Оглавление

[**T-SQL Fundamentals** 2](#_Toc80826153)

[Глава 1. Общие сведения 3](#_Toc80826154)

[Глава 2. Запросы к одиночным таблицам 15](#_Toc80826155)

[Глава 3. Соединения 51](#_Toc80826156)

[Глава 4. Вложенные запросы 57](#_Toc80826157)

[Глава 5. Табличные выражения 64](#_Toc80826158)

[Глава 9. Транзакции и параллелизм 65](#_Toc80826159)

[Глава 10. Программируемые объекты 92](#_Toc80826160)

# **T-SQL Fundamentals**

## Глава 1. Общие сведения

1. Какие элементы включает в себя SQL:

* DDL, DML, DCL,
* DML, DDL, DDT,
* TDD, DDL, DML,
* DDL, DML, TPL.

1. Какие команды включает в себя DDL:

* CREATE, ALTER, DROP,
* CREATE, ALTER, DROP, TRUNCATE,
* INSERT, UPDATE, DELETE,
* CREATE, ALTER, DROP, DELETE.

1. Какие команды включает в себя DML:

* SELECT, INSERT, UPDATE, DELETE, TRUNCATE, MERGE,
* SELECT, INSERT, UPDATE, DELETE,
* INSERT, UPDATE, DELETE,
* DROP, DELETE, TRUNCATE.

1. Какие команды включает в себя DCL:

* GRANT, REVOKE,
* SELECT, INSERT, UPDATE, DELETE, GRANT, REVOKE,
* CREATE, ALTER, DROP, GRANT, REVOKE,
* DROP, DELETE, TRUNCATE.

SQL сочетает элементы языков:

* **DDL** (Data Definition Language — язык описания данных: CREATE, ALTER, DROP),
* **DML** (Data Manipulation Language — язык управления данными: SELECT, INSERT, UPDATE, DELETE, TRUNCATE и MERGE),
* **DCL** (Data Control Language — язык управления БД: GRANT и REVOKE).

1. На чем основана реляционная модель:

* на теории множеств,
* на исчислении предикатов,
* оба варианта правильны.

Реляционная (логическая) модель, предназначенная для управления и манипулирования данными, основывается на теории множеств и исчислении предикатов.

Задача реляционной модели — обеспечить согласованное представление данных с минимальной (или вовсе отсутствующей) избыточностью, не жертвуя при этом их полнотой; целостность данных, вытекающая из их согласованности, является частью модели.

**Тип** — одна из наиболее фундаментальных составляющих отношений. Он ограничивает атрибут определенным набором возможных или корректных значений. Например, тип INT — множество всех целых чисел в диапазоне от 2 147 483 648 до 2 147 483 647.

1. Имеет ли значение порядок размещения элементов с точки зрения теории множеств?
2. Дано множество {a, b, c}. Является ли множество {b, c, a} тем же самым множеством или другое множество?

С точки зрения теории множеств порядок размещения элементов не играет никакой роли. Для перечисления элементов множества используют фигурные скобки: {a, b, c}. Поскольку порядок не имеет значения, одно и то же множество можно записать по-разному: {b, a, c} или {b, c, a}.

1. Какие значение возвращают предикаты в SQL:

* TRUE, FALSE, UNKNOW,
* TRUE, FALSE, NULL,
* 0, 1,
* все варианты правильны.

1. Что такое нормализация?

Нормализация — формальный математический процесс, в результате которого каждая сущность обязательно должна быть представлена только одним отношением. Нормализованная БД не подвержена аномалиям, возникающим при редактировании содержимого; она обеспечивает полноту данных, сохраняя минимальную избыточность. Если вы работаете в рамках модели ERM (Entity Relationship Modeling, то есть сущность — связь), представляя каждое отношение вместе со всеми атрибутами, вам не нужно думать о нормализации; она может понадобиться лишь для усиления модели и обеспечения ее корректности.

1. Что такое первая нормальная форма (1NF):

* кортежи внутри отношения должны быть уникальными,
* атрибуты внутри отношения должны быть атомарными,
* верны оба утверждения.

Согласно первой нормальной форме кортежи (строки) внутри отношения (таблицы) должны быть уникальными, а атрибуты атомарными. Другими словами, таблица, которая полностью соответствует отношению, имеет первую нормальную форму.

Вы можете оперировать атрибутами только с помощью действий, которые определены как часть типа атрибута. Их атомарность субъективна в том же смысле, что и определение множества. К примеру, сколькими столбцами должно выражаться имя сотрудника в отношении Employees: одним (полное имя), двумя (имя и фамилия) или тремя (имя, фамилия и отчество)? Это зависит от ситуации. Если вы будете работать с отдельными составляющими имени (скажем, для выполнения поиска), необходимо разбить его на части; в противном случае делать этого не стоит.

1. Правила для построения второй нормальной формы (2NF):

* обязательное соответствие первой форме,
* атрибуты, для которых не определено ни одного ключа, должны полностью зависеть от атрибутов с потенциальными ключами,
* верны оба утверждения.

Вторая нормальная форма вводит два правила: одно предусматривает обязательное соответствие первой форме, другое касается связи между атрибутами с потенциальным ключом и без него. Атрибуты, для которых не определено ни одного ключа, должны полностью функционально зависеть от атрибутов с потенциальными ключами. Если вы хотите получить содержимое атрибута без ключа, вам необходимо предоставить значения всех атрибутов кортежа, на основе которого сформирован потенциальный ключ. Зная данные столбцов потенциального ключа, вы получите содержимое любого столбца в любой строке.

1. Правила для построения третьей нормальной формы (3NF):

* обязательное соответствие второй форме,
* атрибуты без ключа должны быть независимыми друг от друга,
* верны оба утверждения.

Третья нормальная форма тоже имеет два правила. Первое — данные должны соответствовать второй нормальной форме. Второе — зависимость всех атрибутов, для которых не определен ключ, от потенциального ключа должна быть не транзитивной.

Проще говоря, атрибуты без ключа должны быть независимыми друг от друга.

Если отбросить формальности, суть второй и третьей форм можно сформулировать следующим образом: «Каждый атрибут, для которого не определен ключ, зависит от целого ключа, только от ключа и ни от чего, кроме ключа».

1. Какие системные базы данных создает установщик MS SQL Server’а:

* master, resource, model, tempdb, msdb;
* master, temp;
* resource, model, master.

***Master***. Содержит метаданные об экземпляре SQL Server, конфигурацию сервера, сведения обо всех БД в текущем экземпляре, а также ресурсы, необходимые для инициализации.

***Resource***. Скрытая БД, доступная только для чтения. Хранит определения всех системных объектов. Запрашивая системные объекты, обычно обращаются к схеме sys локальной БД, но на деле все определения находятся в БД Resource.

***Model***. Используется в качестве шаблона для новых БД. Каждая БД, которую вы создаете, изначально является копией model. Если хотите, чтобы новые БД были сконфигурированы определенным образом или содержали какой-то набор объектов (таких как типы данных), приведите к нужному виду БД model. Следует понимать, что изменения коснутся только новых БД, не затрагивая существующие.

***Tempdb***. Место, где SQL Server хранит временные данные (рабочие таблицы, пространство для сортировки, сведения о версиях строк и т. д.). Эта БД уничтожается и создается заново (на основе БД model) при каждом перезапуске экземпляра SQL Server.

***Msdb***. Здесь служба агента SQL Server хранит свои данные. Она отвечает за репликацию и автоматизацию задач, планирования и оповещения. Msdb содержит сведения о таких компонентах и возможностях SQL Server, как Database Mail, Service Broker, механизм резервного копирования и т. д.

На уровне БД можно определить свойство под названием **collation**, которое отвечает за поддержку языка, чувствительность к регистру и порядок сортировки символьной информации. Если не указать это свойство при создании БД, для сравнения символов будет использоваться стандартная конфигурация.

БД состоит из файлов, в которых хранятся как сами данные, так и журнал транзакций. При создании БД для каждого файла можно указывать различные свойства: местоположение, начальный и максимальный размеры, автоувеличение прироста размера.

Все расширения файлов в БД имеют понятные значения. Расширение .mdf расшифровывается как Master Data File — главный файл данных (не путать с БД master), а .ldf означает Log Data File — файл журнала. Говорят, когда выбирали расширение для вторичных файлов данных, один из разработчиков в шутку предложил вариант .ndf (Not Master Data File — неглавный файл данных), и идею поддержали.

1. Где хранятся таблицы:

* таблицы хранятся внутри схем, а схемы находятся в БД,
* таблицы хранятся в БД,
* таблицы хранятся в контейнерах.

Схема — это нечто вроде контейнера для таких объектов, как таблицы, представления, хранимые процедуры и т. д. Такой механизм позволяет управлять полномочиями. Например, вы можете выдать пользователю доступ к оператору SELECT внутри определенной схемы, чтобы он запрашивал данные из всех ее объектов. Безопасность — один из факторов, которые следует учитывать, разделяя объекты на схемы.

1. Какая схема создается автоматически в каждой БД и используется по умолчанию:

* dbo,
* master,
* temp.

Схема dbo создается автоматически в каждой БД и используется по умолчанию для всех пользователей, которые не были явно привязаны к другой схеме.

1. Какую команду надо использовать, если необходимо совершить несколько запросов к одной и той же базе данных:

* USE,
* GO,
* EXEC,
* SELECT.

Инструкция USE устанавливает контекст, то есть делает так, чтобы указанная БД стала текущей. Благодаря этому скрипты создают объекты в определенной БД. В локальной версии SQL Server инструкция способна изменять текущий контекст. В SQL Database нельзя переключаться между БД, но если вы уже подключены к указанной БД, инструкция USE выполнится успешно.

1. Какой из следующих запросов наиболее корректен:

* CREATE TABLE Customers (

Id INT NOT NULL,

Name VARCHAR(50) NULL,

Birthday DATE NULL

);

* CREATE TABLE Customers (

Id INT NOT NULL,

Name VARCHAR(50),

Birthday DATE

);

* CREATE TABLE Customers (

Id INT NOT NULL,

Name VARCHAR(50) NULL,

Birthday DATE

);.

Если явно не указать, должен ли столбец поддерживать значения NULL, SQL Server использует стандартную конфигурацию. Согласно спецификации языка SQL все столбцы по умолчанию могут быть пустыми (принимать отметки NULL), но в настройках это поведение можно изменить. Я настоятельно рекомендую вам явно определять такое свойство, не полагаясь на значения по умолчанию. Также советую всегда использовать запись NOT NULL и опускать ее только в случаях, когда возникает серьезная необходимость поддерживать значения NULL.

В документации к SQL Server сказано, что возможность игнорировать точку с запятой, разделяя инструкции языка T-SQL, является устаревшей. Проще говоря, в будущих версиях продукта разработчики не смогут ей воспользоваться. Это еще один повод, чтобы начать разделять свои инструкции точкой с запятой.

1. Целостность данных бывает:

* декларативной,
* процедурной,
* оба варианта верны.

Одно из прекрасных качеств реляционной модели – изначальная поддержка целостности данных. Целостность бывает декларативной (является частью определений таблиц) и процедурной (описывается внутри кода с помощью хранимых процедур или триггеров).

1. Сколько ограничений первичного ключа (PRIMARY KEY) может быть в таблице:

* одно,
* несколько.

1. Сколько ограничений UNIQUE может быть в таблице:

* одно,
* несколько.

Ограничение UNIQUE обеспечивает уникальность строк и реализует такую концепцию реляционной модели, как резервные ключи. В отличие от первичных ключей в одной таблице возможно присутствие нескольких ограничений UNIQUE. Кроме того, уникальность распространяется на все столбцы, включая те, которые могут иметь пустые значения. В соответствии со спецификацией SQL столбцы с ограничением UNIQUE поддерживают разные отметки типа NULL (как будто они отличаются). Однако язык T-SQL, реализованный в SQL Server, запрещает дублировать эти отметки (любые два значения NULL не могут быть равными).

1. На что ссылается внешний ключ:

* на первичный ключ (PRIMARY KEY),
* на уникальный ключ (UNIQUE),
* оба варианта верны.

Внешний ключ отвечает за ссылочную целостность. Это ограничение определяется для одного или нескольких атрибутов в так называемой ссылающейся таблице и указывает на атрибуты потенциального ключа (ограничения PRIMARY KEY или UNIQUE) в таблице, на которую ссылаются (ее также называют ссылочной). Стоит заметить, что это может быть одна и та же таблица. Внешний ключ нужен для того, чтобы его атрибуты принимали только те значения, которые существуют в ссылочных столбцах.

1. Создана следующая таблица:

CREATE TABLE Customers (

Id INT IDENTITY PRIMARY KEY,

Name VARCHAR(30) NOT NULL,

Birthday DATE NOT NULL,

BossId INT NULL

);

Является ли корректным следующий запрос

ALTER TABLE Customers

ADD CONSTRAINT FK\_Customers\_Customers

FOREIGN KEY(BossId)

REFERENCES Customers(Id);

Можно сослаться на столбец в той же таблице. С помощью этого кода мы ограничим содержимое столбца BossId значениями, которые хранятся в столбце Id (все это в таблице Customers). Обратите внимание: отметки NULL допускаются в столбцах внешнего ключа (в нашем случае BossId), даже если их нет в столбцах, на которые они ссылаются.

1. Какие действие поддерживают параметры ON DELETE и ON UPDATE:

* CASCADE, SET DEFAULT, SET NULL,
* CASCADE, SET DEFAULT,
* CASCADE.

Вы можете определить внешний ключ с действиями, компенсирующими попытки удалить строку или обновить атрибуты потенциального ключа в таблице, на которую указывает внешняя ссылка. Речь идет о параметрах ON DELETE и ON UPDATE. В рамках определения внешнего ключа они поддерживают действия CASCADE, SET DEFAULT и SET NULL. Действие CASCADE приводит к тому, что выполняемые операции (удаление или обновление) будут распространяться и на связанные строки. Например, выражение ON DELETE CASCADE означает, что при удалении строки из таблицы, на которую указывает ссылка, СУРБД удалит также все связанные с ней строки из ссылающейся таблицы. Действия SET DEFAULT и SET NULL назначат атрибутам внешнего ключа в связанных строках соответственно значение по умолчанию и NULL. Независимо от выбранного действия указывающие «в никуда» строки могут появиться в ссылающейся таблице только в исключительном случае с отметками типа NULL.

1. Создана следующая таблица:

CREATE TABLE Customers (

Id INT IDENTITY PRIMARY KEY,

Name VARCHAR(30) NOT NULL,

Age INT NOT NULL CHECK(Age > 0 AND Age < 100)

);

Какие из значений 5, 100, NULL являются допустимыми?

* 5 и NULL допустимы, 100 недопустимо;
* 5 допустимо, 100 и NULL недопустимы;
* 5 и 100 допустимы, NULL недопустимо;
* все значения допустимы.

Обратите внимание: для ограничения CHECK проверка считается не пройденной, если предикат возвращает FALSE. Если результат равен TRUE или UNKNOWN, изменения принимаются.

1. Может ли поле с первичным ключом (PRIMARY KEY) хранить значение NULL?

Ограничение первичного ключа (PRIMARY KEY) обеспечивает уникальность строк и запрещает хранить значения NULL в соответствующих столбцах.

1. Может ли поле с ключом UNIQUE хранить значение NULL?

Ограничение UNIQUE обеспечивает уникальность строк и реализует такую концепцию реляционной модели, как резервные ключи. В отличие от первичных ключей в одной таблице возможно присутствие нескольких ограничений UNIQUE. Кроме того, уникальность распространяется на все столбцы, включая те, которые могут иметь пустые значения. В соответствии со спецификацией SQL столбцы с ограничением UNIQUE поддерживают разные отметки типа NULL (как будто они отличаются).

1. Может ли поле, имеющее ограничение DEFAULT, хранить значение NULL?

Ограничение DEFAULT связывается с определенным атрибутом. Выражение используется в качестве значения по умолчанию для пустых атрибутов строки, добавляемой в таблицу. Но можно принудительно записать NULL.

1. С помощью какой команды можно удалить таблицу из БД:

* DROP,
* DELETE,
* TRUNCATE.

1. Какая из нижеперечисленных команд удалит все строки таблицы Orders:

* DELETE Orders,
* TRUNCATE TABLE Orders,
* оба варианта верны.

1. Что хранит тип данных BIT:

* 0 или 1,
* TRUE или FALSE,
* оба варианта верны.

1. Даны несколько строк для описания столбца Sum:
2. Sum DECIMAL(9, 2),
3. Sum DECIMAL(5),
4. Sum DECIMAL(4, 0),
5. Sum DECIMAL(),
6. Sum DECIMAL(4, 5),

Какая из этих строк неверна?

* 5,
* 4 и 5,
* 3 и 4,
* 2, 3 и 4,
* все верны.

Данный тип может принимать два параметра precision и scale: DECIMAL(precision, scale).

Параметр precision представляет максимальное количество цифр, которые может хранить число. Это значение должно находиться в диапазоне от 1 до 38. По умолчанию оно равно 18.

Параметр scale представляет максимальное количество цифр, которые может содержать число после запятой. Это значение должно находиться в диапазоне от 0 до значения параметра precision. По умолчанию оно равно 0.

1. Какие из нижеперечисленных типов хранят строку в кодировке Unicode:

* NCHAR и NVARCHAR,
* CHAR и NCHAR,
* VARCHAR и NVARCHAR,
* все варианты верны.

1. Тип данных MONEY эквивалентен:

* DECIMAL(19, 4),
* DECIMAL(9, 2),
* DECIMAL(10, 2),
* DECIMAL(10).

1. Какие из нижеперечисленных типов хранят бинарные данные:

* BINARY,
* VARBINARY,
* IMAGE,
* все варианты верны.

1. Даны несколько строк для описания столбца Id как первичного ключа:
2. Id INT PRIMARY KEY,
3. Id DECIMAL(7, 1) PRIMARY KEY,
4. Id CHAR(10) PRIMARY KEY,

Какая из этих строк неверна?

* все верны,
* 2,
* 3,
* 2 и 3.

1. Даны несколько строк для описания столбца Id как первичного ключа с идентификатором:
2. Id INT PRIMARY KEY INDENTITY,
3. Id INT PRIMARY KEY INDENTITY(1, 1),
4. Id INT PRIMARY KEY INDENTITY(2, 3),

Какая из этих строк неверна?

* все верны,
* 2,
* 3,
* 2 и 3.

## Глава 2. Запросы к одиночным таблицам

1. Напишите последовательность обработки инструкций SELECT, FROM, WHERE, GROUP BY, HAVING и ORDER BY.

Несмотря на то что инструкция SELECT находится в самом начале, обрабатываться она будет одной из самых последних. Выполнение операторов происходит в следующем порядке:

1) FROM;

2) WHERE;

3) GROUP BY;

4) HAVING;

5) SELECT;

6) ORDER BY.

Запрос нельзя записывать в логическом порядке. Необходимо начинать с инструкции SELECT. Существует причина, по которой логический порядок обработки запроса и фактическая запись должны отличаться. SQL задумывался как декларативный язык программирования. Он позволяет оформлять запросы в виде выражений, похожих на английскую речь.

1. Какой из следующий запросов неверен:

1) SELECT \*

FROM [Order Details];

2) SELECT \*

FROM “Order Details”;

3) SELECT \*

FROM Order Details;

* 3,
* 2,
* 1 и 2,
* все верны,
* все не верны.

Если имена идентификаторов, которые используются в схемах, таблицах и столбцах, отвечают определенному формату, вам не нужно их выделять. Правила форматирования идентификаторов вы найдете в электронном справочнике для SQL Server по адресу msdn.microsoft.com/ru-ru/library/ms175874. Если идентификатор оформлен неправильно (содержит пробелы или специальные символы, начинается с цифры или совпадает с зарезервированным ключевым словом), вы должны его выделить. В SQL Server это можно сделать несколькими способами. Обычно используются двойные кавычки (стандартная форма) — "Order Details". T-SQL также поддерживает квадратные скобки — [Order Details].

1. Какие из нижеперечисленный утверждений об инструкции WHERE верны:
2. инструкция WHERE использует только индексы,
3. в зависимости от содержимого фильтра SQL Server может использовать индексы для доступа к нужным данным,
4. фильтры снижают объем данных, передаваемых по сети, поскольку фильтрация происходит на стороне сервера, а клиенту передается конечный результат.

* 2 и 3,
* 1 и 3,
* 1 и 2,
* все утверждения верны.

Инструкция WHERE играет большую роль с точки зрения производительности запросов. В зависимости от содержимого фильтра SQL Server может использовать индексы для доступа к нужным данным. В некоторых ситуациях индексы избавляют СУРБД от сканирования всей таблицы. Фильтры также снижают объем данных, передаваемых по сети, поскольку фильтрация происходит на стороне сервера, а клиенту передается конечный результат.

1. Как работает инструкция WHERE:

* опускает все записи, для которых логическое выражение является ложным (FALSE) или неопределенным (UNKNOWN),
* опускает все записи, для которых логическое выражение является ложным (FALSE).

На этапе WHERE возвращаются только строки, которые удовлетворяют логическому выражению. Никогда не забывайте, что в языке T-SQL для предикатов используется троичная логика. Она предусматривает три возможных результата: TRUE, FALSE и UNKNOWN. То есть «вернуть TRUE» — не то же самое, что «не вернуть FALSE». Инструкция WHERE опускает все записи, для которых логическое выражение является ложным (FALSE) или неопределенным (UNKNOWN).

1. Дана таблица Table, в которой есть поле Number, содержащее следующие значения: ***0, 6, 21, NULL, 5, -3***. Что будет выведено в результате запроса

SELECT Number

FROM Table

WHERE Number > 5;

* 6, 21;
* 5, 6, 21;
* 6, 21, 5;
* 6, 21, NULL;
* 6, 21, NULL, 5.

1. Дана таблица Table, в которой есть поле Name, содержащее следующие значения: ***John, Steve, Freddy, NULL, Adam, Paul***. Что будет выведено в результате запроса

SELECT Name

FROM Table

WHERE Name > ‘O’;

* Steve, Paul;
* Steve, NULL, Paul;
* John, Freddy, NULL, Adam;
* John, Freddy, Adam.

1. Дана таблица Table, в которой есть поле Name, содержащее следующие значения: ***John, Steve, Freddy, NULL, Adam, Paul***. Что будет выведено в результате запроса

SELECT Name

FROM Table

WHERE Name > ‘O’ AND Name < ‘S’;

* Paul;
* Steve, Paul;
* Steve, NULL, Paul;
* NULL;
* ничего.

1. Дана таблица Table, в которой есть поле Name, содержащее следующие значения: ***John, Steve, Freddy, NULL, Adam, Paul***. Что будет выведено в результате запроса

SELECT Name

FROM Table

WHERE Name > ‘P’ AND Name < ‘S’;

* Paul;
* Steve, Paul;
* Steve, NULL, Paul;
* NULL;
* ничего.

1. Дана таблица Table, в которой есть поле Name, содержащее следующие значения: ***John, Steve, Freddy, NULL, Adam, Paul***. Что будет выведено в результате запроса

SELECT Name

FROM Table

WHERE Name > ‘Q’ AND Name < ‘S’;

* ничего;
* Steve;
* Steve, NULL;
* NULL.

1. Дана таблица Table, в которой есть поле Name, содержащее следующие значения: ***John, \_***(пустая строка)***, Luc, NULL, Adam, Paul***. Что будет выведено в результате запроса

SELECT Name

FROM Table

WHERE LEN(Name) > 3;

* John, Adam, Paul;
* John, NULL, Adam, Paul;
* John, Luc, Adam, Paul;
* John, Luc, NULL, Adam, Paul.

1. Дана таблица Table, в которой есть поле Date, содержащее следующие значения:

|  |
| --- |
| 06.11.2020 |
| 17.02.2016 |
| 22.10.2010 |
| 01.05.2017 |
| 07.04.2020 |
| 30.06.2005 |

Какой из двух запросов будет выполнен быстрее при условии, что таблица проиндексирована по полю Date

1. SELECT Date

FROM Table

WHERE YEAR(Date) = 2021;

1. SELECT Date

FROM Table

WHERE Date >= 01.01.2021 AND Date <= 31.12.2021;

* первый запрос,
* второй запрос,
* оба выполнятся за одинаковое время.

1. Является ли корректным следующий запрос:

SELECT \*

FROM Orders

WHERE CustomerId = 71

GROUP BY OrderId;

1. Является ли корректным следующий запрос:

SELECT OrderId, Price

FROM Orders

WHERE CustomerId = 71

GROUP BY OrderId;

1. Является ли корректным следующий запрос:

SELECT OrderId, SUM(Price \* Quantity)

FROM Orders

WHERE CustomerId = 71

GROUP BY OrderId

ORDER BY OrderDate;

1. Является ли корректным следующий запрос:

SELECT OrderId, SUM(Price \* Quantity) AS Total

FROM Orders

WHERE CustomerId = 71

GROUP BY OrderId

ORDER BY Total;

Если данные в ходе запроса группируются, инструкции, которые следуют за этапом GROUP BY (включая HAVING, SELECT и ORDER BY), должны работать с группами, а не с отдельными строками. В итоге каждая группа будет представлена одной строкой. Это значит, что все выражения в инструкциях, следующих за оператором SELECT, должны возвращать для каждой группы скалярную величину (одиночное значение).

1. Какие из перечисленных функций можно указывать на этапе SELECT, если поля, используемые в этих функциях, не входят в GROUP BY:
2. COUNT, SUM, MAX;
3. MAX, AVG, CONCAT;
4. MIN, LEN, SUM;
5. CAST, LEN, SUM, COUNT.
6. Является ли корректным следующий запрос:

SELECT EmployeeId, LastName, YEAR(OrderDate)

FROM Orders

GROUP BY EmployeeId, YEAR(OrderDate);

Элементы, не упоминающиеся на этапе GROUP BY, можно указывать только в агрегатных функциях (COUNT, SUM, AVG, MIN или MAX).

1. Есть группа из пяти строк со значениями 30, 10, NULL, 10, 10 в столбце под названием Number. Сколько строк вернет выражение COUNT(\*)?

* 5,
* 4,
* 3,
* 2.

1. Есть группа из пяти строк со значениями 30, 10, NULL, 10, 10 в столбце под названием Number. Сколько строк вернет выражение COUNT(Number)?

* 4,
* 5,
* 3,
* 2.

1. Есть группа из пяти строк со значениями 30, 10, NULL, 10, 10 в столбце под названием Number. Сколько строк вернет выражение COUNT(DISTINCT Number)?

* 2,
* 3,
* 4,
* 5.

1. Какие из перечисленных агрегатных функций для столбца Number игнорируют отметки типа NULL:

* SUM(Number),
* AVG(Number),
* COUNT(Number),
* COUNT(DISTINCT Number),
* все вышеперечисленные.

1. Какие из перечисленных агрегатных функций для столбца Number НЕ игнорируют отметки типа NULL:

* COUNT(\*)
* COUNT(Number),
* SUM(Number),
* AVG(Number),
* все вышеперечисленные.

Все агрегатные функции, за исключением COUNT(\*), игнорируют отметки типа NULL. Возьмем для примера группу из пяти строк со значениями 30, 10, NULL, 10, 10 в столбце под названием qty. Выражение COUNT(\*) вернет 5, так как в группе находится пять записей. Однако результатом выражения COUNT(qty) будет 4, поскольку здесь учитываются только конкретные значения. Если вас интересуют лишь отдельные вхождения определенных результатов, укажите в скобках агрегатной функции ключевое слово DISTINCT. К примеру, выражение COUNT(DISTINCT qty) вернет 2, потому что у нас есть только два уникальных значения. Ключевое слово DISTINCT можно использовать и с другими функциями. Если выражение SUM(qty) возвращает 60, то результатом выполнения SUM(DISTINCT qty) будет 40. AVG(qty), вернет 15, тогда как AVG(DISTINCT qty) выдаст 20.

1. Какой параметр в языке SQL обеспечивает уникальность результата после выполнения инструкции SELECT

* DISTINCT,
* UNIQUE,
* FIRST,
* DEFAULT.

В языке SQL существуют средства, которые обеспечивают уникальность результата после выполнения инструкции SELECT. Речь идет о параметре DISTINCT. Он убирает из набора повторяющиеся строки.

1. Для чего нужна инструкция HAVING:

* для фильтрации групп,
* для фильтрации отдельных строк,
* оба варианта верны.

1. Какие группы попадают на следующий этап обработки после инструкции HAVING:

* для которых истинно логическое выражение,
* для которых ложно логическое выражение,
* для которых предикат вернул FALSE или UNKNOWN,
* для которых предикат вернул TRUE или UNKNOWN.

С помощью инструкции HAVING определяется предикат для фильтрации групп (напомню, что фильтрация отдельных строк происходит на этапе WHERE). На следующий этап обработки попадают только группы, для которых истинно логическое выражение в инструкции HAVING. Если предикат вернул FALSE или UNKNOWN, группы отсеиваются.

1. Могут ли попасть в результат строки, в которых EmployeeId = 72:

SELECT EmployeeId, YEAR(OrderDate), COUNT(\*)

FROM Orders

WHERE CustomerId = 71

GROUP BY EmployeeId, YEAR(OrderDate)

HAVING COUNT(\*) > 1;

1. Сколько столбцов будет выведено в результате следующего запроса:

SELECT OrderId OrderDate

FROM Orders;

* 1,
* 2,
* возникнет синтаксическая ошибка.

Если случайно пропустить запятую между столбцами в инструкции SELECT, код не завершится ошибкой. SQL Server посчитает, что второе имя является псевдонимом для первого. С точки зрения синтаксиса запрос является корректным и подразумевает, что вы хотели создать для столбца orderid псевдоним под названием orderdate. В результате вы получите всего один столбец с именем orderdate, который хранит идентификаторы заказов.

1. Сколько столбцов будет выведено в результате следующего запроса:

SELECT OrderId, YEAR(OrderDate) AS OrderYear

FROM Orders

WHERE OrderYear > 2006;

* 1,
* 2,
* возникнет синтаксическая ошибка.

1. Сколько столбцов будет выведено в результате следующего запроса:

SELECT OrderId, YEAR(OrderDate) AS OrderYear, COUNT(\*) AS NumOrders

FROM Orders

WHERE CustomerId = 26

GROUP BY OrderId, YEAR(OrderDate)

HAVING NumOrders > 1;

* 1,
* 2,
* возникнет синтаксическая ошибка.

Этап SELECT выполняется после обработки инструкций FROM, WHERE, GROUP BY и HAVING. Значит, псевдонимы, присвоенные выражениям в инструкции SELECT, на всех предыдущих этапах просто не существуют.

1. Корректен ли следующий запрос

SELECT OrderId

,YEAR(OrderDate) AS OrderYear

,OrderYear + 1 AS NextYear

FROM Orders;

В инструкции SELECT нельзя ссылаться на псевдоним столбца, который был создан в той же инструкции, и не важно, используется он справа или слева от места создания.

1. Является ли корректным следующий запрос:

SELECT ProductName, Price, ProductCount

FROM Products

ORDER BY ProductCount \* Price;

В качестве критерия сортировки также можно использовать сложное выражение на основе столбцов произведения значений столбцов.

1. Что такое курсор:

* нереляционный результат с гарантированным порядком следования строк,
* любое множество,
* мультимножество.

Одна из важнейших особенностей языка SQL состоит в том, что таблица не гарантирует какой-либо упорядоченности, поскольку представляет собой множество (или мультимножество при наличии дубликатов), которое по определению не имеет порядка. Таким образом, если вы не воспользуетесь инструкцией ORDER BY, SQL Server сам решит, в каком порядке возвращать записи. Отсортировать строки в результирующем наборе можно исключительно с помощью инструкции ORDER BY.

При этом результат нельзя называть таблицей. В спецификации языка SQL упорядоченные записи, которые возвращены подобным способом, называют **курсором** — нереляционным результатом с гарантированным порядком следования строк.

1. Является ли множеством результат следующего запроса:

SELECT EmployeeId, YEAR(OrderDate) AS OrderYear

FROM Orders

WHERE CustomerId = 26;

В реляционной модели операции с отношениями основаны на реляционной алгебре; в результате их выполнения появляется новое отношение (множество). В языке SQL все немного иначе: операция SELECT не всегда возвращает настоящее множество, то есть набор уникальных неупорядоченных строк. В SQL таблица не обязана быть множеством. Уникальность строк не гарантируется без ключа, а это одно из обязательных свойств отношения в реляционной модели; без него получится мультимножество. Если таблицы, с которыми вы работаете, имеют ключи и являются множествами, результат выполнения инструкции SELECT все равно может содержать повторяющиеся строки. Такой итог часто называют результирующим набором, и он не обязан соответствовать математическому определению множества. Например, таблица Orders является множеством, поскольку уникальность ее строк обеспечена ключом, однако запрос к ней возвращает повторяющиеся записи.

1. Корректен ли следующий запрос

SELECT OrderId, YEAR(OrderDate) AS OrderYear

FROM Orders

ORDER BY OrderYear;

Обратите внимание: инструкция ORDER BY ссылается на псевдоним столбца orderyear, который был создан на этапе SELECT. Ранее я утверждал, что так делать нельзя, но инструкция ORDER BY — исключение, ведь она выполняется в последнюю очередь. Если вы определите псевдоним, название которого совпадает с именем исходного столбца (например, col1 AS col1), и сошлетесь на него в инструкции ORDER BY, для сортировки будет использован новый столбец. Если вы хотите выполнить сортировку по возрастанию, укажите сразу после выражения параметр ASC (такой порядок сортировки используется по умолчанию) — orderyear ASC. Чтобы упорядочить строки по убыванию, примените параметр DESC — orderyear DESC.

1. Корректен ли следующий запрос

SELECT OrderId, YEAR(OrderDate) AS OrderYear

FROM Orders

ORDER BY 2;

1. Корректен ли следующий запрос

SELECT OrderId, YEAR(OrderDate) AS OrderYear

FROM Orders

ORDER BY 2, 1;

Язык T-SQL позволяет ссылаться на столбцы в инструкции ORDER BY по их порядковому номеру. Он определяется размещением столбцов в списке атрибутов для инструкции SELECT.

Однако существует несколько причин, по которым такой подход считается нежелательным. Во-первых, в реляционной модели атрибуты никак не упорядочены, и к ним нужно обращаться по имени. Во-вторых, изменяя инструкцию SELECT, вы можете не внести соответствующие правки на этапе ORDER BY. Используя имена столбцов, вы защищаете код от подобных ошибок.

1. Корректен ли следующий запрос

SELECT OrderId, OrderDate

FROM Orders

ORDER BY Price;

1. Корректен ли следующий запрос

SELECT DISTINCT Country

FROM Employees

ORDER BY EmployeeId;

1. Какое выражение можно подставить в следующий запрос, чтобы он был корректен

SELECT …

FROM Orders

ORDER BY Price;

1. \*
2. Price,
3. OrderId, OrderDate,
4. OrderId, OrderDate, Price.

* все перечисленные,
* 3 и 4,
* 2, 3 и 4,
* только 2.

1. Какое выражение можно подставить в следующий запрос, чтобы он был корректен

SELECT DISTINCT …

FROM Orders

ORDER BY Price;

1. Price,
2. OrderId,
3. OrderId, OrderDate,
4. OrderId, OrderDate, Price.

* только 1,
* 1 и 2,
* 2, и 3,
* все перечисленные.

Язык T-SQL позволяет указывать в инструкции ORDER BY элементы, отсутствующие на этапе SELECT. Это означает, что вы можете выполнять сортировку по столбцам, которые не войдут в конечный результат.

Однако, используя параметр DISTINCT, вы можете указывать в инструкции ORDER BY лишь элементы, которые присутствуют в списке SELECT. Причина ограничения заключается в том, что в результате работы параметра DISTINCT одна итоговая строка может представлять несколько исходных; поэтому SQL Server не будет знать, какое из возможных значений следует использовать на этапе ORDER BY.

1. Что делает параметр TOP:

* ограничивает количество строк в абсолютных числах, возвращаемых запросом,
* ограничивает количество строк в относительных числах, возвращаемых запросом,
* оба варианта верны.

Параметр TOP — один из фирменных инструментов языка T-SQL. Он ограничивает количество строк (в абсолютных или относительных числах), возвращаемых запросом.

1. Есть таблица Table с полем Numbers, записи в котором содержат следующие значения: -2, 0, -5, 7, -5, 0, 11, 7, 0. Что вернет следующий запрос

SELECT DISTINCT TOP(2) Numbers

FROM Table

ORDER BY Numbers;

1. -5, -2;
2. -5, -5;
3. 11, 7,
4. 0, 7.

Инструкция ORDER BY вызывается после этапа SELECT, в котором указан параметр DISTINCT. То же справедливо и для фильтра TOP. Для фильтрации он использует инструкцию ORDER BY. То есть, если указать параметр DISTINCT на этапе SELECT, инструкция TOP будет выполнена после удаления повторяющихся строк.

1. Что такое недетерминированный запрос:

* в результате запроса строки вернутся в неопределенном порядке, то есть запрос имеет более одного корректного результата,
* в результате запроса не было получено никаких данных,
* любой запрос, в котором не используется инструкция SELECT.

Возможна ситуация, когда список атрибутов в инструкции ORDER BY не уникален, то есть несколько строк содержат одно и тоже значение. Если не указать дополнительных условий, то строки вернутся в неопределенном порядке. Данный запрос является **недетерминированным**, то есть имеет более одного корректного результата. При прочих равных строки будут упорядочены с учетом того, когда их использовали в первый раз.

Чтобы запрос был детерминированным, список атрибутов в инструкции ORDER BY следует сделать уникальным; проще говоря, необходимо добавить дополнительное условие.

1. Можно ли указывать параметр TOP в запросе, не содержащем инструкцию ORDER BY?
2. Корректен ли следующий запрос

SELECT TOP(5) OrderId

FROM Orders;

Фильтр TOP можно указывать в запросе, не содержащем инструкцию ORDER BY. Тогда порядок следования записей будет полностью неопределенным — SQL Server вернет n первых строк, к которым осуществлялся физический доступ (где n — количество запрашиваемых записей).

1. Таблица Orders содержит 200 записей. Сколько записей вернет следующий запрос

SELECT TOP (1) PERCENT OrderId, OrderDate

FROM Orders

ORDER BY OrderDate DESC;

* 2,
* 1,
* 10.

Вы можете использовать инструкцию TOP в сочетании с ключевым словом PERCENT. В таком случае число возвращаемых столбцов будет вычисляться в процентах от общего количества с использованием округления.

1. Имеется таблица Table с полем Age, в котором содержаться следующие значения: 25, 31, 26, 33, 26, 41, 26. Сколько строк вернет следующий запрос?

SELECT TOP(3) WITH TIES Age

FROM Table

ORDER BY Age;

* 4,
* 3,
* 5.

Вместо добавления дополнительного условия в список элементов инструкции ORDER BY вы можете сделать так, чтобы возвращались все равнозначные записи. Чтобы их вернуть, добавьте параметр WITH TIES.

1. Корректен ли следующий запрос

SELECT \*

FROM Employees

OFFSET 10 ROWS FETCH NEXT 5 ROWS ONLY;

В T-SQL фильтр OFFSET-FETCH является частью этапа ORDER BY, на котором результат сортируется для последующего вывода. С помощью инструкции OFFSET можно указать, сколько строк необходимо пропустить, а инструкция FETCH позволяет определить количество фильтруемых записей, начиная с пропущенных.

У языка T-SQL есть одна интересная особенность, которая касается синтаксиса фильтра OFFSET-FETCH: параметры ROW и ROWS являются взаимозаменяемыми. Они придают запросу сходство с английским языком, позволяя выбирать между единственным и множественным числом. Представьте, что вы хотите извлечь только одну строку. Выражение FETCH 1 ROWS синтаксически корректно, но с точки зрения английской грамматики выглядит несуразным. Поэтому вы можете написать FETCH 1 ROW. То же относится к инструкции OFFSET. Кроме того, если не пропускается ни одной строки (OFFSET 0 ROWS), вместо NEXT (следующих) есть возможность указать FIRST (первых) — эти ключевые слова также взаимозаменяемы.

1. Как используется инструкция OFFSET-FETCH:

* инструкцию FETCH нельзя использовать без OFFSET,
* инструкцию OFFSET нельзя использовать без FETCH,
* оба варианта верны.

Запрос, в котором используется фильтр OFFSET-FETCH, должен содержать инструкцию ORDER BY. Кроме того, инструкцию FETCH нельзя использовать без OFFSET. Чтобы применить фильтр, не пропуская ни одной строки, необходимо использовать выражение OFFSET 0 ROWS. Однако инструкция OFFSET работает и без FETCH. В этом случае запрос пропустит определенное количество строк и вернет все, что останется.

1. При использовании инструкции OFFSET-FETCH:

* поддерживается параметр PERCENT,
* поддерживается параметр WITH TIES,
* оба варианта верны,
* оба варианта неверны.

Фильтр OFFSET-FETCH более гибкий, чем TOP, так как умеет пропускать записи. Но в отличие от TOP он не поддерживает параметры PERCENT и WITH TIES.

1. Что возвращают оконные функции:

* скалярное(одиночное) значение,
* одну строку,
* набор строк, называемых окном.

**Оконными** называют функции, которые на основе окна (набора строк) вычисляют **скалярное** (одиночное) значение. Окно определяется с помощью инструкции OVER.

1. Имеется следующая таблица Orders:

|  |  |  |
| --- | --- | --- |
| OrderId | CustomerId | Price |
| 1 | 2 | 28 |
| 2 | 1 | 80 |
| Иет 3 | 1 | 45 |
| 4 | 2 | 15 |
| 5 | 1 | 67 |
| 6 | 3 | 5 |

Что вернет следующий запрос

SELECT OrderId

,CustomerId

,Price

,ROW\_NUMBER() OVER

(PARTITION BY CustomerId ORDER BY Price) AS Num

FROM Orders

ORDER BY CustomerId, Price;

|  |  |  |  |
| --- | --- | --- | --- |
| OrderId | CustomerId | Price | Num |
| 3 | 1 | 45 | 1 |
| 5 | 1 | 67 | 2 |
| 2 | 1 | 80 | 3 |
| 4 | 2 | 15 | 1 |
| 1 | 2 | 28 | 2 |
| 6 | 3 | 5 | 1 |

Инструкция ORDER BY, отраженная внутри OVER, не имеет отношения к конечному выводу и не влияет на реляционную природу результата. Если вы не укажете инструкцию ORDER BY на уровне запроса, порядок следования строк в результирующем наборе будет неопределенным.

1. Укажите порядок обработки следующих инструкций: SELECT, DISTINCT, TOP / OFFSET-FETCH, выражения вида YEAR(Date) и т.п., FROM, WHERE, GROUP BY, HAVING, ORDER BY.

Порядок обработки рассмотренных нами инструкций:

1) FROM;

2) WHERE;

3) GROUP BY;

4) HAVING;

5) SELECT;

6) выражения;

7) DISTINCT;

8) ORDER BY;

9) TOP / OFFSET-FETCH.

1. Какие из перечисленных инструкций НЕ являются предикатами: WHERE, GROUP BY, HAVING, CHECK, IN, CASE, BETWEEN, LIKE.

* GROUP BY, CASE;
* GROUP BY, HAVING, CHECK;
* GROUP BY, CHECK, CASE;
* GROUP BY, HAVING, CHECK, CASE, LIKE.

Язык T-SQL содержит элементы, в рамках которых можно задавать предикаты (предикаты являются логическими выражениями, которые возвращают одно из трех значений: TRUE, FALSE или UNKNOWN.), — это фильтры запросов вроде WHERE или ограничения HAVING, CHECK, а также IN, BETWEEN и LIKE.

1. Имеется таблица Orders с полем OrderId, в котором записаны значения: 1, 2, 3, 4, 5, 6. Что вернет следующий запрос

SELECT OrderId

FROM Orders

WHERE OrderId IN(2, 4, 5);

* 2, 4, 5;
* 1, 2;
* 2;
* 2, 3, 4, 5.

Предикат IN проверяет, равно ли значение (или скалярное выражение) как минимум одному элементу множества.

1. Имеется таблица Orders с полем Price, в котором записаны значения: 28, 80, 45, 15, 67, 5. Сколько строк вернет следующий запрос:

SELECT Price

FROM Orders

WHERE Price BETWEEN 50 AND 80;

* 2,
* 3,
* 1.

Предикат BETWEEN определяет, находится ли значение в диапазоне, заданном с помощью двух граничных значений (включая границы).

1. В запросе необходимо вывести результат деления содержимого двух целочисленных столбцов (типа INT). Какая из следующих строк даст корректный результат, чтобы в итоге получилось действительное число:

* CAST(col1 AS NUMERIC(12, 2)) / CAST(col2 AS NUMERIC(12, 2)),
* col1 / col2,
* col1 % col2.

Тип скалярного выражения, в котором участвуют два операнда, определяется на основе приоритета типов данных. Если операнды имеют один тип данных, тип результата будет соответствующим. Например, при делении двух целых чисел (INT) получается целое число. Выражение 5/2 возвращает 2, а не 2,5. При работе с константами это не вызывает проблем, потому что вы всегда можете представить значение в виде числа с запятой. Однако если вы делите содержимое двух целочисленных столбцов (скажем, col1/col2), необходимо привести операнды к подходящим типам, чтобы в результате получить действительное число:

CAST(col1 AS NUMERIC(12, 2))/CAST(col2 AS NUMERIC(12, 2)). Тип данных NUMERIC(12, 2) имеет точность 12 и масштаб 2, то есть он состоит из 12 цифр, две из которых находятся после запятой.

1. Чему равен результат SELECT 5 / 2:

* 2,
* 2.5,
* 0,25\*Е1.

1. Чему равен результат SELECT 5 / 2.0:

* 2.5,
* 2,
* 0,25\*Е1.

Если операнды имеют разные типы, выражение сводится к типу с большим приоритетом. Например, в выражении 5/2.0 первый операнд целочисленный (INT), а второй представлен числом с плавающей запятой (NUMERIC). Поскольку у типа NUMERIC более высокий приоритет по сравнению с INT, операнд 5 перед математической операцией автоматически превращается в действительное число (5,0). В итоге вы получите 2,5.

1. Чему равен результат SELECT '5' + 2:

* 7,
* 52,
* возникнет синтаксическая ошибка.

1. Чему равен результат SELECT 5 + '2a':

* возникнет синтаксическая ошибка,
* 52a,
* 7.

SQL Server может выполнять неявные преобразования от типа с меньшим приоритетом к типу с большим приоритетом. Таблица приоритетов (чем выше, тем больший приоритет):

|  |
| --- |
| datetime |
| smalldatetime |
| float |
| real |
| decimal |
| money |
| smallmoney |
| int |
| smallint |
| tinyint |
| bit |
| nvarchar |
| nchar |
| varchar |
| char |

В тех случаях, когда необходимо выполнить преобразования от типов с высшим приоритетом к типам с низшим приоритетом, то надо выполнять явное приведение типов. Для этого в T-SQL определены две функции: CONVERT и CAST.

1. Что вернет следующий запрос для строки со значением CategoryId = 5:

SELECT CategoryId,

CASE CategoryId

WHEN 1 THEN 'Beverage'

WHEN 2 THEN 'Spice'

WHEN 3 THEN 'Candy'

WHEN 4 THEN 'Milk'

END AS CategoryName

FROM Products;

* NULL,
* '' (пустую строку),
* 5,
* возникнет синтаксическая ошибка.

1. Что вернет следующий запрос для строки со значением Price = 100:

SELECT Price,

CASE

WHEN Price < 100 THEN 'Cheap'

WHEN Price > 100 THEN 'Expensive'

END AS PriceResult

FROM Products;

* NULL,
* '' (пустую строку),
* 100,
* возникнет синтаксическая ошибка.

1. Какую строку надо вставить в следующий запрос, чтобы для записей со значениями Price < 1 и Price > 10000 выводилось «So what's the price?»:

SELECT Price,

CASE

WHEN Price BETWEEN 1 AND 99 THEN 'Cheap'

WHEN Price BETWEEN 100 AND 999 THEN 'Expensive'

WHEN Price BETWEEN 1000 AND 9999 THEN 'Very Expensive'

END AS PriceResult

FROM Products;

ELSE 'So what's the price?'

Выражение CASE является скалярным и возвращает значение на основе условной логики.

У выражения CASE есть две формы: простая и поисковая. Простая форма сравнивает одно значение или скалярное выражение со списком возможных вариантов, возвращая первое совпадение. Если ни один из элементов списка не совпал с заданным значением, выражение CASE возвращает то, что указано в инструкции ELSE. Если инструкция ELSE отсутствует, возвращается NULL.

1. В некоторых строках в поле Quantity записаны значения NULL. Что вернет следующий запрос для таких строк:

SELECT Price, ISNULL(Quantity, 0) AS Quantity

FROM Products;

* 0,
* NULL,
* '' (пустую строку).

Функция ISNULL принимает два аргумента, возвращая тот, который первым идентифицируется как «не пустой» (не равен NULL); если оба аргумента равны NULL, возвращается NULL. Например, выражение ISNULL(col1, '') вернет col1, если значение не равно NULL; в противном случае результатом будет пустая строка.

Функция COALESCE работает аналогичным образом, только ей можно передавать два и более аргументов.

1. Могут ли функции ISNULL и COALESCE вернуть значение NULL?

* да, если все аргументы этих функций равны NULL;
* только COALESCE может вернуть значение NULL, если все ее аргументы NULL;
* только ISNULL может вернуть значение NULL, если все ее аргументы NULL;
* обе эти функции всегда возвращают не NULL.

1. Что вернет функция COALESCE(A, B, C, D), если

A = 5, B = NULL, C = 6, D = 1;

A = NULL, B = NULL, C = 'Hi', D = 'Bye';

A = NULL, B = NULL, C = NULL, D = NULL;

* 5, 'Hi', NULL;
* 5, NULL, NULL;
* 6, 'Hi', NULL;
* '' (пустую строку).

1. Что вернет функция CHOOSE (3, A, B, C, D)?

* C;
* 3;
* NULL;
* '' (пустую строку).

В языке T-SQL есть несколько функций, которые можно считать урезанными разновидностями выражения CASE: ISNULL, COALESCE, IIF и CHOOSE. Только COALESCE является стандартной. IIF и CHOOSE доступны исключительно в SQL Server 2012.

Функция CHOOSE (<индекс>, <выр1>, <выр2>, …, <вырn>) возвращает выражение из списка по заданному индексу. Например, CHOOSE(3, col1, col2, col3) вернет значение col3.

1. Таблица Customers содержит 10 строк:

5 строк, в которых Region = Ryazan,

3 строки, в которых Region = Moscow,

2 строки, в которых Region = NULL.

Сколько строк вернет следующий запрос

SELECT CustomerId, LastName, Region

FROM Customers

WHERE Region = NULL;

* NULL;
* 2;
* 0;
* 10.

1. Таблица Customers содержит 10 строк:

5 строк, в которых Region = Ryazan,

3 строки, в которых Region = Moscow,

2 строки, в которых Region = NULL.

Сколько строк вернет следующий запрос

SELECT CustomerId, LastName, Region

FROM Customers

WHERE Region <> ‘Ryazan’;

* 3;
* 5;
* 2;
* 10.

1. Являются ли взаимозаменяемыми следующие запросы:

SELECT CustomerId, LastName, Region

FROM Customers

WHERE Region = NULL;

и

SELECT CustomerId, LastName, Region

FROM Customers

WHERE Region IS NULL;

Если вы хотите получить строки, для которых регион равен NULL, не используйте предикат region = NULL, так как он всегда возвращает UNKNOWN, независимо от реального значения атрибута region. В этой ситуации необходимо применить предикат IS NULL.

1. Одинаков ли результат выражений WHERE(salary > 0) и CHECK(salary > 0) для строк, содержащих NULL?

Значения UNKNOWN обрабатываются по-разному в зависимости от контекста.

Например, фильтры в языке SQL принимают только TRUE, то есть значения FALSE и UNKNOWN всегда отсеиваются. Ограничение CHECK, наоборот, отклоняет только FALSE, в результате принимается не только значение TRUE, но и UNKNOWN. В двоичной системе исчисления предикатов между «принимать TRUE» и «отклонять FALSE» нет разницы. Однако мы работаем с троичной системой, поэтому предикат «принимать TRUE» отсеивает как UNKNOWN, так и FALSE, а выражение «отклонять FALSE» подразумевает, что UNKNOWN проходит проверку наравне с TRUE. Предикат salary > 0 вернет UNKNOWN, если salary равно NULL. Если поместить такой предикат в инструкцию WHERE, все строки, у которых salary равняется NULL, отсеются. Однако в ограничении CHECK он будет означать, что пустые строки должны приниматься.

Некоторые считают странным, что операция сравнения двух пустых значений (NULL = NULL) возвращает UNKNOWN. С точки зрения языка SQL отметка NULL представляет сущность, которая отсутствует либо не определена. Нельзя сказать с уверенностью, равны ли два неизвестных значения. В связи с этим в SQL предусмотрено два предиката — IS NULL и IS NOT NULL, которые необходимо использовать вместо выражений = NULL и <> NULL.

Различные элементы языка SQL по-разному обрабатывают отметки NULL при сравнении и сортировке. Иногда два значения NULL могут быть равными, а иногда нет. Например, при группировании и сортировке отметки NULL считаются одинаковыми, то есть инструкция GROUP BY поместит все пустые значения в одну группу. Аналогичные действия произойдут при использовании инструкции ORDER BY.

1. Дана таблица Table, которая содержит целочисленные столбцы col1 и col2. Необходимо получить все строки, для которых выражение col2/col1 больше 2 (при этом надо учесть, что в столбце col1 могут храниться нули, и появится ошибка деления на нуль). Корректен ли следующий запрос

SELECT col1, col2

FROM Table

WHERE col1 <> 0 AND col2/col1 > 2;

Язык SQL поддерживает концепцию одновременных операций. Это значит, что все выражения, которые находятся на одном этапе обработки, логически выполняются одновременно. Именно поэтому в инструкции SELECT нельзя ссылаться на псевдонимы столбцов, присвоенные в том же списке.

SQL Server поддерживает сокращенные вычисления, но из-за концепции одновременного выполнения выражения в инструкции WHERE могут обрабатываться в любом порядке. Обычно программа исходит из потенциальной производительности, то есть первым выполняется выражение, которое требует меньше всего системных ресурсов. Таким образом, если SQL Server решит, что сначала нужно обработать операцию col2/col1 > 2, запрос завершится неудачно.

Этих неприятностей можно избежать несколькими способами. Например, выражение CASE всегда выполняет вложенные инструкции WHEN последовательно.

1. Необходимо создать таблицу TodayFeeling. В ней будут три символьных поля: FirstName, LastName, Feeling (не надо обращать внимание на корректность архитектуры таблицы). Каждый день пользователь в строке со своим именем и фамилией может менять значение в поле Feeling. Какие типы данных следует выбрать для каждого поля?
2. FirstName NVARCHAR(50),

LastName NVARCHAR(50),

Feeling NCHAR(300)

1. FirstName NVARCHAR(50),

LastName NVARCHAR(50),

Feeling NVARCHAR(300)

1. FirstName NCHAR(50),

LastName NCHAR(50),

Feeling NCHAR(300)

* 1;
* 2;
* 3;
* любой из предложенных вариантов.

SQL Server поддерживает два вида символьных данных: обычные и представленные в кодировке Unicode. К первым относятся типы CHAR и VARCHAR, ко вторым — NCHAR и NVARCHAR. Для хранения обычного символа достаточно одного байта; в Unicode символы имеют разный размер (обычно два байта, но в случае с суррогатными парами может достигать четырех). Выбрав для столбца обычный тип, вы будете ограничены двумя языками, один из которых английский. Поддержка языков определяется правилами сравнения, которые действуют в столбце. Типы данных на основе кодировки Unicode поддерживают множество языков. Поэтому, если вы собираетесь хранить символы из разных алфавитов, используйте типы NCHAR или NVARCHAR.

Для записи строки, состоящей из обычных символов, используются одинарные кавычки: 'Это обычная строка'. При использовании кодировки Unicode необходимо указать префикс N: N'Это строка в формате Unicode'.

Любой тип данных, в названии которого нет приставки VAR (CHAR, NCHAR), имеет фиксированную длину. Это значит, что SQL Server выделяет место в строке, учитывая заранее определенный размер столбца, а не реальное количество символов. Например, если тип столбца CHAR(25), ему выделяется 25 символов, и не важно, строку какой длины он хранит. Такие типы больше подходят для систем, ориентированных на запись данных, поскольку строки таблицы не должны расширяться вместе с содержимым. В условиях когда основной операцией является чтение, появляется проблема неоптимального использования дискового пространства.

Типы данных с приставкой VAR в названии (VARCHAR, NVARCHAR) обладают переменной длиной: им выделяется столько места, сколько занимают хранимые в них строки (плюс два дополнительных байта для разделения столбцов). Допустим, столбец типа VARCHAR(25) хранит не более 25 символов, но на самом деле его размер определяется объемом содержимого. Поскольку эти типы данных потребляют меньше дискового пространства, они обеспечивают лучшую производительность при чтении. Однако во время обновления строки таблицы могут расширяться. В итоге данные выходят за пределы текущей страницы и, как следствие, замедляются операции записи.

1. Есть таблица Customers следующего вида

|  |  |  |
| --- | --- | --- |
| **Id** | **FirstName** | **LastName** |
| 1 | John | Lennon |
| 2 | John | Smith |
| 3 | john | doe |

Сколько строк вернет следующий запрос?

SELECT FirstName

FROM Customers

WHERE FirstName COLLATE Latin1\_General\_CS\_AS = ‘john’;

* 1;
* 2;
* 3.

Операция сравнения символьных данных включает поддержку языков, порядок сортировки, чувствительность к регистру и диакритическим знакам. Чтобы получить список поддерживаемых параметров и их описание, можно выполнить запрос к табличной функции fn\_helpcollations.

Создавая БД, вы можете переопределить эти параметры с помощью инструкции COLLATE.

Параметр Latin1\_General\_CI\_AS означает сравнение с учетом регистра.

1. Что вернет следующий запрос для строки, в которой FirstName = NULL?

SELECT FirstName + ‘ ’+ LastName

FROM Customers;

* NULL;
* значение поля LastName;
* пробел и значение поля LastName.

1. Что вернет следующий запрос для строки, в которой FirstName = NULL?

SELECT CONCAT(FirstName,‘ ’, LastName)

FROM Customers;

* пробел и значение поля LastName;
* значение поля LastName;
* NULL.

1. Будет ли одинаковым результат запросов

SELECT FirstName + ‘ ’+ LastName

и

SELECT CONCAT(FirstName,‘ ’, LastName)

для строки, в которой FirstName = NULL?

В стандартной разновидности языка SQL объединение со значением NULL возвращает NULL. SQL Server соблюдает это правило по умолчанию.

В SQL Server 2012 появилась новая функция под названием CONCAT. Она принимает список аргументов для их последующего объединения; замена отметок NULL пустыми строками здесь происходит автоматически. Например, выражение CONCAT('a', NULL, 'b') возвращает строку 'ab'.

1. Будет ли одинаковым результат для следующих запросов

SELECT SUBSTRING (FirstName, 1, 1)

FROM Customers;

и

SELECT LEFT(FirstName,1)

FROM Customers;

Функция SUBSTRING возвращает определенный фрагмент заданной строки. Синтаксис: SUBSTRING(строка, начало, длина).

Эта функция принимает входящую строку и извлекает из нее фрагмент заданной длины, начиная с определенной позиции. Следующий код возвращает 'abc':

SELECT SUBSTRING('abcde', 1, 3);

Если значение третьего аргумента выйдет за пределы исходной строки, функция ограничит результат последним символом, не выдавая ошибки. Это удобно, когда требуется фрагмент начиная с определенной позиции и до конца — вы просто указываете максимальную длину типа данных или значение, представляющее полную длину входящей строки.

Функции LEFT и RIGHT Являются сокращенными вариантами SUBSTRING. Они возвращают заданное количество символов, начиная с левого или правого края строки. Синтаксис: LEFT(строка, n), RIGHT(строка, n).

1. Будет ли одинаковым результат

SELECT LEN(N'Hello');

и

SELECT SELECT DATALENGTH(N'Hello');

Функции LEN(строка) возвращает количество символов в заданной строке.

Количество символов в строке не всегда совпадает с количеством байтов. В обычных строках каждый символ равен одному байту; в кодировке Unicode символы занимают в основном два байта. Таким образом, полученный результат может быть в два раза меньше, чем размер строки. Для измерения строк в байтах существует функция DATALENGTH. Например, следующий код вернет 5.

SELECT LEN(N'abcde');

А результатом этого выражения будет 10.

SELECT DATALENGTH(N'abcde');

Кроме того, в отличие от DATALENGTH функция LEN вырезает замыкающие пробельные символы.

1. Что вернет следующий запрос?

SELECT CHARINDEX(' ','John Lennon');

* 5;
* 4;
* 0.

Функция CHARINDEX возвращает позицию (не индекс!!!) первого вхождения одной строки в другую. Синтаксис:

CHARINDEX(подстрока, строка[, нач\_поз])

Функция возвращает местоположение первого аргумента (подстроки) во втором (строке). Вы также можете указать третий аргумент, чтобы сообщить позицию, с которой следует искать вхождения (по умолчанию поиск начинается с первого символа). Если подстрока не найдена, функция возвращает 0.

1. Что вернет следующий запрос?

SELECT PATINDEX('%[0-9]%', 'abcd123efgh');

* 5;
* 4;
* 0.

Функция PATINDEX возвращает позицию (не индекс!!!) первого вхождения шаблона в строку. Синтаксис:

PATINDEX(шаблон, строка)

1. Что вернет следующий запрос?

SELECT REPLACE('2021/07/29', '/', '.');

* '2021.07.29';
* '2021/07/29';
* '20210729'.

Функция REPLACE заменяет все вхождения одной строки другой строкой. Синтаксис:

REPLACE(строка, подстрока1, подстрока2)

Все вхождения подстроки1 в строку заменяются подстрокой2.

1. Что вернет следующий запрос?

SELECT REPLICATE('Hello! ', 3);

* 'Hello! Hello! Hello!';
* 'Hello!Hello!Hello!';
* 'Hel'.

Функция REPLICATE повторяет значение строки заданное количество раз. Синтаксис:

REPLICATE(строка, n)

1. Что вернет следующий запрос?

SELECT STUFF('It is a day!', 8, 0, ' beautiful');

* 'It is a beautiful day!';
* 'It is a beautiful';
* 'beautiful'.

Функция STUFF удаляет из строки заданный фрагмент и вставляет на его место новый. Синтаксис:

STUFF(строка, позиция, длина\_фрагмента, новый\_фрагмент)

Эта функция оперирует исходной строкой. Она удаляет последовательность символов определенной длины, начиная с указанной позиции. Затем на то же место помещает новый фрагмент.

Если вы хотите вставить строку, ничего не удаляя, в качестве третьего аргумента укажите 0.

1. Что вернет следующий запрос?

SELECT RTRIM(LTRIM(UPPER(' Web Web ')));

* 'WEB WEB';
* 'WEBWEB';
* 'Web web'.

Функции UPPER и LOWER переводят заданную строку в верхний или нижний регистр соответственно.

Функции RTRIM и LTRIM возвращают исходную строку, удаляя из нее начальные или завершающие пробелы соответственно.

1. Что вернет следующий запрос?

SELECT FORMAT(1941, '0000000000');

* 0000001941;
* 00000000001941;
* 1941.

Функция FORMAT преобразовывает заданное значение, используя региональные параметры и формат, принятый на платформе Microsoft .NET. Синтаксис:

FORMAT(ввод, формат, региональные\_параметры)

Существует множество вариантов, как преобразовать исходное значение; для этого можно использовать как стандартные, так и собственные строки форматирования. Больше информации на эту тему вы найдете в статье по адресу go.microsoft.com/fwlink/?LinkId=211776.

1. Будет ли одинаковым результат следующих запросов

SELECT OrderId, OrderDate

FROM Orders

WHERE OrderDate = '20070212';

и

SELECT OrderId, OrderDate

FROM Orders

WHERE OrderDate = CAST('20070212' AS DATETIME);

* да, в зависимости от текущей языковой среды,
* да,
* нет,
* возникнет синтаксическая ошибка.

SQL Server воспринимает литерал '20070212' как строковую константу, а не как обозначение даты и времени. Однако в выражении участвуют операнды двух разных типов, поэтому один из них должен автоматически трансформироваться в другой. Обычно такое неявное преобразование выполняется с учетом приоритетов типов данных, определенных на уровне SQL Server: операнд автоматически приводится к более приоритетному типу.

В примере строковый литерал преобразуется в тип данных столбца (DATETIME), потому что строка менее приоритетна, чем дата и время. Правила неявного приведения типов не всегда работают так просто — например, свои особенности есть у фильтров и других выражений. Полное описание приоритетов типов данных вы найдете в электронном справочнике в разделе «Приоритет типов данных (Transact-SQL)». А неявное приведение типов выполняется автоматически.

1. Будет ли одинаковым результат

SET LANGUAGE Russian;

SELECT CAST('02/12/2007' AS DATETIME);

и

SET LANGUAGE us\_english;

SELECT CAST('02/12/2007' AS DATETIME);

В языке us\_english используется формат mdy, а в языке Russian — dmy.

Параметры LANGUAGE/DATEFORMAT влияют только на интерпретацию вводимых значений. Они не относятся к формату представления даты и времени, который определяется в интерфейсе БД с помощью клиентских инструментов (таких как ODBC). Например, OLEDB и ODBC выводят значения типа DATETIME в формате 'YYYY-MM-DD hh:mm:ss.nnn'.

Обратите внимание на несколько моментов. Если в составном типе (который хранит дату и время) не указать значение времени, SQL Server автоматически поставит «полночь». Если опустить часовой пояс, будет предполагаться, что он равен 00:00. Форматы 'YYYY-MM-DD' и 'YYYY-MM-DD hh:mm...' являются нейтральными для типов DATETIME и SMALLDATETIME, но зависят от языка, если литерал преобразовывается в типы DATE, DATETIME2 или DATETIMEOFFSET.

Если вы хотите следовать формату, принятому в определенном языке, существует два варианта. Первый — конвертировать строковые литералы в нужные типы данных с помощью функции CONVERT. При этом в качестве ее третьего аргумента указывается число, представляющее выбранный стиль. В электронном справочнике по SQL Server в разделе «Функции CAST и CONVERT (Transact-SQL)» приведена таблица со всеми значениями и описанием форматов, которые они представляют. Например, чтобы указать литерал '02/12/2007' в формате mm/dd/yyyy, используйте число 101.

SELECT CONVERT(DATETIME, '02/12/2007', 101);

Литерал будет интерпретирован как 12 февраля 2007 г. независимо от действующих языковых настроек.

Если необходимо использовать формат dd/mm/yyyy, укажите стиль 103.

SELECT CONVERT(DATETIME, '02/12/2007', 103);

На этот раз мы получим 2 декабря 2007 г.

1. Будет ли одинаковым результат

SELECT OrderId, OrderDate

FROM Orders

WHERE OrderDate = '20070212';

и

SELECT OrderId, OrderDate

FROM Orders

WHERE OrderDate >= '20070212'

AND OrderDate < '20070213';

Если поле OrderDate в таблице Orders имеет тип DATETIME, но нас интересуют только даты, поэтому все значения «были записаны» в полночь. Если необходимо отобрать заказы с определенной датой, вы можете обойтись без фильтрации по диапазону. Для этой задачи подойдет оператор равенства.

Преобразовывая строковый литерал в тип DATETIME, SQL Server считает, что полночь означает отсутствующее время. Так как все значения в столбце orderdate указывают на полночь, запрос вернет все заказы с заданной датой. Чтобы разрешить хранение только тех дат, у которых временная часть равна полночи, воспользуйтесь ограничением CHECK.

Если время принимает другое значение, вы можете выполнить фильтрацию по диапазону.

1. Будет ли одинаковым результат

SELECT OrderId, OrderDate

FROM Orders

WHERE YEAR(OrderDate) = 2017;

и

SELECT OrderId, OrderDate

FROM Orders

WHERE OrderDate >= '20170101'

AND OrderDate < '20180101';

Когда необходимо отфильтровать определенный диапазон дат (например, месяц или год), логично использовать функции YEAR и MONTH.

Однако, используя функции в условии WHERE, SQL Server не обеспечивает эффективного использования индексов. Поэтому результат работы хоть и будет одинаковый, но по эффективности/производительности второй запрос лучше.

1. Будет ли одинаковым результат

SELECT CONVERT(DATETIME, '02/12/2007', 101);

и

SELECT PARSE('02/12/2007' AS DATETIME USING 'en-US');

Функции CAST, CONVERT и PARSE используются, чтобы привести значения к определенным типам. Если преобразование прошло успешно, возвращается результат; в противном случае запрос завершается ошибкой. У каждой из этих функций существуют аналоги с префиксом TRY\_: TRY\_CAST, TRY\_CONVERT и TRY\_PARSE. Они содержат те же аргументы и выполняют идентичные действия, но в случае неудачного преобразования запрос возвращает значение NULL.

Функции TRY\_CAST, TRY\_CONVERT, PARSE и TRY\_PARSE появились в SQL Server 2012.

CAST / TRY\_CAST (значение AS тип)

CONVERT / TRY\_CONVERT (тип, значение [, номер\_стиля])

PARSE / TRY\_PARSE (значение AS тип [USING регион])

1. Укажите результат следующего запроса

SELECT DATEADD (YEAR, 1, '20090729');

* 29 июля 2010,
* 30 июля 2009,
* 2009 год.

Функция DATEADD добавляет заданный интервал в определенную часть даты и времени. Синтаксис: DATEADD(часть, интервал, значение\_даты).

1. Укажите результат следующего запроса

SELECT DATEDIFF(DAY, '20200212', '20210212');

* 366,
* 365,
* 1.

Функция DATEDIFF возвращает разницу между двумя значениями даты и времени в определенных единицах (годы, месяцы, дни и т. д.). Синтаксис: DATEDIFF(единицы, значение1, значение2).

Функция DATEPART возвращает целое число, представляющее определенную часть значения даты и времени.

YEAR, MONTH и DAY — упрощенные версии функции DATEPART. Они возвращают целое число, которое представляет определенную часть (год, месяц или день) значения даты и времени.

1. Что вернет следующий запрос

SELECT EOMONTH(SYSDATETIME());

* последнюю дату текущего месяца,
* последнюю дату текущего года,
* 30 или 31.

Функция EOMONTH в качестве аргумента она принимает дату и время и возвращает значение типа DATE, представляющее полночь последнего дня соответствующего месяца. Существует и необязательный аргумент, посредством которого можно добавить определенное количество месяцев. Синтаксис:

EOMONTH(дата [, дополнительные\_месяцы])

1. Что делает функция EXEC sys.sp\_tables?

* возвращает список объектов (таблиц и представлений), к которым можно получить доступ в текущей БД,
* возвращает структуру текущей БД,
* возникнет синтаксическая ошибка.

Системные хранимые процедуры и функции выполняют внутренние запросы к каталогу, предоставляя систематизированный набор метаданных.

Хранимая процедура sp\_tables возвращает список объектов (таблиц и представлений), к которым можно получить доступ в текущей БД.

SQL Server содержит инструменты, которые позволяют получить метаданные таких объектов, как таблицы в БД и столбцы в таблице. Процедура sp\_help принимает в качестве аргумента название объекта и возвращает информацию о нем, его столбцах, индексах, ограничениях и т. д. Следующий код вернет несколько результирующих наборов с подробными данными о таблице Orders.

EXEC sys.sp\_help @objname = N'Sales.Orders';

Процедура sp\_columns возвращает информацию о столбцах объекта.

Процедура sp\_helpconstraint возвращает информацию об ограничениях объекта.

## Глава 3. Соединения

1. Имеется таблица Table1, содержащая поле Number со значениями 1, 2, 3, 4, и таблица Table2, содержащая поле Letter со значениями A, B, C. Сколько строк вернет следующий запрос:

SELECT T1.Number, T2.Letter

FROM Table1 T1

CROSS JOIN Table2 T2;

* 12,
* 7,
* 4.

С точки зрения логики перекрестное соединение является простейшим видом оператора JOIN. Оно реализует всего один этап обработки — декартово произведение. В нем участвуют две таблицы, которые в результате сопоставления строк превращаются в одну. Таким образом, если в первой таблице m строк, а во второй n, то результат будет содержать m \* n строк.

1. Имеется таблица Digits, состоящая из одного поля Digit, в котором записаны цифры от 0 до 9. Напишите запрос, который выводит числа от 0 до 99 (с использованием самосоединения).

SELECT D2.Digit \* 10 + D1.Digit AS Number

FROM Digits D1

CROSS JOIN Digits D2

ORDER BY Number;

1. Имеется таблица Customers, состоящая из полей CustomerId, FirstName, LastName, Address. Таблица Products с полями ProductId, Name, Price. И таблица Orders, которая состоит из столбцов OrderId, CustomerId, ProductId, OrderDate. Напишите запрос для всех заказов, который выводит значения столбцов OrderId, OrderDate, Name, LastName + FirstName.

SELECT O.OrderId

,O.OrderDate

,P.Name

,CONCAT(C.FirstName + C.LastName)

FROM Orders O

JOIN Customers C ON C.CustomerId = O.CustomerId

JOIN Products P ON P.ProductId = O.ProductId;

Для создания внутреннего соединения подразумевает наличие между именами таблиц ключевого слова INNER JOIN. На самом деле слово INNER можно опустить, поскольку любое соединение по умолчанию является внутренним. Чтобы указать предикат, который будет фильтровать строки (условие соединения), используйте инструкцию ON.

Суть данного запроса можно переформулировать так: мы сопоставляем все записи о заказах, у которых совпадает идентификатор заказа. Это помогает понять принцип работы соединения. Но к подобному вопросу можно подойти более формально, используя концепции реляционной алгебры: сначала выполняется декартово произведение двух таблиц, затем строки фильтруются по предикату C.CustomerId = O.CustomerId.

Вспомните предыдущую главу, в которой рассматривалось исчисление предиката в троичной системе. Инструкция ON по аналогии с WHERE и HAVING возвращает только те строки, для которых предикат является истинным (TRUE); при любом другом результате выполнения предиката (FALSE или UNKNOWN) строки отбрасываются.

1. Имеется таблица Orders, которая состоит из столбцов OrderId, CustomerId, ProductId, OrderDate. Таблица Products с полями ProductId, Name, Price. И таблица Dicounts, которая состоит из столбцов OrderId, ProductId, Discount. Напишите запрос, который выводит скидки для всех заказов и отображает столбцы OrderId, OrderDate, Name, Price, Discount.

SELECT O.OrderId

,OrderDate

,P.Name

,P.Price

,D.Discount

FROM Orders O

JOIN Discounts D ON D.OrderId = O.OrderId

AND D.ProductId = O.ProductId

JOIN Products P ON P.ProductId = O.ProductId;

Составным называют обычное соединение, предикат которого содержит больше одного атрибута с каждой стороны. Этот вид соединений обычно использует составное отношение первичного и внешнего ключей (отношение, основанное сразу на нескольких атрибутах).

1. Имеется таблица Users, которая состоит из столбцов UserId, FirstName, LastName. Напишите запрос, который выводит все уникальные пары пользователей. В результате должны отображаться два поля: FullName User1, FullName User2.

SELECT

CONCAT(U1.FirstName, ‘ ’, U1.LastName) AS [FullName User1]

,CONCAT(U2.FirstName, ‘ ’, U2.LastName) AS [FullName User2]

FROM Users U1

JOIN Users U2 ON U1.UserId < U2.UserId;

Соединение, в условии которого участвует только оператор равенства, называют эквисоединением. Условие, которое записывается с помощью любых других операторов, называется не-эквисоединением.

1. Имеется таблица Numbers, которая состоит из одного столбца Number, в каждой строке которого записаны цифры от 1 до 9 (всего 9 строк). И таблица Letters, в которой 2 столбца: Number, Symbol. В столбце Letter в каждой строке записана буква латинского алфавита от A до D, а в столбце Number – порядковый номер буквы в алфавите (всего 4 строки). Будут равнозначны следующие запросы

SELECT L.Symbol, N.Number

FROM Numbers N

CROSS JOIN Letters L;

и

SELECT L.Symbol, N.Number

FROM Numbers N

FULL JOIN Letters L ON L.Number = N.Number;

1. Имеется таблица Customers, состоящая из полей CustomerId, FirstName, LastName, Address. И таблица Orders, которая состоит из столбцов OrderId, CustomerId, ProductId, OrderDate. Напишите запрос, который выводит всех покупателей и их заказы (даже если они не делали заказов). Необходимо вывести значения для столбцов LastName + FirstName, OrderId, OrderDate.

SELECT CONCAT(C.FirstName + C.LastName)

,O.OrderId

,O.OrderDate

FROM Customers C

LEFT JOIN Orders O ON C.CustomerId = O.CustomerId;

1. Имеется таблица Customers, состоящая из полей CustomerId, FirstName, LastName, Address. И таблица Orders, которая состоит из столбцов OrderId, CustomerId, ProductId, OrderDate. Напишите запрос, который выводит всех покупателей (LastName + FirstName), которые не сделали ни одного заказа.

SELECT CONCAT(C.FirstName + C.LastName)

FROM Customers C

LEFT JOIN Orders O ON C.CustomerId = O.CustomerId

WHERE C.OrderId IS NULL;

Внешние соединения: ключевое слово JOIN находится между именами таблиц, а условие обозначается инструкцией ON.

Внешнее соединение состоит из тех же двух логических этапов, что и внутреннее (декартово произведение и фильтрация), плюс третий уникальный этап, на котором в результирующий набор добавляются внешние строки.

Чтобы оставить таблицы в результирующем наборе, между их именами указываются ключевые слова LEFT OUTER JOIN, RIGHT OUTER JOIN или FULL OUTER JOIN. Слово OUTER является необязательным. Слова LEFT и RIGHT означают, что в результат попадают строки из соответственно левой или правой таблиц; чтобы сохранить строки из обеих сторон, используется слово FULL. На третьем этапе логической обработки внешнее соединение использует предикат ON, чтобы определить записи, для которых не нашлось соответствия в другой таблице (это и есть внешние строки). Позже данные записи добавляются к тем, что уже были отобраны на двух предыдущих этапах; всем атрибутам с противоположной стороны, для которых не нашлось соответствия, устанавливается значение NULL.

1. Возможна ли такая ситуация, когда в результате запроса в столбце Id, который во время создания таблицы был помечен как NOT NULL, будут значения NULL.

Во внешних запросах важен атрибут, используемый для фильтрации, который берется из той таблицы, что не полностью попадает в результирующий набор. Нужно отдать предпочтение атрибуту, который во внешней строке может содержать исключительно отметку NULL (другие значения, берущиеся из основной таблицы, не допускаются). Таким образом, у нас есть три варианта: столбец первичного ключа, столбец, по которому выполняется соединение, и столбец, определенный с помощью ограничения NOT NULL. Первичный ключ не может равняться NULL; если его атрибут содержит такое значение, это свидетельствует только об одном — строка является внешней. Если атрибут, по которому выполняется соединение, содержит NULL, строка отбрасывается на втором этапе, когда применяется предикат; иными словами, это также будет признаком внешней строки. То же самое можно сказать о столбце с ограничением NOT NULL, которому присваивается NULL.

1. Имеется таблица Customers, состоящая из полей CustomerId, FirstName, LastName, Address. И таблица Orders, которая состоит из столбцов OrderId, CustomerId, ProductId, OrderDate. Следующий запрос выводит всех покупателей (LastName + FirstName), которые сделали заказ после 17 февраля 2021 года. Есть ли логическая ошибка в этом запросе?

SELECT CONCAT(C.FirstName + C.LastName)

FROM Customers C

LEFT JOIN Orders O ON C.CustomerId = O.CustomerId

WHERE C.OrderDate > ‘20211702’;

При поиске логических ошибок в коде, содержащем внешние соединения, особое внимание следует уделять инструкции WHERE. Если предикат, который в ней указан, ссылается на атрибут из противоположной стороны соединения и при этом используется выражение вида <атрибут> <оператор> <значение>, это обычно свидетельствует о некорректности запроса. Дело в том, что атрибуты, взятые из противоположной стороны соединения, во внешних строках равны NULL, а выражение NULL <оператор> <значение> возвращает UNKNOWN (разве что указан оператор IS NULL, который ищет отметки NULL). Напомним, что инструкция WHERE отбрасывает значения UNKNOWN. В итоге такой предикат отфильтровывает все внешние строки, нивелируя действие внешнего соединения. Другими словами, внешнее соединение превращается во внутреннее. Получается, что программист допустил ошибку либо при выборе типа соединения, либо в предикате.

1. Будут ли эквивалентны результаты следующих запросов:

SELECT CONCAT(C.FirstName + C.LastName)

FROM Customers C

LEFT JOIN Orders O ON C.CustomerId = O.CustomerId;

и

SELECT CONCAT(C.FirstName + C.LastName)

FROM Orders O

RIGHT JOIN Customers C ON C.CustomerId = O.CustomerId;

1. Имеется таблица Customers, состоящая из полей CustomerId, FirstName, LastName, Address. И таблица Orders, которая состоит из столбцов OrderId, CustomerId, ProductId, OrderDate. Следующий запрос выводит количество заказов для каждого покупателя. Есть ли ошибка в этом запросе? Что будет выведено в поле NumOrders для покупателей, которые не сделали ни одного заказа?

SELECT C.CustomerId, COUNT(\*) AS NumOrders

FROM Customers C

LEFT JOIN Orders O ON C.CustomerId = O.CustomerId

GROUP BY C.CustomerId;

Еще одна распространенная логическая ошибка связана с совместным использованием внешних соединений и функции COUNT. Когда вы группируете результат внешнего соединения и агрегируете его с помощью выражения COUNT(\*), SQL Server обрабатывает как внешние, так и внутренние строки, независимо от их содержания. Обычно внешние строки не должны учитываться при подсчете количества элементов в результирующем наборе.

Однако агрегатная функция COUNT(\*) подсчитает все строки, независимо от их содержимого. Чтобы решить эту проблему, вместо COUNT(\*) нужно использовать выражение COUNT(<столбец>), указывая столбец из противоположной стороны соединения. В результате внешние строки будут игнорироваться, потому что в данном столбце они содержат значения NULL. При этом нужно использовать атрибут, который во внешних строках всегда равен NULL, например столбец первичного ключа.

## Глава 4. Вложенные запросы

Язык SQL поддерживает написание вложенных запросов. Запрос, который возвращает конечный результат, называют внешним. Внутри себя он может использовать результат выполнения других запросов. Внутренние запросы вызываются во время выполнения кода и являются аналогами выражений, основанных на переменных или константах. В отличие от выражений их результат может изменяться в зависимости от содержимого таблицы. Использование вложенных запросов избавляет от необходимости хранить промежуточные результаты в отдельных переменных.

Вложенные запросы могут быть либо автономными, либо коррелирующими.

Первые, в отличие от вторых, никак не связаны со своим внешним кодом. Результат вложенного запроса может состоять из одного или нескольких значений или же вовсе являться таблицей.

1. Будут ли одинаковы результаты следующих запросов?

DECLARE @MaxId AS INT =

(SELECT MAX(OrderId)

FROM Orders);

SELECT OrderId, OrderDate

FROM Orders

WHERE OrderId = @MaxId;

и

SELECT OrderId, OrderDate

FROM Orders

WHERE OrderId =

(SELECT MAX(OrderId)

FROM Orders);

* да,
* нет,
* возникнет синтаксическая ошибка.

Любой вложенный запрос вызывается извне, однако автономные не зависят от внешнего кода. Их очень удобно отлаживать, вы всегда можете выполнить их отдельно и посмотреть, делают ли они то, что нужно. Сначала выполняется вложенный запрос, а потом внешний код использует полученный результат в своих вычислениях.

1. Корректен ли следующий запрос?

SELECT OrderId, OrderDate

FROM Orders

WHERE OrderDate =

(SELECT MAX(OrderDate)

FROM Orders);

* да,
* нет,
* может возникнуть синтаксическая ошибка.

Скалярный вложенный запрос всегда возвращает одно значение независимо от того, автономный он или коррелирующий. Он может находиться в любом месте внешнего кода, где допустимо использование скалярных выражений (например, в инструкциях WHERE или SELECT). Скалярный вложенный запрос должен возвращать не больше одного значения, в противном случае он может завершиться ошибкой во время выполнения кода.

1. Что вернет следующий запрос, если в таблице Customers нет покупателей, чьи фамилии начинаются на букву А?

SELECT OrderId, OrderDate, CustomerId

FROM Orders

WHERE CustomerId =

(SELECT CustomerId

FROM Customers

WHERE LEFT(LastName, 1) = N‘A’);

* пустой набор,
* NULL,
* возникнет синтаксическая ошибка.

Если у скалярного вложенного запроса нет результата, он возвращает NULL. Сравнение NULL с любым значением дает UNKNOWN, в результате чего запись не проходит фильтр и отбрасывается.

1. Корректен ли следующий запрос?

SELECT OrderId, OrderDate, CustomerId

FROM Orders

WHERE CustomerId IN

(SELECT CustomerId

FROM Customers

WHERE LEFT(LastName, 1) = N‘A’);

* да,
* нет,
* может возникнуть синтаксическая ошибка.

Существуют вложенные запросы (не важно, автономные или нет), которые возвращают несколько значений в виде единого столбца. Для работы с ними потребуются специальные предикаты, например инструкция IN.

Есть и другие предикаты, поддерживающие вложенные запросы с множественными значениями (SOME, ANY и ALL). Но они редко используются на практике.

1. Будут ли одинаковы результаты следующих запросов?

SELECT OrderId, OrderDate, CustomerId

FROM Orders

WHERE CustomerId IN

(SELECT CustomerId

FROM Customers

WHERE LEFT(LastName, 1) = N‘A’);

и

SELECT O.OrderId, O.OrderDate, C.CustomerId

FROM Orders O

JOIN Customers C ON C.CustomerId = O.CustomerId

WHERE LEFT(C.LastName, 1) = N‘A’;

* да,
* нет,
* возникнет синтаксическая ошибка.

Существует множество ситуаций, в которых можно использовать как вложенные запросы, так и соединения. Нельзя сказать однозначно, какой из этих подходов является более удачным. Иногда они интерпретируются совершенно одинаково, иногда проявляются отличия в производительности. Я предпочитаю решать задачу самым очевидным способом и только потом, если меня не устраивает скорость работы, пытаюсь видоизменить свой запрос, например заменить вложенные запросы на соединения (и наоборот).

1. Имеется таблица Orders, которая состоит из столбцов OrderId, CustomerId, ProductId, OrderDate. Напишите запрос, который для всех покупателей, сделавших заказ, выводит дату последнего заказа (вывести поля CustomerId и OrderDate).

SELECT CustomerId, OrderDate

FROM Orders O1

WHERE OrderDate =

(SELECT MAX(O2.OrderDate)

FROM Orders O2

WHERE O2.CustomerId = O1.CustomerId);

Коррелирующие вложенные запросы ссылаются на атрибуты таблицы, указанной во внешнем коде, то есть вложенный запрос зависит от внешнего кода и не может работать самостоятельно. С логической точки зрения это выглядит так, как будто вложенный запрос запускается отдельно для каждой внешней строки.

Коррелирующие вложенные запросы, в отличие от автономных, зависят от внешнего кода. Их сложнее отлаживать, поскольку нельзя запускать отдельно. Чтобы отлаживать коррелирующие вложенные запросы, вместо корреляций нужно подставлять константы; убедившись, что код работает корректно, вы сможете произвести обратную замену.

1. Имеется таблица Customers, состоящая из полей CustomerId, FirstName, LastName, Region. И таблица Orders, которая состоит из столбцов OrderId, CustomerId, ProductId, OrderDate. Напишите запрос с использованием предиката EXISTS, который выводит всех покупателей из региона Ryazan, которые делали заказы.

SELECT CONCAT(C.LastName, ‘ ’, FirstName) AS Name

FROM Customers C

WHERE C.Region = N‘Ryazan’

AND EXIST

(SELECT \*

FROM Orders O

WHERE C.CustomerId = O.CustomerId);

Язык T-SQL поддерживает предикат под названием EXISTS, который возвращает TRUE или FALSE в зависимости от того, генерирует ли переданный ему вложенный запрос какие-либо строки.

Одно из преимуществ предиката EXISTS заключается в том, что он позволяет формировать запросы в виде фраз на английском языке.

Как и другие предикаты, инструкцию EXISTS можно использовать в сочетании с логическим оператором отрицания NOT.

Предикат EXISTS хорошо поддается оптимизации. Ядро Microsoft SQL Server знает, что не нужно обрабатывать все запрашиваемые строки — достаточно получить хотя бы один результат (или его отсутствие).

Это тот редкий случай, когда в использовании символа звездочки (\*) внутри инструкции SELECT нет ничего плохого. Предикат EXISTS следит исключительно за наличием подходящих строк, игнорируя атрибуты, указанные для выборки, словно инструкция SELECT является лишней. Ядро SQL Server знает об этом и в целях оптимизации пропускает обработку списка SELECT внутри вложенного запроса. Поэтому использование группового символа \* вместо перечисления констант не влияет на производительность. Справедливости ради нужно отметить, что на процесс поиска столбцов все же могут быть выделены незначительные ресурсы, но вряд ли вы это заметите.

Следует упомянуть об еще одной интересной особенности инструкции EXISTS: в отличие от большинства других предикатов в языке T-SQL она использует двоичную логику. Действительно, не существует такой ситуации, в которой на вопрос «возвращает ли код строки?» нельзя было бы ответить однозначно.

1. Имеется таблица Numbers, которая состоит из одного столбца Number, в каждой строке которого записаны натуральные числа (уникальные в пределах этого столбца). Например, 5, 16, 11, 107, 45… Напишите запрос, который выводит два столбца:
2. в столбце CurrentNumber записано число из столбца Number,
3. в столбце PreviousNumber записано предыдущее число для числа из столбца Number.

Необходимо упорядочить результаты по столбцу CurrentNumber.

SELECT N1.Number AS CurrentNumber

,(SELECT MAX(N2.Number

FROM Numbers N2

WHERE N1.Number > N2.Number)

AS PreviousNumber

FROM Numbers N1

ORDER BY CurrentNumber;

1. Имеется таблица Numbers, которая состоит из одного столбца Number, в каждой строке которого записаны натуральные числа (уникальные в пределах этого столбца). Например, 5, 16, 11, 107, 45… Напишите запрос, который выводит два столбца:
2. в столбце CurrentNumber записано число из столбца Number,
3. в столбце NextNumber записано следующее число для числа из столбца Number.

Необходимо упорядочить результаты по столбцу CurrentNumber.

SELECT N1.Number AS CurrentNumber

,(SELECT MIN(N2.Number

FROM Numbers N2

WHERE N1.Number < N2.Number)

AS NextNumber

FROM Numbers N1

ORDER BY CurrentNumber;

Само понятие «предыдущий» подразумевает наличие какой-то логической упорядоченности. Так как строки в таблице не имеют никакого порядка, вам необходимо составить выражение на языке T-SQL, которое будет его эмулировать. За основу можно взять принцип, согласно которому предыдущим является «максимальное значение из тех, которые меньше текущего». Точно так же можно сказать, что следующая запись — это «минимальное значение из тех, которые больше текущего».

В SQL Server 2012 появились новые оконные функции, LAG и LEAD, которые позволяют возвращать элемент из предыдущей или следующей строки, исходя из определенного порядка сортировки.

1. Имеется таблица Customers, состоящая из полей CustomerId, FirstName, LastName, Region. И таблица Orders, которая состоит из столбцов OrderId, CustomerId, ProductId, OrderDate. Что будет выведено в результате следующего запроса (который находит всех покупателей без заказов), если в таблице Orders есть записи, в которых CustomerId = NULL?

SELECT CustomerId

,CONCAT(C.LastName, ‘ ’, FirstName) AS Name

FROM Customers

WHERE CustomerId NOT IN

(SELECT O.CustomerId

FROM Orders O);

* пустой набор,
* NULL,
* список покупателей без заказов,
* возникнет синтаксическая ошибка.

Запрос вернет пустой набор. Очевидно, виновником в данной ситуации является запись с неопределенным идентификатором CustomerId.

Оператор NOT инвертирует предикат IN; следовательно, NOT TRUE превращается в FALSE, а для отметок NULL – в UNKNOWN. Выражение FALSE OR UNKNOWN тоже дает UNKNOWN.

В качестве более наглядного примера возьмем выражение 22 NOT IN (1, 2, NULL). Его можно записать как NOT 22 IN (1, 2, NULL) или NOT (22 = 1 OR 22 = 2 OR 22 = NULL). Заменим все операторы равенства, находящиеся в скобках, их значениями истинности; получится выражение NOT (FALSE OR FALSE OR UNKNOWN), которое в свою очередь превращается в NOT UNKNOWN или просто UNKNOWN. Результат UNKNOWN до применения к нему оператора NOT означает, что нельзя доподлинно установить, входит ли идентификатор клиента в множество, поскольку значение NULL может быть чем угодно, в том числе этим идентификатором. Вся хитрость заключается в том, что отрицание значения UNKNOWN тоже дает UNKNOWN

Проще говоря, если вы используете предикат NOT IN в сочетании с вложенным запросом, который возвращает хотя бы одно значение NULL, внешний код всегда будет возвращать пустой набор. Значения таблицы, которые точно встречаются в наборе, отсеиваются, поскольку внешний код должен вернуть записи, которых в этом наборе нет. Не попадают в результат и те строки, нахождение которых в наборе не определено, потому что нельзя сказать наверняка, входит ли значение в множество, одним из элементов которого является отметка NULL.

Как же избежать подобной проблемы?

Во-первых, если столбец не должен принимать значения NULL, нужно обязательно указать для него ограничение NOT NULL. Обеспечение целостности данных играет намного более важную роль, чем принято считать.

Во-вторых, при написании любых запросов необходимо помнить, что вы имеете дело с тремя значениями истинности (TRUE, FALSE и UNKNOWN). Обращайте внимание на возможность появления отметок NULL и на то, подходит ли ваш код для работы с ними (если нет, придется вносить изменения).

Вместо NOT IN безопаснее использовать NOT EXISTS. Напомню, что в отличие от инструкции IN предикат EXISTS использует исчисление в двоичной системе. Он может вернуть либо TRUE, либо FALSE (значение UNKNOWN исключено).

## Глава 5. Табличные выражения

**Табличными** называют выражения, результат выполнения которых является корректным с точки зрения реляционной модели. Такие выражения можно подставлять в различные инструкции запроса вместо имен таблиц.

Табличные переменные (table variable) позволяют сохранить содержимое целой таблицы.

DECLARE @ABrends TABLE (ProductId INT,  ProductName NVARCHAR(20))

INSERT INTO @ABrends

VALUES(1, 'iPhone 8'),

(2, 'Samsumg Galaxy S8')

SELECT \* FROM @ABrends

Однако следует учитывать, что такие переменные не полностью эквивалентны таблицам. Они живут в пределах одного пакета, после завершения работы которого они удаляются. То есть они носят временный характер, и физически их данные нигде не хранятся на жестком диске.

**Производные таблицы**

Производные таблицы (их еще называют **вложенными табличными запросами**) объявляются в инструкции FROM в рамках внешнего запроса, где они и существуют. Как только запрос завершает свою работу, производные таблицы исчезают. Определение производной таблицы содержится внутри круглых скобок, а ее имя указывается с помощью инструкции AS. Например,

SELECT \*

FROM (SELECT custid, companyname

FROM Sales.Customers

WHERE country = N'США') AS USACusts;

Чтобы запрос можно было считать корректным табличным выражением, он должен отвечать трем основным требованиям:

1. порядок не гарантируется. Табличное выражение должно возвращать реляционный результат, что подразумевает неупорядоченность строк. Напомню, что это одна из фундаментальных особенностей реляционной модели, относящаяся к теории множеств. В связи с этим стандартная разновидность языка SQL не поддерживает использование инструкции ORDER BY в запросах, с помощью которых объявляются табличные выражения; исключение составляют ситуации, когда данная инструкция не участвует в выводе конечного результата (например, в случае использования фильтра OFFSET-FETCH). Похожие правила действуют и в языке T-SQL, только к списку исключений добавляется еще и нестандартный фильтр TOP. Инструкция ORDER BY может служить частью определения выражений TOP или OFFSET-FETCH; в этом случае она занимается исключительно фильтрацией и не участвует в выводе конечного результата. Если инструкцию ORDER BY не указать в конце внешнего запроса, упорядоченность результирующего набора не гарантируется.
2. у всех столбцов должны быть имена. Каждый столбец должен как-то называться; то есть любым табличным выражениям, которые объявляются в списке SELECT, нужно присваивать псевдонимы.
3. имена всех столбцов должны быть уникальными. Все столбцы в таблице следует называть по-разному. Табличное выражение, содержащее столбцы с одинаковыми именами, является некорректным. Такая ситуация возможна в том случае, когда в основе выражения лежит запрос, соединяющий две таблицы. Чтобы решить данную проблему, столбцам с одинаковыми именами можно назначить разные псевдонимы.

Эти требования вытекают из того факта, что табличное выражение должно возвращать реляционный результат. Всем атрибутам отношения следует дать имена; имена атрибутов должны быть уникальными; тело отношения является множеством кортежей, а множество не может быть упорядоченным.

**Обобщенные табличные выражения**

ОТВ очень похожи на производные таблицы, но имеют несколько важных преимуществ. Они определяются с помощью ключевого слова WITH и в целом выглядят следующим образом.

WITH <имя\_выражения>[(<список\_задействованных\_столбцов>)]

AS

(

<внутренний\_запро\_определяющий\_выражение>

)

<запрос\_к\_выражению>;

Внутренний запрос должен соответствовать всем требованиям, которые предъявляются к любому табличному выражению и которые мы уже перечислили ранее. Как и в случае с производными таблицами, результат ОТВ становится недоступным сразу после выполнения внешнего кода.

ОТВ имеют одну уникальную особенность: они поддерживают рекурсию. Рекурсивные ОТВ определяются несколькими запросами, которые называются закрепленными и рекурсивными элементами (должен присутствовать как минимум один элемент каждого типа). Общая форма рекурсивного ОТВ выглядит следующим образом.

WITH <имя\_ОТВ>[(<список\_столбцов>)]

AS

(

<закрепленный\_элемент>

UNION ALL

<рекурсивный\_элемент>

)

<внешний\_запрос\_к\_ОТВ>;

**Закрепленный элемент** — это запрос, который возвращает корректный реляционный результат и вызывается всего один раз; здесь нет никаких отличий от нерекурсивного табличного выражения.

**Рекурсивный элемент** — это запрос, который содержит ссылку на имя ОТВ (результирующий набор, полученный на предыдущей итерации). При первом вызове рекурсивного элемента предшествующий результат представлен значениями, которые вернул закрепленный элемент. При каждом следующем вызове имя ОТВ представляет результат предыдущего выполнения рекурсивного элемента. В этой замкнутой цепочке отсутствует явное условие преждевременного завершения. Выход из рекурсии происходит в тот момент, когда рекурсивный элемент возвращает пустой набор или достигает какого-то предела. Оба запроса должны быть совместимы по количеству и типу возвращаемых столбцов.

Те два вида табличных выражений, которые мы успели рассмотреть (производные таблицы и ОТВ), могут работать в рамках одного-единственного запроса. Они исчезают в момент, когда внешний код завершает работу. Это означает, что их нельзя использовать больше одного раза.

**Представления** и **встроенные ФТЗ** — это два вида табличных выражений, которые пригодны для многократного использования; их определения представляют собой объекты БД. После своего создания они помещаются на постоянное хранение в БД и могут быть удалены оттуда только отдельной командой. В остальном представления и встроенные ФТЗ ведут себя так же, как производные таблицы и ОТВ. Например, выполняя запрос к представлению или ФТЗ, SQL Server разворачивает определение табличного выражения, обращаясь к исходным объектам напрямую, как мы видели раньше.

**Представление** — это табличное выражение, пригодное для многократного использования и хранящееся в БД.

CREATE VIEW Sales.USACusts

AS

SELECT …

Вы можете управлять доступом к представлению (как и к любому другому объекту, к которому можно выполнить запрос) посредством полномочий; это касается прав на использование команд SELECT, INSERT, UPDATE и DELETE. Например, вы можете закрыть прямой доступ к исходным таблицам, разрешив обращаться к ним только через представления.

Инструкцию ORDER BY, которая формирует конечный результат, нельзя использовать в запросах, определяющих табличные выражения, потому что в реляционном наборе строки не должны быть упорядоченными. Если вы хотите упорядочить результат, полученный из представления, укажите инструкцию ORDER BY во внешнем запросе.

**Встроенные ФТЗ** — это табличные выражения, которые поддерживают входящие параметры. Фактически, если отбросить формальности, их можно считать параметризованными представлениями.

CREATE FUNCTION dbo.GetCustOrders

(@cid AS INT) RETURNS TABLE

AS

RETURN

SELECT orderid, custid, empid, orderdate,

requireddate, shippeddate, shipperid,

freight, shipname, shipaddress, shipcity,

shipregion, shippostalcode, shipcountry

FROM Sales.Orders

WHERE custid = @cid;

Обращение к данной функции выполняется так же, как и к обычным таблицам — с помощью элементов языка DML. Параметры указываются в скобках, которые следуют за названием функции.

SELECT orderid, custid

FROM dbo.GetCustOrders(1) AS O;

Табличный оператор APPLY отличается широкими возможностями. Как и другие табличные операторы, он используется внутри инструкции FROM. У него есть две разновидности: CROSS APPLY и OUTER APPLY. Первая реализует один логический этап обработки, а вторая — два.

Оператор APPLY принимает две таблицы (я будут называть их левой и правой), вторая может быть табличным выражением — обычно это производная таблица или встроенная ФТЗ. Оператор CROSS APPLY реализует один логический этап обработки; он применяет правое табличное выражение к каждой строке левой таблицы, возвращая объединенный результирующий набор.

## Глава 9. Транзакции и параллелизм

1. Является ли каждая отдельная команда транзакцией:

* да, каждая команда является транзакцией и подтверждается по умолчанию, если установлен параметр сессии IMPLICIT\_TRANSACTIONS,
* нет, каждая команда не равно транзакции,
* транзакция – это только то, что находится между инструкциями BEGIN TRAN / COMMIT TRAN.

В SQL Server каждая отдельная команда считается транзакцией и подтверждается по умолчанию. Вы можете изменить это поведение, используя параметр сессии под названием IMPLICIT\_TRANSACTIONS (по умолчанию SET IMPLICIT\_TRANSACTIONS OFF), который требует явного обозначения конца транзакции посредством инструкций COMMIT TRAN или ROLLBACK TRAN.

Транзакцию можно обозначить явно или неявно. Для явного открытия используется инструкция BEGIN TRAN (или BEGIN TRANSACTION). Чтобы подтвердить выполнение транзакции, нужно указать инструкцию COMMIT TRAN; если вы не хотите ее подтверждать (в этом случае все изменения будут отменены), используйте инструкцию ROLLBACK TRAN (или ROLLBACK TRANSACTION).

1. Какими свойствами характеризуется транзакция:

* ACID,
* IMAP,
* AIDA.

Транзакции характеризуются четырьмя свойствами — атомарностью, согласованностью, изоляцией и надежностью, которые принято обозначать аббревиатурой ACID (atomicity, consistency, isolation, durability).

1. Что такое атомарность транзакции:

* транзакция либо полностью выполняет все запланированные действия, либо не делает ничего,
* транзакцию нельзя разделить на отдельные команды,
* оба утверждения верны.

Атомарность. Транзакция – это атомарная единица работы. Она либо полностью выполняет все запланированные действия, либо не делает ничего. Если до ее завершения (то есть до того, как инструкция COMMIT TRAN будет записана в журнал) произошел какой-то системный сбой, после перезапуска SQL Server отменит все изменения. В случае возникновения обычных ошибок транзакция автоматически откатывается обратно. Бывают ситуации, когда автоматическая отмена результатов транзакции считается неоправданной, например, нарушение условий первичного ключа или истечение времени блокирования. Такие ошибки легко перехватывать с помощью кода обработки, принимая все необходимые меры (например, вы можете записать ошибку в журнал и откатить транзакцию).

В SQL Server есть функция @@TRANCOUNT. Если текущий участок кода выполняется внутри транзакции, она возвращает положительное число. В противном случае возвращается 0.

1. Что такое согласованность транзакции:

* такое состояние данных, которое должно быть при параллельном доступе на чтение и запись,
* выполнение правил, связанных с целостностью данных, таких как первичные и внешние ключи, ограничения уникальности,
* оба утверждения верны.

Согласованность. Таким термином обозначают состояние данных, к которым предоставляется параллельный доступ на чтение и запись. Как вы понимаете, это довольно субъективное понятие, суть которого зависит от нужд приложения. Под согласованностью понимают еще и тот факт, что БД должна соблюдать все аспекты целостности, определенные внутри нее с помощью различных правил, таких как первичные и внешние ключи, ограничения уникальности и т. д. В результате транзакции база данных меняет одно согласованное состояние на другое.

1. Что такое изоляция транзакции:

* механизм, который используется для управления доступом к данным,
* изоляция может использовать блокировки или управление версиями строк.
* оба утверждения верны.

Изоляция. Этот механизм используется для управления доступом к данным. Он следит за тем, чтобы транзакция получала доступ только к той информации, которая обладает необходимым уровнем согласованности. SQL Server поддерживает два вида изоляции: традиционный, основанный на блокировании, и новый, который подразумевает управление версиями строк. Первый используется по умолчанию в локальных версиях SQL Server и при выполнении чтения требует наличия разделяемых блокировок. Чтение блокируется до тех пор, пока состояние данных не станет согласованным. Второй вид изоляции, основанный на управлении версиями строк, используется по умолчанию в Windows Azure SQL Database. Он снимает необходимость в разделяемых блокировках и позволяет клиентам производить чтение без задержек. Если текущее состояние данных не согласовано, клиент получает более старую версию, которая годится к употреблению.

1. Что такое надежность транзакции:

* транзакция считается надежной, если в журнале транзакций записана инструкция COMMIT TRAN,
* транзакция считается надежной, если данные, изменяемые в результате транзакции, были записаны на диск,
* оба утверждения верны.

Надежность. Прежде чем сохранять данные непосредственно в БД, SQL Server всегда записывает изменения в журнал транзакций. Если в него попала инструкция COMMIT TRAN, транзакция считается надежной; при этом нет никакой гарантии, что данные действительно были записаны на диск. При запуске (не важно, в обычном режиме или после сбоя) система сверяется с журналом транзакций каждой БД и запускает процесс восстановления, который состоит из двух этапов — повторного выполнения и отмены. Первый этап подразумевает повторение всех действий, принадлежащих транзакциям, которые были зафиксированы в журнале, но не закончили процесс записи. На втором этапе выполняется откат (отмена) всех изменений, произведенных в рамках транзакций, которые не были подтверждены в журнале с помощью инструкции COMMIT TRAN.

1. Есть таблица Orders, в которой хранится список всех заказов с полями OrderId (номер заказа), CustomerId (номер покупателя), OrderDate (дата создания заказа), и таблица OrderDetails, в которой хранится список товаров во всех заказах: OrderId (номер заказа), ProductId (номер товара), Price (цена товара), Quantity (количество проданного товара). В следующем коде записана транзакция, в результате работы которой создается новый заказ:

BEGIN TRAN

DECLARE @newOrderId AS INT;

INSERT INTO Orders (CustomerId, OrderDate)

VALUES (85, '20210218');

INSERT INTO OrderDetails

(OrderId, ProductId, Price, Quantity)

VALUES(@newOrderId, 11, 14.00, 7),

(@newOrderId, 42, 9.80, 10),

(@newOrderId, 72, 34.80, 5);

COMMIT TRAN;

Какую строчку и куда надо вставить, чтобы транзакция прошла успешно?

* SET @newOrderId = SCOPE\_IDENTITY();
* SET @newOrderId = @@IDENTITY();
* SELECT @newOrderId = OrderId

FROM Orders

WHERE CustomerId = 85 AND OrderDate = '20210218';

* можно использовать любой из вышеописанных вариантов.

1. Какие модели одновременного доступа поддерживает компонент Database Engine:

* пессимистический и оптимистический,
* параллельный и последовательный,
* изолированный и неизолированный.

Когда пользователи обращаются к ресурсу одновременно, говорят, что они делают это параллельно. Параллельный доступ к данным требует наличия механизмов предотвращения нежелательных последствий, которые могут возникнуть при попытке пользователей изменить ресурсы, активно используемые другими.

Если несколько пользователей одновременно пытаются выполнять изменения в базе данных, следует реализовать систему элементов управления, с тем чтобы изменения, проводимые одним пользователем, не затрагивали работу другого пользователя. Такая система называется управлением параллелизмом.

Существует два способа управления параллелизмом:

1. **пессимистическое** управление параллелизмом использует блокировки.

В модели пессимистического одновременного конкурентного доступа для предотвращения одновременного доступа к данным, которые используются другим процессом, применяются блокировки. Иными словами, система баз данных, использующая модель пессимистического одновременного конкурентного доступа, предполагает, что между двумя или большим количеством процессов в любое время может возникнуть конфликт и поэтому блокирует ресурсы (строку, страницу, таблицу), как только они потребуются в течение периода транзакции.

1. **оптимистическое** управление параллелизмом использует версии строк.

Работа оптимистического одновременного конкурентного доступа основана на предположении маловероятности изменения данных одной транзакцией одновременно с другой. Компонент Database Engine применяет оптимистический одновременный конкурентный доступ, при котором сохраняются старые версии строк, и любой процесс при чтении данных использует ту версию строки, которая была активной, когда он начал чтение. Поэтому процесс может модифицировать данные без каких-либо ограничений, поскольку все другие процессы, которые считывают эти же данные, используют свою собственную сохраненную версию. Конфликтная ситуация возможна только при попытке двух операций записи использовать одни и те же данные. В таком случае система выдает ошибку, которая обрабатывается клиентским приложением.

Понятие оптимистического одновременного конкурентного доступа обычно определяется в более широком смысле. Работа управления оптимистического одновременного конкурентного доступа основана на предположении маловероятности конфликтов между несколькими пользователями, поэтому разрешается исполнение транзакций без установки блокировок. Только когда пользователь пытается изменить данные, выполняется проверка ресурсов, чтобы определить наличие конфликтов. Если таковые возникли, то приложение требуется перезапустить.

1. Что такое эксклюзивная блокировка (exclusive lock):

* блокировка на изменяемые данные, чтобы никакой другой процесс не мог их считать или модифицировать,
* блокировка на изменяемые данные, чтобы никакой другой процесс не мог их модифицировать,
* блокировка на считываемые данные, чтобы никакой другой процесс не мог их считать или модифицировать.

1. Что такое разделяемая блокировка (shared lock):

* блокировка на считываемые данные, чтобы никакой другой процесс не мог их модифицировать,
* блокировка на изменяемые данные, чтобы никакой другой процесс не мог их модифицировать,
* блокировка на считываемые данные, чтобы никакой другой процесс не мог их считать или модифицировать.

1. Если строка имеет блокировку exclusive lock, то можно ли на эту строку установить еще и shared lock:

* нельзя, на ресурс с установленной exclusive lock нельзя установить никакую другую блокировку,
* можно, но бессмысленно, так как exclusive lock включает в себя все функции shared lock,
* можно, чтобы после снятия одной транзакцией exclusive lock на строке осталась установлена shared lock другой транзакции.

1. Если строка имеет блокировку shared lock, то можно ли на эту строку установить еще и exclusive lock:

* нельзя, так как exclusive lock можно установить только на «свободную» строку,
* можно, при этом на всю строку будет установлена exclusive lock, так как она включает в себя все функции shared lock,
* можно, чтобы после снятия одной транзакцией shared lock на строке осталась установлена exclusive lock другой транзакции.

1. Если строка имеет блокировку shared lock, то можно ли на эту строку установить еще одну shared lock:

* можно, именно для этого и существует shared lock: чтобы получить одновременный доступ к одной строке (ресурсу) от нескольких транзакций,
* нельзя, так как shared lock можно установить только на «свободную» строку,
* можно, при этом две блокировки shared lock автоматически поменяются на exclusive lock.

1. Строка имеет две блокировки shared lock от двух транзакций. Что будет, когда одна из транзакций завершится:

* на строке останется одна shared lock от той транзакции, которая еще не завершена,
* на строке не останется никаких блокировок.

Блокировки используются только в пессимистической модели параллельного доступа. Существует два режима блокировок — exclusive lock и shared lock.

Модель пессимистического одновременного конкурентного доступа устанавливает блокировку с обеспечением разделяемого доступа, иначе немонопольную блокировку (*shared lock*) на считываемые данные, чтобы никакой другой процесс *не мог изменить* эти данные. Кроме этого, механизм пессимистического одновременного конкурентного доступа устанавливает монопольную блокировку (*exclusive lock*) на изменяемые данные, чтобы никакой другой процесс *не мог их считывать или модифицировать*.

*shared lock* резервирует ресурс (страницу или строку) только для чтения. Другие процессы не могут изменять заблокированный таким образом ресурс, но, с другой стороны, несколько процессов могут одновременно накладывать разделяемую блокировку на один и тот же ресурс. Иными словами, чтение ресурса с разделяемой блокировкой могут одновременно выполнять несколько процессов.

*exclusive lock* резервирует страницу или строку для монопольного использования одной транзакцией. Блокировка этого типа применяется инструкциями DML (INSERT, UPDATE и DELETE), которые модифицируют ресурс. Монопольную блокировку нельзя установить, если на ресурс уже установлена разделяемая или монопольная блокировка другим процессом, т.е. на ресурс может быть установлена только одна монопольная блокировка. На ресурс (страницу или строку) с установленной монопольной блокировкой нельзя установить никакую другую блокировку.

1. Для каких ресурсов существует разделяемая блокировка с намерениями (intent shared lock):

* для таблицы,
* для страницы или строки,
* для строки.

На уровне таблицы существует пять разных типов блокировок:

1. разделяемая (shared, S);
2. монопольная (exclusive, X);
3. разделяемая с намерением (intent shared, IS);
4. монопольная с намерением (intent exclusive, IX);
5. разделяемая с монопольным намерением (shared with intent exclusive, SIX).

Разделяемые и монопольные типы блокировок для таблицы соответствуют одноименным блокировкам для строк и страниц. Обычно блокировка с намерением (intent lock) означает, что транзакция намеревается блокировать следующий нижележащий в иерархии объектов базы данных ресурс. Таким образом, блокировка с намерением помещается на уровне иерархии объектов, который выше того объекта, который этот процесс намеревается заблокировать. Это является действенным способом узнать, возможна ли подобная блокировка, а также устанавливается запрет другим процессам блокировать более высокий уровень, прежде чем процесс может установить требуемую ему блокировку.

Компонент Database Engine может блокировать следующие ресурсы: строки, страницы, индексный ключ или диапазон индексных ключей, таблицы, экстент, саму базу данных. Строка является наименьшим ресурсом, который можно заблокировать. Блокировка уровня строки также включает как строки данных, так и элементы индексов. Блокировка на уровне строки означает, что блокируется только строка, к которой обращается приложение. Поэтому все другие строки данной таблицы остаются свободными и их могут использовать другие приложения. Компонент Database Engine также может заблокировать страницу, на которой находится подлежащая блокировке строка.

Чтобы заблокировать ресурс определенного типа, ваша транзакция должна получить намеренную блокировку того же режима, но на более высоком уровне детализации. Например, чтобы заблокировать строку в монопольном режиме, транзакция сначала должна получить намеренную монопольную блокировку страницы, в которой находится эта строка, и объекта, который хранит страницу. Точно такой же принцип действует для разделяемых блокировок. Намеренное блокирование позволяет определять и отклонять запросы на получение несовместимых блокировок на более высоком уровне детализации. Например, если одна транзакция блокирует строку, а другая запрашивает несовместимую блокировку для страницы или целой таблицы, в которой находится эта строка, SQL Server легко обнаруживает конфликт, поскольку он знает о намеренной блокировке первой транзакции, установленной для страницы и таблицы. Намеренные блокировки не конфликтуют с запросами на более низком уровне детализации. Например, намеренное блокирование страницы не мешает другим транзакциям получать блокировки в совместимом режиме для строк, которые там находятся.

1. Что такое укрупнение блокировок (lock escalation):

* преобразование большого числа блокировок уровня строки в одну блокировку уровня таблицы,
* изменение блокировки с shared lock на exclusive lock.

Если в процессе транзакции имеется большое количество блокировок одного уровня, то компонент Database Engine автоматически объединяет эти блокировки в одну уровня таблицы. Этот процесс преобразования большого числа блокировок уровня строки, страницы или индекса в одну блокировку уровня таблицы называется укрупнением блокировок (lock escalation). Порогом укрупнения называется граница, на которой система баз данных применяет укрупнение блокировок. Пороги укрупнения устанавливаются динамически системой и не требуют настройки. (В настоящее время пороговым значением укрупнения блокировок является 5000 блокировок.)

1. Что такое взаимоблокировка (deadlock):

* ситуация, в которой две транзакции блокируют друг друга,
* одновременная установка двух блокировок shared lock,
* одновременная установка блокировок shared lock и exclusive lock.

Взаимоблокировка (deadlock) - это особая проблема одновременного конкурентного доступа, в которой две транзакции блокируют друг друга. В частности, первая транзакция блокирует объект базы данных, доступ к которому хочет получить другая транзакция, и наоборот. (В общем, взаимоблокировка может быть вызвана несколькими транзакциями, которые создают цикл зависимостей.)

Система баз данных обрабатывает взаимоблокировку, выбирая одну из транзакций (на самом деле, транзакцию, которая замыкает цикл в запросах блокировки) в качестве "жертвы" и выполняя ее откат. После этого выполняется другая транзакция. На уровне прикладной программы взаимоблокировку можно обрабатывать посредством реализации условной инструкции, которая выполняет проверку на возврат номера ошибки (1205), а затем снова выполняет инструкцию, для которой был выполнен откат.

Вы можете повлиять на то, какая транзакция будет выбрана системой в качестве "жертвы" взаимоблокировки, присвоив в инструкции SET параметру DEADLOCK\_PRIORITY один из 21 (от -10 до 10) разных уровней приоритета взаимоблокировки. Константа LOW соответствует значению -5, NORMAL (значение по умолчанию) - значению 0, а константа HIGH - значению 5. Сеанс "жертва" выбирается в соответствии с приоритетом взаимоблокировки сеанса.

1. Что такое уровни изоляции:

* возможность влиять на процесс блокирования при выполнении чтения,
* способ задать степень защищенности выбираемых транзакцией данных от возможности изменения другими транзакциями,
* оба утверждения верны.

Теоретически, все транзакции должны быть изолированы друг от друга. Но в таком случае доступность данных значительно бы понизилась, поскольку операции чтения транзакции блокировали бы операции записи в других транзакциях, и наоборот. Если доступность данных является важным требованием, то это свойство можно ослабить, используя уровни изоляции.

Уровень изоляции задает степень защищенности выбираемых транзакцией данных от возможности изменения другими транзакциями.

У вас нет возможности поменять режим блокировки и ее продолжительность, но у вас есть возможность влиять на процесс блокирования при выполнении чтения — именно для этого были созданы уровни изоляции.

Если блокировка не используется и, следовательно, транзакции не изолированы друг от друга, то могут возникнуть следующие проблемы: потеря обновлений, "грязное чтение", неповторяемое чтение и фантомы.

Проблема потери обновлений при одновременном конкурентном доступе к данным возникает, когда транзакция не изолирована от других транзакций. Это означает, что несколько транзакций одновременно могут считывать и обновлять одни и те же данные. При этом теряются все обновления данных, за исключением обновлений, выполненных последней транзакцией.

Проблема неповторяемого чтения при одновременном конкурентном доступе к данным возникает, когда один процесс считывает данные несколько раз, а другой процесс изменяет эти данные между двумя операциями чтения первого процесса. В таком случае значения двух чтений будут разными.

Проблема фантомов при параллельном одновременном конкурентном доступе к данным подобна проблеме неповторяемого чтения, поскольку две последовательные операции чтения могут возвратить разные значения. Но в данном случае причиной этому является считывание разного числа строк при каждом чтении. Дополнительные строки называются фантомами и вставляются другими транзакциями.

1. Где можно посмотреть установленный уровень изоляции для текущей сессии SQL Server:

* обратиться к SQL Server Profiler,
* просмотреть информацию в Execution plan,
* SQL Server всегда имеет уровень изоляции READ COMMITTED.

В большинстве приложений уровень изолированности редко меняется и используется значение по умолчанию (например, в SQL Server это READ COMMITTED).

Но периодически возникают, задачи, в которых поиск лучшего баланса между высокой согласованностью данных или скоростью выполнения транзакций может помочь решить некоторую прикладную задачу.

1. Как можно установить уровень изоляции:

* уровень изоляции можно выставить посредством параметра сессии,
* уровень изоляции можно выставить в самом запросе в виде табличного указания,
* оба утверждения верны.

Уровни изоляции выставляются либо посредством параметра сессии (параметр TRANSACTION ISOLATION LEVEL инструкции SET), либо в самом запросе в виде табличного указания. Задание уровня изоляции в предложении FROM перекрывает текущее его значение, установленное инструкцией SET TRANSACTION ISOLATION LEVEL.

1. Расположите четыре уровня изоляции (для пессимистического управлении параллелизмом) в порядке убывания согласованности:

* SERIALIZABLE,
* REPEATABLE READ,
* READ COMMITTED,
* READ UNCOMMITTED.

SQL Server поддерживает четыре традиционных уровня изоляции, основанных на пессимистическом управлении параллелизмом (блокировании):

* READ UNCOMMITTED,
* READ COMMITTED (используется по умолчанию в локальных версиях SQL Server),
* REPEATABLE READ,
* SERIALIZABLE.

Поддерживаются также два дополнительных уровня, в основе которых лежит оптимистичный подход к параллелизму (управление версиями строк):

* SNAPSHOT,
* READ COMMITTED SNAPSHOT (используется по умолчанию в SQL Database);

Они являются «оптимистичными» аналогами уровней READ COMMITTED и SERIALIZABLE соответственно.

В некоторых статьях READ COMMITTED и READ COMMITTED SNAPSHOT рассматриваются как один и тот же уровень, только с разной семантикой.

Если взять первые четыре уровня, то чем они выше, тем более строгие блокировки запрашиваются при чтении и тем длиннее время их ожидания; вместе с уровнем повышается согласованность, но ухудшаются условия параллельной работы. Обратное утверждение тоже верно.

С двумя другими уровнями изоляции все обстоит не так. SQL Server способен хранить ранее подтвержденные версии строк в БД tempdb. При чтении пользователи не запрашивают разделяемые блокировки; вместо этого они пытаются получить более старую версию записи, если текущая им не подходит.

1. Что такое READ UNCOMMITTED:

* наиболее низкий уровень изоляции, который не требует получения разделяемой блокировки для чтения данных,
* наиболее низкий уровень изоляции, который тем не менее требует получения разделяемой (shared) блокировки для чтения данных.

READ UNCOMMITTED — наиболее низкий уровень изоляции, который не требует получения разделяемой блокировки для чтения данных. Тем самым исключается конфликт с пользователями, которые производят запись в эксклюзивном режиме. Это означает, что вы можете считывать неподтвержденные данные (данный процесс называется грязным чтением). Пока операции чтения происходит на уровне изоляции READ UNCOMMITTED, пользователи могут свободно записывать данные.

Уровень изоляции READ UNCOMMITTED предоставляет самую простую форму изоляции между транзакциями, поскольку он вообще не изолирует операции чтения других транзакций. Когда транзакция выбирает строку при этом уровне изоляции, она не задает никаких блокировок и не признает никаких существующих блокировок. Считываемые такой транзакцией данные могут быть несогласованными. В таком случае транзакция читает данные, которые были обновлены какой-либо другой активной транзакцией. А если для этой другой транзакции позже выполняется откат, то значит, что первая транзакция прочитала данные, которые никогда по-настоящему не существовали.

Применение уровня изоляции READ UNCOMMITTED обычно крайне нежелательно и его следует применять только в тех случаях, когда точность данных не представляет важности или когда данные редко подвергаются изменениям.

NOLOCK – это синоним для READUNCOMMITTED.

1. Следующий код увеличивает цену товара с кодом 1 на 2.00 (первоначальное значение цены для этого товара 13.00):

BEGIN TRAN

UPDATE Products

SET Price += 2.00

WHERE ProductId = 1;

Затем в той же сессии (назовем ее session1) был сделан следующий запрос

SELECT Price

FROM Products

WHERE ProductId = 1;

А в другой сессии (назовем ее session2) был сделан вот такой запрос

SELECT Price

FROM Products WITH (READUNCOMMITTED)

WHERE ProductId = 1;

Какие результаты будут выведены в результате работы этих запросов?

* в session1 Price = 15.00, в session2 Price = 15.00;
* в session1 Price = 15.00, в session2 Price = 13.00;
* в session1 Price = 15.00, в session2 запрос не будет выполнен, так как транзакция не была завершена и при обновлении данных она установила эксклюзивную блокировку.

Поскольку запрос в session2 не пытается получить разделяемую блокировку, он не конфликтует с другой транзакцией. Мы получим строку в том виде, который она приняла после обновления, несмотря на то что изменения не были подтверждены.

1. Что такое READ COMMITTED:

* уровень изоляции, который допускает чтение только тех изменений, которые были подтверждены,
* уровень изоляции, который требует получить разделяемую (shared) блокировку,
* оба утверждения верны.

Если вы хотите запретить чтение неподтвержденных изменений, вам нужен более строгий уровень изоляции. Самым низким уровнем, который позволяет это сделать, является READ COMMITTED; он используется по умолчанию в локальных версиях SQL Server. Как видно из названия, READ COMMITTED допускает чтение только тех изменений, которые были подтверждены. Для всего остального требуется получить разделяемую блокировку, которая конфликтует с эксклюзивной. Это означает, что если данные уже изменяются, пользователь, которому нужно их прочитать, должен будет подождать. Он получит разделяемую блокировку сразу, как только транзакция подтвердит изменения.

В отличие от READ UNCOMMITTED, уровень изоляции READ COMMITTED исключает грязное чтение.

Транзакция, которая читает строку и использует уровень изоляции READ COMMITTED, выполнят проверку только на наличие монопольной блокировки для данной строки. Если такая блокировка отсутствует, транзакция извлекает строку. (Это выполняется с использованием разделяемой блокировки.) Таким образом предотвращается чтение транзакцией данных, которые не были подтверждены и которые могут быть позже отменены. После того, как данные были прочитаны, их можно изменять другими транзакциями.

Применяемые этим уровнем изоляции разделяемые блокировки отменяются сразу же после обработки данных. (Обычно все блокировки отменяются в конце транзакции.) Это улучшает параллельный одновременный конкурентный доступ к данным, но возможность неповторяемого чтения и фантомов продолжает существовать.

Уровень изоляции READ COMMITTED для компонента Database Engine является уровнем изоляции по умолчанию.

С практической точки зрения уровень изоляции READ COMMITTED не позволяет считывать данные, которые изменяются в рамках другой транзакции. Такой способ управления параллелизмом называют пессимистическим. Но есть и оптимистический подход, который реализован за счет уровня изоляции READ COMMITTED SNAPSHOT: если транзакция попытается прочитать строки, которые в это самое время подлежат изменению, она получит их последнее подтвержденное состояние, актуальное на момент запуска самой транзакции.

1. Следующий код увеличивает цену товара с кодом 1 на 2.00 (первоначальное значение цены для этого товара 13.00):

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

BEGIN TRAN

UPDATE Products

SET Price += 2.00

WHERE ProductId = 1;

Затем в той же сессии (назовем ее session1) был сделан следующий запрос

SELECT Price

FROM Products

WHERE ProductId = 1;

А в другой сессии (назовем ее session2) был сделан вот такой запрос

SELECT Price

FROM Products WITH (READCOMMITTEDLOCK)

WHERE ProductId = 1;

Какие результаты будут выведены в результате работы этих запросов?

* в session1 Price = 15.00, в session2 запрос не будет выполнен, так как транзакция не была завершена и при обновлении данных она установила эксклюзивную блокировку;
* в session1 Price = 15.00, в session2 Price = 15.00;
* в session1 Price = 15.00, в session2 Price = 13.00.

Чтобы прочитать эти данные, вашей сессии нужна разделяемая блокировка. Поскольку строка уже эксклюзивно блокируется другой сессией, а эксклюзивные и разделяемые блокировки несовместимы между собой, вашей транзакции придется остановиться и подождать.

1. В сессии (назовем ее session1) следующий код читает цену товара с кодом 1 (она равна 13.00):

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

BEGIN TRAN

SELECT Price

FROM Products

WHERE ProductId = 1;

Затем в другой сессии (назовем ее session2) был сделан вот такой запрос, который увеличивает цену товара с кодом 1 на 2.00 (первоначальное значение цены для этого товара 13.00):

UPDATE Products

SET Price += 2.00

WHERE ProductId = 1;

SELECT Price

FROM Products WITH (READCOMMITTEDLOCK)

WHERE ProductId = 1;

Какие результаты будут выведены в результате работы этих запросов?

* в session1 Price = 13.00, в session2 Price = 15.00;
* в session1 Price = 13.00, в session2 запрос не будет выполнен, так как транзакция в session1 не была завершена;
* в session1 Price = 13.00, в session2 Price = 13.00.

В режиме изоляции READ COMMITTED блокировка на чтение снимается сразу же после выполнения команды, не дожидаясь конца транзакции. Поэтому запросы в session2 будут выполнены.

1. Что такое REPEATABLE READ:

* уровень изоляции, который требует получить разделяемую (shared) блокировку и удерживает ее до конца транзакции,
* уровень изоляции, который требует получить разделяемую (shared) блокировку и отменяет ее сразу после обработки данных,
* уровень изоляции, который требует получить эксклюзивную (exclusive) блокировку.

В отличие от уровня изоляции READ COMMITTED, уровень REPEATABLE READ устанавливает разделяемые блокировки на все считываемые данные и удерживает эти блокировки до тех пор, пока транзакция не будет подтверждена или отменена. Поэтому в этом случае многократное выполнение запроса внутри транзакции всегда будет возвращать один и тот же результат. Недостатком этого уровня изоляции является дальнейшее ухудшение одновременного конкурентного доступа, поскольку период времени, в течение которого другие транзакции не могут обновлять те же самые данные, значительно дольше, чем в случае уровня READ COMMITTED.

Этот уровень изоляции не препятствует другим инструкциям вставлять новые строки, которые включаются в последующие операции чтения, вследствие чего могут появляться фантомы.

HOLDLOCK – это синоним для REPEATABLEREAD.

1. В сессии (назовем ее session1) следующий код читает цену товара с кодом 1 (она равна 13.00):

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN TRAN

SELECT Price

FROM Products

WHERE ProductId = 1;

Затем в другой сессии (назовем ее session2) был сделан вот такой запрос, который увеличивает цену товара с кодом 1 на 2.00 (первоначальное значение цены для этого товара 13.00):

UPDATE Products

SET Price += 2.00

WHERE ProductId = 1;

SELECT Price

FROM Products

WHERE ProductId = 1;

Какие результаты будут выведены в результате работы этих запросов?

* в session1 Price = 13.00, в session2 запрос не будет выполнен, так как транзакция в session1 не была завершена;
* в session1 Price = 13.00, в session2 Price = 15.00;
* в session1 Price = 13.00, в session2 Price = 13.00.

В режиме изоляции REPEATABLE READ  блокировка на чтение снимается только после окончания транзакции. Поэтому запросы в session2 не будут выполнены.

1. Что такое SERIALIZABLE:

* уровень изоляции, который требует получить разделяемую (shared) блокировку на всю область данных, считываемых транзакцией,
* уровень изоляции, который требует получить разделяемую (shared) блокировку и удерживает ее до конца транзакции,
* оба утверждения верны.

Уровень изоляции SERIALIZABLE является самым строгим, потому что он не допускает возникновения всех четырех проблем параллельного одновременного конкурентного доступа, перечисленных ранее. Этот уровень устанавливает блокировку на всю область данных, считываемых соответствующей транзакцией. Поэтому этот уровень изоляции также предотвращает вставку новых строк другой транзакцией до тех пор, пока первая транзакция не будет подтверждена или отменена.

Уровень изоляции SERIALIZABLE реализуется, используя метод блокировки диапазона ключа. Суть этого метода заключается в блокировке отдельных строк включительно со всем диапазоном строк между ними. Блокировка диапазона ключа блокирует элементы индексов, а не определенные страницы или всю таблицу. В этом случае любые операции модификации другой транзакцией невозможны, вследствие невозможности выполнения требуемых изменений элементов индекса.

В заключение обсуждения четырех уровней изоляции следует упомянуть, что требуется знать, что чем выше уровень изоляции, тем меньше степень одновременного конкурентного доступа. Таким образом, уровень изоляции READ UNCOMMITTED меньше всего уменьшает одновременный конкурентный доступ. С другой стороны, он также предоставляет наименьшую изоляцию параллельных конкурентных транзакций. Уровень изоляции SERIALIZABLE наиболее сильно уменьшает степень одновременного конкурентного доступа, но гарантирует полную изоляцию параллельных конкурентных транзакций.

1. В таблице Products есть товары, у которых Price = 10. В рамках одной сессии (назовем ее session1) была сделана следующая выборка:

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRAN

SELECT ProductId, Price

FROM Products

WHERE Price = 10;

Затем в другой сессии (назовем ее session2) в таблицу Products был добавлен новый товар:

INSERT INTO Products (ProductId, ProductName, Price)

VALUES(100, N'Product A', 10.00);

Будет ли добавлен новый товар?

* нет,
* да.

При уровне изоляции SERIALIZABLE блокировка на чтение удерживается до конца транзакции, а также не позволяет вставку данных, которые входят в фильтр читаемых данных.

1. Какие из следующих утверждений о механизме управления оптимистической параллельной работы верны:

* все копии строк сохраняются в системной базе данных tempdb,
* нет необходимости запрашивать разделяемые блокировки (shared lock) прочитанных строк,
* эксклюзивный блокировки (exclusive lock) требуются при модификации данных,
* все утверждения верны.

Компонент Database Engine поддерживает механизм управления оптимистической параллельной работы, который основан на управлении версиями строк. При модифицировании данных с использованием управления версиями строк, для всех выполняемых в базе данных модификаций данных создаются и поддерживаются логические копии данных. При каждой модификации строки система баз данных сохраняет в системной базе данных tempdb исходный вид записи ранее зафиксированной строки. Каждая версия строки помечается порядковым номером транзакции (XSN - transaction sequence number), которая выполнила это изменение. (Порядковый номер транзакции XSN применяется для уникальной идентификации транзакций.)

Самая последняя версия строки всегда сохраняется в базе данных и соединяется в связанном списке с соответствующей версией, сохраненной в базе данных tempdb. Старая версия строки в базе данных tempdb может содержать указатели на другие, еще более старые версии. Каждая версия строки сохраняется в базе данных tempdb до тех пор, пока существуют операции, для которых она может потребоваться.

Управление версиями строк изолируют транзакции от эффектов модификаций, выполненных другими транзакциями, без необходимости запрашивать разделяемые блокировки прочитанных строк. Это значительное уменьшение общего количества блокировок, устанавливаемых на этом уровне изоляции, существенно повышает степень доступности данных. Но использование монопольных блокировок все равно требуется: транзакции, использующие оптимистический уровень изоляции SNAPSHOT, запрашивают блокировки, когда они модифицируют строки.

1. Что такое READ COMMITTED SNAPSHOT:

* является облегченным вариантом уровня изоляции READ COMMITTED,
* основан на управлении версиями строк,
* оба утверждения верны.

В SQL Database стандартным уровнем изоляции является READ COMMITTED SNAPSHOT. Вместо того чтобы полагаться на блокировки, он основывается на управлении версиями строк. Это означает, что для чтения данных не нужно выполнять разделяемое блокирование и, как следствие, не приходится ждать; расширенная изоляция обеспечивается наличием нескольких вариантов каждой отдельной строки.

Уровень READ COMMITED SNAPSHOT является облегченным вариантом уровня изоляции READ COMMITTED. Это изоляция на уровне инструкции, что означает, что любая другая транзакция будет читать зафиксированные значения в том виде, в каком они существуют на момент начала этой инструкции. Для выборки строк для обновлений этот уровень изоляции возвращает версии строк в фактические данные и устанавливает на выбранных строках блокировки обновлений. Реальные строки данных, которые требуется изменить, получают монопольные блокировки.

Основным преимуществом уровня изоляции READ COMMITED SNAPSHOT является то, что операции чтения не блокируют обновлений, а обновления не блокируют операций чтения. Но с другой стороны, обновления блокируют другие обновления, поскольку для выполнения операций обновления устанавливаются монопольные блокировки.

Уровень изоляции READ COMMITED SNAPSHOT разрешается посредством предложения SET в инструкции ALTER DATABASE. После активирования этого уровня изоляции никаких дополнительных изменений выполнять не требуется. Любая инструкция, для которой указан уровень изоляции READ COMMITTED, теперь будет выполняться на уровне READ COMMITED SNAPSHOT.

1. В сессии (назовем ее session1) следующий код увеличивает цену товара с кодом 1 на 2.00 (первоначальное значение цены для этого товара 13.00):

ALTER DATABASE MyDB SET READ\_COMMITTED\_SNAPSHOT ON;

BEGIN TRAN

UPDATE Products

SET Price += 2.00

WHERE ProductId = 1;

SELECT Price

FROM Products

WHERE ProductId = 1;

Затем в другой сессии (назовем ее session2) был сделан вот такой запрос на считывание цены для товара с кодом 1:

BEGIN TRAN

SELECT Price

FROM Products

WHERE ProductId = 1;

После этого транзакция в session1 была подтверждена

COMMIT TRAN;

А в session2 снова был выполнен запрос на чтение данных

SELECT Price

FROM Products

WHERE ProductId = 1;

COMMIT TRAN;

Что вернет последний запрос в session2?

* Price = 15.00;
* Price = 13.00.

Если бы код работал в режиме SNAPSHOT, вы бы получили цену 13.00; но поскольку определен уровень изоляции READ COMMITTED SNAPSHOT, вернулась последняя подтвержденная версия строки, доступная на момент запуска команды, а не транзакции.

1. Что такое SNAPSHOT:

* гарантирует, что клиент, считывающий данные, получит последнюю версию строки, подтвержденную на момент начала транзакции,
* вместо разделяемых блокировок используется разбиение строк на версии,
* оба утверждения верны.

Уровень изоляция SNAPSHOT предоставляет изоляцию на уровне транзакций, что означает, что любая другая транзакция будет читать подтвержденные значения в том виде, в каком они существовали непосредственно перед началом выполнения транзакции этого уровня изоляции. Кроме этого, транзакция уровня изоляции SNAPSHOT будет возвращать исходное значение данных до завершения своего выполнения, даже если в течение этого времени оно будет изменено другой транзакцией. Поэтому другая транзакция сможет читать модифицированное значение только после завершения выполнения транзакции уровня изоляции SNAPSHOT.

Транзакции уровня изоляции SNAPSHOT получают монопольные блокировки на данные перед тем, как выполнять изменения только с целью принудительного обеспечения ограничений. В противных случаях данные не блокируются до тех пор, пока данные не требуется изменить. Когда строка данных удовлетворяет критериям обновления, транзакция уровня изоляции SNAPSHOT проверяет, не была ли эта строка данных изменена и подтверждены изменения данных этой строки в конкурентной транзакции после того, как была запущена текущая транзакция. Если строка данных была изменена параллельной конкурентной транзакцией, то возникает конфликт обновления и дальнейшее выполнение транзакции уровня изоляции SNAPSHOT завершается. Этот конфликт обновления обрабатывается системой баз данных, поэтому способа отключить функцию обнаружения конфликтов обновления не существует.

Разрешение уровня изоляции SNAPSHOT осуществляется в два шага. Сначала на уровне базы данных включается опция базы данных ALLOW\_SNAPSHOT\_ISOLATION (это можно сделать, например, посредством среды Management Studio). После этого для каждого сеанса, который будет использовать этот уровень изоляции, нужно для инструкции SET TRANSACTION ISOLATION LEVEL задать значение SNAPSHOT. Когда эти опции установлены, будут создаваться версии для всех строк, изменяемых в базе данных.

Самое важное различие между этими двумя оптимистическими уровнями изоляции состоит в том, что при уровне SNAPSHOT возможны конфликты обновлений, когда процесс работает с одними и теми же данными во время выполнения транзакции и не является заблокированным. В противоположность этому, уровень изоляции READ COMMITTED SNAPSHOT не использует свой собственный порядковый номер транзакции XSN при выборе версий строк. Каждый раз при запуске инструкции такая транзакция считывает самый последний порядковый номер транзакции XSN, выданный для данного экземпляра системы баз данных, и выбирает строку с этим номером.

Другое отличие состоит в том, что уровень изоляции READ COMMITED SNAPSHOT позволяет другим транзакциям изменять данные до того, как будет завершена транзакция типа управления версиями строк. Это может вызвать конфликт, если в период времени между выполнением операции чтения транзакцией типа управления версиями строк и последующей попыткой этой транзакции выполнить соответствующую операцию записи данные были изменены другой транзакцией. (Для приложений на основе уровня изоляции SNAPSHOT система выявляет возможность конфликта и выдает соответствующее сообщение об ошибке.)

1. В сессии (назовем ее session1) следующий код увеличивает цену товара с кодом 1 на 2.00 (первоначальное значение цены для этого товара 13.00):

ALTER DATABASE MyDB SET ALLOW\_SNAPSHOT\_ISOLATION ON;

BEGIN TRAN

UPDATE Products

SET Price += 2.00

WHERE ProductId = 1;

SELECT Price

FROM Products

WHERE ProductId = 1;

Затем в другой сессии (назовем ее session2) был сделан вот такой запрос на считывание цены для товара с кодом 1:

SET TRANSACTION ISOLATION LEVEL SNAPSHOT;

BEGIN TRAN

SELECT Price

FROM Products

WHERE ProductId = 1;

Что будет выведено в результате работы этих запросов?

* в session1 Price = 15.00, в session2 Price = 13.00;
* в session1 Price = 15.00, в session2 Price = 15.00;
* в session1 Price = 15.00, в session2 запрос не будет выполнен, так как транзакция в session1 не была завершена.

Если бы ваша транзакция работала на уровне изоляции SERIALIZABLE, запрос был бы заблокирован. У нас установлен режим SNAPSHOT, поэтому вы получили последнюю версию строки, которая была доступна на момент запуска транзакции. Эта версия (с ценой 13.00) не является текущей (поскольку цена уже успела обновиться до 15.00), поэтому SQL Server читает ее из БД tempdb.

1. В сессии (назовем ее session1) следующий код увеличивает цену товара с кодом 1 на 2.00 (первоначальное значение цены для этого товара 13.00):

ALTER DATABASE MyDB SET ALLOW\_SNAPSHOT\_ISOLATION ON;

BEGIN TRAN

UPDATE Products

SET Price += 2.00

WHERE ProductId = 1;

SELECT Price

FROM Products

WHERE ProductId = 1;

Затем в другой сессии (назовем ее session2) был сделан вот такой запрос на считывание цены для товара с кодом 1:

SET TRANSACTION ISOLATION LEVEL SNAPSHOT;

BEGIN TRAN

SELECT Price

FROM Products

WHERE ProductId = 1;

После этого транзакция в session1 была подтверждена

COMMIT TRAN;

А в session2 снова был выполнен запрос на чтение данных

SELECT Price

FROM Products

WHERE ProductId = 1;

Что вернет последний запрос в session2?

* Price = 13.00;
* Price = 15.00;
* запрос не будет выполнен.

У нас установлен режим изоляции SNAPSHOT, поэтому вы получили последнюю версию строки, которая была доступна на момент запуска транзакции. Несмотря на то, что транзакция с изменением цены в session1 (Price = 15.00) уже подтверждена, в session2 вы все равно получите ту цену, которая была актуальна на момент начала этой транзакции.

1. В сессии (назовем ее session1) следующий код увеличивает цену товара с кодом 1 на 2.00 (первоначальное значение цены для этого товара 13.00):

BEGIN TRAN

UPDATE Products

SET Price += 2.00

WHERE ProductId = 1;

В другой сессии (назовем ее session2) был сделан вот такой запрос, который устанавливает код сотрудника, обслуживающего заказ с кодом 1, в 7 (первоначальное значение EmployeeId = 5):

BEGIN TRAN

UPDATE Orders

SET EmployeeId = 7

WHERE OrderId = 1;

Затем в рамках session1 было сделано обращение к таблице Orders и закрыта транзакция:

SELECT EmployeeId

FROM Orders WITH (READCOMMITTEDLOCK)

WHERE OrderId = 1;

COMMIT TRAN;

А в рамках session2 – обращение к таблице Products, и также была закрыта транзакция:

SELECT Price

FROM Products WITH (READCOMMITTEDLOCK)

WHERE ProductId = 1;

COMMIT TRAN;

Что будет выведено в результате работы этих SELECT’ов?

* возникнет взаимная блокировка,
* в session1 Employee = 7, в session2 Price = 15.00;
* в session1 Employee = 5, в session2 Price = 13.00;
* в session1 Employee = 7, в session2 запрос не будет выполнен;
* в session1 запрос не будет выполнен, в session2 Price = 15.00.

На данном этапе session1 удерживает монопольную блокировку для строки в таблице Products, а session2 делает то же самое в контексте таблицы Orders. Оба запроса завершились удачно, и ни одна транзакция не была заблокирована.

Затем чтобы выполнить операцию чтения, транзакция в session1 должна получить разделяемую блокировку. Но так как этот ресурс уже монопольно заблокирован другой транзакцией, session1 останавливает свою работу. На данном этапе происходит блокирование (пока что не взаимное). Хотя все еще остается вероятность того, что session2 завершит свою транзакцию, освободит все блокировки и позволит session1 получить доступ к ресурсу.

Чтобы выполнить операцию чтения, транзакция в session2 должна получить разделяемую блокировку для строки с товаром под номером 1, которая находится в таблице Products. Это действие конфликтует с монопольной блокировкой, удерживаемой session1. Процессы блокируют друг друга — то есть мы получили взаимное блокирование.

1. Что нужно сделать, чтобы получить информацию о блокировках:

* обратиться к динамическому управляющему представлению (dynamic management view, или DMV) под названием sys.dm\_tran\_locks,
* обратиться к SQL Server Profiler,
* просмотреть информацию о блокировках в Execution plan.

Чтобы получить информацию о блокировках (как текущих, так и тех, выдача которых все еще ожидается), обратитесь к динамическому управляющему представлению (dynamic management view, или DMV) под названием sys.dm\_tran\_locks.

1. Что нужно сделать, чтобы получить информацию о сессиях, участвующих в блокировках:

* обратиться к динамическому управляющему представлению (dynamic management view, или DMV) под названием sys.dm\_exec\_sessions,
* обратиться к SQL Server Profiler,
* просмотреть информацию о блокировках в Execution plan.

Множество полезной информации относительно сессий, участвующих в блокировании, можно найти в динамическом управляющем представлении под названием sys.dm\_exec\_sessions.

1. Что нужно сделать, чтобы получить информацию обо всех активных запросах, включая участвующих в блокировках:

* обратиться к динамическому управляющему представлению (dynamic management view, или DMV) под названием sys.dm\_exec\_requests,
* обратиться к SQL Server Profiler,
* просмотреть информацию о блокировках в Execution plan.

sys.dm\_exec\_requests — еще одно динамическое управляющее представление, которое может оказаться крайне полезным при решении проблем с блокированием. Оно содержит информацию обо всех активных запросах, включая блокирующие. Фактически вы легко можете извлечь только те из них, которые занимаются получением блокировок; для этого достаточно отобрать строки, в которых атрибут blocking\_session\_id больше нуля.

SELECT \*

FROM sys.dm\_exec\_requests

WHERE blocking\_session\_id > 0;

1. Что устанавливает параметр SET LOCK\_TIMEOUT 5000:

* время в миллисекундах на ожидание снятия блокировки,
* время в миллисекундах на выполнение транзакции с эксклюзивной (exclusive) блокировкой,
* время в миллисекундах на выполнение транзакции с разделяемой (shared) блокировкой.

По умолчанию сессии не имеют времени ожидания блокирования. Если вы хотите ограничить время, отведенное на ожидание получения блокировки, можете использовать параметр LOCK\_TIMEOUT. Значение задается в миллисекундах, например 5000 (5 секунд), 0 (нулевое) или –1 (неограниченное).

## Глава 10. Программируемые объекты

1. Является ли корректным следующий код:

DECLARE @i AS INT = 10;

1. Является ли корректным следующий код:

DECLARE i AS INT = 10;

1. Являются ли эквивалентными команды 1) и 2):
2. DECLARE @i AS INT;

SET @i = 10;

1. DECLARE @i AS INT = 10;
2. Являются ли эквивалентными команды 1) и 2):
3. DECLARE @a, b AS INT;

SET @a = 1, @b = 2;

1. DECLARE @a AS INT = 1;

DECLARE @b AS INT = 2;

Переменные предназначены для временного хранения данных и их дальнейшего использования в рамках одного и того же пакета (это одна или несколько команд, которые передаются в SQL Server и выполняются как единое целое).

Для объявления переменных используется команда DECLARE; с помощью команды SET происходит присваивание значений. Например, следующий код объявляет переменную под названием @i типа INT и присваивает ей число 10.

DECLARE @i AS INT;

SET @i = 10;

SQL Server 2008 и SQL Server 2012 поддерживают объявление и инициализацию переменных в рамках одной команды, как показано ниже.

DECLARE @i AS INT = 10;

Команда SET может одновременно работать только с одной переменной, поэтому чтобы присвоить значения нескольким атрибутам, вам понадобится соответствующее количество таких команд.

1. Есть таблица Persons следующего вида:

|  |  |  |
| --- | --- | --- |
| **PersonId** | **FirstName** | **LastName** |
| 1 | John | Lennon |
| 2 | Paul | McCartney |
| 3 | George | Harrison |
| 4 | Ringo | Starr |

Что будет выведено в результате следующего запроса:

DECLARE @fullName AS NVARCHAR(50);

SET @fullName = (

SELECT CONCAT(FirstName, ‘ ’, LastName)

FROM Persons

WHERE PersonId > 3);

SELECT @fullName;

* Ringo Starr,
* George Harrison, Ringo Starr,
* возникнет синтаксическая ошибка.

1. Есть таблица Persons следующего вида:

|  |  |  |
| --- | --- | --- |
| **PersonId** | **FirstName** | **LastName** |
| 1 | John | Lennon |
| 2 | Paul | McCartney |
| 3 | George | Harrison |
| 4 | Ringo | Starr |

Что будет выведено в результате следующего запроса:

DECLARE @fullName AS NVARCHAR(50);

SET @fullName = (

SELECT CONCAT(FirstName, ‘ ’, LastName)

FROM Persons

WHERE PersonId > 4);

SELECT @fullName;

* NULL,
* Ringo Starr,
* возникнет синтаксическая ошибка.

1. Есть таблица Persons следующего вида:

|  |  |  |
| --- | --- | --- |
| **PersonId** | **FirstName** | **LastName** |
| 1 | John | Lennon |
| 2 | Paul | McCartney |
| 3 | George | Harrison |
| 4 | Ringo | Starr |

Что будет выведено в результате следующего запроса:

DECLARE @fullName AS NVARCHAR(50);

SET @fullName = (

SELECT CONCAT(FirstName, ‘ ’, LastName)

FROM Persons

WHERE PersonId > 2);

SELECT @fullName;

* Ringo Starr,
* George Harrison, Ringo Starr,
* возникнет синтаксическая ошибка.

Команда SET может одновременно работать только с одной переменной, поэтому чтобы присвоить значения нескольким атрибутам, вам понадобится соответствующее количество таких команд.

1. Являются ли эквивалентными команды 1) и 2):
2. DECLARE @fullName AS NVARCHAR(50);

SET @fullName = (

SELECT CONCAT(FirstName, ‘ ’, LastName)

FROM Persons

WHERE PersonId = 1);

1. DECLARE @fullName AS NVARCHAR(50);

SELECT

@fullName = CONCAT(FirstName, ‘ ’, LastName)

FROM Persons

WHERE PersonId = 1;

SQL Server также поддерживает нестандартную версию инструкции SELECT, которая позволяет присваивать переменным множественные значения, полученные из одной строки.

1. Есть таблица Persons следующего вида:

|  |  |  |
| --- | --- | --- |
| **PersonId** | **FirstName** | **LastName** |
| 1 | John | Lennon |
| 2 | Paul | McCartney |
| 3 | George | Harrison |
| 4 | Ringo | Starr |

Что будет выведено в результате следующего запроса:

DECLARE @fullName AS NVARCHAR(50);

SELECT

@fullName = CONCAT(FirstName, ‘ ’, LastName)

FROM Persons

WHERE PersonId > 3;

SELECT @fullName;

* Ringo Starr,
* George Harrison, Ringo Starr,
* возникнет синтаксическая ошибка.

1. Есть таблица Persons следующего вида:

|  |  |  |
| --- | --- | --- |
| **PersonId** | **FirstName** | **LastName** |
| 1 | John | Lennon |
| 2 | Paul | McCartney |
| 3 | George | Harrison |
| 4 | Ringo | Starr |

Что будет выведено в результате следующего запроса:

DECLARE @fullName AS NVARCHAR(50);

SELECT

@fullName = CONCAT(FirstName, ‘ ’, LastName)

FROM Persons

WHERE PersonId > 2;

SELECT @fullName;

* George Harrison или Ringo Starr,
* George Harrison, Ringo Starr,
* возникнет синтаксическая ошибка.

Этот синтаксис имеет предсказуемое поведение, когда возвращается всего одна строка. Даже если строк несколько, код не завершается ошибкой; в данном случае содержимое переменных перезаписывается при доступе к каждой итоговой записи. Таким образом, переменные будут содержать значения той строки, которая оказалась последней в результирующем наборе. Это зависит от того, в каком порядке SQL Server перебирает полученные строки, причем этот порядок невозможно изменить.

1. Что такое пакет (batch):

* это одна или несколько команд языка T-SQL, отправленных клиентским приложением в SQL Server для их дальнейшего выполнения в виде единого целого,
* это одна или несколько команд языка T-SQL, которые выполняются как единое целое и откатываются в случае ошибки,
* оба утверждения верны.

Пакет — это одна или несколько команд языка T-SQL, отправленных клиентским приложением в SQL Server для их дальнейшего выполнения в виде единого целого. К пакету применяются такие этапы обработки, как синтаксический анализ, разрешение имен объектов и столбцов, проверка полномочий и оптимизация. Не стоит путать пакеты с транзакциями. Последние представляют собой атомарную единицу работы. Один пакет может содержать несколько транзакций, а одна транзакция может быть передана в виде нескольких пакетов. Если транзакция отменяется во время выполнения, SQL Server откатывает затронутые ею данные в исходное состояние (в котором они пребывали на момент открытия транзакции), вне зависимости от того, где начинался пакет.

1. Какая команда обозначает конец пакета (batch):

* GO,
* USE,
* EXEC,
* SELECT.

Утилиты SQL Server Management Studio, SQLCMD и OSQL поддерживают команду под названием GO, которая обозначает конец пакета. Стоит отметить, что это клиентская команда, то есть она выполняется не на сервере.

1. Корректен ли следующий запрос:

DECLARE @fullName AS NVARCHAR(50);

SELECT

@fullName = CONCAT(FirstName, ‘ ’, LastName)

FROM Persons

WHERE PersonId = 1;

GO

SELECT @fullName;

Область видимости переменной ограничивается пакетом, в котором она объявлена. При попытке сослаться на нее из другого пакета вы получите сообщение об ошибке, в котором будет сказано, что переменная не была определена.

1. Корректен ли следующий пакет:

IF OBJECT\_ID('dbo.Beatles', 'V') IS NOT NULL

DROP VIEW dbo.Beatles;

CREATE VIEW dbo.Beatles

AS

SELECT CONCAT(FirstName, ‘ ’, LastName) AS fullName

FROM dbo.Persons

WHERE PersonId < 5;

GO

Следующие команды нельзя использовать в одном пакете с любыми другими: CREATE DEFAULT, CREATE FUNCTION, CREATE PROCEDURE, CREATE RULE, CREATE SCHEMA, CREATE TRIGGER и CREATE VIEW. Например, код, представленный выше, является некорректным, поскольку в нем в рамках одного пакета находятся команды IF и CREATE VIEW.

Чтобы обойти эту проблему, разнесите команды по отдельным пакетам, указав GO после первой из них.

1. Корректен ли следующий пакет:

IF OBJECT\_ID('dbo.Persons', 'U') IS NOT NULL

DROP TABLE dbo.Persons;

CREATE TABLE dbo.Persons (

PersonId INT,

FirstName NVARCHAR(50)

);

ALTER TABLE dbo.Persons

ADD LastName NVARCHAR(50);

SELECT CONCAT(FirstName, ‘ ’, LastName)

FROM dbo.Persons;

GO

Изменение структуры объекта и его содержимого в рамках одного пакета может привести к ошибке разрешения имен, поскольку SQL Server будет руководствоваться старой структурой объекта.

В момент, когда внутри команды SELECT происходило разрешение имен, таблица Persons имела всего два столбца, поэтому попытка доступа к столбцу LastName привела к ошибке. Чтобы избежать подобных проблем, элементы языков DML и DDL следует выносить в разные пакеты.

1. Сколько строк будет вставлено в результате работы следующего пакета:

IF OBJECT\_ID('dbo.Numbers', 'U') IS NOT NULL

DROP TABLE dbo.Numbers;

CREATE TABLE dbo.Numbers (

Number INT IDENTITY

);

INSERT INTO dbo.Numbers DEFAULT VALUES;

GO 50

* 50,
* 0,
* 1,
* 2.

Команда GO на самом деле не является частью языка T-SQL; она используется в клиентских инструментах, таких как SSMS, для обозначения конца пакета. Данная команда поддерживает параметр, который указывает, сколько раз пакет будет выполнен на сервере.

1. Какой результат работы следующего пакета:

DECLARE @today AS DATE;

SET @today = '20212703';

IF YEAR(@today) <> YEAR(DATEADD(day, 1, @today))

PRINT 'Today is the last day of the year.';

ELSE

PRINT 'Today is not the last day of the year.';

Изменится ли результат работы, если из пакета удалить строку SET @today = '20212703'?

* 'Today is the last day of the year.', результат не изменится;
* 'Today is the last day of the year.', результат изменится;
* 'Today is not the last day of the year.', результат не изменится;
* 'Today is not the last day of the year.', результат изменится.

1. Что будет выведено в результате работы следующего пакета:

DECLARE @i AS INT = 1;

WHILE @i <= 10

SET @i = @i + 1;

PRINT @i;

* 11,
* 10,
* 1.

Язык T-SQL поддерживает инструкцию WHILE, которая делает возможным циклическое выполнение кода. Она выполняет одну и ту же команду или блок команд, пока предикат, указанный в конце, возвращает TRUE. Цикл прерывается при получении значений FALSE или UNKNOWN. В языке T-SQL нет элементов, которые позволяют выполнить код какое-то определенное количество раз, но такой подход можно легко эмулировать, используя инструкцию WHILE и одну переменную.

1. Что такое курсор:

* нереляционный набор с четко определенным порядком следования строк,
* реляционный набор с четко определенным порядком следования строк,
* нереляционный набор с произвольным порядком следования строк,
* реляционный набор с произвольным порядком следования строк.

1. Расставьте по порядку этапы для работы с курсором:
2. объявление курсора на основе запроса,
3. открытие курсора.
4. извлечение и сохранение внутри переменных содержимого атрибутов первой строки.
5. перебор содержимого курсора в цикле, пока не будет достигнута последняя запись (то есть функция @@FETCH\_STATUS не вернет 0); на каждой итерации производится необходимая обработка, а значения атрибутов текущей записи присваивается переменным,
6. закрытие курсора,
7. освобождение курсора.
8. Существует таблица Orders, в которой записаны все заказы (с полями OrderId, OrderDate). И таблица Order Details, в которой записаны данные по каждому заказу: OrderId, ProductId, Price, Quantity. Следующий код создает курсор, в результате работы которого подсчитывается средняя сумма заказа отдельно в будние и в выходные дни. Расставьте строки этого кода по порядку:
9. DECLARE @orderDate DATETIME, @total NUMERIC(10, 2),

@weekday INT, @weekend INT,

@weekdaySum INT, @weekendSum INT,

@currentWeekDay INT;

1. DECLARE MyCursor CURSOR FOR

SELECT O.OrderDate, SUM(Price \* Quantity) AS Total

FROM [Order Details] OD

JOIN Orders O ON OD.OrderID = O.OrderID

GROUP BY O.OrderDate, O.OrderID;

1. OPEN MyCursor;
2. SELECT @weekday = 0

,@weekend = 0

,@weekdaySum = 0

,@weekendSum = 0;

1. FETCH NEXT FROM MyCursor INTO @orderDate, @total;
2. WHILE @@FETCH\_STATUS = 0

BEGIN

1. SET @currentWeekDay = DATEPART(dw, @orderDate);
2. IF @currentWeekDay = 1 OR @currentWeekDay = 7

BEGIN

SET @weekend += 1;

SET @weekendSum += @total;

END

ELSE

BEGIN

SET @weekday += 1;

SET @weekdaySum += @total;

END

1. FETCH NEXT FROM MyCursor INTO @orderDate, @total;
2. END
3. CLOSE MyCursor;

DEALLOCATE MyCursor;

1. SELECT @weekdaySum / @weekday, @weekendSum / @weekend
2. Что такое временные таблицы:

* ее данные доступны для текущей сессии или текущего пакета,
* существует три вида временных таблиц: локальные, глобальные и табличные переменные,
* содержимое временных таблиц хранится в БД tempdb,
* все утверждения верны.

1. В рамках одной сессии была создана временная таблица Numbers для хранения чисел от 1 до 5:

IF OBJECT\_ID('tempdb.dbo.#Numbers') IS NOT NULL

DROP TABLE dbo.#Numbers;

GO

CREATE TABLE #Numbers (

Number INT NOT NULL

);

INSERT INTO #Numbers(Number)

VALUES (1), (2), (3), (4), (5);

Затем в другой сессии был запущен следующий запрос:

SELECT \*

FROM dbo.#Numbers;

Каков результат его выполнения?

* запрос не будет выполнен, так как таблица Numbers видна только в той сессии, в которой она определена,
* в результате запроса будут выведены числа от 1 до 5,
* запрос не будет выполнен, так как в запросах ко временным таблицам нельзя использовать символ \* (звездочка),
* ни один запрос не будет выполнен, так как в имени таблицы нельзя использовать символ #.

Имена локальных временных таблиц должны начинаться со знака решетки, например #T1. Их содержимое (как и содержимое остальных двух видов временных таблиц) хранится в БД tempdb. Локальные временные таблицы видны только в контексте сессии, в которой они были созданы; доступ к ним можно получить на том уровне, где находится их определение, и далее по стеку вызовов (в том числе внутри вложенных процедур, функций, триггеров и динамических пакетов). Они автоматически уничтожаются, когда текущий уровень выходит за рамки пространства имен.

Одним из очевидных способов применения временных таблиц является хранение промежуточных результатов (например, во время выполнения цикла) с возможностью последующего доступа к ним.

Иногда возникает необходимость многократного обращения к данным, которые представляют собой результаты ресурсоемких вычислений. Чтобы не повторять все эти ресурсоемкие операции, результаты проще сохранить в локальной временной таблице; после этого достаточно взять два экземпляра имеющихся у нас данных и выполнить соединение.

1. В рамках одной сессии была создана временная таблица Numbers для хранения чисел от 1 до 5:

IF OBJECT\_ID('tempdb.dbo.##Numbers') IS NOT NULL

DROP TABLE dbo.##Numbers;

GO

CREATE TABLE ##Numbers (

Number INT NOT NULL

);

INSERT INTO ##Numbers(Number)

VALUES (1), (2), (3), (4), (5);

Затем в другой сессии был запущен следующий запрос:

SELECT \*

FROM dbo.##Numbers;

Каков результат его выполнения?

* в результате запроса будут выведены числа от 1 до 5,
* запрос не будет выполнен, так как таблица Numbers видна только в той сессии, в которой она определена,
* запрос не будет выполнен, так как в запросах ко временным таблицам нельзя использовать символ \* (звездочка),
* ни один запрос не будет выполнен, так как в имени таблицы нельзя использовать символы ##.

Глобальная временная таблица доступна для всех сессий и обозначается с помощью двойного знака решетки, например ##T1. Ее удаление происходит в момент отключения сессии, в которой она была создана (при условии отсутствия внешних ссылок). Глобальные временные таблицы используются для хранения общедоступных промежуточных данных. Для обращения к ним (в том числе с помощью элементов DDL и DML) не требуется никаких специальных полномочий. Конечно, это означает, что любой пользователь может изменять или даже удалять их содержимое, поэтому прежде чем использовать их в своих запросах, следует взвесить все за и против.

1. В рамках одной сессии была создана табличная переменная Numbers для хранения чисел от 1 до 5:

DECLARE @Numbers TABLE (

Number INT NOT NULL

);

INSERT INTO @Numbers(Number)

VALUES (1), (2), (3), (4), (5);

Спустя некоторое время в той же сессии был запущен следующий запрос:

SELECT \*

FROM @Numbers;

Каков результат его выполнения?

* запрос не будет выполнен, так как табличная переменная Numbers видна только в том пакете, в котором она определена,
* в результате запроса будут выведены числа от 1 до 5,
* запрос не будет выполнен, так как в запросах ко временным таблицам нельзя использовать символ \* (звездочка),
* ни один запрос не будет выполнен, так как в имени таблицы нельзя использовать символы @.

Табличные переменные, которые объявляются с помощью команды DECLARE, имеют много общего с локальными временными таблицами.

Как и другие виды временных таблиц, табличные переменные физически хранятся в БД tempdb (вопреки расхожему мнению о том, что они находятся исключительно в оперативной памяти). Их область видимости ограничена не только сессией, в которой они были созданы, но и текущим пакетом. При этом доступ к ним закрыт даже для вложенных и всех последующих пакетов, объявленных в рамках сессии.

При откате транзакции, которая была объявлена вручную, отменяются все изменения, внесенные ею во временные таблицы; однако в случае с табличными переменными отменяется только действие последней команды, которая завершилась ошибкой или не успела выполниться до конца.

1. В рамках одной сессии были созданы два пакета. В первом пакете была создана табличная переменная Numbers для хранения чисел от 1 до 5:

DECLARE @Numbers TABLE (

Number INT NOT NULL

);

INSERT INTO @Numbers(Number)

VALUES (1), (2), (3), (4), (5);

SELECT \*

FROM @Numbers;

Во втором пакете также была создана табличная переменная Numbers для хранения чисел от 1 до 5:

IF TYPE\_ID('dbo.NumbersType') IS NOT NULL

DROP TYPE dbo.NumbersType;

CREATE TYPE dbo.NumbersType AS TABLE (

Number INT NOT NULL

);

GO

DECLARE @Numbers AS dbo.NumbersType;

INSERT INTO @Numbers(Number)

VALUES (1), (2), (3), (4), (5);

SELECT \*

FROM @Numbers;

Что будет выведено в результате выполнения этих пакетов:

* оба пакета будут выполнены и иметь одинаковые результаты: будут выведены числа от 1 до 5,
* первый пакет будет выполнен, а второй нет, так как табличная переменная Numbers уже была объявлена в текущей сессии,
* первый пакет будет выполнен, в второй – нет, так как переменная Numbers объявлена некорректно,
* ни один пакет не будет выполнен, так как в имени таблицы нельзя использовать символы @.

Табличные типы поддерживаются в SQL Server 2008 и SQL Server 2012. Чтобы создать табличный тип, в БД необходимо сохранить определение таблицы, на основе которого впоследствии будут объявляться табличные переменные. Позже его можно будет использовать во входящих параметрах для хранимых процедур и пользовательских функций.

1. Какие инструменты существуют для выполнения динамического кода:

* команда EXEC,
* хранимая процедура sp\_executesql,
* оба утверждения верны.

SQL Server позволяет создавать пакеты в виде символьных строк, которые можно выполнять как обычный код. Для динамического выполнения кода существует два инструмента: команда EXEC (сокращенно от EXECUTE) и хранимая процедура sp\_executesql.

Будьте осторожны, формируя запросы на основе пользовательского ввода. Злоумышленники могут внедрить в ваше приложение вредоносный код. Лучше откажитесь от выполнения строковых данных, которые приходят извне (например, посредством параметров) или по крайней мере, выполняйте тщательную проверку и старайтесь отслеживать попытки внедрения SQL-кода. В электронном справочнике есть хорошая статья на эту тему — «Атака SQL Injection».

1. Что будет выведено в результате выполнения следующего кода:

DECLARE @sql AS VARCHAR(100);

SET @sql = 'SELECT \* FROM Persons';

EXEC(@sql);

* запрос будет выполнен: будут выведены все строки из таблицы Persons (если они есть),
* запрос не будет выполнен, так как в динамических запросах нельзя использовать символ \* (звездочка),
* запрос не будет выполнен, так как в записи динамического запроса отсутствует символ ; (точка с запятой).

1. Что будет выведено в результате выполнения следующего кода:

DECLARE @sql AS NVARCHAR(100);

SET @sql =

N'SELECT \*

FROM Persons

WHERE PersonId = @john';

EXEC sp\_executesql

@stmt = @sql,

@params = N'@john AS INT',

@john = 1;

* запрос будет выполнен: будет выведена строка из таблицы Persons, для которой PersonId = 1 (если такая есть),
* запрос будет только в том случае, если в таблице Persons есть строка PersonId = 1,
* запрос не будет выполнен, так как в динамических запросах нельзя использовать символ \* (звездочка),
* запрос не будет выполнен, так как в записи динамического запроса отсутствует символ ; (точка с запятой).

Вслед за командой EXEC в SQL Server появилась поддержка хранимой про цедуры sp\_executesql. Она более безопасная и гибкая, так как ее интерфейс заключается в поддержке входящих и исходящих параметров. Пакеты с кодом, которые передаются этой процедуре, должны иметь формат Unicode.

Предоставленная возможность использовать входящие и исходящие параметры при формировании динамического кода позволяет создавать куда более безопасные и эффективные запросы. Параметры, которые передаются извне, не могут быть интерпретированы как часть кода — их реально использовать только в качестве операндов в рамках выражения. Благодаря этому легко избежать атак типа SQL injection.

Хранимая процедура sp\_executesql может иметь более высокую производительность по сравнению с командой EXEC, поскольку поддержка параметризации помогает при многократном использовании закэшированных планов выполнения. Такой план автоматически генерируется для каждого запроса; в нем описывается набор инструкций, которые должны применяться к объекту, порядок их обработки, список индексов и способ их использования, тип алгоритма соединения данных и т. д.

Хранимые процедуры с параметрами позволяют наиболее эффективно использовать имеющиеся планы выполнения, ведь даже если параметры изменятся, строка запроса останется прежней. Если вы по каким-то причинам решили не сохранять свой код в БД, у вас все равно остается возможность применять процедуру sp\_executesql, повышая тем самым вероятность многократного использования планов выполнения.

Процедура sp\_executesql имеет два входящих значения (оба в формате Unicode) и блок, в котором они присваиваются. Первое значение, @stmt — это строка с кодом, который вы хотите выполнить. Во втором значении, @params, передаются вход ящие и исходящие параметры, которые разделяются запятыми и инициализируются в следующем блоке.

1. Какие виды процедур поддерживаются в SQL Server:

* пользовательские функции, хранимые процедуры и триггеры,
* хранимые процедуры и триггеры,
* пользовательские функции и триггеры.

Процедуры — это программируемые объекты, которые инкапсулируют код, предназначенный для вычисления результата или выполнения каких-то задач. SQL Server поддерживает три вида таких объектов: пользовательские функции, хранимые процедуры и триггеры. Процедуры можно создавать как на языке T-SQL, так и посредством кода Microsoft .NET, используя интеграцию с CLR.

1. Код, представленный ниже, создает пользовательскую функцию под названием GetAge, которая возвращает возраст человека на текущий момент:

IF OBJECT\_ID('dbo.GetAge') IS NOT NULL

DROP FUNCTION dbo.GetAge;

CREATE FUNCTION dbo.GetAge (

@birthDate AS DATETIME

)

RETURNS INT

AS

BEGIN

DECLARE @now DATETIME = SYSDATETIME();

DECLARE @age INT = YEAR(@now) - YEAR(@birthDate);

IF (@birthDate > DATEADD(YEAR, -@age, @now))

SET @age -= 1;

RETURN @age;

END;

Возникнет ли ошибка при запуске этого пакета?

* да, так как в одном пакете нельзя использовать команду CREATE FUNCTION и любую другую,
* да, так как неправильно указан тип возвращаемого значения,
* да, так как в функции DATEADD нельзя использовать отрицание,
* ошибки не возникнет.

1. Код, представленный ниже, создает пользовательскую функцию под названием GetAge, которая возвращает возраст человека на текущий момент, а затем использует ее в запросе SELECT:

IF OBJECT\_ID('dbo.GetAge') IS NOT NULL

DROP FUNCTION dbo.GetAge;

GO

CREATE FUNCTION dbo.GetAge (

@birthDate AS DATETIME

)

RETURNS INT

AS

BEGIN

DECLARE @now DATETIME = SYSDATETIME();

DECLARE @age INT = YEAR(@now) - YEAR(@birthDate);

IF (@birthDate > DATEADD(YEAR, -@age, @now))

SET @age -= 1;

RETURN @age;

END;

SELECT FirstName

,LastName

,BirthDate

,dbo.GetAge(BirthDate) AS Age

FROM Person

Возникнет ли ошибка при запуске этого пакета?

* да, так как в одном пакете нельзя использовать команду CREATE FUNCTION и любую другую,
* да, так как неправильно указан тип возвращаемого значения,
* ошибки не возникнет.

1. Код, представленный ниже, создает пользовательскую функцию под названием GetAge, которая возвращает возраст человека на текущий момент:

CREATE FUNCTION dbo.GetAge (

@birthDate AS DATETIME

)

RETURNS

AS

BEGIN

DECLARE @now DATETIME = SYSDATETIME();

DECLARE @age INT = YEAR(@now) - YEAR(@birthDate);

IF (@birthDate > DATEADD(YEAR, -@age, @now))

SET @age -= 1;

RETURN @age;

END;

Возникнет ли ошибка при запуске этого пакета?

* да, так как не указан тип возвращаемого значения,
* да, так как в функции DATEADD нельзя использовать отрицание,
* ошибки не возникнет.

Пользовательские функции инкапсулируют код, предназначенный для вычисления и возвращения определенного результата (с возможностью использования входящих параметров).

В SQL Server пользовательские функции могут возвращать как скалярные, так и табличные значения. Любую функцию реально разместить внутри запроса — либо вместо скалярного выражения, либо в рамках инструкции FROM (в случае с табличными значениями).

У пользовательских функций не должно быть никаких побочных эффектов. Они не способны изменять структуру или содержимое БД. Это ограничение не столь очевидно, как может показаться. Например, вызов функций RAND (возвращает случайное значение) или NEWID (возвращает глобальный уникальный идентификатор GUID) тоже не проходит бесследно. Если функции RAND не передать начальное значение, SQL Server сгенерирует его автоматически, основываясь на предыдущем вызове; следовательно, промежуточные результаты должны храниться в БД. То же самое касается функции NEWID: система должна сохранять определенную информацию, которая будет использоваться при следующем вызове. Таким образом, вы не можете делать вызовы RAND и NEWID в своих пользовательских функциях.

1. Код, представленный ниже, создает хранимую процедуру под названием GetAge, которая возвращает возраст человека на текущий момент:

CREATE PROC dbo.GetAge

@birthDate AS DATETIME

,@today AS DATETIME = '20210331'

,@age AS INT OUTPUT

AS

SET @age = YEAR(@today) - YEAR(@birthDate);

IF (@birthDate > DATEADD(YEAR, -@age, @today))

SET @age -= 1;

Можно ли вызвать эту процедуру только с одним входным параметром, используя следующий код:

DECLARE @myAge AS INT;

EXEC dbo.GetAge

@birthDate = '20040720',

@age = @myAge OUTPUT;

Хранимые процедуры — это наборы кода на языке T-SQL, которые находятся на серверной стороне. Они поддерживают входящие и исходящие параметры, могут возвращать результирующие наборы и вызывать код, который имеет побочные эффекты. С их помощью легко изменять не только сами данные, но и их структуру.

1. В теле какого типа процедуры может находиться код, представленный ниже:

SET @age = YEAR(@today) - YEAR(@birthDate);

IF (@birthDate > DATEADD(YEAR, -@age, @today))

SET @age -= 1;

* в пользовательской функции,
* в хранимой процедуре,
* и в пользовательской функции, и в хранимой процедуре.

1. Необходимо написать процедуру, которая вставляет дату (она передается в качестве параметра или действует значение по умолчанию, если параметр не указан) в поле OrderDate таблицы Orders, если значение в этом поле равно NULL. Какой тип процедуры надо выбрать для решения этой задачи:

* пользовательскую функцию,
* хранимую процедуру,
* пользовательскую функцию или хранимую процедуру.

1. Необходимо написать процедуру, которая подсчитывает и возвращает среднюю стоимость заказа за определенный период (период передается в качестве двух входных параметров) для таблицы Orders. Какой тип процедуры надо выбрать для решения этой задачи:

* пользовательскую функцию,
* хранимую процедуру,
* пользовательскую функцию или хранимую процедуру.

1. Необходимо написать процедуру, которая подсчитывает и возвращает среднюю стоимость заказа за определенный период (период передается в качестве двух входных параметров) для таблицы Orders. Результат работы этой процедуры планируется использовать в запросах на этапе SELECT. Какой тип процедуры надо выбрать для решения этой задачи:

* пользовательскую функцию,
* хранимую процедуру,
* пользовательскую функцию или хранимую процедуру.

1. В каком типе процедуры можно использовать следующую команду:

INSERT INTO Persons(FirstName, LastName)

VALUES (‘John’, ‘Lennon’);

* в пользовательской функции,
* в хранимой процедуре,
* и в пользовательской функции, и в хранимой процедуре.

1. В каком типе процедуры можно использовать следующую команду:

INSERT INTO Persons(FirstName, LastName)

SELECT FirstName, LastName

FROM Employees;

* в пользовательской функции,
* в хранимой процедуре,
* в триггере,
* и в хранимой процедуре, и в триггере,
* и в пользовательской функции, и в хранимой процедуре, и в триггере.

1. В каком типе процедуры можно использовать следующую команду:

UPDATE Persons

SET FirstName = ‘John’

WHERE PersonId = 1;

* в пользовательской функции,
* в хранимой процедуре,
* и в пользовательской функции, и в хранимой процедуре.

1. В каком типе процедуры можно использовать следующую команду:

BEGIN TRY

SET @result = @value1 / value2;

END TRY

BEGIN CATCH

SET @result = -1;

END CATCH;

* в пользовательской функции,
* в хранимой процедуре,
* и в пользовательской функции, и в хранимой процедуре.

Отличия пользовательских функций от хранимых процедур:

|  |  |
| --- | --- |
| **Функция** | **Хранимая процедура** |
| Функция должна возвращать значение. | Хранимая процедура может как возвращать, так и не возвращать значение. |
| Функции не могут возвращать несколько результирующих наборов. | Хранимая процедура может сформировать и вернуть несколько результирующих наборов данных. |
| Функции можно использовать в операторе SELECT. | Процедуры нельзя использовать в операторе SELECT и во всех его секциях (WHERE, JOIN, HAVING и т.д.), так как процедуры вызываются с помощью команды EXECUTE или EXEC. |
| В функциях можно использовать только оператор SELECT на выборку данных.  Операторы DML (INSERT, UPDATE, DELETE) для модификации данных использовать нельзя. | В хранимых процедурах можно использовать оператор SELECT, а также операторы DML (INSERT, UPDATE, DELETE) для модификации данных. |
| Из функции нельзя вызвать хранимые процедуры. | В хранимых процедурах можно вызывать и функции, и другие хранимые процедуры. |
| Конструкцию для обработки ошибок TRY CATCH нельзя использовать в функциях. Так же как нельзя использовать инструкцию RAISERROR. | В хранимых процедурах можно использовать и конструкцию TRY CATCH, и инструкцию RAISERROR. |
| В функциях запрещено использование транзакций. | В хранимых процедурах транзакции разрешены. |
| В функциях можно использовать только табличные переменные, временные таблицы использовать не получится. | В хранимых процедурах можно использовать как табличные переменные, так и временные таблицы. |
| В функциях нельзя использовать динамический SQL. | В процедурах можно использовать динамический SQL. |
| В функциях можно использовать только входные параметры. | В хранимых процедурах можно использовать как входные, так и выходные параметры. |

1. Каким способом можно запустить триггер:

* триггер нельзя вызвать вручную,
* как пользовательскую функцию <имя триггера>(<параметры>)
* как хранимую процедуру EXEC <имя триггера>.

1. Триггер срабатывает, когда происходит событие, связанное с:

* операциями изменения данных (DML-триггер),
* операциями изменения структуры БД (DDL-триггер),
* операциями изменения доступа к данным (DCL-триггер),
* операциями изменения данных и доступа к данным (DML и DDL триггеры),
* с любыми изменениями в БД.

Триггер — это специальный вид хранимых процедур, который нельзя вызвать вручную. Он срабатывает и запускает свой код в момент, когда происходит определенное событие. SQL Server поддерживает два типа событий. Первый связан с операциями изменения данных (DML-триггеры), такими как INSERT, а второй относится к командам, которые определяют структуру БД (DDL-триггеры), например CREATE TABLE.

Триггеры используются для разных задач: это может быть проверка данных, обеспечение целостности, которое нельзя гарантировать путем ограничений, соблюдение различных правил и т. д.

Если к срабатыванию триггера привело событие, которое входит в транзакцию, то сам триггер тоже будет считаться частью этой транзакции. Если вызвать команду ROLLBACK TRAN в коде триггера, то все изменения, которые он успел выполнить, будут отменены. Также будут отменены все изменения, которые произошли в транзакции, связанной с триггером.

1. Возможно ли приостановить действие триггера, не удаляя его при этом?

Бывает, что мы хотим приостановить действие триггера, но удалять его полностью не хотим. В этом случае его можно временно отключить с помощью команды DISABLE TRIGGER <TriggerName> ON Table1. Включить его обратно можно с помощью команды ENABLE TRIGGER.

1. DML-триггеры срабатывают:

* после завершения соответствующего события,
* вместо соответствующего события,
* оба утверждения верны.

DML-триггеры могут срабатывать либо после завершения соответствующего события, либо вместо него. В первом случае допускается использование только постоянных таблиц, а во втором можно дополнительно применить представления.

Каждый триггер ассоциируется с определенной таблицей или представлением, имя которых указывается после слова ON. Затем устанавливается тип триггера. Мы можем использовать один из двух типов:

AFTER: выполняется после выполнения действия. Определяется только для таблиц.

INSTEAD OF: выполняется вместо действия (то есть по сути действие - добавление, изменение или удаление - вообще не выполняется). Определяется для таблиц и представлений.

После типа триггера идет указание операции, для которой определяется триггер: INSERT, UPDATE или DELETE.

Для триггера AFTER можно применять сразу для нескольких действий, например, UPDATE и INSERT. В этом случае операции указываются через запятую. Для триггера INSTEAD OF можно определить только одно действие.

И затем после слова AS идет набор выражений SQL, которые собственно и составляют тело триггера.

Таблицы, изменения в которых приводят к срабатыванию триггера, иногда помечаются ключевыми словами inserted или deleted. Первое позволяет получить новое содержимое строк, затронутых командами INSERT, MERGE или UPDATE; второе дает доступ к старым значениям, которые были актуальны перед выполнением команд DELETE, MERGE или UPDATE. Если триггер сработает вместо события, полученная вами таблица будет содержать строки, которые должны были быть затронуты в результате соответствующих операций.

1. Есть таблица Table1 с полем Field1 типа INT. Можно ли для этой таблицы определить триггер, который не позволяет вводить значения от 10 до 20 в поле Field1?

CREATE TRIGGER myTrigger

ON Table1

AFTER INSERT

AS BEGIN

DELETE FROM Table1

WHERE Field1 BETWEEN 10 AND 20

END;

Триггеры не могут изменять измененные данные (Inserted или Deleted), иначе вы могли бы получить бесконечную рекурсию, поскольку изменения снова вызвали триггер. Один из вариантов решения этой проблемы заключается в том, чтобы триггер откатил транзакцию.

Стандарт для SQL заключается в том, что вставленные и удаленные строки не могут быть изменены триггером. Основная причина этого заключается в том, что модификации могут вызвать бесконечную рекурсию. В общем случае эта оценка может включать несколько триггеров во взаимно рекурсивном каскаде.