Урок 7

План заняття:

1. Індекси в SQL Server
2. Індексовані представлення
3. Повнотекстове індексування та пошук
4. Тригери
5. Домашнє завдання

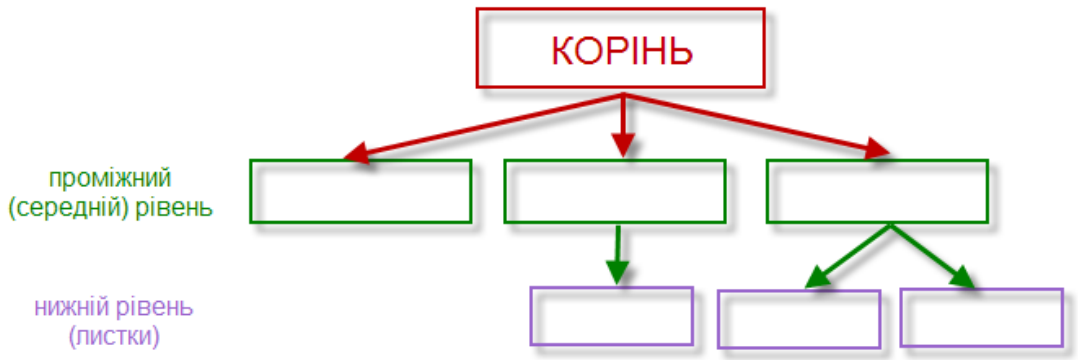
1. Індекси в SQL Server

Для підвищення швидкості виконання запитів та доступу до даних бази даних рекомендується використовувати індекси. Без індексів СУБД читає дані, переглядаючи кожну сторінку таблиць, вказаних в операторі SQL, проводить так зване **сканування таблиці**.

Поняття індекс зустрічається не лише в теорії баз даних і скоріше за все ви з ним вже стикались у повсякденному житті. Індекси баз даних можна порівняти з бібліотечним каталогом, в якому всі книги записані в карточки і впорядковані певним чином: по алфавіту чи по темам, а в кожній карточці написано, де саме в архівах розміщується дана книга. Іншим прикладом може слугувати предметний вказівник (індекс) в кінці книги, який допомагає читачу легко знайти потрібне місце. Варто нагадати, що в нього входить лише невелика кількість термінів, які відсортовані по алфавіту, що дозволяє швидко знайти необхідну інформацію. Схожим чином організовані і індекси в MS SQL Server.

Сам **індекс** являє собою впорядкований логічний вказівник на записи в таблиці. **Вказівник** означає, що індекс містить значення одного або кількох полів в таблиці і адреси сторінок даних, на яких розміщуються ці значення. Іншими словами, індекс складається з пар «значення поля – фізичне розміщення цього поля». Таким чином, по значенню поля (чи полів), що входять в індекс, можна швидко знайти те місце в таблиці, де розміщений запис з шуканим значенням. **Впорядкований** означає, що значення полів, що зберігаються в індексі, впорядковані.

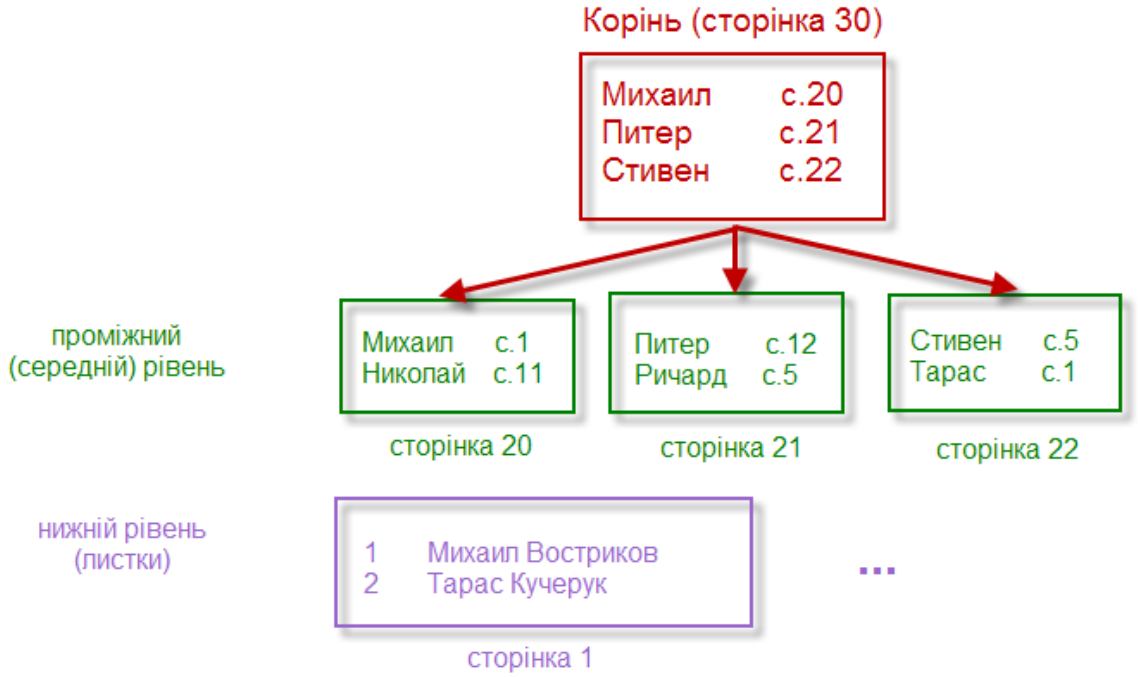
Індекси в SQL Server організовані в так зване **збалансоване дерево (balanced tree)** або **B-дерево**. Його орієнтовний вигляд наступний:



Кожен рівень індекса складається з кількох сторінок даних, за виключенням кореня, який містить тільки одну сторінку. На цих сторінках розміщуються дані, на які вказує індекс. Всі сторінки мають однакову структуру та можуть містити 8192 байти даних, включаючи заголовок розміром 96 байт. Виходячи з цього, число проміжних рівнів індекса, а також кількість сторінок на кожному з них визначається шляхом нескладних розрахунків.

Пошук даних починається з кореневої сторінки, вказівник на яку знаходиться в системній таблиці **sysindexes** та системному представленні **sys.indexes**. Кожен рівень індекса являє собою список з подвійним зв'язком (на рівень нижче та вище), а кожне значення кореневого та проміжного рівня – це перший елемент сторінки рівня нижчого по порядку. Нижній рівень індекса містить як правило записи таблиці бази даних.

Приведемо приклад індекса для поля з іменем автора.



Зверніть увагу, що дані на всіх рівнях індекса, включаючи нижній, впорядковані по ключу індекса (по імені автора). При цьому впорядкування не відображає фізичну послідовність розміщення даних.

Індекси в SQL Server, аналогічно як і в інших СУБД, слід використовувати з розумом, адже кожен створюваний індекс має об'єм, який дорівнює об'єму даних в індексному полі плюс об'єм додаткової інформації про розміщення записів. Отже, якщо створити індекси на кожне поле в таблиці, то їх сумарний об'єм буде більше, чим об'єм даних в таблиці. Крім того, існує так зване **«правило 20%»**: якщо запит на вибірку повертає більше 20% записів з таблиці, то використання індекса може сповільнити вибірку даних. Звичайно, ситуація залежить від конкретного запису і умов, накладених на вибірку, але потрібно пам'ятати, що 20% являються межею, коли ефективність використання індексів ставиться під питання.

Враховуючи вищесказане визначимо **основні випадки**, коли поле необхідно проіндексувати:

a) коли це поле дуже часто використовується в умовах пошуку в запитах;
b) коли це поле використовується при зв'язуванні таблиць (JOIN), оскільки індекси також забезпечують унікальність записів таблиці, гарантуючи цим самим цілісність даних;
c) коли це поле використовується при сортуванні (ORDER BY).

В MS SQL Server розрізняють наступні **типи індексів**:

1. В ЗАЛЕЖНОСТІ ВІД ФІЗИЧНОГО РОЗМІЩЕННЯ ІНДЕКСА:
   1.1. **кластеризовані** – це індекси, які сортують записи в таблицях або представленнях на основі їх ключових значень (але не фізично на диску (!)). Цими значеннями являються поля, включені в індекс. Проміжний рівень такого індекса по суті відсутній, а нижній рівень містить дані таблиці.
   Значення кластеризованого індекса завжди є унікальними і тому в таблиці може бути лише один такий індекс. Крім того, при додаванні кластеризованого індекса доступ до даних майже завжди здійснюється швидше, ніж у випадку некластеризованого індекса, оскільки у першому випадку виключається додатковий пошук запису даних.
   На практиці для кожної таблиці необхідно створити кластеризований індекс, який буде первинним ключем (але це не обов'язково мусить бути саме поле первинного ключа). Після цього, така таблиця буде називатись **кластеризованою**. Таблиці, які не містять кластеризованих індексів називають **кучами**.

   1.2. **некластеризовані** – це індекси, які задають логічне впорядкування для таблиці. Структура такого індекса має структуру класичного збалансованого дерева. Нижній рівень такого індекса містить значення ключа, разом з ідентифікатором запису. Для кожної таблиці можна створити до 999 некластеризованих індексів. **По замовчуванню всі індекси некластеризовані.**

2. В ЗАЛЕЖНОСТІ ВІД ТИПУ ЗНАЧЕННЯ, ЯКЕ В НИХ ЗБЕРІГАЄТЬСЯ:
   2.1. **унікальні (UNIQUE KEY) –** індекси, які утворюються при вказанні характеристики **UNIQUE** при створенні ключа. Унікальність індекса означає, що його значення не повторюються;
   2.2. **неунікальні** - створюються по замовчуванню і вказують на те, що значення їх ключів можуть повторюватись.

3. В ЗАЛЕЖНОСТІ ВІД КІЛЬКОСТІ ПОЛІВ, ЩО В НИХ ВХОДЯТЬ:
   3.1. **прості** - індексується лише одне поле;
   3.2. **складові** – це індекси, які вказують на кілька полів. Такий індекс може містити від 2 до 16 полів. Максимальний загальний розмір значень складового індекса - 900 байт. Наприклад, якщо таблиця містить поля Прізвище та Ім'я певного клієнта, для підвищення продуктивності запитів, які постійно доступаються до їх повного імені, варто створити індекс на ці поля.

4. В ЗАЛЕЖНОСТІ ВІД УПОРЯДКОВАНОСТІ ДАНИХ В ІНДЕКСІ:
   4.1. **зростаючі**;
   4.2. **спадаючі.**

5. В ЗАЛЕЖНОСТІ ВІД СТУПЕНЯ ОХОПЛЕННЯ ДАНИХ:
   5.1. **повнотабличні** – це індекси, які охоплюють всі записів індексуємої таблиці;
   5.2. **фільтровані** – це некластеризовані індекси, які охоплюють лише частину записів в індексуємій таблиці. Вони з'явились в версії SQL Server 2008 і мають переваги у випадку, якщо типові запити до таблиці в результаті повертають невелику кількість записів. Наприклад, коли у запитах приймають участь поля, які переважно містить NULL-значення.
   Такий тип індекса рекомендується використовувати у випадку наявності:
   - розріджених полів, які містять невелику кількість відмінних від NULL значень;
   - розріджених полів, які по-різному заповнені для різних категорій записів;
   - полів, що містять діапазони значень (гроші, час, дату тощо).

Крім того, існують ще такі **видиндексів**, як:

- **Повнотекстові індекси** – індекси, які використовуються у повнотекстовому пошуку (див. розділ «Повнотекстове індексування та пошук»).
- **Просторові індекси** – це індекси, які дозволяють ефективніше використовувати операції з просторовими даними в полях типу geometry, geography. Для їх створення використовується оператор CREATE SPATIAL INDEX. Більш детальніше про цей тип індексів див. в розділі «Огляд просторового індексування» електронної документації по SQL Server 2008.
- **XML-індекси** – це індекси для полів типу даних xml. Для їх створення використовується оператор CREATE XML INDEX. Більш детальніше про цей тип індексів див. в розділі «Індекси для полів типу даних xml» електронної документації по SQL Server 2008.

Не слід забувати також про те, що індекс не є частиною таблиці – це окремий об'єкт, який зв'язаний з таблицею і іншими об'єктами БД. Синтаксис створення індекса має наступний вигляд:

```
CREATE [ UNIQUE ]                          /* унікальність */
       [ CLUSTERED | NONCLUSTERED ]        /* кластеризований чи ні */
INDEX ім'я_індекса
ON     [БД.][схема.][таблиця | представлення] (поле ASC | DESC [,... n])
[ INCLUDE ( поле [ ,...n ] ) ]          /* додаткові поля на нижній рівень */
[ WHERE предикатний_вираз ]             /* для фільтрованого індекса */

/* додаткові параметри */
[ WITH  (    [ PAD_INDEX = { ON | OFF } ]
             [, FILLFACTOR = x  ]
             [, SORT_IN_TEMPDB = { ON | OFF } ]
             [, IGNORE_DUP_KEY = { ON | OFF } ]
             [, STATISTICS_NORECOMPUTE = { ON | OFF } ]
             [, DROP_EXISTING  = { ON | OFF } ]
             [, ONLINE = { ON | OFF } ]
             [, ALLOW_ROW_LOCKS = { ON | OFF } ]
             [, ALLOW_PAGE_LOCKS = { ON | OFF } ]
             [, MAXDOP = максимальна_ступінь_паралелізму ]
             [, DATA_COMPRESSION = { NONE | ROW | PAGE}
                [ ON PARTITIONS ( { вираз_номера_секції | діапазон } [ , ...n ] ) ]
             ]
)]
/* де створити індекс */
[ ON { ім'я_схеми_секціонування ( назва_поля )
       | файлова_група
       | default      /*в файловій групі по замовчуванню */
   } ]
[ FILESTREAM_ON { ім'я_файлової_групи_filestream | схема_секціонування | "NULL" } ]
```

Параметр **INCLUDE** вказує на неключові поля, які додаються на нижній рівень некластеризованого індекса. Поля типів text, ntext та image не допускаються.

Предикатний вираз в параметрі **WHERE** використовується для фільтрованих індексів і визначає ті записи таблиці, які повинні бути проіндексовані. Варто відмітити, що фільтрованими індексами не можуть бути повнотекстові та XML-індекси. Для унікальних індексів (UNIQUE) лише обрані записи повинні мати унікальне значення. Крім того, фільтровані індекси не підтримують параметр IGNORE_DUP_KEY.

Параметр **ON** дозволяє вказати місце збереження індекса. Якщо його не вказати, а таблиця або представлення несекціоновані, SQL Server розміщує індекс в тій же файловій групі, де знаходиться таблиця або представлення, для яких він створений.

Параметр **FILESTREAM_ON** дозволяє переміщувати дані FILESTREAM в іншу файлову групу FILESTREAM або схему секціонування.

Розшифруємо додаткові **параметри створення індекса**:

- **PAD_INDEX** – використовується як правило з фактором заповнення і визначає ступінь заповнення сторінок на різних рівнях індекса (розрідженість індекса). Параметр ON вказує на те, що ступінь заповнення сторінок проміжного рівня прирівнюється до відсоткового відношення вільного простору на диску, вказаному в параметрі FILLFACTOR. По замовчуванню (OFF) сторінки проміжного рівня заповнюються майже повністю – вільного місця вистачає не більше, ніж на один запис від максимального значення розміру індекса.
- **FILLFACTOR** – це коефіцієнт (фактор) заповнення, який визначає повноту заповнення кожної сторінки нижнього рівня індекса. По замовчуванню він рівний 0. Значення фактора заповнення 0 або 100 говорить про те, що сторінки нижнього рівня при створенні заповнені повністю (на 100%).
- **SORT_IN_TEMPDB** – чи зберігати проміжні результати сортування, які утворюються при створенні індекса, в базі даних tempdb. По замовчуванню проміжні результати сортування зберігаються в тій же базі, що і індекс (OFF).
- **IGNORE_DUP_KEY** – вмикає ігнорування значення ключа, яке повторюється, для унікальних індексів. По замовчуванню при повторі значень генерується повідомлення про помилку (OFF).
- **STATISTICS_NORECOMPUTE** – вказує на необхідність збереження статистичних даних про склад індекса. Значення по замовчуванню OFF вмикає автоматичне обновлення статистики.
- **DROP_EXISTING** – вказує на те, що індекс повинен знищуватись і створюватись заново, тобто на перестворюваність індекса. По замовчуванню (OFF) при існуванні одноіменних індексів генерується повідомлення про помилку, інакше (ON) існуючий індекс видаляється і перебудовується.
- **ONLINE** – забороняє запити і зміну даних в індексованих таблицях та їх індексах під час операцій над індексами по мережі (OFF; по замовчуванню). Дана опція доступна лише для випусків SQL Server Enterprise, Developer і Evaluation.
- **ALLOW_ROW_LOCKS** – дозволене чи ні блокування записів. По замовчуванню блокування записів дозволене при доступі до індексів (ON).
- **ALLOW_PAGE_LOCKS -** дозволене чи ні блокування сторінок. По замовчуванню блокування сторінок можливе при доступі до індекса (ON).
- **MAXDOP** – встановлює параметр конфігурації «максимальна ступінь паралелізму», який задає час операції індекса. MAXDOP обмежує число процесорів, які приймають участь у паралельному виконанні плану. Максимальна кількість процесорів - 64. Дана опція доступна лише для випуску SQL Server Enterprise.
- **DATA_COMPRESSION** – задає режим компресії даних для вказаного індекса або вказаних секцій. Він може приймати одне з наступних значень:
  - NONE – компресія відсутня;
  - ROW – компресія записів;
  - PAGE – компресія сторінок.

- **ON PARTITIONS** – вказує секції, до яких застосовується параметр DATA_COMPRESSION. Якщо опція ON PARTITIONS не вказана, то параметр DATA_COMPRESSION застосовується до всіх секцій секціонованого індекса.

  У виразі номера секції можна вказувати:
  - номер секції, наприклад, ON PARTITIONS (2);
  - перелік секцій, розділений комами, наприклад, ON PARTITIONS (1, 3, 7);
  - діапазон секцій разом з окремими секціями, наприклад, ON PARTITIONS (2, 4, 6 TO 8).

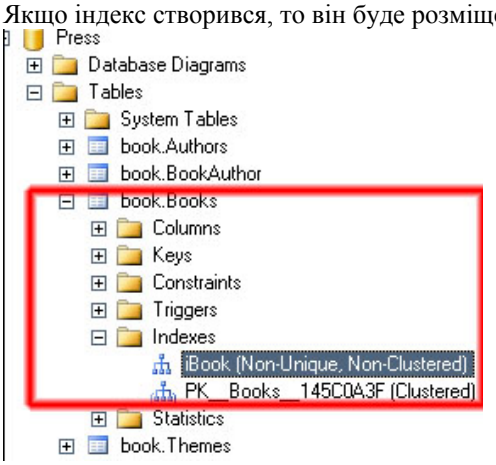
  При визначенні діапазону, номера секцій розділяються ключовим словом TO, наприклад, ON PARTITIONS (2 TO 5).

Для прикладу створимо кілька індексів:

```
-- простий некластеризований індекс на назву книги, що зберігається в таблиці Books
create index iBook on book.Books(NameBook);

-- складовий некластеризований індекс, що містить ім'я автора книги таблиці Authors
create index iNameAuthor on book.Authors(FirstName, LastName);

-- простий кластеризований індекс для поля DateOfSale таблиці Sales
create clustered index ci_salesdate on sale.Sales(DateOfSale);
```

```
-- простий некластеризований індекс для поля ID_COUNTRY таблиці Country, який
-- перебудовується. Визначається також фактор заповнення сторінки нижнього рівня індекса
create nonclustered index i_country
on global.Country(ID_COUNTRY)
with ( fillfactor = 80,
       pad_index = on,
       drop_existing = on);

-- простий некластеризований індекс з фільтром по 'США'
create index i_country_USA
on global.Country(NameCountry)
where NameCountry = 'США'
```

Якщо індекс створився, то він буде розміщений в папці **Indexes** необхідної таблиці БД:



Щоб визначити **<u>чи використовується індекс при виконанні SQL запиту</u>** використовуються наступні оператори T-SQL:

1) **SET SHOWPLAN_ALL ON/OFF** – показати весь план виконання запиту;
2) **SET SHOWPLAN_TEXT ON/OFF** – показати текст плану виконання запиту.

Наприклад:

```
use Press
go
set showplan_all on
go
select NameBook, DateOfPublish
from book.Books
```

Результатом буде виведення інформації про те, які дані та звідки беруться для формування результуючого набору, які індекси при цьому використовуються та звідки вони беруться тощо.

Індекс можна тимчасово відключити або знову включити за допомогою інструкції **ALTER INDEX**:

```
-- вимкнути індекс
ALTER INDEX { ім'я_індекса | ALL } ON об'єкт DISABLE

-- ввімкнути індекс; при цьому він перебудовується
ALTER INDEX { ім'я_індекса | ALL } ON об'єкт REBUILD
```

Варто відмітити, що при відключенні кластеризованого індекса, вся таблиця стає недоступна.

Наприклад:

```
alter index ci_salesdate on sale.Sales disable
go
alter index ci_salesdate on sale.Sales rebuild
go
```

Повний синтаксис оператора ALTER INDEX дозволяє змінити і його первинні налаштування:

```
ALTER INDEX { ім'я_індекса | ALL }
ON [БД.][схема.][таблиця | представлення]
{ REBUILD                /* перебудувати індекс (по бажанню з наступними умовами) */
  [ [PARTITION = ALL]    /* перебудовуються всі секції індекса */
    [ WITH ( додаткові_опції [ ,...n ] ) ]      /* згідно таких опцій */
      | [ PARTITION = номер_секції              /* перебудовується тільки одна секція з
                                                   наступними параметрами */
            [ WITH ( параметри_перебудови_одиничної_секції [ ,...n ] ) ]
       ]
  ]
  | DISABLE       /* відключити індекс */
  | REORGANIZE    /* реорганізувати кінцевий рівень індекса */
    [ PARTITION = номер_секції ]                /* для конкретної секції */
    [ WITH ( LOB_COMPACTION = { ON | OFF } ) ] /* всі сторінки з даними типу LOB (image,
                                                  text, ntext, varchar(max), xml
                                                  nvarchar(max), varbinary(max)) зжимаються*/
  | SET ( параметри_індекса [ ,...n ] )         /* параметри без перебудови або
                                                   реорганізації індекса */
}

-- параметри перебудови одиничної секції
{
    SORT_IN_TEMPDB = { ON | OFF }
  | MAXDOP = максимальна_ступінь_паралелізму
  | DATA_COMPRESSION = { NONE | ROW | PAGE } }
}

-- параметри індекса без перебудови або реорганізації індекса
{
    ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS = { ON | OFF }
  | IGNORE_DUP_KEY = { ON | OFF }
  | STATISTICS_NORECOMPUTE = { ON | OFF }
}
```

Видалення реляційного індекса здійснюється інструкцією **DROP INDEX**.

```
DROP INDEX ім'я_індекса
ON [БД.][схема.][таблиця | представлення]
[ WITH ( додаткові_опції_індекса [ ,...n ] ) ]
[ ,...n ]

-- додаткові опції індекса
{
    MAXDOP = максимальна_ступінь_паралелізму
  | ONLINE = { ON | OFF }
  /* куди будуть переміщенні після видалення рядки даних кінцевого рівня індекса */
  | MOVE TO {   ім'я_схеми_секціонування ( назва_поля )
              | файлова_група
              | default
            }
  /* в яку папку буде переміщена таблиця filestream кінцевого рівня індекса */
  [ FILESTREAM_ON { ім'я_файлової_групи_filestream | схема_секціонування | "NULL" } ]
}
```

Наприклад:

```
drop index ci_salesdate on sale.Sales,
           iBook on book.Books
```

Для отримання статистичних даних по індексам використовуються наступні інструкції T-SQL та системні зберігаємі процедури:

1. Створення ститистики - **CREATE STATISTICS**;
2. Обновлення статистики - **sp_updatestats**;

3. Автостатистика – **AUTO_UPDATE_STATISTICS**;
4. Перегляд статистики - **sp_helpstats**.

2. Індексовані представлення

Індексовані представлення (indexed view) дозволяють наперед розрахувати результуючий набір, який буде повертатись представленням. Наприклад, провести всі необхідні об'єднання таблиць, обрахувати функції агрегування тощо. При цьому відпадає необхідність вводити умову в кожен запит. Таким чином, продуктивність додатків, які використовують індексовані представлення замість базових таблиць, зростає в десятки разів. Це дуже суттєво, коли робота стандартного представлення пов'язана з складною обробкою великої кількості запитів, наприклад, при веденні та обрахунку статистики.

Теоретично, створення таких представлень зводиться до створення представлення та побудови для нього кластеризованого індекса. Але це не зовсім так, оскільки таблиці, які будуть приймати участь в індексованих представленнях, індекси для них, а також самі представлення повинні відповідати ряду критеріїв. Ці критерії введені для того, щоб уникнути неоднозначних розрахунків. В зв'язку з цим в індексованих представленнях дозволяється використання тільки детермінованих функцій. Більш детальніше про обмеження читайте в розділі «Створення індексованих представлень» електронної документації по SQL Server 2008.

Доречі, в редакцію SQL Server Enterprise та Developer вбудована цікава можливість оптимізатора запитів. Якщо оптимізатор визначить, що для виконання запиту ефективніше використовувати індексоване представлення, а не базову таблицю, він перепише запит. При цьому в запиті навіть не потрібно вказувати індексоване представлення – достатньо вказати таблицю, для якої воно визначено.

Преведемо невеличкий приклад побудови індексованого представлення.

```
-- задаємо параметри для підтримки індексованих представлень (якщо вони не задані)
set NUMERIC_ROUNDABORT off;
set ANSI_PADDING, ANSI_WARNINGS, CONCAT_NULL_YIELDS_NULL, ARITHABORT,
    QUOTED_IDENTIFIER, ANSI_NULLS on;
go
-- створюємо представлення
create view sale.iv_Orders
with schemabinding
as
select sh.NameShop as 'Магазин',
       s.DateOfSale as 'Дата продажу',
       sum(s.Price * s.Quantity) as 'Вартість продажу'
from sale.Sales s, sale.Shops sh
where s.ID_SHOP = sh.ID_SHOP
group by sh.NameShop, s.DateOfSale
go
-- створюємо індекс для представлення
create unique clustered index uci_ivorders
on sale.iv_Orders
go
```

Після цього в наступних запитах буде використовуватись створене нами індексоване представлення, хоча на нього і немає явного посилання.

```
select sh.NameShop as 'Магазин',
       s.DateOfSale as 'Дата продажу',
       sum(s.Price * s.Quantity) as 'Вартість продажу'
from sale.Sales s, sale.Shops sh
where s.ID_SHOP = sh.ID_SHOP AND s.DateOfSale BETWEEN '2006.12.01' AND '2007.02.01'
group by sh.NameShop, s.DateOfSale
go

select sh.NameShop as 'Магазин',
       sum(s.Price * s.Quantity) as 'Вартість продажу'
from sale.Sales s, sale.Shops sh
where s.ID_SHOP = sh.ID_SHOP AND DATEPART(mm, s.DateOfSale) = 2
group by sh.NameShop
go
```

Ряд СУБД підтримують аналоги індексованих представлень SQL Server. Наприклад, в СУБД Oracle це матеріалізовані представлення.

Підсумовуючи також можна виділити різницю між звичайним представленням та індексованим. **Стандартне представлення** – це SELECT запит, на який ссилаються по імені та який зберігається в SQL Server. Саме по собі воно не містить даних. **Індексоване представлення** – це представлення з кластеризованим індексом. Так SQL Server матеріалізує та зберігає на диску результати запиту, заданого в представленні.

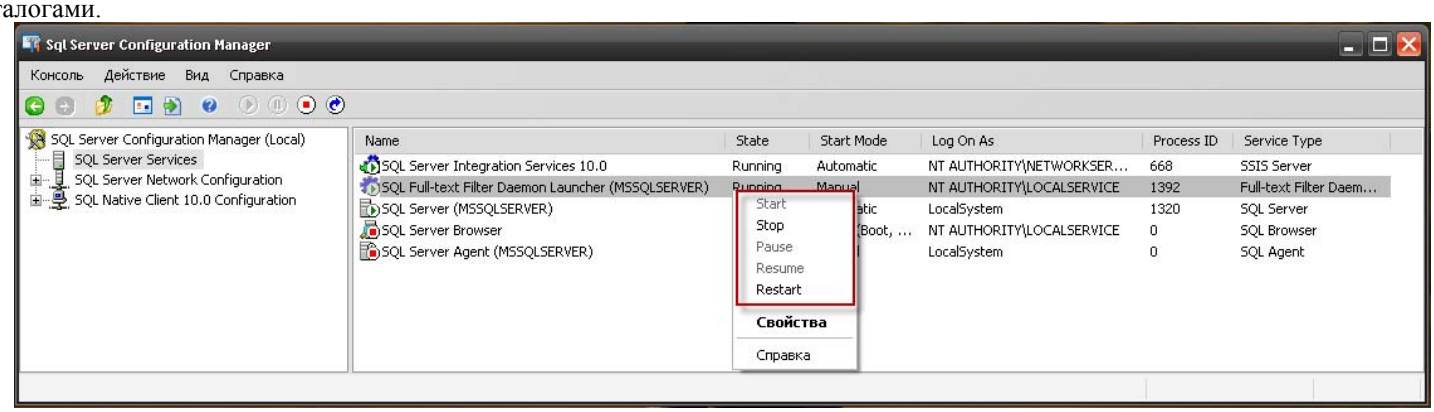
3. Повнотекстове індексування та пошук

Для підвищення ефективності вибірки при пошуку текстових даних замість операторів LIKE або оператора рівності (=), а також пошуку в неструктурованих даних, SQL Server та ряд інших СУБД рекомендують використовувати повнотекстовий пошук. В SQL Server повнотекстовий пошук забезпечує окремий компонент **Full-Text Search**.

Для того, щоб організувати повнотекстовий пошук **необхідно**:

1. Створити повнотекстовий каталог.
2. Визначити таблиці і поля, для яких необхідно використовувати повнотекстовий пошук.
3. Створити повнотекстові індекси на вказані поля.
4. Задати метод заповнення повнотекстового індекса, який буде забезпечувати узгодженість повнотекстового індекса з даними. Тобто метод заповнення відповідає за те, щоб всі зміни у відібраних полях таблиць були відображені у відповідних повнотекстових індексах.
5. Створити запит, який буде за допомогою спеціальних функцій здійснювати повнотекстовий пошук.

Повнотекстове індексування має свою власну службу – **Засіб повнотекстового пошуку диспетчера конфігурації SQL Server (SQL Full-Text Filter Daemon Launcher, MSSQLFDLaucher)**, - призначену для управління повнотекстовими каталогами.



Отже, процес організації повнотекстового індексування розпочинається з створення повнотекстового каталога, оскільки саме він забезпечує підтримку повнотекстових індексів в SQL Server. Фактично, **повнотекстовий каталог** – це набір повнотекстових індексів бази даних. **Починаючи з версії SQL Server 2008, в базі даних повнотекстовий каталог являється віртуальним об'єктом і не входить в файлову групу.** В попередніх версіях, його рекомендувалось розміщувати в окремій файловій групі бази даних.

Для створення повнотекстового каталога використовується інструкція T-SQL **CREATE FULLTEXT CATALOG**:

```
CREATE FULLTEXT CATALOG назва_каталога
     [ ON FILEGROUP файлова_група ] -- для сумісності з попередніми версіями
     [ IN PATH 'кореневий_шлях' ]   -- для сумісності з попередніми версіями,
                                    -- в наступній версії планується її виключення
     [ WITH ACCENT_SENSITIVITY = { ON | OFF } ]
     [ AS DEFAULT ]                 -- даний каталог буде каталогом по замовчуванню
     [ AUTHORIZATION ім'я_власника ]
```

Параметр **WITH** вказує на те, чи буде каталог враховувати діакритичні знаки для повнотекстового індексування. Важливо знати, що при зміні даного параметра, індекс автоматично перебудовується, на відміну від попередніх версій SQL Server, де необхідно було вручну перебудувати всі повнотекстові індекси каталога.

Після створення повнотекстового каталога необхідно створити повнотекстові індекси. Одна таблиця або індексоване представлення може мати лише один повнотекстовий індекс, хоча в ній може бути проіндексовано кілька полів. Для реалізації повнотекстового індексування таблиця повинна мати одне унікальне поле з не NULL значенням, а також мати поля типу char, varchar, varchar(max), nchar, nvarchar, text, ntext, image, xml, varbinary або varbinary(max).

Щоб створити повнотекстові індекси для двійкових типів даних (image, varbinary, varbinary(max)) використовуються спеціальні **обробники протоколу** і **фільтри**. Вони дозволяють отримувати текст з файлів Word, Excel, PowerPoint, PDF тощо, що зберігаються в базі даних. Для роботи цих сервісів необхідно в таблицю додати поле, яке містить інформацію про тип документа, який зберігається в полі двійкового типу. Після цього фільтр завантажує двійковий потік, який зберігається

в полі, видаляє всю інформацію про форматування і повертає текст документа в засіб розбиття на слова. Після цього механізм повнотекстового пошуку:

1) вичисляє лексеми, які являються зжатими формами самих слів;
2) шикує всі лексеми в полі в спеціальну зжату структуру файла, який використовується для подальшого пошуку. В зв'язку з цим, розміри самого повнотекстового індекса обмежені доступними ресурсами пам'яті комп'ютера.

Починаючи з версії SQL Server 2008 повнотекстові індекси вбудовані в компонент Database Engine, а не розміщені в файловій системі, як у попередніх версіях SQL Server.

Для створення повнотекстових індексів використовується інструкція **CREATE FULLTEXT INDEX**.

```
CREATE FULLTEXT INDEX ON таблиця
      [ ( назва_поля [ TYPE COLUMN поле_типу ] [ LANGUAGE мова_даних ] [ ,...n] ) ]
    KEY INDEX назва_унікального_індекса
        [ ON повнотекстовий_каталог ]
        [ WITH [ ( ] опції [ ,...n] [ ) ] ]

-- опції
  { CHANGE_TRACKING [ = ] { MANUAL | AUTO | OFF [, NO POPULATION ] }
    | STOPLIST [ = ] { OFF | SYSTEM | список_стоп-слів }
  }
```

Параметр **TYPE COLUMN** містить назву поля таблиці, в якому зберігається тип документа (розширення файлу, яке вказується користувачем (.DOC, .PDF, .XLS тощо)) для документів varbinary, varbinary(max) або image. Якщо індекс створюється не для двійкових даних, тоді параметр TYPE COLUMN не вказується. Це поле називається полем типу і повинно мати тип char, nchar, varchar або nvarchar.

Параметр **LANGUAGE** вказує на мову даних, які зберігаються в індексуємому полі. Даний параметр рекомендується використовувати, якщо в індексуємій таблиці міститься поля на різних мовах. Наприклад, якщо дані поля перекладені на кілька мов. Отримати список доступних кодів для мов зберігається в системному представленні **sys.fulltext_languages**.

Опція **CHANGE_TRACKING** вказує на те, як будуть відслідковуватись зміни індексованих даних (при update, insert, delete):

- MANUAL – синхронізацію даних слід здійснювати вручну, шляхом виклику інструкцій ALTER FULLTEXT INDEX … START UPDATE POPULATION (заповнення вручну) або за допомогою встановлення розкладу синхронізації змін в SQL Server Agent.
- AUTO (по замовчуванню) - зміни поширюються автоматично (за допомогою фонового процеса) в ході модифікації даних в базовій таблиці.
- OFF [, NO POPULATION] – зміни даних не відслідковуються. Аргумент NO POPULATION вказує на те, що повнотекстовий індекс буде створений без заповнення. Подальше заповнення індекса здійснюється за допомогою оператора ALTER FULLTEXT INDEX.

Варто відмітити, що **процес заповнення повнотекстового індекса дуже ресурсоємкий**, тому створення такого індекса повинно бути здійснено, коли активність бази даних невелика. В зв'язку з цим, в більшості випадків повнотекстові індекси створюють з опцією OFF і NO POPULATION, а коли активність бази даних мінімальна створюють завдання для заповнення всіх повнотекстових індексів. Після цього, для індексуємих полів, які не часто змінюються, заповнення індексів виставляється в режим AUTO.

Параметр **STOPLIST** зв'язує повнотекстовий список стоп-слів з індексом (до версії SQL Server 2008 - пропускаємих слів). Індекс не заповнюється значеннями, які являються частиною даного списка. Якщо даний параметр не вказаний, тоді з індексом зв'язується системний повнотекстовий список стоп-слів.

- OFF – вказує на те, що з повнотекстовим індексом не зв'язане жодне значення з списку стоп-слів. Тобто список не використовується.
- SYSTEM – для повнотекстового індекса буде використовуватись системний список стоп-слів.
- Також можна вказати конкретне ім'я списку стоп-слів, який буде зв'язуватись з повнотекстовим індексом.

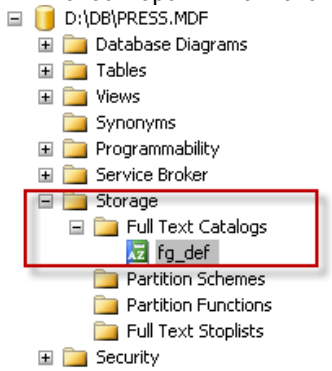
Переглянути спосок стоп-слів можна з системного представлення **sys.fulltext_stopwords**. Управляти списком стоп-слів можна за допомогою операторів **CREATE/ALTER/DROP FULLTEXT STOPLIST**, але лише при рівні сумісності 100.

Для прикладу візьмемо таблицю, яка містить інформацію про авторів. Щоб організувати повнотекстовий пошук по полям FirstName та LastName необхідно створити для них повнотекстові індекси.

Отже, спочатку створюємо повнотекстовий каталог, в якому будуть визначатись поля для індексування:

```
-- створюємо повнотекстовий каталог по замовчуванню
create fulltext catalog fg_def as default;
```

Новостворений повнотекстовий каталог в базі даних розміщується в папці **Storage->Full Text Catalogs**.



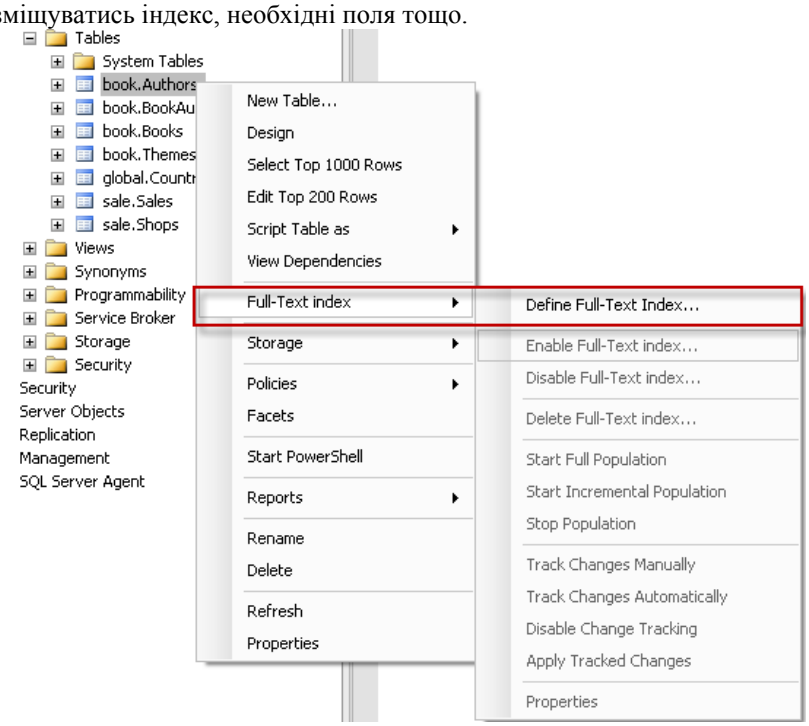
Наступним кроком є створення повнотекстового індекса на необхідні поля. Але, якщо в таблиці відсутній унікальний індекс (зазвичай ним є первинний ключ), тоді слід його створити.

```
-- якщо у таблиці немає унікального індекса, тоді створимо його
create unique index ui_AuthorId on book.Authors(ID_AUTHOR);
go
-- створюємо повнотекстовий індекс для полів FirstName та LastName з використанням
-- повнотекстового каталога fg_def та системного списку стоп-слів
create fulltext index on book.Authors(FirstName, LastName)
   key index ui_AuthorId
   on fg_def;     -- параметр необов'язковий
```

Можна також вказати код мови, яка буде являтись мовою даних в полях. Код (LCID) англійської мови – 1033, російської – 1049, української - 1058.

```
create fulltext index on book.Authors
(FirstName language 1049,
 LastName language 1049)
   key index ui_AuthorId
```

Щоб створити повнотекстовий індекс засобами Management Studio, слід обрати пункт контекстного меню **Full-Text index->Define Full-Text Index...** таблиці, в якій передбачається його наявність. Після цього запуститься візард створення повнотекстового індекса, в якому необхідно буде вказати унікальний індекс таблиці, повнотекстовий каталог, в якому буде розміщуватись індекс, необхідні поля тощо.



Для зміни встановлених параметрів повнотекстового індекса слід скористатись інструкцією **ALTER FULLTEXT INDEX**:

```
ALTER FULLTEXT INDEX ON таблиця
{
      -- активізувати чи відключити повнотекстовий індекс
      ENABLE   | DISABLE

      -- відображати зміни даних поля у повнотекстовому індексі, який йому відповідає
    | SET CHANGE_TRACKING { MANUAL | AUTO | OFF }

      -- додати поле для індекса
    | ADD ( назва_поля [ TYPE COLUMN поле_типу ] [ LANGUAGE мова_даних ] [ ,...n] )
      [ WITH NO POPULATION ]

      --видалити поле з індекса
    | DROP (назва_поля [,...n] ) [ WITH NO POPULATION ]

      -- почати заповнення повнотекстового індекса
    | START { FULL | INCREMENTAL | UPDATE } POPULATION

      -- зупинити, призупинити чи відновити процес заповнення індекса
    | {STOP | PAUSE | RESUME } POPULATION

      -- як зв'язувати повнотекстовий список стоп-слів з індексом
    | SET STOPLIST { OFF| SYSTEM | список_стоп-слів } [ WITH NO POPULATION ]
}
```

Опція **WITH NO POPULATION** вказує на те, що повнотекстовий індекс не буде заповнений після добавлення (ADD) або видалення (DROP) поля повнотекстового індекса, а також після операції SET STOPLIST. Індекс буде заповнюватись лише після виконання команди START ... POPULATION.

Параметр **START ... POPULATION** вказує SQL Server, що слід розпочати заповнення повнотекстового індекса одним з наступних способів:

- FULL – кожен рядок таблиці повинен приймати участь у повнотекстовому індексуванні, навіть якщо її записи вже були проіндексовані.
- INCREMENTAL – вказує, що для повнотекстового індексування використовуються лише ті записи, які були змінені після останнього індексування. Даний аргумент можна використовувати лише, якщо в таблиці існує поле типу timestamp. Якщо такого поля немає, тоді виконується заповнення FULL.
- UPDATE – обробляються всі операції INSERT, DELETE і UPDATE після останнього обновлення. При цьому таблиця повинна підтримувати відслідковування змін (включена опція), але індекс фонового обновлення або автоматичне відслідковування змін вимкнені.

Параметр **{STOP | PAUSE | RESUME} POPULATION** дозволяє зупинити, призупинити чи відновити призупинений процес заповнення індекса. Але в їх роботі існують невеликі обмеження:

- STOP - не дивлячись на те, що процес заповнення індекса зупиняється, автоматичне відслідковування змін або фонове обновлення індекса продовжується. Щоб їх зупинити, слід використати команду SET CHANGE_TRACKING OFF.
- PAUSE і RESUME можуть використовуватись лише для повних (FULL) заповнень. Для інших типів заповнень вони несуттєві, оскільки сканування відновлюється з моменту останньої перевірки.

Наприклад:

```
-- активізувати повнотекстовий індекс в таблиці book.Authors
alter fulltext index on book.Authors enable
go
-- заповнити повнотекстовий індекс в таблиці book.Authors
alter fulltext index on book.Authors start update population
```

Для видалення повнотекстового індекса використовується інструкція **DROP FULLTEXT INDEX**:

```
-- синтаксис
DROP FULLTEXT INDEX ON таблиця
-- наприклад
drop fulltext index on book.Authors
```

Тепер можна створювати і виконувати повнотекстові запити. Для повнотекстового пошуку використовуються свої функції, які значно ефективніші за оператор LIKE та (=). В SQL Server це предикати:

- ✓ **FREETEXT -** використовується для пошуку нечіткого співпадання. Вона визначає список різних форм шукаємого слова, фактично здійснюючи пошук по його суті. Синтаксис:

```
FREETEXT ( { назва_поля | ( список_полів ) | * },
           'текст для пошуку'
           [ , LANGUAGE мова ]
         )
```

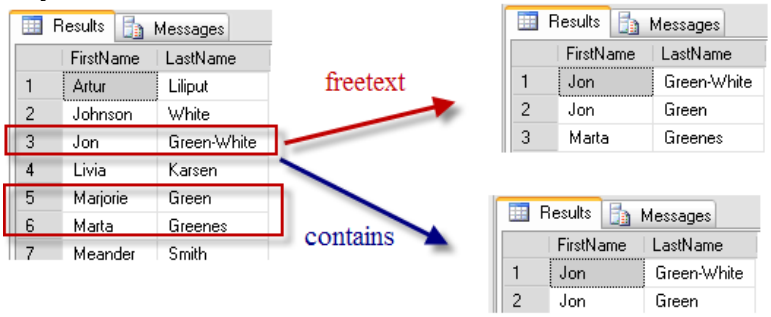
- ✓ **CONTAINS -** шукає точні співпадання слів та префіксів слів. Скорочений синтаксис даного предикату аналогічний FREETEXT.

Примітка! Тип другого параметра вищеописаних предикатів **nvarchar**, тому для коректної роботи запитів необхідно, щоб і поля для пошуку мали аналогічний тип. Крім того, щоб пошук фраз або слів на російській, українській та інших європейських мовах працював коректно, не забувайте при створенні повнотекстового індекса також вказувати параметр LANGUAGE.

Наприклад, необхідно знайти всіх авторів, в прізвищі яких зустрічається Green:

```
select FirstName, LastName
from book.Authors
where freetext(LastName, N'Green');

select FirstName, LastName
from book.Authors
where contains(LastName, N'Green');
```

Результат:



Щоб функція contains повернула результат, аналогічний freetext, слід скористатись **префіксами**. Для ідентифікації префікса необхідно використовувати подвійні лапки. При їх відсутності запит буде здійснювати пошук слова.

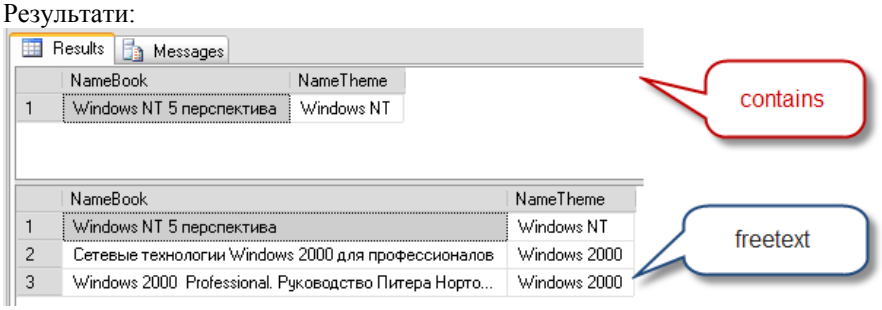
```
select FirstName, LastName
from book.Authors
where contains(LastName, N'"Green*"');
```

Тепер будуть знайдені всі прізвища авторів, які починаються з Green.

Обидві функції дозволяють використовувати в шаблонах пошуку фрази. Наприклад, серед книг небхідно знайти такі, які містять фразу «Windows NT»:

```
-- Запит знайде всі книги, в яких є слова 'Windows' та 'NT'
select b.NameBook, t.NameTheme
from book.Books b, book.Themes t
where b.ID_THEME = t.ID_THEME and contains(B.NameBook, '"Windows NT"')

-- з використання and (&), or (|), and not (&!)
select b.NameBook, t.NameTheme
from book.Books b, book.Themes t
where b.ID_THEME = t.ID_THEME and contains(B.NameBook, '"Windows" OR "NT"')

-- Запит знайде всі книги, в яких є слова 'Windows' АБО 'NT'
select b.NameBook, t.NameTheme
from book.Books b, book.Themes t
where b.ID_THEME = t.ID_THEME and freetext(B.NameBook, '"Windows NT"')
```

Оператор **AND NOT** повертає true, якщо перша умова істинна, а друга хибна.
Результати:



Примітка! Пошук символов в слові або фразі регістронезажений. Такі слова як «a», «and», «is», «the» (в залежності від мови) тощо, а також знаки пунктуації при пошуку пропускаються, оскільки вони вважаються непотрібними і включені до списку стоп-слів.

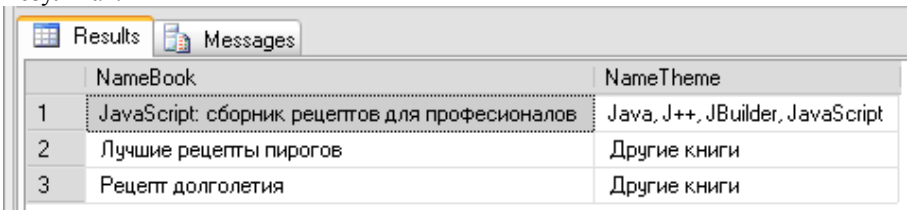
Доречі, в предикаті contains префікси використовуються і для фраз. В такому випадку, кожне слово з фрази вважається окремим префіксом. Наприклад, при пошуку фрази «local wine*» будуть відібрані рядки з текстом «local winery», «locally wined and dined» тощо.

Функця contains може приймати різні параметри, що дозволяє налаштувати її використання. Два варіанти використання, крім «класичного» пошуку слів, ми вже розглянули – це використання префіксів та фраз в шаблонах. Розглянемо її можливості ширше:

1. Для пошуку варіантів фраз використовують функцію **FORMSOF**, перший параметр якої вказує на принцип генерації словоформ:
   - **INFLECTIONAL** - пошук по основі слова, наприклад, слово «погода» видасть відповідність для «погода», «погоди», «погоджувати» тощо;
   - **THESAURUS** - пошук синонімів, які визначені в XML-файлі тезауруса. Наприклад, слово «метал» може мати синонім «золото», «алюмінвй», «залізо» тощо. По замовчуванню файл тезауруса пустий і розміщується по шляху місце_встановлення_SqlServer\Microsoft SQL Server\MSSQL10_50.MSSQLSERVER\MSSQL\FTDATA\.

```
create fulltext index on book.Books(NameBook language 1049)
key index ui_BooksId
go
select b.NameBook, t.NameTheme
from book.Books b, book.Themes t
where b.ID_THEME = t.ID_THEME
      and contains(B.NameBook, 'FORMSOF(INFLECTIONAL, рецепт)')
```

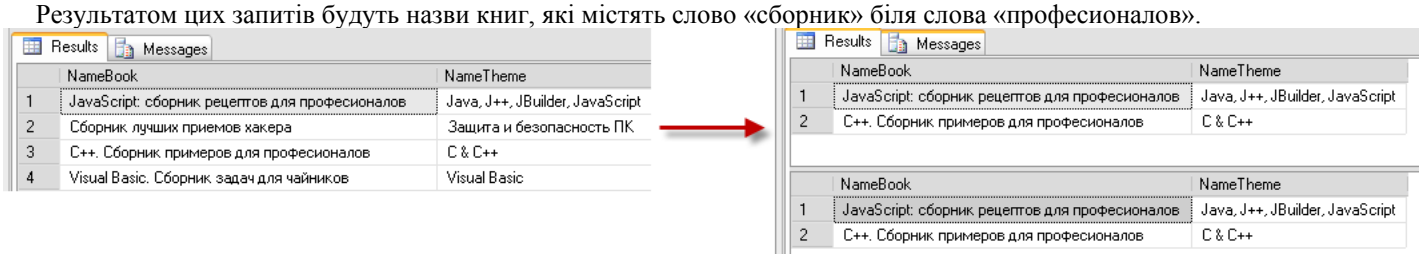
Результат:



2. Пошук слів або фраз згідно наближеності їх місцерозташування. Для цього використовується ключове слово NEAR (~). Відстань між заданими словами вимірюється також в словах, з врахуванням приналежності до речення або абзаца. Предикат CONTAINS рідко використовується з цим ключовим словом, оскільки ранг результатуючих слів не може бути визначений напряму.

```
select b.NameBook, t.NameTheme
from book.Books b, book.Themes t
where b.ID_THEME = t.ID_THEME and contains(B.NameBook, 'сборник NEAR професионалов')

-- або
select b.NameBook, t.NameTheme
from book.Books b, book.Themes t
where b.ID_THEME = t.ID_THEME and contains(B.NameBook, 'сборник ~ професионалов')
```

Результатом цих запитів будуть назви книг, які містять слово «сборник» біля слова «професионалов».



Results | Messages

	NameBook	NameTheme
1	JavaScript: сборник рецептов для професионалов	Java, J++, JBuilder, JavaScript
2	Сборник лучших приемов хакера	Защита и безопасность ПК
3	C++. Сборник примеров для професионалов	C & C++
4	Visual Basic. Сборник задач для чайников	Visual Basic

Results | Messages

	NameBook	NameTheme
1	JavaScript: сборник рецептов для професионалов	Java, J++, JBuilder, JavaScript
2	C++. Сборник примеров для професионалов	C & C++

	NameBook	NameTheme
1	JavaScript: сборник рецептов для професионалов	Java, J++, JBuilder, JavaScript
2	C++. Сборник примеров для професионалов	C & C++

3. Функція **ISABOUT** дозволяє задавати відносну вагу певним критеріям пошуку (в діапазоні від 0.0 до 1.0).

```
select b.NameBook, t.NameTheme
from book.Books b, book.Themes t
where b.ID_THEME = t.ID_THEME and contains(B.NameBook, 'ISABOUT(Windows weight(.8),
                                                        NT weight (.3))')
```

В даному запиті нас більше цікавить входження слова «Windows», тому йому ми встановили більший ваговий коефіцієнт.

Варто відмітити, що ключове слово WEIGHT не впливає на запити з CONTAINS, але впливає на значення RANK, яке повертає CONTAINSTABLE.

Функції **CONTAINS** і **FREETEXT** мають свої аналоги для роботи з таблицями – це **CONTAINSTABLE** та **FREETEXTTABLE**. Відмінність останніх заключається лише в тому, що вони повертають набір записів, який повинен бути з'єднаний з іншою таблицею на основі значення ключа. Разом з результатами пошуку вони повертають **<u>додаткові поля</u>**:

- **RANK** – визначає ранг відповідності кожного запису шаблону: чим вище ранг, тим точніше співпадіння (число від 0 до 1000);
- **KEY** – ключеве поле вхідної таблиці, яке відповідає знайденим записам.

Синтаксис їх наступний:

```
CONTAINSTABLE (  таблиця, { поле | (список_полів) | * },
                 'текст для пошуку'
                  [ , LANGUAGE мова ]
                  [ , перші_n_по_рангу ]
              )

FREETEXTTABLE (  таблиця, { поле | (список_полів) | * },
                 'текст для пошуку'
                 [ , LANGUAGE мова ]
                 [ , перші_n_по_рангу ]
              )
```

Для прикладу напишемо запит, який шукає збірники в таблиці book.Books. Пошук здійснюється по всім полям.

```
select *
from containstable(book.Books, *, N'сборник')
```

Результат:

Results | Messages

	KEY	RANK
1	27	80
2	35	80
3	36	80
4	37	80

Вищерозглянутий приклад можна переписати наступним чином:

```
select  cb.Rank as 'Ранг', b.ID_BOOK as 'Код',
        b.NameBook as 'Назва книги',
        a.FirstName+' '+a.LastName as 'Автор'
from containstable(book.Books, *, N'сборник') as cb,
      book.Books as b, book.Authors as a
where cb.[KEY] = b.ID_BOOK AND b.ID_AUTHOR = a.ID_AUTHOR
```

Як видно з прикладу, поле KEY повинно бути вказано в квадратних дужках, оскільки воно являється ключовим словом T-SQL.

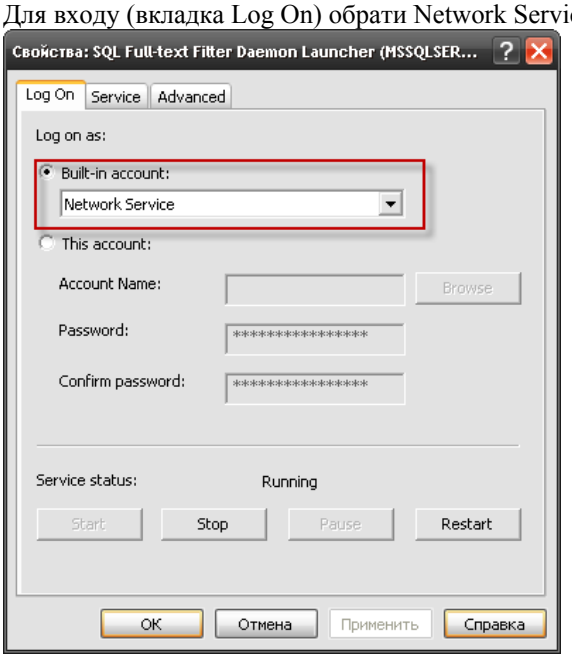
Результат:

Results | Messages

	Ранг	Код	Назва книги	Автор
1	80	27	JavaScript: сборник рецептов для професионалов	Сергей Парижский
2	80	35	Сборник лучших приемов хакера	Jon Green-White
3	80	36	C++. Сборник примеров для професионалов	Михаил Мочерный
4	80	37	Visual Basic. Сборник задач для чайников	Livia Karsen

Примітка! Іноді, SQL Server при виконанні запитів з повнотекстовим пошуком може видавати помилку «SQL Server encountered error 0x80070218 while communicating with full-text filter daemon host (FDHost) proces». Для того, щоб уникнути даної помилки необхідно:

1. Зайти в Диспетчер конфігурації SQL Server.
2. Обрати властивості служби SQL Full-text Filter Deamon Launcher.
3. Для входу (вкладка Log On) обрати Network Service.



4. Перезапустити службу і екземпляр сервера.

4. Тригери

Тригер – це спеціалізована процедура, яка автоматично викликається SQL Server при виникненні подій в базі даних. В SQL Server з версії SQL Server 2005 підтримуються два **типи тригерів**:

1. **DML-тригери** – виконуються при виникненні DML (**D**ata **M**anipulation **L**anguage - Мова Маніпулювання Даними) подій: додаванні (INSERT), видаленні (DELETE) або обновленні (UPDATE) записів таблиць або представлень. Такі тригери завжди прив'язані до певної таблиці або представлення і можуть перехвачувати дані лише її/його.
2. **DDL-тригери** – виконуються при виникненні DDL (**D**ata **D**efinition **L**anguage - Мова Визначення Даних) подій: створення (CREATE), зміна (ALTER) або видалення (DROP) об'єктів. Окрему підгрупу утворюють тригери входу, які спрацьовують на подію LOGON, яка виникає при встановленні сеансу користувача. DDL-тригери використовуються для адміністрування бази даних, наприклад, для аудиту і управління доступом до об'єкта.

Тригери можуть бути створені як за допомогою інструкцій Transact-SQL, так і за допомогою методів збірок, створених в середовищі CLR платформи .NET Framework та передані екземпляру SQL Server. Всі тригери не мають параметрів і не виконуються явно. При виникненні події, до якої прив'язаний тригер, SQL Server автоматично його запускає.

Отже, розпочнемо наше знайомство, а почнемо ми з **DML-тригерів**. Синтаксис створення такого тригера має наступний вигляд:

```
CREATE TRIGGER [схема.] ім'я_тригера
ON { таблиця | представлення }      -- для кого створюється тригер
[ WITH ENCRYPTION                   -- зашифрувати вхідний код тригера
      [, EXECUTE AS умова ]         -- контекст виконання
]
{ FOR | AFTER           -- після зміни даних
      | INSTEAD OF }    -- замість SQL команди, для якої вони оголошені
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] } -- на яку дію з даними
[ WITH APPEND ]         -- додати тригер існуючого типу
[ NOT FOR REPLICATION ] -- тригер не виконається, якщо в ході реплікації буде змінена
                        -- таблиця, на яку він вказує
AS
 { тіло_тригера | EXTERNAL NAME збірка.клас.метод }
```

Після вказання імені тригера в інструкції **ON** необхідно вказати назву таблиці або представлення, для якого створюється тригер.

За допомогою інструкції **WITH** можна ввімкнути шифрування коду тригера (ENCRIPTION) або вказати контекст виконання (EXECUTE AS), в якому буде працювати тригер. Параметр EXECUTE AS в основному використовується для перевірки привілеїв (прав доступу) на об'єкти бази даних, на які ссилається тригер.

Як видно з синтаксису, DML-тригери можна запускати в **двох режимах**: AFTER та INSTEAD OF. Тригера BEFORE, який наявний в багатьох СУБД, в MS SQL Server не існує.

Тригер INSTEAD OF може породжувати подію (команду), для якої він оголошений. Причому ця подія буде виконуватись так, ніби тригера INSTEAD OF не існувало. Наприклад, якщо ви хочете перевірити певну умову до виконання команди INSERT, ви можете оголосити тригер INSTEAD OF INSERT. Тригер INSTEAD OF буде виконувати перевірку, а потім виконувати команду INSERT для таблиці. Оператор INSERT буде виконуватись звичним чином, не породжуючи рекурсивних викликів тригера INSTEAD OF.

Слід також пам'ятати, що **по замовчуванню всі тригери активні** та виконуються після здійсненої дії, тобто мають встановлений параметр **AFTER**.

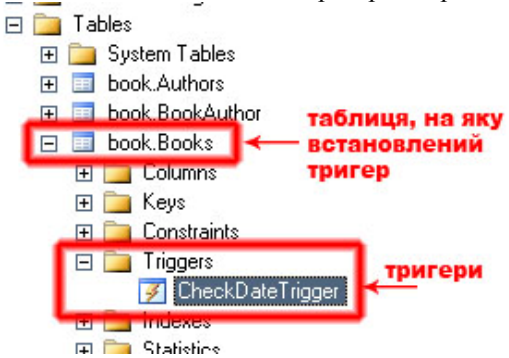
Зауваження щодо побудови тригерів**:**

1. Якщо при оголошенні тригера, вказати єдине ключове слово FOR, то аргумент AFTER використовується по замовчуванню.
2. Не можна створювати тригери INSTEAD OF для модифікованих представлень.
3. Тригери AFTER використовуються лише для таблиць.
4. Параметр WITH ENCRYPTION не може бути вказаний для тригерів CLR.
5. Аргумент WITH APPEND встановлений **лише для сумісності** з попередніми версіями (на рівні 65). При цьому він може використовуватись лише при вказанні параметра FOR без INSTEAD OF і не може вказуватись для тригерів CLR. **В зв'язку з тим, що в наступній версії SQL Server аргумент WITH APPEND планується повністю вилучити, рекомендується його уникати.**

Існує також ряд **правил**, яких слід дотримуватись при **створенні тригерів**:

- ✓ НЕ МОЖНА створювати тригери для тимчасових таблиць, але вони можуть звертатись до них;
- ✓ НЕ МОЖНА створювати, змінювати, видаляти резервні копії або відновлювати з резервної копії бази даних;
- ✓ тригери можуть використовувати для забезпечення цілісності даних, але їх не слід використовувати замість оголошення цілісності шляхом встановлення обмеження FOREIGN KEY;
- ✓ тригери не можуть повертати результуючі набори, тому при використанні оператора select в тілі тригера слід бути дуже уважним. При цьому в багатьох випадках, разом з оператором select використовується директива IF EXISTS;
- ✓ підтримуються рекурсивні AFTER тригери, за умови встановлення параметра бази даних RECURSIVE_TRIGGERS в значення ON;
- ✓ можна створювати вкладені тригери (підтримується до 32 рівнів вкладеності), які фактично являються неявною рекурсією. Для їх підтримки необхідно встановити параметр NESTED TRIGGERS;
- ✓ в тілі DML-тригера НЕ МОЖНА використовувати оператори:
  - всі операції CREATE/ALTER/DROP;
  - TRUNCATE TABLE;
  - RECONFIGURE;
  - LOAD DATABASE або TRANSACTION;
  - GRANT і REVOKE;
  - SELECT INTO;
  - UPDATE STATISTICS.

Тригери також являються об'єктами БД і в SQL Management Studio вони розміщуються в папці **«Triggers»** таблиці, на яку встановлений той чи інший тригер. Наприклад:



В MS SQL Server в межах DML-тригерів використовують довідкові таблиці **INSERTED** і **DELETED**, які мають ту ж структуру, що і базові таблиці тригера. Вони розміщуються в оперативній пам'яті, оскільки являються логічними таблицями. Працюють вони наступним чином:

- **Коли в базову таблицю додаються нові дані** (новий запис), то ці ж дані додаються спочатку в базову таблицю, а потім в таблицю inserted. Їх наявність в таблиці inserted позбавляє необхідності створювати спеціальні змінні для доступу до цієї інформації.
- **Коли рядок видаляється з таблиці**, то він записується в таблицю deleted, а потім видаляється з базової таблиці.
- **Коли рядок обновлюється**, то старе значення запису записується в таблицю deleted і видаляється з базової таблиці, потім обновлений запис записується в базову таблицю, а далі в таблицю inserted.

Тобто ми можемо використовувати таблицю DELETED для отримання значень записів, які видаляються з таблиці, а таблицю INSERTED для отримання нових записів перед їх фактичною вставкою. В інших серверних СУБД такі задачі виконують схожі механізми, наприклад, в InterBase/Firebird та Oracle – це контекстні змінні OLD і NEW.

Для кращого розуміння роботи з тригерами напишемо **кілька прикладів**.

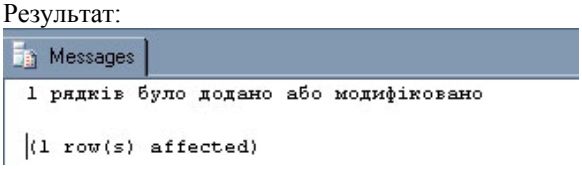
1. Класичний тригер, який буде спрацьовувати при кожній вставці даних в таблицю Authors і повертає повідомлення про кількість змінених рядків. В цій нелегкій на перший погляд справі нам допоможе глобальна змінна **@@rowcount**, що містить дані про кількість модифікованих рядків, в результаті роботи тригера.

```
create trigger addAuthor
on book.Authors
for insert, update
as
      raiserror('%d рядків було додано або модифіковано', 0, 1, @@rowcount)
      return
```

Додаємо нового автора в нашу БД:

```
insert into book.Authors(FirstName, LastName, id_country)
values ('Artur', 'Liliput', 1)
```

Результат:



2. Тригер, який при додаванні нових даних про книгу, дата видавництва якої більше місяця, буде викидати повідомлення про помилку.

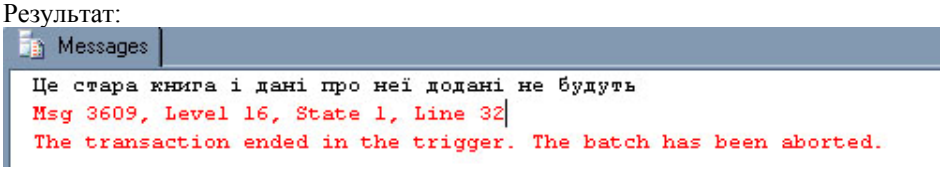
```
create trigger CheckDateTrigger
on book.Books
for insert
as
begin
      declare @InsDate smalldatetime

-- отримуємо дату видавництва книги, яка додається
      select @InsDate = DateOfPublish
      from inserted
```

```
-- перевіряємо скільки пройшло днів з дня її видання
      if (@InsDate <= getdate()-30)
      begin
            raiserror('Це стара книга і дані про неї додані не будуть',0,1)
            rollback transaction
      end
      else
            PRINT('Дані додані успішно')
end
```

Додамо нову книгу:

```
insert into book.Books(NameBook, id_theme, id_author, Price, DrawingOfBook,
                       DateOfPublish, Pages)
values ('Адміністрування MS SQL Server 2005', 21, 1, 125.0, 3500, '2007.09.01', 726)
```

Результат:



3. Тригер, що забороняє видалення книги, якщо вона найбільше продається, тобто є лідером продаж.

```
create trigger CheckBookDelete
on book.Books
for delete
as
begin
      declare @NameBook varchar(25), @BestBook varchar(25)

-- Отримуємо назву видаляємої книги
      select @NameBook = deleted.NameBook
      from deleted

      declare @Zvit table (nameB varchar(25), quantity int)
-- Отримуємо інформацію про назви книг та їх кількості продажу (популярність)
      insert @Zvit
            select b.NameBook, count(s.id_book)
            from book.Books b, sale.Sales s
            where b.id_book = s.id_book
            group by b.NameBook

-- знаходимо саму продаваєму книгу
      select @BestBook = z.nameB
      from @Zvit z
      where z.quantity = (select max(quantity)
                          from @Zvit)

-- перевіряємо чи назви співпали
      if (@BestBook = @NameBook)
      begin
            raiserror('Ви не можете видалити дану книгу',0,1)
            rollback transaction
      end
      else
      begin
            print ('Книга видалена успішно')
      end
end
```

4. Тригер, який при видаленні книги тематики «Програмування» викидає повідомлення про помилку.

```
create trigger NotProgrammingDelete
on book.Books
instead of delete
as
begin
      declare @ProgID int

-- отримуємо ідентифікатор тематики "Програмування"
      select @ProgID = id_theme
      from book.Themes
      where NameTheme = 'Програмування'
-- перевіряємо чи співпадають ідентифікатори в книгах тематики, які видаляються
      if exists ( select *
                  from deleted
                  where id_theme = @ProgID)
        raiserror ('Книгу не можна видалити,0,1)
end
```

Як вже було сказано вище, для DDL-операцій, таких як CREATE, ALTER, DROP, GRANT, DENY, REVOKE та UPDATE STATISTICS, використовуються **DDL-тригери**. Крім контролю і моніторингу даних операцій, DDL-тригери дозволяють обмежувати їх виконання, навіть якщо користувач має відповідні права. Наприклад, можна заборонити користувачам певних груп змінювати і видаляти таблиці. Для цього достатньо створити DDL-тригер для подій DROP TABLE та ALTER TABLE, який буде робити відкат цих операцій та видавати відповідне повідомлення про помилку.

Синтаксис створення DDL-тригера має наступний вигляд:

```
CREATE TRIGGER ім'я_тригера
ON { ALL SERVER | DATABASE }       -- область дії тригера
[ WITH ENCRYPTION [, EXECUTE AS умова ] ]
{ FOR | AFTER } { ім'я_події | група_подій } [ ,...n ]
AS
 { тіло_тригера | EXTERNAL NAME збірка.клас.метод }
```

Як видно з синтаксису, після інструкції **ON** необхідно вказати **область дії тригера**:

- **ALL SERVER** – поточний сервер. Тригер буде спрацьовувати при виникненні визначених подій в будь-якому місці в рамках поточного сервера. Наприклад, CREATE_DATABASE, ALTER_LOGIN, ALTER_INSTANCE тощо.
- **DATABASE** – поточна база даних. Тригер буде спрацьовувати при виникненні визначених подій на рівні поточної бази даних або нижчих рівнях. Наприклад, CREATE_TABLE, DROP_DEFAULT, ALTER_USER тощо.

Для зручності адміністрування можна використовувати групи подій, наприклад, група подій DDL_TABLE_EVENTS включає в себе події, які стосуються таблиці: CREATE_TABLE, ALTER_TABLE і DROP_TABLE. Групи подій мають ієрархічну структуру. Наприклад, група подій DDL_TABLE_VIEW_EVENTS охоплює всі інструкції T-SQL в групах DDL_TABLE_EVENTS, DDL_VIEW_EVENTS, DDL_INDEX_EVENTS та DDL_STATISTICS_EVENTS.

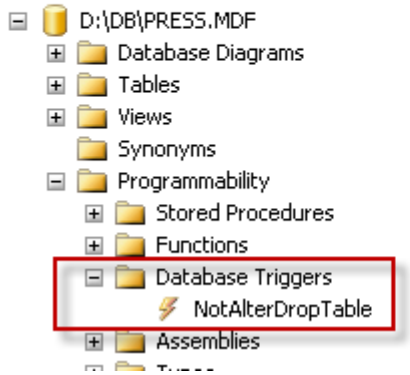
Детальний список назв подій приведений в розділі «DDL-події» електронної документації по SQL Server 2008, а груп подій у розділі «Групи DDL-подій».

Проілюструємо код вищенаведеного прикладу використання DDL-тригера, тобто на заборону зміни і видалення таблиць:

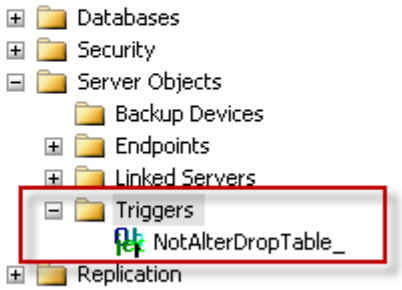
```
create trigger NotAlterDropTable
on DATABASE
for DROP_TABLE, ALTER_TABLE
as
begin
      print 'Модифікація та видалення таблиць заборонені. Зверніться до адміністратора.'
      rollback
end
```

Варто відмітити, що DDL-тригери не викликаються на подій, які впливають на тимчасові таблиці і зберігаємі процедури. DDL-тригери також не обмежені областю схеми і після створення тригери рівня бази даних знаходяться в папці **«Programmability/Database Triggers»**, а тригери рівня сервера - в папці **«Triggers»** папки **«Server Objects»** (об'єкти сервера).

DDL-тригер рівня бази даних



DDL-тригер рівня сервера



Окрему підгрупу DDL-тригерів становлять **тригери входу (logon trigger)**:

```
CREATE TRIGGER ім'я_тригера
ON ALL SERVER
[ WITH ENCRYPTION [, EXECUTE AS умова ] ]
{ FOR | AFTER } LOGON
AS
 { тіло_тригера | EXTERNAL NAME збірка.клас.метод }
```

Такі тригери виконуються у відповідь на подію **LOGON**, яка викликається при встановленні користувацького сеансу з екземпляром сервера. Тригери входу спрацьовують після завершення етапу аутентифікації при вході, але перед тим, як сеанс користувача реально встановлюється. Отже, всі повідомлення про помилку функції raiserror або інструкції PRINT, які викликаються в тригері, перенаправляються в журнал помилок SQL Server. Якщо ж користувач ввів невірний логін чи пароль, то тригер входу не спрацьовує.

Відмітимо, що в тригерах входу не підтримуються розприділені транзакції. Тригери входу можуть створюватись з будь-якої бази даних, але належать вони базі даних **master**.

В наступному прикладі заборонимо користувачу з логіном 'vasja_pupkin' підключення до SQL Server.

```
use master;
go
create trigger TriggerConnection
on ALL SERVER
with execute as 'vasja_pupkin'
for logon
as
begin
      if ORIGINAL_LOGIN()= 'vasja_pupkin'
      begin
            print 'Такий логін заборонений на сервері. Зверніться до адміністратора.'
            rollback
      end
end
```

Тригери можна змінювати, тимчасово відключити або видалити.

Модифікувати тригер можна за допомогою інструкції **ALTER TRIGGER**:

```
-- для DML-тригера
ALTER TRIGGER [схема.] ім'я_тригера
ON { таблиця | представлення }
[ WITH ENCRYPTION [, EXECUTE AS умова ] ]
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ NOT FOR REPLICATION ]
AS
 { тіло_тригера | EXTERNAL NAME збірка.клас.метод }
```

```
-- для DDL-тригера
ALTER TRIGGER ім'я_тригера
ON { ALL SERVER | DATABASE }
[ WITH ENCRYPTION [, EXECUTE AS умова ] ]
{ FOR | AFTER } { ім'я_події | група_подій } [ ,...n ]
AS
{ тіло_тригера | EXTERNAL NAME збірка.клас.метод }

-- для тригера входу
ALTER TRIGGER ім'я_тригера
ON ALL SERVER
[ WITH ENCRYPTION [, EXECUTE AS умова ] ]
{ FOR | AFTER } LOGON
AS
{ тіло_тригера | EXTERNAL NAME збірка.клас.метод }
```

Відключити або включити тригер можна за допомогою інструкції ALTER TABLE або за допомогою наступних інструкцій:

```
-- включити тригер
ENABLE TRIGGER { [ схема. ] ім'я_тригера [ ,...n ] | ALL }
ON { таблиця | представлення | DATABASE | ALL SERVER }

-- вимкнути тригер
DISABLE TRIGGER { [ схема. ] ім'я_тригера [ ,...n ] | ALL }
ON { таблиця | представлення | DATABASE | ALL SERVER }
```

Опція **ALL** вказує на те, що всі тригери в області дії значення параметра ON будуть включені або вимкнені.

Опції **DATABASE** та **ALL SERVER** вказує на те, що інструкція стосується DDL-тригера рівня бази даних або сервера (включаючи тригер входу) відповідно.

Видалення тригера здійснюється оператором **DROP TRIGGER**:

```
-- для DML-тригера
DROP TRIGGER [схема.] ім'я_тригера [ ,...n ] [ ; ]

-- для DDL-тригера
DROP TRIGGER ім'я_тригера [ ,...n ]
ON { DATABASE | ALL SERVER }

-- для тригера входу
DROP TRIGGER ім'я_тригера [ ,...n ]
ON ALL SERVER
```

Імена всіх тригерів зберігаються в системній таблиці **sysobjects** та системному представленні **sys.objects.** Метадані DDL- та DML-тригерів рівня бази даних можна переглянути за допомогою нового представлення **sys.triggers**. Якщо поле parent_class_desc даного представлення має значення «DATABASE», тоді це DDL-тригер і його областю дії є база даних. Тіло тригера можна отримати з представлення **sys.sql_modules** (зв'язавши його з sys.triggers по полю object_id), а код створення – з системної таблиці **syscomments**. Метадані CLR-тригерів доступні з представлення **sys.assembly_modules**, яке також слід зв'язати з sys.triggers по полю.

Інформація про DDL-тригери рівня сервера, включаючи тригери входу зберігається в системному представленні **sys.server_triggers**. Тіло тригера рівня сервера можна отримати з представлення **sys.server_sql_modules**, а метадані CLR-тригера серверного рівня – з представлення **sys.server_assembly_modules**.

На завершення відмітимо, що SQL Server надає інформацію про події, які він відловлює, у вигляді XML. Вони доступні через нову вбудовану функцію **EVENTDATA()**, яка повертає XML-дані. Ця можливість дозволяє застосовувати DDL-тригери для аудиту DDL-операцій в базі даних.

Для цього, наприклад, можна створити таблицю аудиту з полем, яке містить XML-дані. Потім створюємо DDL-тригер з параметром EXECUTE AS SELF для DDL-подій або груп подій, які вас цікавлять. Тіло такого DDL-тригера може просто виконувати вставку (INSERT) XML-даних, які повертаються EVENTDATA(), в таблицю аудиту.

5. Домашнє завдання

1. В своїй базі даних знайдіть всі таблиці, які не мають кластеризованих індексів. Для кожної з них додайте кластеризований індекс або змініть первинний ключ на кластеризований.
2. Знайдіть повільно виконувані запити. Створіть некластеризовані індекси, для підвищення швидкості викнання цих запитів.
3. Візьміть одне з представлень, створених на попередньому занятті (в якості домашнього завдання), та перетворіть його в індексоване.
4. В базі даних створіть повнотекстовий каталог.
5. Написати запит з повнотекстовим пошуком, який буде виводити назви магазинів, які містять фразу «книга». Разом з назвою магазину необхідно вивести країну їх розташування.
6. Здійсніть повнотекстовий пошук всіх книг, в назвах яких містяться слова «мова», «скрипти» або «Web», при цьому для кожного слова задається своя вага.
7. Напишіть запит, який в результаті роботи виведе книги, з вказанням їх авторів та рангами відповідності умові в наступному порядку важливості тематик: «C & C++», «Visual C», «.NET Framework». Для цього скористайтесь можливістю встановлення вагових коефіцієнтів для частин шаблону.
8. Виведіть всі назви магазинів, які продавали книги на протязі останнього року. Вкажіть також обсяг продажу книг, місцерозташування магазину та ранг відповідності умові пошуку. Умова: в назвах книг зустрічаються наступні співпадіння: «підручник» і «професіоналів» або «посібник» і «чайників».

   Примітка! Для запитів з повнотекстовим пошуком створіть необхідні повнотекстові індекси.

Написати наступні тригери:

1. Тригер, який при продажу книги автоматично змінює кількість книг в таблиці Books. (Примітка! Додати до таблиці Books необхідне поле кількості наявних книжок QuantityBooks).
2. Тригер на перевірку, щоб кількість продажу книг не перевищила наявну.
3. Тригер, який при видаленні книги, копіює дані про неї в окрему таблицю «DeletedBooks».
4. Тригер, який слідкує, щоб ціна продажу книги не була меньше основної ціни книги з таблиці Books.
5. Тригер, що забороняє додавання нової книги, для якої не вказана дата випуску та викидає відповідне повідомлення про помилку.
6. Тригер або набір тригерів, які забороняють видалення об’єктів будь-якої бази даних на сервері (таблиць, значень по замовчуванню тощо).
7. Додайте до бази даних тригер, який виконує аудит змін даних в таблиці Books.

Copyright © 2010 Iryn Petrova