# Транзакции

## Общие сведения о транзакциях

Транзакция — это последовательность операций, выполняемая как единое целое. В составе транзакций можно исполнять почти все операторы языка Transact-SQL. Если при выполнении транзакции не возникает никаких ошибок, то все модификации базы данных, сделанные во время выполнения транзакции, становятся постоянными. Транзакция выполняется по принципу «все или ничего». Транзакция не оставляет данные в промежуточном состоянии, в котором база данных не согласована. Транзакция переводит базу данных из одного целостного состояния в другое.

В качестве примера транзакции рассмотрим последовательность операций по приему заказа в коммерческой компании. Для приема заказа от клиента приложение ввода заказов должно:

a) выполнить запрос к таблице *товаров* и проверить наличие товара на складе;
b) добавить заказ к таблице *счетов*;
c) обновить таблицу *товаров*, вычтя заказанное количество товаров из количества товара, имеющегося в наличии;
d) обновить таблицу *продаж*, добавив стоимость заказа к объему продаж служащего, принявшего заказ;
e) обновить таблицу *офисов*, добавив стоимость заказа к объему продаж офиса, в котором работает данный служащий.

Для поддержания целостности транзакция должна обладать четырьмя свойствами АСИД: *атомарность, согласованность, изоляция* и *долговечность*. Эти свойства называются также ACID-свойствами (от англ., atomicity, consistency, isolation, durability).

- **Атомарность (Atomicity).** Транзакция должна представлять собой атомарную (неделимую) единицу работы (исполняются либо все модификации, из которых состоит транзакция, либо ни одна).
- **Согласованность** (или **Непротиворечивость**) **(Consistency).** По завершении транзакции все данные должны остаться в согласованном состоянии. Чтобы сохранить целостность всех данных, необходимо выполнение модификации транзакций по всем правилам, определенным в реляционных СУБД.
- **Изоляция (Isolation).** Модификации, выполняемые одними транзакциями, следует изолировать от модификаций, выполняемых другими транзакциями параллельно. Уровни изоляции транзакции могут изменяться в широких пределах. На каждом уровне изоляции достигается определенный компромисс между степенью распараллеливания и степенью непротиворечивости. Чем выше уровень изоляции, тем выше степень непротиворечивости данных. Но чем выше степень непротиворечивости, тем ниже степень распараллеливания и тем ниже степень доступности данных.
- **Долговечность** (или **Устойчивость**) **(Durability)**. По завершении транзакции ее результат должен сохраниться в системе, несмотря на сбой системы, либо (что касается незафиксированных транзакций) может быть полностью отменен вслед за сбоем системы.

**Способы определения границ транзакций**

**Замечания.**

1) В данной теме не будут учитываться особенности режима MARS (Multiple Active Result Sets). Режим MARS — это функция, которая появилась в SQL Server 2005 и ADO.NET 2.0 для выполнения нескольких пакетов по одному соединению. По умолчанию режим MARS отключен. Включить его можно, добавив в строку соединения параметр «MultipleActiveResultSets=True».
2) В данной теме не будут рассматриваться распределенные транзакции.

По признаку определения границ различают *автоматические, неявные и явные* транзакции.

**Автоматические транзакции.** Режим автоматической фиксации транзакций является режимом управления транзакциями SQL Server по умолчанию. В этом режиме каждая инструкция T-SQL выполняется как отдельная транзакция. Если выполнение инструкции завершается успешно, происходит фиксация; в противном случае происходит откат. Если возникает ошибка компиляции, то план выполнения пакета не строится и пакет не выполняется.

В следующем примере ни одна из инструкций INSERT во втором пакете не выполнится из-за ошибки компиляции. При этом произойдет откат первых двух инструкций INSERT, и они не будут выполняться:

```
CREATE TABLE Tab1 (
      Col1 int NOT NULL PRIMARY KEY,
      Col2 char(3)
);
GO
INSERT INTO Tab1 VALUES (1, 'aaa');
INSERT INTO Tab1 VALUES (2, 'bbb');
INSERT INTO Tab1 VALUSE (3, 'ccc');  -- Синтаксическая ошибка.
```

```
GO
SELECT * FROM Tab1;  -- Результат пустой.
GO
```

В следующем примере третья инструкция INSERT вызывает ошибку дублирования первичного ключа во время выполнения. Поэтому первые две инструкции INSERT выполняются успешно и фиксируются, а третья инструкция INSERT вызывает ошибку времени выполнения и не выполняется.

```
CREATE TABLE Tab1 (
      Col1 int NOT NULL PRIMARY KEY,
      Col2 char(3)
);
GO
INSERT INTO Tab1 VALUES (1, 'aaa');
INSERT INTO Tab1 VALUES (2, 'bbb');
INSERT INTO Tab1 VALUES (1, 'ccc');  -- Ошибка времени исполнения.
GO
SELECT * FROM Tab1;  -- Возвращаются две строки.
GO
```

**Неявные транзакции.** Если соединение работает в режиме неявных транзакций, то после фиксации или отката текущей транзакции SQL Server автоматически начинает новую транзакцию. В этом режиме явно указывается только граница окончания транзакции с помощью инструкций COMMIT TRANSACTION и ROLLBACK TRANSACTION. Для ввода в действие поддержки неявных транзакций применяется инструкция SET IMPLICIT_TRANSACTION ON. В конце каждого пакета необходимо отключать этот режим. По умолчанию режим неявных транзакций в SQL Server отключен.

В следующем примере сначала создается таблица Tab1, затем включается режим неявных транзакций, после чего начинаются две транзакции. После их исполнения режим неявных транзакций отключается:

```
CREATE TABLE Tab1 (
      Col1 int NOT NULL PRIMARY KEY,
      Col2 char(3) NOT NULL
)
GO
SET IMPLICIT_TRANSACTIONS ON
-- Первая неявная транзакция, начатая оператором INSERT
INSERT INTO Tab1 VALUES (1, 'aaa')
INSERT INTO Tab1 VALUES (2, 'bbb')
COMMIT TRANSACTION -- Фиксация первой транзакции
-- Вторая неявная транзакция, начатая оператором INSERT
INSERT INTO Tab1 VALUES (3, 'ccc')
SELECT * FROM Tab1
COMMIT TRANSACTION -- Фиксация второй транзакции
SET IMPLICIT_TRANSACTIONS OFF
GO
```

**Явные транзакции.** Для определения явных транзакций используются следующие инструкции:

- BEGIN TRANSACTION – задает начальную точку явной транзакции для соединения;
- COMMIT TRANSACTION или COMMIT WORK – используется для успешного завершения транзакции, если не возникла ошибка;
- ROLLBACK TRANSACTION или ROLLBACK WORK – используется для отмены транзакции, во время которой возникла ошибка.
- SAVE TRANSACTION – используется для установки точки сохранения или маркера внутри транзакции. Точка сохранения определяет место, к которому может возвратиться транзакция, если часть транзакции условно отменена. Если транзакция откатывается к точке сохранения, то ее выполнение должно быть продолжено до завершения с обработкой дополнительных инструкций языка T-SQL, если необходимо, и инструкции COMMIT TRANSACTION, либо транзакция должна быть полностью отменена откатом к началу. Для отмены всей транзакции следует использовать инструкцию ROLLBACK TRANSACTION; в этом случае отменяются все инструкции транзакции.

В следующем примере демонстрируется использование точки сохранения транзакции для отката изменений:

```
USE pubs
GO
BEGIN TRANSACTION royaltychange
   UPDATE titleauthor SET royaltyper = 65
```

```
   FROM titleauthor, titles
   WHERE royaltyper = 75 AND titleauthor.title_id = titles.title_id
      AND title = 'The Gourmet Microwave'

   UPDATE titleauthor SET royaltyper = 35
   FROM titleauthor, titles
   WHERE royaltyper = 25 AND titleauthor.title_id = titles.title_id
      AND title = 'The Gourmet Microwave'

   SAVE TRANSACTION percentchanged
/* После того, как обновлено royaltyper для двух авторов, вставляется точка сохранения
percentchanged, а затем определяется, насколько изменится заработок авторов после
увеличения на 10 процентов цены книги */
   UPDATE titles SET price = price * 1.1
   WHERE title = 'The Gourmet Microwave'

   SELECT (price * royalty * ytd_sales) * royaltyper
   FROM titles, titleauthor
   WHERE title = 'The Gourmet Microwave' AND titles.title_id = titleauthor.title_id
/* Откат транзакции до точки сохранения и фиксация транзакции в целом */
ROLLBACK TRANSACTION percentchanged
COMMIT TRANSACTION
```

Режим явных транзакций действует только на протяжении данной транзакции. После завершения явной транзакции соединение возвращается в режим, заданный до запуска этого режима, то есть в неявный или автоматический.

**Функции для обработки транзакций**

- @@TRANCOUNT возвращает число активных транзакций для текущего соединения. Инструкция BEGIN TRANSACTION увеличивает значение @@TRANCOUNT на 1, а инструкция ROLLBACK TRANSACTION уменьшает его до 0 (исключение — инструкция ROLLBACK TRANSACTION *имя_точки_сохранения*, которая не влияет на значение @@TRANCOUNT). Инструкции COMMIT TRANSACTION уменьшают значение @@TRANCOUNT на 1.
- XACT_STATE () сообщает о состоянии пользовательской транзакции текущего выполняемого запроса в соответствии с данными, представленными в таблице.

| **Возвращаемое значение** | **Пояснение** |
|---|---|
| 1 | Текущий запрос содержит активную пользовательскую транзакцию и может выполнять любые действия, включая запись данных и фиксацию транзакции. |
| 0 | У текущего запроса нет активной пользовательской транзакции. |
| -1 | В текущем запросе есть активная транзакция, однако произошла ошибка, из-за которой транзакция классифицируется как не фиксируемая. Запросу не удается зафиксировать транзакцию или выполнить откат до точки сохранения; можно только запросить полный откат транзакции. |

**Ограничения.**
1) Функция @@TRANCOUNT не может использоваться для определения фиксируемости транзакции.
2) Функция XACT_STATE не может использоваться для определения наличия вложенных транзакций.

**Ошибки, возникающие в процессе обработки транзакций**

Если ошибка делает невозможным успешное выполнение транзакции, SQL Server автоматически выполняет ее откат и освобождает ресурсы, удерживаемые транзакцией. Если сетевое соединение клиента с SQL Server разорвано, то после того, как SQL Server получит уведомление от сети о разрыве соединения, выполняется откат всех необработанных транзакций для этого соединения. В случае сбоя клиентского приложения, выключения либо перезапуска клиентского компьютера соединение также будет разорвано, а SQL Server выполнит откат всех необработанных транзакций после получения уведомления о разрыве от сети. Если клиент выйдет из приложения, выполняется откат всех необработанных транзакций.

В случае ошибки (нарушения ограничения целостности) во время выполнения инструкции в пакете по умолчанию SQL Server выполнит откат только той инструкции, которая привела к ошибке. Это поведение можно изменить с помощью инструкции SET XACT_ABORT. После выполнения инструкции SET XACT_ABORT ON любая ошибка во время выполнения инструкции приведет к автоматическому откату текущей транзакции.

На случай возникновения ошибок код приложения должен содержать исправляющее действие: COMMIT или ROLLBACK. Эффективным средством для обработки ошибок, включая ошибки транзакций, является конструкция языка Transact-SQL TRY...CATCH.

**Пример**

```
USE MyDB;
GO
IF OBJECT_ID ( N'dbo.SaveTranExample', N'P' ) IS NOT NULL
DROP PROCEDURE dbo.SaveTranExample;
GO
CREATE PROCEDURE dbo.SaveTranExample ... -- Список параметров
AS
-- Надо проверить, была ли процедура вызвана из активной транзакции, и если да,
-- то устанавить точку сохранения для последующего использования.
-- @TranCounter = 0 означает, что активной транзакции нет и процедура явно начинает
-- локальную транзакцию.
-- @TranCounter > 0 означает, что активная транзакция началась еще до вызова процедуры.
DECLARE @TranCounter INT;
SET @TranCounter = @@TRANCOUNT;
IF @TranCounter > 0
      SAVE TRANSACTION ProcedureSave;
ELSE
      BEGIN TRANSACTION;
-- Пытаемся модифицировать базу данных.
BEGIN TRY
-- INSERT, UPDATE, DELETE
...
-- Здесь мы окажемся, если не произойдет никакой ошибки.
-- Если транзакция началась внутри процедуры, то выполнить COMMIT TRANSACTION.
--  Если транзакция началась до вызова процедуры, то выполнять COMMIT TRANSACTION
нельзя.
      IF @TranCounter = 0
-- @TranCounter = 0 означает, что никакая транзакция до вызова процедуры не начиналась.
-- Процедура должна завершить начатую в ней транзакцию.
            COMMIT TRANSACTION;
END TRY
BEGIN CATCH
-- Здесь мы окажемся, если произошла ошибка.
-- Надо определить, какой тип отката транзакции применять.
      IF @TranCounter = 0
-- Транзакция началась в теле процедуры.
-- Полный откат транзакции.
            ROLLBACK TRANSACTION;
      ELSE
-- Транзакция началась еще до вызова процедуры.
-- Нельзя отменять изменения, сделанные до вызова процедуры.
      IF XACT_STATE() <> -1
-- Если транзакция активна, то выполнить откат до точки сохранения.
            ROLLBACK TRANSACTION ProcedureSave;
-- Если транзакция актива, однако произошла ошибка, из-за которой транзакция
-- классифицируется как нефиксируемая,
--  то вернуться в точку вызова процедуры, где должен произойти откат внешней
транзакции.
-- В точку вызова процедуры передается информация об ошибках.
      DECLARE @ErrorMessage NVARCHAR(4000);
      DECLARE @ErrorSeverity INT;
      DECLARE @ErrorState INT;

      SELECT @ErrorMessage = ERROR_MESSAGE();
      SELECT @ErrorSeverity = ERROR_SEVERITY();
      SELECT @ErrorState = ERROR_STATE();

      RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);
END CATCH
```

```
GO
```

**Пояснения к примеру**

Для получения сведений об ошибках в области блока CATCH конструкции TRY…CATCH можно использовать следующие системные функции:

- ERROR_LINE() - возвращает номер строки, в которой произошла ошибка.
- ERROR_MESSAGE() - возвращает текст сообщения, которое будет возвращено приложению. Текст содержит значения таких подставляемых параметров, как длина, имена объектов или время.
- ERROR_NUMBER() - возвращает номер ошибки.
- ERROR_PROCEDURE() - возвращает имя хранимой процедуры или триггера, в котором произошла ошибка. Эта функция возвращает значение NULL, если данная ошибка не была совершена внутри хранимой процедуры или триггера.
- ERROR_SEVERITY() - возвращает уровень серьезности ошибки.
- ERROR_STATE() - возвращает состояние.

Инструкция RAISERROR позволяет вернуть приложению сообщение в формате системных ошибок или предупреждений.

**Управлением параллельным выполнением транзакций**

Когда множество пользователей одновременно пытаются модифицировать данные в базе данных, необходимо создать систему управления, которая защитила бы модификации, выполненные одним пользователем, от негативного воздействия модификаций, сделанных другими. Выделяют два типа управления параллельным выполнением:

- **Пессимистическое управление параллельным выполнением.**
- **Оптимистическое управление параллельным выполнением.**

Пессимистическое управление реализуется с помощью технологии блокировок, оптимистическое управления реализуется с помощью технологии версии строк.

Система блокировок не разрешает пользователям выполнить модификации, влияющие на других пользователей. Если пользователь выполнил какое-либо действие, в результате которого установлена блокировка, то другим пользователям не удастся выполнять действия, конфликтующие с установленной блокировкой, пока владелец не освободит ее. Пессимистическое управление используется главным образом в средах, где высока конкуренция за данные.

В случае управления на основе версий строк пользователи не блокируют данные при чтении. Во время обновления система следит, не изменил ли другой пользователь данные после их прочтения. Если другой пользователь модифицировал данные, генерируется ошибка. Как правило, получивший ошибку пользователь откатывает транзакцию и повторяет операцию снова. Этот способ управления в основном используется в средах с низкой конкуренцией за данные.

SQL Server поддерживает различные механизмы оптимистического и пессимистического управления параллельным выполнением. Пользователю предоставляется право определить тип управления параллельным выполнением, установив уровень изоляции транзакции для соединения и параметры параллельного выполнения для курсоров.

Если в случае пессимистического управления в СУБД не реализованы механизмы блокирования, то при одновременном чтении и изменении одних и тех же данных несколькими пользователями могут возникнуть следующие проблемы одновременного доступа:

**проблема последнего изменения** возникает, когда несколько пользователей изменяют одну и ту же строку, основываясь на ее начальном значении; тогда часть данных будет потеряна, т.к. каждая последующая транзакция перезапишет изменения, сделанные предыдущей. Выход из этой ситуации заключается в последовательном внесении изменений;

**проблема "грязного" чтения (Dirty Read)** возможна в том случае, если пользователь выполняет сложные операции обработки данных, требующие множественного изменения данных перед тем, как они обретут логически верное состояние. Если во время изменения данных другой пользователь будет считывать их, то может оказаться, что он получит логически неверную информацию. Для исключения подобных проблем необходимо производить считывание данных после окончания всех изменений;

**проблема неповторяемого чтения (Non-repeatable or Fuzzy Read)** является следствием неоднократного считывания транзакцией одних и тех же данных. Во время выполнения первой транзакции другая может внести в данные изменения, поэтому при повторном чтении первая транзакция получит уже иной набор данных, что приводит к нарушению их целостности или логической несогласованности;

**проблема чтения фантомов (Phantom)** появляется после того, как одна транзакция выбирает данные из таблицы, а

другая вставляет или удаляет строки до завершения первой. Выбранные из таблицы значения будут некорректны.

Для решения перечисленных проблем в стандарте ANSI SQL определены четыре уровня блокирования. Уровень изоляции транзакции определяет, могут ли другие (конкурирующие) транзакции вносить изменения в данные, измененные текущей транзакцией, а также может ли текущая транзакция видеть изменения, произведенные конкурирующими транзакциями, и наоборот. Каждый последующий уровень поддерживает требования предыдущего и налагает дополнительные ограничения:

**уровень 0** – запрещение «загрязнения» данных. Этот уровень требует, чтобы изменять данные могла только одна транзакция; если другой транзакции необходимо изменить те же данные, она должна ожидать завершения первой транзакции;

**уровень 1** – запрещение «грязного» чтения. Если транзакция начала изменение данных, то никакая другая транзакция не сможет прочитать их до завершения первой;

**уровень 2** – запрещение неповторяемого чтения. Если транзакция считывает данные, то никакая другая транзакция не сможет их изменить. Таким образом, при повторном чтении они будут находиться в первоначальном состоянии;

**уровень 3** – запрещение фантомов. Если транзакция обращается к данным, то никакая другая транзакция не сможет добавить новые или удалить имеющие строки, которые могут быть считаны при выполнении транзакции. Реализация этого уровня блокирования выполняется путем использования блокировок диапазона ключей. Подобная блокировка накладывается не на конкретные строки таблицы, а на строки, удовлетворяющие определенному логическому условию.

Проблемы, связанные с параллельным выполнением транзакций, разрешают, используя уровни изоляции транзакции. От уровня изоляции зависит то, в какой степени транзакция влияет на другие транзакции и испытывает влияние других транзакций. Более низкий уровень изоляции увеличивает возможность параллельного выполнения, но за это приходится расплачиваться согласованностью данных. Напротив, более высокий уровень изоляции гарантирует согласованность данных, но при этом страдает параллельное выполнение.

Стандарт ISO определяет следующие уровни изоляции:

- **read uncommitted** (самый низкий уровень, при котором транзакции изолируются до такой степени, чтобы только уберечь от считывания физически поврежденных данных);
- **read committed** (уровень по умолчанию);
- изоляция повторяющегося чтения **repeatable read**;
- изоляция упорядочиваемых транзакций **serializable** (самый высокий уровень, при котором транзакции полностью изолированы друг от друга).

SQL Server также поддерживает еще два уровня изоляции транзакций, использующих управление версиями строк.

- **read committed** с использованием управления версиями строк;
- уровень изоляции моментальных снимков **snapshot.**

Следующая таблица показывает побочные эффекты параллелизма, допускаемые различными уровнями изоляции.

| Уровень изоляции | «Грязное» чтение | Неповторяющееся чтение | Фантомное чтение |
|---|---|---|---|
| **read uncommitted** | Да | Да | Да |
| **read committed** | Нет | Да | Да |
| **repeatable read** | Нет | Нет | Да |
| **snapshot** | Нет | Нет | Нет |
| **serializable** | Нет | Нет | Нет |

Для установки уровня изоляции используется следующая инструкция SET TRANSACTION ISOLATION LEVEL, имеющая следующий синтаксис:

```
SET TRANSACTION ISOLATION LEVEL
{ READ UNCOMMITTED  | READ COMMITTED | REPEATABLE READ | SNAPSHOT | SERIALIZABLE }
[ ; ]
```

**Описание аргументов**

READ UNCOMMITTED - указывает, что инструкции могут считывать строки, которые были изменены другими транзакциями, но еще не были зафиксированы.

READ COMMITTED - указывает, что инструкции не могут считывать данные, которые были изменены другими транзакциями, но еще не были зафиксированы. Это предотвращает чтение «грязных» данных. Данные могут быть изменены другими транзакциями между отдельными инструкциями в текущей транзакции, результатом чего будет неповторяемое чтение или недействительные данные.

**Напоминание.** Поведение READ COMMITTED зависит от настройки аргумента базы данных READ_COMMITTED_SNAPSHOT (находится в состоянии OFF по умолчанию). REPEATABLE READ - указывает на то, что инструкции не могут считывать данные, которые были изменены, но еще не зафиксированы другими транзакциями, а также на то, что другие транзакции не могут изменять данные, читаемые текущей транзакцией, до ее завершения.

SNAPSHOT - указывает на то, что данные, считанные любой инструкцией транзакции, будут согласованы на уровне транзакции с версией данных, существовавших в ее начале. Транзакция распознает только те изменения, которые были зафиксированы до ее начала. Инструкции, выполняемые текущей транзакцией, не видят изменений данных, произведенных другими транзакциями после запуска текущей транзакции. Таким образом достигается эффект получения инструкциями в транзакции моментального снимка зафиксированных данных на момент запуска транзакции. Перед запуском транзакции, использующей уровень изоляции моментальных снимков, необходимо установить параметр базы данных ALLOW_SNAPSHOT_ISOLATION в ON. Если транзакция с уровнем изоляции моментального снимка обращается к данным из нескольких баз данных, аргумент ALLOW_SNAPSHOT_ISOLATION должен быть включен в каждой базе данных.

SERIALIZABLE - указывает следующее:
- Инструкции не могут считывать данные, которые были изменены другими транзакциями, но еще не были зафиксированы.
- Другие транзакции не могут изменять данные, считываемые текущей транзакцией, до ее завершения.
- Другие транзакции не могут вставлять новые строки со значениями ключа, которые входят в диапазон ключей, считываемых инструкциями текущей транзакции, до ее завершения.

**Замечания.**
1. Одновременно может быть установлен только один параметр уровня изоляции, который продолжает действовать для текущего соединения до тех пор, пока не будет явно изменен.
2. Уровни изоляции транзакции определяют тип блокировки, применяемый к операциям считывания.
3. В любой момент транзакции можно переключиться с одного уровня изоляции на другой, однако есть одно исключение. Это смена уровня изоляции на уровень изоляции SNAPSHOT. Такая смена приводит к ошибке и откату транзакции. Однако для транзакции, которая была начата с уровнем изоляции SNAPSHOT, можно установить любой другой уровень изоляции.
4. Когда для транзакции изменяется уровень изоляции, ресурсы, которые считываются после изменения, защищаются в соответствии с правилами нового уровня. Ресурсы, которые считываются до изменения, остаются защищенными в соответствии с правилами предыдущего уровня. Например, если для транзакции уровень изоляции изменяется с READ COMMITTED на SERIALIZABLE, то совмещаемые блокировки, полученные после изменения, будут удерживаться до завершения транзакции.
5. Если инструкция SET TRANSACTION ISOLATION LEVEL использовалась в хранимой процедуре или триггере, то при возврате управления из них уровень изоляции будет изменен на тот, который действовал на момент их вызова. Например, если уровень изоляции REPEATABLE READ устанавливается в пакете, а пакет затем вызывает хранимую процедуру, которая меняет уровень изоляции на SERIALIZABLE, при возвращении хранимой процедурой управления пакету, настройки уровня изоляции меняются назад на REPEATABLE READ.
6. Определяемые пользователем функции и типы данных среды CLR не могут выполнять инструкцию SET TRANSACTION ISOLATION LEVEL. Однако уровень изоляции можно переопределить с помощью табличной подсказки.

## Общие сведения о блокировках

Блокировка — это механизм, с помощью которого компонент Database Engine синхронизирует одновременный доступ нескольких пользователей к одному фрагменту данных. Прежде чем транзакция сможет распоряжаться текущим состоянием фрагмента данных, например, для чтения или изменения данных, она должна защититься от изменений этих данных другой транзакцией. Для этого транзакция запрашивает блокировку фрагмента данных. Существует несколько режимов блокировки, например общая или монопольная. Режим блокировки определяет уровень подчинения данных транзакции. Ни одна транзакция не может получить блокировку, которая противоречит другой блокировке этих данных, предоставленной другой транзакции. Если транзакция запрашивает режим блокировки, противоречащий предоставленной ранее блокировке тех же данных, экземпляр компонента Database Engine приостанавливает ее работу до тех пор, пока первая блокировка не освободится.

При изменении фрагмента данных транзакция удерживает блокировку, защищая изменения до конца транзакции. Продолжительность блокировки, полученной для защиты операций чтения, зависит от уровня изоляции транзакции. Все

блокировки, удерживаемые транзакцией, освобождаются после ее завершения (при фиксации или откате).
Приложения обычно не запрашивают блокировку напрямую. За управление блокировками отвечает внутренний компонент Database Engine, называемый диспетчером блокировок. Когда экземпляр компонента Database Engine обрабатывает инструкцию Transact-SQL, обработчик запросов компонента Database Engine определяет, к каким ресурсам требуется доступ. Обработчик запросов определяет, какие типы блокировок требуются для защиты каждого ресурса, в зависимости от типа доступа и уровня изоляции транзакции. Затем обработчик запросов запрашивает соответствующую блокировку у диспетчера блокировок. Диспетчер блокировок предоставляет блокировку, если она не противоречит блокировкам, удерживаемым другими транзакциями. Использование блокировки как механизма управления транзакциями может разрешить проблемы параллелизма. Блокировка позволяет запускать все транзакции в полной изоляции друг от друга, что позволяет одновременно выполнять несколько транзакций. Уровень, на котором транзакция готова к принятию противоречивых данных, называется уровнем изоляции. Чем выше уровень изоляции, тем ниже вероятность непоследовательности данных, однако при этом появляется такой недостаток, как сокращение параллелизма.

Каждая блокировка обладает тремя свойствами: *гранулярностью* (или размером блокировки), *режимом* (или типом блокировки) и *продолжительностью*.

**Гранулярность блокировок и иерархии блокировок**

SQL Server поддерживает мультигранулярную блокировку, позволяющую транзакции блокировать различные типы ресурсов. Чтобы уменьшить издержки применения блокировок, компонент Database Engine автоматически блокирует ресурсы на соответствующем задаче уровне. Блокировка при меньшей гранулярности, например на уровне строк, увеличивает параллелизм, но в то же время увеличивает и накладные расходы на обработку, поскольку при большом количестве блокируемых строк требуется больше блокировок. Блокировки на большем уровне гранулярности, например на уровне таблиц, обходится дорого в отношении параллелизма, поскольку блокировка целой таблицы ограничивает доступ ко всем частям таблицы других транзакций. Однако накладные расходы в этом случае ниже, поскольку меньше количество поддерживаемых блокировок.

Компонент Database Engine часто получает блокировки на нескольких уровнях гранулярности одновременно, чтобы полностью защитить ресурс. Такая группа блокировок на нескольких уровнях гранулярности называется иерархией блокировки. Например, чтобы полностью защитить операцию чтения индекса, экземпляру компоненту Database Engine может потребоваться получить разделяемые блокировки на строки и намеренные разделяемые блокировки на страницы и таблицу.

Следующая таблица содержит перечень ресурсов, которые могут блокироваться компонентом Database Engine.

| Ресурс | Описание |
|---|---|
| **RID** | Идентификатор строки, используемый для блокировки одной строки в куче |
| **KEY** | Блокировка строки в индексе, используемая для защиты диапазонов значений ключа в сериализуемых транзакциях. |
| **PAGE** | 8-килобайтовая (КБ) страница в базе данных, например страница данных или индекса. |
| **EXTENT** | Упорядоченная группа из восьми страниц, например страниц данных или индекса. |
| **HOBT** | Куча или сбалансированное дерево. Блокировка, защищающая индекс или кучу страниц данных в таблице, не имеющей кластеризованного индекса. |
| **TABLE** | Таблица полностью, включая все данные и индексы. |
| **FILE** | Файл базы данных. |
| **APPLICATION** | Определяемый приложением ресурс. |
| **METADATA** | Блокировки метаданных. |
| **ALLOCATION_UNIT** | Единица размещения. |
| **DATABASE** | База данных, полностью. |

По умолчанию блокировки укрупняются с уровня отдельных строк до целых страниц даже до уровня таблицы из соображений повышения производительности. Хотя в общем случае укрупнение считается хорошей штукой, оно может создавать проблемы, например, когда один SPID блокирует всю таблицу, препятствуя другому SPID работать ней. Можно настроить параметры блокировки на уровне строк и страниц. Такие блокировки по умолчанию разрешены для индексов.

**Режимы блокировки**

SQL Server блокирует ресурсы с помощью различных режимов блокировки, которые определяют доступ одновременных транзакций к ресурсам. В следующей таблице показаны режимы блокировки ресурсов, применяемые компонентом Database Engine.

| Режим блокировки | Описание |
|---|---|
| **Совмещаемая** | Используется для операций считывания, которые не меняют и не обновляют данные, такие как |

| | |
|---|---|
| **блокировка (S)** | инструкция SELECT. |
| **Блокировка обновления (U)** | Применяется к тем ресурсам, которые могут быть обновлены. Предотвращает возникновение распространенной формы взаимоблокировки, возникающей тогда, когда несколько сеансов считывают, блокируют и затем, возможно, обновляют ресурс. |
| **Монопольная блокировка (X)** | Используется для операций модификации данных, таких как инструкции INSERT, UPDATE или DELETE. Гарантирует, что несколько обновлений не будет выполнено одновременно для одного ресурса. |
| **Блокировка с намерением** | Используется для создания иерархии блокировок. Типы намеренной блокировки: с намерением совмещаемого доступа (IS), с намерением монопольного доступа (IX), а также совмещаемая с намерением монопольного доступа (SIX). |
| **Блокировка схемы** | Используется во время выполнения операции, зависящей от схемы таблицы. Типы блокировки схем: блокировка изменения схемы (Sch-S) и блокировка стабильности схемы (Sch-M). |
| **Блокировка массового обновления (BU)** | Используется, если выполняется массовое копирование данных в таблицу и указана подсказка TABLOCK. |
| **Диапазон ключей** | Защищает диапазон строк, считываемый запросом при использовании уровня изоляции сериализуемой транзакции. Запрещает другим транзакциям вставлять строки, что помогает запросам сериализуемой транзакции уточнять, были ли запросы запущены повторно. |

**Совмещаемые блокировки.** Совмещаемые (S) блокировки позволяют одновременным транзакциям считывать (SELECT) ресурс под контролем пессимистичного параллелизма. Пока для ресурса существуют совмещаемые (S) блокировки, другие транзакции не могут изменять данные. Совмещаемые блокировки (S) ресурса снимаются по завершении операции считывания, если только уровень изоляции транзакции не установлен на повторяющееся чтение или более высокий уровень, а также если совмещаемые блокировки (S) не продлены на все время транзакции с помощью указания блокировки.

**Блокировки обновления.** Блокировки обновления (U) предотвращают возникновение распространенной формы взаимоблокировки. В сериализуемой транзакции или транзакции операцией чтения с возможностью повторения транзакция считывает данные, запрашивает совмещаемую (S) блокировку на ресурс (страницу или строку), затем выполняет изменение данных, что требует преобразование блокировки в монопольную (X). Если две транзакции запрашивают совмещаемую блокировку на ресурс и затем пытаются одновременно обновить данные, то одна из транзакций пытается преобразовать блокировку в монопольную (X). Преобразование совмещаемой блокировки в монопольную потребует некоторого времени, поскольку монопольная блокировка для одной транзакции несовместима с совмещаемой блокировкой для другой транзакции. Начнется ожидание блокировки. Вторая транзакция попытается получить монопольную (X) блокировку для обновления. Поскольку обе транзакции выполняют преобразование в монопольную (X) блокировку и при этом каждая из транзакций ожидает, пока вторая снимет совмещаемую блокировку, то в результате возникает взаимоблокировка.
Чтобы избежать этой потенциальной взаимоблокировки, применяются блокировки обновления (U). Блокировку обновления (U) может устанавливать для ресурса одновременно только одна транзакция. Если транзакция изменяет ресурс, то блокировка обновления (U) преобразуется в монопольную (X) блокировку.

**Монопольные блокировки.** Монопольная (X) блокировка запрещает транзакциям одновременный доступ к ресурсу. Если ресурс удерживается монопольной (X) блокировкой, то другие транзакции не могут изменять данные. Операции считывания будут допускаться только при наличии указания NOLOCK или уровня изоляции незафиксированной операции чтения. Изменяющие данные инструкции, такие как INSERT, UPDATE или DELETE, соединяют как операции изменения, так и операции считывания. Чтобы выполнить необходимые операции изменения данных, инструкция сначала получает данные с помощью операций считывания. Поэтому, как правило, инструкции изменения данных запрашивают как совмещаемые, так и монопольные блокировки. Например, инструкция UPDATE может изменять строки в одной таблице, основанной на соединении данных из другой таблицы. В этом случае инструкция UPDATE кроме монопольной блокировки обновляемых строк запрашивает также совмещаемые блокировки для строк, считываемых в соединенной таблице.

**Блокировки с намерением.** В компоненте Компонент Database Engine блокировки с намерением применяются для защиты размещения совмещаемой (S) или монопольной (X) блокировки ресурса на более низком уровне иерархии. Блокировки с намерением называются так потому, что их получают до блокировок более низкого уровня, то есть они обозначают намерение поместить блокировку на более низком уровне.
Блокировка с намерением выполняет две функции:

- предотвращает изменение ресурса более высокого уровня другими транзакциям таким образом, что это сделает недействительной блокировку более низкого уровня;
- повышает эффективность компонента Компонент Database Engine при распознавании конфликтов блокировок на более высоком уровне гранулярности.

Например, в таблице требуется блокировка с намерением совмещаемого доступа до того, как для страниц или строк этой таблицы будет запрошена совмещаемая (S) блокировка. Если задать блокировку с намерением на уровне таблицы, то другим транзакциям будет запрещено получать монопольную (X) блокировку для таблицы, содержащей эту страницу.

Блокировка с намерением повышает производительность, поскольку компонент Компонент Database Engine проверяет наличие таких блокировок только на уровне таблицы, чтобы определить, может ли транзакция безопасно получить для этой таблицы совмещаемую блокировку. Благодаря этому нет необходимости проверять блокировки в каждой строке и на каждой странице, чтобы убедиться, что транзакция может заблокировать всю таблицу.

**Блокировки схем.** В компоненте Компонент Database Engine блокировка изменения схемы (Sch-M) применяется с операциями языка DDL для таблиц, например при добавлении столбца или очистке таблицы. Пока удерживается блокировка изменения схемы (Sch-M), одновременный доступ к таблице запрещен. Это означает, что любые операции вне блокировки изменения схемы (Sch-M) будут запрещены до снятия блокировки.

- Блокировка изменения схемы (Sch-M) применяется с некоторыми операциями языка обработки данных, например усечением таблиц, чтобы предотвратить одновременный доступ к таблице.
- Блокировка стабильности схемы (Sch-S) применяется компонентом Компонент Database Engine при компиляции и выполнении запросов. Блокировка стабильности схемы (Sch-S) не влияет на блокировки транзакций, включая монопольные (X) блокировки. Поэтому другие транзакции (даже транзакции с монопольной блокировкой (X) для таблицы) могут продолжать работу во время компиляции запроса. Однако одновременные операции DDL и DML, которые запрашивают блокировки изменения схемы (Sch-M), не могут выполняться над таблицей.

**Блокировки массового обновления.** Блокировка массового обновления (BU) позволяет поддерживать несколько одновременных потоков массовой загрузки данных в одну и ту же таблицу и при этом запрещать доступ к таблице любым другим процессам, отличным от массовой загрузки данных. Компонент Database Engine использует блокировки массового обновления (BU), если выполняются два следующих условия. Используется инструкция Transact-SQL BULK INSERT, функция OPENROWSET(BULK) или одна из таких команд массовой вставки API, как .NET SqlBulkCopy, OLEDB Fast Load APIs или ODBC Bulk Copy APIs, для массового копирования данных в таблицу. Выделено указание TABLOCK или установлен параметр таблицы table lock on bulk load с помощью хранимой процедуры sp_tableoption.

**Блокировки диапазона ключа.** Блокировки диапазона ключей защищают диапазон строк, неявно включенный в набор записей, считываемый инструкцией Transact-SQL при использовании уровня изоляции сериализуемых транзакций. Блокировка диапазона ключей предотвращает фантомные чтения. Кроме того, защита диапазона ключей между строк предотвращает фантомную вставку или удаление из набора записи, к которому получает доступ транзакция.

**Совместимость блокировок**

Совместимость блокировок определяет, могут ли несколько транзакций одновременно получить блокировку одного и того же ресурса. Если ресурс уже блокирован другой транзакцией, новая блокировка может быть предоставлена только в том случае, если режим запрошенной блокировки совместим с режимом существующей. В противном случае транзакция, запросившая новую блокировку, ожидает освобождения ресурса, пока не истечет время ожидания существующей блокировки.

Например, с монопольными блокировками не совместим ни один из режимов блокировки. Пока удерживается монопольная (X) блокировка, больше ни одна из транзакций не может получить блокировку ни одного из типов (разделяемую, обновления или монопольную) на этот ресурс, пока не будет освобождена монопольная (X) блокировка. И наоборот, если к ресурсу применяется разделяемая (S) блокировка, другие транзакции могут получать разделяемую блокировку или блокировку обновления (U) на этот элемент, даже если не завершилась первая транзакция. Тем не менее, другие транзакции не могут получить монопольную блокировку до освобождения разделяемой. Полная матрица совместимости блокировок приводится в справочниках.

**Продолжительность блокировки**

Продолжительность блокировок также определяется типами запросов. Когда в рамках транзакции запрос не выполняется, и не используются подсказки блокировки, блокировки для выполнения инструкций SELECT выполняются только на время чтения ресурса, но не во время запроса. Блокировки для инструкций INSERT, UPDATE и DELETE сохраняются на все время выполнения запроса. Это помогает гарантировать согласованность данных и позволяет SQL Server откатывать запросы в случае необходимости.

Когда запрос выполняется в рамках транзакции, продолжительность блокировки определяется тремя факторами:

- типом запроса;
- уровнем изоляции транзакции;
- наличием или отсутствием подсказок блокировки.

Кратковременные (locking) и обычные (blocking) блокировки — нормальное явление в реляционных базах данных, но они могут ухудшать производительность, если блокировки ресурсов сохраняются на протяжении длительного времени. Производительность также страдает, когда, заблокировав ресурс, SPID не в состоянии освободить его.

В первом случае проблема обычно разрешается через какое-то время, так как SPID, в конечном счете, освобождает блокировку, но угроза деградации производительности остается вполне реальной. Проблемы с блокировкой второго типа

могут вызвать серьезное падение производительности, но, к счастью, они легко обнаруживаются при мониторинге SQL Server на предмет кратковременных и обычных блокировок.

**Эскалация блокировок и их влияние на работу системы**

**Эскалация** (**lock escalation**) связана с тем, что по мере увеличения, количества отдельных малых заблокированных объектов накладные расходы, связанные с их поддержкой, начинают значительно сказываться на производительности. Блокировки длятся дольше, что приводит к спорным ситуациям – чем дольше существует блокировка, тем выше вероятность обращения к заблокированному объекту со стороны другой транзакции. Очевидно, что на некотором этапе потребуется выполнить объединение (увеличение масштаба) блокировок, чем собственно и занимается диспетчер блокировок. В инструкции ALTER TABLE предусмотрена опция вида SET (LOCK_ESCALATION = { AUTO | TABLE | DISABLE } ), которая указывает разрешенные методы укрупнения блокировки для таблицы.

**Задание определенного типа блокировки в запросе**

Для повышения эффективности исполнения запросов и получения дополнительного контроля над блокировками в запросе можно давать подсказки (hints или хинты), указывая их непосредственно за именем таблицы, которая нуждается в том или ином типе блокировки. Существуют следующие параметры оптимизатора запросов:

SERIALIZABLE/HOLDLOCK
READUNCOMMITTED/NOLOCK
READCOMMITTED
REPEATABLEREAD
READPAST
ROWLOCK
PAGLOCK
TABLOCK
TABLOCKX
UPDLOCK
XLOCK

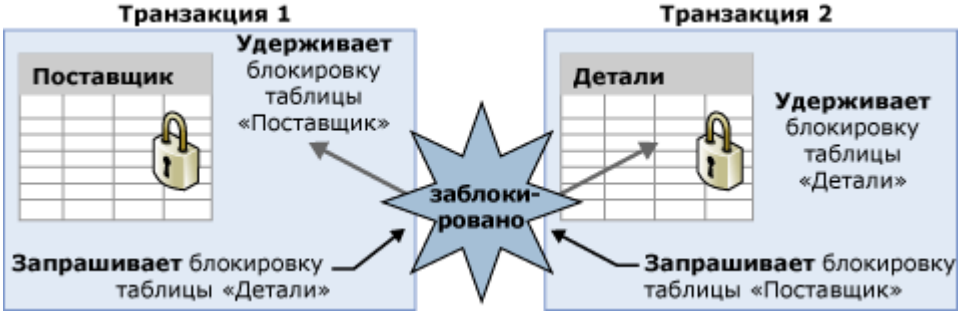
Пример задания эксклюзивной табличной блокировки для таблицы Orders (вместо блокировки на уровне ключей или строк, которую может предложить оптимизатор) может быть таким:

```
SELECT *
FROM Orders AS o WITH (TABLOCKX) JOIN [Order Details] AS od
ON o.OrderID = od.OrderID
```

Поскольку оптимизатор запросов обычно выбирает лучший план выполнения запроса, использовать подсказки рекомендуется только опытным разработчикам и администраторам баз данных в самом крайнем случае. Сведения об активных блокировках на текущий момент времени можно получить с помощью системной хранимой процедуры sp_lock.

**Взаимоблокировки транзакций**

**Взаимоблокировки** или **тупиковые ситуации** (**deadlocks**) возникают тогда, когда одна из транзакций не может завершить свои действия, поскольку вторая транзакция заблокировала нужные ей ресурсы, а вторая в то же время ожидает освобождения ресурсов первой транзакцией. На рисунке транзакция T1 зависит от транзакции T2 для ресурса блокировки таблицы **Детали**. Аналогично транзакция T2 зависит от транзакции T1 для ресурса блокировки таблицы **Поставщик**. Так как эти зависимости из одного цикла, возникает взаимоблокировка транзакций T1 и T2.




Транзакция T1 не может завершиться до того, как завершится транзакция T2, а транзакция T2 заблокирована транзакцией T1. Такое условие также называется цикличной зависимостью: транзакция T1 зависит от транзакции T2, а транзакция T2

зависит от транзакции T1 и этим замыкает цикл. Обе транзакции находятся в состоянии взаимоблокировки и будут всегда находиться в состоянии ожидания, если взаимоблокировка не будет разрушена внешним процессом.

Взаимоблокировки часто путают с обычными блокировками. Если транзакция запрашивает блокировку на ресурс, заблокированный дугой транзакцией, то запрашивающая транзакция ожидает до тех пор, пока блокировка не освобождается. По умолчанию время ожидания транзакций SQL Server не ограничено, если только не установлен параметр LOCK_TIMEOUT.

Значение -1 (по умолчанию) указывает на отсутствие времени ожидания (то есть инструкция будет ждать всегда). Когда ожидание блокировки превышает значение времени ожидания, возвращается ошибка. Значение «0» означает, что ожидание отсутствует, а сообщение возвращается, как только встречается блокировка. Функция @@LOCK_TIMEOUT возвращает значение времени ожидания блокировки в миллисекундах для текущего сеанса.

Запрашивающая транзакция блокируется, но не устанавливается в состояние взаимоблокировки, потому что запрашивающая транзакция ничего не сделала, чтобы заблокировать транзакцию, владеющую блокировкой. Наконец, владеющая транзакция завершится и освободит блокировку, и затем запрашивающая транзакция получит блокировку и продолжится.

**Как SQL Server обнаруживает взаимоблокировки?**

Обнаружение взаимоблокировки выполняется потоком диспетчера блокировок, который периодически производит поиск по всем задачам в экземпляре компонента Database Engine. Следующие пункты описывают процесс поиска:

- Значение интервала поиска по умолчанию составляет 5 секунд.
- Если диспетчер блокировок находит взаимоблокировки, интервал обнаружения взаимоблокировок снижается с 5 секунд до 100 миллисекунд в зависимости от частоты взаимоблокировок.
- Если поток диспетчера блокировки прекращает поиск взаимоблокировок, компонент Database Engine увеличивает интервал до 5 секунд.
- Если взаимоблокировка была только что найдена, предполагается, что следующие потоки, которые должны ожидать блокировки, входят в цикл взаимоблокировки. Первая пара элементов, ожидающих блокировки, после того как взаимоблокировка была обнаружена, запускает поиск взаимоблокировок вместо того, чтобы ожидать следующий интервал обнаружения взаимоблокировки. Например, если текущее значение интервала равно 5 секунд и была обнаружена взаимоблокировка, следующий ожидающий блокировки элемент немедленно приводит в действие детектор взаимоблокировок. Если этот ожидающий блокировки элемент является частью взаимоблокировки, она будет обнаружена немедленно, а не во время следующего поиска взаимоблокировок.
- Компонент Database Engine обычно выполняет только периодическое обнаружение взаимоблокировок. Так как число взаимоблокировок, произошедших в системе, обычно мало, периодическое обнаружение взаимоблокировок помогает сократить издержки от взаимоблокировок в системе.
- Если монитор блокировок запускает поиск взаимоблокировок для определенного потока, он идентифицирует ресурс, ожидаемый потоком. После этого монитор блокировок находит владельцев определенного ресурса и рекурсивно продолжает поиск взаимоблокировок для этих потоков до тех пор, пока не найдет цикл. Цикл, определенный таким способом, формирует взаимоблокировку.
- После обнаружения взаимоблокировки компонент Database Engine завершает взаимоблокировку, выбрав один из потоков в качестве жертвы взаимоблокировки. Компонент Database Engine прерывает выполняемый в данный момент пакет потока, производит откат транзакции жертвы взаимоблокировки и возвращает приложению ошибку 1205. Откат транзакции жертвы взаимоблокировки снимает все блокировки, удерживаемые транзакцией. Это позволяет транзакциям потоков разблокироваться, и продолжить выполнение. Ошибка 1205 жертвы взаимоблокировки записывает в журнал ошибок сведения обо всех потоках и ресурсах, затронутых взаимоблокировкой. Сведения об ошибках см. в

C:\Program Files\Microsoft SQL Server\MSSQL10_50.SQLEXPRESS\MSSQL\Log\ERRORLOG или ERRORLOG.n

[
Каждые пять секунд SQL Server проверяет состояние текущих транзакций на предмет наличия блокировок, которые ожидают своей очереди. В случае наличия подобных блокировок, SQL Server берет их на заметку. Через следующие пять секунд производится повторная проверка всех открытых блокировок, и если одна из отмеченных блокировок по-прежнему находится в состоянии ожидания, выполняется рекурсивная проверка всех открытых транзакций на предмет существования в очереди замкнутых циклов. Если такой цикл обнаружен, в нем выбираются одна или несколько «жертв» взаимоблокировки.
]

**Как выбирается жертва взаимоблокировки?**

По умолчанию в качестве жертвы взаимоблокировки выбирается сеанс, выполняющий ту транзакцию, откат которой потребует меньше всего затрат. В качестве альтернативы пользователь может указать приоритет сеансов, используя инструкцию SET DEADLOCK_PRIORITY. DEADLOCK_PRIORITY может принимать значения LOW, NORMAL или HIGH или в качестве альтернативы может принять любое целочисленное значение на отрезке [-10..10]. По умолчанию DEADLOCK_PRIORITY устанавливается на значение NORMAL. Если у двух сеансов имеются различные приоритеты, то в качестве жертвы взаимоблокировки будет выбран сеанс с более низким приоритетом. Если у обоих сеансов установлен одинаковый приоритет, то в качестве жертвы взаимоблокировки будет выбран сеанс, откат которого потребует наименьших затрат. Если сеансы, вовлеченные в цикл взаимоблокировки, имеют один и тот же приоритет и одинаковую стоимость, то жертва взаимоблокировки выбирается случайным образом.

**Мониторинг транзакций и блокировок**

В SQL Server для получения информации о состоянии сервера применяются динамические административные представления и функции, расположенные в схеме sys. Их имена следуют такому соглашению по именованию: dm_* (от англ. dynamic_management). Есть два типа динамических административных представлений и функций:

- **области сервера** - для них необходимо разрешение VIEW SERVER STATE на сервере;
- **области базы данных** - для них необходимо разрешение VIEW DATABASE STATE на базе данных.

Динамические административные представления и функции разбиты на категории: 20 в SQL Server 2012. Одна из категория связана с транзакциями. В эту категорию входят 10 представлений и функций:

**sys.dm_tran_active_snapshot_database_transactions** - возвращает виртуальную таблицу всех активных транзакций, формирующих или потенциально получающих доступ к версиям строк

**sys.dm_tran_active_transactions** - возвращает данные о транзакциях для экземпляра SQL Server.

**sys.dm_tran_current_snapshot** - возвращает виртуальную таблицу, которая отображает все активные транзакции в момент запуска текущей транзакции моментального снимка.

**sys.dm_tran_current_transaction** - возвращает строку, которая отображает сведения о состоянии транзакции в текущей сессии.

**sys.dm_tran_database_transactions** - возвращает сведения о транзакциях на уровне базы данных.

**sys.dm_tran_locks** - возвращает строки, представляющие текущий активный запрос диспетчеру блокировок на блокировку, которая была наложена или находится в ожидании получения.

**sys.dm_tran_session_transactions** - возвращает информацию о том, как выполняется каждая транзакция во всех сеансах.

**sys.dm_tran_top_version_generators** - возвращает информацию о том, какие версии строк созданы в базах данных.

**sys.dm_tran_transactions_snapshot** - возвращает виртуальную таблицу номеров транзакций, активных при запуске каждой транзакции моментальных снимков.

**sys.dm_tran_version_store** - возвращает виртуальную таблицу, в которой отображаются все записи версий в хранилище версий.

**Рекомендации по предотвращению взаимоблокировок**

Невозможно полностью исключить возникновение взаимоблокировок в сложных программных системах, но с практической точки зрения можно сделать их вероятность настолько малой, что данный вопрос перестанет быть актуальным. Для того чтобы избежать появления взаимоблокировок или, по крайней мере, свести их количество к минимуму, надо следовать простым правилам:

- обращайтесь к объектам в одном и том же порядке;
- делайте транзакции как можно более короткими и размещайте их в одном пакете;
- используйте наименьший возможный уровень изоляции транзакций;
- не допускайте разрывов внутри транзакции (со стороны пользователя либо в результате разделения пакета);
- в контролируемой среде используйте связанные подключения.

## Общие сведения об управлении версиями строк

**Моментальный снимок SNAPSHOT.** Указывает на то, что данные, считанные любой инструкцией транзакции, будут согласованы на уровне транзакции с версией данных, существовавших в ее начале. Транзакция распознает только те изменения, которые были зафиксированы до ее начала. Инструкции, выполняемые текущей транзакцией, не видят изменений данных, произведенных другими транзакциями после запуска текущей транзакции. Таким образом достигается эффект получения инструкциями в транзакции моментального снимка зафиксированных данных на момент запуска транзакции.

Транзакции моментальных снимков не требуют блокировки при считывании данных, за исключением случаев восстановления базы данных. Считывание данных транзакциями моментальных снимков не блокирует запись данных другими транзакциями. Транзакции, осуществляющие запись данных, не блокируют считывание данных транзакциями моментальных снимков. На этапе отката восстановления базы данных транзакция моментальных снимков запросит блокировку, если будет предпринята попытка считывания данных, заблокированных другой откатываемой транзакцией. Блокировка транзакции моментальных снимков сохраняется до завершения отката. Блокировка снимается

сразу после предоставления. Перед запуском транзакции, использующей уровень изоляции моментальных снимков, необходимо установить параметр базы данных ALLOW_SNAPSHOT_ISOLATION в ON. Если транзакция с уровнем изоляции моментального снимка обращается к данным из нескольких баз данных, аргумент ALLOW_SNAPSHOT_ISOLATION должен быть включен в каждой базе данных.

Невозможно изменить на уровень изоляции моментального снимка уровень изоляции транзакции, запущенной с другим уровнем изоляции; в этом случае транзакция будет прервана. Если транзакция запущена с уровнем изоляции моментальных снимков, ее уровень изоляции можно изменять. Транзакция начинается в момент первого доступа к данным.

Транзакция, работающая с уровнем изоляции моментального снимка, может просматривать внесенные ею изменения. Например, если транзакция выполняет инструкцию UPDATE, а затем инструкцию SELECT для одной и той же таблицы, измененные данные будут включены в результирующий набор.

**Моментальный снимок с уровнем изоляции READ COMMITED.** Если параметр READ_COMMITTED_SNAPSHOT находится в состоянии ON, компонент Database Engine использует управление версиями строк для представления каждой инструкции согласованного на уровне транзакций моментального снимка данных в том виде, который они имели на момент начала выполнения инструкции. Для защиты данных от обновления другими транзакциями блокировки не используются.

---

## Дополнительный материал

**Обработка ошибок в T-SQL**

@@ERROR - Возвращает номер ошибки для последней выполненной инструкции Transact-SQL. Возвращает 0, если в предыдущей инструкции Transact-SQL не возникли ошибки. Просмотреть текст, связанный с номером ошибки @@ERROR можно в sys.messages.

@@ROWCOUNT - Возвращает число строк, затронутых при выполнении последней инструкции Transact-SQL.

@@TRANCOUNT - Возвращает число инструкций BEGIN TRANSACTION, выполненных в текущем соединении.

```
BEGIN TRY
    { sql_statement | statement_block }
END TRY
BEGIN CATCH
        [ { sql_statement | statement_block } ]
END CATCH
[ ; ]
```

Реализация обработчика ошибок на языке Transact-SQL. В области блока CATCH для получения сведений об ошибке, приведшей к выполнению данного блока CATCH, можно использовать следующие системные функции:
ERROR_NUMBER() возвращает номер ошибки.
ERROR_SEVERITY() возвращает степень серьезности ошибки.
ERROR_STATE() возвращает код состояния ошибки.
ERROR_PROCEDURE() возвращает имя хранимой процедуры или триггера, в котором произошла ошибка.
ERROR_LINE() возвращает номер строки, которая вызвала ошибку, внутри подпрограммы.
ERROR_MESSAGE() возвращает полный текст сообщения об ошибке. Текст содержит значения подставляемых параметров, таких как длина, имена объектов или время.

XACT_STATE() - Возвращает состояние транзакции текущего выполняемого запроса.
1 Текущий запрос содержит активную пользовательскую транзакцию. Запрос может выполнять любые действия, включая запись данных и фиксирование транзакции.
0 У текущего запроса нет активной пользовательской транзакции.
-1 В текущем запросе есть активная транзакция, однако произошла ошибка, из-за которой транзакция классифицируется как нефиксируемая. Запросу не удается зафиксировать транзакцию или выполнить откат до точки сохранения; можно только запросить полный откат транзакции. Запрос не может выполнить никакие операции записи, пока не будет проведен откат транзакции. До отката транзакции запрос может выполнять только операции считывания. После отката транзакции запросу будут доступны как операции считывания, так и операции записи, а также запуск новых транзакций. После завершения работы пакета компонент Database Engine автоматически выполнит откат любых активных нефиксируемых транзакций. Если при переходе транзакции в нефиксируемое состояние не было отправлено сообщение об ошибке, после завершения выполнения пакета сообщение об ошибке будет отправлено клиентскому приложению. Сообщение показывает, что обнаружены нефиксируемые транзакции и выполнен откат.

SET XACT_ABORT { ON | OFF } - Указывает, выполняет ли SQL Server автоматический откат текущей транзакции, если инструкция языка Transact-SQL вызывает ошибку выполнения. Если выполнена инструкция SET XACT_ABORT ON и инструкция языка Transact-SQL вызывает ошибку, вся транзакция завершается и выполняется ее откат. Если выполнена инструкция SET XACT_ABORT OFF, в некоторых случаях выполняется откат только вызвавшей ошибку инструкции языка Transact-SQL, а обработка транзакции продолжается.

**Обработка ошибок в транзакциях (на примере)**

```
USE tempdb
GO

CREATE TABLE dbo.Employees
(
	empid INT NOT NULL,
	empname VARCHAR(25) NOT NULL,
	mgrid INT NULL,
	CONSTRAINT PK_Employees PRIMARY KEY(empid),
	CONSTRAINT CHK_Employees_empid CHECK(empid > 0),
	CONSTRAINT FK_Employees_Employees FOREIGN KEY(mgrid) REFERENCES Employees(empid)
);

-- Запускаем следующий пример дважды:
BEGIN TRY
	INSERT INTO dbo.Employees(empid, empname, mgrid)
	VALUES(1, 'Emp1', NULL);
	PRINT 'INSERT succeeded.';
END TRY
BEGIN CATCH
	PRINT 'INSERT failed.';
	/* handle error here */
END CATCH

-- 1
-- (1 row(s) affected)
-- INSERT succeeded.

-- 2
-- INSERT failed.

SELECT * FROM dbo.Employees
GO


CREATE PROC dbo.AddEmp @empid AS INT, @empname AS VARCHAR(25), @mgrid AS INT
AS
DECLARE @tc AS INT = @@TRANCOUNT; -- Сохраняем количество внешних транзакций
IF @tc > 0 -- Если уже есть активная транзакция, то создаем точку сохранения S1
	SAVE TRAN S1;
ELSE
BEGIN TRAN -- Если активных транзакций нет, то открываем новую транзакцию
	BEGIN TRY;
		-- Модифицируем данные
		INSERT INTO dbo.Employees(empid, empname, mgrid) VALUES(@empid, @empname, @mgrid);
		IF @tc = 0 -- Если процедура открыла транзакцию, то она ее и фиксирует
			COMMIT TRAN;
	END TRY
	BEGIN CATCH
		IF @tc = 0 -- Если процедура открыла транзакцию, то ...
		BEGIN
			IF XACT_STATE() <> 0 -- Если в текущем запросе есть активная транзакция, однако произошла
ошибка, то
			BEGIN
				PRINT 'Откат Транзакции, открытой Процедурой.';
```

```
                        ROLLBACK TRAN
                END
        END
        ELSE -- Если процедура не открывала транзакцию (т.е. существовали внешние транзакции), то
        BEGIN
                IF XACT_STATE() = 1 -- Если текущий запрос содержит активную пользовательскую транзакцию, то
                BEGIN
                        PRINT 'Процедура была вызвана в открытой Транзакции. Откат до точки сохранения.';
                        ROLLBACK TRAN S1
                END
                ELSE IF XACT_STATE() = -1 -- Если в текущем запросе есть активная транзакция, однако произошла ошибка, из-за которой транзакция классифицируется как нефиксируемая, то
                        PRINT 'Процедура была вызвана в открытой Транзакции, но не зафиксирована. Передать обработку ошибки в точку вызова процедуры.'
        END
        -- Создаем сообщение об ошибке, чтобы 'caller' решал, что делать с отказом в процедуре
        DECLARE @ErrorMessage NVARCHAR(400) = ERROR_MESSAGE(), @ErrorSeverity INT = ERROR_SEVERITY(), @ErrorState INT = ERROR_STATE();
        RAISERROR (@ErrorMessage, @ErrorSeverity, @ErrorState);
    END CATCH
GO

-- Чтобы проверить процедуру, сначала очистить таблицу 'Employees':
TRUNCATE TABLE dbo.Employees;
GO

-- Затем выполняем следующий код дважды, но не в явной транзакции:
EXEC AddEmp @empid = 1, @empname = 'Emp1', @mgrid = NULL;

-- 1)
-- (1 row(s) affected)

-- 2)
-- (0 row(s) affected)
-- Откат Транзакции, открытой Процедурой.
-- Msg 50000, Level 14, State 1, Procedure AddEmp, Line 35
-- Violation of PRIMARY KEY constraint 'PK_Employees'. Cannot insert duplicate key in object 'dbo.Employees'.

-- Теперь запустите процедуру снова, но на этот раз в рамках явной транзакции:
BEGIN TRAN
EXEC AddEmp @empid = 1, @empname = 'Emp1', @mgrid = NULL;
ROLLBACK

-- (0 row(s) affected)
-- Процедура была вызвана в открытой Транзакции. Откат до точки сохранения.
-- Msg 50000, Level 14, State 1, Procedure AddEmp, Line 35
-- Violation of PRIMARY KEY constraint 'PK_Employees'. Cannot insert duplicate key in object 'dbo.Employees'.

-- Последнее испытание
SET XACT_ABORT ON;
BEGIN TRAN
EXEC AddEmp @empid = 1, @empname = 'Emp1', @mgrid = NULL;
-- Обработка ошибки ...
ROLLBACK
SET XACT_ABORT OFF;

-- (0 row(s) affected)
-- Процедура была вызвана в открытой Транзакции, но не зафиксирована. Передать обработку ошибки в точку вызова процедуры.
-- Msg 50000, Level 14, State 1, Procedure AddEmp, Line 35
-- Violation of PRIMARY KEY constraint 'PK_Employees'. Cannot insert duplicate key in object 'dbo.Employees'.
```

**Транзакции и точки сохранения**

```
USE tempdb;
GO

IF OBJECT_ID('dbo.Sequence', 'U') IS NOT NULL DROP TABLE dbo.Sequence;
GO

CREATE TABLE dbo.Sequence(val INT IDENTITY);
GO

IF OBJECT_ID('dbo.GetSequence', 'P') IS NOT NULL DROP PROC dbo.GetSequence;
GO

CREATE PROC dbo.GetSequence @val AS INT OUTPUT
AS
	BEGIN TRAN
		SAVE TRAN S1;
		INSERT INTO dbo.Sequence DEFAULT VALUES;
		SET @val = SCOPE_IDENTITY()
		ROLLBACK TRAN S1;
	COMMIT TRAN
GO

SELECT * FROM dbo.Sequence
GO
-- Пусто

DECLARE @key AS INT;
EXEC dbo.GetSequence @val = @key OUTPUT;
SELECT @key;
GO

SELECT * FROM dbo.Sequence
-- Снова пусто,
-- @key = 1, 2, 3, и т.д.
GO
```

Функции SCOPE_IDENTITY, IDENT_CURRENT и @@IDENTITY идентичны друг другу, поскольку возвращают значения, вставленные в столбцы идентификаторов.

**Взаимоблокировки**

```
SET NOCOUNT ON;

USE tempdb
GO

IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;
IF OBJECT_ID('dbo.T2', 'U') IS NOT NULL DROP TABLE dbo.T2;
GO

CREATE TABLE dbo.T1
(
keycol INT NOT NULL PRIMARY KEY,
col1 INT NOT NULL,
col2 VARCHAR(50) NOT NULL
);
INSERT INTO dbo.T1(keycol, col1, col2) VALUES
(1, 101, 'A'),
(2, 102, 'B'),
(3, 103, 'C');

CREATE TABLE dbo.T2
(
```

```
keycol INT NOT NULL PRIMARY KEY,
col1 INT NOT NULL,
col2 VARCHAR(50) NOT NULL
);
INSERT INTO dbo.T2(keycol, col1, col2) VALUES
(1, 201, 'X'),
(2, 202, 'Y'),
(3, 203, 'Z');
GO

SELECT * FROM dbo.T1
SELECT * FROM dbo.T2
GO

BEGIN TRAN
INSERT INTO dbo.T1(keycol, col1, col2) VALUES(4, 101, 'C');
INSERT INTO dbo.T2(keycol, col1, col2) VALUES(4, 201, 'X');
COMMIT TRAN
GO

SELECT * FROM dbo.T1
SELECT * FROM dbo.T2
GO

-- connection 1:

SET NOCOUNT ON;
USE tempdb;
GO
BEGIN TRAN
UPDATE dbo.T1 SET col1 = col1 + 1 WHERE keycol = 2;

-- connection 2:

SET NOCOUNT ON;
USE tempdb;
GO
BEGIN TRAN
UPDATE dbo.T2 SET col1 = col1 + 1 WHERE keycol = 2;

-- connection 1, which attempts to read data from T2:

SELECT col1 FROM dbo.T2 WHERE keycol = 2;
COMMIT TRAN

-- Ждем

-- connection 2, attempting to query the data from T1:

SELECT col1 FROM dbo.T1 WHERE keycol = 2;
COMMIT TRAN

Msg 1205, Level 13, State 51, Server HOMEPC-ПК\SQLEXPRESS, Line 1
Transaction (Process ID 58) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.

-- Только после этого в connection 1 имеем:

col1
-----------
      202
```

**Тупики вызванные отсутствием индексов**

```
BEGIN TRAN
UPDATE dbo.T1 SET col2 = col2 + 'A' WHERE col1 = 101;

BEGIN TRAN
UPDATE dbo.T2 SET col2 = col2 + 'B' WHERE col1 = 203;

SELECT col2 FROM dbo.T2 WHERE col1 = 201;
COMMIT TRAN

-- Ждем

SELECT col2 FROM dbo.T1 WHERE col1 = 103;
COMMIT TRAN
```

Конечно, возникает взаимоблокировка, и один из процессов "выбран", как жертва тупика (подключение 2 в данном случае), и вы получите следующее сообщение об ошибке:

```
Msg 1205, Level 13, State 51, Server HOMEPC-ПК\SQLEXPRESS, Line 1
Transaction (Process ID 58) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.
```

**Для предотвращения подобных тупиков в будущем, создаем следующие индексы:**

```
CREATE INDEX idx_col1 ON dbo.T1(col1);
CREATE INDEX idx_col1 ON dbo.T2(col1);

BEGIN TRAN
UPDATE dbo.T1 SET col2 = col2 + 'A' WHERE col1 = 101;

BEGIN TRAN
UPDATE dbo.T2 SET col2 = col2 + 'B' WHERE col1 = 203;

SELECT col2 FROM dbo.T2 WITH (index = idx_col1) WHERE col1 = 201;
COMMIT TRAN

SELECT col2 FROM dbo.T1 WITH (index = idx_col1) WHERE col1 = 103;
COMMIT TRAN
```

Запрос выполняется успешно, и транзакция фиксируется.

```
DROP INDEX idx_col1 ON dbo.T1(col1);
DROP INDEX idx_col1 ON dbo.T2(col1);
```

---

## Приложение

**Инструкции SET и управление транзакциями**

Язык T-SQL имеет несколько инструкций SET, которые изменяют параметры текущего сеанса:

1. Инструкции блокировки:
2. Инструкции управления транзакциями:

SET DEADLOCK_PRIORITY определяет относительную важность продолжения текущего сеанса, если произошла взаимоблокировка с другим сеансом. Синтаксис:

SET DEADLOCK_PRIORITY { LOW | NORMAL | HIGH | *числовой_приоритет* | *@символьная_переменная* | *@числовая_переменная* },

где *числовая-переменная* принимает значение из диапазона -10..10. NORMAL - приоритет по умолчанию. LOW соответствует -5, NORMAL - 0, HIGH - 5. Например, SET DEADLOCK_PRIORITY LOW означает, что текущий сеанс будет выбран в качестве жертвы в случае взаимоблокировки с другим сеансом, если другой сеанс входит в цепочку взаимного блокирования с приоритетом NORMAL, HIGH или равным целочисленному значению не менее -5. Текущий сеанс может быть выбран в качестве жертвы, если другому сеансу назначен приоритет LOW или равный целочисленному значению -5.

SET LOCK_TIMEOUT указывает количество миллисекунд, в течение которых инструкция ожидает снятия блокировки. Синтаксис:

SET LOCK_TIMEOUT *количество-миллисекунд*

По умолчанию -1 (то есть инструкция будет ждать всегда). Например, SET LOCK_TIMEOUT 1800 устанавливает время ожидания снятия блокировки равным 1800 миллисекунд. Функция @@LOCK_TIMEOUT возвращает значение времени ожидания блокировки в миллисекундах для текущего сеанса.

SET IMPLICIT_TRANSACTIONS устанавливает для соединения режим неявных транзакций. Синтаксис:

SET IMPLICIT_TRANSACTIONS { ON | OFF }.

SET REMOTE_PROC_TRANSACTIONS указывает, что в момент, когда активна локальная транзакция, выполнение удаленной хранимой процедуры запускает распределенную транзакцию, управляемую координатором распределенных транзакций (MS DTC). Синтаксис:

SET REMOTE_PROC_TRANSACTIONS { ON | OFF }.

Рекомендуется вместо вызова удаленных хранимых процедур использовать распределенные запросы, ссылающиеся на связанные серверы, которые определяются с помощью хранимой процедуры sp_addlinkedserver.

SET XACT_ABORT указывает, выполняет ли SQL Server автоматический откат текущей транзакции, если инструкция T-SQL вызывает ошибку выполнения. Синтаксис:

SET XACT_ABORT { ON | OFF }.

OFF – установка по умолчанию. В случае ON вся транзакция завершается и выполняется ее откат. В случае OFF в зависимости от серьезности ошибки возможен откат всей транзакции или откат только вызвавшей ошибку инструкции.

SET TRANSACTION ISOLATION LEVEL управляет работой блокировки и версиями строк инструкций T-SQL в текущем сеансе. Свойства инструкции:

- Одновременно может быть установлен только один параметр уровня изоляции, который продолжает действовать для текущего соединения до тех пор, пока не будет явно изменен.
- Все операции считывания, выполняемые в рамках транзакции, функционируют в соответствии с правилами уровня изоляции, если только подсказка в предложении FROM инструкции не указывает на другое поведение блокировки или управления версиями строк для таблицы.
- Уровни изоляции транзакции определяют тип блокировки, применяемый к операциям считывания. Совмещаемые блокировки, применяемые для READ COMMITTED или REPEATABLE READ, как правило, являются блокировками строк, но при этом, если в процессе считывания идет обращение к большому числу строк, блокировка строк может быть расширена до блокировки страниц или таблиц. Если строка была изменена транзакцией после считывания, для защиты такой строки транзакция применяет монопольную блокировку, которая сохраняется до завершения транзакции. Например, если транзакция REPEATABLE READ имеет разделяемую блокировку строки и при этом изменяет ее, совмещаемая блокировка преобразуется в монопольную. В любой момент времени выполнения транзакции можно переключиться с одного уровня изоляции на другой, однако есть одно исключение. Если инструкция SET TRANSACTION ISOLATION LEVEL использовалась в хранимой процедуре или триггере, то при возврате управления из них уровень изоляции будет изменен на тот, который действовал на момент их вызова.

Синтаксис:

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

READ COMMITTED — установка по умолчанию.

READ UNCOMMITTED указывает, что транзакция может считывать строки, которые были изменены другими транзакциями, но еще не были зафиксированы.

READ COMMITTED указывает, что транзакция не может считывать строки, которые были изменены другими

транзакциями, но еще не были зафиксированы. Это предотвращает чтение «грязных» данных. Данные могут быть изменены другими транзакциями между отдельными инструкциями в текущей транзакции, результатом чего будет неповторяемое чтение или недействительные данные. Поведение READ COMMITTED зависит от настройки аргумента базы данных READ_COMMITTED_SNAPSHOT.

REPEATABLE READ указывает на то, что инструкции не могут считывать данные, которые были изменены, но еще не зафиксированы другими транзакциями, а также на то, что другие транзакции не могут изменять данные, читаемые текущей транзакцией, до ее завершения.

SNAPSHOT указывает на то, что данные, считанные любой инструкцией транзакции, будут согласованы на уровне транзакции с версией данных, существовавших в ее начале. Транзакция распознает только те изменения, которые были зафиксированы до ее начала. Инструкции, выполняемые текущей транзакцией, не видят изменений данных, произведенных другими транзакциями после запуска текущей транзакции. Таким образом достигается эффект получения инструкциями в транзакции моментального снимка зафиксированных данных на момент запуска транзакции. Перед запуском транзакции, использующей уровень изоляции моментальных снимков, необходимо установить параметр базы данных ALLOW_SNAPSHOT_ISOLATION в ON.

SERIALIZABLE указывает следующее:

- Инструкции не могут считывать данные, которые были изменены другими транзакциями, но еще не были зафиксированы.
- Другие транзакции не могут изменять данные, считываемые текущей транзакцией, до ее завершения.
- Другие транзакции не могут вставлять новые строки со значениями ключа, которые входят в диапазон ключей, считываемых инструкциями текущей транзакции, до ее завершения.