ШАГ компьютерная АКАДЕМИЯ

Программирование и администрирование СУБД

Microsoft®

SQL Server

Урок №7

Содержание

1. Индексы в SQL Server

Для повышения скорости выполнения запросов и доступа к данным базы данных рекомендуется использовать индексы. Без индексов СУБД читает данные, просматривая каждую страницу таблиц, указанных в операторе SQL, проводит так называемое **сканирование таблицы**.

Понятие индекс встречается не только в теории баз данных и скорее всего вы с ним уже сталкивались в повседневной жизни. Индексы баз данных можно сравнить с библиотечным каталогом, в котором все книги записаны в карточки и упорядоченные определенным образом: по алфавиту или по темам, а в каждой карточке написано, где именно в архивах размещается данная книга. Другим примером может служить предметный указатель (индекс) в конце книги, который помогает читателю легко найти нужное место. Стоит напомнить, что в него входит только небольшое количество терминов, отсортированых по алфавиту, что позволяет быстро найти необходимую информацию. Похожим образом организованы и индексы в MS SQL Server.

Сам **индекс** представляет собой упорядоченный логический указатель на записи в таблице. **Указатель** означает, что индекс содержит значение одного или нескольких полей в таблице и адреса страниц данных, на которых размещаются эти значения. Иными словами, индекс состоит из пар "значение поля – физическое размещение этого поля". Таким образом, по значению поля (или полей),

входящих в индекс, можно быстро найти то место в таблице, где расположена запись с искомым значением. **Упорядоченный** означает, что значение полей, хранящихся в индексе, упорядочены.

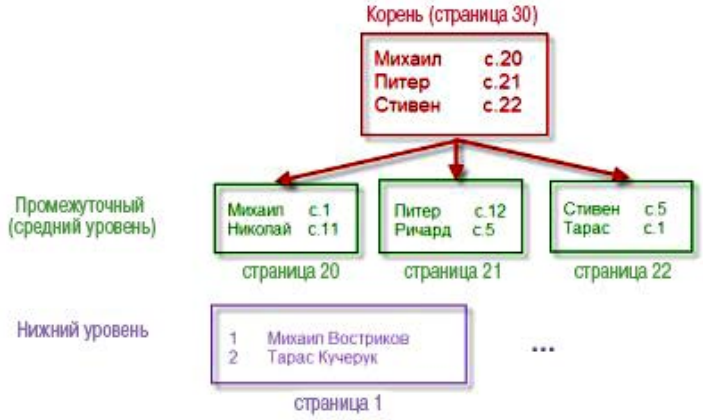
Индексы в SQL Server организованы в так называемое **сбалансированное дерево (balanced tree)** или **B-дерево**. Его ориентировочный вид следующий:

Каждый уровень индекса состоит из нескольких страниц данных, за исключением корня, который содержит только одну страницу. На этих страницах размещаются данные, на которые указывает индекс. Все страницы имеют одинаковую структуру и могут содержать 8192 байта данных, включая заголовок размером 96 байт. Исходя из этого, число промежуточных уровней индекса, а также количество страниц в каждом из них определяется путем несложных расчетов.

Поиск данных начинается с корневой страницы, указатель на которую находится в системной таблице **sysindexes** и системном представлении **sys.indexes**. Каждый уровень индекса представляет собой список с двойной связью (на уровень ниже и выше), а каждое значение корневого и промежуточного уровня – это первый элемент страницы

уровня ниже по порядку. Нижний уровень индекса содержит, как правило, записи таблицы базы данных.

Приведем пример индекса для поля с именем автора.



Обратите внимание, что данные на всех уровнях индекса, включая нижний, упорядочены по ключу индекса (по имени автора). При этом упорядочение не отражает физическую последовательность размещения данных.

Индексы в SQL Server, аналогично тому, как и в других СУБД, следует использовать с умом, ведь каждый создаваемый индекс имеет объем, равный объему данных в индексной поле плюс объем дополнительной информации о размещении записей. Итак, если создать индексы на каждое поле в таблице, то их суммарный объем будет больше, чем объем данных в таблице. Кроме того, существует так называемое "**правило 20%**": если запрос на выборку возвращает более 20% записей из таблицы, то использование индекса может замедлить выборку данных. Конечно, ситуация зависит от конкретной записи и условий, наложенных на выборку, но нужно помнить, что 20%

являются пределом, когда эффективность использования индексов ставится под вопрос.

Учитывая вышесказанное, определим **основные случаи**, когда поле необходимо проиндексировать:

- если это поле очень часто используется в условиях поиска в запросах;
- когда это поле используется при связывании таблиц (JOIN), поскольку индексы также обеспечивают уникальность записей таблицы, гарантируя тем самым целостность данных;
- когда это поле используется при сортировке (ORDER BY).

В MS SQL Server различают следующие **типы индексов**:

1. В зависимости от физического размещения индекса:
   - **кластеризованные** – это индексы, которые сортируют записи в таблицах или представлениях на основе их ключевых значений (но не физически на диске (!)). Этими значениями являются поля, включенные в индекс. Промежуточный уровень такого индекса по сути отсутствует, а нижний уровень содержит данные таблицы.

Значения кластеризованного индекса всегда уникальны и поэтому в таблице может быть только один такой индекс. Кроме того, при добавлении кластеризованного индекса, доступ к данным почти всегда осуществляется быстрее, чем в случае некластеризованого индекса, поскольку в первом случае исключается дополнительный поиск записи данных.

На практике для каждой таблицы необходимо создать кластеризованный индекс, который будет первичным ключом (но это не обязательно должно быть именно поле

первичного ключа). После этого, такая таблица будет называться **кластеризованной**. Таблицы, которые не содержат кластеризованных индексов, называют **кучами**.

- **некластеризованные** – это индексы, которые задают логическое упорядочение для таблицы. Структура такого индекса имеет структуру классического сбалансированного дерева. Нижний уровень такого индекса содержит значение ключа, вместе с идентификатором записи. Для каждой таблицы можно создать до 999 некластеризованих индексов. **По умолчанию все индексы некластеризованные**.

2. В зависимости от типа значения, которое в них хранится:
   - **уникальные (UNIQUE KEY)** – индексы, которые образуются при указании характеристики UNIQUE при создании ключа. Уникальность индекса означает, что его значения не повторяются;
   - **неуникальные** – создаются по умолчанию и указывают на то, что значение их ключей могут повторяться.
3. В зависимости от количества полей, которые в них входят:
   - **простые** – индексируется только одно поле;
   - **составные** – это индексы, указывающие на несколько полей. Такой индекс может содержать от 2 до 16 полей. Максимальный общий размер значений составляющих индекса – 900 байт. Например, если таблица содержит поля Фамилия и Имя определенного клиента, для повышения производительности запросов, которые постоянно доступаються до их полного имени, следует создать индекс на эти поля.

4. В зависимости от упорядоченности данных в индексе:
   - **растущие**;
   - **спадающие**.
5. В зависимости от степени охвата данных:
   - **полнотабличные** – это индексы, которые охватывают все записи индексируемой таблицы;
   - **фильтрованные** – это некластеризованные индексы, которые охватывают только часть записей в индексируемых таблицах. Они появились в версии SQL Server 2008 и имеют преимущества в случае, если типичные запросы к таблице в результате возвращают небольшое количество записей. Например, когда в запросах участвуют поля, которые в основном содержат NULL-значения.

Такой тип индекса рекомендуется использовать в случае наличия:

- разреженных полей, которые содержат небольшое количество отличных от NULL значений;
- разреженных полей, которые по-разному заполнены для различных категорий записей;
- полей, содержащих диапазоны значений (деньги, время, дату и т.д.).

Кроме того, существуют еще такие **виды индексов**, как:

- **полнотекстовые индексы** – индексы, которые используются в полнотекстовом поиске (см. раздел "Полнотекстовое индексирование и поиск").
- **пространственные индексы** – это индексы, которые позволяют более эффективно использовать операции с пространственными данными в полях типа geometry, geography. Для их создания используется оператор

CREATE SPATIAL INDEX. Более подробно об этом типе индексов см. в разделе "Обзор пространственного индексирования" электронной документации по SQL Server 2008.

- **XML-индексы** – это индексы для полей типа данных xml. Для их создания используется оператор CREATE XML INDEX. Более подробно об этом типе индексов см. в разделе "Индексы для полей типа данных xml" электронной документации по SQL Server 2008.

Не следует забывать также о том, что индекс не является частью таблицы – это отдельный объект, который связан с таблицей и другими объектами БД. Синтаксис создания индекса имеет следующий вид:

```
CREATE [ UNIQUE ]                          /* уникальность */
   [ CLUSTERED | NONCLUSTERED ]            /* кластеризованный
или нет */
INDEX имя_индекса
ON [БД.][схема.][таблица | представление] (поле ASC |
DESC [,... n])
[ INCLUDE ( поле [ ,...n ] ) ]             /*дополнительные поля
на нижний уровень */
[ WHERE предикатное_выражение ]            /*для фильтрованного
индекса */
/* дополнительные параметры */
[ 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 = максимальная_ступень_параллелизма ]
  [, DATA_COMPRESSION = { NONE | ROW | PAGE}
```

```
[ ON PARTITIONS ( { выражение_номера_секции | диапазон }
[ , ...n ] ) ]
                ]
)]
/* где создать индекс */
[ ON { имя_схемы_секцианирования ( название_поля )
   | файловая_группа
   | default      /* в файловой группе по умолчанию */
    } ]
[ FILESTREAM_ON { имя_файловой_группы_filestream | схема_
секцианирования | "NULL" } ]
```

Параметр **INCLUDE** указывает на неключевые поля, которые добавляются на нижний уровень некластеризованого индекса. Поля типов text, ntext и image не допускаются.

Предикатное выражение в параметре **WHERE** используется для фильтрованных индексов и определяет те записи таблицы, которые должны быть проиндексированы. Стоит отметить, что фильтруемыми индексами не могут быть полнотекстовые и XML-индексы. Для уникальных индексов (UNIQUE) лишь избранные записи должны иметь уникальное значение. Кроме того, фильтровые индексы не поддерживают параметр IGNORE_DUP_KEY.

Параметр **ON** позволяет указать место сохранения индекса. Если его не указать, а таблица или представление несекционованные, SQL Server размещает индекс в той же файловой группе, где находится таблица или представление, для которых он создан.

Параметр **FILESTREAM_ON** позволяет перемещать данные FILESTREAM в другую файловую группу FILESTREAM или схему секционирования.

Расшифруем дополнительные **параметры создания индекса**:

- **PAD_INDEX** – используется, как правило, с фактором заполнения и определяет степень заполнения страниц на разных уровнях индекса (разреженность индекса). Параметр ON указывает на то, что степень заполнения страниц промежуточного уровня приравнивается к процентному отношению свободного пространства на диске, указанном в параметре FILLFACTOR. По умолчанию (OFF) страницы промежуточного уровня заполняются почти полностью – свободного места хватает не более, чем на одну запись от максимального значения размера индекса.
- **FILLFACTOR** – это коэффициент (фактор) заполнения, который определяет полноту заполнения каждой страницы нижнего уровня индекса. По умолчанию он равен 0. Значение фактора заполнения 0 или 100 говорит о том, что страницы нижнего уровня при создании заполнены полностью (на 100%).
- **SORT_IN_TEMPDB** – сохранять ли промежуточные результаты сортировки, которые образуются при создании индекса, в базе данных tempdb. По умолчанию промежуточные результаты сортировки хранятся в той же базе, что и индекс (OFF).
- **IGNORE_DUP_KEY** – включает игнорирование значения ключа, которое повторяется, для уникальных индексов. По умолчанию при повторе значений генерируется сообщение об ошибке (OFF).
- **STATISTICS_NORECOMPUTE** – указывает на необходимость сохранения статистических данных о составе ин-

декса. Значение по умолчанию OFF включает автоматическое обновление статистики.

- **DROP_EXISTING** – указывает на то, что индекс должен уничтожаться и создаваться заново, то есть на пересоздаваемость индекса. По умолчанию (OFF) при существовании одноименных индексов генерируется сообщение об ошибке, иначе (ON) существующий индекс удаляется и перестраивается.
- **ONLINE** – запрещает запросы и изменения данных в индексированных таблицах и их индексах во время операций над индексами по сети (OFF; по умолчанию). Данная функция доступна только для выпусков SQL Server Enterprise, Developer и Evaluation.
- **ALLOW_ROW_LOCKS** – разрешена или нет блокировка записей. По умолчанию блокировка записей разрешена при доступе к индексам (ON).
- **ALLOW_PAGE_LOCKS** – разрешена или нет блокировка страниц. По умолчанию блокировка страниц возможна при доступе к индексу (ON).
- **MAXDOP** – устанавливает параметр конфигурации "максимальная степень параллелизма", который задает время операции индекса. MAXDOP ограничивает число процессоров, которые принимают участие в параллельном выполнении плана. Максимальное количество процессоров – 64. Данная функция доступна только для выпуска SQL Server Enterprise.
- **DATA_COMPRESSION** – задает режим компрессии данных для указанного индекса или указанных секций. Он может принимать одно из следующих значений:

- NONE – компрессия данных;
- ROW – компрессия записей;
- PAGE – компрессия страниц.

- **ON PARTITIONS** – указывает секции, в которых применяется параметр DATA_COMPRESSION. Если опция ON PARTITIONS не указана, то параметр DATA_COMPRESSION применяется ко всем секциям секционированного индекса.

  В выражении номера секции можно указывать:
  - номер секции, например, ON PARTITIONS (2)
  - перечень секций, разделенный запятыми, например, ON PARTITIONS (1, 3, 7);
  - диапазон секций вместе с отдельными секциями, например, ON PARTITIONS (2, 4, 6 TO 8).

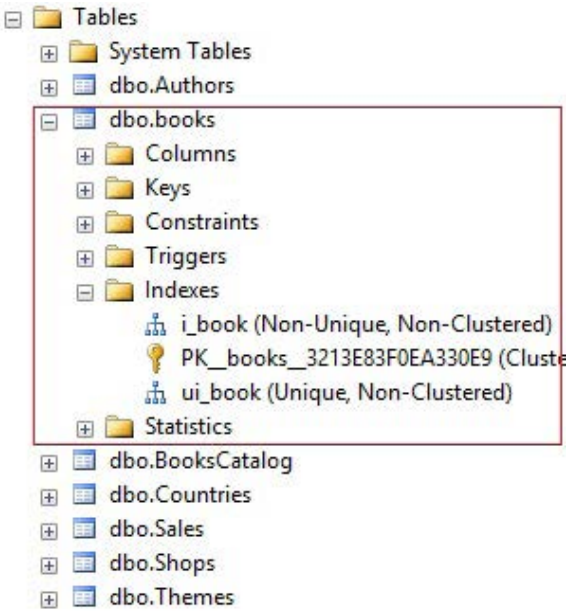
При определении диапазона, номера секций разделяются ключевым словом TO, например, ON PARTITIONS (2 TO 5).

Для примера создадим несколько индексов:

```
-- simple clustered index on a book title stored in the
 Books table
 create index i_book on Books(NameBook);

-- composite clustered index that contains the name of
 book's author
-- of thr Authors table
 create index i_author_name on Authors(FirstName, LastName);

-- simple clustered index for the DateOfSale field of the
 Sales table
 create clustered index ci_salesdate on sales(DateOfSale);
```

```
-- simple nonclustered index for the id field of the
Countries table,that can be rebuild.
-- we also define the filling factor of the leaf level of the
index
create nonclustered index i_country on Countries(id)
with (fillfactor = 80, pad_index = on, drop_existing = on);

-- simple clustered with a filler ny country name (USA)
create index i_country_USA on Countries(Country)
where country = 'USA';
```

Если индекс создался, то он будет размещен в папке **Indexes** необходимой таблицы БД:



Чтобы определить **используется ли индекс при выполнении SQL запроса** используются следующие операторы T-SQL:

- **SET SHOWPLAN_ALL ON / OFF** – показать весь план выполнения запроса;

- **SET SHOWPLAN_TEXT ON / OFF** – показать текст плана выполнения запроса.

Например:

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

Результатом будет вывод информации о том, какие данные и откуда берутся для формирования результирующего набора, которые индексы при этом используются и откуда они берутся и тому подобное.

Индекс можно временно отключить или снова включить с помощью инструкции **ALTER INDEX**:

```
-- включить индекс
 ALTER INDEX { имя_индекса | ALL } ON объект DISABLE
-- включить индекс; при этом он перестраивается
 ALTER INDEX { имя_индекса | ALL } ON объект REBUILD
```

Стоит отметить, что при отключении кластеризованного индекса, вся таблица становится недоступна.

Например:

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

Полный синтаксис оператора ALTER INDEX позволяет изменить и его первичные настройки:

```
ALTER INDEX { имя_индекса | ALL }
ON [БД.][схема.][таблица | представление]
{ REBUILD                  /* перестроить индекс (по
желанию со следующими условиями) */
  [ [PARTITION = ALL]     /* перестраиваются все секции
индекса */
    [ WITH ( добавочные_опции [ ,...n ] ) ]  /* согласно
таких акций */
  | [ PARTITION = номер_секции
/* перестраивается только одна секция со  следующими
параметрами */
      [ WITH ( параметры_перестрочйки_единичной_секции [
,...n ] ) ]
       ]
  ]
  | DISABLE        /* отключить индекс */
  | REORGANIZE     /* реорганизовать конечный уровень
индекса */
    [ PARTITION = номер_секции ]    /* для конкретной
секции */
    [ WITH ( LOB_COMPACTION = { ON | OFF } ) ]
/* все страницы с данными типа LOB (image, text, ntext,
varchar(max), xml nvarchar(max), varbinary(max))
сжимаются*/
  | SET ( параметры_индекса [ ,...n ] )             /*
параметры без перестройки или реорганизации индекса */
}
-- параметры перестройки единичной секции
{
    SORT_IN_TEMPDB = { ON | OFF }
  | MAXDOP = максимальная_ступень_паралеллизма
  | DATA_COMPRESSION = { NONE | ROW | PAGE } }
}
-- параметры индекса без перестройки или реорганизации
индекса
{
    ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS = { ON | OFF }
  | IGNORE_DUP_KEY = { ON | OFF }
  | STATISTICS_NORECOMPUTE = { ON | OFF }
```

Удаление реляционного индекса осуществляется инструкцией **DROP INDEX**.

```
DROP INDEX имя_индекса
ON [БД.][схема.][таблица | представление]
[ WITH ( добавочные_опции_индекса [ ,...n ] ) ]
[ ,...n ]
-- дополнительные опции индекса
{
    MAXDOP = максимальная_ступень_паралеллизма
  | ONLINE = { ON | OFF }
  /* куда будут перемещены после удаления строки данных
конечного уровня индекса */
  | MOVE TO {   имя_схемы_секционирования ( название_поля
)
          | файловая_группа
          | default
            }
  /* в какую папку будет перемещена таблица filestream
конечного уровня индекса */
  [ FILESTREAM_ON { имя_файловой_группы_filestream |
схема_секционирования | "NULL" } ]
}
```

Например:

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

Для получения статистических данных по индексам используются следующие инструкции T-SQL и системные хранимые процедуры:

1. Создание статистики – **CREATE STATISTICS**;
2. Обновление статистики – **sp_updatestats**;
3. Автостатистика – **AUTO_UPDATE_STATISTICS**;
4. Просмотр статистики – **sp_helpstats**.

2. Индексированные представления

Индексированные представления (indexed view) позволяют заранее рассчитать результирующий набор, который будет возвращаться представлением. Например, провести все необходимые объединения таблиц, рассчитать функции агрегирования и тому подобное. При этом отпадает необходимость вводить условие в каждый запрос. Таким образом, производительность приложений, использующих индексированные представления вместо базовых таблиц, возрастает в десятки раз. Это очень существенно, когда работа стандартного представления связана со сложной обработкой большого количества запросов, например, при ведении и расчета статистики.

Теоретически, создание таких представлений сводится к созданию представления и построения для него кластеризованного индекса. Но это не совсем так, поскольку таблицы, которые будут принимать участие в индексированных представлениях, индексы для них, а также сами представления должны соответствовать ряду критериев. Эти критерии введены для того, чтобы избежать неоднозначных расчетов. В связи с этим в индексированных представлениях разрешается использование только детерминированные функции. Более подробно об ограничении читайте в разделе "Создание индексированных представлений" электронной документации по SQL Server 2008.

Кстати, в редакцию SQL Server Enterprise и Developer встроенная интересная возможность оптимизатора запросов. Если оптимизатор определит, что для выполнения запроса эффективнее использовать индексируемые представления, а не базовую таблицу, он перепишет запрос. При этом в запросе даже не нужно указывать индексируемое представление – достаточно указать таблицу, для которой оно определено.

Проведем небольшой пример построения индексированного представления.

```
-- defining parameters for supporting of indexed views (if
they are not defined)
set NUMERIC_ROUNDABORT off;
set ANSI_PADDIND, ANSI_WARNINGS, CONCAT_NULL_YIELDS_NULL,
ARITHABORT,
QUOTED_IDENTIFIER, ANSI_NULLS on;
go
-- creating a view
create view iv_Orders
with schemabinding
as
select sh.Shop as 'Shop',
s.DateOfSale as 'Sale Date',
sum(s.price*s.quantity) as 'Cost',
COUNT_BIG(*) AS COUNT
from dbo.Sales s, dbo.Shops sh
where s.id_shop = sh.id
group by sh.Shop, s.DateOfSale
go
-- creating index for the view
create unique clustered index uci_ivorders on iv_
Orders(Shop)
go
```

После этого, в последующих запросах будет использоваться созданное нами индексируемое представление, хотя на него и нет явной ссылки.

```
select sh.Shop as 'Shop',
s.DateOfSale as 'Sale Date',
sum(s.price*s.quantity) as 'Cost'
from dbo.Sales s, dbo.Shops sh
where s.id_shop = sh.id and
      s.DateOfSale between '01/01/2006' and '01/01/2010'
group by sh.Shop, s.DateOfSale
go
select sh.Shop as 'Shop',
sum(s.price*s.quantity) as 'Cost'
from dbo.Sales s, dbo.Shops sh
where s.id_shop = sh.id and
      datepart(mm, s.DateOfSale) = 1
group by sh.Shop
```

Ряд СУБД поддерживают аналоги индексированных представлений SQL Server. Например, в СУБД Oracle это материализованные представления.

Подытоживая, можно выделить разницу между обычным представлением и индексированным. **Стандартное представление** – это SELECT запрос, на который ссылаются по имени и который сбохраняется в SQL Server. Само по себе оно не содержит данных.

Индексированное представления – это представления с кластеризованным индексом. Так SQL Server материализует и сохраняет на диске результаты запроса, заданного в представлении.

3. Полнотекстовое индексирование и поиск

Для повышения эффективности выборки при поиске текстовых данных вместо операторов LIKE или оператора равенства (=), а также поиска в неструктурированных данных, SQL Server и ряд других СУБД рекомендуют использовать полнотекстовый поиск. В SQL Server полнотекстовый поиск обеспечивает отдельный компонент **Full-Text Search**.

Для того, чтобы организовать полнотекстовый поиск **необходимо**:

1. Создать полнотекстовый каталог.
2. Определить таблицы и поля, для которых необходимо использовать полнотекстовый поиск.
3. Создать полнотекстовые индексы на указанные поля.
4. Задать метод заполнения полнотекстового индекса, который будет обеспечивать согласованность полнотекстового индекса с данными. То есть метод заполнения отвечает за то, чтобы все изменения в отобранных полях таблиц были отражены в соответствующих полнотекстовых индексах.
5. Создать запрос, который будет с помощью специальных функций осуществлять полнотекстовый поиск.

Полнотекстовое индексирование имеет свою собственную службу – **Средство полнотекстового поиска диспетчера**

конфигурации SQL Server (SQL Full-Text Filter Daemon Launcher, MSSQLFDLaucher) – предназначенную для управления полнотекстовыми каталогами.

Итак, процесс организации полнотекстового индексирования начинается с создания полнотекстового каталога, поскольку именно он обеспечивает поддержку полнотекстовых индексов в SQL Server. Фактически, **полнотекстовый каталог** – это набор полнотекстовых индексов базы данных. **Начиная с версии SQL Server 2008, в базе данных полнотекстовый каталог является виртуальным объектом и не входит в файловую группу.** В предыдущих версиях, его рекомендовалось размещать в отдельной файловой группе базы данных.

Для создания полнотекстового каталога используется инструкция T-SQL **CREATE FULLTEXT CATALOG:**

```
CREATE FULLTEXT CATALOG название_каталога
     [ ON FILEGROUP файловая_группа ] -- для
совместимости с предыдущими версиями
     [ IN PATH 'корневой_путь' ]   -- для совместимости
с предыдущими версиями, в следующей версии планируется ее
исключение
     [ WITH ACCENT_SENSITIVITY = { ON | OFF } ]
     [ AS DEFAULT ]                  -- данный каталог
будет каталогом по умолчанию
     [ AUTHORIZATION имя_собственника ]
```

Параметр **WITH** указывает на то, будет ли каталог учитывать диакритические знаки для полнотекстового индексирования. Важно знать, что при изменении данного параметра, индекс автоматически перестраивается, в отличие от предыдущих версий SQL Server, где необходимо было вручную перестроить все полнотекстовые индексы каталога.

После создания полнотекстового каталога необходимо создать полнотекстовые индексы. Одна таблица или индексированное представление может иметь только один полнотекстовый индекс, хотя в ней может быть проиндексирован несколько полей. Для реализации полнотекстового индексирования таблица должна иметь одно уникальное поле с не NULL значением, а также иметь поля типа char, varchar, varchar (max), nchar, nvarchar, text, ntext, image, xml, varbinary или varbinary (max).

Чтобы создать полнотекстовые индексы для двоичных типов данных (image, varbinary, varbinary (max)) используются **специальные обработчики протокола и фильтры**. Они позволяют получать текст из файлов Word, Excel, PowerPoint, PDF и т.д., хранящиеся в базе данных. Для работы этих сервисов необходимо в таблицу добавить поле, которое содержит информацию о типе документа, который хранится в поле двоичного типа. После этого фильтр загружает двоичный поток, который хранится в поле, удаляет всю информацию о форматировании и возвращает текст документа в средство разбиения на слова. После этого механизм полнотекстового поиска:

1) вычисляет лексемы, которые являются сжатыми формами самих слов;

2) строит все лексемы в поле в специальную сжатую структуру файла, который используется для дальнейшего поиска. В связи с этим, размеры самого полнотекстового индекса ограничены доступными ресурсами памяти компьютера.

Начиная с версии SQL Server 2008 полнотекстовые индексы встроены в компонент Database Engine, а не размещены в файловой системе, как в предыдущих версиях SQL Server.

Для создания полнотекстовых индексов используется инструкция **CREATE FULLTEXT INDEX**.

```
CREATE FULLTEXT INDEX ON таблица
      [ ( название_поля [ TYPE COLUMN поле_типа ] [
LANGUAGE язык_данных ] [ ,...n] ) ]
    KEY INDEX название_уникального_индекса
        [ ON полнотекстовый_каталог ]
        [ WITH [ ( ] опции [ ,...n] [ ) ] ]
-- опции
  { CHANGE_TRACKING [ = ] { MANUAL | AUTO | OFF [, NO
POPULATION ] }
    | STOPLIST [ = ] { OFF | SYSTEM | список_стоп-слов }
  }
```

Параметр **TYPE COLUMN** содержит название поля таблицы, в котором хранится тип документа (расширение файла, которое указывается пользователем (.DOC, .PDF, .XLS т.д.)) для документов varbinary, varbinary (max) или image. Если индекс создается не для двоичных данных, тогда параметр TYPE COLUMN не указывается. Это поле называется полем типа и должно иметь тип char, nchar, varchar или nvarchar.

Параметр **LANGUAGE** указывает на язык данных, который хранится в индексируемом поле. Данный параметр

рекомендуется использовать, если в индексируемых таблицах содержатся поля на разных языках. Например, если данные поля переведены на несколько языков. Получить список доступных кодов для языков сохраняется в системном представлении **sys.fulltext_languages**.

Опция **CHANGE_TRACKING** указывает на то, как будут отслеживаться изменения индексированных данных (при update, insert, delete):

- MANUAL – синхронизацию данных следует осуществлять вручную, путем вызова инструкций ALTER FULLTEXT INDEX ... START UPDATE POPULATION (заполнение вручную) или с помощью установки расписания синхронизации изменений в SQL Server Agent.
- AUTO (по умолчанию) – изменения распространяются автоматически (с помощью фонового процесса) в ходе модификации данных в базовой таблице.
- OFF [, NO POPULATION] – изменения данных не отслеживаются. Аргумент NO POPULATION указывает на то, что полнотекстовый индекс будет создан без заполнения. Дальнейшее заполнение индекса осуществляется с помощью оператора ALTER FULLTEXT INDEX.

Стоит отметить, что **процесс заполнения полнотекстового индекса очень ресурсоемкий**, поэтому создание такого индекса должно быть осуществлено, когда активность базы данных невелика. В связи с этим, в большинстве случаев полнотекстовые индексы создают с опцией OFF и NO POPULATION, а когда активность базы данных минимальная создают задачи для заполнения всех полнотекстовых

индексов. После этого, для индексируемых полей, которые редко меняются, заполнение индексов выставляется в режим AUTO.

Параметр **STOPLIST** связывает полнотекстовый список стоп-слов с индексом (до версии SQL Server 2008 – пропускаемых слов). Индекс не заполняется значениями, которые являются частью данного списка. Если данный параметр не указан, тогда с индексом связывается системный полнотекстовый список стоп-слов.

- OFF – указывает на то, что с полнотекстовым индексом не связано ни одно значение из списка стоп-слов. То есть список не используется.
- SYSTEM – для полнотекстового индекса будет использоваться системный список стоп-слов.
- Также можно указать конкретное имя списка стоп-слов, который будет связываться с полнотекстовым индексом.

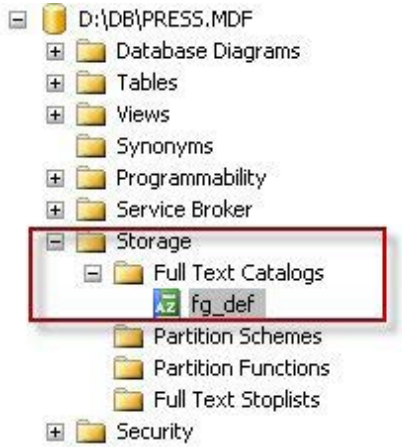
Посмотреть список стоп-слов можно из системного представления **sys.fulltext_stopwords**. Управлять списком стоп-слов можно с помощью операторов **CREATE / ALTER / DROP FULLTEXT STOPLIST**, но только при уровне совместимости 100.

Для примера возьмем таблицу, которая содержит информацию об авторах. Чтобы организовать полнотекстовый поиск по полям FirstName и LastName необходимо создать для них полнотекстовые индексы.

Итак, сначала создаем полнотекстовый каталог, в котором будут определяться поля для индексирования:

```
-- создаем полнотекстовый каталог по умолчанию
 create fulltext catalog fg_def as default;
```

Новосозданный полнотекстовый каталог в базе данных размещается в папке **Storage-> Full Text Catalogs**.



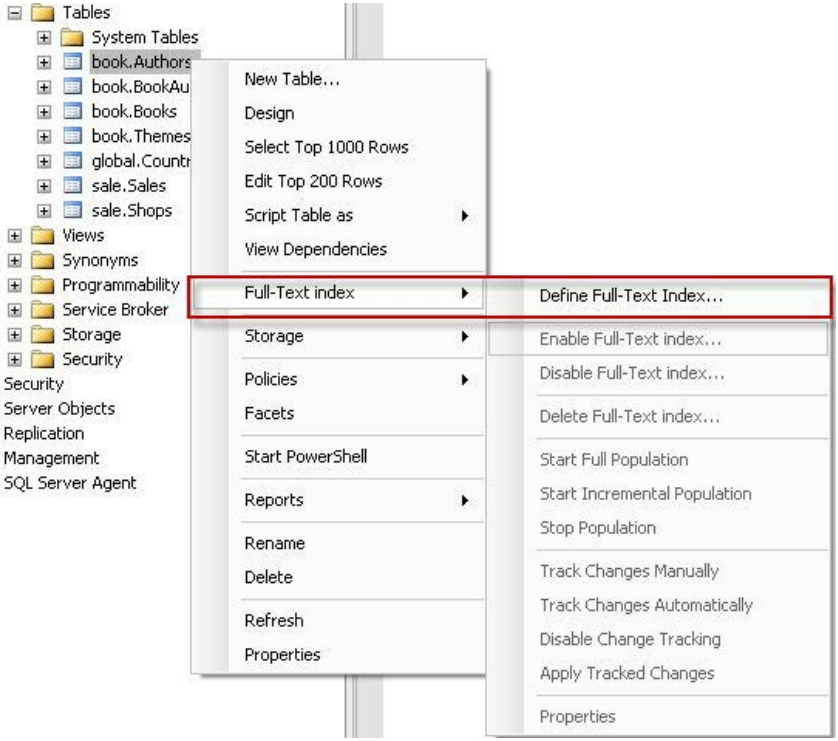
Следующим шагом является создание полнотекстового индекса на необходимые поля. Но, если в таблице отсутствует уникальный индекс (обычно им является первичный ключ), тогда следует его создать.

```
-- если в таблице нет уникального индекса, тогда создадим
его
create unique index ui_AuthorId on book.Authors(ID_AUTHOR);
go
-- создаем полнотекстовый индекс для полей FirstName
и LastName с использованием полнотекстового каталога fg_
def и системного списка стоп-слов
create fulltext index on book.Authors(FirstName,
LastName)
    key index ui_AuthorId
    on fg_def;     -- параметр необязательный
```

Можно также указать код языка, который будет являться языком данных в полях. Код (LCID) английского языка – 1033, русского – 1049, украинского – 1058:

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

Чтобы создать полнотекстовый индекс средствами Management Studio, следует выбрать пункт контекстного меню **Full-Text index-> Define Full-Text Index** ... таблицы, в которой предполагается его наличие. После этого запустится визард создания полнотекстового индекса, в котором необходимо указать уникальный индекс таблицы, полнотекстовый каталог, в котором будет размещаться индекс, необходимые поля и тому подобное.



Для изменения установленных параметров полнотекстового индекса следует воспользоваться инструкцией **ALTER FULLTEXT INDEX**:

```
ALTER FULLTEXT INDEX ON таблица
{
     -- активизировать или отключить полнотекстовый
индекс
     ENABLE  | DISABLE
     -- отражать изменения данных поля в полнотекстовом
индексе, который ему соответствует
   | SET CHANGE_TRACKING { MANUAL | AUTO | OFF }
     -- добавить поле для индекса
   | ADD ( название_поля [ TYPE COLUMN поле_типа ] [
LANGUAGE язык_данных ] [ ,...n] )
     [ WITH NO POPULATION ]
     -- удалить поле из индекса   | DROP (название_поля
[,...n] ) [ WITH NO POPULATION ]
     -- начать заполнение полнотекстового индекса
   | START { FULL | INCREMENTAL | UPDATE } POPULATION
     -- остановить, приостановить или возобновить процесс
заполнения индекса
   | {STOP | PAUSE | RESUME } POPULATION
     -- как завязывать полнотекстовый список стоп-слов
с индексом
   | SET STOPLIST { OFF| SYSTEM | список_стоп-слов } [
WITH NO POPULATION ]
}
```

Опция **WITH NO POPULATION** указывает на то, что полнотекстовый индекс не будет заполнен после добавления (ADD) или удаление (DROP) поля полнотекстового индекса, а также после операции SET STOPLIST. Индекс будет заполняться только после выполнения команды START ... POPULATION.

Параметр **START ... POPULATION** указывает SQL Server, что следует начать заполнение полнотекстового индекса одним из следующих способов:

- FULL – каждая строка таблицы должна принимать участие в полнотекстовом индексировании, даже если ее записи уже были проиндексированы.
- INCREMENTAL – указывает, что для полнотекстового индексирования используются только те записи, которые были изменены после последнего индексирования. Данный аргумент можно использовать только, если в таблице существует поле типа timestamp. Если такого поля нет, тогда выполняется заполнение FULL.
- UPDATE – обрабатываются все операции INSERT, DELETE и UPDATE после последнего обновления. При этом, таблица должна поддерживать отслеживание изменений (включена опция), но индекс фонового обновления или автоматическое отслеживание изменений выключены.

Параметр **{STOP | PAUSE | RESUME} POPULATION** позволяет остановить, приостановить или возобновить приостановленный процесс заполнения индекса. Но в их работе существуют небольшие ограничения:

- STOP – несмотря на то, что процесс заполнения индекса останавливается, автоматическое отслеживание изменений или фоновое обновление индекса продолжается. Чтобы их остановить, следует использовать команду SET CHANGE_TRACKING OFF.
- PAUSE и RESUME могут использоваться только для полных (FULL) заполнений. Для других типов

заполнений они несущественны, поскольку сканирование возобновляется с момента последней проверки.

Например:

```
-- активизировать полнотекстовый индекс в таблице book.
Authors
alter fulltext index on book.Authors enable
go
-- заполнить полнотекстовый индекс в таблице book.Authors
alter fulltext index on book.Authors start update
population
```

Для удаления полнотекстового индекса используется инструкция **DROP FULLTEXT INDEX**:

```
-- синтаксис
DROP FULLTEXT INDEX ON таблица
-- например
drop fulltext index on book.Authors
```

Теперь можно создавать и выполнять полнотекстовые запросы. Для полнотекстового поиска используются свои функции, которые значительно эффективнее чем оператор LIKE и (=). В SQL Server это предикаты:

- **FREETEXT** – используется для поиска нечеткого совпадения. Она определяет список различных форм шукаемого слова, фактически осуществляя поиск по его сути.

Синтаксис:

```
FREETEXT ( { название_поля | ( список_полей ) | * },
           'текст для поиска'
           [ , LANGUAGE язык ]
         )
```

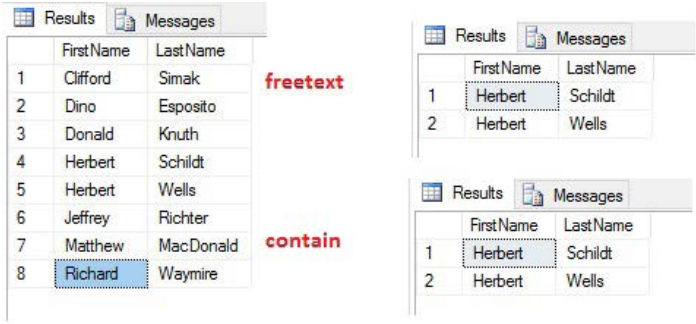
- **CONTAINS** – ищет точные совпадения слов и префиксов слов. Сокращенный синтаксис данного предиката аналогичный FREETEXT.

Примечание! *Тип второго параметра вышеописанных предикатов* **nvarchar**, *поэтому для корректной работы запросов необходимо, чтобы и поля для поиска имели аналогичный тип. Кроме того, чтобы поиск фраз или слов на русском, украинском и других европейских языках работал корректно, не забывайте при создании полнотекстового индекса также указывать параметр LANGUAGE.*

Например, необходимо найти всех авторов, в фамилии которых встречается Herbert:

```
select FirstName, LastName
from Authors
where freetext(FirstName,N'Herbert');
select FirstName, LastName
from Authors
where contains(FirstName,N'Herbert');
```

Результат:



Чтобы функция contains вернула результат, аналогичный freetext, следует воспользоваться **префиксами**. Для

идентификации префикса необходимо использовать двойные кавычки. При их отсутствии запрос будет осуществлять поиск слова.

```
select FirstName, LastName
from Authors
where contains(FirstName,N'H*');
```

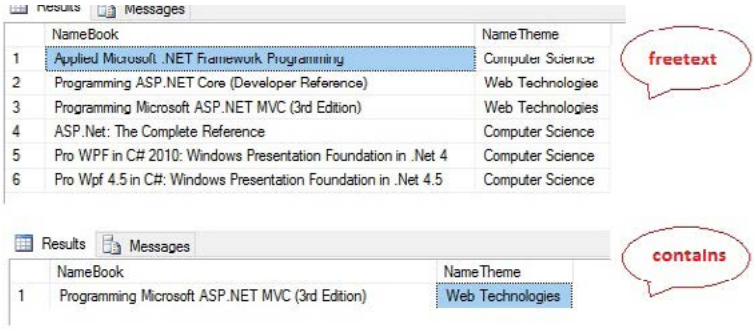
Теперь будут найдены все фамилии авторов, начинающихся с 'H'.

Обе функции позволяют использовать в шаблонах поиска фразы. Например, среди книг небходимо найти такие, которые содержат фразу "NET MVC":

```
-- this query will fin all the books that contain "NET" and
"MVC"
select b.NameBook, t.NameTheme
from Books b, Themes t
where b.id_theme = t.id and
      contains(b.NameBook,'NET MVC')
-- with the use of 'and' (&), 'or' (|), and 'not' (!)
select b.NameBook, t.NameTheme
from Books b, Themes t
where b.id_theme = t.id and
      contains(b.NameBook,'NET' or 'MVC')
-- this quety will find all the books that contain "NET" and
"MVC"
select b.NameBook, t.NameTheme
from Books b, Themes t
where b.id_theme = t.id and
      freetext(b.NameBook,'NET MVC');
```

Оператор **AND NOT** возвращает true, если первое условие истинно, а вторая ошибочно.

Результаты:



Примечание! *Поиск символов в слове или фразе регистронезависимый. Такие слова как "a", "and", "is", "the" (в зависимости от языка) и т.д., а также знаки препинания при поиске пропускаются, поскольку они считаются ненужными и включены в список стоп-слов.*

Кстати, в предикате contains префиксы используются и для фраз. В таком случае, каждое слово из фразы считается отдельным префиксом. Например, при поиске фразы "local wine *" будут отобраны строки с текстом "local winery", "locally wined and dined" и другие.

Функция contains может принимать различные параметры, что позволяет настроить ее использование. Два варианта использования, кроме "классического" поиска слов, мы уже рассмотрели – это использование префиксов и фраз в шаблонах. Рассмотрим ее возможности шире:

1. Для поиска вариантов фраз используют функцию **FORMSOF**, первый параметр которой указывает на принцип генерации словоформ:

- **INFLECTIONAL** – поиск по основе слова, например, слово "погода" выдаст соответствие для "погода", "погоды", "согласовывать" и др;
- **THESAURUS** – поиск синонимов, определенных в XML-файле тезауруса. Например, слово "металл" может иметь синоним "золото", "алюминий", "железо" и другие. По умолчанию файл тезауруса пустой и размещается по пути место_ установления _SqlServer \ Microsoft SQL Server \ MSSQL10_50.MSSQLSERVER \ MSSQL \ FTDATA \.

```
create fulltext index on Books (NameBook language 1049)
key index ui_book_id
go
select b.NameBook, t.NameTheme
from Books b, Themes t
where b.id_theme = t.id and
      contains(b.NameBook,'formsof(inflectional, Java)')
```

Результат:

	NameBook	NameTheme
1	Java: A Beginner Guide	Computer Science
2	Java: The Complete Reference	Computer Science

2. Поиск слов или фраз по близости их месторасположения. Для этого используется ключевое слово NEAR (~). Расстояние между заданными словами измеряется также в словах, с учетом принадлежности к предложению или абзаца. Предикат CONTAINS редко используется с этим ключевым словом, поскольку

ранг результирующих слов не может быть определен напрямую.

```
select b.NameBook, t.NameTheme
from Books b, Themes t
where b.id_theme = t.id and
      contains(b.NameBook,'Complete NEAR Reference')
-- или
select b.NameBook, t.NameTheme
from Books b, Themes t
where b.id_theme = t.id and
      contains(b.NameBook,'Complete ~ Reference')
```

Результатом этих запросов будут названия книг, которые содержат слово "Complete" рядом со словом "Reference".

Results Messages

	NameBook	NameTheme
1	Java: The Complete Reference	Computer Science
2	C++: The Complete Reference, 4th Edition	Computer Science
3	ASP.Net: The Complete Reference	Computer Science

Results Messages

	NameBook	NameTheme
1	Java: The Complete Reference	Computer Science
2	C++: The Complete Reference, 4th Edition	Computer Science
3	ASP.Net: The Complete Reference	Computer Science

3. Функция **ISABOUT** позволяет задавать относительный вес определенным критериям поиска (в диапазоне от 0.0 до 1.0).

```
select b.NameBook, t.NameTheme
from Books b, Themes t
where b.id_theme = t.id and
      contains(b.NameBook,'isabout(Windows weight(.8), C#
weight (.3))')
```

В данном запросе нас больше интересует вхождение слова "Windows", поэтому ему мы установили больший весовой коэффициент.

Стоит отметить, что ключевое слово WEIGHT не влияет на запросы с CONTAINS, но влияет на значение RANK, которое возвращает CONTAINSTABLE.

Функции **CONTAINS** и **FREETEXT** имеют свои аналоги для работы с таблицами – это **CONTAINSTABLE** и **FREETEXTTABLE**. Отличие последних заключается лишь в том, что они возвращают набор записей, который должен быть соединен с другой таблицей на основе значения ключа. Вместе с результатами поиска они возвращают **дополнительные поля:**

- **RANK** – определяет ранг соответствия каждой записи шаблона: чем выше ранг, тем точнее совпадение (число от 0 до 1000);
- **KEY** – ключевое поле входной таблицы, соответствующее найденным записям.

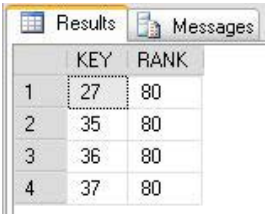
Синтаксис их следующий:

```
CONTAINSTABLE (  таблица, { поле | (список_полей) | * },
                  'текст для поиска'
                  [ , LANGUAGE язык ]
                  [ , первые_n_по_рангу ]
                 )
FREETEXTTABLE (  таблица, { поле | (список_полей) | * },
                  'текст для поиска'
                  [ , LANGUAGE язык ]
                  [ , первые_n_по_рангу ]
               )
```

Например, напишите запрос, который ищет сборники в таблице book.Books. Поиск осуществляется по всем полям.

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

Результат:



Results | Messages

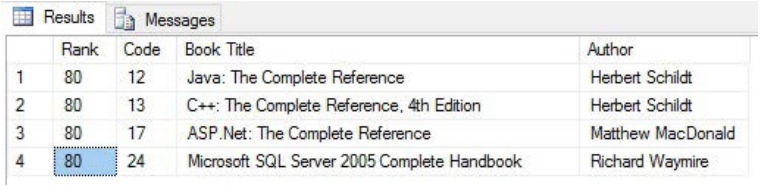
	KEY	RANK
1	27	80
2	35	80
3	36	80
4	37	80

Вышерассмотренный пример можно переписать следующим образом:

```
select cb.Rank as 'Rank', b.id as 'Code',
     b.NameBook as 'Book Title',
     a.FirstName +' '+a.LastName as 'Author'
from containstable(Books, *, N'Complete') as cb,
     Books as b, Authors as a
where cb.[KEY] = b.id and
     b.id_author = a.id
```

Как видно из примера, поле KEY должно быть указано в квадратных скобках, поскольку оно является ключевым словом T-SQL.

Результат:



Results | Messages

	Rank	Code	Book Title	Author
1	80	12	Java: The Complete Reference	Herbert Schildt
2	80	13	C++: The Complete Reference, 4th Edition	Herbert Schildt
3	80	17	ASP.Net: The Complete Reference	Matthew MacDonald
4	80	24	Microsoft SQL Server 2005 Complete Handbook	Richard Waymire

Примечание! *Иногда, SQL Server при выполнении запросов с полнотекстовым поиском может выдавать ошибку "SQL Server encountered error 0x80070218 while communicating with full-text filter daemon host (FDHost) proces". Для того, чтобы избежать данной ошибки необходимо:*

1. Зайти в Диспетчер конфигурации SQL Server.
2. Выбрать свойства службы SQL Full-text Filter Deamon Launcher.
3. Для входа (вкладка Log On) выбрать Network Service.

4. Перезапустить службу и экземпляр сервера.

4. Триггеры

Триггер – это специализированная процедура, которая автоматически вызывается SQL Server при возникновении событий в базе данных. В SQL Server с версии SQL Server 2005 поддерживаются два **типа триггеров**:

1. **DML-триггеры** – выполняются при возникновении DML (Data Manipulation Language – язык манипулирования данными) событий: добавлении (INSERT), удалении (DELETE) или обновлении (UPDATE) записей таблиц или представлений. Такие триггеры всегда привязаны к определенной таблице или представлению и могут перехватывать данные только ее / его.
2. **DDL-триггеры** – выполняются при возникновении DDL (Data Definition Language – Язык Определения Данных) событий: создание (CREATE), изменение (ALTER) или удаление (DROP) объектов. Отдельную подгруппу образуют триггеры входа, которые срабатывают на событие LOGON, которое возникает при установлении сеанса пользователя. DDL-триггеры используются для администрирования базы данных, например, для аудита и управления доступом к объекту.

Триггеры могут быть созданы как с помощью инструкций Transact-SQL, так и с помощью методов сборок, созданных в среде CLR платформы .NET Framework и переданы экземпляру SQL Server. Все триггеры не имеют параметров и не выполняются явно. При возникновении

события, к которому привязан триггер, SQL Server автоматически его запускает.

Итак, начнем наше знакомство, а начнем мы с **DML-триггеров**. Синтаксис создания такого триггера имеет следующий вид:

```
CREATE TRIGGER [схема.] имя_триггера
ON { таблица | представление }      -- для кого создается
триггер
[ WITH ENCRYPTION                   -- зашифровать
исходный код триггера
      [, EXECUTE AS условие ]         -- контекст
выполнения
]
{ FOR | AFTER                 -- после изменения данных
      | INSTEAD OF }          -- вместо SQL  команды, для
которой они объявлены
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] } -- на
какое действие с данными
[ WITH APPEND ]           -- добавить триггер существующего
типа
[ NOT FOR REPLICATION ] -- триггер не выполняется, если
в ходе репликации будет изменена
 - таблица, на которую он указывает
AS
{ тело_триггера | EXTERNAL NAME сборка.класс.метод }
```

После указания имени триггера в инструкции ON необходимо указать название таблицы или представления, для которого создается триггер.

С помощью инструкции **WITH** можно включить шифрование кода триггера (ENCRIPTION) или указать контекст исполнения (EXECUTE AS), в котором будет работать триггер. Параметр EXECUTE AS в основном используется для проверки привилегий (прав доступа) на объекты базы данных, на которые ссылается триггер.

Как видно из синтаксиса, DML-триггеры можно запускать в **двух режимах**: AFTER и INSTEAD OF. Триггера BEFORE, который присутствует во многих СУБД, в MS SQL Server не существует.

Триггер INSTEAD OF может порождать событие (команду), для которой он объявлен. Причем это событие будет выполняться так, будто триггера INSTEAD OF не существовало. Например, если вы хотите проверить определенное условие для выполнения команды INSERT, вы можете объявить триггер INSTEAD OF INSERT. Триггер INSTEAD OF будет выполнять проверку, а затем выполнять команду INSERT для таблицы. Оператор INSERT будет выполняться привычным образом, порождая рекурсивных вызовов триггера INSTEAD OF.

Следует также помнить, что **по умолчанию все триггеры активные** и выполняются после проведенного действия, то есть имеют установленный параметр **AFTER**.

Замечания по построению триггеров:

1. Если при объявлении триггера, указать единственное ключевое слово FOR, то аргумент AFTER используется по умолчанию.
2. Нельзя создавать триггеры INSTEAD OF для модифицированных представлений.
3. Триггеры AFTER используются только для таблиц.
4. Параметр WITH ENCRYPTION не может быть указан для триггеров CLR.
5. Аргумент WITH APPEND установлен **только для совместимости** с предыдущими версиями (на уровне 65). При

этом он может использоваться только при указании параметра FOR без INSTEAD OF и не может указываться для триггеров CLR. **В связи с тем, что в следующей версии SQL Server аргумент WITH APPEND планируется полностью исключить, его рекомендуется избегать.**

Существует также ряд **правил**, которых следует придерживаться при **создании триггеров**:

- НЕЛЬЗЯ создавать триггеры для временных таблиц, но они могут обращаться к ним;
- НЕЛЬЗЯ создавать, изменять, удалять резервные копии или восстанавливать из резервной копии базы данных;
- триггеры могут использоваться для обеспечения целостности данных, но их не следует использовать вместо объявления целостности путем установления ограничения FOREIGN KEY;
- триггеры не могут возвращать результирующие наборы, поэтому при использовании оператора select в теле триггера следует быть очень внимательным. При этом во многих случаях, вместе с оператором select используется директива IF EXISTS;
- поддерживаются рекурсивные AFTER триггеры, при установке параметра базы данных RECURSIVE_TRIGGERS в значение ON;
- можно создавать вложенные триггеры (поддерживается до 32 уровней вложенности), которые фактически являются неявной рекурсией. Для их поддержки необходимо установить параметр NESTED TRIGGERS;

- в теле DML-триггера НЕЛЬЗЯ использовать операторы:
  - все операции CREATE / ALTER / DROP;
  - TRUNCATE TABLE;
  - RECONFIGURE;
  - LOAD DATABASE или TRANSACTION;
  - GRANT и REVOKE;
  - SELECT INTO;
  - UPDATE STATISTICS.

Триггеры также являются объектами БД и в SQL Management Studio они размещаются в папке "**Triggers**" таблицы, на которую установлен тот или иной триггер. К примеру:

В MS SQL Server в пределах DML-триггеров используют справочные таблицы **INSERTED** и **DELETED**, которые имеют ту же структуру, что и базовые таблицы триггера. Они размещаются в оперативной памяти, так как являются логическими таблицами. Работают они следующим образом:

- **когда в базовую таблицу добавляются новые данные** (новая запись), то эти же данные добавляются сначала

в базовую таблицу, а затем в таблицу inserted. Их наличие в таблице inserted избавляет от необходимости создавать специальные переменные для доступа к этой информации.

- **когда строка удаляется из таблицы**, то она записывается в таблицу deleted, а затем удаляется из базовой таблицы.
- **когда строка обновляется**, то старое значение записи записывается в таблицу deleted и удаляется из базовой таблицы, затем обновленная запись записывается в базовую таблицу, а дальше в таблицу inserted.

То есть мы можем использовать таблицу DELETED для получения значений записей, которые удаляются из таблицы, а таблицу INSERTED для получения новых записей перед их фактической вставкой. В других серверных СУБД такие задачи выполняют схожие механизмы, например, в InterBase / Firebird и Oracle – это контекстные переменные OLD и NEW.

Для лучшего понимания работы с триггерами напишем **несколько примеров**.

1. Классический триггер, который будет срабатывать при каждой вставке данных в таблицу Authors и возвращает сообщение о количестве измененных строк. В этом, нелегком на первый взгляд, деле, нам поможет глобальная переменная @@ **rowcount**, содержащая данные о количестве модифицированных строк, в результате работы триггера.

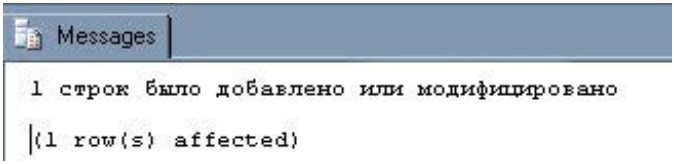
```
create trigger addAuthor
on book.Authors
for insert, update
```

```
as
    raiserror('%d строк было добавлено или модифицировано
', 0, 1, @@rowcount)
    return
```

Добавляем нового автора в нашу БД:

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

Результат:



1 строк было добавлено или модифицировано

2. Триггер, который при добавлении новых данных о книге, дата издательства которой больше месяца, будет выбрасывать сообщение об ошибке.

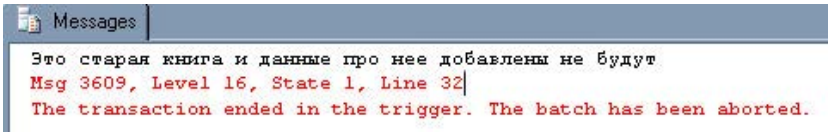
```
create trigger CheckDateTrigger
on book.Books
for insert
as
begin
    declare @InsDate smalldatetime
-- получаем дату издательства книги, которая добавляется
    select @InsDate = DateOfPublish
    from inserted
-- проверяем, сколько прошло дней со дня издания
    if (@InsDate <= getdate()-30)
    begin
        raiserror('Это старая книга и данные о ней
добавлены не будут ',0,1)
        rollback transaction
    end
```

```
    else
        PRINT(' Данные добавлены успешно ')
End
```

Добавляем новую книгу:

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

Результат:



3. Триггер, запрещающий удаления книги, если она больше всего продается, то есть является лидером продаж.

```
create trigger CheckBookDelete
on book.Books
for delete
as
begin
    declare @NameBook varchar(25), @BestBook varchar(25)
-- Получаем название удаляемой книги
    select @NameBook = deleted.NameBook
    from deleted
    declare @Zvit table (nameB varchar(25), quantity int)
-- Получаем информацию о названиях книг и их количестве
продаж (популярность)
    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
-- находим самую продаваемую книгу
     select @BestBook = z.nameB
     from @Zvit z
     where z.quantity = (select max(quantity)
                         from @Zvit)
-- проверяем, совпали ли названия
     if (@BestBook = @NameBook)
     begin
         raiserror("Вы не можете удалить эту книгу ',0,1)
         rollback transaction
     end
     else
     begin
         print ("Книга удалена успешно ')
     end
 end
```

4. Триггер, который при удалении книги тематики "Computer Science" выдает сообщение об ошибке.

```
create trigger NotDeleteComputerScience
on Books
instead of delete
as
begin
declare @ThemeId int
-- get the identifier of 'Computer Science' theme
select @ThemeId = id
from Themes
where
      NameTheme = 'Computer Science'
-- check whether the identifier of removing book matches the
@ThemeId
if exists (select * from deleted where id_theme = @ThemeId)
      raiserror ('This book cannot be deleted!',0,1)
end;
```

Как уже было сказано выше, для DDL-операций, таких как CREATE, ALTER, DROP, GRANT, DENY, REVOKE и UPDATE STATISTICS, используются **DDL-триггеры**. Кроме контроля и мониторинга данных операций, DDL-триггеры позволяют ограничивать их выполнение, даже если пользователь имеет соответствующие права. Например, можно запретить пользователям определенных групп изменять и удалять таблицы. Для этого достаточно создать DDL-триггер для событий DROP TABLE и ALTER TABLE, который будет делать откат этих операций и выдавать соответствующее сообщение об ошибке.

Синтаксис создания DDL-триггера имеет следующий вид:

```
CREATE TRIGGER имя_триггера
ON { ALL SERVER | DATABASE }        -- область действия
триггера
[ WITH ENCRYPTION [, EXECUTE AS условие ] ]
{ FOR | AFTER } { имя_события | группа_событий } [ ,...n
]
AS
 { тело_триггера | EXTERNAL NAME сборка.класс.метод }
```

Как видно из синтаксиса, после инструкции ON необходимо указать **область действия триггера**:

- **ALL SERVER** – текущий сервер. Триггер будет срабатывать при возникновении определенных событий в любом месте в рамках текущего сервера. Например, CREATE_DATABASE, ALTER_LOGIN, ALTER_INSTANCE и тому подобное.
- **DATABASE** – текущая база данных. Триггер будет срабатывать при возникновении определенных событий на уровне текущей базы данных или низших уров-

нях. Например, CREATE_TABLE, DROP_DEFAULT, ALTER_USER и тому подобное.

Для удобства администрирования можно использовать группы событий, например, группа событий DDL_TABLE_EVENTS включает в себя события, которые касаются таблицы: CREATE_TABLE, ALTER_TABLE и DROP_TABLE. Группы событий имеют иерархическую структуру. Например, группа событий DDL_TABLE_VIEW_EVENTS охватывает все инструкции T-SQL в группах DDL_TABLE_EVENTS, DDL_VIEW_EVENTS, DDL_INDEX_EVENTS и DDL_STATISTICS_EVENTS.

Детальный список названий событий приведен в разделе "DDL-события" электронной документации по SQL Server 2008, а групп событий в разделе "Группы DDL-событий".

Проиллюстрируем код вышеприведенного примера использования DDL-триггера, то есть на запрет изменения и удаления таблиц:

```
create trigger NotAlterDropTable
on DATABASE
for DROP_TABLE, ALTER_TABLE
as
begin
  print 'Модификация и удаление таблиц запрещены.
Обратитесь к администратору.'
  rollback
end
```

Стоит отметить, что DDL-триггеры не вызываются на события, которые влияют на временные таблицы и хранимые процедуры. DDL-триггеры также не ограничены областью схемы и после создания триггеры уровня базы

данных находятся в папке "**Programmability / Database Triggers**", а триггеры уровня сервера – в папке "Triggers" папки "**Server Objects**" (объекты сервера).

Отдельную подгруппу DDL-триггеров составляют **триггеры входа (logon trigger)**:

```
CREATE TRIGGER имя_триггера
    ON ALL SERVER
    [ WITH ENCRYPTION [, EXECUTE AS условие ] ]
{ FOR | AFTER } LOGON
AS
 { тело_триггера | EXTERNAL NAME сборка.класс.метод }
```

Такие триггеры выполняются в ответ на событие **LOGON**, которое вызывается при установке пользовательского сеанса с экземпляром сервера. Триггеры входа срабатывают после завершения этапа аутентификации при входе, но перед тем, как сеанс пользователя реально устанавливается. Итак, все сообщения об ошибке функции raiserror или инструкции PRINT, которые вызываются в триггере, перенаправляются в журнал ошибок SQL Server. Если же

пользователь ввел неверный логин или пароль, то триггер входа не срабатывает.

Отметим, что в триггерах входа не поддерживаются распределеные транзакции. Триггеры входа могут создаваться из любой базы данных, но принадлежат они базе данных **master**.

В следующем примере запретим пользователю с логином 'vasja_pupkin' подключение к 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 'Такой логин запрещен на сервере.
Обратитесь к администратору.'
        rollback
    end
end
```

Триггеры можно изменять, временно отключить или удалить.

Модифицировать триггер можно с помощью инструкции **ALTER TRIGGER**:

```
-- для DML-триггера
ALTER TRIGGER [схема.] имя_триггера
    ON { таблица | представление }
     [ WITH ENCRYPTION [, EXECUTE AS условие ] ]
{ FOR | AFTER | INSTEAD OF }
```

```
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ NOT FOR REPLICATION ]
AS
 { тело_триггера | EXTERNAL NAME сборка.класс.метод }
-- для DDL-триггера
ALTER TRIGGER имя_триггера
    ON { ALL SERVER | DATABASE }
     [ WITH ENCRYPTION [, EXECUTE AS условие ] ]
{ FOR | AFTER } { имя_события | группа_событий } [ ,...n
]
AS
{ тело_триггера | EXTERNAL NAME сборка.класс.метод }
-- для триггера входа
ALTER TRIGGER имя_триггера
    ON ALL SERVER
     [ WITH ENCRYPTION [, EXECUTE AS условие ] ]
{ FOR | AFTER } LOGON
AS
{ тело_триггера | EXTERNAL NAME  сборка.класс.метод }
```

Включить или выключить триггер можно с помощью инструкции ALTER TABLE или с помощью следующих инструкций:

```
-- включить триггер
ENABLE TRIGGER { [ схема. ] имя_триггера [ ,...n ] | ALL
}
ON { таблица | представление | DATABASE | ALL SERVER }
-- выключить триггер
DISABLE TRIGGER { [ схема. ] имя_триггера [ ,...n ] | ALL
}
ON { таблица | представление | DATABASE | ALL SERVER }
```

Опция **ALL** указывает на то, что все триггеры в области действия значение параметра ON будут включены или выключены.

Опции **DATABASE** и **ALL SERVER** указывают на то, что инструкция касается DDL-триггера уровня базы данных или сервера (включая триггер входа) соответственно.

Удаление триггера осуществляется оператором **DROP TRIGGER**:

```
-- для DML-триггера
 DROP TRIGGER [схема.] имя_триггера [ ,...n ] [ ; ]
-- для DDL-триггера
 DROP TRIGGER имя_триггера [ ,...n ]
 ON { DATABASE | ALL SERVER }
-- для триггера входа
 DROP TRIGGER имя_триггера [ ,...n ]
 ON ALL SERVER
```

Имена всех триггеров хранятся в системной таблице **sysobjects** и системном представлении **sys.objects**. Метаданные DDL- и DML-триггеров уровня базы данных можно просмотреть с помощью нового представления **sys.triggers**. Если поле parent_class_desc данного представления имеет значение "DATABASE", тогда это DDL-триггер и его областью действия является база данных. Тело триггера можно получить по представлению **sys.sql_modules** (связав его с sys.triggers по полю object_id), а код создания – из системной таблицы **syscomments**. Метаданные CLR-триггеров доступны по представлению **sys.assembly_modules**, которое также следует связать с sys.triggers по полю.

Информация про DDL-триггеры уровня сервера, включая триггеры входа, сохраняются в системном представлении **sys.server_triggers**. Тело триггера уровня сервера можно получить по представлению **sys.server_sql_modules**,

а метаданные CLR-триггера серверного уровня – по представлению **sys.server_assembly_modules**.

В завершение отметим, что SQL Server предоставляет информацию о событиях, которые он отлавливает, в виде XML. Они доступны через новую встроенную функцию **EVENTDATA** (), которая возвращает XML-данные. Эта возможность позволяет применять DDL-триггеры для аудита DDL-операций в базе данных.

Для этого, например, можно создать таблицу аудита с полем, которое содержит XML-данные. Затем создаем DDL-триггер с параметром EXECUTE AS SELF для DDL-событий или групп событий, которые вас интересуют. Тело такого DDL-триггера может просто выполнять вставку (INSERT) XML-данных, которые возвращаются EVENTDATA () в таблицу аудита.

5. Домашнее задание

1. В своей базе данных найдите все таблицы, которые не имеют кластеризованных индексов. Для каждой из них добавьте кластеризованный индекс или измените первичный ключ на кластеризованный.
2. Найдите медленно выполняемые запросы. Создайте некластеризованные индексы, для повышения скорости выполнения этих запросов.
3. Возьмите одно из представлений, созданных на предыдущем занятии (в качестве домашнего задания), и превратите его в индексируемое.
4. В базе данных создайте полнотекстовый каталог.
5. Написать запрос с полнотекстовым поиском, который будет выводить названия магазинов, которые содержат слово "книга". Вместе с названием магазина необходимо вывести страну их расположения.
6. Выполните полнотекстовый поиск всех книг, в названиях которых содержатся слова "Java", "Guide" или "Web", при этом для каждого слова задается свой вес.
7. Напишите запрос, который в результате работы выведет книги, с указанием их авторов и рангами соответствия условию в следующем порядке важности тематик: "Computer Science", "Science Fiction", "Web Technologies". Для этого воспользуйтесь возможностью установки весовых коэффициентов для частей шаблона.

8. Выведите все названия магазинов, которые продавали книги после 01/01/2010. Укажите также объем продаж книг, месторасположение магазина и ранг соответствия условию поиска. Условие: в названиях книг встречаются следующие совпадения: "Windows" и "NET" или "Java" и "guide".

   Примечание! Для запросов с полнотекстовым поиском создайте необходимые полнотекстовые индексы.

   Написать следующие триггеры:

1. Триггер, который при продаже книги автоматически изменяет количество книг в таблице Books. (Примечание Добавить в таблице Books необходимое поле количества имеющихся книг QuantityBooks).
2. Триггер на проверку, чтобы количество продаж книг не превысила имеющуюся.
3. Триггер, который при удалении книги, копирует данные о ней в отдельную таблицу "DeletedBooks".
4. Триггер, который следит, чтобы цена продажи книги не была меньше основной цены книги из таблицы Books.
5. Триггер, запрещающий добавления новой книги, для которой не указана дата выпуска и выбрасывает соответствующее сообщение об ошибке.
6. Триггер или набор триггеров, которые запрещают удаление объектов любой базы данных на сервере (таблиц, значений по умолчанию и т.д.).
7. Добавьте к базе данных триггер, который выполняет аудит изменений данных в таблице Books.

Урок №7

Программирование и администрирование СУБД MS SQL Server