МГТУ ИМ. Н.Э. БАУМАНА

# Базы Данных

## Семинар 6

Описание семинара: Функции и процедуры

# Функции T-SQL

SQL Server содержит набор встроенных функций и предоставляет возможность создавать пользовательские функции (User Defined Function – UDF). Различают детерминированные и недетерминированные функции.

- Функция является **детерминированной**, если при одном и том же заданном входном значении она всегда возвращает один и тот же результат. Например, встроенная функция DATEADD является детерминированной; она возвращает новое значение даты, добавляя интервал к указанной части заданной даты.
- Функция является **недетерминированной**, если она может возвращать различные значения при одном и том же заданном входном значении. Например, встроенная функция GETDATE является недетерминированной; при каждом вызове она возвращает различные значения даты и времени компьютера, на котором запущен экземпляр SQL Server.

Все функции конфигурации, курсора, метаданных, безопасности и системные статистические – недетерминированные. Список этих функций приводится в справочнике. Функции, вызывающие недетерминированные функции и расширенные хранимые процедуры, также считаются недетерминированными.

Пользовательские функции в зависимости от типа данных возвращаемых ими значений могут быть скалярными и табличными. Табличные пользовательские функции бывают двух типов: подставляемые и многооператорные.

**Скалярные пользовательские функции**

Скалярные пользовательские функции обычно используются в списке столбцов инструкции SELECT и в предложении WHERE. Табличные пользовательские функции обычно используются в предложении FROM, и их можно соединять с другими таблицами и представлениями.

```
CREATE FUNCTION [ имя-схемы. ] имя-функции ( [ список-объявлений-параметров ] )
RETURNS скалярный-тип-данных
[ WITH список-опций-функций ]
[ AS ]
BEGIN
    тело-функции
    RETURN скалярное-выражение
END [ ; ]
```

Для создания скалярной функции используется инструкция CREATE FUNCTION, имеющая следующий синтаксис:

```
@имя-параметра [ AS ] [ имя-схемы. ] тип-данных [ = значение-по-умолчанию ] [ READONLY ]
```

- Тип данных параметра – любой тип данных, включая определяемые пользователем типы данных CLR и определяемые пользователем табличные типы, за исключением скалярного типа timestamp и нескалярных типов cursor и table.
- Значение по умолчанию
- Необязательное ключевое слово READONLY указывает, что параметр не может быть обновлен или изменен при определении функции. Если тип параметра является определяемым пользователем табличным типом, то должно быть указано ключевое слово READONLY.

где

- Список объявлений параметров является необязательным, но наличие скобок обязательно.
- Объявление параметра в списке объявлений параметров имеет вид:

- Возвращаемое значение может быть любого скалярного типа данных, включая определяемые пользователем типы данных CLR, за исключением типа данных timestamp.
- В качестве опций функции используются:
  - ENCRIPTION — SQL Server шифрует определение функции.
  - SCHEMABINDING - привязывает функцию к схеме базовой таблицы или таблиц. Если аргумент SCHEMABINDING указан, нельзя изменить базовую таблицу или таблицы таким способом, который может повлиять на определение функции.
    RETURNS NULL ON NULL INPUT или| CALLED ON NULL INPUT - ...

    - Предложение EXECUTE AS - указывает контекст безопасности, в котором может быть выполнена функция. Например, EXECUTE AS CALLER указывает, что функция будет выполнена в контексте пользователя, который ее вызывает. Также могут быть указаны параметры SELF, OWNER и имя-пользователя.
- Операторы BEGIN и END, которыми ограничивается тело функции, являются обязательными.
- Тело функции представляет собой ряд инструкций T-SQL, которые в совокупности вычисляют скалярное выражение.

Синтаксис вызова скалярных функций схож с синтаксисом, используемым для встроенных функций T-SQL:

имя-схемы.имя-функции ( [ список-параметров ] )
где
- Имя схемы (имя владельца) для скалярной функции является обязательным.
- Нельзя использовать синтаксис с именованными параметрами (@имя-параметра = значение).
- Нельзя не указывать (опускать) параметры, но можно применять ключевое слово DEFAULT для указания значения по умолчанию.

Для скалярной функции можно использовать инструкцию EXECUTE:

```
EXECUTE @возвращаемое-значение = имя-функции ( [ список-параметров ] )
```
где
- Не нужно указывать имя схемы (имя владельца).
- Можно использовать именованные параметры (@имя-параметра = значение).
- Если используются именованные параметры, они не обязательно должны следовать в том порядке, в котором указаны в объявлении функции, но необходимо указать все параметры; нельзя опускать ссылку на параметр для использования значения по умолчанию.

Примеры создания и вызова скалярных функций

```
CREATE FUNCTION dbo.AveragePrice() RETURNS smallmoney
WITH SCHEMABINDING AS
BEGIN
    RETURN (SELECT AVG(Price) FROM dbo.R)
END;

CREATE FUNCTION dbo.PriceDifference(@Price smallmoney) RETURNS smallmoney AS
BEGIN
    RETURN @Price - dbo.AveragePrice()
END;

— Вызов функции
SELECT Pname, Price, dbo.AveragePrice() AS Average, dbo.PriceDifference(Price) AS Difference FROM R
WHERE City='Смоленск'
```

**Подставляемая табличная функция**
Тело подставляемой табличной функции фактически состоит из единственной инструкции SELECT. Синтаксис создания подставляемой табличной функции выглядит так:

```
CREATE FUNCTION [ имя-схемы. ] имя-функции ( [ список-объявлений-параметров ] )
RETURNS TABLE
[ WITH список-опций-функций ]
[ AS ]
RETURN [ ( ] выражение-выборки [ ) ] END [ ; ]
```

Пример создания и вызова подставляемой табличной функции

```
CREATE FUNCTION dbo.FullSPJ()
RETURNS TABLE
AS
      RETURN (SELECT S.Sname, P.Pname, J.Jname, SPJ.Qty
                  FROM S INNER JOIN SPJ ON S.Sno=SPJ.Sno
                              INNER JOIN P ON P.Pno=SPJ.Pno
                              INNER JOIN J ON J.Jno=SPJ.Jno)

— Вызов функции
SELECT *
FROM dbo.FullSPJ()
```

**Многооператорная табличная функция**
Подобно скалярным функциям, в многооператорной табличной функции команды T-SQL располагаются внутри блока BEGIN-END. Поскольку блок может содержать несколько инструкций SELECT, в предложении RETURNS необходимо явно определить таблицу, которая будет возвращаться. Поскольку оператор RETURN в многооператорной табличной функции всегда возвращает таблицу, заданную в предложении RETURNS, он должен выполняться без аргументов. Синтаксис создания многооператорной табличной функции выглядит так:

```
CREATE FUNCTION [ имя-схемы. ] имя-функции ( [ список-объявлений-параметров ] )
RETURNS @имя-возвращаемой-переменной TABLE определение-таблицы
[ WITH список-опций-функций ]
[ AS ]
BEGIN
RETURN END [ ; ]
```

Пример создания и вызова многооператорной табличной функции

```
CREATE FUNCTION dbo.fnGetReports ( @EmployeeID AS int )
RETURNS @Reports TABLE ( EmployeeID int NOT NULL, ReportsToID int NULL )
AS BEGIN
      DECLARE @Employee AS int
      INSERT INTO @Reports
        SELECT EmployeeID, ReportsTo FROM Employees
        WHERE EmployeeID = @EmployeeID
SELECT @Employee = MIN(EmployeeID) FROM Employees
WHERE ReportsTo = @EmployeeID
WHILE @Employee IS NOT NULL
  BEGIN
    INSERT INTO @Reports
       SELECT *
END RETURN
END
```

# Хранимые процедуры T-SQL

Хранимая процедура – именованный объект базы данных, представляющий собой набор SQL-инструкций, который компилируется один раз и хранится на сервере. Хранимые процедуры похожи на обыкновенные процедуры языков высокого уровня, у них могут быть входные и выходные параметры и локальные переменные. В хранимых процедурах могут выполняться операторы DDL, DML, TCL, FCL. Процедуры можно создавать для постоянного использования, для временного использования в одном сеансе (локальная временная процедура), для временного использования во всех сеансах (глобальная временная

процедура). Хранимые процедуры могут выполняться автоматически при запуске экземпляра SQL Server.

**Создание хранимых процедур**
Для создания хранимой процедуры используется инструкция CREATE PROCEDURE, имеющая следующий синтаксис:

```
CREATE PROCEDURE [ имя-схемы. ] имя-процедуры [ список-объявлений-параметров ]
[ WITH список-опций-процедуры ]
[ FOR REPLICATION ]
AS
тело-процедуры
[ ;]
```

где
- Если имя схемы не указано при создании процедуры, то автоматически назначается схема по умолчанию для пользователя, который создает процедуру.
- Список объявлений параметров является необязательным.
- Объявление параметра в списке объявлений параметров имеет вид:

```
@имя-параметра тип-данных [ VARYING ] [ = значение-по-умолчанию ] [ OUT | OUTPUT ] [READONLY]
```

- Параметрами процедуры могут быть любые типы данных, за исключением table.
- Для создания параметров, возвращающих табличное значение, можно использовать определяемый пользователем табличный тип. Возвращающие табличное значение параметры могут быть только входными и должны сопровождаться ключевым словом READONLY.
- Тип данных cursor может быть использован только в качестве выходного параметра.
- VARYING применяется только к аргументам типа cursor.
- OUT или OUTPUT показывает, что параметр процедуры является выходным. Параметры типов text, ntext и image не могут быть выходными.
- READONLY указывает, что параметр не может быть обновлен или изменен в тексте процедуры.
- Опциями функции могут быть:
  - ENCRYPTION - SQL Server шифрует определение процедуры.
  - RECOMPILE - SQL Server перекомпилирует процедуру при каждом ее выполнении.
  - Предложение EXECUTE AS - определяет контекст безопасности, в котором должна быть выполнена процедура.
- Тело процедуры - одна или несколько инструкций T-SQL. Инструкции можно заключить в необязательные ключевые слова BEGIN и END.
- FOR REPLICATION указывает, что процедура создается для репликации.
- Тело процедуры может содержать оператор RETURN, возвращающий целочисленное значение вызывающей процедуре или приложению.

Примечания.
- Локальную временную процедуру можно создать, указав один символ номера (#) перед именем процедуры.
- Глобальную временную процедуру можно создать, указав два символа номера (##) перед именем процедуры.
- Временные процедуры создаются в базе данных tempdb.
- Рекомендуется начинать текст процедуры с инструкции SET NOCOUNT ON (она должна следовать сразу за ключевым словом AS). В этом случае отключаются сообщения, отправляемые SQL Server клиенту после выполнения любых инструкций SELECT, INSERT, UPDATE, MERGE и DELETE.

**Выполнение хранимых процедур**
При выполнении процедуры в первый раз она компилируется, при этом определяется оптимальный план получения данных. При последующих вызовах процедуры может быть повторно использован уже созданный план, если он еще находится в кэше планов компонента Database Engine.

Одна процедура может вызывать другую. Уровень вложенности увеличивается на 1, когда начинается выполнение вызванной процедуры, и уменьшается на 1, когда вызванная процедура завершается. Уровень вложенности процедур может достигать 32. Текущий уровень вложенности процедур можно получить при помощи функции @@NESTLEVEL.
Чтобы выполнить процедуру, надо использовать инструкцию EXECUTE. Также можно выполнить процедуру без использования ключевого слова EXECUTE, если процедура является первой инструкцией в пакете.
При выполнении процедуры (в пакете или внутри хранимой процедуры или функции) настоятельно рекомендуется уточнять имя хранимой процедуры указанием, по крайней мере, имени схемы.

**Пример процедуры с входными и выходными параметрами**

```
CREATE PROCEDURE dbo.Factorial @ValIn bigint, @ValOut bigint output
AS
BEGIN
      IF @ValIn > 20 BEGIN
            PRINT N'Входной параметр должен быть <= 20'
            RETURN -99
      END
      DECLARE @WorkValIn bigint, @WorkValOut bigint
      IF @ValIn != 1
      BEGIN
            SET @WorkValIn = @ValIn - 1
            PRINT @@NESTLEVEL
            EXEC dbo.Factorial @WorkValIn, @WorkValOut OUTPUT
            SET @ValOut = @WorkValOut * @ValIn
       END
       ELSE
             SET @ValOut = 1
 END

— Вызов процедуры
DECLARE @FactIn int, @FactOut int
SET @FactIn = 8
EXEC dbo.Factorial @FactIn, @FactOut OUTPUT

PRINT N'Факториал ' + CONVERT(varchar(3),@FactIn) +
          N' равен ' + CONVERT(varchar(20),@FactOut)
```

**Изменение хранимых процедур**

Если нужно изменить инструкции или параметры хранимой процедуры, можно или удалить (DROP PROCEDURE) и создать ее заново (CREATE PROCEDURE), или изменить ее за один шаг (ALTER PROCEDURE). При удалении и повторном создании хранимой процедуры все разрешения, связанные с ней, будут утеряны при восстановлении. При изменении хранимой процедуры ее определение или определение ее параметров меняются, но разрешения, связанные с ней, остаются и все зависящие от нее процедуры или триггеры не затрагиваются.

# Базы Данных

## Семинар 7

Описание семинара: Триггеры и курсоры

# Триггеры T-SQL Триггеры DDL

**Триггер DDL** - это хранимая процедура особого типа, которая выполняет одну или несколько инструкций T-SQL в ответ на событие из области действия сервера или базы данных. Например, триггер DDL может активироваться, если выполняется такая инструкция, как ALTER SERVER CONFIGURATION, или если происходит удаление таблицы с использованием команды DROP TABLE.
Все события (кроме ALTER_SERVER_CONFIGURATION) разделены на две группы:
- события уровня базы данных (DDL_DATABASE_LEVEL_EVENTS)
- события уровня сервера(DDL_SERVER_LEVEL_EVENTS)

Следует обратить внимание на иерархическую природу групп событий. В отличие от триггеров DML, триггеры DDL не ограничены областью схемы. Поэтому для запроса метаданных о триггерах DDL нельзя воспользоваться такими функциями как OBJECT_ID, OBJECT_NAME, OBJECTPROPERTY и OBJECTPROPERTYEX. Используйте вместо них представления каталога.

**Пример триггера DDL**

```
CREATE TRIGGER safety
ON DATABASE
FOR DROP_TABLE, ALTER_TABLE AS
    PRINT 'You must disable Trigger "safety" to drop or alter tables!'
    ROLLBACK;
```

**Триггеры DML**
Для поддержания согласованности и точности данных используются декларативные и процедурные методы.
Триггеры представляют собой особый вид хранимых процедур, привязанных к таблицам и представлениям. Они позволяют реализовать в базе данных сложные процедурные методы поддержания целостности данных. События при модификации данных вызывают автоматическое срабатывание триггеров. Как и в случае хранимых процедур, глубина вложенности триггеров достигает 32 уровней, также возможно рекурсивное срабатывание триггеров.
Прежде чем реализовывать триггер, следует выяснить, нельзя ли получить аналогичные результаты с использованием ограничений или правил. Для уникальной идентификации строк табличных данных используют целостность сущностей (ограничения primary key и unique). Доменная целостность служит для определения значений по умолчанию (определения default) и ограничения диапазона значений, разрешенных для ввода в данное поле (ограничения check и ссылочные ограничения). Ссылочная целостность используется для реализации логических связей между таблицами (ограничения foreign key и check). Если значение обязательного поля не задано в операторе INSERT, то оно определяется с помощью определения default. Лучше применять ограничения, чем триггеры и правила.
Триггеры применяются в следующих случаях:
- если использование методов декларативной целостности данных не отвечает функциональным потребностям приложения. Например, для изменения числового значения в таблице при удалении записи из этой же таблицы следует создать триггер;
- если необходимо каскадное изменение через связанные таблицы в базе данных. Чтобы обновить или удалить данные в столбцах с ограничением foreign key, вместо пользовательского триггера следует применять ограничения каскадной ссылочной целостности;
- если база данных денормализована и требуется способ автоматизированного обновления избыточных данных в нескольких таблицах;
- если необходимо сверить значение в одной таблице с неидентичным значением в другой таблице;
- если требуется вывод пользовательских сообщений и сложная обработка ошибок.

**События, вызывающие срабатывание триггеров (типы триггеров)**
Автоматическое срабатывание триггера вызывают три события: INSERT, UPDATE и DELETE, которые происходят в таблице или представлении. Триггеры нельзя запустить

вручную. В синтаксисе триггеров перед фрагментом программы, уникально определяющим выполняемую триггером задачу, всегда определено одно или несколько таких событий. Один триггер разрешается запрограммировать для реакции на несколько событий, поэтому несложно создать процедуру, которая одновременно является триггером на обновление и добавление. Порядок этих событий в определении триггера является несущественным.

**Классы триггеров**

В SQL Server существуют два класса триггеров:

- INSTEAD OF. Триггеры этого класса выполняются в обход действий, вызывавших их срабатывание, заменяя эти действия. Например, обновление таблицы, в которой есть триггер INSTEAD OF, вызовет срабатывание этого триггера. В результате вместо оператора обновления выполняется код триггера. Это позволяет размешать в триггере сложные операторы обработки, которые дополняют действия оператора, модифицирующего таблицу.
- AFTER/BEFORE. Триггеры этого класса исполняются после действия, вызвавшего срабатывание триггера. Они считаются классом триггеров по умолчанию. Между триггерами этих двух классов существует ряд важных отличий, показанных в таблице.

| Характеристика | INSTEAD OF-триггер | AFTER/BEFORE-триггер |
|---|---|---|
| Объект, к которому можно привязать триггер | Таблица или представление. Триггеры, привязанные к представлению, расширяют список типов обновления, которые может поддерживать представление | Таблица. AFTER-триггеры срабатывают при модификации данных в таблице в ответ на модификацию представления |
| Допустимое число триггеров | В таблице или представлении допускается не больше одного триггера в расчете на одно действие. Можно определять представления для других представлений, каждое<br><br>со своим собственным INSTEAD OF-триггером | К таблице можно привязать несколько AFTER- триггеров |
| Порядок исполнения | Поскольку в таблице или представлении допускается не больше одного такого триггера в расчете на одно событие, порядок не имеет смысла | Можно определять триггеры, срабатывающие первым и последним. Для этого служит системная хранимая процедура sp_settriggerorder. Порядок<br><br>срабатывания других триггеров, привязанных к таблице, случаен |

К таблице разрешено привязывать триггеры обоих классов. Если в таблице определены ограничения и триггеры обоих классов, то первым из них срабатывает триггер INSTEAD OF, затем обрабатываются ограничения и последним срабатывает AFTER-триггер. При нарушении ограничения выполняется откат действий INSTEAD OF-триггера. Если нарушаются ограничения или происходят какие-либо другие события, не позволяющие модифицировать таблицу, AFTER-триггер не исполняется.

**Создание триггеров DML**

```
CREATE TRIGGER имя_триггера
ON имя_таблицы_или_представления
[ WITH ENCRYPTION ]
класс_триггера тип(ы)_триггера
[ WITH APPEND ]
[ NOT FOR REPLICATION ]
AS sql_инструкции
```

В квадратных скобках указаны опции триггера, которые в данной теме не рассматриваются.

Пояснения к инструкции CREATE TRIGGER:

- Триггеры не допускают указания имени базы данных в виде префикса имени объекта. Поэтому перед созданием триггера необходимо выбрать нужную базу данных с помощью конструкции USE имя_базы_данных и ключевого слова GO. Ключевое слово GO требуется, поскольку оператор CREATE TRIGGER должен быть первым в пакете. Право на создание триггеров по умолчанию принадлежит владельцу таблицы.
- Триггер необходимо привязать к таблице или представлению. Любой триггер можно привязать только к одной таблице или представлению. Чтобы привязать к другой таблице триггер, выполняющий ту же самую задачу, следует создать новый триггер с другим именем, но с той же самой функциональностью и привязать его к другой таблице. AFTER-триггеры (этот класс задан по умолчанию) разрешено привязывать только к таблицам, а триггеры класса INSTEAD OF – как к таблицам, так и к представлениям.
- При создании триггера следует задать тип события, вызывающего его срабатывание: INSERT, UPDATE и DELETE. Один и тот же триггер может сработать на одно, два или все три события. Если необходимо, чтобы он срабатывал на все события, то после конструкций FOR, AFTER или INSTEAD OF следует поместить все три ключевых слова: INSERT, UPDATE и DELETE в любом порядке.
- Конструкция AS и следующие за ней команды языка T-SQL определяют задачу, которую будет выполнять триггер.

**Пример простого триггера AFTER на событие UPDATE**

```
CREATE TRIGGER AfterApdateSPJ
ON SPJ
AFTER UPDATE
AS
BEGIN
RAISERROR(N'Произошло обновление в таблице поставок',1,1) END
```

# Курсоры T-SQL

Операции в реляционной базе данных выполняются над множеством строк. Набор строк, возвращаемый инструкцией SELECT, содержит все строки, которые удовлетворяют условиям, указанным в предложении WHERE. Такой полный набор строк, возвращаемых инструкцией, называется результирующим набором. Приложения, особенно интерактивные, не всегда эффективно работают с результирующим набором как с единым целым. Им нужен механизм, позволяющий обрабатывать одну строку или небольшое их число за один раз. Курсоры предоставляют такой механизм.
Microsoft SQL Server поддерживает три способа реализации курсоров:

В SQL Server поддерживаются четыре типа серверных курсоров:

- **Статические курсоры** (STATIC). Создается временная копия данных для использования курсором. Все запросы к курсору обращаются к указанной временной таблице в базе данных tempdb, поэтому изменения базовых таблиц не влияют на данные, возвращаемые выборками для данного курсора, а сам курсор не позволяет производить изменения.
- **Динамические курсоры** (DYNAMIC). Отображают все изменения данных, сделанные в строках результирующего набора при просмотре этого курсора. Значения данных, порядок, а также членство строк в каждой выборке могут меняться. Параметр выборки ABSOLUTE динамическими курсорами не поддерживается.
- **Курсоры, управляемые набором ключей** (KEYSET). Членство или порядок строк в курсоре не изменяются после его открытия. Набор ключей, однозначно определяющих строки, встроен в таблицу в базе данных tempdb с именем keyset.
- **Быстрые последовательные курсоры** (FAST_FORWARD). Параметр FAST_FORWARD указывает курсор FORWARD_ONLY, READ_ONLY, для которого включена оптимизация производительности. Параметр FAST_FORWARD не может указываться вместе с параметрами SCROLL или FOR_UPDATE.

Статическими курсорами обнаруживаются лишь некоторые изменения или не обнаруживаются вовсе, но при этом в процессе прокрутки такие курсоры потребляют

сравнительно мало ресурсов. Динамические курсоры обнаруживают все изменения, но потребляют больше ресурсов при прокрутке. Управляемые набором ключей курсоры имеют промежуточные свойства, обнаруживая большинство изменений, но потребляя меньше ресурсов, чем динамические курсоры.

По области видимости имени курсора различают:
- **Локальные курсоры** (LOCAL). Область курсора локальна по отношению к пакету, хранимой процедуре или триггеру, в которых этот курсор был создан. Курсор неявно освобождается после завершения выполнения пакета, хранимой процедуры или триггера, за исключением случая, когда курсор был передан параметру OUTPUT. В этом случае курсор освобождается при освобождении всех ссылающихся на него переменных или при выходе из области видимости.
- **Глобальные курсоры** (GLOBAL). Область курсора является глобальной по отношению к соединению. Имя курсора может использоваться любой хранимой процедурой или пакетом, которые выполняются соединением. Курсор неявно освобождается только в случае разрыва соединения. Глобальность курсора позволяет создавать его в рамках одной хранимой процедуры, а вызывать его из совершенно другой процедуры, причем передавать его в эту процедуру необязательно.

По способу перемещения по курсору различают:
- **Последовательные курсоры** (FORWARD_ONLY). Курсор может просматриваться только от первой строки к последней. Поддерживается только параметр выборки FETCH NEXT. Если параметр FORWARD_ONLY указан без ключевых слов STATIC, KEYSET или DYNAMIC, то курсор работает как DYNAMIC. Если не указан ни один из параметров FORWARD_ONLY или SCROLL, а также не указано ни одно из ключевых слов STATIC, KEYSET или DYNAMIC, то по умолчанию задается параметр FORWARD_ONLY. Курсоры STATIC, KEYSET и DYNAMIC имеют значение по умолчанию SCROLL.
- **Курсоры прокрутки** (SCROLL). Перемещение осуществляется по группе записей как вперед, так и назад. В этом случае доступны все параметры выборки (FIRST, LAST, PRIOR, NEXT, RELATIVE, ABSOLUTE). Параметр SCROLL не может указываться вместе с параметром для FAST_FORWARD.

По способу распараллеливания курсоров различают:
- **READ_ONLY**. Содержимое курсора можно только считывать.
- **SCROLL_LOCKS**. При редактировании данной записи вами никто другой вносить в нее изменения не может. Такую блокировку прокрутки иногда еще называют «пессимистической» блокировкой. При блокировке прокрутки важным фактором является продолжительность действия блокировки. Если курсор не входит в состав некоторой транзакции, то блокировка распространяется только на текущую запись в курсоре. В противном случае все зависит от того, какой используется уровень изоляции транзакций.
- **OPTIMISTIC**. Означает отсутствие каких бы то ни было блокировок. «Оптимистическая» блокировка предполагает, что даже во время выполнения вами редактирования данных, другие пользователи смогут к ним обращаться. Если кто-то все-таки попытается выполнить модификацию данных одновременно с вами, генерируется ошибка 16394. В этом случае вам придется повторно выполнить доставку данных из курсора или же осуществить полный откат транзакции, а затем попытаться выполнить ее еще раз.

Для выявления ситуаций с преобразованием типа курсора используется опция TYPE_WARNING, которая указывает, что клиенту посылается предупреждающее сообщение при неявном преобразовании типа курсора из запрошенного типа в другой тип.
По умолчанию курсор, разрешающий изменение данных, позволяет выполнять модификацию любых столбцов в своем составе. Опция FOR UPDATE указывает столбцы курсора, которые можно изменять. Все остальные столбцы становятся доступными только для чтения. Если опция FOR UPDATE используется без списка столбцов, то обновление возможно для всех столбцов, за исключением случая, когда был указан параметр параллелизма READ_ONLY.

На содержательном уровне синтаксис инструкции DECLARE CURSOR будет иметь вид:

```
DECLARE имя-курсора CURSOR
[ область-видимости-имени-курсора ]
[ возможность-перемещения-по-курсору ]
[ типы-курсоров ]
[ опции-распараллеливания-курсоров ]
[ выявление-ситуаций-с-преобразованием-типа-курсора ]
FOR инструкция_select
[ опция-FOR-UPDATE ]
```

Формальный синтаксис инструкции DECLARE CURSOR имеет вид:

```
DECLARE имя-курсора CURSOR
[ LOCAL | GLOBAL ]
[ FORWARD_ONLY | SCROLL ]
[ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
[ TYPE_WARNING ]
FOR инструкция_select
[ FOR UPDATE [ OF список-имен_столбцов ] ]
```

**Использование переменной типа cursor**
Microsoft SQL Server поддерживает переменные с типом данных CURSOR. Курсор может быть связан с переменной типа CURSOR двумя способами, например:

- ```
  DECLARE @MyVariable CURSOR
  DECLARE MyCursor CURSOR
  FOR SELECT LastName FROM AdventureWorks.Person.Contact
  SET @MyVariable = MyCursor;
  ```

- ```
  DECLARE @MyVariable CURSOR
  SET @MyVariable = CURSOR SCROLL KEYSET
  FOR SELECT LastName FROM AdventureWorks.Person.Contact;
  ```

После связи курсора с переменной типа CURSOR эта переменная может использоваться в инструкциях курсора языка T- SQL вместо имени курсора. Кроме того, выходным параметрам хранимой процедуры можно назначить тип данных CURSOR и связать их с курсором. Это позволяет управлять локальными курсорами из хранимых процедур.

**Перемещение внутри курсора (прокрутка курсора)**
Основой всех операций прокрутки курсора является ключевое слово FETCH. Оно может использоваться для перемещения по курсору в обоих направлениях, в том числе и для перехода к заданной позиции. В качестве аргументов оператора FETCH могут выступать:

- NEXT – возвращает строку результата сразу же за текущей строкой и перемещает указатель текущей строки на возвращенную строку. Если инструкция FETCH NEXT выполняет первую выборку в отношении курсора, она возвращает первую строку в результирующем наборе. NEXT является параметром по умолчанию выборки из курсора.
- PRIOR – возвращает строку результата, находящуюся непосредственно перед текущей строкой и перемещает указатель текущей строки на возвращенную строку. Если инструкция FETCH PRIOR выполняет первую выборку из курсора, не возвращается никакая строка и положение курсора остается перед первой строкой.
- FIRST – возвращает первую строку в курсоре и делает ее текущей.
- LAST – возвращает последнюю строку в курсоре, и делает ее текущей.
- ABSOLUTE { n | @nvar}. Если n или @nvar имеют положительное значение, возвращает строку, стоящую дальше на n строк от передней границы курсора, и делает возвращенную строку новой текущей строкой. Если n или @nvar имеют отрицательное значение, возвращает строку, отстоящую на n строк от задней границы курсора, и делает возвращенную строку новой текущей строкой. Если n или @nvar равны 0, не возвращается

никакая строка. n должно быть целым числом, а @nvar должна иметь тип данных smallint, tinyint или int.

- RELATIVE { n | @nvar}. Если n или @nvar имеют положительное значение, возвращает строку, отстоящую на n строк вперед от текущей строки, и делает возвращенную строку новой текущей строкой. Если n или @nvar имеют отрицательное значение, возвращает строку, отстоящую на n строк назад от текущей строки, и делает возвращенную строку новой текущей строкой. Если n или @nvar равны 0, возвращает текущую строку. Если при первой выборке из курсора инструкция FETCH RELATIVE указывается с отрицательными или равными нулю параметрами n или @nvar, то никакая строка не возвращается. n должно быть целым числом, а @nvar должна иметь тип данных smallint, tinyint или int.

Опциями оператора FETCH являются:

- GLOBAL – указывает, что параметр имя-курсора ссылается на глобальный курсор.
- INTO – позволяет поместить данные из столбцов выборки в локальные переменные.

Каждая переменная из списка, слева направо, связывается с соответствующим столбцом в результирующем наборе курсора. Тип данных каждой переменной должен соответствовать типу данных соответствующего столбца результирующего набора, или должна обеспечиваться поддержка неявного преобразования в тип данных этого столбца. Количество переменных должно совпадать с количеством столбцов в списке выбора курсора.

**Синтаксис инструкции FETCH имеет вид:**

```
FETCH [ [ NEXT | PRIOR | FIRST | LAST | ABSOLUTE { n | @nvar } | RELATIVE { n | @nvar }] FROM ]
{ { [ GLOBAL ] имя-объявленного-курсора } | @имя-переменной-курсора }
[ INTO список-имен-переменных ]
```

После выполнения каждой инструкции FETCH функция @@FETCH_STATUS обновляется, отражая состояние последней выборки. Функция @@FETCH_STATUS показывает такие состояния, как выборка за пределами первой или последней строк курсора. Функция @@FETCH_STATUS глобальна для соединения и обновляется после любой выборки в любом курсоре, открытом во время соединения. Если состояние нужно посмотреть позже, то перед выполнением следующей инструкции в пределах соединения необходимо сохранить значение функции @@FETCH_STATUS в пользовательской переменной. Даже если следующей инструкцией будет не FETCH, а INSERT, UPDATE или DELETE, сработает триггер, содержащий инструкции FETCH, что приведет к обновлению значения функции @@FETCH_STATUS.

```
-- Объявляем курсор
DECLARE CursorTest CURSOR
GLOBAL SCROLL STATIC FOR
      SELECT OrderID, CustomerID
      FROM CursorTable

-- Объявляем переменные для хранения
DECLARE @OrderID int
DECLARE @CustomerID varchar(5)

-- Откроем курсор и запросим первую запись
OPEN CursorTest
FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID -- Обработаем в цикле все записи курсора
WHILE @@FETCH_STATUS=0
BEGIN
      PRINT CONVERT(varchar(5),@OrderID) + '' + @CustomerID
      FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID
END
```