Academia Tehnică Militară "Ferdinand I"

Bucureşti

Facultatea de Sisteme Informatice şi Securitate Cibernetică

Specializarea Calculatoare şi Sisteme Informatice Pentru Apărare şi Securitate Naţională

Proiect

Proiectarea Sistemelor De Operare

Server Baze de Date Multithreaded

Sistem de Gestiune a Bazelor de Date cu Suport pentru Conexiuni Simultane

Autori:

Patricia Bărbută
Ştefan Ungureanu

Grupa: C113D

9 ianuarie 2026

Rezumat

Acest document prezintă în detaliu implementarea unui sistem complet de gestiune a bazelor de date relaţionale, dezvoltat în limbajul C++ cu utilizarea standardului C++17. Sistemul implementează o arhitectură client-server robustă, capabilă să gestioneze conexiuni simultane de la mai mulţi clienţi prin intermediul mecanismelor de sincronizare oferite de biblioteca pthread. Implementarea suportă trei tipuri fundamentale de date (numere întregi, şiruri de caractere cu lungime variabilă şi date calendaristice) şi oferă un set complet de operaţii pentru manipularea datelor, incluzând inserare, actualizare, ştergere şi interogare cu suport pentru filtrare şi sortare. Documentul explică în profunzime deciziile de design arhitectural, mecanismele de sincronizare utilizate pentru asigurarea integrităţii datelor în mediul multithreaded, precum şi detaliile de implementare ale fiecărei componente a sistemului.

Cuprins

1 Introducere

1.1 Contextul Proiectului

În era modernă a tehnologiei informaţiei, sistemele de gestiune a bazelor de date reprezintă componente fundamentale ale oricărei infrastructuri software. De la aplicaţii web simple până la sisteme enterprise complexe, capacitatea de a stoca, organiza şi interoga eficient volume mari de date structurate este esenţială. Proiectul de faţă se propune să implementeze de la zero un astfel de sistem, oferind o perspectivă profundă asupra mecanismelor interne care stau la baza funcţionării sistemelor de baze de date comerciale.

Implementarea unui server de baze de date propriu prezintă multiple provocări tehnice interesante. Prima provocare constă în gestionarea corectă a accesului concurent la date. Când mai mulţi clienţi încearcă să modifice sau să citească simultan aceleaşi date, sistemul trebuie să asigure că operaţiile se desfăşoară corect şi că integritatea datelor este menţinută. A doua provocare majoră este reprezentată de necesitatea de a oferi un protocol de comunicare eficient între clienţi şi server, care să permită transmiterea comenzilor şi rezultatelor într-un mod fiabil şi performant.

1.2 Obiectivele Sistemului

Sistemul dezvoltat îşi propune să atingă mai multe obiective tehnice specifice. În primul rând, serverul trebuie să fie capabil să accepte şi să gestioneze conexiuni simultane de la mai mulţi clienţi, fiecare client putând lucra independent cu propriile baze de date şi tabele. Această capacitate de concurenţă este implementată prin crearea unui thread separat pentru fiecare client conectat, thread-uri care comunică între ele doar prin structuri de date partajate protejate corespunzător.

Al doilea obiectiv major constă în implementarea unui model de date coerent şi flexibil. Sistemul suportă trei tipuri fundamentale de date care acoperă majoritatea cazurilor de utilizare practice. Numerele întregi permit stocarea identificatorilor, cantităţilor şi altor valori numerice discrete. Şirurile de caractere cu lungime variabilă permit stocarea textelor, numele persoanelor, descrierilor şi altor informaţii textuale. Tipul de dată calendaristică permite gestionarea temporală a informaţiilor, fiind util pentru înregistrarea momentelor de creaţie, modificare sau expirare a datelor.

Cel de-al treilea obiectiv se referă la implementarea unui set complet de operaţii de manipulare a datelor. Sistemul oferă nu doar operaţiile CRUD de bază (Create, Read, Update, Delete), ci şi funcţionalităţi avansate precum filtrarea datelor pe baza unor condiţii complexe şi sortarea rezultatelor după diverse criterii. Aceste capabilităţi transformă sistemul dintr-un simplu mecanism de stocare într-un instrument puternic de interogare şi analiză a datelor.

1.3 Tehnologii şi Instrumente Utilizate

Implementarea sistemului se bazează pe un set de tehnologii moderne şi bine stabilite. Limbajul C++ a fost ales pentru capacităţile sale de control la nivel scăzut şi pentru suportul excelent al programării concurente. Standardul C++17 aduce îmbunătăţiri importante în ceea ce priveşte gestionarea memoriei şi oferirea unor primitive moderne pentru lucrul cu thread-uri şi mutexuri.

Pentru comunicarea în reţea, sistemul utilizează API-ul Berkeley Sockets, un standard de facto pentru programarea la nivel de socket pe sistemele Unix şi Linux. Această interfaţă oferă control complet asupra comunicării TCP/IP, permiţând implementarea unui protocol personalizat optimizat pentru nevoile specifice ale aplicaţiei.

Sincronizarea între thread-uri este realizată prin combinarea mai multor mecanisme complementare. Biblioteca pthread oferă mutexuri pentru protecţia exclusivă a resurselor partajate, precum şi semafoare pentru limitarea numărului de clienţi conectaţi simultan. Standardul C++17 aduce clase wrapper moderne peste aceste primitive, făcând codul mai sigur şi mai uşor de înţeles.

Persistenţa datelor este asigurată prin utilizarea directă a apelurilor de sistem POSIX pentru operaţii cu fişiere. Această abordare de nivel scăzut oferă control maxim asupra modului în care datele sunt scrise pe disc şi permite optimizări specifice pentru paternurile de acces ale aplicaţiei.

2 Arhitectura Sistemului

2.1 Viziunea de Ansamblu

Arhitectura sistemului este organizată pe mai multe niveluri distincte, fiecare cu responsabilităţi bine definite. Această separare clară a responsabilităţilor urmează principiile ingineriei software moderne şi facilitează atât înţelegerea sistemului, cât şi extensia viitoare cu noi funcţionalităţi.

La cel mai de sus nivel se află componenta client, care oferă interfaţa de interacţiune cu utilizatorul. Această componentă este responsabilă pentru citirea comenzilor de la utilizator, transmiterea lor către server şi afişarea rezultatelor primite. Clientul implementează şi o serie de funcţionalităţi locale, precum sistemul de ajutor integrat şi comenzi de utilitate care nu necesită interacţiune cu serverul.

Nivelul intermediar este reprezentat de componenta server, care gestionează aspectele legate de reţea şi concurenţă. Serverul acceptă conexiuni de la clienţi, creează thread-uri dedicate pentru fiecare client şi coordonează accesul la resursele partajate. Această componentă implementează şi mecanismele de limitare a numărului de conexiuni simultane pentru a preveni suprasarcina sistemului.

La baza arhitecturii se află motorul de baze de date, care implementează logica de business a sistemului. Această componentă este responsabilă pentru parsarea comenzilor SQL, validarea lor, executarea operaţiilor solicitate şi returnarea rezultatelor. Motorul

gestionează şi structura logică a bazelor de date şi tabelelor, menţinând informaţii despre schema fiecărui tabel şi asigurând consistența datelor.

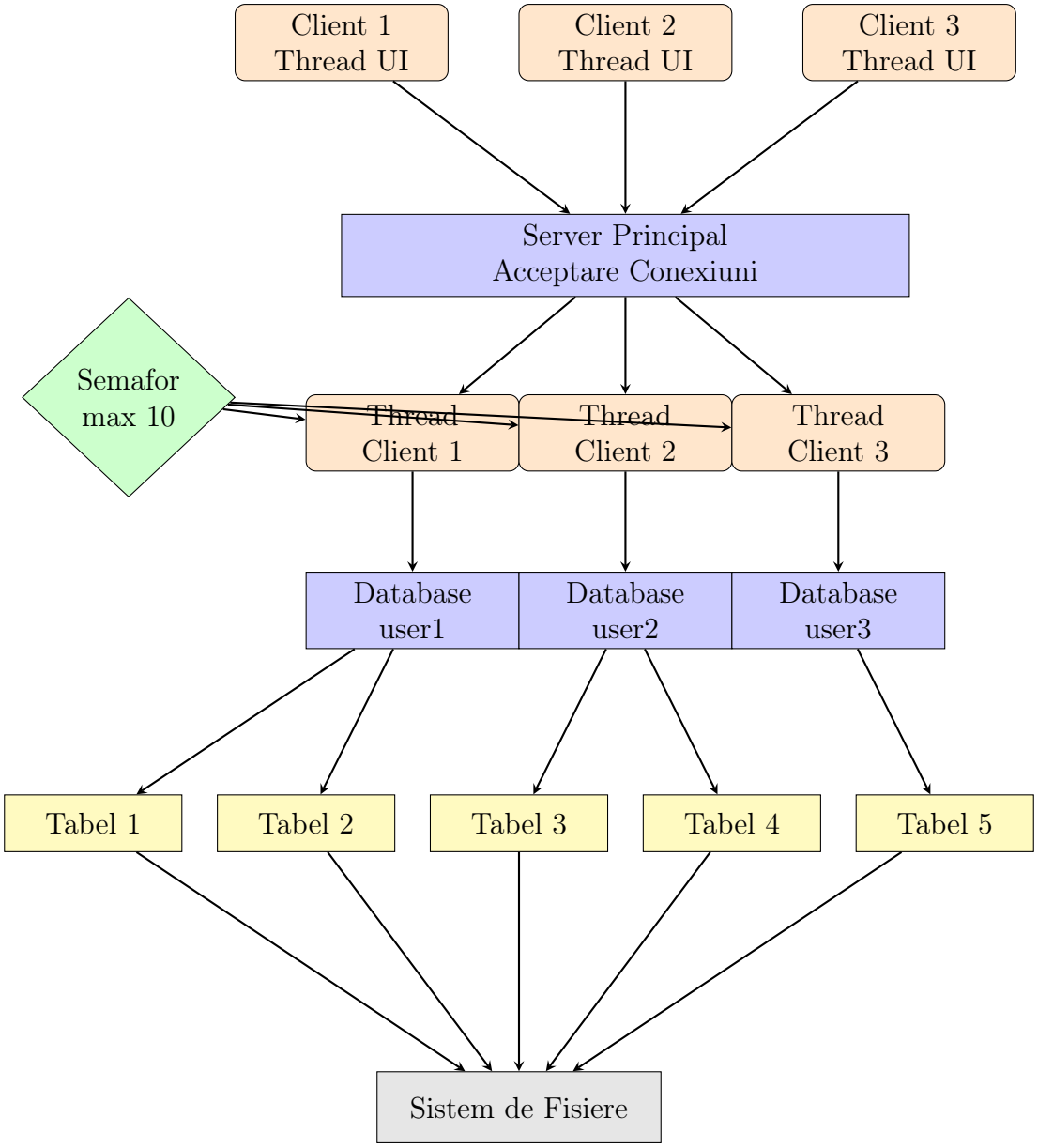

Figura 1: Arhitectura generală a sistemului

2.2 Componenta Client

Componenta client este construită în jurul clasei `Client`, care encapsulează toată logica necesară pentru comunicarea cu serverul. Această clasă utilizează paradigma de programare orientată pe obiecte pentru a organiza datele și comportamentul într-o manieră clară și intuitivă.

Atributul principal al clasei este socket-ul de comunicare, reprezentat de un descriptor de fișier întreg. În sistemele Unix și Linux, socket-urile sunt tratate similar cu fișierele obișnuite, ceea ce permite utilizarea acelorași funcții de citire și scriere pentru comunicarea

în rețea. Acest descriptor este inițializat în constructor prin apelul funcției `socket()`, care creează un nou endpoint de comunicare în domeniul Internet (AF_INET) cu semantică de flux (SOCK_STREAM), ceea ce înseamnă utilizarea protocolului TCP.

Structura `sockaddr_in` stochează informațiile necesare pentru identificarea serverului în rețea. Această structură conține familia de adrese (IPv4), portul pe care ascultă serverul (convertit în format network byte order prin funcția `htons()`) și adresa IP efectivă a serverului. Conversia adresei IP din format textual în format binar este realizată prin funcția `inet_pton()`, care validează și transformă simultan șirul de caractere într-o reprezentare numerică pe 32 de biți.

Procesul de conectare la server începe odată ce socket-ul este configurat corect. Apelul funcției `connect()` inițiază secvența de handshake TCP în trei pași (three-way handshake), stabilind o conexiune fiabilă între client și server. Această conexiune garantează că pachetele trimise vor ajunge în ordinea corectă și fără erori la destinație.

Un aspect important al implementării clientului este protocolul de autentificare inițială. Imediat după stabilirea conexiunii, clientul trimite serverului numele utilizatorului curent, obținut din variabila de mediu USER. Această informație este utilizată de server pentru a crea un spațiu de lucru izolat pentru fiecare utilizator, permițând mai multor utilizatori să lucreze simultan pe același server fără a-și interfera reciproc datele.

Bucla principală de execuție a clientului este implementată în metoda `run()`, care reprezintă inima funcționalității clientului. Această buclă rulează continuu, citind comenzi de la utilizator, procesându-le și afișând rezultatele, până când utilizatorul decide să încheie sesiunea prin comanda EXIT sau prin închiderea forțată a programului.

La fiecare iterație a buclei, clientul afișează un prompt personalizat care include numele utilizatorului, oferind astfel un feedback vizual clar despre contextul curent de lucru. Prompt-ul este construit dinamic prin concatenarea prefixului fix cu numele utilizatorului obținut din sistemul de operare, rezultând un format familiar și intuitiv similar cu cel al shell-urilor Unix.

Citirea comenzilor de la utilizator este realizată prin funcția `std::getline()`, care citește o linie completă de text până la întâlnirea caracterului newline. Această abordare permite utilizatorului să introducă comenzi complexe care pot conține spații și caractere speciale fără probleme de parsare. În cazul în care citirea eșuează (de exemplu, la întâlnirea EOF prin apăsarea Ctrl+D), bucla se încheie în mod elegant, permițând terminarea controlată a programului.

Înainte de a trimite comanda către server, clientul verifică dacă aceasta nu este o comandă locală care poate fi procesată direct. Sistemul implementează două astfel de comenzi locale care nu necesită comunicare cu serverul. Prima comandă este `clear`, care curăță ecranul terminalului pentru o mai bună lizibilitate. Implementarea acestei comenzi utilizează tehnica de fork și exec specifică sistemelor Unix, creând un proces copil care execută utilitarul system `clear`.

Mecanismul de fork creează o copie exactă a procesului curent, inclusiv cu toate descriptoarele de fișiere deschise și variabilele de mediu. Procesul copil execută apoi comanda clear prin înlocuirea completă a spațiului său de adrese cu imaginea executabilului clear. Între timp, procesul părinte (clientul nostru) așteaptă terminarea procesului copil prin

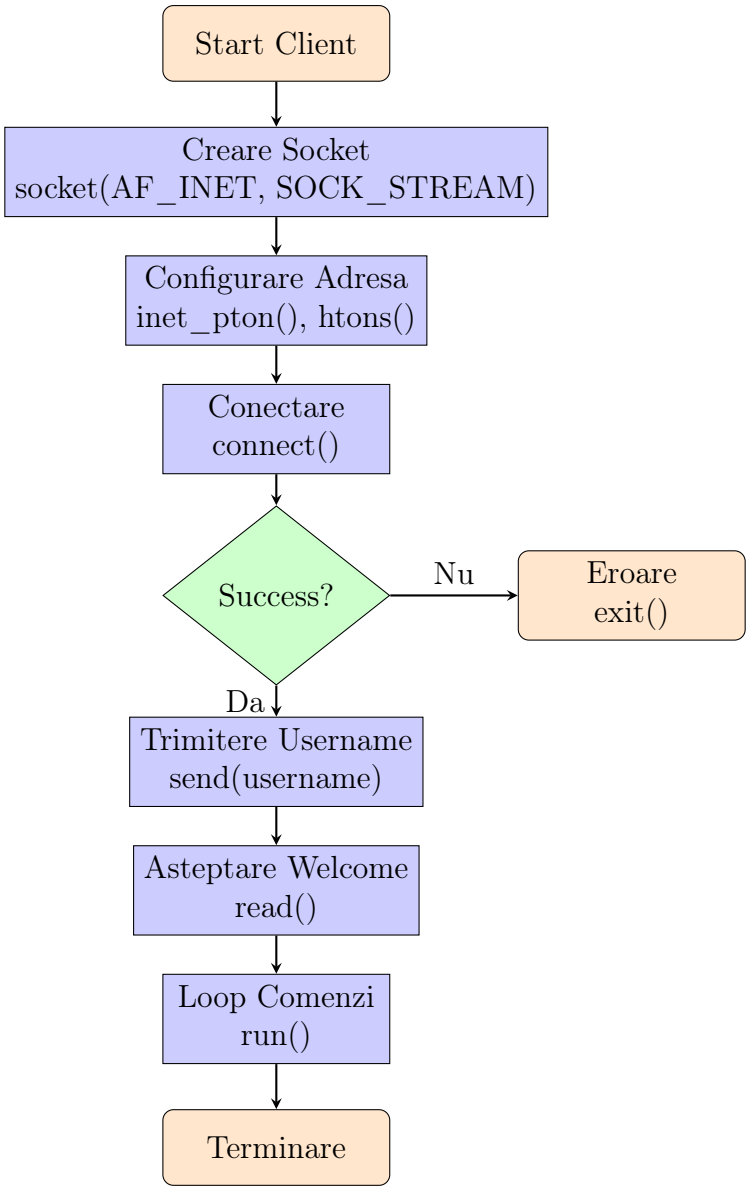

Figura 2: Fluxul de inițializare al clientului

apelul funcției `wait()`, asigurându-se astfel că ecranul este curățat complet înainte de a continua execuția.

A doua comandă locală este `help`, care oferă utilizatorului acces la documentația sistemului. Această comandă deschide fișiere text care conțin explicații detaliate despre diverse aspecte ale limbajului de interogare și ale funcționalităților sistemului. Implementarea utilizează din nou mecanismul de fork, dar de această dată procesul copil execută programul `less`, un pager popular care permite navigarea interactivă prin fișiere text lungi.

Pentru comenzile care nu sunt procesate local, clientul le transmite serverului prin socketul TCP. Funcția `send()` transmite întregul șir de caractere al comenzii, inclusiv lungimea acesteia. Protocolul TCP garantează că datele vor ajunge la destinație în ordinea corectă și fără erori, eliminând necesitatea implementării unui protocol de corectare a erorilor la nivelul aplicației.

După trimiterea comenzii, clientul intră în stare de așteptare, blocându-se în apelul funcției `read()` până când serverul trimite răspunsul. Această abordare sincronă simplifică foarte mult logica clientului, eliminând necesitatea gestionării multiple cereri concurente. În timp ce clientul așteaptă, thread-ul său de execuție este suspendat de sistemul de operare, permitând altor procese să utilizeze procesorul.

Când răspunsul sosește de la server, clientul îl afișează utilizatorului prin scrierea directă pe descriptorul de ieșire standard. Utilizarea funcției `write()` de nivel scăzut în loc de iostream-urile C++ oferă un control mai fin asupra bufferizării și asigură că mesajele sunt afișate imediat, fără întârzieri cauzate de bufferizarea automată.

2.3 Componenta Server - Gestionarea Conexiunilor

Serverul reprezintă componenta centrală a sistemului, fiind responsabil pentru orchestrarea tuturor interacțiunilor dintre clienți și motorul de baze de date. Implementarea serverului urmează modelul arhitectural producer-consumer, unde thread-ul principal (producer) acceptă conexiuni noi și le distribuie thread-urilor worker (consumers) care le procesează.

Inițializarea serverului începe cu crearea socket-ului de ascultare, care va fi folosit pentru acceptarea conexiunilor de la clienți. Spre deosebire de socket-urile clientului care inițiază conexiuni, acest socket este configurat să asculte pentru conexiuni incoming. Opțiunea SO_REUSEADDR este setată pentru a permite reutilizarea imediată a portului chiar dacă există încă conexiuni în starea TIME_WAIT de la rulări anterioare ale serverului. Această configurație este esențială pentru dezvoltare și testare, când serverul este pornit și oprit frecvent.

Operațiunea de bind asociază socket-ul cu o adresă IP și un port specific pe care serverul va asculta. Utilizarea constantei INADDR_ANY permite serverului să accepte conexiuni pe toate interfețele de rețea disponibile, nu doar pe o interfață specifică. Aceasta înseamnă că serverul poate fi accesat atât prin adresa de loopback (127.0.0.1) pentru testare locală, cât și prin adrese IP externe pentru acces în rețea.

Apelul funcției `listen()` marchează socket-ul ca fiind pasiv, adică pregătit să accepte conexiuni. Parametrul backlog specifică numărul maxim de conexiuni care pot aștepta în coadă să fie acceptate. O valoare prea mică poate cauza respingerea conexiunilor legitime în perioade de trafic intens, în timp ce o valoare prea mare poate consuma inutil resurse de memorie. Valoarea de 3 utilizată în implementarea noastră reprezintă un compromis rezonabil pentru un sistem de dimensiuni medii.

Limitarea numărului de clienți conectați simultan este realizată prin intermediul unui semafor POSIX, inițializat cu valoarea maximă de 10 conexiuni simultane. Semaforul funcționează ca un contor protejat împotriva race conditions, permițând decrementarea atomică la fiecare conexiune nouă și incrementarea la deconectarea unui client. Când semaforul ajunge la zero, thread-ul principal se blochează în apelul `sem_wait()`, refuzând implicit noi conexiuni până când un client existent se deconectează.

Pentru fiecare client acceptat, serverul creează un nou thread de execuție care va gestiona în mod independent toate interacțiunile cu acel client. Această abordare de un thread

per client este simplă și robustă, izolând complet procesarea fiecărui client și eliminând necesitatea multiplexării manual a mai multor conexiuni pe același thread. Thread-ul este creat prin clasa `std::thread` din C++11, care oferă o interfață modernă și type-safe peste primitivele pthread de nivel scăzut.

Decizia de a detașa thread-ul imediat după crearea sa este motivată de dorința de a permite serverului să continue acceptarea de noi conexiuni fără a aștepta terminarea thread-urilor existente. Un thread detașat rulează independent, sistemul de operare eliberând automat resursele sale când se termină, fără a fi nevoie de apelul explicit al funcției `join()` din thread-ul părinte.

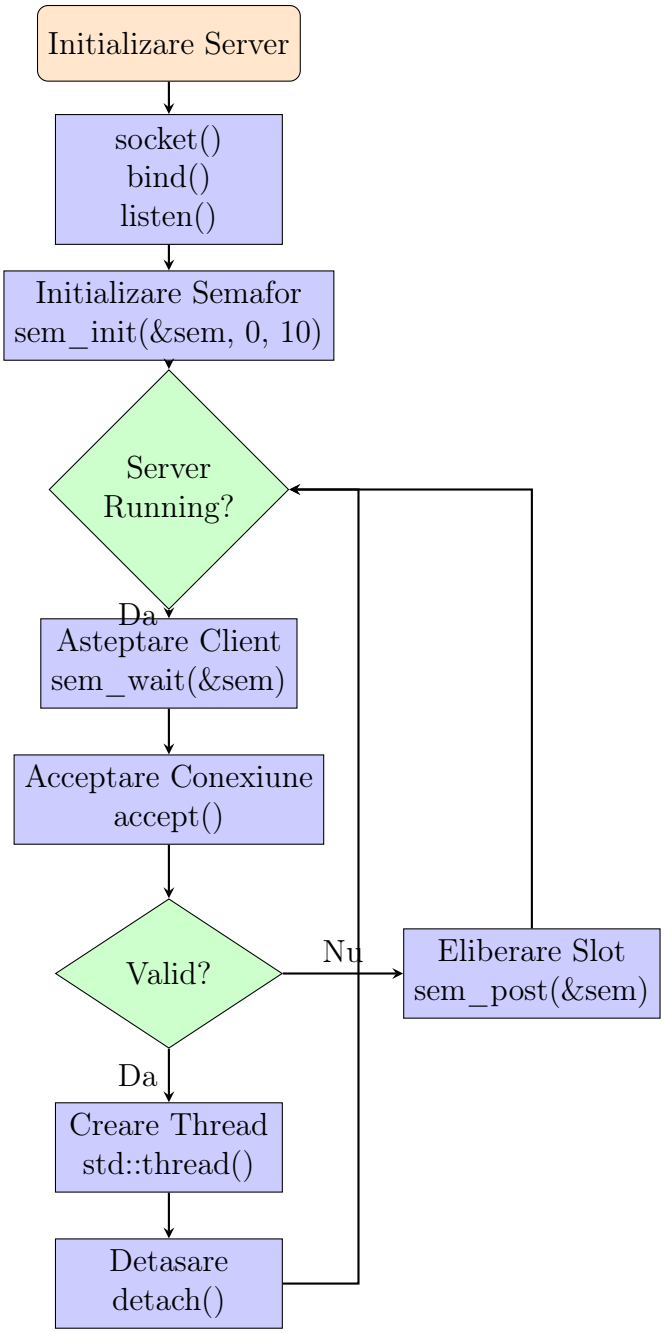

Figura 3: Bucla principală de acceptare a conexiunilor

Fiecare thread client începe prin primirea numelui de utilizator trimis de client imediat

după stabilirea conexiunii. Această informație este crucială pentru izolarea spațiilor de lucru ale diferiților utilizatori. Pe baza numelui de utilizator, thread-ul creează o nouă instanță a clasei Database, care va gestiona toate operațiunile pe baze de date pentru acel utilizator specific. Fiecare instanță Database menține propriul set de tabele încărcate și propria structură de directoare pe disc, asigurând separarea completă a datelor între utilizatori.

După autentificare, thread-ul trimite clientului un mesaj de bun venit și intră în bucla sa principală de procesare. Această buclă citește comenzi de la client, le execută prin intermediul instanței Database și trimite înapoi rezultatele. Procesarea este complet sincronă - thread-ul procesează o singură comandă la un moment dat, așteptând finalizarea executării înainte de a citi următoarea comandă.

Această simplitate a modelului de procesare este posibilă datorită arhitecturii multi-thread a serverului. Fiecare client are propriul său thread dedicat, astfel încât blocarea unui thread în așteptarea unei operații de I/O nu afectează procesarea celorlalți clienți. Această izolare completă între thread-uri simplifică foarte mult logica de sincronizare, fiind necesare mutex-uri doar pentru protejarea structurilor de date efectiv partajate.

Când un client se deconectează (fie prin trimiterea comenzii EXIT, fie prin închiderea conexiunii), thread-ul său efectuează o serie de operații de curățare. Mai întâi, instanța Database este ștearsă, ceea ce declanșează salvarea automată a tuturor modificărilor pendinte pe disc prin destructorul clasei. Apoi socket-ul de comunicare este închis, eliberând resursele de rețea asociate. În final, semaforul este incrementat prin `sem_post()`, semnalând că un slot de conexiune s-a eliberat și că serverul poate accepta un nou client.

2.4 Motorul de Baze de Date - Arhitectura Internă

Motorul de baze de date reprezintă nivelul cel mai complex al sistemului, fiind responsabil pentru întreaga logică de business. Această componentă este structurată în jurul clasei Database, care acționează ca un orchestrator central pentru toate operațiunile cu date. Fiecare instanță a acestei clase gestionează bazele de date și tabelele unui singur utilizator, menținând starea curentă a sesiunii și coordonând accesul la resurse.

Clasa Database menține o hartă (map) a tuturor tabelelor încărcate în memorie, indexate după o cheie compozită formată din numele bazei de date și numele tabelului. Această structură permite căutarea rapidă a tabelelor și evită conflictele de nume între tabele cu același nume din baze de date diferite. Harta este protejată de un mutex pentru a permite accesul concurent sigur din thread-uri diferite, deși în practica implementării noastre curente fiecare instanță Database este accesată doar de un singur thread.

Sistemul de fișiere utilizat pentru persistența datelor reflectă ierarhia logică a bazelor de date și tabelelor. Fiecare utilizator are un director dedicat în structura `.data/username/`, în care sunt create subdirectoare pentru fiecare bază de date. Această organizare permite izolarea completă a datelor între utilizatori și facilitează operațiile de backup și restaurare prin simpla copiere a directorului unui utilizator.

Procesarea comenzilor SQL urmează un pattern de routing bine definit. Metoda centrală `executeCommand()` primește comanda ca șir de caractere, o convertește la majuscule

pentru a permite comparații case-insensitive și o rutează către metoda specializată corespunzătoare. Acest design modular face adăugarea de noi comenzi extrem de simplă - este suficient să adaugi o nouă metodă de procesare și să extinzi lanțul de if-uri din router.

Înainte de executarea efectivă a comenzii, sistemul verifică dacă operația necesită o bază de date selectată. Comenzile la nivel de tabel (CREATE TABLE, INSERT, SELECT, etc.) nu pot fi executate fără ca utilizatorul să fi selectat mai întâi o bază de date prin comanda USE. Această verificare previne erori de execuție și oferă un mesaj clar utilizatorului despre ce trebuie să facă pentru a putea continua.

Parsarea comenzilor SQL este realizată prin tehnici simple dar eficiente de manipulare a șirurilor de caractere. Sistemul identifică cuvintele cheie prin căutarea lor în șirul comenzii, apoi extrage și procesează argumentele. De exemplu, pentru o comandă CREATE TABLE, sistemul identifică parantezele care delimitează definiția coloanelor, extrage această substring și o împarte în definiții individuale de coloane care sunt apoi parsate separat.

Gestionarea erorilor este realizată prin returnarea de mesaje descriptive sub formă de șiruri de caractere. Această abordare simplă dar eficientă permite transmiterea ușoară a informațiilor despre erori către client, unde pot fi afișate utilizatorului într-o formă lizibilă. Mesajele de eroare sunt prefixate cu "ERROR:" pentru a permite clientului să le distingă de rezultatele valide.

3 Sincronizare şi Concurenţă

3.1 Provocarea Accesului Concurent

Gestionarea corectă a accesului concurent la date reprezintă una dintre cele mai complexe aspecte ale implementării unui sistem de baze de date. Problema fundamentală apare atunci când două sau mai multe thread-uri încearcă să modifice simultan aceleaşi date. Fără mecanisme adecvate de sincronizare, astfel de situaţii pot duce la coruperea datelor, rezultate inconsistente sau chiar crash-uri ale aplicaţiei.

Sistemul nostru implementează o strategie de sincronizare pe mai multe niveluri, fiecare nivel protejând un anumit tip de resursă partajată. Această abordare ierarhică permite maximizarea concurenţei în scenariile unde thread-urile accesează resurse diferite, blocare fiind necesară doar când există conflict efectiv pentru aceeaşi resursă.

La nivelul cel mai de sus, un semafor limitează numărul total de clienţi care pot fi conectaţi simultan la server. Această limitare nu este doar una de sincronizare, ci şi o măsură de protecţie a resurselor sistemului. Fiecare client conectat consumă memorie pentru bufferele de comunicare, pentru starea sesiunii şi pentru datele încărcate. Limitarea numărului de clienţi previne epuizarea memoriei şi menţine sistemul funcţional chiar şi sub sarcină mare.

La nivelul intermediar, fiecare instanţă Database are propriul mutex care protejează harta tabelelor încărcate. Acest mutex este necesar deoarece operaţiile de creare sau ştergere de tabele modifică structura hărţii, iar astfel de modificări trebuie să fie atomice pentru a menţine consistența internă. Totuşi, în implementarea curentă unde fiecare utilizator are

propria instanţă Database accesată de un singur thread, această protecţie serveşte mai degrabă ca o măsură preventivă pentru extensii viitoare ale sistemului.

La nivelul cel mai de jos, fiecare tabel are propriul mutex care protejează operaţiile de citire şi scriere pe fişierul de date. Această granularitate fină a blocării permite thread-urilor să lucreze simultan pe tabele diferite, maximizând astfel paralelismul efectiv al sistemului. De exemplu, dacă un client execută un SELECT lung pe un tabel mare, alţi clienţi pot în acelaşi timp să insereze date în alte tabele fără să fie blocaţi.

Utilizarea clasei `std::lock_guard` pentru manipularea mutexurilor este o practică modernă de programare care exploatează principiul RAII (Resource Acquisition Is Initialization) din C++. Această clasă wrapper asigură că mutexul este automat eliberat când variabila lock_guard iese din scop, chiar dacă funcţia se încheie printr-o excepţie sau prin instrucţiunea return. Această garanţie elimină riscul de deadlock cauzat de uitarea eliberaţii manuale a mutexului.

3.2 Prevenirea Deadlock-urilor

Arhitectura ierarhică a sistemului de blocări contribuie esenţial la prevenirea deadlock-urilor. Un deadlock apare atunci când două sau mai multe thread-uri aşteaptă circular unul după celălalt pentru a elibera resurse, rezultând în blocarea definitivă a tuturor thread-urilor implicate. Printr-o ierarhie clară de blocări şi urmând regula de a obţine întotdeauna mutexurile în aceeaşi ordine, sistemul nostru evită complet această problemă.

Regula fundamentală este că mutexurile trebuie obţinute întotdeauna de la nivelurile superioare către cele inferioare ale ierarhiei, iar niciodată în sens invers. Un thread care obţine mai întâi mutexul unui tabel nu va încerca niciodată să obţină apoi mutexul Database, deoarece arhitectura metodelor face ca aceste situaţii să fie structural imposibile.

Mai mult, izolarea pe bază de utilizatori reduce semnificativ spaţiul potenţial de conflicte. Doi clienţi conectaţi sub nume de utilizator diferite lucrează cu instanţe Database complet separate, fără nicio resursă partajată între ei. Această separare completă elimină necesitatea de sincronizare între majoritatea thread-urilor, sincronizarea fiind necesară doar în cazul rar în care mulţi clienţi se conectează sub acelaşi nume de utilizator.

3.3 Scenarii de Concurenţă

Pentru a înţelege mai bine comportamentul sistemului în situaţii de concurenţă, să analizăm câteva scenarii tipice. În primul scenariu, doi clienţi diferiţi, conectaţi sub nume de utilizator diferite, execută operaţii pe propriile lor tabele. În acest caz, cele două thread-uri operează complet independent, fără nicio interacţiune sau blocare. Fiecare thread lucrează cu propria instanţă Database, accesând propriile tabele din propriile directoare pe disc.

Al doilea scenariu este mai interesant: doi clienţi conectaţi sub acelaşi nume de utilizator încearcă să modifice acelaşi tabel simultan. În acest caz, ambele thread-uri vor încerca să obţină mutexul tabelului. Primul thread care ajunge la apelul `lock()` va obţine mutexul

şi va proceda cu operaţia sa. Al doilea thread va fi blocat în apelul `lock()`, aşteptând ca primul thread să termine şi să elibereze mutexul.

Un al treilea scenariu implică operaţii de citire şi scriere pe acelaşi tabel. Chiar dacă citirea nu modifică datele, mutexul este oricum necesar pentru a preveni citirea unor date parțial modificate. Imaginați-vă ce s-ar întâmpla dacă un thread ar citi fișierul în timp ce altul scrie în el - ar putea obține o combinație bizară de date vechi și noi, rezultând într-o stare inconsistentă.

4 Implementarea Tipurilor de Date

4.1 Modelul de Date

Sistemul implementează un model de date relațional simplificat, inspirat de sistemele de baze de date clasice dar adaptat pentru nevoile specifice ale proiectului. Modelul suportă trei tipuri fundamentale de date care acoperă majoritatea scenariilor practice de utilizare, oferind un echilibru între simplitate și funcționalitate.

Tipul INT reprezintă numere întregi și este utilizat pentru stocare valorilor numerice discrete. Acest tip este ideal pentru identificatori unici, contoare, cantități și alte valori care nu necesită componenta fracționară. În implementarea internă, valorile INT sunt stocate ca șiruri de caractere în fișiere, dar sunt validate la inserare pentru a asigura că conțin doar cifre și opțional un semn minus pentru numere negative.

Tipul VARCHAR oferă flexibilitate maximă pentru stocarea datelor textuale de lungime variabilă. Specificarea unei lungimi maxime (de exemplu VARCHAR(100)) servește în principal ca documentație și validare, ajutând utilizatorii să înțeleagă limitările câmpului. În implementarea actuală, această limită nu este strict impusă la nivel de stocare, dar poate fi extinsă în versiuni viitoare pentru a preveni consumul excesiv de spațiu de stocare.

Tipul DATE permite reprezentarea datelor calendaristice într-un format standardizat (YYYY-MM-DD). Acest format ISO 8601 este ales pentru claritatea sa și pentru faptul că permite comparații lexicografice directe - comparând două date ca șiruri de caractere se obține același rezultat ca și când ar fi comparate ca date reale. Această proprietate simplifică foarte mult implementarea operațiilor de sortare și filtrare pe câmpuri de tip DATE.

4.2 Stocarea şi Reprezentarea Datelor

Sistemul utilizează un format de stocare bazat pe text pentru toate datele, indiferent de tipul lor. Această decizie de design are atât avantaje cât şi dezavantaje. Principalul avantaj este simplitatea şi portabilitatea - fişierele de date pot fi inspectate şi modificate manual cu orice editor de text, faciliteză debugging-ul şi permite exportul uşor al datelor. Dezavantajul principal este consumul mai mare de spaţiu şi viteza mai redusă de procesare comparativ cu un format binar optimizat.

Fiecare tabel este stocat în două fişiere separate. Fişierul cu extensia .db conţine datele efective, cu fiecare rând al tabelului reprezentat pe o linie în fişier.

Fişierul cu extensia .schema conţine definiţia structurii tabelului, listând fiecare coloană pe o linie separată împreună cu tipul său de date. Această informaţie este esenţială pentru încărcarea corectă a tabelului la pornirea serverului sau la prima utilizare a tabelului după ce a fost creat. Schema permite sistemului să valideze datele la inserare şi să interprete corect valorile la citire.

Separarea schemei de date prezintă mai multe avantaje importante. Permite modificarea structurii tabelului (adăugarea de noi coloane, de exemplu) fără a fi nevoie să rescrii întregul fişier de date. De asemenea, face ca valoarea lor să fie implicită NULL fără a fi necesară o migrare explicită a datelor. De asemenea, face că parsarea datelor să fie mai eficientă, deoarece tipurile coloanelor sunt cunoscute dinainte şi nu trebuie deduse din datele efective.

5 Operaţii pe Date

5.1 Inserarea Datelor - INSERT

Operaţia INSERT adaugă un nou rând de date într-un tabel existent. Procesul începe cu parsarea comenzii pentru a extrage numele tabelului şi lista de valori de inserat. Sistemul identifică mai întâi cuvântul cheie VALUES care separă numele tabelului de valorile efective. Tot ce se află între paranteze după VALUES este extras şi împărţit în valori individuale pe baza virgulelor.

Valorile extrase sunt apoi curăţate de spaţii şi ghilimele. Această curăţare este necesară deoarece utilizatorii pot introduce valorile în diverse formate - cu sau fără spaţii, cu ghilimele simple sau duble pentru şirurile de caractere. Sistemul acceptă toate aceste variaţii şi le normalizează la un format consistent.

După extragerea şi curăţarea valorilor, sistemul verifică dacă numărul de valori corespunde cu numărul de coloane din definiţia tabelului. Această validare este crucială pentru menţinerea integrităţii datelor - un rând incomplet sau cu prea multe valori ar corupe structura tabelului şi ar face imposibilă citirea corectă a datelor ulterioare.

Odată ce validarea este trecută cu succes, sistemul construieşte reprezentarea CSV a rândului. Valorile sunt concatenate cu virgule între ele, formând un șir de caractere care va fi scris în fișier. La sfârșitul șirului este adăugat caracterul newline pentru a separa acest rând de următoarele rânduri care vor fi adăugate ulterior.

Scrierea efectivă în fișier este realizată cu flag-ul O_APPEND, care garantează că datele vor fi adăugate la sfârșitul fișierului fără a suprascrie conținutul existent. Această operație este atomică la nivelul sistemului de operare pentru scrieri mici, asigurând că chiar dacă mai multe thread-uri scriu simultan în același fișier (situație care nu ar trebui să apară datorită mutexului), datele nu vor fi intercalate în mod corupt.

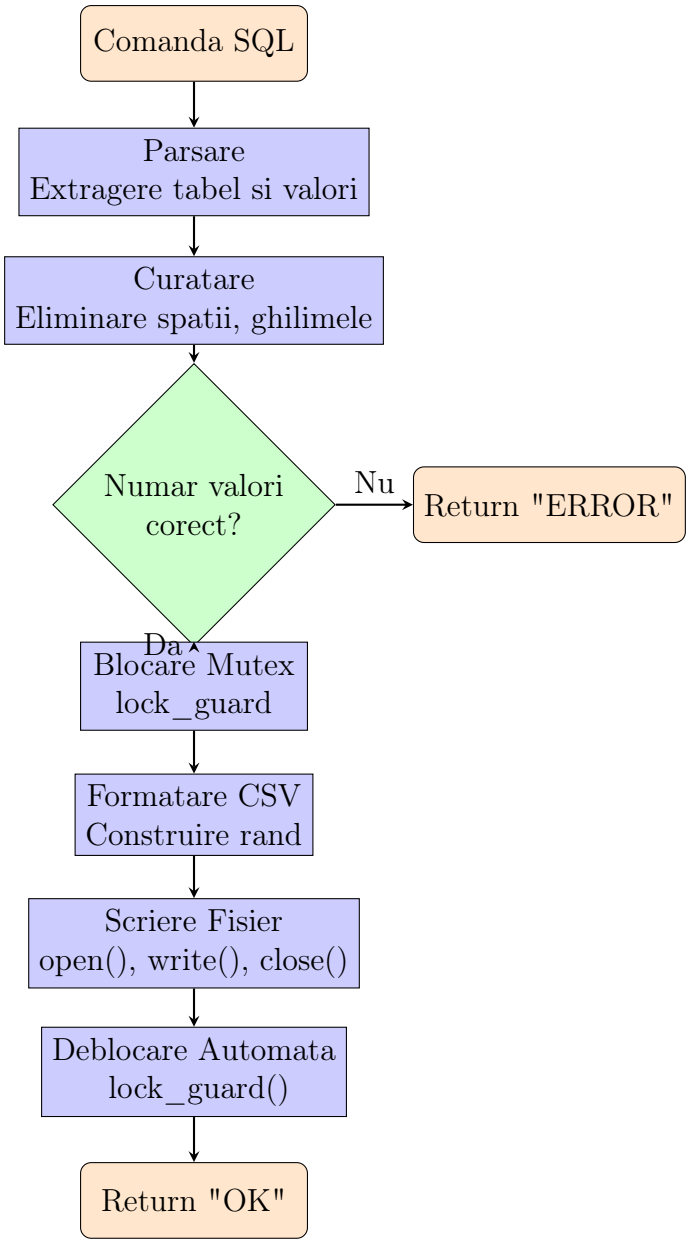

Figura 4: Fluxul operaţiei INSERT

5.2 Interogarea Datelor - SELECT

Operaţia SELECT este cea mai complexă dintre toate operaţiile implementate, oferind capabilităţi avan de filtrare şi sortare a datelor. Procesul începe cu identificarea clauzelor cheie din comandă: SELECT specifică ce coloane să fie afişate, FROM specifică tabelul din care se citesc datele, WHERE (opţional) specifică condiţiile de filtrare, iar ORDER BY (opţional) specifică criteriile de sortare.

Parsarea coloanelor solicitate distinge între cazul special asterisk (*) care indică selecţia tuturor coloanelor şi cazul în care sunt specificate coloane individuale. Pentru fiecare coloană specificată, sistemul verifică existenţa sa în schema tabelului şi determină indexul său pentru a putea extrage valoarea corectă din fiecare rând citit.

Întregul conţinut al fişierului este citit în memorie, apoi împărţit în linii individuale care reprezintă rândurile tabelului. Această abordare este eficientă pentru tabele de dimensiuni mici până la medii, dar ar trebui înlocuită cu o citire streaming pentru tabele foarte mari care nu încap în memorie.

Fiecare linie este apoi parsată pentru a extrage valorile individuale ale coloanelor. Parsarea fişierului este relativ simplă datorită formatului nostru strict - valorile sunt pur și simplu împărțite după virgule. Un sistem de producție ar trebui să gestioneze și cazuri speciale precum virgule care apar în interiorul valorilor (escaping), dar pentru scopurile noastre demonstrative, formatul simplu este suficient.

Evaluarea clauzei WHERE este delegată unei clase specializate ConditionParser care știe să interpreteze expresii booleene complexe. Acest parser suportă operatori de comparație (=, !=, <, >, <=, >=) și operatori logici (AND, OR) permițând construirea de condiții sofisticate. Pentru fiecare rând citit, parserul evaluează expresia în contextul valorilor acelui rând, returnând true dacă rândul satisface condiția și false altfel.

Rândurile care trec filtrul WHERE sunt adăugate într-un vector de rezultate care va fi eventual sortat și afișat. Dacă este specificată o clauză ORDER BY, vectorul este sortat folosind algoritmul standard sort din STL. Funcția de comparație este personalizată pentru a compara rândurile pe baza valorii coloanei specificate, în ordine ascendentă sau descendentă după cum este specificat.

5.3 Actualizarea Datelor - UPDATE

Operaţia UPDATE modifică valori existente în tabele pe baza unor criterii specificate. Spre deosebire de INSERT care adaugă întotdeauna la sfârşitul fişierului, UPDATE trebuie să citească întregul fişier, să modifice rândurile relevante şi apoi să rescrie întregul fişier cu conţinutul actualizat.

Parsarea comenzii UPDATE identifică trei componente esenţiale: numele tabelului, clauza SET care specifică ce coloană să fie modificată şi la ce valoare, şi clauza WHERE (opţională) care specifică ce rânduri să fie afectate. Lipsă clauzei WHERE înseamnă că toate rândurile tabelului vor fi actualizate, o operaţie puternică dar potenţial periculoasă.

Procesul de actualizare începe cu citirea completă a fişierului de date în memorie. Fiecare rând este parsat şi evaluat faţă de clauza WHERE (dacă există). Pentru rândurile care satisfac condiţia, valoarea coloanei specificate în clauza SET este înlocuită cu noua valoare. Rândurile care nu satisfac condiţia rămân nemodificate.

După ce toate rândurile au fost procesate, întregul conținut actualizat este rescris în fișier. Fișierul este deschis cu flag-ul O_TRUNC care îl golește complet înainte de scriere, asigurându-se astfel că nu rămân date vechi la sfârșitul fișierului în cazul în care conținutul nou este mai scurt decât cel vechi.

5.4 Ştergerea Datelor - DELETE

Operaţia DELETE elimină rânduri din tabele pe baza unor criterii specificate. Similar cu UPDATE, DELETE necesită citirea întregului fişier, filtrarea rândurilor care trebuie păstrate şi rescrierea fişierului cu conţinutul rămas.

Parsarea comenzii DELETE extrage numele tabelului şi clauza WHERE opţională. La fel ca la UPDATE, lipsă clauzei WHERE înseamnă ştergerea tuturor rândurilor, efectiv golin tabelul complet. Această operaţie poate fi intenţionată pentru resetarea unui tabel, dar trebuie utilizată cu grijă.

Procesul efectiv de ştergere citeşte toate rândurile din fişier şi le evaluează faţă de clauza WHERE. Rândurile care NU satisfac condiţia sunt adăugate într-un buffer de păstrare, în timp ce rândurile care satisfac condiţia sunt pur şi simplu omise. După procesarea tuturor rândurilor, fişierul este rescris cu doar rândurile păstrate.

6 Funcţionalităţi Avansate

6.1 Parsarea Condiţiilor - ConditionParser

Clasa ConditionParser implementează un mini-limbaj de expresii pentru evaluarea condiţiilor din clauzele WHERE. Această componentă transformă şiruri de caractere care reprezintă condiţii logice în structuri de date care pot fi evaluate eficient împotriva rândurilor de date.

Parserul suportă şase operatori de comparaţie fundamentali: egalitate (=), inegalitate (!=), mai mic (<), mai mare (>), mai mic sau egal (<=) şi mai mare sau egal (>=). Aceşti operatori pot compara atât valori numerice cât şi şiruri de caractere, compararea fiind realizată lexicografic pentru şiruri.

Pentru a permite condiţii mai complexe, parserul suportă şi operatori logici AND şi OR care pot combina mai multe comparaţii simple. Prioritatea operatorilor este gestionată corect, AND având prioritate mai mare decât OR, similar cu convenţiile din majoritatea limbajelor de programare. Parantezele nu sunt încă suportate în versiunea curentă, dar ar putea fi adăugate într-o extensie viitoare.

6.2 Sistemul de Logging

Sistemul include un mecanism integrat de logging care înregistrează toate operaţiile executate de utilizatori. Fiecare comandă executată este înregistrată într-un fişier de log central împreună cu timestamp-ul, numele utilizatorului şi baza de date curentă.

Aceste log-uri servesc mai multor scopuri importante. În primul rând, oferă un audit trail complet al tuturor modificărilor aduse datelor, permiţând reconstituirea istoricului operaţiilor în caz de probleme. În al doilea rând, pot fi utilizate pentru analiza paternurilor de utilizare şi identificarea bottleneck-urilor de performanţă. În al treilea rând, pot ajuta

la debugging prin oferirea unei viziuni complete asupra secvenţei de evenimente care a dus la o anumită stare a sistemului.

Formatul de log este simplu dar complet, fiecare intrare conţinând toate informaţiile necesare pentru a înţelege contextul operaţiei. Timestamp-ul este în format ISO 8601 pentru consistență şi sortabilitate uşoară. Comanda este stocată integral, permiţând reexecutarea exactă a operaţiei dacă este necesar pentru recovery sau testare.

6.3 Exportul Datelor - Saver

Modulul Saver oferă funcţionalităţi de export pentru diverse tipuri de informaţii din sistem. Utilizatorii pot exporta configurația curentă a sesiunii, inclusiv baza de date selectată și alte setări. Pot exporta log-urile pentru analiză externă sau arhivare.

Funcția de export este implementată cu grijă pentru a asigura că datele exportate sunt formatate corect și pot fi citite fără ambiguități. Caractere speciale care ar putea crea confuzie în parsarea fişierelor (precum virgule sau ghilimele în interiorul valorilor) sunt escapate corespunzător, asigurând integritatea datelor în procesul de export-import.

7 Concluzii şi Direcţii Viitoare

7.1 Realizări

Proiectul a realizat cu succes implementarea unui sistem funcțional de gestiune a bazelor de date care demonstrează înțelegerea profundă a conceptelor fundamentale din sisteme de operare și programare concurentă. Arhitectura client-server este robustă și bine structurată, iar mecanismele de sincronizare asigură corectitudinea operațiilor în mediul multithreaded.

Sistemul oferă o interfață intuitivă similară cu sistemele SQL comerciale, făcându-l ușor de învățat și utilizat. Funcționalitățile implementate acoperă majoritatea scenariilor practice de utilizare, de la operații simple de CRUD până la interogări complexe cu filtrare și sortare.

Dezvoltarea acestui sistem a oferit oportunitatea de a aplica în practică numeroase concepte teoretice studiate în cursul de sisteme de operare. Gestionarea corectă a thread-urilor și a sincronizării s-a dovedit mai complexă decât părea inițial, necesitând atenție constantă la detalii pentru a evita race conditions și deadlock-uri.

7.2 Îmbunătățiri Posibile

Sistemul actual oferă o bază solidă care poate fi extinsă în multiple direcții. Adăugarea suportului pentru tranzacții ACID ar transforma sistemul dintr-un tool educațional într-unul utilizabil în producție. Implementarea de indexuri B-tree ar îmbunătăți dramatic performanța interogărilor pe tabele mari.

Suportul pentru tipuri suplimentare de date (FLOAT, BOOLEAN, BLOB) ar extinde versatilitatea sistemului. Implementarea de constrainte (PRIMARY KEY, FOREIGN KEY, UNIQUE) ar asigura mai bine integritatea datelor. Adăugarea suportului pentru JOIN-uri ar permite interogări complexe pe multiple tabele.

Din perspectiva performanței, adoptarea unui format de stocare binar ar reduce semnificativ spațiul ocupat și ar accelera operațiile de citire/scriere. Implementarea de caching ar reduce numărul de accesări ale discului pentru date frecvent accesate. Utilizarea de memory-mapped files ar putea îmbunătăți performanța pentru tabele foarte mari.

Bibliografie

1. Stevens, W. Richard, et al. *Unix Network Programming, Volume 1: The Sockets Networking API*. Addison-Wesley, 2003. Această carte oferă o acoperire comprehensivă a programării la nivel de socket pe sistemele Unix, incluzând detalii despre TCP/IP și managementul conexiunilor.

2. Tanenbaum, Andrew S. *Modern Operating Systems*. 4th Edition, Pearson, 2014. O resursă fundamentală pentru înțelegerea conceptelor de sisteme de operare, inclusiv procese, thread-uri și sincronizare.

3. Butenhof, David R. *Programming with POSIX Threads*. Addison-Wesley, 1997. Ghid detaliat pentru programarea cu pthread, incluzând best practices și pattern-uri comune de sincronizare.

4. Date, C.J. *An Introduction to Database Systems*. 8th Edition, Pearson, 2003. Acoperire completă a teoriei bazelor de date relaționale și a principiilor de design.

5. Stroustrup, Bjarne. *The C++ Programming Language*. 4th Edition, Addison-Wesley, 2013. Referința definitivă pentru limbajul C++, inclusiv features-urile moderne din C++11/14/17.