбудовану XDR-схему, не додаючи корневий елемент до результату. Директива **XMLDATA** для параметра **XML FOR** являється застарілою і в наступній версії MS SQL Server планується її вилучення. В зв'язку з цим варто уникати її використання.
- XMLSCHEMA** повертає вбудовану XSD-схему.

Результатом роботи попереднього запиту є фрагмент XML, оскільки в ньому відсутній корневий елемент. Для того, щоб отримати на виході XML-документ, необхідно додати інструкцію **ROOT** після додатку **FOR XML RAW**. Корневим елементом по замовчуванню буде елемент з іменем `<root>`, але ви можете задати власне ім'я.

Отже, розширимо наш приклад:

```
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'), root('catalog'), elements, xmlschema('BooksCatalogSchema')
```

Частковий результат:

The screenshot shows the XML output of the query. The root element is `<catalog>`. Inside `<catalog>`, there is an `<xsd:schema targetNamespace="BooksCatalogSchema" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:sqltypes="http://schemas.microsoft.com/sqlserver/2004/sqltypes" sc="1">` element. This schema element contains an `<xsd:element name="book">` element. The `<book>` element is a complex type containing a sequence of four elements: `<xsd:element name="name">`, `<xsd:element name="author">`, `<xsd:element name="DateOfPublish" type="sqltypes:datetime" minOccurs="0">`, and `<xsd:element name="Price" type="sqltypes:money" minOccurs="0" />`. The `<book>` element is also marked with `xmlns="BooksCatalogSchema"`. The XML output shows two instances of the `<book>` element, each with the same data: `<name>JavaScript: сборник рецептов для профессионалов</name>`, `<author>Сергей Парижский</author>`, `<DateOfPublish>2006-10-07T00:00:00</DateOfPublish>`, and `<Price>265.3398</Price>`.

Йдемо далі. По замовчуванню інструкція **FOR XML** повертає XML як текст, тому результат її роботи може бути присвоєний як рядковій змінній, так і змінній типу XML. В останньому випадку здійснюється неявне приведення до типу. Але додаток **FOR XML** підтримує також явне конвертування за допомогою інструкції **TYPE**.

В зв'язку з тим, що інструкція **TYPE** повертає дані типу XML, до вихідного набору даних можна застосувати всі методи для роботи з XML. Це значно розширює можливості по обробці даних.

В режимі **AUTO** результати запиту повертаються у вигляді простого вкладеного дерева XML. Для кожної таблиці, яка вказана в запиті **SELECT**, створюється новий рівень в XML-структурі. Список **SELECT** задає порядок вкладеності XML-даних. Але, якщо поля в запиті змішані, тоді XML-вузли перевпорядковуються таким чином, щоб всі вузли, які відносяться до одного рівня, були згруповані під одним і тим же батьківським елементом.

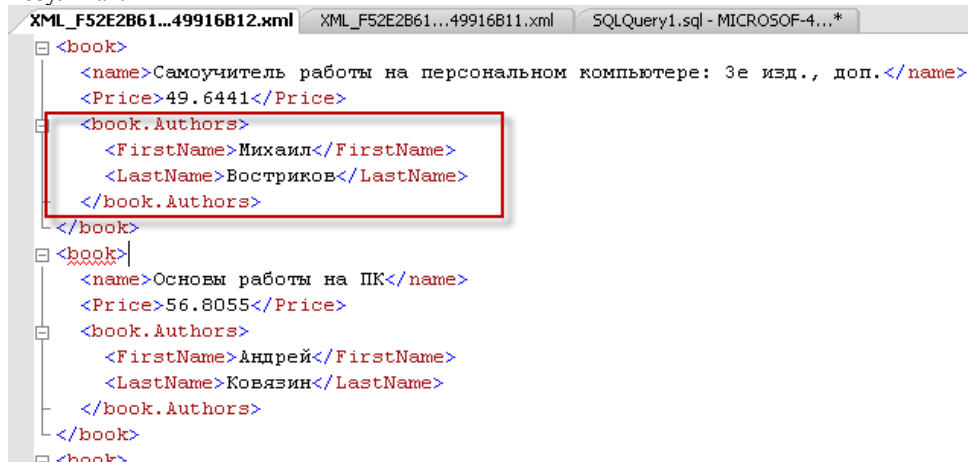
Наприклад,

```
select book.NameBook as 'name',
       book.Authors.FirstName, book.Authors.LastName,
       book.Price
```



```
from book.Books as book, book.Authors
where book.ID_AUTHOR=book.Authors.ID_AUTHOR
for xml auto, elements
```

Результат:



```
<book>
  <name>Самоучитель работы на персональном компьютере: 3е изд., доп.</name>
  <Price>49.6441</Price>
  <book.Authors>
    <FirstName>Михаил</FirstName>
    <LastName>Востриков</LastName>
  </book.Authors>
</book>
<book>
  <name>Основы работы на ПК</name>
  <Price>56.8055</Price>
  <book.Authors>
    <FirstName>Андрей</FirstName>
    <LastName>Ковязин</LastName>
  </book.Authors>
</book>
</book>
```

Режим **EXPLICIT** дозволяє більш гнучко визначити вихідну XML-структуру. Кожне поле налаштовується окремо, тобто дозволяється змішане використання елементів та атрибутів. При цьому результат запиту порівнюється з шаблоном, так званою **універсальною таблицею**. Універсальна таблиця повинна містити кілька **обов'язкових полів**:

- **Tag** – перше поле результуючої множини, яке вказує глибину XML-структури, починаючи з 1;
- **Parent** – друге поле, яке вказує на батьківський вузол.

Псевдоніми полів формуються по наступному шаблону:

```
[директива] ім'я_XML_елемента!рівень_вкладеності!ім'я_атрибута_XML
```

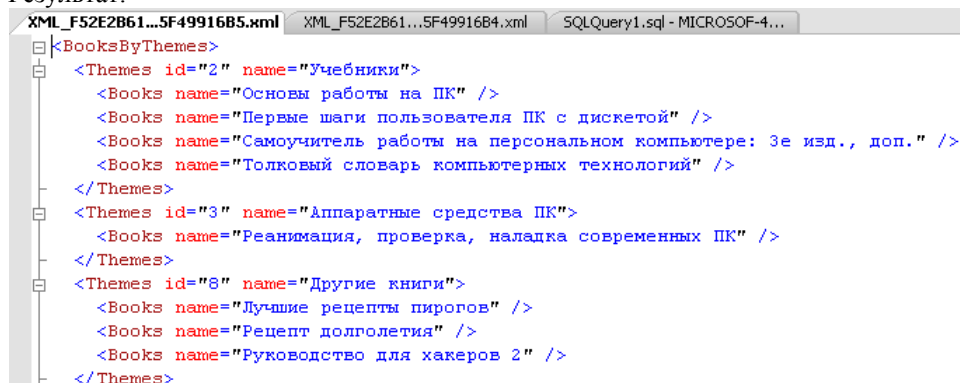
Директива являється необов'язковим параметром та надає додаткову інформацію для форматування XML. Доречі, **псевдоніми таблиць в режимі EXPLICIT ігноруються (!)**.

Для об'єднання кількох запитів в одну повну універсальну таблицю можна використати оператор UNION або UNION ALL. В такому випадку кожен запит представляє окремий рівень ієрархії.

Наприклад, сформуємо XML-документ, який містить список книг, в розрізі 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:



```
<BooksByThemes>
  <Themes id="2" name="Учебники">
    <Books name="Основы работы на ПК" />
    <Books name="Первые шаги пользователя ПК с дискетой" />
    <Books name="Самоучитель работы на персональном компьютере: 3е изд., доп." />
    <Books name="Толковый словарь компьютерных технологий" />
  </Themes>
  <Themes id="3" name="Аппаратные средства ПК">
    <Books name="Реанимация, проверка, наладка современных ПК" />
  </Themes>
  <Themes id="8" name="Другие книги">
    <Books name="Лучшие рецепты пирогов" />
    <Books name="Рецепт долголетия" />
    <Books name="Руководство для хакеров 2" />
  </Themes>
</BooksByThemes>
```



Без використання сортування, результат буде наступним:

```
XML_F52E2B61...49916B6.xml* SQLQuery1.sql - MICROSOFT-4...*
<BooksByThemes>
  <Themes id="2" name="Учебники" />
  <Themes id="3" name="Аппаратные средства ПК" />
  .....
  <Books name="Самоучитель работы на персональном компьютере: 3е изд., доп." />
  <Books name="Основы работы на ПК" />
  <Books name="Толковый словарь компьютерных технологий" />
  <Books name="Первые шаги пользователя ПК с дисками" />
```

Як видно з прикладу, написання запитів в режимі EXPLICIT доволі складне. В якості альтернати для досягнення аналогічного результату (створення XML-ієрархій) SQL Server пропонує використовувати вкладені запити FOR XML в режимі RAW, AUTO, PATH. Наприклад, побудуємо XML-документ з даними про авторів в розрізі країн їх проживання:

```
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:

```
XML_F52E2B61...49916B13.xml SQLQuery1.sql - MICROSOFT-4...*
<ListAuthorsByCountry>
  <country>
    <NameCountry>Україна</NameCountry>
    <authors>
      <author fname="Тарас" lname="Тимошок" />
      <author fname="Михаил" lname="Востриков" />
      <author fname="Artur" lname="Liliput" />
    </authors>
  </country>
  <country>
    <NameCountry>Росія</NameCountry>
    <authors>
      <author fname="Сергей" lname="Парижский" />
      <author fname="Сергей" lname="Михайлов" />
      <author fname="Андрей" lname="Ковязин" />
      <author fname="Михаил" lname="Мочерный" />
      <author fname="Олег" lname="Лисовский" />
      <author fname="Jon" lname="Green-White" />
      <author fname="Marta" lname="Greenes" />
    </authors>
  </country>
</country>
```

Режим **PATH** являється спрощеною формою режиму **EXPLICIT**. В цьому режимі за допомогою звичайних XPath-виразів полям можна задавати псевдоніми, які визначають місцезорозташування їх даних у вихідному фрагменті XML. По замовчуванню всі поля додаються в елемент <row />, як і в режимі 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:

```
xmlresult2.xml  xmlresult1.xml  SQLQuery1.sql - MICROSOFT-4...*
<shops>
  <shop nameShop="All about PC">
    <!--Великобританія-->
    <book name="Толковий словарь комп'ютерних технологій">
      <price>82.7405</price>
    </book>
  </shop>
  <shop nameShop="Book">
    <!--США-->
    <book name="Как программировать на C">
      <price>432.2473</price>
    </book>
  </shop>
  <shop nameShop="Booksworld">
    <!--США-->
    <book name="Толковий словарь комп'ютерних технологій">
      <price>82.7405</price>
    </book>
  </shop>
</shops>
```

Як видно з результату, оголошена XML-структура повторюється для кожного запису. Для створення складних вкладень XML-структур, використовуються вкладені запити FOR XML.

Доречі, якщо вказати пустий запис після назви режиму (наприклад, FOR XML PATH ("")), тоді упаковщик елементів не створюється.

В ході своєї розповіді, ми не зачіпали ще один механізм роботи з XML-даними в SQL Server 2008 – технологію для маніпулювання XML-даними **SQLXML**. Поточна версія якої 4.0. SQLXML являється API-інтерфейсом середнього рівня, побудованого на базі COM, яка дозволяє працювати з реляційними даними без використання інструкцій T-SQL. Більш детально про дану технологію можна прочитати в розділі [«Основні поняття про програмування для SQLXML 4.0»](#) електронної документації по SQL Server 2008.

4. Домашнє завдання

Написати наступні користувацькі функції:

1. Функцію, яка повертає кількість магазинів, які не продали жодної книги видавництва.
2. Функцію, яка повертає мінімальне з трьох параметрів.
3. Багатооператорну функцію, яка повертає кількість проданих книг по кожній з тематик і в розрізі кожного магазину.
4. Функцію, яка повертає список книг, які відповідають набору критеріїв (ім'я та прізвище автора, тематика), і відсортовані по прізвищу автора у вказаному в 4-му параметрі напрямку.
5. Функцію, яка повертає середнє арифметичне цін всіх книг, проданих до вказаної дати.
6. Функцію, яка повертає найдорожчу книгу видавництва вказаної тематики.
7. Функцію, яка по ID магазину повертає інформацію про нього (ID, назву, місце розташування, середню вартість продаж за останній рік книг вашого видавництва) в табличному вигляді.

За допомогою курсорів:

1. Створіть послідовний курсор, який містить дані про те, скільки кожен український магазин продав книг за минулий рік. Виведіть дані курсора орієнтовно в наступному вигляді:

```
Messages
Магазин Слово продав 100 книг (-и)
Магазин Букініст продав 54 книг (-и)
Магазин Іскра продав 158 книг (-и)
Магазин Booksworld продав 5 книг (-и)
```

2. Створіть локальний ключовий курсор, який містить список тематик та інформацію про те, скільки авторів пишуть в кожному окремому жанрі. Збережіть значення полів в змінних.. За допомогою курсора і зв'язаних змінних здійсніть наступні дії:
 - виведіть в циклі всю множину даних курсора;
 - виведіть окремо останній запис;
 - виведіть окремо 5-й запис з кінця;
 - виведіть окремо 3-й запис з початку.



3. За допомогою локального ключового курсора здійснити позиціоноване оновлення даних в таблиці «Themes», а саме: змінити назву тематики, яка стоїть на 3-й позиції від початку.
4. За допомогою глобального курсора здійснити позиціоноване видалення 2-го з кінця запису. В курсорі міститься інформація про авторів, книги яких за два останні роки ще жодного разу не продались.

Використати технологію XML для виконання наступних задач:

1. Створити змінну типу XML, яка містить в ієрархічному вигляді дані про тематики та авторів, книги яких в них представлені. Дані повинні бути відсортовані по авторам, книги яких були найперші видані видавництвом. Від найстаріших авторів до нових. Вивести на екран дані змінної.
2. Відфільтрувати та вивести за допомогою метода query() та XPath значення змінної з завдання (1), щоб отримати список авторів лише певної тематики, наприклад, лише тематики «HTML/XHTML/XML».
3. За допомогою метода modify() та XPath видалити всі тематики з змінної завдання (1), в яких не представлено жодного автора.
4. Необхідно розробити таблицю Order, яка буде містити звіти про продаж книг магазинами в наступному вигляді:
 - 1) Магазин, який представляє звіт (для забезпечення цілісності даних зв'язується з таблицею Shops);
 - 2) Дата формування звіту (по замовчуванню приймається поточна дата);
 - 3) Звіт у вигляді XML-документа.

Форма звіту повинна відповідати певній XML-схемі, яка містить коротку анотацію. Орієнтовна структура звіту повинна бути наступна:

```

<order>
  <!-- 28.08.2010 -->
  <shop name="Слово" country="Украина" />
  <books>
    <book id="5">
      <name>Колобок идет по следу</name>
      <author>Эдуард Успенский</author>
      <theme>Фантастика</theme>
      <dateOfSale>15.05.2010</dateOfSale>
      <totalCost um="{UAH | EUR | USD | RUB}">80</totalCost>
    </book>
    <book id="10" writeDown="{true | false}">
      <name>Тайный сыск царя Гороха</name>
      <author>Андрей Белянин</author>
      <theme>Фэнтези</theme>
      <dateOfSale>01.08.2010</dateOfSale>
      <totalCost um="UAH">100</totalCost>
    </book>
  </books>
  <Results>
    <totalQuantity>2</totalQuantity>
    <totalCost um="UAH">180</totalCost>
  </Results>
</order>

```

Дата формування звіту

Грошова одиниця

Уцінений товар?

Загальна кількість продажу

Загальна вартість продажу

5. По максимуму спробувати автоматизувати процес внесення даних в таблицю.
6. Вибрати всі записи з таблиці Order з використанням метода query() та XPath. Орієнтовний вигляд представлений в завданні (4).
7. Відфільтрувати результати таблиці Order з використанням метода query() та XPath, щоб отримати лише списки уцінених проданих книг на протязі останнього місяця.
8. Вибрати всі записи з таблиці Order з використанням метода query() та XPath, але включити при цьому в XML-структуру дані, які зберігаються в реляційній інфраструктурі, наприклад, поле ID магазину.
9. Вибрати незалежні значення з таблиці Order за допомогою метода value() та XPath. Поверніть табличну структуру, яка містить інформацію про всі продані книги, ціна яких більше 200 грош.од. тематики «WEB-програмування» та «Java, J++, JBuilder, JavaScript».
10. За допомогою метода modify() та XPath додати до звіту магазину, наприклад, «Букініст» ще одну книгу. Після вставки даних, додайте їй відмітку про уцінений товар. Здійснить необхідні зміни в елементі «Results».

ПРИМІТКА! За кожен блок задач виставляється окрема оцінка.