**Seminarul 9 — Persistență SQLite/Sequelize — Partea 1 (Teorie)**

HOOK: De la fișiere JSON și foi de calcul la o bază relațională cu SQLite + Sequelize

Imaginați-vă o federație de asociații studențești care își gestionează activitatea prin foi de calcul și câteva fișiere JSON ținute pe laptopurile voluntarilor. La început totul pare simplu: un tabel Members cu nume și email, un tabel Clubs cu denumiri și categorii, iar înscrierile sunt doar niște rânduri adăugate într-un sheet Registrations. După primele evenimente apar însă semnele de uzură: două documente conțin aceeași persoană cu emailuri diferite, ștergerea unui club nu „cascadează” în lista de înscrieri, rapoartele sunt inconsistent agregate, iar un voluntar suprascrie din greșeală o versiune mai veche a fișierului. Pe scurt, lipsesc integritatea (integrity), consistența (consistency), izolarea (isolation) și durabilitatea (durability) pe care le cer sistemele cu date partajate și concurente.

Acesta este momentul în care trecem de la persistență ad‑hoc (fișiere, JSON, spreadsheets) la un model relațional (relational model) și la un motor de baze de date (database engine) care aplică regulile jocului în mod sistematic. În acest seminar, vom pune reflectorul pe \*\*SQLite\*\* (motor embedded, fișier unic) și pe \*\*Sequelize\*\* (ORM pentru Node.js) ca să înțelegem, la nivel conceptual, cum modelăm entități, relații și constrângeri, cum obținem tranzacții ACID, cum evităm erorile tipice (de pildă N+1) și cum pregătim terenul pentru laboratorul practic.

1. De ce model relațional și de ce SQLite în faza incipientă

Modelul relațional a fost introdus de E. F. Codd în 1970 (Communications of the ACM) și a schimbat radical modul în care proiectăm date: tabele (relații), rânduri (tupluri), coloane (atribute), chei primare/străine și operatori algebrici (JOIN, SELECT, PROJECT etc.). Ideea centrală este separarea clară între \*\*ce\*\* stocăm (structură logică) și \*\*cum\*\* stocăm (reprezentare fizică), ceea ce permite independență față de implementare și evoluție. [Codd, 1970] (DOI în bibliografie).

SQLite este un motor \*\*embedded\*\*: aplicația deschide un fișier `.db` și execută comenzi SQL direct în procesul ei. Nu există un server separat de administrat; distribuirea aplicației devine, de multe ori, o chestiune de a livra binarul + fișierul bazei. Această simplitate îl face excelent pentru \*\*etapele inițiale\*\* ale proiectelor didactice și prototipurilor, și este motivul pentru care SQLite este astăzi printre cele mai răspândite motoare de baze de date, cu utilizare masivă în dispozitive mobile și aplicații desktop. Analize riguroase din zona academică (PVLDB) documentează răspândirea și evoluția SQLite, inclusiv optimizările recente. citeturn0search11

\*\*Jurnalizarea\*\* (journaling) și \*\*Write‑Ahead Logging (WAL)\*\* oferă durabilitate și performanță bună la citire concurentă (cititorii pot citi în timp ce un scriitor scrie), iar comutarea în WAL se face printr‑un simplu `PRAGMA journal\_mode=WAL;`, pornind de la modul implicit `DELETE`. Înțelegerea diferențelor între rollback journal și WAL, precum și efectele asupra concurenței, este esențială pentru alegerea corectă a parametrilor. citeturn3search1

2. Din foi de calcul la scheme relaționale: principii de modelare

La baza corectitudinii stau \*\*cheile\*\* și \*\*constrângerile\*\*. Cheia primară (primary key) identifică un tuplu, iar cheile străine (foreign keys) mențin legăturile între tabele, protejând integritatea referențială. \*\*Normalizarea\*\* (1NF, 2NF, 3NF) reduce redundanța și anomaliile de inserare/ștergere/actualizare; pentru situații mai sofisticate, literatura introduce forme normale superioare (BCNF, 4NF). Înțelegerea acestor concepte vine din literatura clasică, de la Codd (1970) până la lucrările ulterioare ale lui Fagin (1977). citeturn0search2turn6search1

Cazul nostru didactic – „Student Clubs” – conține cel puțin patru entități: \*\*Club\*\*, \*\*Member\*\*, \*\*Registration\*\* (tabel de legătură pentru M:N între Member și Club), \*\*Event\*\* (evenimente asociate unui Club). În regimul foi de calcul, tendința este să duplicăm coloane (de ex. numele clubului în foaia de înscrieri). În schema relațională, păstrăm \*\*id‑uri\*\* (surrogate keys) și conectăm tabelele prin \*\*FK\*\* (foreign keys) cu politici de \*\*ON DELETE/UPDATE\*\* (restrict, cascade, set null) în funcție de semantica dorită.

3. SQLite: arhitectură, jurnale și pragmas (cu capcane)

\*\*Arhitectură embedded\*\*. SQLite compilează interogările în bytecode intern și operează direct pe fișierul bazei. Nu există suprataxă de IPC client–server, ceea ce e ideal pentru aplicații locale, CLI‑uri, testare. Documentația oficială explică în detaliu „PRAGMA” – extensii SQLite pentru configurare: `journal\_mode`, `synchronous`, `foreign\_keys`, `wal\_autocheckpoint` etc. citeturn3search4

\*\*WAL vs. rollback journal.\*\* WAL permite citiri concurente în timp ce se scrie; conversia se face cu `PRAGMA journal\_mode=WAL;`. Modul implicit este `DELETE`. WAL este persistat pe fișier (rămâne activ după redeschidere), are un fișier separat `-wal`, iar checkpoint‑urile (mutarea paginilor din jurnal în fișierul principal) pot fi automatizate. citeturn3search1turn0search17

\*\*Chei străine (FK).\*\* Un detaliu adesea omis de începători: \*\*în SQLite, constrângerile de tip foreign key nu sunt aplicate dacă nu sunt activate\*\*, tipic prin `PRAGMA foreign\_keys=ON;` pe fiecare conexiune. Practic, trebuie să te asiguri că driverul sau ORM‑ul (Sequelize) pornește explicit enforcement‑ul, altfel ștergerile/inserările pot lăsa date orfane fără să observe nimeni. Documentația oficială precizează introducerea suportului FK în 3.6.19 și faptul că enforcement‑ul se activează explicit. citeturn0search7

\*\*Synchronous & durability.\*\* În WAL, `PRAGMA synchronous=NORMAL` e o alegere frecventă pentru echilibru între performanță și siguranță; `FULL` oferă garanții mai puternice dar presupune `fsync()` la fiecare commit. (Pragmatica exactă depinde de mediu; pe Android, ghidurile oficiale recomandă WAL + `synchronous=NORMAL` pentru performanță bună.) citeturn0search19

4. Sequelize: de la modele (models) la asocieri (associations)

\*\*De ce un ORM?\*\* Sequelize oferă o mapare obiect–relațională (object‑relational mapping) peste SQL, standardizând operațiile CRUD și expresivitatea interogărilor. În plus, definește clar \*\*modelul canonic\*\* al datelor (clase JS/TS), oferă \*\*validări\*\* și \*\*hook‑uri\*\* pe ciclul de viață, și expune \*\*asocieri\*\* care reflectă relațiile: `hasOne`, `hasMany`, `belongsTo`, `belongsToMany (through)`.

\*\*Asocieri de bază.\*\*

- `hasOne` / `belongsTo` pentru 1–1 (de ex. un profil pentru un membru).

- `hasMany` / `belongsTo` pentru 1–N (de ex. un club are mai multe evenimente).

- `belongsToMany` pentru M–N, \*\*cu tabel de legătură\*\* („through”), care poate avea la rândul lui atribute (de ex. rolul unui membru în club). Documentația oficială face clar modul de declarare și efectele asupra metodelor generate (e.g. `user.getProjects()`). citeturn3search5turn3search2turn3search8

\*\*Tipuri de date și particularități SQLite.\*\* SQLite are \*\*typing dinamic\*\* la nivel de coloană; Sequelize mapează tipuri comune (`STRING`, `INTEGER`, `REAL`, `BOOLEAN`, `DATE`, `DATEONLY`, `JSON` – ultimul cu emulare pe SQLite). Timestamps recomandate: \*\*RFC 3339\*\* (ISO‑8601 profile) pentru portabilitate și claritate. citeturn3search0

\*\*Constrângeri și politici de ștergere/actualizare.\*\* Prin opțiunile de asociere (`onDelete`, `onUpdate`, `constraints`), decidem comportamentul semantic: `CASCADE` (propagăm), `RESTRICT` (interzicem), `SET NULL` (de‑referențiem). Atenție: enforcement‑ul în SQLite depinde de `PRAGMA foreign\_keys=ON;` – testați explicit! citeturn0search7

5. Tranzacții (transactions) și proprietăți ACID în context SQLite

Literatura clasică definește un cadru riguros pentru tranzacții și recuperare (recovery): atomicitate, consistență, izolare, durabilitate. Articolul „Principles of transaction‑oriented database recovery” (ACM Computing Surveys, 1983) rămâne o referință pentru concepte (journaling, checkpointing, redo/undo). În practică, cu SQLite obținem ACID pe un singur fișier, cu o politică de blocare „single‑writer” dar \*\*multi‑reader\*\* în WAL; tranzacțiile scurte și bine delimitate sunt esențiale pentru concurență sănătoasă. citeturn0search4turn0search11

În aplicațiile Node/Sequelize, tranzacțiile se gestionează la nivel de ORM (`sequelize.transaction(async t => { ... })`). Modelarea corectă a \*\*unității de lucru\*\* (unit of work) – adică ce schimbări trebuie să reușească sau să eșueze împreună – previne stări parțiale și defecte greu de reprodus.

6. „N+1 problem”, încărcare lazy vs. eager și cum evităm runde inutile

„N+1 problem” apare atunci când, pentru a încărca N entități și relațiile lor, aplicația face N interogări suplimentare (de exemplu, un `SELECT` per rând pentru a încărca membrii unui club). Soluția canonică este \*\*eager loading\*\* (`include` în Sequelize) și proiecții selective (`attributes`). Există chiar lucrări care propun \*\*refactorizare automată\*\* pentru aplicații ORM (inclusiv Sequelize), eliminând modele N+1 prin analiză statică și generare de interogări compuse. citeturn0search16

Recomandare practică: începeți orice interogare complexă cu o variație a întrebării „Care este payload‑ul minim util?” (de ex. nume club + număr membri) și adăugați treptat relații/coloane, măsurând efectul. Eager loading este o unealtă, nu un scop: proiecția în exces poate produce payload mare și planuri mai puțin eficiente.

7. Migrații (migrations) și seeders: reproducibilitate și control al schimbării

Migrațiile descriu \*\*evoluția schemei\*\* în pași aplicabili/rollback‑abili (`up/down`). Într‑un context didactic, recomand separarea clară între: (1) schema de bază (L1), (2) adăugări cu indecși, atribute pe tabele de legătură (L2), (3) mecanisme avansate (tranzacții, validări complexe, seeders pentru seturi de date mari) (L3). Fiecare pas trebuie să fie \*\*determinist\*\*, idempotent la nivel de \*deployment\* și \*\*reproductibil\*\* în medii de test (în special când folosim `sqlite::memory:` pentru unit/integration).

Seeders (populate inițială) transformă „exemplele” în \*\*fixtures\*\*: date minime, dar realiste, care fac testele robuste. Încercați să evitați seed‑uri implicite în cod; folosiți scripturi CLI reproduse în CI.

8. Indexare, PRAGMA‑uri și performanță (cu SQLite în minte)

În SQLite, un index bine ales (ex. `UNIQUE` pe `Club.name`) este adesea suficient pentru fluxurile OLTP didactice. Pentru interogări frecvente cu filtrare pe două coloane, un \*\*index compus\*\* ajută; pentru M:N, verificați dacă tabelele de legătură au indexe pe ambele FK.

La nivel de runtime, \*\*WAL\*\* + `synchronous=NORMAL` sunt combinații rezonabile pentru medii de test/dev; pentru încărcări mai intense (mobile, I/O sensibil), literatura arată cum jurnalizarea influențează amplificarea scrierilor și latența, inclusiv optimizări prin SQL statement logging (studiu PVLDB). citeturn5search1turn5search0

9. Calitatea datelor: validări multilayer și hook‑uri

Validați la două niveluri: \*\*aplicație\*\* (Sequelize validators/custom checks) și \*\*bază\*\* (CHECK, NOT NULL, UNIQUE, FK). În Sequelize, hook‑urile (`beforeCreate`, `afterUpdate`) pot asigura invariante (de exemplu, normalizarea e‑mailurilor) sau auditarea modificărilor. În SQLite, unele tipuri de verificări (ex. regex sofisticate) sunt mai potrivite la nivel de aplicație pentru portabilitate.

10. Testabilitate și reproducere: in‑memory, tranzacții pe test, fixturi

O strategie robustă pentru teste: pornește o DB \*\*in‑memory\*\* (`sqlite::memory:`) sau un fișier temporar, rulează `sequelize.sync({ force: true })`, aplică seed‑uri minime, rulează testele (Vitest/Jest) și \*\*șterge\*\* mediul. Pentru teste care verifică enforcement‑ul FK, asigură‑te că pornirea DB activează \*\*`PRAGMA foreign\_keys=ON`\*\* (de preferință în codul de inițializare al ORM). citeturn0search7

Separă testele de \*\*unitate\*\* (validators, pure functions pentru mapări DTO) de testele de \*\*integrare\*\* (queries reale cu DB). Pentru problema N+1, scrie un test metrice: numărul de interogări (eventual cu un „query logger”) nu trebuie să depășească o limită când faci eager loading.

11. Strategii VSL (Very Short Loop) cu GitHub Copilot / LLM‑uri

Folosește AI‑assist doar acolo unde accelerează munca fără a înlocui gândirea de modelare. Trei locuri sigure:

1) \*\*Declarațiile de modele\*\*: Copilot poate genera scheletul `Club/Member/Registration/Event` cu `belongsToMany` corect, incluzând `through` și `foreignKey`/`otherKey`.

2) \*\*Migrații\*\*: generează scaffold‑ul `up/down` și completează‑l manual cu \*\*constraint‑uri\*\* și \*\*indexe\*\*; rulează imediat în `sqlite::memory:`.

3) \*\*Teste\*\*: Copilot poate propune cazuri negative (violări FK, UNIQUE), dar tu stabilești \*\*invariantele\*\* și verșiunile finale.

Bucle scurte în practică: scrii 10–20 linii (sau ceri o completare punctuală), rulezi testele, ajustezi. Evită „prompturi macro” care cer „aplicația completă” – îți vor produce mult cod de colț, greu de întreținut.

12. Legarea noțiunilor: din teorie în laborator

În laboratorul din Partea 2 vei porni de la un schelet Node/Express + Sequelize + SQLite, vei activa `foreign\_keys`, vei defini modele și asocieri (`belongsToMany` pentru M:N), vei rula migrații/seed‑uri, vei scrie interogări cu `include` (eager loading) și vei demonstra tranzacții atomice pentru înscrierea unui membru la un club (creare Member + Registration + actualizare contor în Club), totul testat automat cu Vitest și Jest. Apoi vei măsura empiric diferența între o interogare N+1 și un eager loading corect, pentru a înțelege concret „de ce” contează modelarea corectă. citeturn0search16

— — —

Note bibliografice (citații-cheie din text)

• Modelul relațional și normalizarea: Codd (1970) și Fagin (1977). citeturn0search2turn6search1

• Răspândire/arch. SQLite + WAL: PVLDB 2022 (Gaffney et al.). citeturn0search11

• WAL și `journal\_mode`: documentație SQLite. citeturn3search1

• Foreign keys în SQLite: documentație oficială. citeturn0search7

• Timestamps interoperabile: RFC 3339. citeturn3search0

• N+1 și refactorizare automată (Sequelize): ASE 2022. citeturn0search16

• I/O și jurnalizare în SQLite (performanță): ACM 2016, PVLDB 2017. citeturn5search1turn5search0

Bibliografie (stil APA 7 — cu DOI)

Codd, E. F. (1970). A relational model of data for large shared data banks. \*Communications of the ACM, 13\*(6), 377–387. https://doi.org/10.1145/362384.362685

Fagin, R. (1977). Multivalued dependencies and a new normal form for relational databases. \*ACM SIGMOD Record, 7\*(3), 262–266. https://doi.org/10.1145/320557.320571

Gaffney, K. P., Hipp, D. R., Kennedy, W. A., Mistachkin, J., Nagan, D., & Owens, R. (2022). SQLite: Past, present, and future. \*Proceedings of the VLDB Endowment, 15\*(12), 3535–3548. https://doi.org/10.14778/3554821.3554842

Härder, T., & Reuter, A. (1983). Principles of transaction-oriented database recovery. \*ACM Computing Surveys, 15\*(4), 287–317. https://doi.org/10.1145/289.291

Klyne, G., & Newman, C. (2002). \*Date and time on the Internet: Timestamps\* (RFC 3339). RFC Editor. https://doi.org/10.17487/RFC3339

Park, J.-H., Oh, G., & Lee, S.-W. (2017). SQL statement logging for making SQLite truly lite. \*Proceedings of the VLDB Endowment, 11\*(4), 513–525. https://doi.org/10.1145/3164135.3164146

Tuan, D. Q., Cheon, S., & Won, Y. (2016). On the I/O characteristics of the SQLite transactions on Android platform. In \*Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys ’16)\*. https://doi.org/10.1145/2897073.2897093

Xue, Y., Kolekar, S., Liu, Y., Gupta, N., & Ryoo, J. (2022). Reformulator: Automated refactoring of the N+1 problem in database-backed applications. In \*Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)\* (pp. 1–12). https://doi.org/10.1145/3551349.3556911