Хранимые процедуры (stored procedures)

Хранимые процедуры Transact SQL аналогичны подпрограммам в других алгоритмических языках.

Они могут принимать данные через входные параметры и возвращать результат через выходные параметры.

Программный код, являющийся содержанием хранимой процедуры состоит из одного пакета.

Помимо результатов, возвращаемых через выходные параметры, хранимая процедура может возвращать один или более наборов записей, таких же, какие возвращает оператор *SELECT*.

Хранимая процедура создаётся оператором *CREATE PROC[EDURE]*:

CREATE PROC [EDURE] [<владелец>.]<имя процедуры>
[;номер]
{[<параметр> <тип> [VARYING] [=<значение по умолчанию>] [OUTPUT]] }…

[WITH { RECOMPILE | ENCRYPTION | RECOMPILE, ENCRYPTION}]

AS <операторы Transact SQL>

<имя процедуры> - идентификатор (без @)может иметь длину более 128 символов. Процедура может быть квалифицирована именем схемы, но это необязательно.

~~*;номер*. После имени процедуры может следовать~~

***;номер*.**

~~*Номер* – произвольное целое. Используется для того, чтобы можно было создать группу процедур с одним и тем же именем, различающихся по номеру. Группа таких процедур может быть удалена одним оператором DROP. Например, если имеются процедуры с именами **MyProc;1, MyProc;2**, то их можно удалить одним оператором *DROP PROCEDURE MyProc.*~~

Обещали удалить с 2008

< *параметр* > - идентификатор, начинающийся с символа @ .

Максимальное число параметров – 2100. Параметры локальны в процедуре. Параметры могут использоваться там, где могут быть использованы константы.

Они не могут выступать в качестве имен таблиц, полей и других объектов БД.

<тип > - Допустимы все типы, включая **text, ntext** и **image.**

Тип **cursor** может быть использован только для выходного параметра (OUTPUT). Для параметра типа **cursor** должны быть указаны спецификации VARYING и OUTPUT.

VARYING – указывает, что результирующий набор может изменяться.

<значение по умолчанию> - если значению по умолчанию задано для параметра, то к ней можно обратиться, не указывая значения этого параметра. Значение по умолчанию может быть константой или NULL.

OUTPUT – указывает, что параметр является выходным. Используется для того, чтобы возвратить значение вызывающей программе.

{RECOMPILE | ENCRYPTION | RECOMPILE, ENCRYPTION} - RECOMPILE указывает, что план выполнения процедуры перекомпилируется перед исполнением процедуры. ENCRYPTION требует от SQL Server шифровать текст процедуры, помещаемый в системную таблицу **syscomments.**

Процедуры могут быть вложенными, в том смысле, что одна процедура вызывает другую. Текущий уровень вложенности возвращается функцией @@NESTLEVEL.

Максимальная глубина вложения равна 32. Процедуры могут быть рекурсивными, то есть, способны вызывать сами себя. Поскольку рекурсия есть частный случай вложенного вызова, максимальная глубина рекурсии также ограничена 32.

Пример процедуры, выполняющей удаление из базы данных «Склад» всего, что относится к уровню классификации товара @Tov_ID.

```
CREATE PROCEDURE dbo.DeleteTovar @Tov_ID int as
declare @IsTovar bit, @Tovar_ID int
-- выясним, является ли @Tov_ID товаром или
  уровнем классификации
select @IsTovar=IsTovar from Tovar where
  Tovar_ID=@Tov_ID
-- если это товар, удалим его и упоминание его в
  PriceList и  SostNakl
if @IsTovar=1 begin
  delete from PriceList where Tovar_ID =
   @Tov_ID
  delete from SostNakl where Tovar_ID=@Tov_ID
  delete from Tovar where Tovar_ID=@Tov_ID
end
```

```
else begin
  -- это не товар, а уровень классификации
     -- пройдем по всем его сыновьям в дереве классификации
   declare dt cursor local forward_only
    for select Tovar_ID from Tovar where Parent_ID=@Tov_ID
   open dt

   while 1=1 begin
    fetch next from dt into @Tovar_ID
    if @@fetch_status<>0  break
       -- потомков обрабатываем точно также
       exec dbo.DeleteTovar @Tovar_ID
   end

   close dt
  deallocate dt
end
delete from Tovar where Tovar_ID=@Tov_ID
GO
```

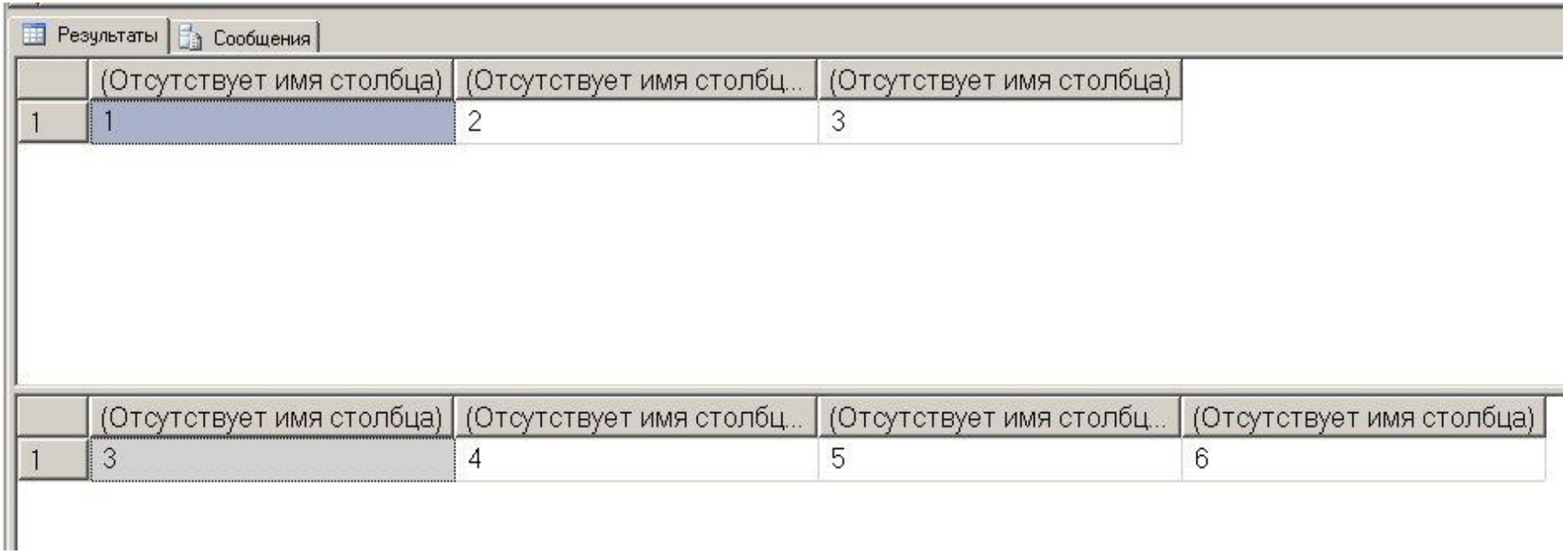
Хранимые процедуры могут возвращать результат своей работы четырьмя способами:

1) с помощью выходных параметров
2) код возврата (тип int)
3) наборы данных для каждого оператора select, выполняемого процедурой или другими процедурами, которые из неё вызываются
4) в виде глобального курсора, к которому можно обратиться после вызова процедуры

Пусть, например имеется процедура:

```
CREATE PROCEDURE MyProc AS
select 1,2,3
select 3,4,5,6
```

Ниже изображен результат выполнения оператора *exec MyProc* в Management Studio



Процедура вызывается оператором *Exec[ute]*.

Синтаксис вызова процедуры:

EXEC[UTE] [*@return_status* =]

имя процедуры [параметр [output]] [, параметр…]]

Параметр может передаваться как позиционный и как ключевой. Если они передаются как ключевые, то их следование необязательно такое же, как у параметров. Например:

```
create proc ff @x int, @y int...
   . . . . . .
   exec ff @y=8, @x=3
```

Оператор EXECUTE может выполнить текст Transact SQL, находящийся в символьной строке или символьной переменной. Например:

```
exec ('select * from Tovar')
```

или

```
declare @s varchar(100)
set @s='select * from Tovar'
execute (@s)
```

Имя процедуры может быть присвоено переменной:

```
DECLARE @proc_name varchar(30);
SET @proc_name = 'MyProc';
EXEC @proc_name;
```

@return_status – переменная, которой присваивается возвращаемое значение.

Переменная должна быть объявлена в пакете, вызывающем процедуру. Процедура может содержать оператор return или return <значение>.

Если в операторе return указано значение, то именно оно возвращается в качестве @return_status. Если используется оператор return, то возвращаемое значение равно 0.

Пример:

```
declare @x int
exec @x=MyProc
select @x
```

Триггеры

Триггер – это специфический тип процедуры, которая вызывается автоматически, когда выполняются операции INSERT, UPDATE, DELETE.

Никакая процедура, или функция не вызывают триггер явно. Триггер относится к одной конкретной таблице и неявно вызывается, когда в неё вносятся изменения операторами *insert, update, delete*.

Целями, которые преследует триггер, могут быть:

- отслеживание ссылочной и семантической целостности базы данных
- Выполнение действий, обеспечивающих дополнительный побочный эффект при выполнении операций вставки, модификации и удаления.

CREATE TRIGGER < имя триггера > ON { < имя таблицы или view }
[WITH ENCRYPTION]
{ FOR | AFTER | INSTEAD OF }
< любая комбинация ключевых слов INSERT, UPDATE, DELETE>

[WITH APPEND]
AS
< операторы Transact SQL >

- *имя таблицы или view* – триггер будет вызван при попытке внесения изменений в указанную таблицу или view.
- *with encryption* – текст триггера хранится в системной таблице *syscomments*. Если указано *with encryption*, то он будет зашифрован.

- *after* – указывает, что триггер должен стартовать после того, как действия оператора, вызвавшего триггер, успешно завершены. Проверка ссылочной целостности и ограничений *CHECK* предшествуют запуску триггера.

Опция *after* является умолчанием, если *for* – единственное ключевое слово в определении триггера. Опция a*fter* не может быть указана для view.

- *instead of* – означает, что триггер будет выполняться вместо выполнения одной из операций *INSERT, DELETE, UPDATE*.

Подробно триггеры *instead of* рассматриваются далее.

В триггере доступны две дополнительные таблицы – *INSERTED* и *DELETED*, которые содержат вставленные и удалённые записи таблицы, для которой предназначен триггер.

Модификация записи (*UPDATE*) выполняется как удаление старой записи и вставка новой, следовательно, при модификации одной записи, таблицы *INSERTED* и *DELETED* будут содержать по одной записи.

Триггер может оценить выполняемые изменения таблицы как недопустимые и возбудить состояние ошибки с передачей сообщения клиенту с помощью функции *RAISERROR*.

```
RAISERROR ( { msg_id | msg_str } { , severity , state }
    [ , argument [ ,...n ] ] )
    [ WITH option [ ,...n ] ]
```

Аргументы:

- msg_id – целочисленный идентификатор сообщения из системной таблицы сообщений SQL Server master.sysmessages.
- msg_str – строка, содержащая формат сообщения, который полностью соответствует соглашениям о форматах оператора printf языка Си.

Пример:

```
RAISERROR(' Удаление товара %s недопустимо, так как имеются данные о продажах ', 16,1, @TovarName)
```

Здесь «%s» - формат для аргумента @TovarName.

severity – уровень серьёзности ошибки.

Уровни серьёзности от 0 до 18 могут быть использованы любым пользователем.

Уровни от 19 до 25 могут исходить только от членов роли sysadmin. Уровни от 20 до 25 являются фатальными и влекут немедленный разрыв соединения в котором это произошло.

Уровни с 11 по 16 – ошибки, которые могут быть исправлены конечным пользователем.

Уровень, равный 10, определяет информационное сообщение, не влияющее

state – произвольное целое от 1 до 127. Может быть использовано как признак, позволяющий определить место, в котором была вызвана функция RAISERROR.

Далее пример триггера в «триггер для SostNakl в бд Warehouse.txt»

Триггеры *instead of*

Для каждой из операций *INSERT, DELETE, UPDATE* для таблицы или *view* может быть определён триггер, который будет вызываться вместо выполнения стандартной операции.

Эта возможность особенна важна для применения по отношению к представлениям (*view*), построенном на основании нескольких таблиц.

Напомним, что стандартные операции *INSERT, UPDATE, DELETE* к таким *view* неприменимы.

Рассмотрим пример выполнения операции *INSERT* для *view* TovarWithCurPrice, содержащего данные товара и его текущую цену:

```
CREATE VIEW TovarWithCurPrice
  AS
  SELECT Tovar.Tovar_ID, Tovar.TovarName,
     Tovar.IsTovar, Tovar.Amount, Tovar.MeasUnit_ID,
     Tovar.Parent_ID, PriceList.Price, PriceList.DateStart
  FROM  PriceList INNER JOIN
    Tovar ON PriceList.Tovar_ID = Tovar.Tovar_ID
  WHERE (PriceList.DateStart =
   (SELECT MAX(DateStart) FROM PriceList WHERE
  PriceList.Tovar_ID = Tovar.Tovar_ID))
```

Допустим что последняя цена - текущая

При выполнении операции *INSERT* для этого *view* должна быть добавлена одна запись в таблицу Tovar и одна запись с его текущей ценой – в таблицу PriceList. Это может быть реализовано триггером *instead of insert*:

```
CREATE TRIGGER InsTovarWithPrice ON
   [dbo].[TovarWithCurPrice] instead of INSERT
   AS begin
   declare @Tovar_ID int,@TovarName varchar(30),@IsTovar
   bit,@Amount float, @MeasUnit_ID int,@Parent_ID int,
   @Price smallmoney, @DateStart smalldatetime

   declare ps cursor for
     select TovarName, IsTovar, Amount, MeasUnit_ID,
   Parent_ID,
       Price, DateStart
     from inserted
   open ps
```

```
while  1=1 begin
  fetch next from ps into
    @TovarName, @IsTovar, @Amount, @MeasUnit_ID,
    @Parent_ID, @Price, @DateStart
  if @@fetch_status<>0 break;
  -- добавим новый товар...
  insert into Tovar( TovarName, IsTovar, Amount,
  MeasUnit_ID, Parent_ID) values(@TovarName, @IsTovar,
  @Amount,@MeasUnit_ID,@Parent_ID)

  set @Tovar_ID=@@identity
  --... и его цену
  insert into PriceList(Tovar_ID, Price, DateStart)
    values(@Tovar_ID,@Price, @DateStart)
  end
  close ps
  deallocate ps
  end
```

Оператор *INSERT*, выполняющий вставку записи(ей) во *view* обязан предоставить значения всех полей *view*, которые не допускают неопределенных значений. Для приведенного примера оператор *INSERT* мог бы иметь вид:

```
insert into TovarWithCurPrice (Tovar_ID, TovarName, IsTovar,
   Amount,   MeasUnit_ID, Parent_ID, Price, DateStart)
values(
  0 /* Tovar_ID */,
  'Новый товар' /* TovarName */
  1, /*IsTovar*/
  23.66, /* Amount */
  3, /* MeasUnit_ID /
  7, /* Parent_ID */
  22.76, /* PriceList.Price */
  '20120901' /* PriceList.DateStart */)
```

Обратите внимание на то, что приведённый оператор *INSERT* предоставляет значение для автоинкрементного поля Tovar_ID. Триггер не использует это значение, однако, оно обязано присутствовать в операторе *INSERT*.

То же касается вычисляемых полей. Если вычисляемое поле имеет свойство *NOT NULL*, то значение для него должно иметься в списке *VALUES*, хотя оно и не будет использовано триггером.

Функции, возвращающие скаляр, создаются оператором CREATE FUNCTION, имеющим следующий синтаксис:

```
CREATE  FUNCTION [<имя владельца>.] <имя функции>
    ( [ { <параметр1> [AS] <тип> [ = default ] } [ ,...n ] ]
    )
RETURNS <тип возвращаемого значения>
[ WITH <опции> [ [,] ...n] ]
[ AS ]
BEGIN
        <тело функции>
        RETURN <скалярное выражение>
END
```

- *параметр.* Имена параметров должны удовлетворять соглашения об именах переменных.
- Параметру может быть дано значение по умолчанию, которое он будет иметь, если для соответствующего аргумента не задано значение.
- Параметр может иметь любой скалярный тип, кроме timestamp.

- *тип возвращаемого значения* может быть любым, кроме text, ntext, image.
- *опции*
  - ENCRYPTION – текст функции будет зашифрован
  - SCHEMABINDING – означает, что функция связывается с объектами базы данных, которые от неё зависят. Это могут быть вычисляемые поля, другие функции или процедуры. Невозможно удалить или модифицировать функцию, на которую ссылаются другие объекты базы данных, если она создана *with schemabinding*.

Пример 1. Приведенная ниже функция вычисляет

$$s = \sum_{i=1}^{n} \frac{1}{i^2}$$

```
CREATE FUNCTION dbo.f (@n int)
RETURNS float AS
BEGIN
declare @i int, @s float
set @s=0
set @i=1
while @i<=@n begin
     set @s=@s+1./@i/@i
     set @i=@i+1
end
return @s
END
```

Пример 2. Вычислить суммарную стоимость товара @Tovar_ID по текущей цене.

```
CREATE FUNCTION dbo.TovarCost (@Tovar_ID int)
-- суммарная стоимость товара @Tovar_ID на складе по текущей цене
RETURNS money AS
BEGIN
declare @Price money
select @Price=Price
from PriceList
where DateStart=
    (select max(DateStart)
    from PriceList
    where PriceList.Tovar_ID=@Tovar_ID)
        and PriceList.Tovar_ID=@Tovar_ID
return coalesce(@Price*(select Amount from Tovar where Tovar_ID=@Tovar_ID),0)
END
```

Пример 3. Найти цену товара на текущую дату.

```
CREATE FUNCTION dbo.CurCost (@Tovar_ID
   int)
-- возвращает текущую цену товара
RETURNS float AS
BEGIN
declare @curDate smalldatetime
--поскольку нельзя употреблять
   недетерминированную функцию getdate()
-- внутри функции, обратимся к view,
   которое возвращает текущую дату
```

```
select @curDate=CurDate from CurrentDate
-- здесь CurrentDate - представление
declare @Price float
select @Price=Price
from PriceList
where DateStart<=@curDate
    and (DateEnd is null or DateEnd >=@curDate)
    and Tovar_ID=@Tovar_ID
return @Price
END
```

Функция, возвращающая скаляр может входить как операнд в любое выражение, например:

```
set @x=@Amount*dbo.CurCost(25)
```

Функции, возвращающие таблицу

Функции, возвращающие таблицу, создаются оператором CREATE FUNCTION, имеющим следующий синтаксис:

```
CREATE FUNCTION [<имя владельца>. ] <имя функции>
    ( [ { <параметр1> [AS] <тип> [ = default ] } [ ,...n ] ] )
RETURNS <имя переменной-таблицы> TABLE
    <определение таблицы>
[ WITH <опции> [ [,] ...n ] ]
[ AS ]
BEGIN
        <тело функции>
        RETURN
END
```

Пример. Получить состояние склада на дату @d. Возвращаемая таблица должна иметь структуру (Tovar_ID, TovarName, Amount)

```
CREATE FUNCTION dbo.Otkat (@d datetime)
RETURNS @x table(
  Tovar_ID int,
  TovarName varchar(30),
  Amount float null
)
 AS
```

```
BEGIN
insert into @x(Tovar_ID,TovarName,Amount)
select Tovar_ID,TovarName,
  t.Amount-
  coalesce((select sum(Amount) -- вычесть все поступления
          from Nakl n,SostNakl s
          where n.Dat>=@d
            and n.Nakl_ID=s.Nakl_ID
            and s.Tovar_ID=t.Tovar_ID
            and n.Inout='+'),0)
 +coalesce((select sum(Amount) -- прибавить все отгрузки
          from Nakl n,SostNakl s
          where n.Dat>=@d
            and n.Nakl_ID=s.Nakl_ID
            and s.Tovar_ID=t.Tovar_ID
            and n.Inout='-'),0)
from Tovar t
where IsTovar=1
return
END
```

Обращение к функции, возвращающей таблицу, имеет вид такой же, как и к любому другому источнику данных, например:

```
select * from Otkat('20061001')
```

Уровни изоляции

При условии работы многих пользователей с одной и той же базой данных они могут мешать друг другу. В качестве примера рассмотрим проблему *потерянных обновлений*.

Пусть два пользователя (две транзакции) присылают на ваш банковский счёт соответственно 1 и 2 рубля. Исходно на счёте лежало 5 рублей.

Последовательность действий может быть такой:

- Транзакция 1 (T1) читает сумму на счёте (5 рублей)
- Транзакция 2 (T2) читает сумму на счете (5 рублей)
- T1 складывает 5+1=6 и записывает результат в БД. Теперь на счёте 6 рублей.
- T2 складывает 5+2=7 и записывает результат в БД. Теперь на счёте 7 рублей вместо 8, как это должно было бы быть.

Как видим, обновление выполненное транзакцией T1 потеряно. Это произошло потому, что транзакция T2 читала данные незавершённой транзакции T1. Если бы T1 и T2 выполнялись последовательно одна за другой, то не возникло бы никаких проблем.

Транзакции называются *сериализуемыми*, если их результат всегда эквивалентен их последовательному выполнению.

Для изоляции одной транзакции от другой используются блокировки. В приведенном примере транзакция T1 должна была блокировать (запретить) чтение и запись суммы на счёте до своего завершения.

Тогда транзакции T2 пришлось бы ждать завершения T1. Для длительно выполняемых транзакций это может создать у пользователей впечатление медленной работы программы.

Таким образом, в процессе разработки приложения, программист должен иметь в виду две противоречащих друг другу цели:

1) пользователь по возможности не должен ощущать задержек, создаваемых присутствием в сети других пользователей

2) целостность базы данных должна быть гарантирована.

Программируя транзакцию, следует понимать, что гарантии того, что она будет успешно завершена, не существует, и поэтому каждый раз следует предусматривать действия, которые нужно выполнить в этом случае.

Как правило, программист не указывает какие данные и как следует блокировать (хотя опытные программисты имеют возможность вмешаться в этот процесс).

Управлением блокировками занимается *менеджер блокировок* (lock manager), который руководствуется *уровнем изоляции* транзакций, который назначил программист.

В стандарте ANSI SQL-92 [MS, ANSI] определяются четыре уровня изолированности.

1) Незафиксированное (грязное) чтение (READ UNCOMMITED).
2) Зафиксированное чтение (READ COMMITED).
3) Запрет неповторяемого чтения (REPEATABLE READ).
4) Сериализуемость (SERIALIZABLE).

Они определяются с помощью определения сериализуемости и трех запрещенных последовательностей операций, названных феноменами:

1) грязное чтение,

2) неповторяемое чтение

3) фантомы.

В стандарте нет четкого определения феномена, предполагается что феномен - это последовательность операций, обладающая аномальным (возможно, не сериализуемым) поведением.

Грязное чтение.

1. t1 изменяет строку данных.
2. t2 читает эту строку
3. t1 выполняет откат.

Теперь t2 работает со строкой, которая никогда не существовала в БД.

Неповторяемое чтение.

1. t1 читает строку
2. t2 обновляет или удаляет эту строку
3. t2 завершается

Если t1 попытается повторить чтение, то либо этой строки уже нет, либо она содержит другие данные.

Иллюзии. (Фантомы)

1. t1 выбирает множество строк, удовлетворяющих некоторому критерию поиска.
2. t2 добавляет новую строку тоже удовлетворяющую этому критерию.

Если t1 повторно выполнит свой запрос, то результат будет содержать новую строку.

Уровень изоляции транзакций устанавливается оператором **set transaction isolation level**.

Синтаксис: (Уровень sql server 2000)

```
SET TRANSACTION ISOLATION LEVEL
      {    READ COMMITTED
         | READ UNCOMMITTED
         | REPEATABLE READ
         | SERIALIZABLE
      }
```

(в 2005 добавлен snapshot)

Аргументы:

- **READ COMMITTED** – Указывает, что на время чтения удерживается блокировка, чтобы избежать *грязного чтения*. Возможны феномены *неповторяемое чтение* и *фантомы*. Этот уровень изоляции устанавливается по умолчанию.
- **READ UNCOMMITTED –** допускает «грязное чтение».
- **REPEATABLE READ –** Блокируются все данные, используемые в запросе. Возможны *фантомы*. Этот уровень более жесткий, чем **READ COMMITED**.
- **SERIALIZABLE** – блокировки не допускают изменения и добавления данных. Это наиболее ограничительный уровень.

Для изоляции транзакций друг от друга используются блокировки. Транзакция может установить блокировку на тот или иной ресурс, что препятствует другим транзакциям выполнять те или иные манипуляции над данными. В SQL Server объектом блокировки может быть:

- запись
- страница (8 кб) данных или индекса
- Extent – 8 страниц данных или индекса
- Таблица
- База данных

Оператор BEGIN TRANSACTION

Отмечает стартовую точку явно объявляемой транзакции. Выполнение оператора BEGIN TRANSACTION увеличивает счетчик числа вложенных транзакций @@TRANCOUNT на 1.

Синтаксис

BEGIN TRAN [SACTION] [*имя транзакции* | *@tran_name_variable*]

Аргументы

transaction_name – имя транзакции длиной не более 32 символов. Если используются вложенные транзакции, то имя может иметь только самая внешняя.

@tran_name_variable – переменная, содержащая имя транзакции.

Если опция **IMPLICIT_TRANSACTIONS** установлена в **on**, SQL Server неявным образом открывает транзакцию при выполнении каждого из операторов:

ALTER TABLE, FETCH, REVOKE, CREATE , GRANT, SELECT

DELETE, INSERT, TRUNCATE TABLE, DROP, OPEN

UPDATE

Спросить значение опции можно следующим образом:

```
select @@OPTIONS & 2
```

Установить значение опции:

```
set IMPLICIT_TRANSACTIONS {on | off}
```

Блокировки

Для изоляции транзакций друг от друга используются блокировки. Транзакция может установить блокировку на тот или иной ресурс, что препятствует другим транзакциям выполнять те или иные манипуляции над данными. В SQL Server объектом блокировки может быть:

- запись
- страница (8 кб) данных или индекса
- Extent – 8 страниц данных или индекса
- Таблица
- База данных

Блокировки

SQL Server может применять следующие разновидности блокировок:

Блокировка	Описание
Shared (S)	Разрешается читать данные, но запрещается изменять их. Блокировка снимается, как только данные прочитаны.
Exclusive (X)	Монопольная блокировка. Запрещает как чтение, так и запись.
Update (U)	Блокировка для обновления. Используется для того, чтобы избежать создания тупиков.
Intent (намерение)	Используется для установки иерархии блокировок. Имеет типы: intent shared (IS), intent exclusive (IX), and shared with intent exclusive (SIX).
Schema	Используется для выполнения операции, изменяющей свойства или схему таблицы. Существует два вида такой блокировки: Sch-M и Sch-S.
Bulk Update (BU)	Для передачи данных в таблицу потоком.

Блокировка для обновления (Update).

Типичная ситуация при модификации данных заключается в следующем. Транзакция T1 читает данные, что требует блокировки типа S.

Затем она намеревается изменить эти данные, что требует монопольной блокировки (X). Если другая транзакция (T2) в это же время попытается сделать то же самое, то возможно создание тупика.

Действительно, после того, как обеим транзакциям удалось установить взаимно совместимые блокировки типа S, они будут бесконечно ждать освобождения ресурса другой транзакцией, так как требуемая монопольная блокировка несовместима ни с какой другой.

Для того, чтобы избежать этой ситуации, используется блокировка на обновление (U). В каждый данный момент только одна транзакция может установить блокировку типа U на ресурс.

Действительно, после того, как обеим транзакциям удалось установить взаимно совместимые блокировки типа S, они будут бесконечно ждать освобождения ресурса другой транзакцией, так как требуемая монопольная блокировка несовместима ни с какой другой.

Для того, чтобы избежать этой ситуации, используется блокировка на обновление (U). В каждый данный момент только одна транзакция может установить блокировку типа U на ресурс.

Блокировка Intent.

Блокировка типа «намерение» означает, что SQL Server намерен выполнить блокировку части ресурса. Например, блокировка типа «намерение» может быть наложена на таблицу, если транзакция намерена блокировать (S или X) строки или страницы этой таблицы.

Установка такой блокировки преследует цель помешать другой транзакции установить блокировку типа X на таблицу.

Блокировка типа **intent** улучшает быстродействие SQL Server, так как проверяется только наличие блокировки на уровне таблицы, и не требуется искать блокировки для каждой строки или страницы в таблице для того, чтобы выяснить, можно ли блокировать таблицу.

Разновидности **intent** – блокировок: intent shared (IS), intent exclusive (IX), и shared with intent exclusive (SIX).

Блокировка	Описание
Intent shared (IS)	Индицирует намерение транзакции прочитать некоторые, (но не все) данные, накладывая S-блокировку на ресурсы, находящиеся ниже в иерархии.
Intent exclusive (IX)	Намерение блокировать монопольно часть ресурсов объекта..
Shared with intent exclusive (SIX)	Намерение прочитать весь ресурс и модифицировать часть (но не весь), накладывая IX-блокировку на эту часть. Другая транзакция может одновременно выполнить IS-блокировку ресурса верхнего уровня. Например, транзакция накладывает SIX-блокировку на таблицу (при этом оказываются возможны одновременные IS-блокировки другими транзакциями), затем IX-блокировку на страницы и X-блокировку на записи. В каждый данный момент может существовать только одна SIX-блокировка ресурса, предотвращающая модификацию ресурса другими транзакциями.

Ниже приведена таблица совместимости блокировок.

	Уже существующая блокировка					
Требуемая блокировка	**IS**	**S**	**U**	**IX**	**SIX**	**X**
Intent shared (IS)	Да	Да	Да	Да	Да	Нет
Shared (S)	Да	Да	Да	Нет	Нет	Нет
Update (U)	Да	Да	Нет	Нет	Нет	Нет
Intent exclusive (IX)	Да	Нет	Нет	Да	Нет	Нет
Shared with intent exclusive (SIX)	Да	Нет	Нет	Нет	Нет	Нет
Exclusive (X)	Нет	Нет	Нет	Нет	Нет	Нет

Оператор COMMIT TRANSACTION

Помечает конец успешной транзакции, неявной или объявленной пользователем.

Если @@TRANCOUNT равно 1, COMMIT TRANSACTION делает все изменения БД, совершенные после начала транзакции, окончательными, освобождает все ресурсы, занятые соединением и уменьшает @@TRANCOUNT до 0.

Если @@TRANCOUNT больше 1, COMMIT TRANSACTION уменьшает @@TRANCOUNT на 1.

Синтаксис

```
COMMIT [TRAN[SACTION] [ transaction_name |
    @tran_name_variable ] ]
```

Аргументы:

transaction_name – игнорируется SQL Server. Используется для повышения читабельности.

@tran_name_variable имя переменной, содержащей имя транзакции. Тоже игнорируется.

Оператор ROLLBACK TRANSACTION

Выполняет откат к началу транзакции или к **savepoint.** О savepoint не рассказываю

Синтаксис:

ROLLBACK [TRAN [SACTION]
[*transaction_name* | *@tran_name_variable*
| *savepoint_name* | *@savepoint_variable*]]

ROLLBACK TRANSACTION без имени *savepoint* или транзакции выполняет откат к началу транзакции.

Системные таблицы

Системные таблицы в каждой базе данных

Вся информация о базе данных хранится в ней самой в виде совокупности системных таблиц, образующих базу данных, содержащую описание базы данных пользователя. Таким образом, совокупность системных таблиц образует метабазу данных. В частности, она содержит описание самоё себя.

К системным таблицам неприменимы операторы insert, update, delete. Системные таблицы изменяются, когда выполняются операторы create, drop, alter… Тем не менее, к ним применим оператор select, что позволяет извлекать информацию о тех или иных свойствах базы данных.

Таблица sysobjects

Таблица содержит по одной строке для каждого объекта в базе данных. Объектами являются, например, ограничения (constraint), правила, таблицы, функции.

Имя поля	Тип данных	Описание
name	sysname	Имя объекта
Id	Int	Идентификатор объекта.
xtype	char(2)	Тип объекта. Может принимать значения (перечислены не все): F - ограничение FOREIGN KEY FN – скалярная функция P – хранимая процедура PK – ограничение PRIMARY KEY S - системная таблица TF – функция, возвращающая таблицу TR - триггер U - таблица пользователя V - View
parent_obj	Int	Идентификатор родительского объекта. Например, для триггера это может быть идентификатор таблицы, к которой он относится.
crdate	Datetime	Дата и время создания объекта

Таблица syscolumns

Содержит по одной строке для каждого поля в таблице или view и по одной строке для каждого параметра хранимой процедуры. Ниже описаны некоторые поля таблицы syscolumns.

Имя поля	Тип	Описание
name	sysname	Имя поля таблицы (view) или имя параметра процедуры.
id	int	Идентификатор таблицы, view или процедуры в sysobjects.
xtype	tinyint	Physical storage type from systypes.
xusertype	smallint	ID of extended user-defined data type.
length	smallint	Maximum physical storage length from systypes.
colid	smallint	Column or parameter ID.
type	tinyint	Physical storage type from systypes.
usertype	smallint	ID of user-defined data type from systypes.
prec	smallint	Level of precision for this column.
scale	int	Scale for this column.
iscomputed	int	Flag indicating whether the column is computed: 0 = Noncomputed. 1 = Computed.
isoutparam	int	Indicates whether the procedure parameter is an output parameter: 1 = True. 0 = False.

```
--Функция возвращающая размер поля в байтах
CREATE FUNCTION dbo.k_sysFieldWidth (@Table
    sysname, @Column sysname)
RETURNS int AS
BEGIN
declare @width int
select @width=c.length
from sysobjects o ,syscolumns c
where o.id=c.id
  and o.name=@Table
  and c.name=@Column
return @width
END
```

Список таблиц, содержащих поле @FieldName

```
CREATE FUNCTION
    dbo.k_sys_ListOfTablesWithField(@FieldName sysname)
RETURNS @x table(TableName sysname) as
BEGIN
insert into @x(TableName)
select sysobjects.name
from sysobjects,syscolumns
where sysobjects.id=syscolumns.id
  and syscolumns.name=@FieldName
  and sysobjects.xtype='U'
order by sysobjects.name
return
END
```

Список объектов (функций, процедур, триггеров), содержащих текст @txt.

```
CREATE FUNCTION k_sys_ObjectsContainingText (@txt
  varchar(256))
RETURNS @x table(ObjName sysname) AS
BEGIN
insert into @x(ObjName)
select distinct o.name
from sysobjects o, syscomments c
where o.id=c.id
  and c.text like ('%'+@txt+'%')
order by o.name
return
END
```