20 · Database

Flutter Practice Notes

Contents

20 · Database

Introduction

When data is structured, queryable, relational, or large, you need a real on-device **database** — the backing store for offline-first and rich caching. This module covers choosing SQL vs NoSQL and the main Flutter options: SQLite/ `sqflite`, Drift (typed reactive SQL), and Isar/Hive (object stores), plus modeling, migrations, and performance.

Why this module exists

`SharedPreferences`/files can't query or scale (Module 15). Real apps need indexed queries, relations, reactive reads, and migrations. Choosing and using the right database — and modeling/migrating it well — determines correctness, performance, and maintainability of the whole data layer.

Module map

#	File	Topic	Level
1	01_database_options.md	SQL vs NoSQL; choosing an engine	🔵
2	02_sqlite_sqflite.md	Raw SQLite via `sqflite`	🔵
3	03_drift.md	Typed, reactive SQL (compile-safe)	🔴
4	04_isar_hive.md	NoSQL/object stores	🔵
5	05_modeling_migrations_performance.md	Schema, migrations, indexing, performance	🔴

> **Cross-references:** Storage decision framework: 15 · storage_options_overview. Offline-first (uses the DB as source of truth): Module 19. Repository boundary/DTO mapping: 05 · repository. Streams (reactive reads): 02 · streams. Isolates (heavy queries): 02 · isolates. Firestore (cloud NoSQL): 18 · firestore.

Prerequisites

15 Local Storage, 05 · repository, 02 · streams.

What you'll be able to do after this module

- Choose SQL vs NoSQL and a specific engine by data shape/needs.
- Use `sqflite` for raw SQLite; Drift for typed reactive SQL; Isar/Hive for object storage.

- Model schemas with relations/indexes and write safe migrations.
- Build reactive, repository-fronted data layers and optimize query performance.

Capstone

Typed reactive data layer: Model a small domain (users/tasks) in Drift with relations + indexes, expose reactive queries behind a repository (mapped to entities), write a v1→v2 migration, and benchmark an indexed vs unindexed query.

Summary

Databases give queryable, relational, reactive, migratable on-device storage. Pick SQL (relational/queries) or NoSQL (object/speed), front it behind repositories, model with indexes, and migrate carefully — the foundation beneath caching and offline-first.

Database Options (SQL vs NoSQL; Choosing an Engine)

> Pick by data shape and needs: **SQL** (SQLite/ `sqflite` /Drift) for relational, queryable, ACID data; **NoSQL object stores** (Isar/Hive) for fast schemaless/object persistence — and prefer a **typed, reactive** solution (Drift/Isar) so reads are compile-safe and stream updates.

Introduction

Flutter has several embedded databases with different models and ergonomics. This file frames the choice — SQL vs NoSQL, typed vs untyped, reactive vs one-shot — so the engine files (`sqflite` /Drift/Isar/Hive) are applied to the right problem.

Why this concept exists

The wrong engine causes pain: raw SQL strings everywhere (untyped, error-prone), NoSQL where you needed joins/aggregations, or a non-reactive DB in a reactive UI. A clear decision framework matches the engine to the data model and app requirements.

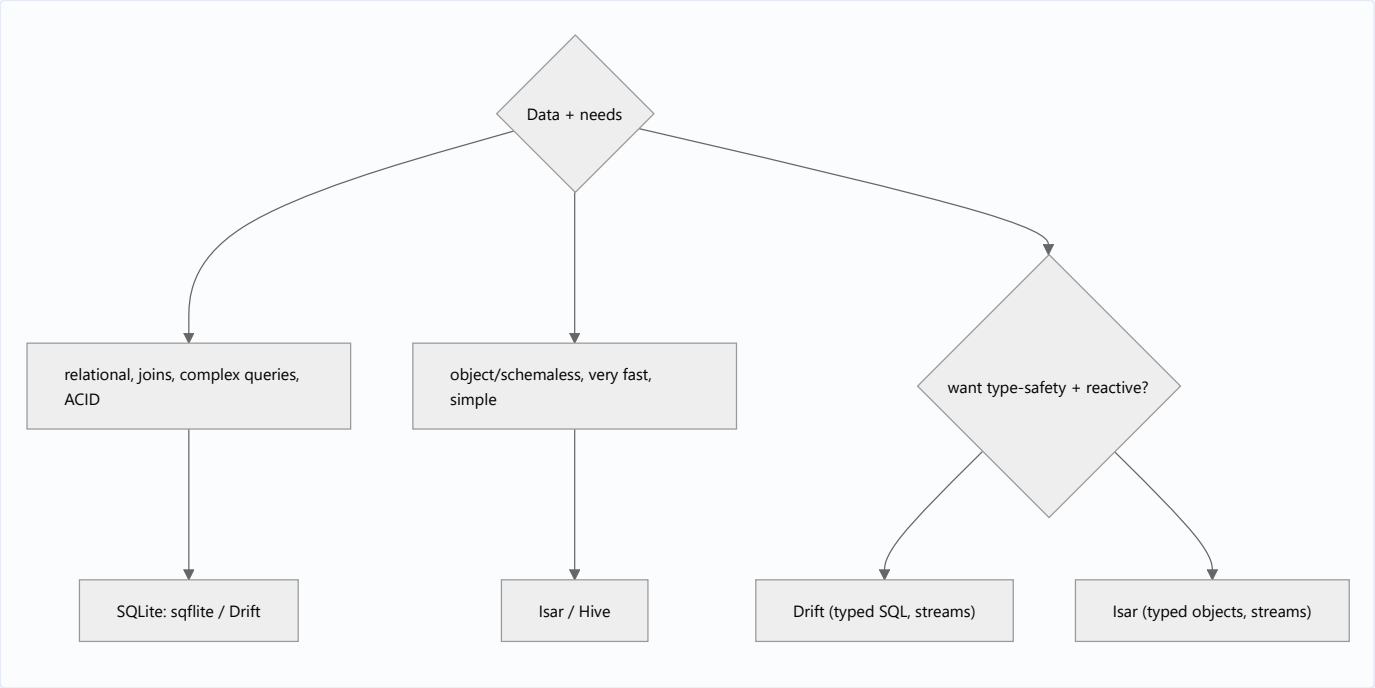
Real-world analogy

Choosing a database is choosing **how you organize a warehouse**: labeled shelves with a cross-reference index and strict rules (SQL — great for querying/relations) vs bins you toss labeled objects into and grab fast (NoSQL object store — great for speed/flexibility). Pick by how you'll store *and retrieve*.

Problem Statement

Your app needs: relational data with joins/reporting, plus a large set of objects read very fast, and reactive lists that auto-update. Which engine(s)? By the end you'll route each and pick defaults.

Internal Working



Engine	Model	Typed	Reactive	Query power	Notes
`sqflite`	SQL (SQLite)	❌ (raw strings/maps)	❌ (manual)	Full SQL	Battle-tested, low-level, verbose
Drift	SQL (SQLite)	✅ (codegen)	✅ (`watch`)	Full SQL + typed DSL	Recommended typed reactive SQL
Isar	NoSQL objects	✅ (codegen)	✅ (streams)	Rich indexed queries	Fast, typed object DB
Hive	NoSQL key-object	✅ (adapters)	⚠️ (box listeners)	Key/box lookups (limited query)	Very simple/fast; weak querying
ObjectBox	NoSQL objects	✅	✅	Indexed queries	Fast alternative to Isar

Decision heuristics:

- **Relational / joins / complex queries / reporting / ACID** → **SQL** (Drift preferred; `sqflite` if you want raw control).
- **Object-oriented / schemaless / speed-first / simple queries** → **NoSQL** (Isar for typed+queryable; Hive for simple key-object).
- **Want compile-safe queries + reactive streams** → **Drift** or **Isar** (avoid raw `sqflite` maps for large apps).
- **Just a few key-values / secrets / blobs** → not a DB — use prefs/secure/files (15).

Memory Representation

DBs store on disk; queries load result sets into memory (bound with `LIMIT`/pagination). Reactive engines re-emit on change (02 · streams, 02 · memory_and_gc).

Compiler Behavior

Typed engines (Drift/Isar) **codegen** query/row types → compile-safe queries; raw `sqflite` returns untyped `Map`s (runtime-cast, error-prone).

Runtime Behavior

Queries are async; typed engines validate at compile time; reactive engines push updates on writes. Heavy queries should run without blocking the UI isolate.

Flutter Engine Behavior

DB plugins cross the embedder to native SQLite/engine; some (Drift/Isar) can run on background isolates for heavy work (02 · isolates).

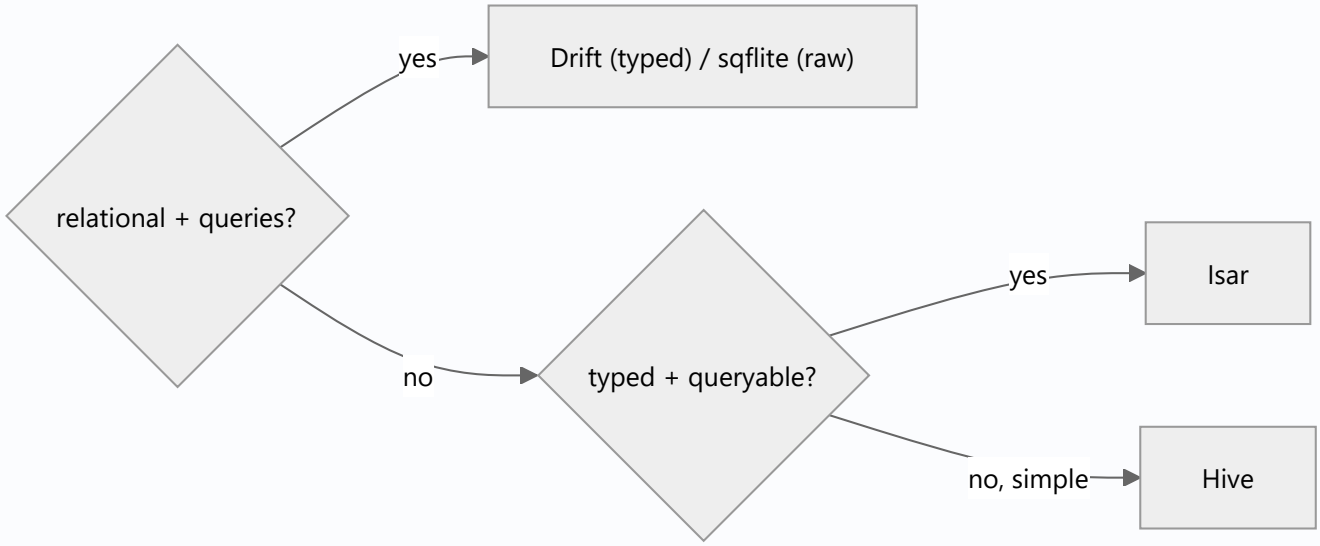
Dart VM Behavior

Not applicable beyond async.

Examples

```
// Routing data to an engine (mental model):
// - users + orders with joins/reports        -> SQL (Drift preferred, or sqflite)
// - 50k cached objects, very fast reads       -> Isar (typed) / ObjectBox
// - a few key->object records, simple lookup   -> Hive
// - reactive task list auto-updating the UI    -> Drift .watch() or Isar streams
// - a handful of settings / a token / a file   -> NOT a DB (prefs/secure/files, Module 15)
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Raw `sqflite` maps across a big app	Untyped, error-prone	Use Drift/Isar (typed)
NoSQL where you need joins/reports	Poor fit	Use SQL
Non-reactive DB in a reactive UI	Manual refresh, stale UI	Use `watch` /streams (Drift/Isar)
Using a DB for a few key-values	Overkill	Prefs/secure/files (15)
Leaking DB rows to the UI	Coupling	Map rows→entities in a repository
Unbounded queries	Memory/perf	`LIMIT` /pagination + indexes (05_modeling_migrations_performance.md)

Best Practices

- **Match engine to data**: SQL for relational/queryable/ACID; NoSQL objects for speed/flexibility.
- Prefer **typed + reactive** engines (**Drift** for SQL, **Isar** for NoSQL) for compile safety and auto-updating UIs.
- Front the DB behind a **repository** mapping rows/objects→entities; the app depends on the domain, not the DB.
- Bound queries (indexes, `LIMIT`, pagination); run heavy queries off the UI isolate.
- Don't use a DB for tiny key-values/secrets/blobs (use prefs/secure/files).

Performance

Indexed, bounded queries are fast; typed engines avoid parse/cast overhead; reactive streams re-run only affected queries. Heavy work off-isolate keeps the UI smooth (05_modeling_migrations_performance.md, Module 21).

Advantages / Disadvantages

- **+ SQL:** relations/joins/aggregations/ACID, mature. **+ NoSQL object:** speed, simple mapping, schemaless flexibility.
- **− SQL:** more setup/migrations, verbose (raw). **− NoSQL:** weaker ad-hoc querying/relations; per-engine lock-in.

Interview Questions

1. 🟢 **SQL vs NoSQL on-device — when each?** — SQL for relational/queryable/ACID data (joins, reports); NoSQL object stores for fast, schemaless/object persistence with simpler queries.
2. 🟢 **Why prefer Drift/Isar over raw `sqflite`?** — Typed, compile-safe queries and reactive streams — vs raw untyped `Map`s and manual refresh.
3. 🟡 **What does "reactive" mean for a DB?** — Queries return `Stream`s that re-emit when underlying data changes, auto-updating the UI (Drift `watch`, Isar streams).
4. 🟡 **When is Hive appropriate vs Isar?** — Hive for simple key-object storage/caches with minimal querying; Isar when you need typed objects + rich indexed queries.
5. 🟡 **When should you NOT use a database?** — For a few key-values, secrets, or blobs — use prefs/secure storage/files instead.
6. 🔴 **How do you keep the DB from coupling the app?** — Wrap it behind a repository that maps rows/objects to domain entities; the domain never sees DB types.
7. 🔴 **How do you bound query cost/memory?** — Indexes + `LIMIT`/pagination + selecting needed columns; heavy queries off the UI isolate.

Senior Engineer Tips

- Default to **Drift** for relational apps (typed + reactive SQL) and **Isar** for object-heavy/fast-read apps; reach for raw `sqflite` only when you need low-level control.
- Keep DB types out of the domain (repository mapping) so you can migrate engines and stay testable.
- Choose reactive from the start — retrofitting `watch()`/streams into a one-shot DB layer is invasive.

Architect Perspective

The database is the structured core of the data layer, underpinning caching (15) and offline-first (19). Choosing a typed, reactive engine behind repositories yields a compile-safe, auto-updating, swappable data foundation; the SQL-vs-NoSQL decision follows the data's relational/query needs and is costly to reverse — decide deliberately.

Summary

- Choose SQL (SQLite/`sqflite`/Drift) for relational/queryable/ACID; NoSQL (Isar/Hive) for object/speed.
- Prefer typed+reactive (Drift/Isar); front behind repositories; bound queries with indexes/pagination.
- Not for tiny key-values/secrets/blobs; the DB backs caching and offline-first.

Revision Notes

- SQL (relational/joins/ACID): `sqflite` (raw) / Drift (typed+reactive). NoSQL (objects/fast): Isar (typed+queryable) / Hive (simple key-object) / ObjectBox.
- Prefer typed + reactive (Drift/Isar); avoid raw maps at scale.
- Repository maps rows→entities; bound queries (index/LIMIT); heavy off-isolate.
- Tiny key-values/secrets/blobs → Module 15, not a DB.

Practice Questions

1. Route four data sets to SQL vs a specific NoSQL engine.
2. Why choose Drift over raw `sqflite` for a large app?
3. When is Hive enough vs needing Isar?

Coding Questions

1. Design a repository interface over an unspecified DB (rows→entities, reactive `watch`).
2. Justify SQL vs NoSQL for a described feature set.
3. Identify a DB misuse (tiny key-values) and re-route it to Module 15 storage.

Mini Project

Engine decision (docs + skeleton): For a described app (relational reports + a large fast-read object cache + a reactive list), write `DB_CHOICE.md` selecting engine(s) with rationale, and sketch a repository interface (reactive reads, entity mapping) that could sit over any of them. Acceptance:

correct SQL/NoSQL routing; typed+reactive preference justified; repository abstraction.

SQLite via `sqflite`

> `sqflite` is the battle-tested, low-level SQLite plugin: you open a database, run raw SQL (or helper methods), and get back untyped `Map`s — full SQL power and control, at the cost of manual typing, migrations, and no built-in reactivity.

Introduction

SQLite is an embedded relational database; `sqflite` (package) is Flutter's direct binding. This file covers opening a DB, CRUD via raw SQL / helpers, transactions, batch, and versioned migrations — plus why larger apps often move to Drift on top of it.

Why this concept exists

SQLite is ubiquitous, reliable, and file-based (no server). `sqflite` exposes it directly, giving full SQL and precise control. It's the foundation many higher-level tools (Drift) build on, and knowing it demystifies them.

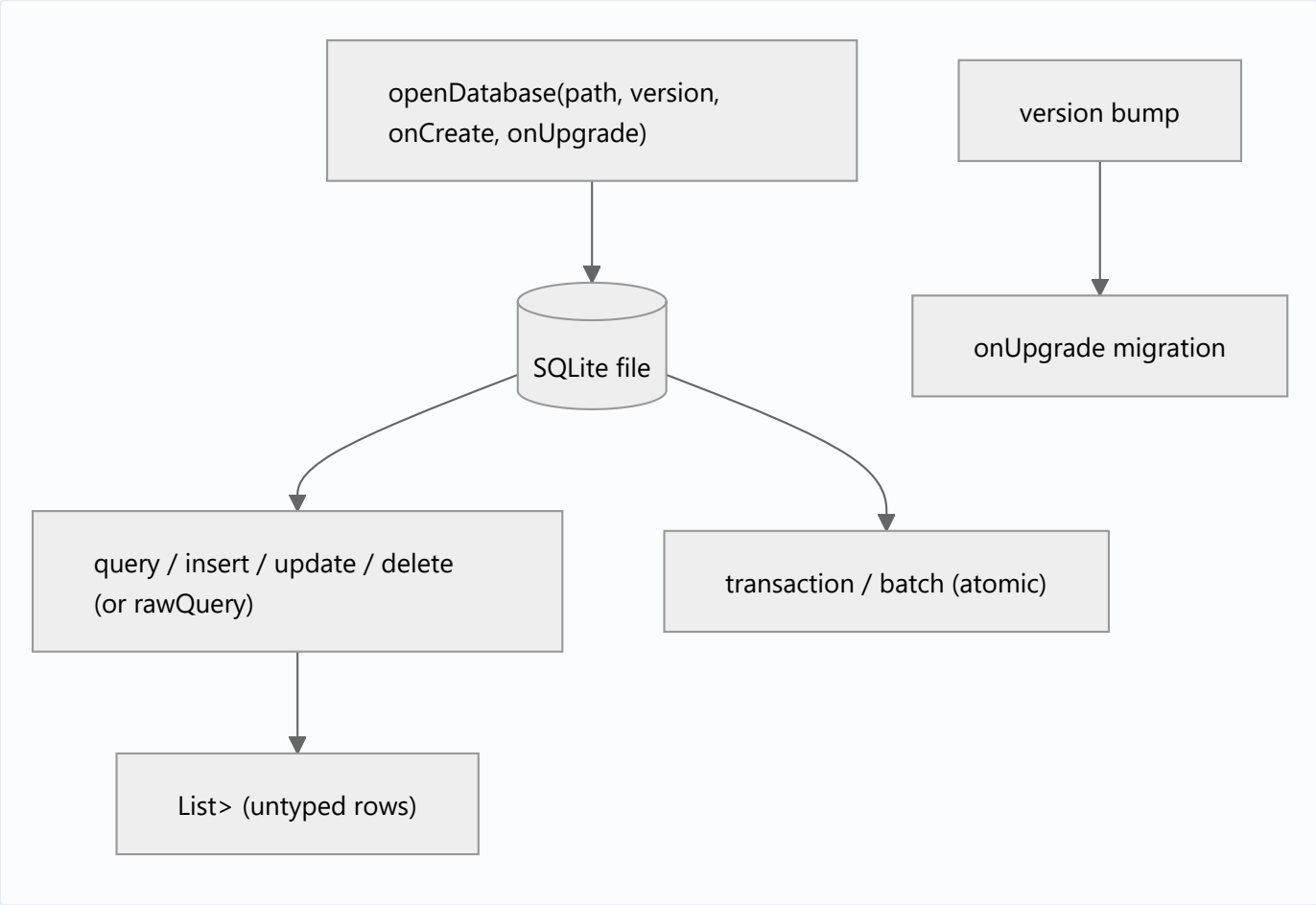
Real-world analogy

`sqflite` is **driving a manual-transmission car**: full control over the engine (SQL), but you work the clutch yourself (typing, migrations, refresh). Drift is the automatic on top — same engine, less manual labor.

Problem Statement

Persist tasks in SQLite: create the schema, insert/query/update/delete, use transactions, and handle a v1→v2 schema change — all with `sqflite`, wrapped in a repository. You'll write the raw SQL layer.

Internal Working



- **Open**: `openDatabase(path, version:, onCreate:, onUpgrade:, onConfigure:)` (path via `path_provider` / `getDatabasesPath`). `onCreate` builds the schema; `onUpgrade` migrates on version bumps (05_modeling_migrations_performance.md).
- **CRUD helpers**: `db.insert/query/update/delete` (map-based) or `db.rawQuery/rawInsert(...)` (raw SQL). Results are `List<Map<String, Object?>>` — untyped; you cast/map manually.
- **Transactions**: `db.transaction((txn) async { ... })` for atomic multi-statement writes; `db.batch()` for bulk operations.
- **Migrations**: bump `version`; in `onUpgrade(db, oldV, newV)` run `ALTER TABLE`/data migrations step-by-step.
- **No reactivity**: `sqflite` doesn't stream changes — you re-query manually (or add your own change notifications). This (plus untyped rows) is why big apps prefer Drift (03_drift.md).
- **Repository**: map rows→entities at the boundary; keep SQL out of the UI (05 · repository).

Memory Representation

Query results are lists of maps in memory — bound with `LIMIT`/pagination. The DB is a single file on disk (15 · file_storage).

Compiler Behavior

No compile-time schema/query checking — raw SQL strings and `Map` casts are validated only at runtime (a key `sqflite` drawback).

Runtime Behavior

All ops are async; SQL errors/typos throw at runtime; transactions roll back on exception. Heavy queries block the calling isolate unless offloaded.

Flutter Engine Behavior

Crosses the embedder to native SQLite; runs on a background thread internally but marshals via the plugin (02 · isolates).

Dart VM Behavior

Not applicable.

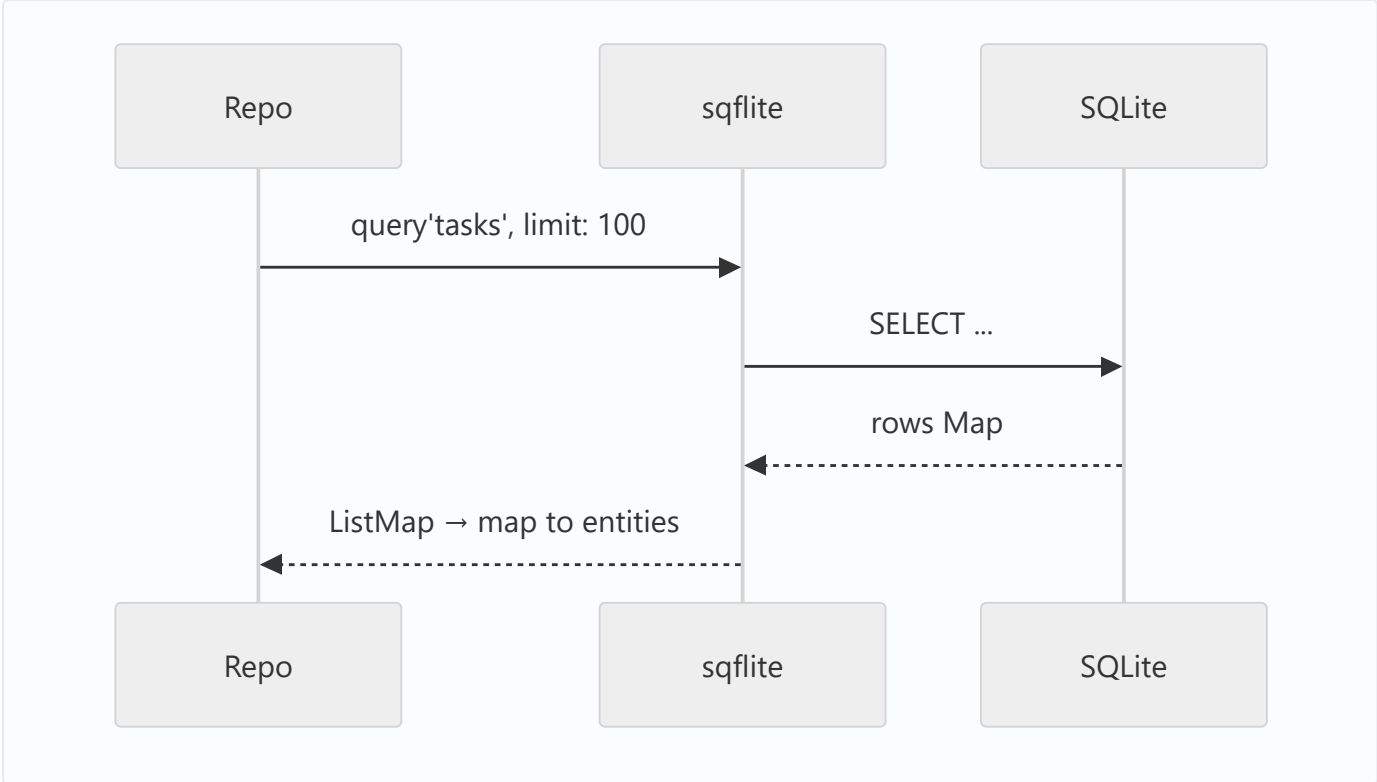
Examples

```
# pubspec.yaml
dependencies:
  sqflite: ^2.3.0
  path: ^1.9.0
```

```
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart' as p;

class Task { final int? id; final String title; final bool done;
  const Task({this.id, required this.title, this.done = false}); }

class TaskDao {
  late final Database _db;

  Future<void> open() async {
    final path = p.join(await getDatabasesPath(), 'app.db');
    _db = await openDatabase(
      path,
      version: 2,
      onCreate: (db, v) async {
```

```
        await db.execute('''
          CREATE TABLE tasks(
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            title TEXT NOT NULL,
            done INTEGER NOT NULL DEFAULT 0
          )''');
        await db.execute('CREATE INDEX idx_tasks_done ON tasks(done)'); // index
      },
      onUpgrade: (db, oldV, newV) async {
        if (oldV < 2) {
          await db.execute('ALTER TABLE tasks ADD COLUMN priority INTEGER DEFAULT 0'); // migration
        }
      },
    );
  }

  Future<int> add(Task t) =>
      _db.insert('tasks', {'title': t.title, 'done': t.done ? 1 : 0});

  Future<List<Task>> all() async {
    final rows = await _db.query('tasks', orderBy: 'id DESC', limit: 100); // bound!
    return rows.map((r) => Task(                 // map row -> entity (manual)
          id: r['id'] as int,
          title: r['title'] as String,
          done: (r['done'] as int) == 1,
        )).toList();
  }

  Future<void> toggle(int id, bool done) =>
      _db.update('tasks', {'done': done ? 1 : 0}, where: 'id = ?', whereArgs: [id]);

  Future<void> addManyAtomically(List<Task> tasks) async {
    await _db.transaction((txn) async {           // atomic
      for (final t in tasks) {
        await txn.insert('tasks', {'title': t.title, 'done': 0});
      }
    });
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
String-concatenating values into SQL	SQL injection/errors	Use `?` placeholders + `whereArgs`
Untyped `Map` casts everywhere	Runtime errors	Map to entities in a DAO/repository
No migrations / wrong version handling	Crash/data loss on upgrade	Bump `version` + `onUpgrade` step-by-step
Unbounded queries	Memory/perf	`LIMIT` + pagination + indexes
Expecting reactivity	`sqflite` doesn't stream	Re-query or use Drift
Heavy queries on UI isolate	Jank	Offload large work

Best Practices

- Use **parameterized queries** (`?` + `whereArgs`) — never string-concat values (injection).
- Wrap in a **DAO/repository** mapping rows→entities; keep SQL out of the UI.
- **Version +** `onUpgrade` for migrations (step-by-step, additive when possible).
- Add **indexes** and use `LIMIT` /pagination; select only needed columns.
- Use **transactions/batch** for atomic/bulk writes.
- For typed queries + reactivity, prefer **Drift** (built on SQLite) (03_drift.md).

Performance

Fast for indexed, bounded queries; transactions batch writes efficiently. Untyped mapping adds a little overhead; offload heavy queries; index hot columns (05_modeling_migrations_performance.md).

Advantages / Disadvantages

- + Full SQL power/control, mature/reliable, ubiquitous, transactions, foundation for Drift.
- – Untyped (runtime errors), verbose, manual migrations, **no reactivity**, manual entity mapping.

Interview Questions

1. 🟢 **What is `sqflite`?** — Flutter's low-level SQLite plugin: open a DB, run SQL/helpers, get untyped `Map` rows.
2. 🟢 **How do you migrate schema with `sqflite`?** — Bump the `version` and handle changes in `onUpgrade(db, oldV, newV)` (e.g., `ALTER TABLE`), stepwise.
3. 🟡 **How do you avoid SQL injection?** — Parameterized queries: `?` placeholders with `whereArgs`, never string concatenation.
4. 🟡 **Does `sqflite` support reactive queries?** — No; you re-query manually. Drift adds reactive `watch()` on top of SQLite.
5. 🟡 **How do you do atomic multi-writes?** — `db.transaction((txn) => ...)` (rolls back on error); `db.batch()` for bulk.
6. 🔴 **What are `sqflite`'s main drawbacks for large apps?** — Untyped rows (runtime cast errors), verbosity, manual migrations, and no reactivity — motivating Drift.
7. 🔴 **How do you keep DB code from coupling the app?** — Wrap it in a DAO/repository mapping rows→entities; the UI/domain never sees SQL/`Map`s.

Senior Engineer Tips

- Centralize SQL in DAOs with entity mapping; scattering raw `sqflite` maps across features is a maintenance trap.
- Write migrations defensively (additive, tested against real old DBs) — botched `onUpgrade` corrupts user data.
- If you find yourself wanting types + reactivity, switch to Drift rather than reinventing them over `sqflite`.

Architect Perspective

`sqflite` is the reliable low-level SQLite layer — great when you want raw control or a minimal dependency, and the substrate beneath Drift. For most apps, its lack of type-safety/reactivity pushes toward Drift; either way, DAO/repository boundaries + disciplined migrations keep the data layer maintainable (03_drift.md, Module 40).

Summary

- `sqflite` = direct SQLite: SQL/helpers → untyped `Map` rows; transactions/batch; version+ `onUpgrade` migrations.
- Parameterize queries, wrap in DAOs/repositories (rows→entities), index + bound queries.
- No types/reactivity — prefer Drift for large apps; `sqflite` for raw control/minimal deps.

Revision Notes

- `openDatabase(version, onCreate, onUpgrade)`; CRUD via `query/insert/update/delete` or `raw*` → `List<Map>`.
- Parameterize (`?` + `whereArgs`); transactions/batch for atomic/bulk.
- Migrations: bump version + `onUpgrade` stepwise (additive).
- Untyped + no reactivity → map in DAO; large apps → Drift.

Practice Questions

1. How do you migrate from v1 to v2 safely?
2. Why parameterize queries?
3. What does `sqflite` lack that Drift provides?

Coding Questions

1. Build a `TaskDao` (CRUD + index + entity mapping) over `sqflite`.
2. Add a v1→v2 migration adding a column.
3. Insert many tasks atomically in a transaction.

Mini Project

SQLite task store (Flutter): Implement a `TaskDao` /repository over `sqflite` with parameterized CRUD, an index, `LIMIT` pagination, row→entity mapping, a transaction for bulk insert, and a v1→v2 migration. Acceptance: no string-concatenated SQL; entities (not maps) exposed; migration works on an old DB; bounded queries; runs.

Drift (Typed, Reactive SQL)

> Drift is a reactive persistence library over SQLite with **compile-time-checked** queries and typed rows (via codegen): you declare tables/queries in Dart, get generated data classes and DAOs, and read data as **auto-updating streams** — SQLite's power without raw strings or manual refresh.

Introduction

Drift (package: `drift`, formerly Moor) sits on SQLite and adds type safety + reactivity. You define tables as Dart classes; codegen produces typed row classes, a type-safe query DSL, and `watch()` streams. This file covers defining tables, typed/reactive queries, DAOs, and migrations.

Why this concept exists

Raw `sqflite` gives runtime-only SQL with untyped maps and no reactivity (02_sqlite_sqflite.md). Drift fixes both: queries are checked at compile time, rows are typed, and reads stream updates — dramatically reducing bugs and boilerplate for relational Flutter apps.

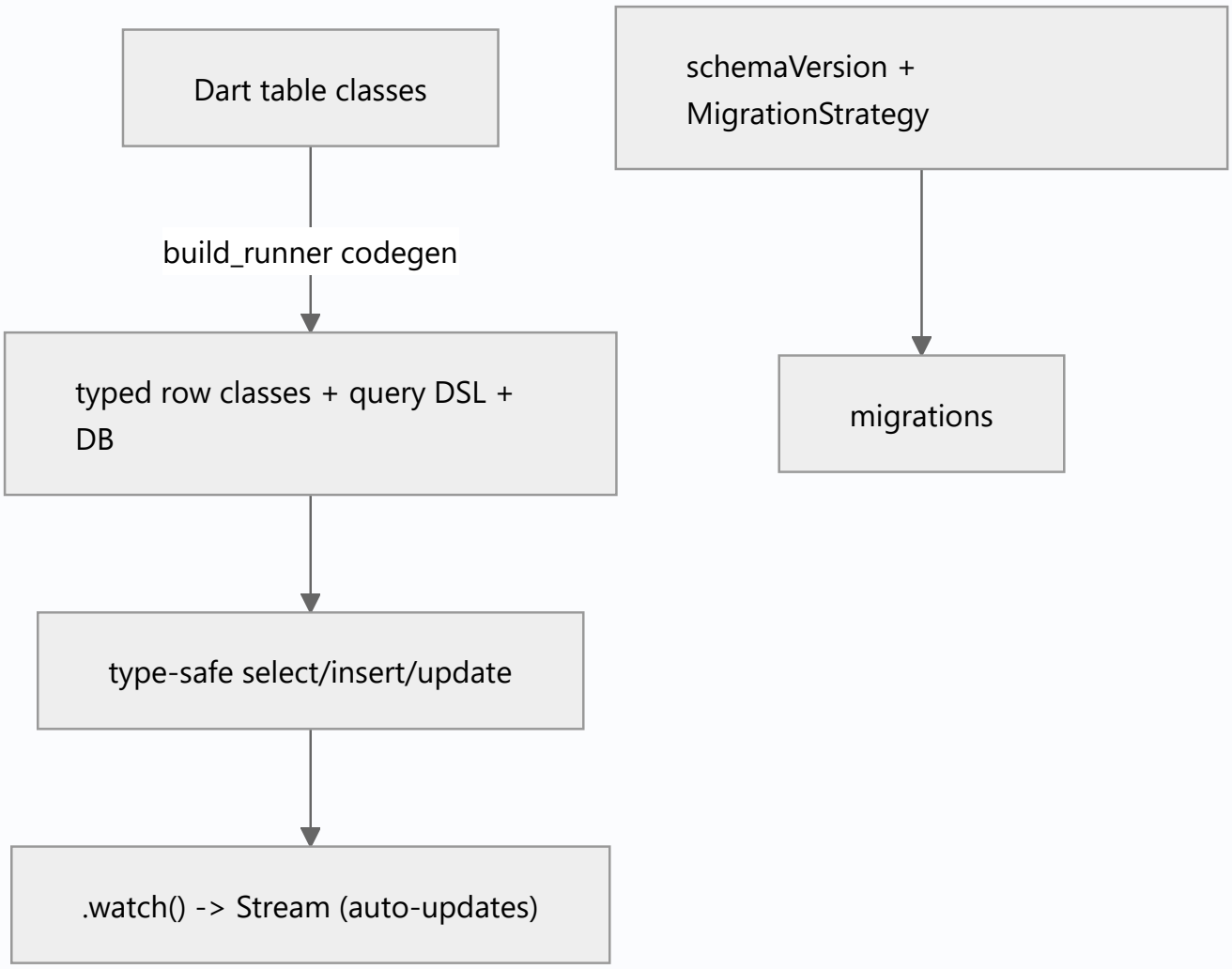
Real-world analogy

If `sqflite` is a manual car, Drift is the **automatic with lane-assist**: same SQLite engine, but it checks your route at compile time (typed queries), and the dashboard auto-updates as conditions change (reactive streams) — less manual work, fewer mistakes.

Problem Statement

Model tasks with a typed schema, query them type-safely, get a reactive task list that auto-updates the UI on writes, and migrate the schema — using Drift behind a repository. You'll define tables, use `watch()`, and write a migration.

Internal Working



- **Define tables** as classes extending `Table` (columns via typed getters); the DB extends `_$AppDatabase` (generated).
- **Codegen** (`build_runner`) produces: typed **row classes**, **companions** for inserts/updates, and a type-safe query API — queries are checked at **compile time**.
- **Reactive reads**: `select(...).watch()` returns a `Stream<List<Row>>` that re-emits on relevant writes — plug into `StreamBuilder` /state (02 · streams).
- **Queries**: type-safe DSL (`select(tasks)..where((t) => t.done.equals(false))`) or custom SQL with typed results; joins, aggregates supported.
- **DAOs**: group queries per table/feature (`@DriftAccessor`) for organization.
- **Migrations**: `schemaVersion` + `MigrationStrategy(onCreate, onUpgrade)` ; Drift has schema tooling/step-by-step migration helpers.
- Runs on SQLite (native, or WASM on web); can execute on a **background isolate** for heavy work.

Memory Representation

Query results are typed row objects; streams hold current results. Bound with `.limit()` /pagination; the DB is a SQLite file (15 · file_storage).

Compiler Behavior

Queries and schema are compile-time checked (generated code) — typos/type mismatches are build errors, not runtime crashes. Requires a `build_runner` codegen step.

Runtime Behavior

`watch()` streams re-emit on writes affecting the query; one-shot `get()` returns once. Migrations run on version bump; transactions supported.

Flutter Engine Behavior

Uses native SQLite via the plugin; supports background-isolate execution for heavy queries (02 · isolates).

Dart VM Behavior

Not applicable beyond async.

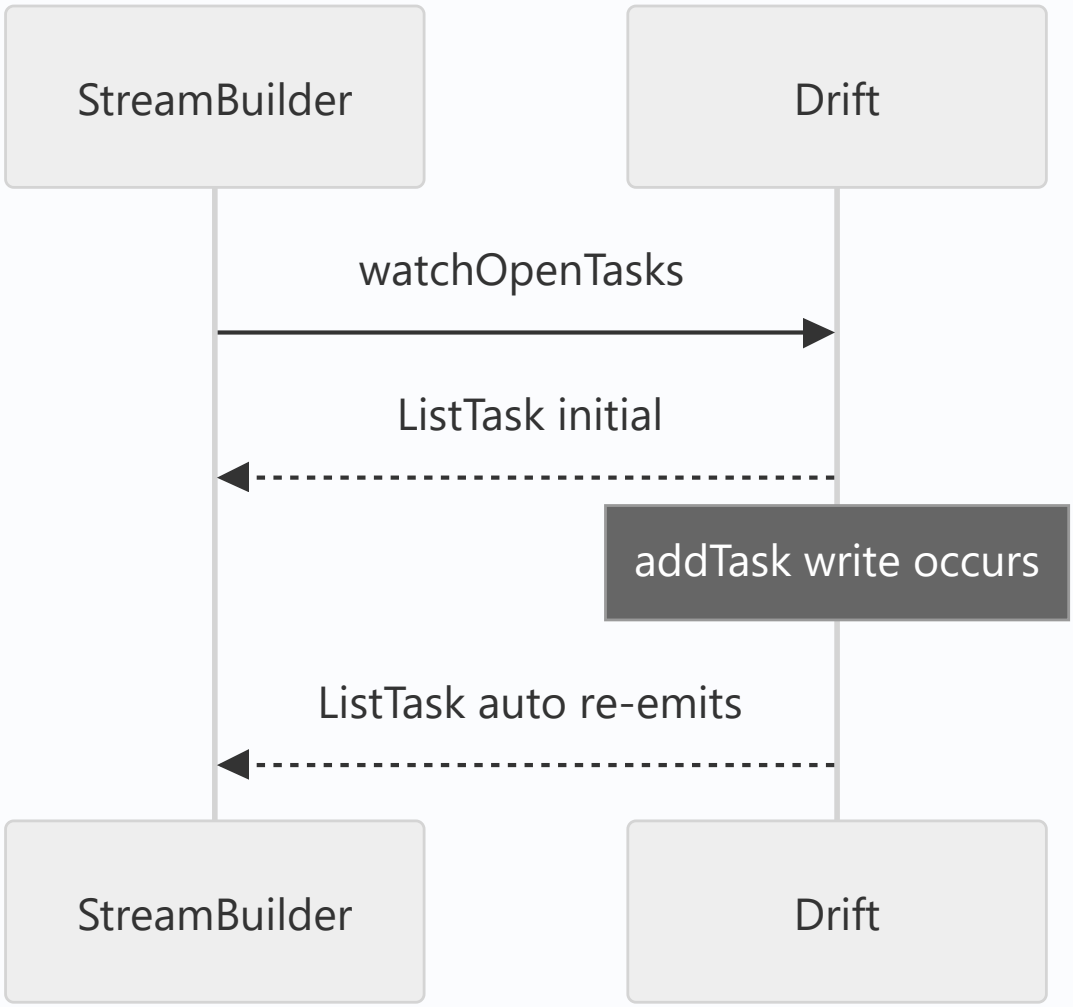
Examples

```yaml
# pubspec.yaml
dependencies:
  drift: ^2.16.0
  sqlite3_flutter_libs: ^0.5.0
  path_provider: ^2.1.0
  path: ^1.9.0
dev_dependencies:
  drift_dev: ^2.16.0
  build_runner: ^2.4.0
```

```
import 'package:drift/drift.dart';
// part 'db.g.dart'; // generated

// Typed table definition
class Tasks extends Table {
```

```
  IntColumn get id => integer().autoIncrement()();
  TextColumn get title => text().withLength(min: 1, max: 200)();
  BoolColumn get done => boolean().withDefault(const Constant(false))();
}

@DriftDatabase(tables: [Tasks])
class AppDatabase extends _$AppDatabase {
  AppDatabase(super.e);

  @override
  int get schemaVersion => 2;

  @override
  MigrationStrategy get migration => MigrationStrategy(
        onCreate: (m) => m.createAll(),
        onUpgrade: (m, from, to) async {
          if (from < 2) {
            await m.addColumn(tasks, tasks.priority); // migration (if column added)
          }
        },
      );

  // Reactive, TYPE-SAFE query -> auto-updates the UI on writes:
  Stream<List<Task>> watchOpenTasks() =>
      (select(tasks)..where((t) => t.done.equals(false))..limit(100)).watch();

  Future<int> addTask(String title) =>
      into(tasks).insert(TasksCompanion.insert(title: title));

  Future<void> toggle(int id, bool done) =>
      (update(tasks)..where((t) => t.id.equals(id))).write(TasksCompanion(done: Value(done)));
}
// `Task` (row class) + `TasksCompanion` are generated — fully typed.
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Forgetting `build_runner`	Generated code missing/stale	Run/watch codegen
Unbounded `watch()`	Memory/perf on large tables	`.limit()` /pagination
Leaking Drift row classes to the UI	Coupling	Map rows→domain entities in a repository
Bad migrations	Data loss on upgrade	Use `MigrationStrategy` /schema tooling; test
Heavy queries on UI isolate	Jank	Background isolate execution
Overusing custom raw SQL	Loses type-safety	Prefer the typed DSL where possible

Best Practices

- Define tables in Dart; **run codegen**; use the **typed DSL** (compile-safe) over raw SQL where possible.
- Expose **reactive** `watch()` streams for lists that must auto-update; bound with `.limit()`/pagination.
- Organize queries in **DAOs**; wrap in a **repository** mapping generated rows→domain entities.
- Write **migrations** with `MigrationStrategy` + Drift's schema tooling; test against old schemas.
- Run heavy queries on a **background isolate**; select only needed columns; index hot columns (05_modeling_migrations_performance.md).

Performance

Compile-checked, indexed, bounded queries are fast; reactive streams re-run only affected queries. Background-isolate execution avoids UI jank on heavy work (Module 21).

Advantages / Disadvantages

- **+** Compile-time-safe queries, typed rows, **reactive streams**, full SQL (joins/aggregates), migrations tooling, background isolates.
- **−** Codegen/build step, learning curve vs raw `sqflite`, more setup, still SQLite lock-in (but portable schema).

Interview Questions

1. 🟢 **What does Drift add over `sqflite`?** — Compile-time-checked queries, typed row classes, and reactive `watch()` streams — over the same SQLite engine.
2. 🟢 **How do you get auto-updating data?** — `select(...).watch()` returns a `Stream` that re-emits when writes affect the query.
3. 🟡 **How are Drift's types generated?** — `build_runner` codegen from your table/DB classes produces typed rows, companions, and a query DSL.
4. 🟡 **How do migrations work in Drift?** — Bump `schemaVersion` and implement `MigrationStrategy(onCreate/onUpgrade)`, using schema tooling for step-by-step changes.
5. 🟡 **Why prefer the typed DSL over custom SQL?** — It's compile-time-checked; raw SQL reintroduces runtime-only errors (though Drift can type custom SQL too).
6. 🔴 **How do you keep Drift from coupling the app?** — Map generated row classes to domain entities in a repository; the domain doesn't import Drift.
7. 🔴 **How does Drift help performance/UX?** — Reactive queries update only affected UI, indexed/bounded queries are fast, and heavy work can run on a background isolate.

Senior Engineer Tips

- Prefer Drift for relational Flutter apps — the compile-safety + reactivity eliminate whole bug classes vs raw `sqflite`.
- Map row classes to entities at the repository; don't leak `Task`/companions into widgets.
- Use Drift's schema-version tooling and test migrations against real old databases; migrations are the riskiest DB code.

Architect Perspective

Drift is the recommended relational data layer for Flutter: typed, reactive SQL behind repositories gives a compile-safe, auto-updating, migratable foundation — ideal as the local source of truth for offline-first (Module 19) and rich caching (15). It trades a codegen step for major safety/productivity gains.

Summary

- Drift = typed, reactive SQL over SQLite via codegen: compile-checked queries, typed rows, `watch()` streams, migrations.
- Use the typed DSL, expose reactive bounded queries, organize in DAOs, map rows→entities in a repository.
- Recommended relational engine; ideal local source of truth for offline-first.

Revision Notes

- Define `Table` classes → `build_runner` → typed rows/companions + type-safe DSL + `.watch()` streams.
- `schemaVersion` + `MigrationStrategy(onCreate/onUpgrade)`; DAOs for organization.
- Compile-time query checks; bound with `.limit()`; map rows→entities in repo; heavy work off-isolate.
- Built on SQLite; recommended relational engine (vs raw `sqflite`).

Practice Questions

1. How does Drift make queries compile-safe?
2. How do you build an auto-updating task list?
3. How do Drift migrations differ from raw `sqflite`?

Coding Questions

1. Define a `Tasks` table + reactive `watchOpenTasks()` and a typed `addTask`.
2. Add a v1→v2 migration adding a column via `MigrationStrategy`.
3. Wrap the Drift DB in a repository mapping rows→entities.

Mini Project

Typed reactive tasks (Flutter): Build a Drift `AppDatabase` (Tasks table), a repository exposing `watchOpenTasks()` (mapped to entities) + typed CRUD, a v1→v2 migration, and a `StreamBuilder` UI that auto-updates on writes. Acceptance: compile-safe typed queries; reactive UI; migration works; rows mapped to entities; bounded queries; runs.

Isar & Hive (NoSQL Object Stores)

> **Isar** is a fast, typed NoSQL object database with rich indexed queries and reactive streams; **Hive** is a simpler, extremely fast key-object store with weak querying — both persist Dart objects directly (no SQL/joins), ideal for object-shaped, speed-first data.

Introduction

Not all data is relational. Isar and Hive store **Dart objects** directly. Isar offers typed collections, indexes, links, queries, and reactive watchers; Hive offers dead-simple, blazing-fast key→object "boxes." This file covers both, their tradeoffs, and when to pick each over SQL.

Why this concept exists

For object-oriented, schemaless, or performance-critical local data (large caches, offline object stores, settings-plus), mapping to relational tables/joins is friction. Object stores persist objects natively — faster and simpler for that shape, at the cost of relational querying.

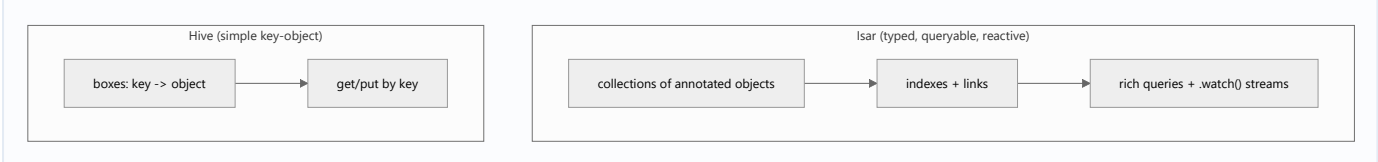
Real-world analogy

Isar is a **smart storage unit with a labeled index** — toss in objects, find them fast by many attributes (indexed queries). Hive is a **set of labeled bins** — drop an object in a bin by key and grab it instantly, but there's no cross-bin search engine.

Problem Statement

Cache 50k product objects for very fast reads with attribute queries (Isar), and store a handful of key→object records with instant lookup (Hive). You'll model both and query them.

Internal Working



- **Isar** (`isar`): annotate classes (`@collection` , `Id id`), codegen generates schemas; **indexes** (`@Index`) enable fast typed queries (`.filter().xEqualTo(...).findAll()`), **links** relate objects, and `.watch()` provides reactive streams. Fast (mmap), typed, supports queries/sorting/aggregation and background isolates.

- **Hive** (`hive` / `hive_ce`): **boxes** are key→value maps persisting primitives or objects (via `TypeAdapter`s, often codegen'd). `box.put(key, obj)` / `box.get(key)` ; iterate a box; limited querying (no rich filters/joins). Extremely fast/simple; great for caches/settings-plus.
- **Choosing**: **Isar** when you need typed objects **and** queries/relations/reactivity; **Hive** for simple key-object storage where you look up by key and don't need querying.
- Both **persist Dart objects directly** (no SQL); front behind a repository mapping stored objects→entities where the domain model differs.

Memory Representation

Isar uses memory-mapped files (fast, lazy loading); Hive keeps boxes largely in memory (fast, but a huge box costs RAM). Bound reads (queries/ `limit` in Isar; avoid giant Hive boxes) (02 · memory_and_gc).

Compiler Behavior

Both use **codegen** (Isar schemas / Hive adapters) via `build_runner` ; Isar queries are typed (compile-checked filters). Hive get/put by key is typed by the box's type but has no query DSL.

Runtime Behavior

Isar queries are fast (indexed) and can `watch()` for reactivity; Hive lookups are O(1) by key; scanning a Hive box for "queries" is O(n). Both async-open; writes persist immediately.

Flutter Engine Behavior

Native fast I/O via the plugin; Isar supports background-isolate queries (02 · isolates).

Dart VM Behavior

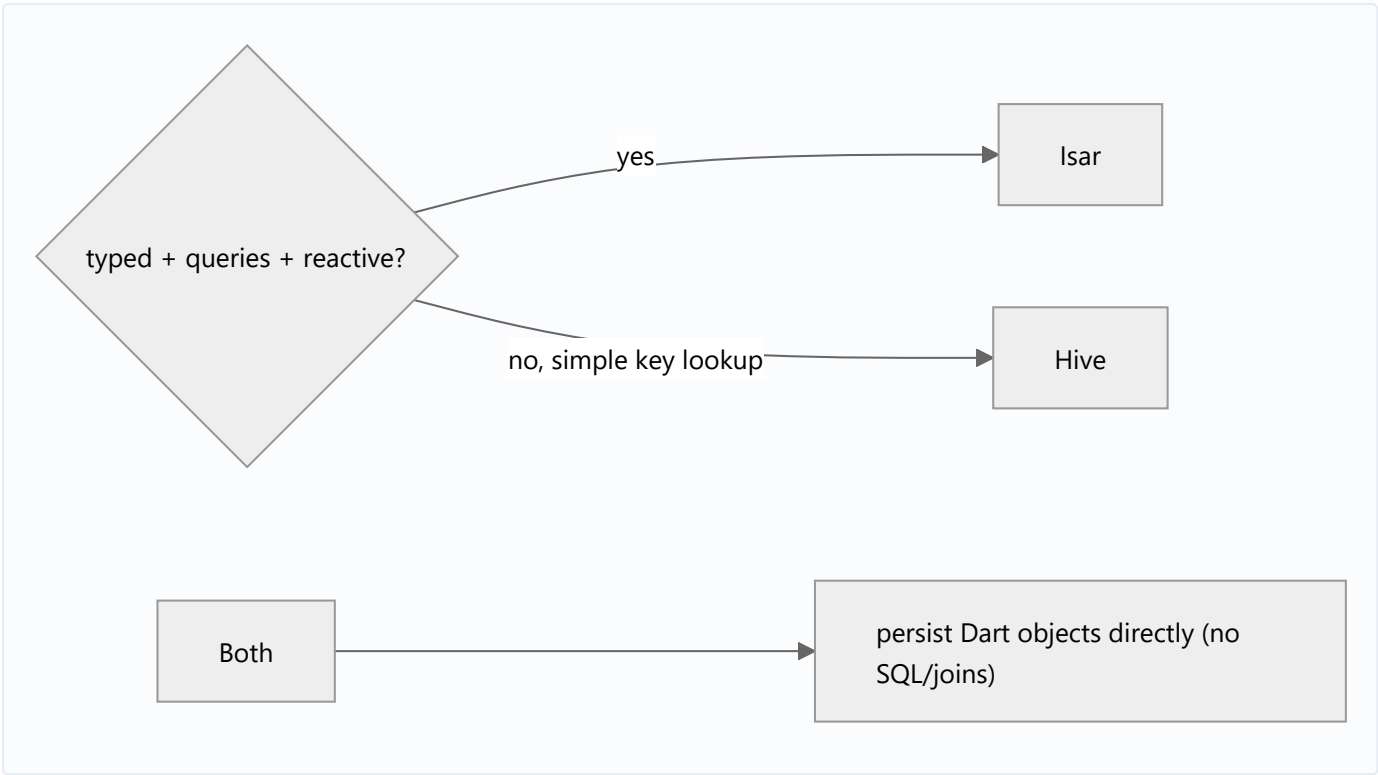
Not applicable beyond async.

Examples

```yaml
# pubspec.yaml
dependencies:
  isar: ^3.1.0
  isar_flutter_libs: ^3.1.0
  hive: ^2.2.0
  hive_flutter: ^1.1.0
dev_dependencies:
  isar_generator: ^3.1.0
  hive_generator: ^2.0.0
  build_runner: ^2.4.0
```

```dart
// --- Isar: typed, indexed, reactive ---
import 'package:isar/isar.dart';
// part 'product.g.dart';

@collection
class Product {
  Id id = Isar.autoIncrement;
  @Index() late String category;   // indexed -> fast query
  late String name;
  late double price;
}

Future<void> isarDemo(Isar isar) async {
  await isar.writeTxn(() => isar.products.put(Product()..category = 'book'..name = 'Dart'..price = 9.99));
  // typed, indexed query:
  final books = await isar.products.filter().categoryEqualTo('book').sortByPrice().findAll();
  // reactive stream (auto-updates):
  final stream = isar.products.filter().categoryEqualTo('book').watch(fireImmediately: true);
  stream.listen((list) {/* UI updates */});
}

// --- Hive: simple key-object ---
import 'package:hive/hive.dart';
Future<void> hiveDemo() async {
  final box = await Hive.openBox<String>('settings'); // key -> object box
  await box.put('theme', 'dark');   // instant put
  final theme = box.get('theme');   // O(1) lookup
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Hive for queryable/relational data	No real querying/joins	Use Isar or SQL (Drift)
Huge Hive box in memory	RAM bloat	Use Isar (lazy/mmap) or bound data
No indexes in Isar for filtered fields	Slow scans	`@Index()` filtered/sorted fields
Forgetting `build_runner` (adapters/schemas)	Codegen missing	Run/watch codegen
Leaking stored objects to UI	Coupling/lock-in	Map to entities in a repository
Expecting joins in either	They're NoSQL	Model with links (Isar) or denormalize

Best Practices

- Pick **Isar** for typed objects needing queries/relations/reactivity; **Hive** for simple key-object storage/caches.
- **Index** Isar fields you filter/sort by; bound queries (`limit`); use `.watch()` for reactive UI.
- Keep **Hive boxes reasonable**; don't treat a box as a queryable table (no filters/joins).
- Run codegen (adapters/schemas); front behind a **repository** mapping stored objects→entities.
- For relational/reporting needs, prefer **SQL (Drift)** instead (03_drift.md).

Performance

Isar (mmap + indexes) is very fast for typed queries and scales to large datasets; Hive is fastest for key lookups but scans are O(n) and big boxes eat RAM. Index Isar queries; keep Hive boxes bounded (Module 21).

Advantages / Disadvantages

- **+ Isar:** fast, typed, indexed queries, links, reactive, large datasets. **+ Hive:** trivial API, extremely fast key lookups, minimal setup.
- **− Isar:** codegen/setup, NoSQL (no SQL joins). **− Hive:** weak querying, in-memory box cost, not for relational/large-query data.

Interview Questions

1. 🟢 **Isar vs Hive?** — Isar: typed NoSQL with indexed queries, links, and reactive streams (large/queryable data); Hive: simple, very fast key→object boxes with minimal querying.
2. 🟢 **When choose a NoSQL object store over SQL?** — For object-shaped, schemaless, or speed-first data without relational joins/reporting needs.
3. 🟡 **How do you get fast queries in Isar?** — Add `@Index()` to filtered/sorted fields; use typed filter queries with `limit`.
4. 🟡 **Why is scanning a Hive box for "queries" a problem?** — Hive has no query engine; filtering iterates the whole box (O(n)) — use Isar/SQL for real queries.
5. 🟡 **How do both persist data?** — As Dart objects directly (via codegen schemas/adapters), no SQL tables.
6. 🔴 **What's the memory difference between Isar and Hive at scale?** — Isar uses memory-mapped files (lazy, large-dataset friendly); Hive tends to hold boxes in memory (large boxes = high RAM).
7. 🔴 **How do you avoid lock-in/coupling with these?** — Wrap in a repository mapping stored objects→domain entities so the domain doesn't depend on Isar/Hive types.

Senior Engineer Tips

- Default to **Isar** for object-heavy apps needing queries + reactivity; use **Hive** only for simple key-object caches/settings.
- Index every field you filter/sort on in Isar; unindexed queries silently degrade to scans.
- Keep Hive boxes small and purpose-specific; if you're iterating a box to "query," you picked the wrong tool.

Architect Perspective

Isar/Hive give object-native persistence — Isar as a fast, queryable, reactive NoSQL DB (viable local source of truth for offline-first — Module 19); Hive as a lightweight key-object cache. The SQL-vs-NoSQL choice (01_database_options.md) hinges on relational/query needs; behind repositories, either keeps the domain clean and swappable.

Summary

- Isar: typed NoSQL, indexed queries, links, reactive — fast and scalable. Hive: simple/fast key→object boxes, weak querying.
- Persist Dart objects directly (no SQL); index Isar queries, keep Hive boxes small, front behind repositories.
- Prefer SQL (Drift) for relational/reporting; NoSQL object stores for object/speed-first data.

Revision Notes

- Isar: `@collection` / `@Index` + codegen; typed filter queries + `.watch()`; mmap, large datasets, background isolates.
- Hive: boxes (key→object via `TypeAdapter`), O(1) get/put, minimal querying, in-memory (keep small).
- Both NoSQL (objects, no joins); index Isar; repository maps objects→entities.
- Relational/reporting → SQL (Drift); object/speed → Isar/Hive.

Practice Questions

1. When is Hive enough vs needing Isar?
2. Why index Isar query fields?
3. Why is a NoSQL object store a poor fit for relational reporting?

Coding Questions

1. Model a `Product` Isar collection with an index + a filtered reactive query.
2. Store/retrieve settings in a typed Hive box.
3. Wrap an Isar collection in a repository mapping objects→entities.

Mini Project

Object cache (Flutter): Build an Isar-backed `ProductRepository` with an indexed category query and a reactive `watch`, mapped to entities, plus a Hive box for simple app settings. Benchmark an indexed vs unindexed Isar query. Acceptance: typed indexed queries + reactive stream; Hive for key-object settings; objects mapped to entities; index benefit shown; runs.

Data Modeling, Migrations & Performance

> Cross-engine essentials: model schemas around your queries (relations/indexes), evolve them with **safe, versioned migrations** (never lose user data), and make queries fast with **indexes, bounded reads, and off-isolate execution**.

Introduction

Whatever engine you chose (01_database_options.md), three concerns decide your data layer's quality: **modeling** (schema, relations, normalization vs denormalization), **migrations** (evolving the schema without data loss), and **performance** (indexing, query cost, memory). This file covers all three, engine-agnostically.

Why this concept exists

Poor modeling causes slow/awkward queries; botched migrations corrupt or wipe user data (a catastrophic bug); and unindexed/unbounded queries jank the UI or blow memory. These are the recurring, high-stakes database skills across `sqflite`/Drift/Isar/Hive.

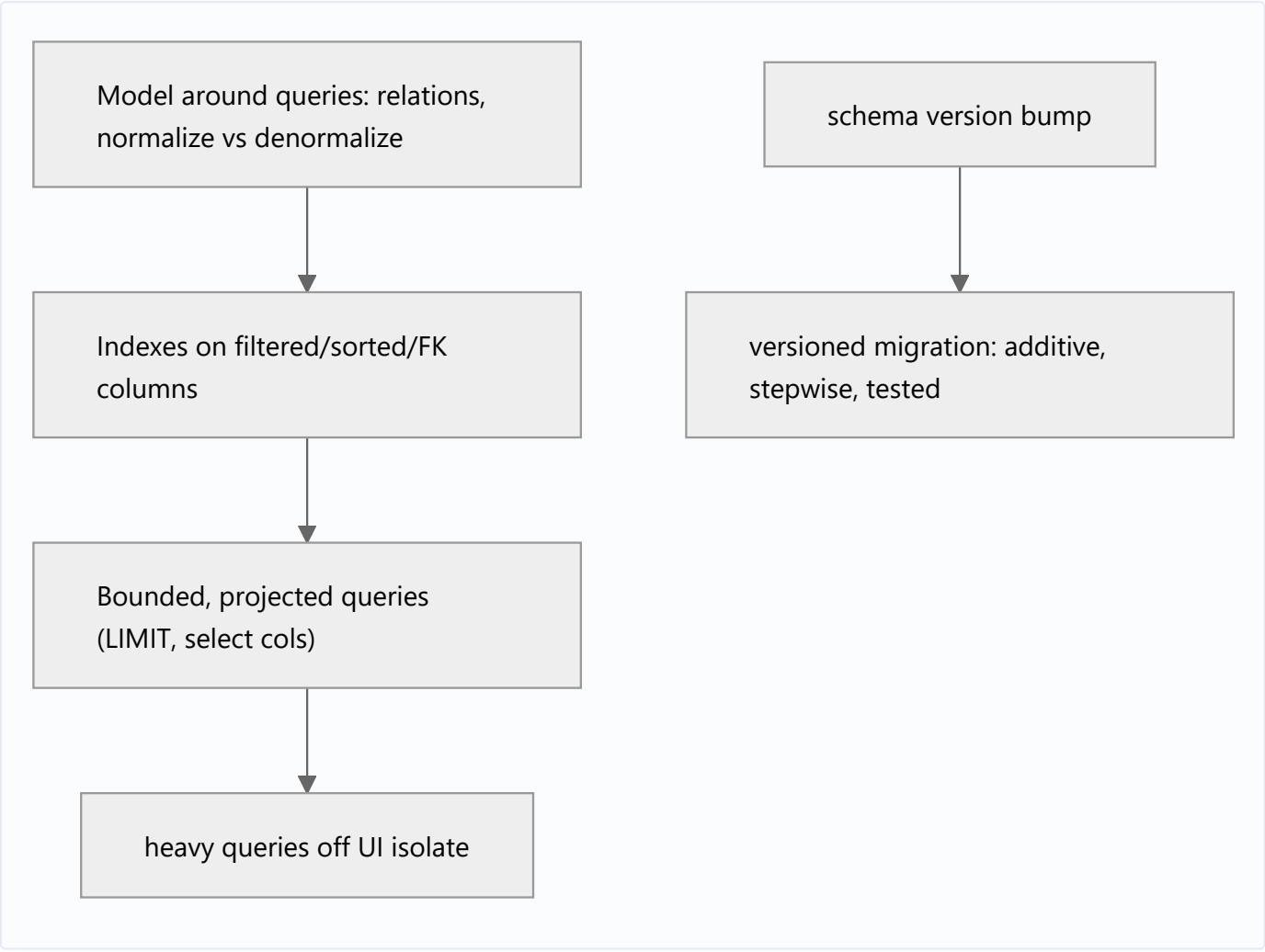
Real-world analogy

Modeling is **designing a warehouse layout** for how you'll fetch goods; migrations are **renovating the warehouse while it's full of inventory** (you must not lose stock); performance is **adding aisle signs/indexes and forklifts** so retrieval is fast and you don't haul the whole warehouse to find one box.

Problem Statement

Design a users↔tasks schema for common queries, add a new column in v2 without losing data, and make "open tasks by user" fast at scale. You'll model relations/indexes, write a migration, and optimize the query.

Internal Working



- **Modeling**:
  - **Relations**: one-to-many/many-to-many via foreign keys (SQL) or links (Isar); avoid duplicating what you must keep consistent.
  - **Normalize vs denormalize**: normalize for integrity/writes; denormalize selectively for read speed (trade duplication for fewer joins) — mirror the caching/offline tradeoff (15 · caching_strategies).
  - **Model around queries**: shape tables/indexes for the reads you actually do.
- **Migrations**:
  - Bump the **schema version**; run **stepwise** migrations (v1→v2→v3), **additive** where possible (add columns/tables), avoid destructive changes.
  - **Never lose data**: back-fill new columns, migrate/transform rows in-place; test against **real old databases**.
  - Engine tooling: `sqflite` `onUpgrade`, Drift `MigrationStrategy`/schema tests, Isar schema migration.
- **Performance**:

- **Indexes** on columns you filter/sort/join by (and FKs); unindexed filters = full scans.
- **Bound reads**: `LIMIT`/pagination; **project** only needed columns; avoid `SELECT *` on wide rows.
- **Batch/transaction** bulk writes; use **reactive queries** so only affected reads re-run.
- **Off-isolate**: run heavy queries/imports on a background isolate (02 · isolates).
- **Analyze**: use `EXPLAIN QUERY PLAN` (SQLite) to confirm index usage.

Memory Representation

Result sets load into memory — bound them. Indexes add storage/write cost but speed reads. Large migrations/imports should stream/batch off-isolate (02 · memory_and_gc).

Compiler Behavior

Typed engines (Drift/Isar) validate schema/queries at build time; migrations are still runtime logic you must test.

Runtime Behavior

Migrations run on version bump at open; a failed/incorrect migration can corrupt data — hence testing. Unindexed queries scale O(n); indexed lookups are ~O(log n).

Flutter Engine Behavior / Dart VM Behavior

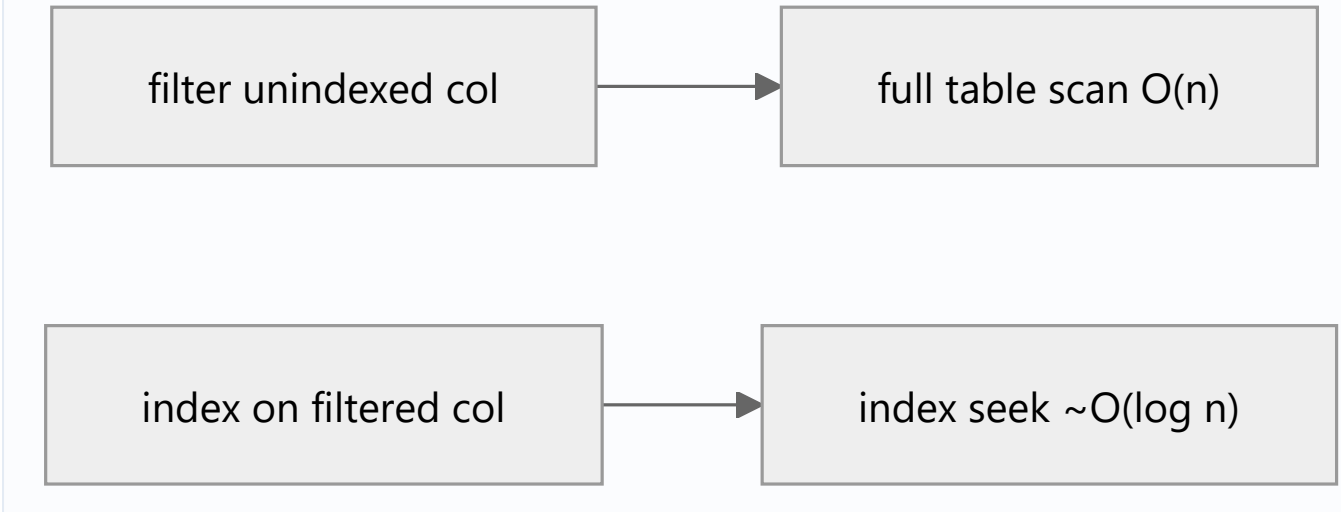
Not applicable beyond async; heavy DB work off the UI isolate.

Examples

```
// SQLite index + bounded, projected query (fast "open tasks by user"):
// CREATE INDEX idx_tasks_user_done ON tasks(user_id, done);   -- composite index
// SELECT id, title FROM tasks WHERE user_id = ? AND done = 0 ORDER BY id DESC LIMIT 50;
//   -> uses the index; projects only needed columns; bounded.

// Safe additive migration (sqflite onUpgrade), stepwise:
Future<void> onUpgrade(dynamic db, int oldV, int newV) async {
  if (oldV < 2) {
    await db.execute('ALTER TABLE tasks ADD COLUMN priority INTEGER NOT NULL DEFAULT 0'); // additive +
default
  }
  if (oldV < 3) {
    await db.execute('CREATE INDEX idx_tasks_user_done ON tasks(user_id, done)'); // add index
  }
}

// Bulk import off the UI isolate + in a transaction:
// await Isolate.run(() => db.transaction((txn) => bulkInsert(txn, rows)));
```

```
Diagnose index usage (SQLite):
  EXPLAIN QUERY PLAN SELECT ... WHERE user_id = ? AND done = 0;
  -> should show "USING INDEX idx_tasks_user_done", not "SCAN"
```

Diagrams



Common Mistakes

Mistake	Why	Fix
No indexes on filtered/sorted/FK columns	Full scans → slow	Add (composite) indexes; verify with EXPLAIN
Destructive/untested migrations	Data loss/corruption	Additive + stepwise + tested on real old DBs
`SELECT *` / unbounded queries	Memory/perf	Project columns + `LIMIT` /pagination
Over-normalizing read-hot data	Costly joins	Denormalize selectively for reads
Heavy queries/imports on UI isolate	Jank	Off-isolate + batch/transaction
Skipping schema version discipline	Broken upgrades	Bump version + migration per change

Best Practices

- **Model around your queries**; use relations + **indexes** (composite where filters combine); denormalize selectively for read speed.
- Treat **migrations as critical code**: versioned, **additive**, **stepwise**, back-fill data, and **test against real old databases**.
- **Bound and project** queries (`LIMIT`, pagination, needed columns); index filtered/sorted/FK columns; verify with `EXPLAIN QUERY PLAN`.
- **Batch/transaction** bulk writes; use **reactive queries**; run heavy work **off-isolate**.
- Front the DB behind a **repository**; keep DB types out of the domain.

Performance

Indexes turn scans into seeks (huge at scale); bounded/projected queries cap memory; transactions batch writes; off-isolate execution keeps frames smooth. Profile queries (EXPLAIN, DevTools) rather than guessing (Module 21).

Advantages / Disadvantages

- **+** Fast, correct, evolvable data layer; safe upgrades; scalable queries.
- **−** Indexing/migration discipline required; denormalization adds duplication/consistency work; migrations are risky if untested.

Interview Questions

1. 🟢 **Why add indexes?** — To turn full-table scans (O(n)) into index seeks (~O(log n)) for filtered/sorted/joined columns.
2. 🟢 **What makes a migration safe?** — Versioned, additive, stepwise (v1→v2→v3), data back-filled, and tested against real old databases — no destructive/untested changes.

3. 🟡 **Normalize vs denormalize — tradeoff?** — Normalize for integrity/write-efficiency; denormalize selectively for read speed (fewer joins) at the cost of duplication/consistency work.
4. 🟡 **How do you bound query cost/memory?** — `LIMIT`/pagination, project only needed columns, and index the query — avoid `SELECT *`/unbounded reads.
5. 🟡 **How do you verify a query uses an index?** — `EXPLAIN QUERY PLAN` (SQLite) — it should show index usage, not a scan.
6. 🔴 **How do you migrate without losing user data?** — Stepwise additive changes with data back-fill/transformation, run in order at open, tested on real old-schema databases (and consider backups).
7. 🔴 **How do you keep heavy DB work from janking the UI?** — Run large queries/imports on a background isolate and batch writes in transactions.

Senior Engineer Tips

- Design indexes from your **actual query patterns** (composite indexes matching WHERE+ORDER BY); confirm with EXPLAIN, don't assume.
- Keep a **migration test suite** that opens fixtures of every prior schema version and upgrades them — migrations are the highest-risk DB code.
- Denormalize deliberately and document it; uncontrolled duplication becomes a consistency nightmare.

Architect Perspective

Modeling, migrations, and performance are the durable database concerns independent of engine choice. A query-shaped, indexed schema; disciplined, tested migrations; and bounded, off-isolate queries behind repositories yield a fast, correct, evolvable data layer — the backbone under caching (15) and offline-first (19) and a frequent scaling/system-design topic (Module 48).

Summary

- Model around queries with relations + indexes; denormalize selectively for reads.
- Migrations: versioned, additive, stepwise, data-preserving, tested against real old DBs.
- Performance: index filtered/sorted/FK columns, bound+project queries, batch writes, run heavy work off-isolate; verify with EXPLAIN.

Revision Notes

- Model around queries; relations + composite indexes; denormalize selectively.
- Migrations: bump version, additive+stepwise, back-fill, test on real old DBs (highest-risk code).

- Perf: index filtered/sorted/FK; `LIMIT` +project (no `SELECT *`); transactions/batch; off-isolate; EXPLAIN QUERY PLAN.
- Repository-fronted; DB types out of domain.

Practice Questions

1. What columns should you index for "open tasks by user, newest first"?
2. What makes a migration data-safe?
3. How do you confirm a query uses an index?

Coding Questions

1. Add a composite index and rewrite a query to use it; verify with EXPLAIN.
2. Write a stepwise additive migration (v1→v3) that back-fills a column.
3. Run a bulk import in a transaction on a background isolate.

Mini Project — Module capstone

Optimized, migratable data layer (Flutter): Model users↔tasks (relations + composite index) in your chosen engine, expose reactive bounded queries behind a repository (rows→entities), write a v1→v2→v3 migration with a back-filled column + added index, add a migration test opening old-schema fixtures, and benchmark indexed vs unindexed "open tasks by user." Acceptance: query-shaped indexed schema; safe tested migrations; bounded/projected queries; measured index benefit; heavy import off-isolate; runs.