Оглавление

[**Книга С. Куликова** 2](#_Toc182856845)

[Глава 1. Отношения 3](#_Toc182856846)

[Глава 2. Ключи 6](#_Toc182856847)

[Глава 2. Связи 9](#_Toc182856848)

[Глава 3. Индексы 15](#_Toc182856849)

[Глава 4. Нормализация 26](#_Toc182856850)

[Глава 5. Проектирование БД 28](#_Toc182856851)

[Глава 5. View, check, trigger, хранимые процедуры и функции… 30](#_Toc182856852)

# **Книга С. Куликова**

## Глава 1. Отношения

1. Что такое база данных:

* это большая совокупность данных, подчинённых определенным правилам,
* программное обеспечение, которое позволяет создавать, хранить и использовать большие объемы данных,
* приложение, с помощью которого пользователь может просматривать данные.

1. Что такое система управления базой данных (СУБД):

* программное обеспечение для описания, создания, использования, контроля и управления базами данных,
* приложение, с помощью которого пользователь может просматривать/изменять данные.

1. Что такое Microsoft SQL Server Management Studio:

* программное обеспечение, позволяющее взаимодействовать с SQL Server от компании Microsoft,
* СУБД от компании Microsoft,
* SQL Server от компании Microsoft,
* база данных от компании Microsoft.

База данных — это большая совокупность данных, подчинённых дополнительным правилам. Такие правила зависят от вида базы данных, а за их соблюдение отвечает система управления базами данных (СУБД). Система управления базами данных, СУБД (database management system, DBMS) — система (базирующаяся на программном и аппаратном обеспечении) для описания, создания, использования, контроля и управления базами данных. Упрощённо: программное средство, управляющее базами данных.

Стоит отметить важный для начинающих: подавляющее большинство СУБД не имеет никакого «человеческого интерфейса», представляет собой сервис (демон в \*nix-системах) и взаимодействует с внешним миром по специальным протоколам (чаще всего, построенным поверх TCP/IP). Такие известные продукты как MySQL Workbench, Microsoft SQL Server Management Studio, Oracle SQL Developer и им подобные — это не СУБД, это лишь клиентское программное обеспечение, позволяющее нам взаимодействовать с СУБД.

Домен данных (attribute domain) — набор всех возможных значений атрибута отношения.

Упрощённо, на примере: номер телефона, фамилия, название улицы и т.д.

Суть этого понятия проще всего пояснить на примере. Допустим, в некотором отношении есть атрибут «Код товара», который представлен строкой. И также известно, что все коды товара формируются по правилу «три буквы английского алфавита в верхнем регистре, четыре цифры, две буквы английского алфавита в нижнем регистре» (например, «ABC1234yz»). Очевидно, что строка «понедельник» не является кодом товара, т.е. не входит в соответствующий домен, хоть и относится к тому же типу данных (к строкам).

Понятие домена важно в силу того, что данные считаются сравнимыми только в случае принадлежности к одному домену, т.е. бессмысленно сравнивать код товара и название дня недели (хотя технически такую операцию позволит вам выполнить любая СУБД).

В подавляющем большинстве СУБД приходится использовать триггеры или проверки для контроля принадлежности данных одному домену. Несмотря на некоторую сложность такого контроля и тот факт, что выполнение соответствующих действий снижает производительность некоторых операций с данными, оно позволяет ощутимо повысить защищённость данных, снижая риск передачи (и сохранения) некорректных значений.

Реляционная модель (relational model) — математическая теория, описывающая структуры данных, логику контроля целостности данных и правила управления данными.

Упрощённо: модель для описания реляционных баз данных.

Отношение (relation) — множество сущностей, обладающих одинаковым набором атрибутов. В контексте реляционных баз данных отношение состоит из заголовка (схемы) и тела (набора кортежей).

Упрощённо: математическая модель таблицы базы данных.

Атрибут (attribute) — именованное свойство сущности (отношения).

Упрощённо: столбец (колонка) таблицы.

Кортеж (tuple) — часть отношения, представляющая собой уникальную взаимосвязанную комбинацию значений, каждое из которых соответствует своему атрибуту.

Упрощённо: строка (запись) таблицы.

Метод доступа (storage engine, database engine) — программный компонент, используемый СУБД для добавления, чтения, обновления и удаления данных.

Упрощённо: специальная часть СУБД, с помощью которой она работает с диском (файлами).

К сожалению, почти бессмысленно описывать «методы доступа в принципе», т.к. их набор, конкретная реализация, структура, возможности и т.д. весьма различны и жёстко связаны с конкретной СУБД. При этом многие СУБД позволяют использовать несколько методов доступа одновременно, что расширяет возможности проектирования базы данных на физическом уровне, но и усложняет его.

1. Есть таблица Numbers, состоящая из одного поля Number, в которой записаны числа 1, 2, 4, 5 (всего в таблице 4 строки). Как вставить в таблицу новую строку со значением 3 между строками со значениями 2 и 4:

* это невозможно, так как порядком записи данных нельзя управлять,
* при вставке строки нужно применить специальную конструкцию (см. документацию по T-SQL).

Очень частый (и в корне неверный) вопрос: «Как мне вставить в таблицу новую строку между строками такой-то и такой-то?» Никак.

Тот факт, что СУБД вынуждена хранить данные в некоторой последовательности (это не только объективная необходимость, но и способ оптимизации операций поиска) никак не означает, что вам будет что-то известно об этой последовательности, и уж тем более, что СУБД позволит вам ею управлять.

1. Можно ли создать таблицу и записать в нее две абсолютно одинаковые строки?

В отношениях не допускается наличие одноимённых атрибутов и совпадающих кортежей, в таблицах же (как минимум — производных, т.е. полученных выполнением запроса) это требование может нарушаться.

Начнём с дублирования строк. Почти любая современная реляционная СУБД позволит вам создать таблицу без первичных ключей и/или уникальных индексов, и затем размещать там полностью идентичные строки. Зачем вам это нужно, и как вы теперь собираетесь различать эти строки — отдельный вопрос (подсказка: не делайте так никогда, это приводит к уйме проблем), но технологически здесь нет никаких препятствий.

Что до одноимённых атрибутов, то СУБД не допустит такого при создании таблицы, но при выполнении SQL-запросов с JOINтакая ситуация возможна.

## Глава 2. Ключи

1. Что такое потенциальный ключ (candidate key):

* несократимое подмножество атрибутов отношения, уникально идентифицирующее любой кортеж,
* поле или набор полей, по совокупности значений которых одну строку таблицы можно гарантированно отличить от другой,
* оба утверждения верны.

Потенциальный ключ (candidate key) — несократимое подмножество атрибутов отношения, уникально идентифицирующее любой кортеж. Иными словами, потенциальный ключ — это такой суперключ, в котором нет «лишних» элементов. Как правило, один из потенциальных ключей в дальнейшем становится первичным ключом (а остальные потенциальные ключи становятся альтернативными ключами).

Упрощённо: кандидат в первичные ключи, т.е. поле или набор полей, по совокупности значений которых одну строку таблицы можно гарантированно отличить от другой; в таком наборе нет лишних полей, т.е. убрав из набора хотя бы одно поле, мы уже не сможем гарантированно различить любые строки таблицы.

1. Что такое первичный ключ (primary key, PK):

* потенциальный ключ, выбранный в качестве основного средства гарантированной идентификации кортежей отношения,
* уникальный идентификатор строки в таблице,
* оба утверждения верны.

Первичный ключ (primary key, PK) — потенциальный ключ, выбранный в качестве основного средства гарантированной идентификации кортежей отношения. Упрощённо: уникальный идентификатор строки в таблице.

Первичный ключ должен обладать свойством несократимости (т.е. не должен включать в себя «лишние» элементы, которые можно отбросить без потери уникальности значений).

Также (поскольку понятие «первичного ключа» относится уже не столько к общей теории баз данных, сколько к практической работе СУБД) стоит отметить, что первичный ключ должен обладать свойством минимальности, которое следует трактовать буквально: если существует несколько потенциальных ключей, состоящих из одинакового количества атрибутов, стоит выбрать в качестве первичного тот потенциальный ключ, который «физически меньше», т.е. содержит меньше данных: чем меньше данных СУБД придётся сравнивать (для обеспечения свойства уникальности значений), тем быстрее она выполнит эту операцию.

1. Что такое альтернативный ключ (alternate key):

* потенциальный ключ, не выбранный в качестве первичного ключа,
* уникальный идентификатор строки в таблице,
* оба утверждения верны.

Альтернативный ключ (alternate key) — потенциальный ключ отношения, не выбранный в качестве первичного ключа. Упрощённо: кандидат в первичные ключи, так и не ставший первичным ключом.

Чтобы привести поведение СУБД в соответствие с требованиями предмет-ной области, на альтернативных ключах создают т.н. «ограничение уникальности» (CHECK UNIQUE), которое в большинстве СУБД обозначает построение на данном ключе т.н. «уникального индекса».

Простой ключ (simple key) — ключ, состоящий из одного атрибута отношения. Упрощённо: ключ, построенный на ровно одной колонке таблицы.

Составной ключ (compound key / composite key) — ключ, состоящий из двух и более атрибутов отношения. Упрощённо: ключ, построенный на двух (и более) колонках таблицы.

1. Что такое внешний ключ (foreign key, FK):

* атрибут (или группа атрибутов) отношения, содержащий в себе копии значений первичного ключа другого отношения,
* уникальное значение строки в таблице,
* оба утверждения верны.

Внешний ключ (foreign key, FK) — атрибут (или группа атрибутов) отношения, содержащий в себе копии значений первичного ключа другого отношения. Упрощённо: поле в дочерней (подчинённой) таблице, содержащее в себе копии значений первичного ключа родительской (главной) таблицы.

Рекурсивный внешний ключ (recursive foreign key, RFK) — атрибут (или группа атрибутов) отношения, содержащий в себе копии значений первичного ключа этого же отношения. Упрощённо: внешний ключ, находящийся в той же таблице, что и первичный ключ, значения которого он содержит; вариант внешнего ключа в случае, когда таблица ссылается на саму себя. Классическим случаем их использование является организация иерархических структур

Критически важно! Типы данных первичного ключа и соответствующих внешних ключей должны быть полностью идентичными (т.е. тип, размер, «знаковость», кодировка, формат и т.д. — любые подобные свойства, которые есть у соответствующих полей, должны совпадать (кроме автоинкрементируемости, которой не должно быть у внешнего ключа)). Это важно потому, что в такой ситуации СУБД не тратит дополнительные ресурсы на преобразование данных при сравнении значений первичного и внешнего ключа. Также существует рекомендация создавать на внешних ключах индексы, что может ускорить выполнение JOIN-запросов.

Почти все современные СУБД позволяют очень грубо нарушать это правило (допустим, сделать первичный ключ BIGINT-числом, а внешний ключ — VARCHAR-строкой), но такое издевательство над базой данных как минимум выливается в катастрофическое падение производительности, а как максимум и вовсе к ошибкам при выполнении некоторых запросов.

1. Имеет ли значение последовательность полей в составном первичном ключе?

Последовательность полей в составном первичном ключе (равно как и в составном индексе) играет очень важную роль. Чаще всего первичный ключ является кластерным индексом, т.е. СУБД физически упорядочивает данные на диске по значениям первичного ключа. Если такой ключ состоит из одного поля, никаких проблем нет. Но если он состоит из нескольких полей, СУБД может эффективно производить поиск только по комбинации входящих в состав первичного ключа полей или по первому полю отдельно. Но по второму (третьему и т.д.) полю отдельно такой эффективный поиск не работает.

Отсюда следует очень простой вывод: если при работе с вашей базой данных часто будут выполняться запросы, в которых поиск происходит по отдельному полю составного ключа, это поле должно быть первым в составе такого ключа.

## Глава 2. Связи

1. Что такое связь (relationship):

* ассоциация, объединяющая несколько сущностей,
* указание того факта, что отношения находятся в некоей взаимосвязи друг с другом,
* оба утверждения верны.

Связь (relationship) — ассоциация, объединяющая несколько сущностей. Упрощённо: указание того факта, что отношения находятся в некоей взаимосвязи друг с другом.

1. Что такое связь один-ко-многим (one-to-many correspondence)?

Связь один ко многим (one to many correspondence) — ассоциация, объединяющая два отношения таким образом, что одному кортежу родительского отношения (или даже нулю таких кортежей) может соответствовать произвольное количество кортежей дочернего отношения.

Упрощённо: одной записи в таблице A может соответствовать много записей в таблице B; при этом может быть ситуация, когда неким записям в таблице B нет соответствия в таблице A.

1. При реализации связи один-ко-многим (one-to-many) возможна ли такая ситуация, когда одной записи в таблице А не соответствует ни одна запись из таблицы В, и одновременно для одной записи из таблицы В нет ни одного соответствия в таблице А.

Неидентифицирующая связь (non-identifying relationship) — ассоциация, объединяющая два отношения таким образом, что кортежу дочернего отношения может не соответствовать ни одного кортежа родительского отношения, а сам кортеж дочернего отношения может быть полностью определён без использования значения первичного ключа соответствующего кортежа родительского отношения.

Упрощённо: запись в дочерней таблице может существовать без соответствующей записи в родительской таблице; в случае последовательной связи в любом дочернем отношении хранятся данные только об одном (ближайшем) родительском отношении.

1. Есть таблица Students, в которой хранится информация о каждом студенте (его ФИО, дата рождения и т.п.). И есть таблица посещений Visits, в которой хранится время входа и выхода студента из ВУЗа. Какой вид связи нужно выбрать в данном случае для соединения этих таблиц:

* один-ко-многим,
* многие-ко-многим,
* один-к-одному.

1. Что такое связь многие-ко-многим (many-to-many correspondence)?

Связь многие ко многим (many to many correspondence) — ассоциация, объединяющая два отношения таким образом, что одному кортежу любого из объединённых отношений может соответствовать произвольное количество кортежей второго отношения.

Упрощённо: одной записи в таблице A может соответствовать много записей в таблице B, и в то же время одной записи в таблице B может соответствовать много записей в таблице A.

1. Есть таблица Students, в которой хранится информация о каждом студенте (его ФИО, дата рождения и т.п.). И есть таблица предметов Subjects, в которой хранится список всех предметов, изучаемых в ВУЗе. Какой вид связи нужно выбрать в данном случае для хранения информации о том, какой студент какой предмет изучает:

* многие-ко-многим,
* один-ко-многим,
* один-к-одному.

В таком случае создаётся т.н. «таблица связи» (associacion relation), которая является дочерней по отношению к обеим связываемым таблицам. И именно в эту таблицу связей мигрируют первичные ключи связываемых таблиц.

На физическом же уровне мы получаем, фактически, две отдельных связи «один ко многим», через которые и формируется нужная нам связь «многие ко многим».

1. Есть таблица Students, в которой хранится информация о каждом студенте (его ФИО, дата рождения и т.п.). И есть таблица предметов Subjects, в которой хранится список всех предметов, изучаемых в ВУЗе. Нам необходимо хранить оценки, которые каждый студент получил за изучаемый им предмет. В какую таблицу надо добавить поле Mark для хранения этой оценки:

* в третью таблицу, которую надо создать для связи таблиц Students и Subjects,
* в таблицу Students,
* в таблицу Subjects.

1. Что такое связь один-к-одному (one-to-one correspondence)?

Связь один к одному (one to one correspondence) — ассоциация, объединяющая два отношения таким образом, что одному кортежу родительского отношения может соответствовать не более одного кортежа дочернего отношения. Упрощённо: одной записи в таблице A может соответствовать не более одной записи в таблице B.

Связи «один к одному» используются, если:

• в предметной области действительно есть сущности разных типов, объединённые именно такой связью (например, «водительские права» и «талон фиксации нарушений ПДД»; хотя даже в этой ситуации очень велика вероятность помещения всей информации в одно отношение);

• при описании некоторой сущности с огромным количеством свойств мы достигли ограничения СУБД на количество полей в одной таблице или на максимальный размер записи — тогда можно «продолжить» в следующей таблице, объединив её с предыдущей этой связью (но сразу возникает сомнение в том, что модель построена правильно, т.к. в реальности такая ситуация крайне маловероятна);

• в целях оптимизации производительности мы хотим разделить на две от-дельных таблицы те данные, к которым доступ нужен очень часто, и те, к которым доступ нужен очень редко, что позволит СУБД обрабатывать при частых операциях меньшие объёмы данных (и, опять же, существуют другие способы оптимизации производительности для таких ситуаций);

• в описываемой предметной области есть много разнотипных сущностей, у каждой из которых, тем не менее, есть одинаковый набор совпадающих свойств (пожалуй, это — единственный случай, в котором без связи «один к одному» не обойтись).

1. Что такое ссылочная целостность (referential integrity)?

Ссылочная целостность (referential integrity) — свойство реляционной базы данных, состоящее в неукоснительном соблюдении правила: если внешний ключ дочернего отношения содержит некоторое значение, это значение обязательно должно присутствовать в первичном ключе родительского отношения.

Упрощённо: запись в дочерней таблице не может ссылаться на несуществующую запись родительской таблицы.

1. Что такое консистентность базы данных (database consistency)?

Консистентность базы данных (database consistency) — свойство реляционной базы данных, состоящее в неукоснительном соблюдении в любой момент времени всех ограничений, заданных неявно реляционной моделью или явно конкретной схемой базы данных.

Упрощённо: СУБД в любой момент времени контролирует типы данных, ссылочную целостность, ограничения типа CHECK, корректность выполнения триггеров, корректность (валидность) запросов и т.д. и т.п.

Критически важно понимать, что СУБД не обладает «телепатией» и не может «догадаться» о том, какие значения надо сравнивать, если связи между таблицами не проведены явно.

Иногда в целях повышения совместимости с различными СУБД или из иных соображений связи между таблицами могут отсутствовать (хотя и нужны), а соответствующие проверки возлагаются на приложение, работающее с базой данных. Несмотря на то, что такое решение в некоторых редких случаях оправдано, оно является крайне рискованным.

1. Есть две таблицы Table1 и Table2 из некоторой предметной области (код их создания см. ниже):

CREATE TABLE Table1 (

Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,

Name NVARCHAR(200) NOT NULL,

Description NVARCHAR(2000) NULL

);

CREATE TABLE Table2 (

Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,

Table1Id INT NULL FOREIGN KEY REFERENCES Table1 (Id) UNIQUE,

CreateDate DATETIME NOT NULL,

Count INT NOT NULL

);

Какую каскадную операцию необходимо прописать при создании этих таблиц?

* каскадное удаление,
* каскадное обновление,
* установка пустых ключей,
* установка значения по умолчанию,
* запрет каскадной операции,
* любую из вышеперечисленных.

Очень часто можно услышать вопрос следующего вида: «А какую каскадную операцию тут правильно сделать?» (И при этом спрашивающий про-сто рисует схему из двух отношений.) Единственный честный и правильный ответ — любую. Выбор каскадной операции зависит только от предметной области.

Иными словами: СУБД совершенно всё равно, что делать с вашими данными. Она в равной степени готова выполнить любую допустимую операцию. А вот какую именно операцию выполнять — решать вам как создателю базы данных.

Поскольку с каскадными операциями связано очень много непонимания, сразу рассмотрим несколько важных замечаний.

1. Любая каскадная операция активируется только при модификации данных в **родительской** таблице. Никогда никакие изменения данных в дочерней таблице не могут привести к запуску каскадной операции.

2. Любая каскадная операция активируется только при условии, что модификации данных в родительской таблице затрагивает **первичный ключ**. Первичный ключ гарантированно затрагивает только операция удаления записи (т.к. невозможно удалить «часть записи»). Если же выполняется изменение других полей, не входящих в состав первичного ключа, каскадные операции не активируются.

3. Вставка данных (в любую таблицу, не важно — родительскую или дочернюю) и выборка данных (также не важно, откуда) никогда не активируют никаких каскадных операций.

4. Не все СУБД умеют «своими силами» выполнять все виды каскадных операций. Если используемая вами СУБД не поддерживает ту или иную операцию, необходимой функциональности можно добиться с использованием триггеров.

5. Почти все каскадные операции являются взаимоисключающими (кроме каскадного обновления, которое «совместимо» со всеми остальными операциями), а потому на некоторой одной связи не может быть включено несколько разных каскадных операций (например, одновременно **и** каскадное удаление, **и** установка значений по умолчанию).

1. Есть две таблицы Table1 и Table2 из некоторой предметной области (код их создания см. ниже):

CREATE TABLE Table1 (

Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,

Name NVARCHAR(200) NOT NULL,

Description NVARCHAR(2000) NULL

);

CREATE TABLE Table2 (

Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,

Table1Id INT NULL FOREIGN KEY REFERENCES Table1 (Id) UNIQUE,

CreateDate DATETIME NOT NULL,

Count INT NOT NULL

);

Обе таблицы пустые. Какие из следующих операций возможны:

1. INSERT INTO Table1(Name)

VALUES (N'Music');

1. INSERT INTO Table1(Id, Name)

VALUES (25, N'Music');

1. SET IDENTITY\_INSERT Table1 ON;

INSERT INTO Table1(Id, Name)

VALUES (25, N'Music');

1. INSERT INTO Table1(Name, Description)

VALUES (N'Music', N'About music');

1. INSERT INTO Table2(CreateDate, Count)

VALUES ('20210701', 7);

1. INSERT INTO Table2(Table1Id, CreateDate, Count)

VALUES (1, '20210701', 7);

## Глава 3. Индексы

1. Что такое индекс:

* структура БД, используемая для ускорения поиска записей,
* объявление ключа в таблице,
* оба утверждения верны.

Индекс (index) — специальная структура базы данных, используемая для ускорения поиска записей и физического доступа к записям. Упрощённо: механизм, значительно ускоряющий поиск необходимой информации в базе данных.

Однако, наличие индексов ощутимо замедляет операции модификации данных (вставки, удаления, обновления), т.к. при изменении данных СУБД необходимо обновить индексы, приведя их в соответствие с новыми значениями индексированных данных.

1. Выделите те признаки, наличие которых говорит о необходимости создания индекса:

* операции чтения из таблицы выполняются гораздо чаще, чем операции модификации,
* поле или совокупность полей часто фигурируют в запросах в секции WHERE,
* необходимо обеспечить уникальность значения поля,
* все вышеперечисленное.

Есть несколько признаков, говорящих в пользу создания индекса:

• операции чтения из таблицы выполняются гораздо чаще, чем операции модификации;

• поле или совокупность полей часто фигурируют в запросах в секции WHERE;

• исследование показало, что наличие индекса повышает производительность запросов;

• необходимо обеспечить уникальность значения поля (или совокупности полей), не являющегося первичным ключом (в таком случае строится т.н. уникальный индекс);

• поле (или совокупность полей) является внешним ключом — в таком случае индексы могут значительно ускорить выполнение JOIN-запросов.

Поскольку очень часто звучит вопрос о том, как составной индекс организован физически, поясним этот момент на примере сравнения простого и составного индекса на основе сбалансированного дерева. В общем случае для индексирования второго (третьего и т.д.) полей применяется две стратегии:

• построение комбинированных ключей, когда значения дополнительных полей хранятся рядом со значениями основных полей — такой подход технически проще, но может привести к ощутимому увеличению затрат памяти, если в хранимых данных наблюдается ситуация, при которой одному значению первого индексируемого поля соответствует много различных значений последующих полей;

• использование вложенных деревьев (фактически, каждый узел основного дерева является корнем дополнительного вложенного дерева) — такой подход технически сложнее, но позволяет сэкономить память в случае, когда одному значению первого индексируемого поля соответствует много различных значений последующих полей.

В MS SQL Server включение свойства уникальности значения поля приводит к созданию на этом поле уникального индекса (и наоборот — создание на поле уникального индекса приводит к включению свойства уникальности значений этого поля).

1. Что такое кластерный индекс (clustered index):

* индекс, построенный на поле, по которому произведено физическое упорядочивание данных в файле (значения поля могут быть неуникальными),
* индекс, построенный на основе сбалансированного дерева,
* оба утверждения верны.

1. Возможно ли построение кластерного индекса на поле с неуникальными значениями?

Кластерный индекс (clustered index) — индекс, построенный на поле (возможно, с неуникальными значениями), по которому произведено физическое упорядочивание данных в файле.

Упрощённо: по значению индексированного поля данные таблицы физически отсортированы на диске; значения поля могут повторяться.

1. Что такое первичный индекс (primary index):

* индекс, построенный на поле с уникальными значениями, по которому произведено физическое упорядочивание данных в файле,
* индекс, построенный на основе сбалансированного дерева,
* оба утверждения верны.

Первичный индекс (primary index) — индекс, построенный на поле с уникальными значениями, по которому произведено физическое упорядочивание данных в файле.

Упрощённо: по значению индексированного поля данные таблицы физически отсортированы на диске; значения поля не могут повторяться.

1. Что такое некластерный индекс (non-clustered index):

* индекс, построенный на поле, по которому не произведено физическое упорядочивание данных в файле (значения поля могут быть неуникальными),
* индекс, построенный на основе сбалансированного дерева,
* оба утверждения верны.

Некластерный индекс (non-clustered index) — индекс, построенный на поле, по которому не произведено физическое упорядочивание данных в файле.

Упрощённо: «просто индекс», принципы упорядочивания в котором никак не связаны с физическим расположением данных на диске.

1. Как связаны между собой «первичный ключ» и «первичный индекс»?

В теории «первичный ключ» относится к связям, ссылочной целостности и нормализации — т.е. теоретическим основам реляционных баз данных, а «первичный индекс» относится к способу организации данных на диске и методам доступа — т.е. к физическому уровню проектирования и способам реализации конкретной СУБД.

MS SQL Server позволяет построить «кластерный уникальный» (фактически — первичный) индекс на поле (полях), не являющемся первичным ключом.

1. В чем отличие «первичного индекса» и «кластерного индекса»?

Ключевое отличие состоит в том, что первичный индекс содержит данные о расположении каждой записи с **уникальным значением индексируемого поля**, а кластерный — о начале блока записей с **одинаковыми значениями индексируемого поля**. Если не вдаваться в технические особенности реализации алгоритмов построения и использования таких индексов — в остальном они идентичны.

1. В чем отличие «кластерного индекса» и «некластерного индекса»?

Некластерные же индексы отличаются от кластерных тем, что последовательность записей в индексе и физическом файле не совпадает. Как правило, это приводит к усложнению структуры самого индекса из-за необходимости хранения нескольких разных адресов записей с совпадающими значениями индексируемого поля.

1. Что такое разделённый индекс (partitioned index) и неразделённый индекс (non-partitioned index)?

Разделённый индекс (partitioned index) — индекс, хранимый и обрабатываемый в виде отдельных частей (разделов, фрагментов) с целью повышения производительности.

Упрощённо: индекс, разделённый на несколько фрагментов.

Неразделённый индекс (non-partitioned index) — индекс, хранимый и обрабатываемый как единая структура данных.

Упрощённо: «просто индекс» (без применения специальных команд по умолчанию все индексы создаются как неразделённые).

Идея разделения индексов очень близка к идее разделения таблиц — способа хранения данных таблицы в виде нескольких частей (что позволяет размещать их на отдельных физических накопителях и обрабатывать параллельно).

1. Что такое плотный индекс (dense index) и неплотный индекс (sparse index)?

По степени детализации индексы бывают плотными и неплотными: Плотный индекс (dense index) — индекс, содержащий указатель на расположение записи для каждого значения индексируемого поля.

Упрощённо: в индексе хранятся адреса каждой записи таблицы.

Неплотный индекс (sparse index) — индекс, содержащий указатель на расположение блока записей для каждого значения (в случае их неуникальности) или группы значений (в случае их уникальности) индексируемого поля.

Упрощённо: в индексе хранятся адреса блоков (групп) записей.

Проясним ситуацию с плотными и неплотными первичными индексами. По определению такие индексы строятся на полях таблицы, содержащих уникальные значения. Но сам индекс может содержать указатели как на каждую отдельную запись, так и на блок записей (в таком случае конкретные значения индексируемого поля будут лежать в диапазоне «от текущего значения в индексе включительно, до следующего значения в индексе»).

К преимуществам неплотных индексов относится их меньший размер и возможность более редкого обновления. Но это преимущество меркнет на фоне недостатка, состоящего в необходимости обращения к таблице для поиска конкретной записи, значение индексированного поля которой попало в тот или иной блок. Потому на практике разработчики СУБД предпочитают использовать плотные первичные индексы.

По соотношению с SQL-запросом индексы бывают покрывающими и непокрывающими — и данное разделение во всей рассматриваемой нами классификации является единственным условным, т.е. один и тот же индекс может быть как покрывающим, так и не покрывающим, потому что здесь есть зависимость не только от самого индекса, но и от SQL-запроса.

Покрывающий индекс (covering index) — индекс, содержащий в явном виде внутри себя информацию, достаточную для выполнения SQL-запроса без обращения к данным, хранящимся вне этого индекса (в самой таблице).

Упрощённо: индекс, информации в котором достаточно для выполнения SQL-запроса без обращения к данным в таблице.

Непокрывающий индекс (non-covering index) — индекс, позволяющий лишь ускорить нахождение нужной информации, в то время как сама искомая информация внутри индекса не содержится или содержится не полностью.

Упрощённо: «просто индекс», информации в котором недостаточно для выполнения SQL-запроса без обращения к данным в таблице.

1. На основе каких структур могут быть построены индексы:

* основе B-дерева (сбалансированного дерева),
* на основе T-дерева,
* на основе R-дерева,
* на основе AVL дерева,
* на основе красно-черного дерева,
* на основе хэш-таблицы.

По базовой структуре индексы могут быть представлены очень большим количеством разновидностей, основными из которых являются:

Индекс на основе B-дерева (сбалансированного дерева) (B-tree index) — индекс, структурно организованный с использованием B-дерева (сбалансированного дерева), оптимизированный для выполнения поиска на основе диапазонов и для операций с большими блоками данных. Допускает хранение части индекса во внешней памяти. Упрощённо: одна из основных форм организации индексов в большинстве СУБД и методов доступа. (Буква «B» в названии индекса идёт от слова «balanced».)

Индекс на основе T-дерева (T-tree index) — индекс, структурно организованный с использованием T-дерева (разновидности сбалансированного дерева, в котором вместо самих данных хранятся указатели на адреса данных в памяти), оптимизированный для выполнения операций в случае, когда и индекс и данные целиком находятся в оперативной памяти. Упрощённо: индекс для СУБД и методов доступа, предполагающих хранение всего объёма обрабатываемых данных в оперативной памяти. (Буква «T» в названии индекса идёт от графической формы представления вершин T-дерева в статье, в которой оно было впервые представлено.)

Индекс на основе R-дерева (R-tree index) — индекс, структурно организованный с использованием R-дерева (специальной формы представления географических и геометрических данных), оптимизированный для выполнения операций со специфическими типами данных, хранящих географические координаты или информацию о геометрических фигурах. Упрощённо: индекс для ускорения обработки географических и геометрических данных (Буква «R» в названии индекса идёт от слова «rectangle».).

Индекс на основе хэш-таблицы (hash-table index) — индекс, структурно организованный с использованием хэш-таблицы (специальной структуры для хранения пар ключ-значение), оптимизированный для выполнения поиска на основе строгого сравнения и обработки относительно редко изменяемых данных. Упрощённо: одна из основных форм организации индексов в большинстве СУБД и методов доступа (наряду с индексами на основе B-деревьев).

1. Что такое индекс на вычисляемых столбцах (index on computed columns)?

Индекс на вычисляемых столбцах (index on computed columns) — индекс, построенный на значениях виртуальных столбцов, данные которых могут не храниться физически в базе данных. Упрощённо: индекс на столбце, значение которого вычисляет сама СУБД.

Задача, решаемая с использованием этих индексов, может быть сформулирована как ускорение доступа к данным, значения которых вычисляются на уровне СУБД, а не передаются и сохраняются в базу данных явно (как это происходит в подавляющем большинстве случаев). На таких данных невозможно (или крайне сложно и неэффективно) строить «классические» индексы, именно поэтому некоторые СУБД предлагают рассматриваемое здесь решение.

Пояснение: допустим, что (для MS SQL Server) в каких-то операциях нам необходимо очень быстро искать людей по их инициалам (для этого создаётся вычисляемый столбец u\_initials), нам необходимо выполнять поиск по полному ФИО в верхнем регистре, где данные идут в порядке имя, отчество, фамилия (для этого создаётся функция, значения которой индексируются).

Технически такие индексы могут реализовываться почти любым из ранее рассмотренных способов, но чаще всего это будут плотные некластерные индексы на основе B-деревьев.

1. Что такое полнотекстовый индекс (full text index)?

Полнотекстовый индекс (full text index) — индекс, оптимизированный для ускорения операций поиска вхождений подстрок в значения текстовых полей. Упрощённо: индекс для поиска текста в тексте.

Все «классические» индексы направлены на строгое сравнение искомых и хранимых значений или на проверку вложенности и пересечения областей. Все эти подходы оказываются бесполезными при решении одной специфической, но очень востребованной задачи — поиска текста в тексте.

Несмотря на всю красоту и мощь полнотекстовых индексов, стоит понимать, что они обладают широким спектром ограничений, а их использование требует точного понимания того, что вы делаете (для осознания масштаба сложности рекомендуется ознакомиться с документацией по MS SQL Server, где полнотекстовым индексам посвящено 28 разделов). Иными словами: чудес ожидать не стоит, но некоторые запросы можно ощутимо ускорить.

1. Какие операции можно осуществить с индексами:

* создать/удалить индекс,
* отключить/включить существующий индекс,
* проанализировать использование индекса при выполнении запроса,
* дать СУБД подсказку по использованию индекса,
* все вышеперечисленное.

С индексами на уровне базы данных и выполнения запросов к ней можно осуществить следующие основные операции:

• создать (удалить) индекс;

• отключить (включить) существующий индекс;

• проанализировать использование индекса при выполнении запроса;

• дать СУБД подсказку по использованию индекса.

1. Какие из нижеследующих инструкций правильно описывают создание и удаление индекса:
2. CREATE INDEX Idx\_Persons\_Name

ON Persons (Name);

DROP INDEX Idx\_Persons\_Name

ON Persons;

1. CREATE INDEX Idx\_Persons\_Name

ON Persons (Name);

DELETE INDEX Idx\_Persons\_Name

ON Persons;

Обратите внимание: в MS SQL Server имена индексов локальны в рамках таблицы (да, можно создать на разных таблицах индексы с одинаковыми именами, но делать этого не стоит, т.к. такой подход усложняет дальнейшую работы с базой данных).

1. Если вы по каким-либо причинам перевели индекс в отключенное состояние (disabled), а затем вновь включили его, то какую операции при этом выполнит БД:

* rebuild,
* enable,
* никакую не выполнит.

По умолчанию индекс после создания сразу находится во включённом состоянии (enabled), однако в целях исследования производительности (или по иным причинам) может возникнуть потребность временно перевести индекс в отключённое состояние (disabled).

Отключённый индекс автоматически становится устаревшим (информация в нём не соответствует реальному набору данных в таблице), и после включения требует перестроения (rebuild), т.е. достаточно трудоёмкой для СУБД операции.

1. Каким образом можно проверить эффективно ли использование индекса при выполнении запроса:

* проанализировать план выполнения запроса,
* отключить/включить индекс,
* никак нельзя проверить.

Создавая индексы, мы предполагаем, что с их помощью СУБД сможет более эффективно выполнять те или иные запросы. Но всегда стоит проверить, оправдались ли наши надежды.

Одним из самых простых и в то же время рациональных способов такой про-верки является анализ плана выполнения запроса, в котором СУБД предоставит нам информацию в т.ч. и об используемых индексах.

Если в MS SQL Server Management Studio выполнить запрос комбинацией клавиш Ctrl+L, мы не только получим красивое визуальное представление плана выполнения запроса, но и дополнительную статистическую информацию по реально выполненным СУБД операциям, а также ещё и подсказки от СУБД относительно того, какие индексы могли бы ускорить выполнение запроса.

Вернёмся к вопросу о том, почему на небольших объёмах данных создание индексов может оказывать гораздо меньший положительный (а иногда и вовсе отрицательный) эффект на скорость выполнения запросов.

Если в двух словах пересказать несколько сот страниц документации и заме-ток с AskTOM: на малых объёмах данных слишком большое влияние на результаты эксперимента оказывают как внешние факторы (работа операционной системы, сторонних приложений, аппаратного обеспечения, сети и т.д.), так и внутренние (не отражаемые в плане запроса) особенности поведения СУБД.

Если многократно повторять этот эксперимент на небольшом объёме данных, можно получить кратковременное увеличение медианного значения времени выполнения почти всех запросов. Однако, тенденция к его снижению явно просматривается по мере того, как данных в таблице становится всё больше и больше.

Крайне важный вывод: тестовые данные, на которых вы производите экспериментальную оценку производительности СУБД при выполнении тех или иных операций в тех или иных условиях, должны быть максимально приближены к реальным. Т.к. от набора данных зависит не только реалистичность результатов, но и даже само поведение СУБД (использованные ею алгоритмы и последовательность их применения), а потому решение, эффективное в одной ситуации, может оказаться губительным в другой.

1. Как запретить использование индекса (стр. 157):

* WITH,
* FORCE,
* никак нельзя проверить.

Ранее мы уже говорили о том, что отключение и повторное включение индексов в повседневной работе — плохая идея, вместо которой можно использовать специальные решения и/или подсказки (hints), с помощью которых мы можем повлиять на использование СУБД индексов.

В MS SQL Server подсказки по использованию индексов не являются обособленной частью синтаксиса и размыты по общей структуре подсказок по выполнению запросов.

Применяйте подсказки по использованию индексов (index hints) равно как и подсказки по выполнению запросов (query hints) лишь в том случае, если вы совершенно точно уверены в том, что делаете (или для проведения исследований). Скорее всего, СУБД без вашей помощи построит более оптимальный план выполнения запроса.

1. Какие действия можно предпринять, чтобы увеличить скорость обработки данных при их модификации в БД:

* проверить отсутствие дублирующихся индексов (по первому полю составного индекса СУБД умеет производить поиск столь же быстро, как и по всему составному индексу),
* выполнять несколько операций модификации в рамках одной транзакции,
* убедитесь, что вы оптимально выбрали кластерный индекс,
* изучить план выполнения запроса.

Пояснение: как бороться с проблемой снижения производительности модификации данных при наличии большого количества индексов. Чаще всего этот вопрос возникает в контексте операции вставки, хоть многие решения будут полезны и при выполнении операций обновления и удаления данных.

Удостоверьтесь, что на вашей таблице нет дублирующихся индексов (помните, что по первому полю составного индекса СУБД умеет производить поиск столь же быстро, как и по всему составному индексу — потому нет необходимости создавать отдельный индекс на этом поле).

Выполняйте несколько операций модификации в рамках одной транзакции. В большинстве случаев СУБД не будет обновлять индексы до завершения транзакции.

Убедитесь, что вы оптимально выбрали кластерный индекс модифицируемой таблицы и выполняйте операции модификации так, чтобы значения со-ответствующего поля в модифицируемых данных шли в той же последовательности, что и значения этого же поля в кластерном индексе.

Изучите план выполнения запроса и соответствующие рекомендации по вашей СУБД.

## Глава 4. Нормализация

1. Существует ли функциональная зависимость между {Номер паспорта} → {ФИО}?

Зависимости, на которых базируется вторая нормальная форма: при наличии функциональной зависимости между множествами атрибутов X и Y существует правило, что если в двух строках таблицы совпадают значения X, то обязаны совпадать и значения Y. Из определения функциональной зависимости следует, что одному значению множества X должно соответствовать строго одно значение множества Y. Для очень быстрого понимания сути функциональной зависимости можно вспомнить школьный курс математики, где часто звучало выражение «Y — это функция от X». Действительно, если «множество X» является потенциальным ключом таблицы, этот рисунок в полной мере и без никаких допущений графически иллюстрирует понятие функциональной зависимости.

До тех пор, пока мы не указали ограничения предметной области, мы не можем гарантировать наличие или отсутствие функциональной зависимости. Если в рамках предметной области сказано, что одному номеру паспорта со-ответствует строго одно ФИО, функциональная зависимость {Номер паспорта} → {ФИО} присутствует. Если же сказано, что одному номеру паспорта может соответствовать несколько ФИО (например, на разных языках или с учётом девичей фамилии), функциональной зависимости здесь нет.

1. В рамках одной БД есть таблица Orders, в которой хранится информация о заказах, и таблица Customers, в которой хранится информация о покупателях. Нам необходимо хранить информацию о сумме заказа. В какой из таблиц ее лучше всего разместить?

* в таблице Orders: там хранится вся информация о заказе,
* в таблице Customers: так будет проще выводить информацию о заказах в разрезе покупателей,
* в обеих таблицах: так мы снизим вероятность потери данных о сумме заказа.

Требование неизбыточности данных: аномалии обновления и удаления процветают в случае, когда одни и те же данные хранятся в нескольких строках одной таблицы. Ситуация оказывается весьма схожей, когда одни и те же данные хранятся в разных таблицах.

Иногда можно услышать возражение, что, сохраняя данные в нескольких местах, мы повышаем шансы не потерять их в случае сбоев и повреждения хранилища. Это в корне неверный подход: ни в коем случае не стоит путать безусловно важную задачу создания резервных копий, зеркалирования и иных мер по обеспечению сохранности данных и задачу проектирования базы данных. Нельзя инструментами решения одной из этих задач решать другую.

Стр. 215 – создание индексов.

Стр. 224 – ФИО это атомарное поле или надо разбить на 3.

Стр. 243 – Адрес это атомарное поле?

Атомарность атрибута – СУБД не должна оперировать никакой отдельной частью атрибута.

Если у person несколько телефонов, то можно их записать в одно поле через запятую?

Стр. 225 – 2НФ

Стр. 229 – 3НФ

Переменная отношения находится во второй нормальной форме (2НФ, 2NF) тогда и только тогда, когда она находится в первой нормальной форме, и каждый её непервичный атрибут функционально полно зависит от первичного ключа.

Упрощённо: ни один атрибут, не входящий в состав первичного ключа, не должен функционально зависеть от части первичного ключа.

Из такой формулировки определения 2НФ следует, что если первичный ключ отношения является простым, и отношение находится в 1НФ, то оно автоматически находится и во 2НФ. Казалось бы, всё очень хорошо и просто (достаточно отношению, находящемуся в 1НФ, добавить простой искусственный первичный ключ), но существует более полное определение 2НФ.

Переменная отношение находится во второй нормальной форме (2НФ, 2NF) тогда и только тогда, когда она находится в первой нормальной форме, и каждый её неключевой атрибут функционально полно зависит от любого потенциального ключа.

Упрощённо: ни один атрибут, не входящий в состав потенциального ключа, не должен функционально зависеть от части какого бы то ни было из потенциальных ключей.

Стр. 279 – формулировки всех нормальных форм.

## Глава 5. Проектирование БД

Инфологический уровень проектирования – изучение предметной области с точки зрения сбора информации о ней. Здесь определяются границы области исследования: что будет включено в предметную область, а что – нет. Например, проект «Библиотека» будет включать список книг в ней, но не будет включать то, как они туда попали (логистику).

Здесь используются ER-диаграммы (entity-relation, сущность-связь), графовые модели или UML-диаграммы.

Даталогический уровень проектирования – здесь формируется будущая структура базы данных, вносятся изменения в инфологическую модель. Определяется список таблиц и связей между ними, поля и их типы, проводится нормализация таблиц, если это необходимо.

1. Что из перечисленного формируется на физическом уровне проектирования БД:

* права и роли,
* индексы,
* кодировки,
* методы доступа (к файлам БД),
* все вышеперечисленное.

Физический уровень проектирования – здесь определяются типы пользователей и их права, индексы, кодировки, методы доступа (к файлам БД).

Для создания эффективных индексов необходимо:

* наполнить базу данных тестовыми данными, максимально приближенными к реальным (как по объёму, так и по содержанию);
* на основе уже имеющейся информации о типичных запросах провести нагрузочное тестирование;
* определить те узкие места в производительности базы данных, которые можно устранить с использованием индексов;
* создать необходимые индексы;
* сделать пометку на будущее о необходимости периодической проверки эффективности созданных индексов как в процессе разработки базы данных и работающих с ней приложений, так и в процессе реальной работы созданного проекта.

Поскольку создание и удаление индексов (за исключением уникальных и первичных) не влияет на структуру и логику работы базы данных, необходимые правки можно вносить в любой момент времени, но в подавляющем большинстве случаев требуется не только тщательный анализ ситуации, но и изучение планов выполнения запросов и проведение серии экспериментов.

## Глава 5. View, check, trigger, хранимые процедуры и функции…

1. Что такое представление (view):

* SQL-запрос, который можно выполнять, обращаясь к нему по заранее указанному имени,
* объект базы данных, к которому можно обратиться по имени,
* оба утверждения верны.

Представление (view) — виртуальная производная переменная отношения, значением которой является результат вычисления реляционного выражения (выполнения запроса), заданного при создании представления (такое выражение должно ссылаться хотя бы на одну переменную отношения).

Упрощённо: SQL-запрос, который можно выполнять, обращаясь к нему по заранее указанному имени.

При создании представления просто фиксируется тот факт, что теперь некий SQL-запрос можно не писать заново, а вызывать по заранее заданному имени. Только при вызове такого запроса по его имени происходит (каждый раз заново) реальное чтение данных из таблиц, на которые ссылается представление. Результат чтения этих данных нигде не сохраняется.

1. Что такое материализованное представление (materialized view):

* SQL-запрос, который можно выполнять, обращаясь к нему по заранее указанному имени, и результат выполнения которого сохраняется для дальнейшего использования,
* временная таблица, к которой можно обратиться по заранее указанному имени,
* оба утверждения верны.

Материализованное представление (materialized view) — производная переменная отношения, значением которой является сохранённый результат заранее вычисленного реляционного выражения (выполнения запроса), заданного при создании материализованного представления (такое выражение должно ссылаться хотя бы на одну переменную отношения). Повторное вычисление и сохранение полученного результата происходит согласно правилам, определённым при создании материализованного представления.

Упрощённо: SQL-запрос, который можно выполнять, обращаясь к нему по заранее указанному имени, и результат выполнения которого сохраняется для дальнейшего использования.

При создании представления фиксируется тот факт, что теперь некий SQL-запрос можно не писать заново, а вызывать по заранее заданному имени. Результат выполнения запроса фиксируется во временном хранилище. Обновление хранилища происходит по алгоритму, указанному при создании представления. При вызове такого запроса чтение данных происходит из временного хранилища.

1. Для создания представления (view) можно использовать SQL-запросы, в которых содержится:

* только инструкция SELECT,
* только инструкции INSERT, UPDATE, DELETE,
* могут содержаться любые инструкции.

Поскольку представления строятся на основе SQL-запросов, следует уточнить, что:

• «основой» для построения представления может быть только SELECT запрос (т.е. нельзя построить представление на основе запросов с INSERT, UPDATE, DELETE);

• в некоторых случаях представление может допускать «двунаправленность», т.е. с его использованием можно будет не только читать данные, но и модифицировать их (редко используется, так как есть определенные сложности).

1. В рамках БД для автоматизации работы предприятия необходимо создать таблицу Person, в которой надо хранить следующие данные: ФИО, дату рождения, паспортные данные, фотографию, скан документа об образовании. Доступ к ФИО и фотографии должны иметь многие пользователи этой БД, а вот доступ к остальным данным необходимо ограничить только работниками отдела кадров. Как можно это реализовать?

* создать две таблицы, соединив их отношением один-к-одному, и выдавать права на доступ к каждой из этих таблиц,
* создать одну таблицу Person для хранения всей информации о сотруднике, а затем создать представления (view), которые будут запрашивать разный набор данных, и на эти представления выдать права.

Представления являются объектами базы данных, потому к ним применимы все механизмы СУБД по контролю прав доступа. Поскольку представления дают опосредованный доступ к таблицам базы данных, появляется возможность запретить такой доступ напрямую к таблицам, а разрешить его только через специально созданные и продуманные представления. Дополнительный уровень безопасности можно обеспечить за счёт логики запросов, на которых строятся представления.

1. Что такое проверка (check)?

* правило, которое ограничивает все значения некоторого поля таблицы,
* условие, которому должно соответствовать значение поля таблицы,
* оба утверждения верны.

Проверка (check) — правило, ограничивающее все значения некоторого поля таблицы определёнными условиями.

Упрощённо: условие, которому должно соответствовать значение поля таблицы.

Использование проверок является более простым и лёгким способом контроля консистентности базы данных, чем использование триггеров, т.к. проверки (как правило) не только более просты в описании, но и выполняются быстрее, чем триггеры.

И к ещё одному (пусть и небольшому) недостатку проверок можно отнести тот факт, что в случае нарушения условия проверки мы крайне ограничены в возможностях по формированию информативного сообщения об ошибке. В то время как при использовании триггеров мы можем не только формировать такие сообщения, но и (в некоторых СУБД) порождать исключения и иным образом контролировать весь дальнейший процесс выполнения запроса (вплоть до автоматической корректировки ошибочных данных).

1. Что такое триггер (trigger)?

* объект базы данных, описывающий перечень действий, которые необходимо автоматически выполнить при наступлении указанного события,
* процедура, в которую можно вызвать при наступлении определенного события,
* оба утверждения верны.

Триггер (trigger) — специальный объект базы данных, описывающий перечень действий, которые необходимо автоматически выполнить при наступлении указанного события.

Упрощённо: описание действия, которое нужно автоматически выполнить при определённых условиях.

Наиболее классическим вариантом использования триггеров является обеспечение реакции на модификацию данных, т.е. на операции вставки, обновления и удаления.

Поскольку ограничение CHECK может содержать ссылки только на столбцы, для которых определены ограничения на уровне столбцов или таблицы, любые межтабличные ограничения должны быть заданы в виде триггеров.

К недостаткам триггеров стоит отнести их влияние на производительность. И оно тем сильнее, чем более сложные операции выполняются внутри триггеров (а такие операции могут включать обращение к другим таблицам, сложные вычисления и т.д.)

1. Когда может быть выполнен триггер в MS SQL Server?

* до наступления некоторого события (BEFORE Trigger),
* после наступления некоторого события (AFTER Trigger),
* вместо наступления некоторого события (INSTEAD OF Trigger).

Очевидно, что при наступлении некоторого события, СУБД может выполнить триггер до, после или вместо соответствующей операции.

Отдельно стоит подчеркнуть, что триггеры, выполняемые вместо операции (т.н. INSTEAD OF триггеры) на самом деле выполняются вместо соответствующей операции, а потому саму операцию необходимо реализовывать внутри триггера и выполнять в том случае, если пройдены все проверки и соблюдены все необходимые условия. Это особенно актуально для MS SQL Server, в котором отсутствуют BEFORE триггеры, и их логику приходится реализовывать в INSTEAD OF триггерах.



1. Какие данные находятся в специальной таблице DELETED:

* копии строк, которые были удалены инструкцией DELETE,
* копии строк, которые были обновлены инструкцией UPDATE,
* оба утверждения верны.

1. Какие данные находятся в специальной таблице INSERTED:

* копии строк, которые были вставлены инструкцией INSERT,
* копии строк, которые были обновлены инструкцией UPDATE,
* оба утверждения верны.

Теперь рассмотрим, как именно триггеры выполняются и обрабатывают данные. В этом контексте триггеры делятся на две группы:

• триггеры уровня строки (row-level triggers) — запускаются каждый раз заново для каждой отдельной строки (записи таблицы), затронутой SQL-запросом;

• триггеры уровня запроса (statement-level triggers) — запускаются один раз для всего SQL-запроса.

Триггер уровня запроса будет активирован один раз, и ему будет доступна информация обо всех затронутых обновлением строках. Поскольку строк много, здесь можно использовать специальные «виртуальные» таблицы DELETED (содержит данные до обновления) и INSERTED (содержит данные после обновления).  SQL Server автоматически создает эти таблицы и управляет ими. Эти временные таблицы находятся в оперативной памяти.

В таблице DELETED находятся копии строк, с которыми работали инструкции DELETE или UPDATE. При выполнении инструкции DELETE или UPDATE происходит удаление строк из таблицы триггера и их перенос в таблицу DELETED. У таблицы DELETED обычно нет общих строк с таблицей триггера.

В таблице INSERTED находятся копии строк, с которыми работали инструкции INSERT или UPDATE. При выполнении вставки или обновления происходит одновременное добавление строк в таблицу триггера и в таблицу INSERTED. Строки таблицы INSERTED являются копиями новых строк таблицы триггера.

Порядок строк в таблицах DELETED и INSERTED может не совпадать, потому в случае изменения значения первичного ключа мы должны полагаться на значения других уникальных полей (если они есть), чтобы сопоставить строки в их старом и новом состоянии. Если в таблице нет иных уникальных полей, но нам необходимо сопоставить строки в их старом и новом состоянии, то для СУБД, не поддерживающих триггеры уровня ряда, эта задача не имеет решения.

С помощью триггеров в общем случае решается следующий спектр задач:

• организация каскадных операций (если используемый метод доступа таковые не поддерживает) или реализация более сложной логики каскадных операций, чем предоставляет СУБД;

• обновление данных кэширующих (или агрегирующих) полей и таблиц (это — один из самых частых случаев использования триггеров);

• обеспечение консистентности, т.е. контроль и изменение таких значений полей, которые находятся в строгой зависимости от значений других полей (или иных условий);

1. Хранимая процедура (stored procedure) это:

* подпрограмма, хранимая в БД,
* может иметь входные параметры,
* может иметь выходное значение,
* может иметь выходные значения,
* не может иметь выходного значения,
* может быть вызвана из кода других процедур,
* может быть использована в теле SQL-запроса.

Хранимая процедура (stored procedure) — подпрограмма (возможно, параметризованная) предназначенная для выполнения ряда операций с данными и структурами базы данных, хранимая на стороне базы данных и доступная как для вызова из кода других процедур и триггеров, так и для непосредственного исполнения.

Упрощённо: подпрограмма, вызываемая напрямую или из других подпрограмм, и выполняющая некоторые полезные действия.

1. Хранимая процедура (stored procedure) это:

* подпрограмма, хранимая в БД,
* может иметь входные параметры,
* может иметь выходное значение,
* может иметь выходные значения,
* не может иметь выходного значения,
* может быть вызвана из кода других процедур,
* может быть использована в теле SQL-запроса.

Хранимая функция, пользовательская функция (stored function, user-defined function) — подпрограмма (возможно, параметризованная), расширяющая возможности языка SQL и работающая аналогично встроенным в СУБД функциям; обязана возвращать значения.

Упрощённо: функция, расширяющая возможности СУБД и предназначенная для упрощения часто повторяющихся операций.

Итак, первое и самое главное отличие: функция обязана возвращать значение, процедура — не обязана. Второе отличие состоит в том, что функцию можно использовать в любом SQL-запросе, в то время как работа с процедурами имеет свой особый синтаксис и ряд ограничений.

Если свести все отличия к одной фразе, получается, что функции мы используем тогда, когда в SQL-запросе нужно получить некоторое значение (по аналогии со встроенными функциями наподобие SUM(), AVG() и т.д.), а процедуры — когда нужно выполнить ряд сложных действий (обновить данные, создать новую таблицу и т.д.)

**Транзакция (**transaction227) — набор операций с базой данных, который представляет собой неделимую логическую единицу. Такой набор опера-ций может быть выполнен либо целиком и успешно (с соблюдением всех правил консистентности базы данных{71} и независимо от параллельно вы-полняемых транзакций), либо не выполнен вообще (в таком случае ни одна из операций, входящих в данный набор, не должна произвести ника-ких изменений в базе данных).

*Упрощённо: набор операций, который либо целиком и успешно заверша-ется, либо также целиком отменяется в случае ошибки при выполнении любой из его операций.*

приведённом в сноске англоязычном определении227 также сказано, что транзакция является единицей восстановимости и конкурентности.

Итого, получается, что транзакция:

• всегда или выполняется, или не выполняется целиком;

• используется при восстановлении после различных сбоев и отказов;

• обеспечивает механизм конкурентного доступа к данным.

Прежде, чем мы рассмотрим сами уровни изоляции, необходимо пояснить, какие ти-пичные проблемы могут возникать при одновременном доступе к одним и тем же данным нескольких транзакций.

**Потерянное обновление** (lost update) — сохраняются только те изменения данных, которые были выполнены позже всего.

*Очень упрощённый пример: несколько сотрудников в офисе ругаются по поводу настроек кондиционера, т.е. кто-то его включает, кто-то выключает, кто-то делает теплее, кто-то холоднее — не важно, кто и что сделал ранее, кондиционер всегда настроен так, как его настроили «в самый последний мо-мент».*

**Грязное чтение** (dirty read) — становится доступным временное состояние данных, которые в дальнейшем будут удалены или изменены в силу отмены рабо-тавшей с ними транзакции.

*Очень упрощённый пример: ребёнок подслушал, что родители собираются подарить ему на день рождения велосипед, и радостный побежал рассказывать об этом друзьям; через пять минут родители передумали.*

**Неповторяющееся чтение** (non-repeatable read) — происходит изменение одних и тех же данных за время работы транзакции (т.е. при повторном чтении ра-нее прочитанных данных получается новый результат).

*Очень упрощённый пример: вы решили попить чаю; заглянули в шкафчик и посмотрели, какой там есть чай; пока вы грели чайник, кто-то заменил в шкаф-чике имевшийся там чай на другой (или вообще на кофе); вы снова открываете шкафчик и сильно удивляетесь, т.к. только что видели там другую картину.*

**Фантомное чтение** (phantom reads) — происходит изменение количества строк, подпадающих под выборку (в силу добавления или удаления строк или из-менения значений в их полях).

*Очень упрощённый пример: вы хотите сфотографировать трёх сидящих на ветке воробьёв; пока вы отвлеклись на настройки фотоаппарата, прилетело ещё два воробья.*

Ключевое отличие неповторяющегося и фантомного чтения состоит в при-роде их возникновения: при неповторяющемся чтении меняются сами данные, а при повторном чтении меняется *количество* данных (т.е. добавляются или удаля-ются строки).

Теперь рассмотрим сами уровни изолированности. Когда транзакция выпол-няется с неким уровнем изолированности, СУБД обеспечивает такой транзакции «защиту» от тех или иных описанных выше проблем. Уровни изолированности представляют собой иерархию, где каждый следующий (более высокий) уровень включает в себя все «защитные механизмы» предыдущих (более низких) уровней.

**Чтение неподтверждённых данных** (read uncommitted) — допускает чтение незафиксированных (т.е. до подтверждения или отмены транзакции) изменений — выполненных любой транзакцией (как той, что произво-дит чтение, так и выполняющихся параллельно с ней).

Этот уровень обеспечивает отсутствие потерянных обновлений, т.е. если не-сколько параллельных транзакций изменяют одни и те же данные, в итоге эти дан-ные будут иметь значение, полученное последовательным применением всех вне-сённых изменений. Все остальные проблемы (грязное чтение, неповторяющееся чтение, фантомное чтение) продолжают оставаться актуальными.

Физически для защиты от потерянного обновления применяется блокировка изменяемых данных, т.е. изменения одних и тех же данных, выполняемые парал-лельно, на самом деле выполняются последовательно (выстраиваются в очередь).

Никакие операции чтения на данном уровне изоляции не блокируются.

**Чтение подтверждённых данных (**read committed) — допускает чтение всех изменений, выполненных самой транзакцией, и только подтверждённых (зафикси-рованных) изменений, выполненных другими (параллельными) транзакциями.

Этот уровень обеспечивает отсутствие потерянных обновлений и грязного чтения, при этом допускаются неповторяющееся чтение и фантомное чтение.

Физически данный уровень реализуется с использованием блокировки или версионирования данных:

• при блокировке транзакция, изменяющая данные, блокирует чтение этих данных для параллельных транзакций, выполняемых на уровне read committed и более высоких;

• при версионировании СУБД создаёт новую версию (копию) изменяемых дан-ных для той транзакции, которая эти данные изменяет, а всем остальным (параллельным) транзакциям предоставляет доступ к старой (неизменённой) версии.

Оба варианта имеют множество преимуществ и недостатков, а также особен-ностей реализации, потому подробности можно найти только в документации по конкретной версии вашей конкретной СУБД.

**Повторяющееся чтение (**repeatable read) — допускает только чтение изме-нений, выполненных самой транзакцией, а прочитанные ею данные становятся не-доступными для изменения параллельным транзакциям.

Этот уровень обеспечивает отсутствие потерянных обновлений, грязного чтения и неповторяющегося чтения, но допускает фантомное чтение. Общие сведения о транзакциях

Физически данный уровень реализуется через блокировку прочитанных дан-ных, что запрещает параллельным транзакциям изменять соответствующие строки таблиц. Но параллельные транзакции могут вставлять новые строки, что может по-рождать проблему фантомного чтения.

**Снимок (**snapshot) — является частным (более высоким) случаем повторяю-щегося чтения, поддерживается не всеми СУБД, и допускает только чтение изме-нений, выполненных самой транзакцией, а *прочитанные ею данные остаются до-ступны для изменения параллельным транзакциям* (в этом состоит основное от-личие от уровня повторяющегося чтения).

**Сериализация (**serializable) — допускает только такое выполнение изменений данных, словно все модифицирующие данные транзакции выполняются не параллельно, а последовательно.

Этот уровень обеспечивает отсутствие всех проблем, т.е. потерянных обновлений, грязного чтения, неповторяющегося чтения и фантомного чтения.

Физически это достигается за счёт управления очередью транзакций и сложной системы блокировок. Это самый надёжный в плане точности работы с данными уровень изолированности транзакций, но он же — и самый медленный с точки зрения производительности.

Однако, можно пойти ещё дальше. Ведь как выполняется «просто запрос» (если мы не обрамляем его командами **BEGIN TRANSACTION** / **COMMIT**)?

На самом деле — всё происходит точно так же, потому что СУБД реализует т.н. «неявные транзакции» (implicit transactions): фактически, команды **BEGIN TRANSACTION** / **COMMIT** выполняются без нашего участия.

Неявные транзакции удобны потому, что очень часто желаемый эффект достигается выполнением всего лишь одного SQL-запроса, и нет необходимости объединять в транзакцию несколько действий: потому логично, что такая «микротранзакция», состоящая из одного запроса, автоматически фиксируется (в случае успешного выполнения запроса) или автоматически отменяется (в случае ошибки выполнения запроса).

Неявные транзакции выполняются на «уровне изолированности по умолчанию» (у каждой СУБД он свой, настраивается в конфигурации СУБД и может быть определён соответствующими командами).