

## 3. Запросы CRUD в однотабличную базу данных

### Создание таблицы CREATE

Давайте создадим простую таблицу SQLite для учебных целей, шаг за шагом. Таблица будет называться `Students` и предназначена для хранения информации о студентах.

```
-- Создаем новую таблицу с именем Students
CREATE TABLE Students (
  -- Определяем столбец ID как целое число
  -- Это будет первичный ключ таблицы, что означает, что каждое значение в этом столбце уникально
  -- AUTOINCREMENT указывает, что значение в этом столбце будет автоматически увеличиваться
  ID INTEGER PRIMARY KEY AUTOINCREMENT,

  -- Столбец для хранения имени студента
  -- Тип данных TEXT используется для хранения текстовой строки
  Name TEXT NOT NULL,

  -- Столбец для хранения возраста студента
  -- Тип данных INTEGER используется для хранения целых чисел
  Age INTEGER NOT NULL,

  -- Столбец для хранения специальности студента
  -- Также используется тип TEXT для текстовых данных
  Major TEXT,

  -- Столбец для хранения среднего балла студента
  -- REAL используется для хранения чисел с плавающей точкой
  GPA REAL
);
```

Объяснение каждой части запроса:

- `CREATE TABLE Students`: Эта команда создает новую таблицу с именем `Students`.
- `ID INTEGER PRIMARY KEY AUTOINCREMENT`: Это определение столбца `ID`. `INTEGER` означает, что столбец будет хранить целые числа. `PRIMARY KEY` указывает, что каждое значение в этом столбце уникально для каждой строки таблицы, и это поле используется для идентификации строк. `AUTOINCREMENT` означает, что SQLite будет автоматически увеличивать значение этого поля для каждой новой строки, обеспечивая уникальность.
- `Name TEXT NOT NULL`: Определяет столбец `Name`, который будет хранить имя студента. `TEXT` означает, что данные в этом столбце будут текстовыми. `NOT NULL` указывает, что это поле не может быть пустым, то есть для каждого студента обязательно должно быть указано имя.
- `Age INTEGER NOT NULL`: Столбец `Age` предназначен для хранения возраста студента. Тип данных `INTEGER` используется для хранения целых чисел. Аналогично `Name`, `NOT NULL` гарантирует, что для каждого студента будет указан возраст.
- `Major TEXT`: Столбец `Major` предназначен для хранения специальности студента. Поскольку специальность — это текстовая информация, используется тип данных `TEXT`. В отличие от `Name` и `Age`, здесь отсутствует ограничение `NOT NULL`, что означает, что для некоторых студентов специальность может не быть указана.
- `GPA REAL`: Столбец `GPA` (Grade Point Average — средний балл) предназначен для хранения среднего балла студента. Тип данных `REAL` используется для хранения чисел с плавающей точкой, что подходит для представления таких значений, как средний балл.

В целом, эта таблица `Students` предназначена для хранения информации о студентах, включая их ID, имя, возраст, специальность и средний балл. Использование разных типов данных и ограничений (`NOT NULL`) обеспечивает целостность и актуальность хранимых данных.

### Первичные ключи (Primary Keys)

Первичный ключ в базе данных — это уникальный идентификатор для каждой строки в таблице. Он гарантирует уникальность записей и используется для установления связей между таблицами.

## Особенности Первичных Ключей:

1. **Уникальность:** Каждое значение первичного ключа должно быть уникальным. Никакие две строки в таблице не могут иметь одинаковое значение первичного ключа.
2. **Неизменяемость:** Значение первичного ключа, как правило, не изменяется после создания строки.
3. **Индексация:** Первичные ключи автоматически индексируются, что обеспечивает быстрый поиск и доступ к данным.
4. **Связи между таблицами:** В реляционных базах данных первичные ключи используются для создания связей (отношений) между таблицами через внешние ключи.

## Автоинкремент (Autoincrement)

Автоинкремент — это свойство, которое можно применить к столбцу (обычно к первичному ключу), чтобы при добавлении новой строки значение в этом столбце автоматически увеличивалось на единицу относительно предыдущей строки.

### Особенности Автоинкремента:

1. **Автоматическое Увеличение:** Каждый раз при добавлении новой строки значение в столбце автоинкремента увеличивается, обеспечивая уникальность.
2. **Удобство:** Упрощает процесс добавления новых записей, так как не требует ручного ввода уникального идентификатора.
3. **Порядковые номера:** Часто используется для присвоения порядковых номеров записям.

### Пример использования:

```
CREATE TABLE Students (  
  ID INTEGER PRIMARY KEY AUTOINCREMENT,  
  Name TEXT NOT NULL  
);
```

В этом примере:

- `ID` — это первичный ключ для таблицы `Students`.
- `AUTOINCREMENT` гарантирует, что каждый новый студент будет получать уникальный ID автоматически.

## Применение

Первичные ключи и автоинкремент широко используются в проектировании баз данных для обеспечения целостности данных и упрощения их управления. Они критически важны в реляционных базах данных, где отношения между таблицами и быстрый доступ к данным являются ключевыми аспектами.

В реляционных базах данных ключи играют важную роль в управлении и организации данных. Рассмотрим подробно различные типы ключей, используя в качестве примера SQLite и вашу таблицу `Students`.

## Примеры UPDATE Синтаксиса SQLite для Обновления Данных в Таблице Students

### Обновление Всех Строк

Для обновления всех строк в таблице используется следующий синтаксис:

```
UPDATE Students  
SET Major = 'Обновленная Специальность';
```

Этот запрос изменит значение столбца `Major` на 'Обновленная Специальность' для всех студентов в таблице.

### Выборочное Обновление

Для обновления определенных строк на основе условия используйте `WHERE` :

```
UPDATE Students
SET Major = 'Биология'
WHERE Name = 'Мария Петрова';
```

Этот запрос изменит специальность на 'Биология' только для студента с именем 'Мария Петрова'.

## Использование Условий

Вы можете использовать различные условия в `WHERE` для более точного определения строк, которые нужно обновить:

```
UPDATE Students
SET GPA = 5.0
WHERE Age > 20 AND Major = 'Математика';
```

Этот запрос установит средний балл (GPA) в 5.0 для всех студентов старше 20 лет, обучающихся на специальности 'Математика'.

## Особенности Обновления Различных Типов Данных

- **Текст (TEXT):** При обновлении текстовых данных помните о необходимости заключать текст в одинарные кавычки.

```
SET Name = 'Новое Имя'
```

- **Целые числа (INTEGER) и числа с плавающей точкой (REAL):** Для числовых данных кавычки не используются.

```
SET Age = 23, GPA = 4.8
```

- **NULL значения:** Для очистки значения столбца и установки его в NULL, используйте `NULL` без кавычек.

```
SET Major = NULL
```

## Пример Обновления

Предположим, что вы хотите обновить информацию о студенте 'Алексей Сидоров', изменить его возраст и специальность:

```
UPDATE Students
SET Age = 20, Major = 'Компьютерные науки'
WHERE Name = 'Алексей Сидоров';
```

В этом запросе:

- `UPDATE Students` указывает, что обновление будет производиться в таблице `Students`.
- `SET Age = 20, Major = 'Компьютерные науки'` меняет возраст на 20 и специальность на 'Компьютерные науки'.
- `WHERE Name = 'Алексей Сидоров'` ограничивает обновление только строкой, где имя студента - 'Алексей Сидоров'.

## Важные Моменты

- При использовании `UPDATE` важно указывать условие `WHERE`, чтобы не изменить все строки в таблице случайно.
- Всегда проверяйте типы данных и их соответствие столбцам таблицы.
- Обновление данных с автоинкрементным первичным ключом (например, `ID`) обычно не рекомендуется, так как это может нарушить целостность данных.
- Перед выполнением масштабных обновлений полезно сначала выполнить запрос `SELECT` с теми же условиями, чтобы убедиться в корректности выборки.

## Примеры Синтаксиса DELETE SQLite для Удаления Данных из Таблицы Students

## Удаление Всех Строк

Для удаления всех строк из таблицы используется следующий синтаксис:

```
DELETE FROM Students;
```

Этот запрос удалит все записи из таблицы `Students`, оставив ее пустой.

## Удаление с Условием

Чтобы удалить строки на основе определенного условия, используйте `WHERE`:

```
DELETE FROM Students
WHERE Name = 'Алексей Сидоров';
```

Этот запрос удалит все строки, где значение в столбце `Name` равно 'Алексей Сидоров'.

## Удаление с Использованием Агрегирующих Функций

SQLite не поддерживает прямое использование агрегирующих функций в запросах `DELETE`. Однако, вы можете использовать подзапросы для достижения аналогичного результата. Например, если вы хотите удалить всех студентов с GPA ниже среднего:

```
DELETE FROM Students
WHERE GPA < (SELECT AVG(GPA) FROM Students);
```

Здесь подзапрос `(SELECT AVG(GPA) FROM Students)` вычисляет средний балл среди всех студентов, и основной запрос удаляет тех студентов, чей GPA ниже этого среднего значения.

## Удаление с Использованием WHERE и Других Условий

Вы можете комбинировать условия в `WHERE` для более точного определения строк, которые нужно удалить:

```
DELETE FROM Students
WHERE Age > 20 AND Major = 'Физика';
```

Этот запрос удалит всех студентов старше 20 лет, обучающихся на специальности 'Физика'.

## Важные Моменты

- При использовании `DELETE` важно осторожно применять условие `WHERE`, чтобы случайно не удалить больше данных, чем было запланировано.
- В SQLite, как и в других СУБД, удаленные данные не могут быть восстановлены. Поэтому перед выполнением запроса `DELETE` рекомендуется убедиться в его корректности.
- Удаление строк, особенно большого количества, может повлиять на производительность базы данных. В некоторых случаях может потребоваться оптимизация или восстановление базы данных после массового удаления.

CRUD - это акроним, описывающий четыре основные операции, используемые во взаимодействии с базами данных и веб-приложениях: **C**reate (Создание), **R**ead (Чтение), **U**ppdate (Обновление) и **D**eleate (Удаление). Эти операции являются фундаментальными для работы с базами данных, включая SQLite, и они лежат в основе многих веб-приложений.

## CRUD в Контексте Веб-Приложений и SQLite

Используя вашу учебную базу данных `Students`, давайте рассмотрим, как каждая из операций CRUD реализуется в SQLite:

### 1. Create (Создание)

- Операция:** Вставка новых записей в таблицу.
- SQL-команда:** `INSERT`

- **Пример:** Добавление нового студента в таблицу `Students` :

```
INSERT INTO Students (Name, Age, Major, GPA) VALUES ('Иван Иванов', 20, 'Информатика', 4.0);
```

## 2. Read (Чтение)

- **Операция:** Чтение и извлечение данных из таблицы.
- **SQL-команда:** `SELECT`
- **Пример:** Выборка информации о всех студентах:

```
SELECT * FROM Students;
```

## 3. Update (Обновление)

- **Операция:** Изменение существующих записей в таблице.
- **SQL-команда:** `UPDATE`
- **Пример:** Обновление информации о студенте (например, изменение специальности):

```
UPDATE Students SET Major = 'Математика' WHERE ID = 1;
```

## 4. Delete (Удаление)

- **Операция:** Удаление записей из таблицы.
- **SQL-команда:** `DELETE`
- **Пример:** Удаление записи о студенте из таблицы `Students` :

```
DELETE FROM Students WHERE ID = 1;
```

## Важность CRUD в Веб-Приложениях

- **Интерфейс Пользователя:** В типичном веб-приложении, операции CRUD соответствуют действиям пользователя, таким как добавление новых данных (например, регистрация пользователя), просмотр данных (просмотр профилей или записей), редактирование (обновление профиля) и удаление (удаление учетной записи).
- **API и Бэкэнд:** На стороне сервера веб-приложений эти операции реализуются через API, который взаимодействует с базой данных, выполняя соответствующие SQL-команды.
- **SQLite в Роли Базы Данных:** SQLite идеально подходит для малых и средних веб-приложений, мобильных приложений или приложений с ограниченным масштабом пользователей. Он обеспечивает легкую и быструю реализацию операций CRUD, не требуя сложной настройки сервера баз данных.

## Реализация CRUD в Различных Фреймворках

- **Django и Flask с SQLAlchemy:** В популярных Python-фреймворках, таких как Django и Flask, операции CRUD могут быть реализованы через ORM (Object-Relational Mapping), что упрощает работу с базой данных, позволяя разработчикам работать с объектами в коде, а не напрямую с SQL-запросами.

## Заключение

CRUD является основополагающим концептом в проектировании веб-приложений и взаимодействии с базами данных. Понимание этих четырех основных операций и их реализация в контексте базы данных, такой как SQLite, имеет решающее значение для разработки эффективных и функциональных веб-приложений.

## Подробнее про Ключи (Primary Keys)

1. **Определение:** Первичный ключ — это уникальный идентификатор каждой строки в таблице. Он обеспечивает уникальность каждой записи и служит средством быстрого доступа к строкам данных.
2. **Использование в SQLite:** В вашей таблице `Students`, `ID` определен как первичный ключ с использованием `AUTOINCREMENT`. Это означает, что каждая новая запись в таблице будет автоматически получать уникальный `ID`.

ID INTEGER PRIMARY KEY AUTOINCREMENT,

3. **Значение:** Первичные ключи используются для установления связей между таблицами (через внешние ключи) и обеспечения целостности данных.

## Внешние Ключи (Foreign Keys)

1. **Определение:** Внешний ключ — это столбец или набор столбцов, используемый для установления связи с первичным ключом другой таблицы. Он указывает на первичный ключ в другой таблице, создавая таким образом связь между двумя таблицами.
2. **Использование в SQLite:** Допустим, у вас есть другая таблица, назовем её `Classes`, где каждый класс связан со студентом из таблицы `Students`. В таблице `Classes` можно создать внешний ключ, который будет ссылаться на `ID` из `Students`.

```
CREATE TABLE Classes (  
    ClassID INTEGER PRIMARY KEY AUTOINCREMENT,  
    StudentID INTEGER,  
    ClassName TEXT,  
    FOREIGN KEY (StudentID) REFERENCES Students(ID)  
);
```

3. **Значение:** Внешние ключи обеспечивают ссылочную целостность, гарантируя, что связи между таблицами логически корректны.

## Составные Ключи (Composite Keys)

1. **Определение:** Составной ключ состоит из двух или более столбцов, которые вместе образуют уникальный идентификатор строки. Составные ключи используются, когда уникальность не может быть обеспечена одним столбцом.
2. **Типы:** Составные ключи могут быть как первичными, так и внешними. Составной первичный ключ используется, когда необходимо уникально идентифицировать запись по комбинации нескольких полей. Составной внешний ключ ссылается на составной первичный ключ в другой таблице.
3. **Использование в SQLite:** Если бы в вашей таблице `Students` уникальность студента определялась не только `ID`, но и, скажем, сочетанием `Name` и `Major`, это могло бы быть составным ключом.

```
CREATE TABLE Students (  
    Name TEXT,  
    Major TEXT,  
    Age INTEGER,  
    GPA REAL,  
    PRIMARY KEY (Name, Major)  
);
```

4. **Значение:** Составные ключи полезны в случаях, когда отдельные столбцы сами по себе не обеспечивают уникальности записей. Они обеспечивают большую гибкость в определении уникальности и используются для создания более сложных связей между таблицами.

## Общие Принципы Использования Ключей в Реляционных Базах Данных

1. **Обеспечение Уникальности и Целостности:** Ключи гарантируют, что каждая запись в таблице уникальна и что данные между связанными таблицами соответствуют друг другу.
2. **Оптимизация Запросов:** Индексация по первичным и внешним ключам может значительно ускорить выполнение запросов, особенно в больших базах данных.
3. **Облегчение Связей Между Таблицами:** Ключи позволяют легко связывать данные из разных таблиц, что является основой реляционной модели данных.

## Важность Правильного Проектирования Ключей

Правильное проектирование и использование ключей в базе данных критически важно для эффективности, масштабируемости и целостности базы данных. Неправильное использование ключей может привести к дублированию данных, замедлению запросов и затруднениям в поддержке и масштабировании базы данных.

## Автоматически Создаваемый Ключ в SQLite

SQLite действительно автоматически создает специальный столбец под названием `ROWID` для каждой таблицы, если только не указано иное. `ROWID` представляет собой уникальный идентификатор для каждой строки в таблице и функционирует похоже на первичный ключ с автоинкрементом. Однако, `ROWID` не всегда является частью определения таблицы, которое вы видите или задаете в SQL-запросе.

## Создание Первичного Ключа Без AUTOINCREMENT

Если вы создадите первичный ключ без `AUTOINCREMENT`, SQLite не будет автоматически увеличивать значение этого ключа. В этом случае ответственность за обеспечение уникальности значений ключа ложится на вас. Если вы вставите строку без указания значения для такого первичного ключа или укажете дублирующееся значение, SQLite выдаст ошибку из-за нарушения уникальности первичного ключа.

## Риски Дублирования Без Уникального Ключа

Если первичный ключ не объявлен как уникальный, существует риск вставки дублирующихся значений, что может привести к нарушению целостности данных. В SQLite, однако, первичный ключ всегда уникален по определению. Если вы объявляете столбец как `PRIMARY KEY`, он автоматически становится уникальным, и SQLite не позволит вставить дублирующееся значение.

## Использование Составных Ключей

Составной ключ необходим, когда одного столбца недостаточно для уникальной идентификации каждой записи в таблице. Это может происходить, например, в случаях, когда у вас есть таблица с данными, где комбинация нескольких полей определяет уникальность записи.

Пример использования составного ключа:

```
CREATE TABLE Enrollments (  
  StudentID INTEGER,  
  CourseID INTEGER,  
  EnrollmentDate TEXT,  
  PRIMARY KEY (StudentID, CourseID)  
);
```

В этом примере комбинация `StudentID` и `CourseID` используется для уникальной идентификации записей в таблице `Enrollments`.

## Составные Первичные и Внешние Ключи

Составные первичные ключи часто используются в реляционных базах данных для уникальной идентификации записей в таблицах с отношениями "многие ко многим" или в других сложных случаях.

Составные внешние ключи используются для ссылки на составные первичные ключи в других таблицах. Это распространенная практика в сложных базах данных, где отнош

ения между данными в разных таблицах не могут быть однозначно представлены с помощью простого (одностолбцового) первичного или внешнего ключа.

## Пример с Составным Внешним Ключом

Рассмотрим две таблицы: `Students` (как у вас) и `Courses`, где каждый курс может иметь множество студентов, и каждый студент может быть записан на множество курсов. В таком случае мы могли бы создать связующую таблицу `Enrollments`:

```
CREATE TABLE Courses (  
  CourseID INTEGER PRIMARY KEY AUTOINCREMENT,
```



```
CourseName TEXT
);

CREATE TABLE Enrollments (
    StudentID INTEGER,
    CourseID INTEGER,
    EnrollmentDate TEXT,
    PRIMARY KEY (StudentID, CourseID),
    FOREIGN KEY (StudentID) REFERENCES Students(ID),
    FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)
);
```

В этом примере `Enrollments` имеет составной первичный ключ, состоящий из `StudentID` и `CourseID`, который одновременно является составным внешним ключом, ссылающимся на `Students` и `Courses`.

Создание таблицы с составным первичным ключом, как в примере с `Enrollments`, который связывает студентов (`Students`) и курсы (`Courses`), является классическим подходом к реализации отношений "многие ко многим" в реляционных базах данных. Давайте разберем, для чего это делается и какие преимущества это может предоставить.

## Цель Использования Составного Ключа в Отношениях "Многие ко Многим"

- Уникальность Записей:** Составной первичный ключ гарантирует, что каждая комбинация студента и курса будет уникальной. Это предотвращает дублирование записей, где один и тот же студент мог бы быть записан на один и тот же курс несколько раз.
- Гибкость:** Такая структура позволяет одному студенту быть записанным на несколько курсов и одному курсу включать множество студентов. Это идеально подходит для моделирования реальных сценариев обучения.
- Целостность Данных:** Использование внешних ключей в таблице `Enrollments`, ссылающихся на `Students` и `Courses`, обеспечивает ссылочную целостность. Это означает, что нельзя добавить запись в `Enrollments`, если соответствующие студент или курс отсутствуют в их основных таблицах.

## Сравнение с Альтернативными Подходами

- Стандартная Модель "Многие ко Многим"** без составного ключа часто реализуется похожим образом, но может включать дополнительные поля (например, уникальный ID для каждой записи в `Enrollments`). Это добавляет сложности и не всегда необходимо, особенно если уникальная комбинация студента и курса уже обеспечивает нужную функциональность.
- Производительность:** Во многих случаях использование составного первичного ключа может улучшить производительность, особенно для операций поиска и соединения таблиц, так как ключ оптимизируется и индексируется для быстрого доступа.
- Простота и Читаемость:** Использование составного ключа может упростить структуру базы данных, делая ее более понятной и легкой для навигации, особенно когда взаимосвязи между таблицами являются ключевым аспектом данных.

## Заключение

Использование составного первичного ключа в отношениях "многие ко многим" представляет собой эффективный и распространенный

подход в проектировании баз данных. Это позволяет точно и эффективно моделировать сложные взаимоотношения, обеспечивая при этом целостность и уникальность данных. В то время как существуют и другие способы организации отношений "многие ко многим", использование составных ключей часто является наиболее прямым и оптимальным решением, особенно в ситуациях, когда связи между данными являются ключевым фактором структуры базы данных. Этот метод облегчает как обслуживание базы данных, так и разработку приложений, использующих эти данные, предоставляя четкую и эффективную структуру для управления сложными связями.