**TRƯỜNG ĐẠI HỌC BÁCH KHOA**

**KHOA ĐIỆN TỬ VIỄN THÔNG**

**BÁO CÁO**

**LẬP TRÌNH ĐA NỀN TẢNG**

**SQLite và Drift - Quản lý cơ sở dữ liệu quan hệ**

**Sinh viên thực hiện:**

**01. Nguyễn Trung Thành Lớp: 22KTMT2**  **MSSV:**  106220270

**02. Văn Tiến Lộc**  **Lớp: 22KTMT2**  **MSSV:**  106270257

**Người hướng dẫn:**

**TS. Nguyễn Duy Nhật Viễn**

**Đà Nẵng, 2025.**

**THUYẾT MINH**

**BÁO CÁO**

**LẬP TRÌNH ĐA NỀN TẢNG**

**SQLite và Drift - Quản lý cơ sở dữ liệu quan hệ**

BẢNG PHÂN CÔNG CÔNG VIỆC TRONG NHÓM

| STT | HỌ VÀ TÊN | NHIỆM VỤ | KHỐI LƯỢNG |
| --- | --- | --- | --- |
| 01 | NGUYỄN TRUNG THÀNH | Code, tìm nội dung  làm powerpoint | 50% |
| 02 | VĂN TIẾN LỘC | Code, tìm nội dung  Làm word | 50% |

 Link code github: [http://github/](https://github.com/Cherikyatto/SQLite-and-Drift-.git)

**Mục lục**

[**1. Tổng quan về SQLite và Drift 1**](#_pwdi1w41ldzt)

[**a. SQLite 1**](#_d89q83azhs68)

[**b. Drift 1**](#_x5xhsb1dwgo7)

[**2. Ứng dụng quản lý sản phẩm sử dụng SQLite 2**](#_dm6e0ooqoz7b)

[**a. Giới thiệu 2**](#_9fy20zf2evjm)

[**b. Các chức năng chính 2**](#_xhhjbxcp7lwk)

[**c. Code demo 2**](#_s6j7ybvvzoux)

[**d. Demo giao diện ứng dụng 5**](#_kiwql0gmpcwc)

[**3. Migrate sang Drift ORM 7**](#_s6kkxclzglr)

[**a. Cấu trúc dữ liệu trong drift 7**](#_4d7qfbc82lou)

[**b. Lớp điều khiển cơ sở dữ liệu (Database Layer) 7**](#_c4y42o8uklhs)

[**c. Thêm và cập nhật dữ liệu 8**](#_756zmf95dnio)

[**4. So sánh với SQLite 8**](#_vafyk7c9sn5x)

[**5. Truy vấn phức tạp và relationship. 9**](#_aox8phtlk1c8)

[**a. Lý thuyết chung 9**](#_vvrvhmp4ios1)

[**b. Truy vấn Join và lọc dữ liệu 9**](#_cx6nx3r019y8)

[**c. Xoá dữ liệu có quan hệ 10**](#_9zpavqq09g9t)

[**d. Demo các truy vấn phức tạp và relationship trong ứng dụng 11**](#_php4sav12h2k)

[**6. Kiểm thử hiệu năng và tối ưu hoá. 12**](#_m4ncr4jlgslh)

[**a. Mục tiêu kiểm thử. 12**](#_ii6bkmgb67td)

[**b. Code kiểm thử hiệu năng 12**](#_rsk3hpcelbh2)

[**c. Kết quả 14**](#_eqipb7pf5r4g)

[**d. Nhận xét 15**](#_f2wjeqbgrt0i)

[**e. Tối ưu hóa 15**](#_baz49bc9pvr)

[**7. Kết luận 16**](#_xi1umy9xzlls)

**NỘI DUNG**

# Tổng quan về SQLite và Drift

# SQLite

* SQLite là một hệ quản trị cơ sở dữ liệu quan hệ *(RDBMS)* nhúng, nổi bật với đặc điểm nhẹ, không cần server, dễ tích hợp vào các ứng dụng di động và máy tính để bàn. SQLite lưu trữ toàn bộ cơ sở dữ liệu trong một tệp đơn, giúp việc triển khai và phân phối trở nên đơn giản, đồng thời giảm thiểu chi phí quản lý.
* Một số đặc điểm chính của SQLite:
* Nhúng trực tiếp vào ứng dụng: Không cần cài đặt server riêng, giúp ứng dụng chạy độc lập.
* Hỗ trợ SQL chuẩn: Hầu hết các truy vấn SQL cơ bản như SELECT, JOIN, INSERT, UPDATE, DELETE đều được hỗ trợ.
* Nhẹ và nhanh: SQLite rất nhẹ, dung lượng chỉ khoảng vài MB và hiệu năng cao cho các ứng dụng với số lượng dữ liệu vừa và nhỏ.
* Giao dịch ACID: Hỗ trợ tính toàn vẹn dữ liệu thông qua các giao dịch Atomic, Consistent, Isolated, Durable.
* Phù hợp với Flutter/Android/iOS: Trong Flutter, thư viện **”sqflite”** giúp thao tác SQLite dễ dàng, gồm các chức năng tạo bảng, truy vấn, cập nhật và xóa dữ liệu.

=> Tuy SQLite tiện lợi, nhưng việc thao tác với cơ sở dữ liệu phức tạp thường đòi hỏi viết nhiều mã SQL thủ công, điều này có thể dẫn đến khó bảo trì và dễ gây nhầm lẫn trong các ứng dụng lớn.

# Drift

* Drift *(trước đây là Moor)* là một ORM *(Object Relational Mapping)* cho Flutter/Dart, giúp giao tiếp với SQLite một cách an toàn và tiện lợi hơn. Drift tự động sinh mã từ table definition, hỗ trợ type-safe query, giảm thiểu lỗi khi thao tác cơ sở dữ liệu.
* Một số điểm nổi bật của Drift:
* Type-safe: Truy vấn được kiểm tra kiểu dữ liệu ngay trong thời gian biên dịch, giảm lỗi runtime.
* Tích hợp tốt với Flutter: Drift hỗ trợ Streams, giúp UI tự động cập nhật khi dữ liệu có thay đổi.
* Hỗ trợ migration: Drift quản lý version database và migration dễ dàng, giúp nâng cấp cơ sở dữ liệu an toàn.
* Tối ưu cho query phức tạp: Drift hỗ trợ join, group by, aggregate và các query phức tạp mà vẫn giữ type safety.
* Tự sinh code: Chỉ cần định nghĩa các bảng và query trong Dart, Drift sẽ sinh code thực thi SQLite tương ứng.

=> Drift giúp giảm thiểu mã SQL thủ công, nâng cao tính maintainable cho dự án Flutter, đặc biệt là các dự án có cơ sở dữ liệu phức tạp và nhiều bảng.

# Ứng dụng quản lý sản phẩm sử dụng SQLite

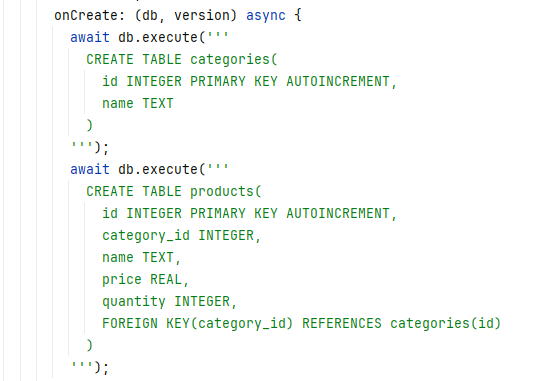
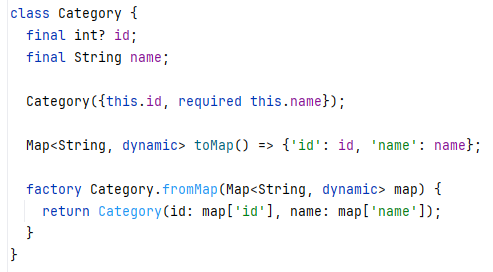
# Giới thiệu

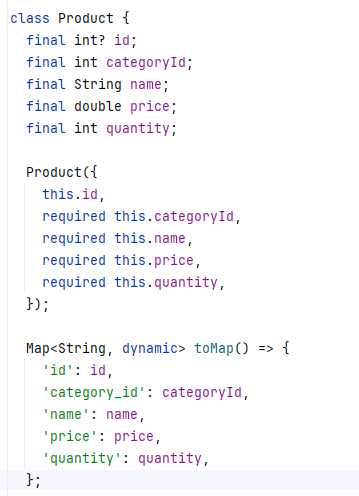
* Dựa trên cơ sở lý thuyết về SQLite, xây dựng một ứng dụng quản lý sản phẩm (Product Management App) nhằm lưu trữ là hợp lý nhất, hiển thị và thao tác dữ liệu cục bộ ngay trên thiết bị di động.
* Ứng dụng được lập trình bằng Flutter, sử dụng thư viện sqflite để kết nối và thao tác trực tiếp với cơ sở dữ liệu SQLite.
* Mục tiêu của ứng dụng là giúp người dùng:
* Quản lý danh mục sản phẩm và loại hàng (Category, Product);
* Thực hiện các thao tác CRUD (Create, Read, Update, Delete);
* Thử nghiệm hiệu năng ghi dữ liệu hàng loạt (batch insert).

# Các chức năng chính

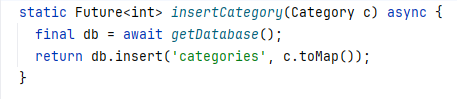
* Ứng dụng cung cấp các chức năng quản lý sản phẩm cơ bản:
* Thêm danh mục (Add Category)
  + Người dùng nhập tên danh mục mới qua hộp thoại, sau đó thông tin được lưu vào bảng categories.
* Thêm hoặc chỉnh sửa sản phẩm (Add/Edit Product)
  + Khi thêm mới, người dùng chọn danh mục, nhập tên, giá và số lượng.
  + Khi chỉnh sửa, ứng dụng tự động hiển thị dữ liệu cũ để người dùng cập nhật.
  + Các giá trị số (như giá bán) được xử lý hiển thị thông minh.
* Xóa sản phẩm (Delete Product)
  + Ứng dụng hiển thị hộp thoại xác nhận trước khi xóa sản phẩm khỏi bảng products.
* Xóa danh mục (Delete Category)
  + Khi xóa một danh mục, toàn bộ sản phẩm thuộc danh mục đó cũng bị xóa để đảm bảo toàn vẹn dữ liệu.
* Tìm kiếm (Search)
  + Thanh tìm kiếm cho phép lọc sản phẩm theo tên hoặc tên danh mục.

# Code demo

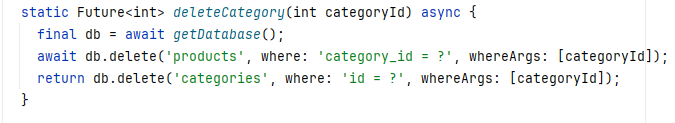
* Thiết kế cơ sở dữ liệu.
* Khởi tạo 2 bảng trong hàm onCreate() của SQLiteService như sau:
* Trong đó:
* **id** là khóa chính tự tăng.
* **category\_id** là khóa ngoại, liên kết mỗi sản phẩm với một danh mục.
* **price** và **quantity** dùng kiểu REAL và INTEGER để lưu giá và số lượng.
* Mô hình dữ liệu
* Trong ứng dụng, hai lớp dữ liệu được định nghĩa là Category và Product:



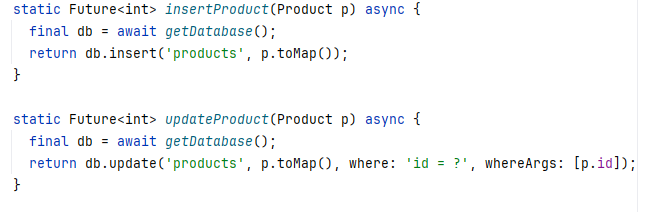
* Các lớp này giúp ánh xạ dữ liệu giữa Flutter và SQLite dễ dàng, thuận tiện cho thao tác CRUD.
* Các chức năng quản lý
* Thêm mới danh mục



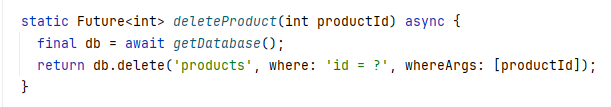
* Xoá danh mục



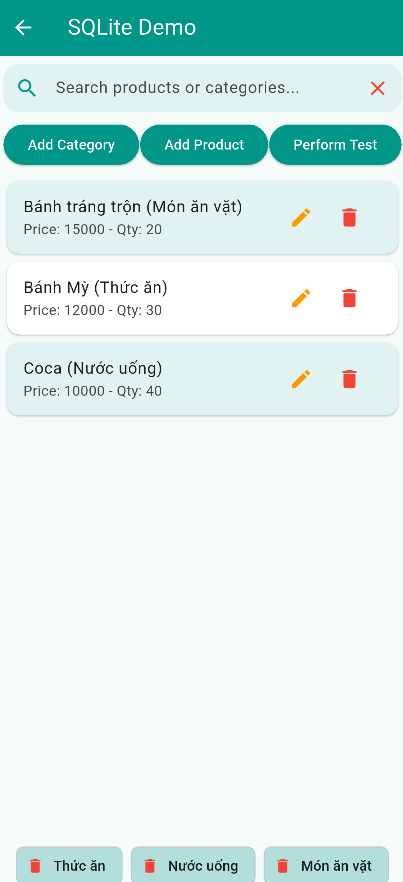
* Thêm và sửa sản phẩm
  + Tuỳ vào việc đối tượng có id hay không, ứng dụng sẽ gọi insertProduct() hoặc updateProduct()

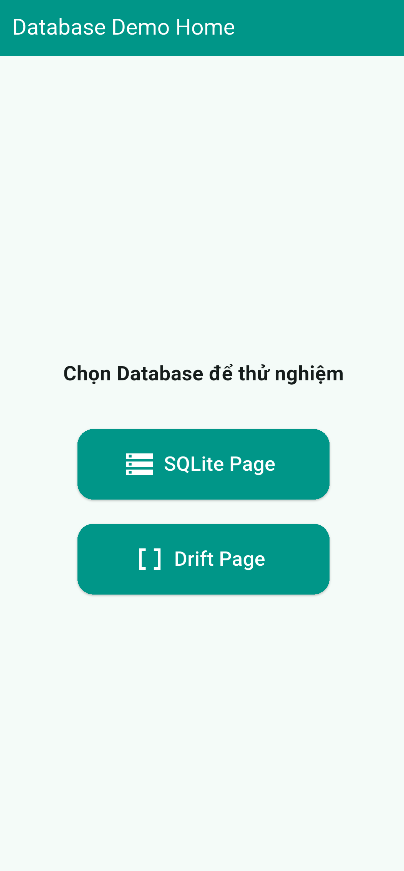


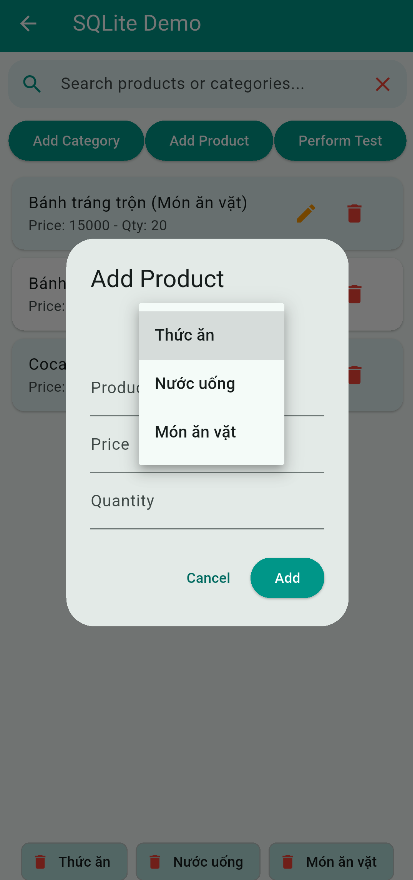
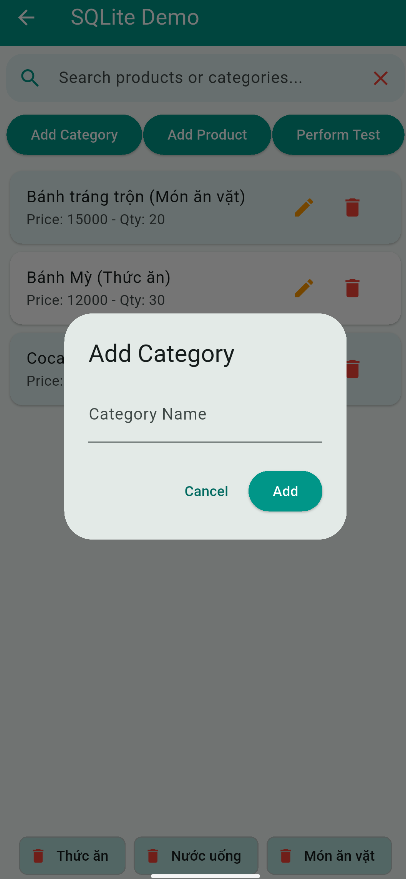
* Xoá sản phẩm



# Demo giao diện ứng dụng

* Trang chủ lựa chọn giữa SQLite, Drift và giao diện SQLite Demo

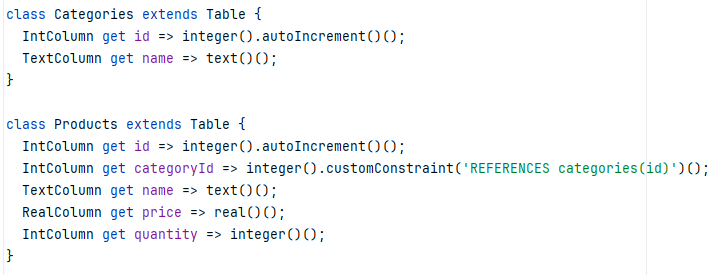




# Migrate sang Drift ORM

# Cấu trúc dữ liệu trong drift

* Thay vì tạo bảng bằng câu lệnh CREATE TABLE, Drift cho phép khai báo bảng trực tiếp bằng class Dart kế thừa từ Table.



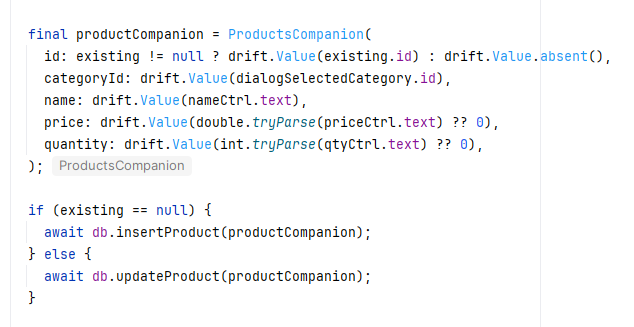
* Drift sẽ tự động sinh mã SQL tương ứng trong file .g.dart, đảm bảo đồng bộ giữa code Dart và cơ sở dữ liệu

# Lớp điều khiển cơ sở dữ liệu (Database Layer)

* Thay vì tự mở kết nối SQLite và viết lệnh thủ công, Drift cung cấp lớp AppDatabase kế thừa \_$AppDatabase, giúp gom toàn bộ CRUD vào một nơi duy nhất.

# Thêm và cập nhật dữ liệu

* Khi chuyển sang Drift, phần giao diện chỉ cần gọi các hàm CRUD trong AppDatabase. Việc thêm hoặc cập nhật sản phẩm trở nên đơn giản và an toàn hơn. Thay vì tự viết INSERT hay UPDATE, Drift cho phép truyền dữ liệu bằng Companion.Trình biên dịch sẽ kiểm tra kiểu dữ liệu, đảm bảo đúng cột và kiểu.



# So sánh với SQLite

| **Tiêu chí** | **SQLite thuần** | **Drift ORM** |
| --- | --- | --- |
| **Cách viết truy vấn** | Viết bằng SQL thủ công | Viết bằng Dart (an toàn cú pháp, dễ maintain) |
| **Tự động sinh code** | Không có | Có (sử dụng build\_runner) |
| **Type safety** | Không đảm bảo | Có đảm bảo chặt chẽ |
| **Khả năng mở rộng** | Khó bảo trì khi bảng phức tạp | Dễ mở rộng, hỗ trợ relationships |
| **Hiệu năng** | Tốt | Gần tương đương, có tối ưu hóa tự động |
| **Độ phức tạp cài đặt** | Dễ (ít phụ thuộc) | Cần setup thêm build\_runner và code gen |
| **Phù hợp với dự án nhỏ** | Rất phù hợp | Có thể hơi nặng nề |
| **Phù hợp với dự án lớn** | Khó maintain | Rất phù hợp |

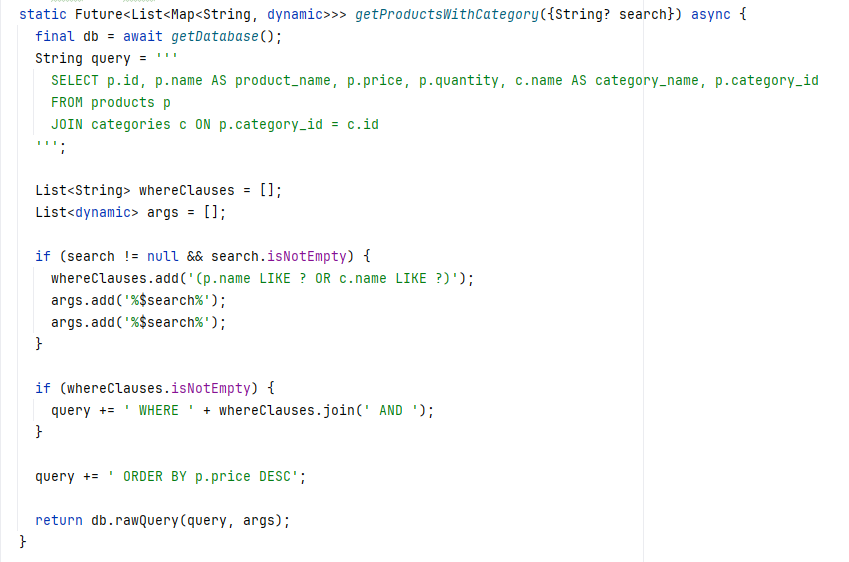
# Truy vấn phức tạp và relationship.

# Lý thuyết chung

* Trong các ứng dụng quản lý dữ liệu, đặc biệt là quan hệ one-to-many như Danh mục - Sản phẩm, việc xử lý truy vấn phức tạp đóng vai trò quan trọng để:
* Hiển thị dữ liệu có liên kết (ví dụ: sản phẩm kèm tên danh mục).
* Lọc và sắp xếp dữ liệu theo điều kiện (tên, giá, loại).
* Cập nhật, xoá dữ liệu có ràng buộc quan hệ.
* Quan hệ one-to-many trong hệ quản trị cơ sở dữ liệu
* Một danh mục (Category) có thể chứa nhiều sản phẩm (Product).
* Mỗi sản phẩm có một khóa ngoại **category\_id** trỏ tới **categories.id.**
* Mối quan hệ này được thể hiện bằng cú pháp

**FOREIGN KEY(category\_id) REFERENCES categories(id)**

# Truy vấn Join và lọc dữ liệu

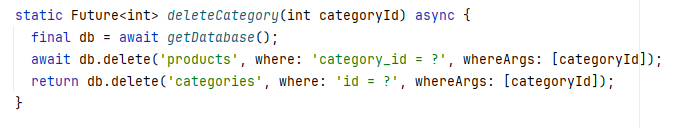
* Đây là cách viết “truy vấn phức tạp” trong SQLite thuần: linh hoạt nhưng dễ bị lỗi cú pháp và khó bảo trì khi ứng dụng lớn.
* Trong đó
* **JOIN categories c ON p.category\_id = c.id:** kết hợp 2 bảng theo khóa ngoại.
* **LIKE '%$search%'**: lọc sản phẩm hoặc danh mục theo từ khóa.
* **ORDER BY p.price DESC:** sắp xếp theo giá giảm dần.
* So với SQLite thuần, Drift giúp viết truy vấn phức tạp rõ ràng hơn, giảm lỗi cú pháp và dễ bảo trì



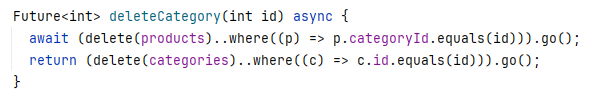
* Trong đó:
* **join([...]):** định nghĩa quan hệ giữa 2 bảng bằng cú pháp Dart.
* **where(...):** thêm điều kiện lọc tương tự SQL nhưng type-safe.
* **readTable(...):** đọc dữ liệu từng bảng trong kết quả JOIN.

# Xoá dữ liệu có quan hệ

* SQLite:

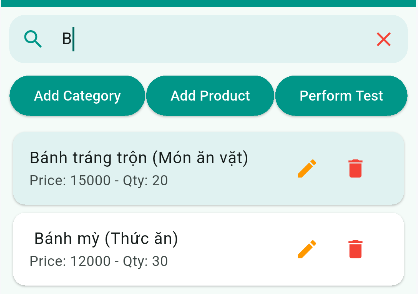


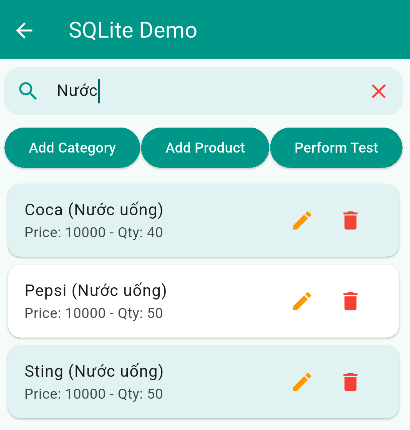
* Drift:

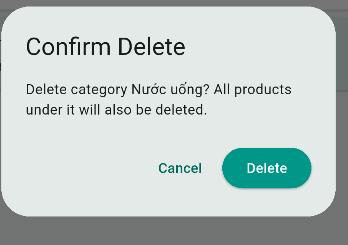


* Ở đây quan hệ cha-con được thiết lập giữa categories-products, khi xoá categories, các product thuộc category đó cũng bị xoá theo. Đối với SQLite, chức năng xoá phải viết thủ công bằng các câu lệnh SQL, trong khi Drift làm bằng ORM type-safe nên dễ bảo trì và ít lỗi hơn.

# Demo các truy vấn phức tạp và relationship trong ứng dụng

* Thanh tìm kiếm có thể tìm các sản phẩm theo tên hoặc loại sản phẩm  
  



* Khi xoá 1 danh mục thì có thể xoá tất cả sản phẩm bên trong nó  
  

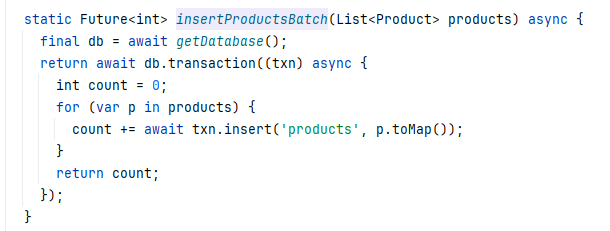
# Kiểm thử hiệu năng và tối ưu hoá.

# Mục tiêu kiểm thử.

* Đánh giá tốc độ ghi dữ liệu (insert) vào cơ sở dữ liệu SQLite.
* So sánh hiệu năng giữa ghi tuần tự từng bản ghi và ghi hàng loạt trong một transaction.
* Kiểm tra khả năng tối ưu hoá thao tác I/O trong SQLite để phục vụ cho khối lượng dữ liệu lớn.

# Code kiểm thử hiệu năng

* Đối với SQLite
* Tạo dữ liệu mẫu:
* Ứng dụng sinh ra 1000 sản phẩm ngẫu nhiên bằng List.generate().  
   Mỗi sản phẩm thuộc cùng một danh mục thử nghiệm (TEST).
* Thử nghiệm 1: Ghi tuần tự:  
  
* Mỗi sản phẩm được ghi riêng biệt, mỗi lần ghi là một transaction riêng.
* Cách này an toàn nhưng chậm, vì SQLite phải mở và đóng transaction 1000 lần.
* Thử nghiệm 2 : Ghi hàng loạt trong một transaction:
* Trong hàm insertProductsBatch, tất cả các bản ghi được chèn trong cùng một transaction:



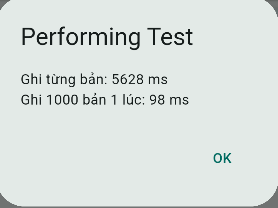
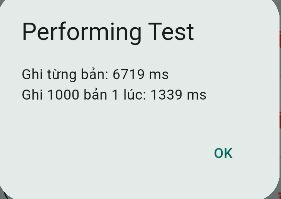
* SQLite chỉ commit 1 lần, giảm đáng kể thời gian I/O đĩa.
* Đối với Drift



# Kết quả

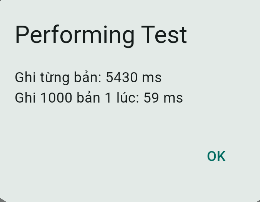
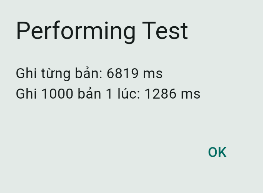
* Lần 1:

SQLite Drift



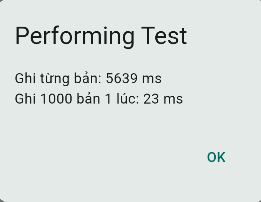
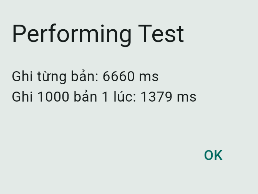
* Lần 2

SQLite Drift



* Lần 3

SQLite Drift



# Nhận xét

* Khi ghi từng bản:
* SQLite thuần
  + Mỗi lần gọi db.insert() sẽ thực hiện đầy đủ các bước: mở kết nối, biên dịch câu lệnh SQL, bắt đầu và kết thúc giao dịch, ghi dữ liệu rồi đóng transaction. Việc lặp lại chu trình này 1000 lần làm tăng đáng kể thời gian I/O và chi phí khởi tạo.
* Drift:
  + Drift thực hiện logic tương tự SQLite nhưng đã tối ưu câu lệnh SQL ở compile-time và tái sử dụng kết nối. Tuy nhiên, vì mỗi bản ghi vẫn là một transaction riêng, hiệu suất chỉ cải thiện nhẹ hoặc có thể là không quá khá biệt so với SQLite thuần
* Thời gian thực thi của SQLite thuần và Drift có tỉ lệ chênh lệch không đáng kể, tuy nhiên con số nhận có thể bị ảnh hưởng bởi nhiều nguyên nhân khách quan khác(Thiết bị, môi trường thực thi,...)
* Khi ghi 1.000 bản trong một lần
* Kết quả cho thấy sự chênh lệch cực kỳ lớn.
* SQLite thuần: Dữ liệu được ghi bằng vòng lặp for và mỗi lần chèn vẫn là một transaction độc lập, dẫn đến việc mỗi thao tác ghi đều phải mở và đóng một giao dịch. Gây tốn thời gian xử lý, đặc biệt với ổ đĩa thật hoặc bộ nhớ flash.
* Drift ORM: Drift cung cấp phương thức batch cho phép gom nhiều thao tác ghi vào cùng một transaction. Toàn bộ 1000 bản ghi được ghi xuống cơ sở dữ liệu chỉ trong một giao dịch duy nhất.
* Kết luận: Khi ghi hàng loạt, Drift thể hiện rõ ưu thế vượt trội nhờ cơ chế gom transaction, giảm overhead và tối ưu hoá luồng xử lý dữ liệu, còn ghi từng bản thì cả 2 tương đương nhau, tuy nhiên còn phụ thuộc khác nhiều những yếu tố khác.

# Tối ưu hóa

* Sử dụng giao dịch (Transaction) hợp lý: Khi thực hiện nhiều thao tác ghi hoặc xoá liên tiếp , gom chúng vào cùng một transaction sẽ giúp giảm đáng kể thời gian xử lý.
* Thiết lập quan hệ (Relationship) và ràng buộc khóa ngoại (Foreign key) giúp dữ liệu đồng bộ và tránh lỗi rác.
* Tận dụng Batch, Prepared Statement và Cache:
* Batch: gom nhiều câu lệnh SQL lại thực hiện một lần (Drift hỗ trợ mặc định).
* Prepared Statement: Drift tự động chuẩn bị và tái sử dụng câu lệnh SQL, giảm thời gian biên dịch truy vấn.
* Cache: khi dữ liệu hiển thị lặp lại nhiều, có thể lưu tạm trong bộ nhớ (state hoặc provider) để giảm truy vấn không cần thiết.
* Phân tách luồng và tránh chặn giao diện:
* SQLite và Drift đều là tác vụ I/O nặng.
* Nên thực hiện các truy vấn trong Isolate hoặc Future async để không làm “đơ” giao diện người dùng (UI thread).
* Drift hỗ trợ sẵn điều này khi chạy trên nền Flutter thông qua DriftIsolate.
* Kiểm thử và đo đạc thường xuyên.

# Kết luận

* Ứng dụng quản lý sản phẩm được xây dựng với SQLite thuần giúp nắm rõ quy trình thao tác dữ liệu từ khâu tạo bảng, truy vấn, đến cập nhật. Cách tiếp cận này phù hợp để hiểu sâu cơ chế hoạt động của cơ sở dữ liệu, nhưng đòi hỏi viết nhiều mã SQL thủ công, khó mở rộng khi ứng dụng lớn dần.
* Khi migrate sang Drift ORM, việc tổ chức cơ sở dữ liệu trở nên chặt chẽ, rõ ràng và an toàn hơn. Drift hỗ trợ tự động sinh code, type-safety, và reactive query, giúp rút ngắn thời gian phát triển, đồng thời giảm lỗi do truy vấn sai cú pháp hoặc sai kiểu dữ liệu.
* Trong phần truy vấn phức tạp và quản lý mối quan hệ (relationships), Drift tỏ ra vượt trội nhờ khả năng kết hợp giữa cú pháp SQL và biểu thức Dart. Việc thực hiện JOIN, filter, sort, và liên kết nhiều bảng trở nên ngắn gọn và dễ bảo trì hơn nhiều so với SQLite thuần.
* Ở khía cạnh hiệu năng và tối ưu hóa, kết quả thử nghiệm cho thấy SQLite thuần có tốc độ xử lý tốt trong các thao tác đơn giản nhờ truy cập trực tiếp. Tuy nhiên, Drift có khả năng tối ưu tự động, caching và quản lý kết nối tốt hơn, nên khi khối lượng dữ liệu tăng, hiệu năng ổn định và nhất quán hơn.
* Tổng thể, Drift là lựa chọn phù hợp cho ứng dụng Flutter tầm trung đến lớn, cần tính ổn định, bảo trì dài hạn và mở rộng trong tương lai, trong khi SQLite thuần vẫn hữu ích cho mục đích học tập, thử nghiệm hoặc dự án nhỏ, nhẹ.

**TÀI LIỆU THAM KHẢO**

[1]. <https://drift.simonbinder.eu>

[2].<https://www.geeksforgeeks.org/flutter/sqlite-in-flutter/>

[3].<https://r1n1os.medium.com/drift-local-database-for-flutter-part-1-intro-setup-and-migration-09a64d44f6df>

[4]. <https://docs.flutter.dev/cookbook/persistence/sqlite>

[5]. <https://developer.android.com/topic/performance/sqlite-performance-best-practices>

[6]. <https://pub.dev/packages/sqflite>

….

***Chú ý:***

Phải tuân thủ format của tài liệu tham khảo.

Nếu có tham khảo các thuyết minh, phải trích dẫn và thể hiện cụ thể trong các tài liệu tham khảo. Code của khóa trước phải upload đầy đủ code cũ lẫn code mới, trong thuyết minh phải chỉ ra được cái mới của code nhóm thực hiện. Nếu phát hiện sao chép mà không trích dẫn, không có trong mục tài liệu tham khảo thì toàn nhóm sẽ bị điểm 0.