5.6. Реляционные базы данных и язык SQL

Сайт: Samsung Innovation Campus
Курс: Мобильная разработка на Kotlin
Книга: 5.6. Реляционные базы данных и язык SQL

Напечатано:: Murad Rezvan
Дата: понедельник, 3 июня 2024, 18:01

Оглавление

1. Введение

С увеличением объемов информации возникла потребность в ее надежном хранении и быстром поиске.

Когда покупатель в супермаркете оплачивает товары на кассе, то кассир через сканер получает штрих-код, и информационная система ищет товар с таким штрих-кодом в БД, получает его цену, записывает информацию, что данный товар куплен и одновременно удаляет купленный товар из списка тех, которые есть в наличии. Отдел закупок супермаркета со своей стороны видит, сколько товаров в остатке, и может проанализировать, с какой скоростью товар раскупается, далее принимает решение, сколько товара заказать на следующую неделю.

В каком виде можно хранить информацию из приведенных примеров? Каким образом обеспечить различные права на доступ и изменение информации для разных сторон, которые используют одну и ту же информацию? Если у нас хранится информация о миллионах товаров или школьников, как обеспечить быстрый поиск нужной информации?

Обычные текстовые файлы не могут решить описанные выше задачи по ряду причин:

- любой поиск и изменение данных не эффективны, потому что текстовые файлы требуют последовательного чтения;
- один и тот же объект (товар, ученик) можно описать по-разному, поэтому если использовать естественный язык, то информацию будет очень сложно найти и в принципе провести какой-то анализ или обработку;
- из-за того, что текстовый файл изначально не разделен на части, невозможно обеспечить доступ множества пользователей к отдельным его частям с разным уровнем доступа: кому-то для чтения, кому-то для изменения, причем одновременно.

Для этих задач применяют базы данных: хранение данных определенным способом и программные средства для их эффективной обработки.

Можно дать такие определения.

- База данных (БД) — это специальным образом организованная совокупность данных о некоторой предметной области, хранящаяся во внешней памяти компьютера.
- Система управления базами данных (СУБД) _ — это программные средства для создания и обработки (добавления, изменения, поиска данных).

В этом модуле мы рассмотрим программные средства для работы с базами данных под Android. Для мобильных приложении часто используют реляционные базы данных, в которых данные представляются в виде таблиц. Чаще всего в мобильных приложениях используется SQLite - очень компактную бессерверную реализацию СУБД. SQL в названии означает, что эта СУБД, как и почти все реляционные базы данных управляется командами на языке SQL, основы которого мы рассмотрим в этом модуле.

2. Реляционная модель данных

Исторический прорыв в области хранения информации был совершен в 1970 году, когда англичанин Эдгар Кодд из компании IBM предложил реляционную модель данных. С ее появлением пришли понятия «база данных» (БД) и система управления БД (СУБД).

Понятно, что можно придумать множество различных способов, как хранить и обрабатывать данные в БД. Этот способ организации данных в БД определяется моделью данных.

Модель данных применительно к БД — это некоторые логические правила представления информации в памяти вычислительного устройства.

Позже была предложена более универсальная модель «сущность-связь», и Кодд предложил расширенную реляционную модель RM/V2. Модель «сущность-связь» очень легко перекладывается на реляционную модель, которая на данный момент остается самой распространенной и лежит в основе большинства СУБД.

Таблицы: поля и записи

Реляционная модель данных — это представление совокупности данных в виде двумерных таблиц.

Рассмотрим понятия, лежащие в основе реляционной модели, на конкретном примере.

Пусть перед нами стоит задача хранить информацию по сотрудникам некоторой компании. Рассмотрим на примере сущности «Сотрудник». Для проектирования и иллюстрации модели «сущность-связь» широко используют схемы БД, которые аналогичны диаграмме классов UML для отражения jобъектно-ориентированной модели.

У каждой сущности есть ряд свойств, которые ее описывают, — *атрибуты*. Пусть в нашем примере для каждого сотрудника мы хотим хранить такие поля: «Фамилия», «Имя», «Отчество», «Отдел», «Должность», «Адрес».

Сотрудник
Фамилия
Имя
Отчество
Отдел
Должность
Адрес
В заголовке мы указали наименование сущности, внутри список атрибутов.

В реляционной модели этой картинке соответствует таблица «Сотрудник» с столбцами: «Фамилия», «Имя», «Отчество», «Отдел», «Должность», «Оклад».

Если мы заполним таблицу набором значений атрибутов, например, по двум сотрудникам, то в терминах реляционной модели мы получим две записи. А таблица имеет 6 полей (см. табл. 4.7).

Адрес	Должность	Отдел	Отчество	Имя	Фамилия
г. Тверь ул. Мира д.6 кв.2	кассир	Бухгалтерия	Игоревич	Олег	Иванов
г. Тверь ул. Весенняя д.9	кладовщик	Склад	Петрович	Пётр	Соколов

Реляционная модель тесно связана с реляционной алгеброй, в терминах которой набор значений атрибутов (запись) носит название кортеж.

Ключевые поля

Каждый объект любой таблицы БД должен быть уникальным, иначе могут быть неприятности, связанные с неоднозначностью размещения одного и того же объекта несколько раз. Убирают эту неоднозначность ключевые поля, которые хранят только уникальную информацию об объекте и никогда не повторяются. Как быть в ситуации, когда в БД у людей полностью совпадают ФИО и дата рождения? Тогда ключом человека может быть серия и номер паспорта, а этого же человека в налоговой службе — ИНН, в пенсионном фонде — номер СНИЛС.

Пример с человеком описывает важность ключа, а также тот факт, что один и тот же человек может храниться в разных БД, а значит иметь разные свойства в таблицах. То есть человек один, а сущности разные. Еще примеры ключевых данных: серийный номер изделия, регистрационный номер авто, ну и, конечно же, номер на денежной банкноте, который хранится в БД казначейства и различает одинаковые купюры.

Таким образом, у каждой сущности должно быть определено специальное поле — ключ.

Ключ — это атрибут или группа атрибутов, значения которых уникальны для каждого экземпляра сущности. Ключ позволяет быстро идентифицировать каждый экземпляр сущности. Для ранее рассмотренного примера одним ключом может быть поле «Табельный номер», он уникален для каждого сотрудника и по нему легко найти конкретного сотрудника

Сотрудник

Табельный номер

Фамилия

Имя

Отчество

Отдел

Должность

Адрес

В схемах БД для выделения ключевые атрибуты отделяются от остальных чертой либо какими-то знаками (ключиками, звездочками и т. д.).

Идеи Кодда на языке таблиц

- Каждая таблица описывает одну сущность/отношение — порядок расположения полей в таблице не имеет значения;
- все значения одного поля имеют одинаковый тип данных;
- в таблице не может быть двух полностью одинаковых записей;
- порядок записей в таблице не определен.

При работе с реляционной БД характерно:

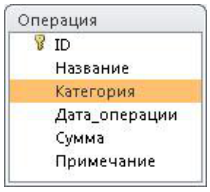
- количество и состав полей определяет разработчик БД, пользователи же работают с записями таблицы (добавляют, удаляют, редактируют);
- любое поле должно иметь уникальное имя;
- поля имеют тип (схоже с понятием тип переменной). Как правило, в СУБД можно задать следующие типы полей: строка символов, текст — целое число — вещественное число — денежная сумма — дата, время — логическое поле (истина или ложь, да или нет). Поля могут быть обязательными для заполнения или нет;
- таблица может содержать столько записей, сколько позволяет объем памяти;
- для работы с ними используют язык запросов SQL. Программисты не работают напрямую с файлами БД, для этого используют специальные программы — системы управления базой данных (СУБД), о которых пойдет речь далее.

До сих пор мы рассматривали примеры отдельных таблиц, но очевидно, что весь объем данных предметной области нельзя уместить в одну таблицу. Причем это связано не только с тем соображением, что это будет слишком большая таблица. Вспомним классы в ООП. Здесь в чем-то похожий подход --- данные группируются в соответствии с логикой принадлежности к той или иной сущности. На практике БД --- это набор таблиц, которые связаны между собой.

Пример

Перед нами стоит задача разработать БД для ведения семейного бюджета, главная задача которой фиксировать доходы и расходы семьи, чтобы затем можно было провести их анализ.

Начнем с того, что БД должна хранить данные о всех расходах и доходах, то есть какие происходили денежные операции. И для каждой операции еще нужно знать, на что были потрачены деньги или откуда пришли (зарплата, кредит и т. д.). Рассмотрим таблицу 4.9 «Операция» со следующей структурой



ID --- это ключевое поле, поэтому на схеме слева от наименования нарисован ключик.

В этой структуре плохо то, что мы храним все детальные операции, но не можем провести их анализ, сгруппировать: например, посчитать, сколько за месяц мы потратили на питание, сколько на транспорт и т. д. Значит, нам нужно указание, из какой категории каждая операция.

Заполним таблицу «Операция».

Примечание	Сумма	Дата операции	Категория	Название	ID
	120 000	01.03.2018	Остаток	Начальный баланс	1
	-3 300	06.03.2018	Продукты	Продукты	2
Поздравление на 8 марта	-1 800	08.03.2018	Разное	Разное	3
	-1 000	08.03.2018	Развлечения	Развлечения	4
	-210	09.03.2018	Продукты	Продукты	5
	50 000	10.03.2018	Зарплата	Зарплата	6

Нормализация

Данные этой таблицы страдают избыточностью из-за совпадений некоторых значений поля «Категория». Причем, из-за того, что это текстовые поля, будет расходоваться много памяти. Более того, если мы захотим изменить наименование категории, то нам придется сделать это во всех записях таблицы. Убрать избыточность позволит отдельная таблица или таблицы. В нашем случае можно создать таблицу, которая хранит сущность «Категория»

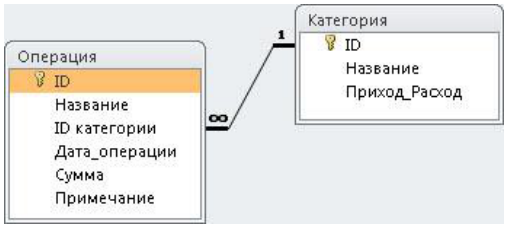
Приход/Расход	Название	ID
1	Остаток	1
0	Продукты	2
0	Разное	3
0	Развлечения	4
1	Зарплата	0
0	Транспорт	6
0	Медицина	7
0	Образование	8
1	Кредит	9
0	Выплата по кредиту	10

В таблицу мы дополнительно включили новое поле, в котором будем фиксировать, какая это категория: доход или расход. Это позволит группировать операции не только по категориям, но и еще более укрупненно: например, все расходы за месяц. Если приход, то ставим в таблице 1, если расход, то --- 0.

Связи

Кроме этого, мы должны связать таблицу операций с категориями. Для этого нужно выяснить, какая из них будет главная, а какая подчиненная. Поскольку таблица «Категория» хранит неповторяющиеся значения, она будет *главной*, а таблица «Операция» будет *подчиненной*.

Как физически связываются таблицы? Для этого подчиненной таблице добавляют еще одно поле для связи с главной. То есть таблице «Операция» нужно добавить поле ID_категории. Затем свяжем ключевое поле главной таблицы с новым полем подчиненной таблицы. Новое поле подчиненной таблицы называют «внешний ключ&raquo» и оно отображает информацию, которая хранится в другой таблице, а именно в ключевом поле главной таблицы. То ключевое поле главной таблицы называется главным. Для отображения связи между сущностями на схемах БД используют линии.



Наполнение таблицы «Операция» станет следующим

Примечания	Сумма	ID категории	Дата операции	Наименование	ID
	120000	1	01.03.2018	Начальный баланс	1
	-3 300	2	06.03.2018	Хлеб, мясо, овощи, фрукты	2
Поздравление с 8 марта	-1 800	3	08.03.2018	Цветы	3
	-1 000	4	08.03.2018	Билеты в цирк	4
	-210	1	09.03.2018	Хлеб, творог, сметана	5
	50 000	5	10.03.2018	Зарплата Сергея, февраль	6

Ссылочная целостность

Что случится, если в таблице «Категория» удалить любую запись? Очевидно, что возникнет проблема в таблице «Операция», потому что для операции по заданному коду категории мы не сможем получить нужную информацию. То есть сущность «Операция» зависима от сущности «Категория». Если в таблице есть внешний ключ, то он должен обязательно совпадать с одним из значений первичного ключа в другой таблице. Это называется соблюдением *ссылочной целостности*. Современные СУБД имеют различные средства поддержки ссылочной целостности БД. Например, пользователю не разрешат удалить таблицу, если есть таблицы, которые от нее зависят.

Типы связей

При проектировании БД и работы с ней очень важно устанавливать связи между таблицами. Не менее важно понимать какого типа эти связи. Рассмотрим основные из них.

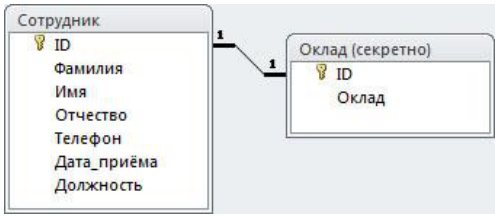
1. Связь «один ко многим»

На рисунке со схемой БД, соединяющей таблицы, отображены 1 и ∞. Это условное обозначение типа связи 1:M --- «один ко многим». Это самая распространенная связь.

Связь «один ко многим» означает, что с одной записью в таблице, на стороне которой стоит «1», могут быть связаны сколько угодно записей из другой таблицы, где стоит «∞». То есть во второй таблице может быть несколько записей с одинаковым значением внешнего ключа, что мы и наблюдали.

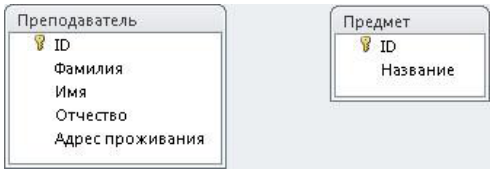
2. Связь «один к одному»

Редко, но встречается связь «1:1», когда каждой записи в первой таблице соответствует 0 или 1 запись в связанной таблице. Это обычно используют для вертикального разделения большой таблицы на две части. Например, сделать часть полей таблицы доступной всем, а остальные поля вынести в другую таблицу и показывать в зависимости от прав пользователя.



3. Связь «многие ко многим»

Наиболее сложная, но при этом часто встречаемая связь «M:M». Например, в школе преподаватель может вести несколько предметов и в то же время предмет с таким наименованием могут преподавать несколько учителей.

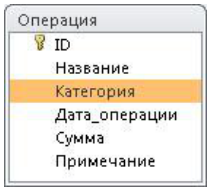


Значит есть сущности «Преподаватель» и «Предмет», между которыми связь «многие ко многим».

Реляционные БД не могут на физическом уровне явно реализовать связь «многие ко многим». И действительно, невозможно в плоской таблице к одной записи без ее дублирования привязать разные значения внешнего ключа.

Появление при проектировании связи «многие ко многим» всегда говорит о том, что не достает третьей сущности, которая описывает процесс взаимодействия этих двух сущностей.

В данном случае явно не хватает сущности «Преподавание». То есть нужно просто ввести новую сущность, которая и будет хранить все нужные соответствия ключей из таблиц «Преподаватель» и «Предмет»



Пример содержимого таблиц.

Таблица "Преподаватель"

Адрес проживания	Отчество	Имя	Фамилия	ID
г. Москва, ул. Орджоникидзе 3	Иванович	Иван	Иванов	1
г. Москва, ул. Рабочая 12	Петрович	Пётр	Петров	2
г. Москва, 1-й Зборовский пер. 3	Николаевна	Анна	Сидорова	3

Таблица "Предмет"

ID	Название
1	Математика
2	Алгебра
3	Геометрия
4	География
5	Информатика
6	Информатика

ID	Название
7	Русский язык

Таблица "Преподавание"

ID_Предмета	ID_Предподавателя	ID
1	1	1
2	1	2
3	1	3
6	2	4
7	2	5
5	3	6
1	3	7
2	3	8

Из таблицы «Преподавание» легко определить, что, к примеру, преподаватель Петров ведет литературу и русский язык.

Обратим еще внимание на то, что в полях ID_преподавателя и ID_предмета только целочисленные значения. Это существенно увеличивает производительность при поиске информации. Но главное --- не приведет к избыточности и не позволит допустить синтаксическую ошибку при ручном вводе данных.

3. Язык SQL

Одним из языков, появившихся в результате разработки реляционной модели данных, является язык SQL (Structured Query Language), который в настоящее время получил очень широкое распространение и фактически стал стандартом реляционных баз данных.

Язык SQL достаточно компактен, содержит небольшое количество команд. Мы рассмотрим их подмножество, необходимое в обычной практической работе.

Запись SQL-операторов

Идентификаторы языка SQL предназначены для обозначения объектов в базе данных: имен таблиц, столбцов и других объектов базы данных. В соответствии со стандартом SQL идентификатор:

- по умолчанию может содержать строчные и прописные буквы латинского алфавита (A-Z, a-z), цифры (0–9), символ подчеркивания (_);
- может иметь длину до 128 символов;
- должен начинаться с буквы;
- не может содержать пробелы.

Большинство компонентов языка не чувствительны к регистру. Отдельные SQL-операторы и их последовательности будут иметь произвольное количество отступов и переносятся по строкам, что позволяет создавать более читаемый вид при использовании выравнивания.

Комментарии в SQLite обозначаются двумя последовательными минусами (-), которые комментируют остаток строки. Многострочные комментарии, как в языке Java, обозначаются парами символов (/* */).

СУБД SQLite

SQLite — компактная локальная реляционная СУБД. SQLite не использует парадигму клиент-сервер, то есть движок SQLite не является отдельно работающим процессом, что особенно важно при работе на мобильных устройствах. Такой подход уменьшает накладные расходы, время отклика и упрощает программу. SQLite хранит всю базу данных (включая определения, таблицы, индексы и данные) в единственном стандартном файле на том устройстве, на котором исполняется программа.

Несколько процессов или потоков могут одновременно без каких-либо проблем читать данные из одной базы. Запись в базу можно осуществить только в том случае, если никаких других запросов в данный момент не обслуживается; в противном случае попытка записи заканчивается неудачей, и в программу возвращается код ошибки. Благодаря архитектуре движка возможно использовать SQLite как на встраиваемых системах, так и на выделенных машинах с гигабайтными массивами данных.

По ряду оценок функциональность SQLite находится где-то между MySQL и PostgreSQL. Однако на практике SQLite нередко оказывается в 2–3 раза (и даже больше) быстрее. Такое возможно благодаря высокоупорядоченной внутренней архитектуре и устранению необходимости в соединениях типа «сервер-клиент» и «клиент-сервер».

На практике приложения под Android не используют СУБД SQLite для хранения данных большого объема и со сложной структурой. Для таких случаев наиболее распространено применение клиент-серверных СУБД.

Далее мы приступим к изучению команд SQL — языка запросов для реляционных БД. И по ходу изложения будем отмечать особенности, характерные для СУБД SQLite.

3.1. Создание таблиц CREATE TABLE

Таблицы базы данных создаются с помощью оператора CREATE TABLE. Эта команда создает пустую таблицу, то есть таблицу, не имеющую строк. Значения в эту таблицу добавляются с помощью оператора INSERT. Оператор CREATE TABLE определяет имя таблицы и множество поименованных столбцов в указанном порядке. Для каждого столбца могут быть определены тип и размер. Каждая создаваемая таблица должна иметь по крайней мере один столбец.

Базовый синтаксис оператора создания таблицы имеет следующий вид:

```
 CREATE TABLE <имя_таблицы>
(<имя_столбца> <тип_данных>
[NULL | NOT NULL ] [,...n])`
```

Здесь и далее квадратные скобки используются для необязательных частей конструкции, то есть сами квадратные скобки не пишутся, а то, что в них, можно опускать. Угловые скобки тоже не пишутся, а указывают на тип данных, вместо которых нужно подставить реальные значения, например, имя таблицы.

Конструкция NOT NULL обозначает, что поле должно быть обязательно заполнено.

Пример

Создать таблицу для хранения данных о товарах. Необходимо учесть такие сведения, как название, тип товара и его цену.

```
CREATE TABLE Product
    (id INTEGER PRIMARY KEY AUTOINCREMENT,
    Name  TEXT  NOT NULL UNIQUE,
    Price      REAL)`
```

Обратите внимание на поле id. Это поле встречается в таблицах SQL и является первичным ключом, однозначно определяет объект, в нашем случае — товар.

Первый столбец обозначен, как *primary key* (первичный ключ), это понятие мы уже разобрали в предыдущей теме. Слово *autoincrement* указывает, что база данных будет автоматически увеличивать значение ключа при добавлении каждой записи, что и обеспечивает его уникальность (поле — счетчик).

Создание таблицы в SQLite

В SQLite существует договоренность первый столбец всегда называть _id, это не жесткое требование SQLite, однако может понадобиться при использовании контент-провайдера в Android.

Стоит также иметь в виду, что в SQLite, в отличие от многих других СУБД, типы данных столбцов являются лишь подсказкой, то есть не вызовет никаких нареканий попытка записать строку в столбец, предназначенный для хранения целых чисел или наоборот. Этот факт можно рассматривать как особенность базы данных, а не как ошибку, на это обращают внимание авторы SQLite.

Возможные типы полей:

- TEXT — строки или символы в кодировке UTF-8, UTF-16BE или UTF-16LE;
- INTEGER — целое число;
- REAL — дробное число;
- BLOB — бинарные данные;
- TIMESTAMP — метка времени;
- NULL — пустое значение.

В SQLite отсутствует тип, работающий с датами. Можно использовать строковые значения, например, как 2015-02-16 (16 февраля 2015 года). Для даты со временем рекомендуется использовать формат 2015-02-16T07:58. В таких случаях можно использовать некоторые функции SQLite для добавления дней, установки начала месяца и т. д. Учтите, что SQLite не поддерживает часовые пояса. Также не поддерживается тип boolean. Используйте числа 0 для false и 1 для true. Не используйте тип BLOB для хранения данных (картинки) в Android. Лучше хранить в базе путь к изображениям, а сами изображения хранить в файловой системе.

Как было отмечено выше, SQLite поддерживает типы TEXT (аналог String в Kotlin), INTEGER (аналог Long в Kotlin) и REAL (аналог Double в Kotlin). Остальные типы следует конвертировать, прежде чем сохранять в базе данных. SQLite сама по себе не проверяет типы данных, поэтому вы можете записать целое число в колонку, предназначенную для строк и наоборот.

3.2. Выборка данных SELECT

Инструкция SELECT

Оператор SELECT — один из наиболее важных и самых распространенных операторов SQL. Он позволяет производить выборки данных из таблиц и преобразовывать к нужному виду полученные результаты. Оператор SELECT имеет следующий формат:

```
SELECT [ALL | DISTINCT ] {*|[имя_столбца
    [AS     новое_имя]]} [,...n]
    FROM  имя_таблицы [[AS] псевдоним] [,...n]
    [WHERE  <условие_поиска>]
    [GROUP BY  имя_столбца [,...n]]
    [HAVING <критерии выбора групп>]
    [ORDER BY имя_столбца [,...n]]
```

Оператор SELECT определяет поля (столбцы), которые будут входить в результат выполнения запроса. В списке они разделяются запятыми и приводятся в такой очередности, в какой должны быть представлены в результате запроса. Символом * можно выбрать все поля, а вместо имени поля применить выражение из нескольких имен.

Предложение FROM

Предложение FROM задает имена таблиц и представлений, которые содержат поля, перечисленные в операторе SELECT. Необязательный параметр псевдонима — это сокращение, устанавливаемое для имени таблицы.

Обработка элементов оператора SELECT выполняется в следующей последовательности:

FROM — определяются имена используемых таблиц; WHERE — выполняется фильтрация строк объекта в соответствии с заданными условиями; GROUP BY — образуются группы строк, имеющих одно и то же значение в указанном столбце; HAVING — фильтруются группы строк объекта в соответствии с указанным условием; SELECT — устанавливается, какие столбцы должны присутствовать в выходных данных; ORDER BY — определяется упорядоченность результатов выполнения операторов.

Порядок предложений и фраз в операторе SELECT не может быть изменен. Только два предложения SELECT и FROM являются обязательными, все остальные могут быть опущены. SELECT — закрытая операция: результат запроса к таблице представляет собой другую таблицу.

Пример 1. Составить список всех данных о всех продуктах.

```
SELECT * FROM Product
```

Пример 2. Составить список всех продуктов.

```
SELECT Name FROM Product
```

Результат выполнения запроса может содержать дублирующиеся значения.

Предикат DISTINCT

Для исключения дублирования используют предикат DISTINCT. Необходимо помнить, что использование DISTINCT может резко замедлить выполнение запросов. Откорректированный пример выглядит следующим образом:

```
SELECT DISTINCT Name FROM Product
```

Предложение WHERE

С помощью WHERE-параметра пользователь определяет, какие строки появятся в результате запроса. За ключевым словом WHERE следует перечень условий поиска, определяющих те строки, которые должны быть выбраны при выполнении запроса.

Существует пять основных типов условий поиска (или предикатов).

Сравнение: сравниваются результаты вычисления одного выражения с результатами вычисления другого.

Диапазон: проверяется, попадает ли результат вычисления выражения в заданный диапазон значений.

Принадлежность множеству: проверяется, принадлежит ли результат вычислений выражения заданному множеству значений.

Соответствие шаблону: проверяется, отвечает ли некоторое строковое значение заданному шаблону.

Значение NULL: проверяется, содержит ли данный столбец определитель NULL (неизвестное значение).

Сравнение

В языке SQL можно использовать следующие операторы сравнения: = — равенство; < — меньше; > — больше; <= — меньше или равно; >= — больше или равно; <> — не равно.

Пример 3. Показать все товары, стоимостью более 100.

```
SELECT * FROM Product WHERE Price > 100
```

В выражениях можно использовать логические операторы AND, OR или NOT, а также скобки.

Пример 4. Вывести список товаров, цена которых больше 100 и меньше или равна 200.

```
SELECT
     FROM Product
     WHERE Price > 100 AND Price <= 200
```

Соответствие шаблону

С помощью оператора LIKE можно выполнять сравнение выражения с заданным шаблоном, в котором допускается использование символов-заменителей:

символ % — вместо этого символа может быть подставлено любое количество произвольных символов; символ _ заменяет один символ строки;

[] — вместо символа строки будет подставлен один из возможных символов, указанный в этих ограничителях;

[^] — вместо соответствующего символа строки будут подставлены все символы, кроме указанных в ограничителях.

Пример 5. Вывести список колбас и сыров.

```
SELECT Name, Price
     FROM Product
     WHERE Name LIKE "Сыр%" OR
     Name LIKE "Колбаса%"
```

Обратите внимание. Вместо союза «и», который мы использовали в русском языке, нужно использовать «OR», то есть «или».

Диапазон

Оператор BETWEEN используется для поиска значения внутри некоторого интервала, определяемого своими минимальным и максимальным значениями. При этом указанные значения включаются в условие поиска.

Пример 6. Вывести список товаров, цена которых лежит в диапазоне от 100 до 150

```
SELECT Name, Price
     FROM Product
     WHERE Price BETWEEN 100 And 150
```

Пример 7. Вывести список товаров, цена которых НЕ лежит в диапазоне от 100 до 150.

```
 SELECT Name, Price
     FROM Product
     WHERE Price NOT BETWEEN 100 AND 150
```

Принадлежность множеству

Оператор IN используется для сравнения некоторого значения со списком заданных значений, при этом проверяется, соответствует ли результат вычисления выражения одному из значений в предоставленном списке. При помощи оператора IN может быть достигнут тот же результат, что и в случае применения оператора OR, однако оператор IN выполняется быстрее.

Пример 8/

```
SELECT Name, Price
     FROM Product
     WHERE Name IN ("Сырок Глазированный", "Сыр домашний")
```

Значение NULL

Оператор IS NULL используется для сравнения текущего значения со значением NULL — специальным значением, указывающим на отсутствие любого значения.

Пример 9. Вывести товары с неуказанными ценами.

```
SELECT Name FROM Product
   WHERE Price IS NULL
```

IS NOT NULL используется для проверки присутствия значения в поле.

Предложение ORDER BY

В общем случае строки в результирующей таблице SQL-запроса никак не упорядочены. Однако их можно требуемым образом отсортировать, для чего в оператор SELECT помещается фраза ORDER BY. Сортировка может выполняться по нескольким полям, в этом случае они перечисляются за ключевым словом ORDER BY через запятую.

Пример 10. Вывести список продуктов в алфавитном порядке.

```
SELECT Name
    FROM Product
    ORDER BY Name
```

Пример 11. Вывести список продуктов от самого дорогого до самого дешевого.

```
SELECT Name, Price FROM Product ORDER BY Price DESC
```

3.3. Агрегированные запросы

С помощью итоговых (агрегатных) функций запроса можно получить статистические сведения о множестве отобранных значений выходного набора.

Пользователю доступны следующие основные итоговые функции:

- Count (выражение) — определяет количество записей в выходном наборе SQL-запроса;
- Min/Max (выражение) — определяет наименьшее и наибольшее из множества значений в некотором поле запроса;
- Avg (выражение) — эта функция позволяет рассчитать среднее значение множества значений, хранящихся в определенном поле отобранных запросом записей. Оно является арифметическим средним значением, то есть суммой значений, деленной на их количество;
- Sum (выражение) — вычисляет сумму множества значений, содержащихся в определенном поле отобранных запросом записей.

Все эти функции оперируют со значениями в единственном столбце таблицы или с арифметическим выражением и возвращают единственное значение. Функции COUNT, MIN и MAX применимы как к числовым, так и к нечисловым полям, тогда как функции SUM и AVG могут использоваться только в случае числовых полей. При вычислении результатов любых функций сначала исключаются все пустые значения, после чего требуемая операция применяется только к оставшимся конкретным значениям столбца.

Вариант COUNT(*) — особый случай использования функции COUNT, его назначение состоит в подсчете всех строк в результирующей таблице, независимо от того, содержатся там пустые, дублирующиеся или любые другие значения.

Если до применения обобщающей функции необходимо исключить дублирующиеся значения, следует перед именем столбца в определении функции поместить ключевое слово DISTINCT. Оно не имеет смысла для функций MIN и MAX, однако его использование может повлиять на результаты выполнения функций SUM и AVG, поэтому необходимо заранее обдумать, должно ли оно присутствовать в каждом конкретном случае. Кроме того, ключевое слово DISTINCT может быть указано в любом запросе не более одного раза.

Агрегатные функции используются в запросах SELECT.

Пример 1. Вывести первый товар по алфавиту.

```
sql
SELECT MIN(Name)
     FROM Product
```

```
**Пример 2.** Определить количество товаров.
```

```
sql
SELECT COUNT(*)
     FROM Product
```

3.4. Добавление данных INSERT

Оператор INSERT применяется для добавления записей в таблицу. Формат оператора:

```
INSERT INTO <имя_таблицы>
    [(имя_столбца [,...n])]
    {VALUES (значение[,...n])|
    <SELECT_оператор>}
```

Первая форма оператора INSERT с параметром VALUES предназначена для вставки единственной строки в указанную таблицу. Список столбцов указывает столбцы, которым будут присвоены значения в добавляемых записях. Список может быть опущен, тогда подразумеваются все столбцы таблицы (кроме объявленных как счетчик), причем в определенном порядке, установленном при создании таблицы.

Если в операторе INSERT указывается конкретный список имен полей, то любые пропущенные в нем столбцы должны быть объявлены при создании таблицы как допускающие значение NULL, за исключением тех случаев, когда при описании столбца использовался параметр DEFAULT. Список значений должен следующим образом соответствовать списку столбцов:

- количество элементов в обоих списках должно быть одинаковым;
- должно существовать прямое соответствие между позицией одного и того же элемента в обоих списках, поэтому первый элемент списка значений должен относиться к первому столбцу в списке столбцов, второй — ко второму столбцу и т. д.;
- типы данных элементов в списке значений должны быть совместимы с типами данных соответствующих столбцов таблицы.

Пример

Добавить в таблицу Product новую запись.

```
INSERT INTO Product (Name, Price) VALUES ("Ливерная колбаса", 50.70)
```

Если столбцы таблицы Product указаны в полном составе и в том порядке, в котором они перечислены при создании, оператор можно упростить:

```
INSERT INTO Product VALUES ("Ливерная колбаса", 50.70)
```

3.5. Изменение данных - UPDATE

Оператор UPDATE применяется для изменения значений в группе записей или в одной записи указанной таблицы. Формат оператора:

```
UPDATE имя_таблицы SET имя_столбца=<выражение>[,...n]
    [WHERE <условие_отбора>]
```

Пример 1. Установить нулевую цену для товаров с неуказанной ценой.

```
UPDATE Product SET Price=0 WHERE Price IS NULL
```

Пример 2. Увеличить цену дешевых товаров на 25%.

```
UPDATE Product SET Price=Price*1.25
     WHERE Price < 10
```

3.6. Удаление данных - DELETE

Оператор DELETE предназначен для удаления группы записей из таблицы. Формат оператора:

```
FROM <имя_таблицы>[WHERE <условие_отбора>]
```

Пример 1. Удалить все записи о товарах.

```
DELETE FROM Product
```

В документации по SQLite указано, что не поддерживаются конструкции SQL, которые позволяют удалить или изменить существующий столбец в таблице (например, его имя или тип). Но можно прибегнуть к «хитрости»: создать другую таблицу, скопировав в нее нужные данные из старой:

```
-- создаем временную таблицу
CREATE TEMPORARY TABLE Temper_backup(name,temp);
-- копируем данные из таблицы Temper во временную таблицу Temper_backup
INSERT INTO Temper_backup SELECT name,temp FROM Temper;
-- удаляем таблицу Temper
DROP TABLE Temper;
-- создаем таблицу Temper
CREATE TABLE Temper(name,NEW);
-- вставляем данные из таблицы Temper_backup в таблицу Temper
INSERT INTO Temper SELECT name,temp FROM Temper_backup;
-- удаляем таблицу Temper_backup
DROP TABLE Temper_backup;
```

4. Работа с базой данных SQLite на Android-устройстве

Когда ваше приложение создает базу данных, она сохраняется как

```
DATA/data/имя_пакета/databases/имя_базы.db
```

Например, с помощью контент-провайдера несколько приложений могут использовать одну и ту же базу данных.

Приведенный метод возвращает путь к каталогу DATA.

```
Environment.getDataDirectory()
```

Основными пакетами для работы с базой данных являются android.database и android.database.sqlite.

Давайте создадим простейший менеджер заметок и в процессе реализуем все шаги необходимые шаги по реализации работы с SQlite в android-приложении. Будем сохранять информацию в одной таблице из двух полей текстового и числового, например, набранные очки или рейтинг игроков в какой-нибудь игре.

Определяем класс данных

Для работы с БД очень удобно сгруппировать данные в классе

```
data class Result(val name: String, val result: Int)
```

С этим классом будут общаться все: интерфейс собирать данные из полей ввода, "комплектовать" их в Result и передавать собранный ответ в базу данных, вернее, в методы класса, который будет ее обслуживать. А этот обслуживающий класс в свою очередь будет забирать данные из таблиц базы данных и передавать интерфейсу для отображения.

Определяем структуру таблицы

Удобно организовать отдельный класс для хранения информации о структуре таблицы и команды по созданию и удалению таблицы в отдельном классе

```
object DBContract {
    // Группируем данные
  object Entry : BaseColumns {
        const val TABLE_NAME =  "results"
  const val COLUMN_NAME_NAME =  "name"
  const val COLUMN_NAME_RESULT = "result"
  }
    // Команды для работы
  const val SQL_CREATE =
        "CREATE TABLE ${Entry.TABLE_NAME} (" +
                "${BaseColumns._ID} INTEGER PRIMARY KEY," +
                "${Entry.COLUMN_NAME_NAME} TEXT," +
                "${Entry.COLUMN_NAME_RESULT} TEXT)"  const val SQL_DELETE = "DROP TABLE IF EXISTS ${Entry.TABLE_NAME}"
}
```

Обратите внимание на поле _ID. Это поле обычно добавляется в таблицу, потому что оно используется в некоторых стандартных методах.

Переопределяем SQLiteOpenHelper

Для создания и обновления базы данных в Android предусмотрен класс SQLiteOpenHelper. При разработке приложения, работающего с базами данных, необходимо создать класс-наследник от SQLiteOpenHelper, в котором обязательно реализовать два абстрактных метода:

- `onCreate(SQLiteDatabase db)` — вызывается один раз при создании БД;
- `onUpgrade (SQLiteDatabase db, int oldVersion, int newVersion)` — вызывается, когда необходимо обновить БД (в данном случае под обновлением имеется в виду не обновление записей, а обновление структуры базы данных).

По желанию можно реализовать метод:

- `onOpen()` — вызывается при открытии базы данных.

Класс создания может выглядеть примерно так

```
class SimpleDbHelper(context: Context) : SQLiteOpenHelper(context, DATABASE_NAME, null, DATABASE_VERSION) {

    object DBContract {
        // Группируем данные
  object Entry : BaseColumns {
            const val TABLE_NAME =  "results"
  const val COLUMN_NAME_NAME =  "name"
  const val COLUMN_NAME_RESULT = "result"
  }
  // Команды для работы
  const val SQL_CREATE =
            "CREATE TABLE ${Entry.TABLE_NAME} (" +
                    "${BaseColumns._ID} INTEGER PRIMARY KEY," +
                    "${Entry.COLUMN_NAME_NAME} TEXT," +
                    "${Entry.COLUMN_NAME_RESULT} TEXT)"  const val SQL_DELETE = "DROP TABLE IF EXISTS ${Entry.TABLE_NAME}"
  }

    override fun onCreate(db: SQLiteDatabase) {
        db.execSQL(DBContract.SQL_CREATE)
    }
    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
        onCreate(db)
    }
    override fun onDowngrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
        onUpgrade(db, oldVersion, newVersion)
    }
    companion object {
        // If you change the database schema, you must increment the database version.
  const val DATABASE_VERSION = 1
  const val DATABASE_NAME = "results.db"
  }
}
```

В реальном приложении изменение структуры базы данных и ее таблиц, конечно, должно происходить без потери пользовательских данных, то есть метод `Upgrade` стоит модернизировать.

Для приложения можно создать несколько БД, все они будут доступны из любого класса программы, но недоступны для других программ. Класс, который предоставляет методы для добавления, обновления, удаления и выборки данных из БД носит имя SQLiteDatabase. Он работает с базой данных SQLite напрямую и имеет свои методы для открытия, запроса, обновления данных и закрытия базы, такие как insert(), update(), delete() соответственно.

Реализация добавления данных в БД

Для вставки нужно открыть базу данных для записи и вызвать команду INSERT. Конечно, можно использовать непосредственно SQL-синтаксис, но проще и безопаснее использовать функцию insert() класса SQLiteDatabase:

```
fun insert(
    table: String!,
    nullColumnHack: String!,
    values: ContentValues!
): Long
```

Она принимает имя таблицы и данные в виде Map, в котором названиям полей соответствуют их значения. Второй параметр используется для вставки пустых строк, для случая, когда values пусто, используется это значение как имя какого-нибудь столбца. Обычно значение этого параметра null.

Удобно оформить вставку результатов в БД в виде метода DBHelper-а:

```
public fun insert(result: Result){
  // Открываем базу на запись
  val db = writableDatabase
  // Комплектуем данные для вставки
  val values = ContentValues().apply {
  put(DBContract.Entry.COLUMN_NAME_NAME, result.name)
        put(DBContract.Entry.COLUMN_NAME_RESULT, result.result)
    }
  // вставляем данные в базу данных
  db?.insert(DBContract.Entry.TABLE_NAME, null, values)
}
```

Получение результатов из БД

Для получения данных из БД используется команда SELECT. Как и в случае вставки, удобно использовать для этого функцию query():

```
query(table: String!,
      columns: Array<String!>!,
      selection: String!,
      selectionArgs: Array<String!>!,
      groupBy: String!,
      having: String!,
      orderBy: String!,
      limit: String!
): Cursor
```

У нее значительно больше параметров, потому что команда SELECT более сложна, она состоит из нескольких частей и в функцию query() значения этих частей передаются по-отдельности.

Проще всего понять на примере

Если мы хотим сделать запрос

```
SELECT col-1, col-2 FROM tableName WHERE col-1='apple',col-2='mango'
GROUPBY col-3 HAVING count(col-4) > 5  ORDERBY col-2 DESC LIMIT 15;
```

То можем написать в коде так:

```
val table = "tableName"
val columns = arrayOf("col-1", "col-2")
val selection = "col-1 =? AND col-2=?"
val selectionArgs = arrayOf("apple", "mango")
val groupBy: String = col - 3
val having = " COUNT(col-4) > 5"
val orderBy = "col-2 DESC"
val limit = "15"

val cursor = query(tableName, columns, selection, selectionArgs, groupBy, having, orderBy, limit)
```

В этом коде нужно обратить внимание на значение `selection`: `col-1 =? AND col-2=?"`. Вместо реальных значений в нем используются знаки вопроса, в которые потом последовательно подставляются значения из `selectionArgs`. Это удобнее, чем формировать часть запроса WHERE самостоятельно, и безопаснее, потому что при этом происходит экранирование (значения автоматически заключаются в апострофы), что снижает риск SQL-инъекций.

В нашем проекте все записи можно получить, например, так:

```
val cursor = readableDatabase.query(DBContract.Entry.TABLE_NAME, null, null,
    null, null,null, null)
```

Так как нам надо получить просто все записи без каких бы то ни было уточнений, то большинство параметров - `null`.

Работа с объектом Cursor

Конечно, по запросу возвращается не вся база огромным массивом, а курсор - объект, при помощи которого можно забрать отобранные данные из базы. Обычная схема работы с курсором такова:

```
// ставим курсор в начало сета данных
cursor.moveToFirst()
// проходимся по всем отобранным данным
while (cursor.moveToNext()){
     // забираем из каждой записи значения по столбцам
     val value1 = cursor.getString(getString(cursor.getColumnIndex("column1")).toInt())
     val value2 = cursor.getInt(getString(cursor.getColumnIndex("column2")).toInt())
     //...
}
```

Для нашей задачи можно написать функцию, которая будет возвращать все записи в виде списка из элементов Result

```
fun getAll(order: String): List<Result> {
    val allRecords = mutableListOf<Result>()
    val cursor = readableDatabase.query(DBContract.Entry.TABLE_NAME, null, null,
        null, null,null, order)
    cursor.moveToFirst()
    while(cursor.moveToNext()){
        allRecords.add(Result(cursor.getString(cursor.getColumnIndex(DBContract.Entry.COLUMN_NAME_NAME)),
                        cursor.getInt(cursor.getColumnIndex(DBContract.Entry.COLUMN_NAME_RESULT))))
    }
    return allRecords;
}
```

Приведем код DB-helper'а полностью

```
package study.android.kotlindbnotes

import android.content.ContentValues
import android.content.Context
import android.database.sqlite.SQLiteDatabase
import android.database.sqlite.SQLiteOpenHelper
import android.provider.BaseColumns

class SimpleDBHelper(context: Context) : SQLiteOpenHelper(context, DATABASE_NAME, null, DATABASE_VERSION) {

    object DBContract {
        // Группируем данные
  object Entry : BaseColumns {
            const val TABLE_NAME =  "results"
  const val COLUMN_NAME_NAME =  "name"
  const val COLUMN_NAME_RESULT = "result"
  }
        // Команды для работы
  const val SQL_CREATE =
            "CREATE TABLE ${Entry.TABLE_NAME} (" +
                    "${BaseColumns._ID} INTEGER PRIMARY KEY," +
                    "${Entry.COLUMN_NAME_NAME} TEXT," +
                    "${Entry.COLUMN_NAME_RESULT} INTEGER)"  const val SQL_DELETE = "DROP TABLE IF EXISTS ${Entry.TABLE_NAME}"
  }

    override fun onCreate(db: SQLiteDatabase) {
        db.execSQL(DBContract.SQL_CREATE)
    }
    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
        onCreate(db)
    }
    override fun onDowngrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
        onUpgrade(db, oldVersion, newVersion)
    }
    companion object {
  const val DATABASE_VERSION = 1
  const val DATABASE_NAME = "results.db"
  }

    fun insert(result: Result){
        // Открываем базу на запись
  val db = writableDatabase
  // Комплектуем данные для вставки
  val values = ContentValues().apply {
  put(DBContract.Entry.COLUMN_NAME_NAME, result.name)
            put(DBContract.Entry.COLUMN_NAME_RESULT, result.result)
        }
  // вставляем данные в базу данных
  db?.insert(DBContract.Entry.TABLE_NAME, null, values)
    }

     fun getAll(order: String): List<Result> {
        val allRecords = mutableListOf<Result>()
        val cursor = readableDatabase.query(DBContract.Entry.TABLE_NAME, null, null,
            null, null,null, order)
        cursor.moveToFirst()
        while(cursor.moveToNext()){
            allRecords.add(Result(cursor.getString(cursor.getColumnIndex(DBContract.Entry.COLUMN_NAME_NAME)),
                        cursor.getInt(cursor.getColumnIndex(DBContract.Entry.COLUMN_NAME_RESULT))))
        }
        return allRecords;
    }
}
```

Добавляем интерфейс

Здесь мы будем использовать самый простой интерфейс для того, чтобы сфокусировать внимание именно на реализации работы с базой данных.

Просто приведем файл разметки

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  tools:context=".MainActivity">

    <LinearLayout
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical">

        <EditText
  android:id="@+id/editTextName"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_margin="10dp"
  android:padding="8dp" />

        <EditText
  android:id="@+id/editTextResult"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_margin="10dp"
  android:autofillHints="Age"
  android:inputType="number"
  android:padding="8dp"
  android:textColor="@android:color/background_dark" />

        <Button
  android:id="@+id/btnInsert"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_margin="10dp"
  android:padding="8dp"
  android:text="Add data" />
    </LinearLayout>

    <LinearLayout
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:gravity="center"
  android:orientation="horizontal"
  android:weightSum="3">

        <Button
  android:id="@+id/btnRead"
  android:layout_width="0dp"
  android:layout_height="wrap_content"
  android:layout_margin="10dp"
  android:layout_weight="1"
  android:padding="8dp"
  android:text="Read" />
    </LinearLayout>

    <ScrollView
  android:layout_width="match_parent"
  android:layout_height="match_parent">

        <TextView
  android:id="@+id/tvResult"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:padding="8dp"
  android:textSize="16sp"
  android:textStyle="bold" />
    </ScrollView>
</LinearLayout>
```

и код активности

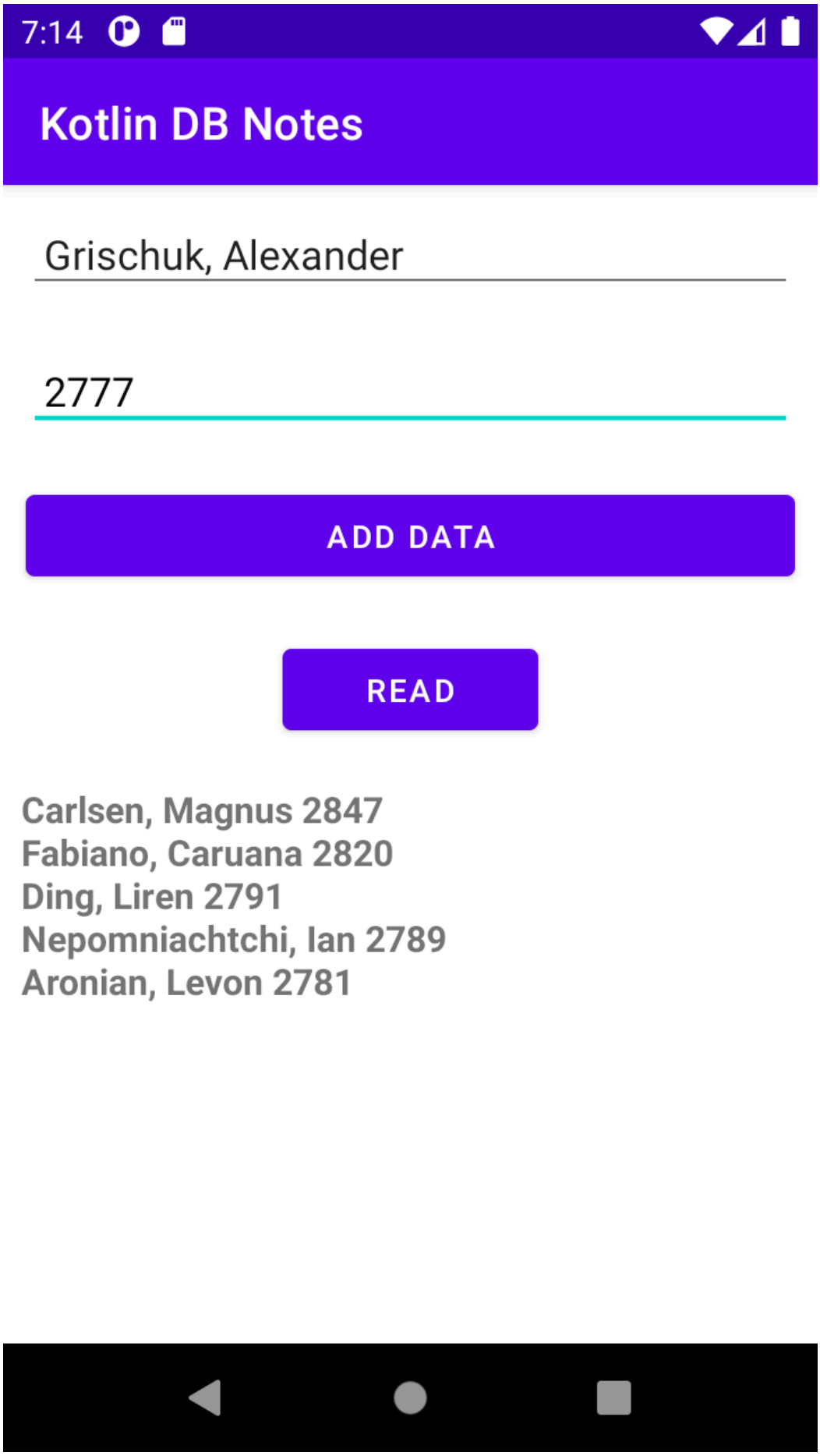
```
package study.android.kotlindbnotes

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.Toast
import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val context = this
 val dbHelper = SimpleDBHelper(context)
        btnInsert.setOnClickListener {
  if (editTextName.text.toString().isNotEmpty() &&
                editTextResult.text.toString().isNotEmpty()
            ) {
                val result =
                    Result(editTextName.text.toString(), editTextResult.text.toString().toInt())
                dbHelper.insert(result)
                clearFields()
            } else {
                Toast.makeText(context, "Please Fill All Data's", Toast.LENGTH_SHORT).show()
            }
        }
  btnRead.setOnClickListener {
  val data = dbHelper.getAll("RESULT DESC")
            tvResult.text = ""
  for (d in data) {
                tvResult.append("${d.name} ${d.result}\n")
            }
        }
  }

    private fun clearFields() {
        editTextName.text.clear()
        editTextResult.text.clear()
    }
}
```

Учебное приложение готово!



Полный код приложения можно посмотреть на гитхабе https://github.com/vv73/Kotlin_DB_Notes.

Начать тур для пользователя на этой странице