Analiza i razvoj raspodijeljene provjerljive baze podataka u Rust programskom jeziku

Simić, Antun

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Sveučilišni prijediplomski studij računarstva

Završni rad

ANALIZA I RAZVOJ RASPODIJELJENE PROVJERLJIVE BAZE PODATAKA U RUST PROGRAMSKOM JEZIKU

Rijeka, rujan 2024.

Antun Simić

0069093391

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Sveučilišni prijediplomski studij računarstva

Završni rad

ANALIZA I RAZVOJ RASPODIJELJENE PROVJERLJIVE BAZE PODATAKA U RUST PROGRAMSKOM JEZIKU

Mentor: prof.dr.sc. Kristijan Lenac

Rijeka, rujan 2024.

Antun Simić

0069093391

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
POVJERENSTVO ZA ZAVRŠNE ISPITE

Rijeka, 20. ožujka 2024.

Zavod: **Zavod za računarstvo**
Predmet: **Operacijski sustavi**
Grana: **2.09.06 programsko inženjerstvo**

ZADATAK ZA ZAVRŠNI RAD

Pristupnik: **Antun Simić (0069093391)**
Studij: Sveučilišni prijediplomski studij računarstva

Zadatak: **Analiza i razvoj raspodjeljene provjerljive baze podataka u Rust programskom jeziku**

Opis zadatka:

Testirati implementaciju raspodjeljene provjerljive baze podataka razvijene u Rust programskom jeziku. Ispitati performanse sustava i analizirati mogućnost optimizacije sustava raspodjelom podataka ili/i izračuna na odabrane podskupove računalnih čvorova.

Rad mora biti napisan prema Uputama za pisanje diplomskih / završnih radova koje su objavljene na mrežnim stranicama studija.

Zadatak uručen pristupniku: 20. ožujka 2024.

Mentor:

Prof. dr. sc. Kristijan Lenac

Predsjednik povjerenstva za završni ispit:

Prof. dr. sc. Miroslav Joler

IZJAVA

Izjavljujem da sam samostalno izradio završni rad prema zadatku Povjerenstva za završne ispite prijediplomskog sveučilišnog studija računarstva, znanjem stečenim tijekom studija, a pod vodstvom mentora prof. dr. sc. Kristijana Lenca te uz korištenje literature i izvora navedenih u radu.

Rijeka, rujan 2024.

Antun Simić

0069093391

ZAHVALA

Zahvaljujem se mentoru prof. dr. sc. Kristijanu Lencu na vremenu koje mi je posvetio tijekom konzultacija usmjeravajući me pri izradi ovog rada.

Zahvaljujem se i svima ostalima koji su na bilo koji način pomogli u izazovima dosadašnjeg studiranja.

Antun Simić

Sadržaj:

1. UVOD

U svakodnevnom privatnom i poslovnom svijetu kontinuirano se generiraju novi podaci. Podaci se spremaju u povezane računalne sustave s ciljem korištenja kada budu potrebni, što znači da moraju biti spremljeni na način koji ih štiti od nestajanja, oštećenja pa i neovlaštenog korištenja. Ovakvo upravljanje podacima traži od vlasnika nekakve materijalne i/ili financijske resurse i znanje kako to postići. Outsource-anjem podataka na diskove poslovnih subjekata koji nude usluge udomljavanja podataka (hosting) s jedne strane vlasnici podataka rješavaju pitanja na koji disk ih spremiti i hoće li se disk oštetiti, ali otvaraju novi problem provjere sadržaja podataka jer više nisu pod fizičkom kontrolom vlasnika. Jedno od rješenja za ovakva pitanja provjere integriteta podataka jest primjena provjerljive baze podataka koja je predmet ovoga rada. Provjerljiva baza podataka je tip baze podataka dizajniran za rješavanje problema provjere integriteta i autentičnosti podataka, omogućujući korisnicima da verificiraju da su podaci kojima pristupaju točni i nisu bili neautorizirano mijenjani, čak i kada su pohranjeni izvan njihove izravne kontrole.

Kroz rad je potrebno upoznati se s konkretnom provjerljivom bazom podataka koja je napisana u Rust programskom jeziku te je funkcionalno i performansno testirati.
U prvim poglavljima ukratko su opisane IT tehnologije koje koristi baza podataka, što uključuje programski jezik i ostale povezane pojmove. U nastavku je opisano programsko okruženje u kojem je baza nastala; prije svega za potrebe korištenja unutar okruženja kriptovalute, ali i samostalnog korištenja. Dodatno su u trećem poglavlju opisana svojstva baze podataka, osobine, arhitektura baze podataka i način korištenja. U zadnjem poglavlju je opisan jedan program koji koristi provjerljivu bazu podataka s implementiranom funkcijom replikacije i provjerom integriteta replicirane baze podataka preko root hash-a podataka baze. U istom poglavlju opisani su i performansni testovi.

2. KORIŠTENI ALATI, TEHNOLOGIJE I STRUKTURE PODATAKA

2.1 Rust

Rust je programski jezik opće namjene čiji je razvoj 2006. godine započela grupa programera Mozille koja je prihvatila podržati razvoj jezika pa je tako već 2010. objavljena prva verzija programskog jezika koji se od tada kontinuirano razvija [15]. Logotip Rusta je prikazan na slici 2.1.

Značajno razdoblje u razvoju Rusta je razdoblje između 2013. i 2019. godine, u kojem se razvojnom timu Rusta priključuju i programeri C++, Jave i Pythona koji su prenijeli svoja iskustva u razvoj Rusta [16].

Slika 2.1 Logotip Rusta

Instalacija programskog jezika Rust je opisana na Internet stranicama navedenim u literaturi pod [1]. Rezultat instalacije Rusta su dobiveni programi:

- *rustc* – kompajler za prevođenje izvornog koda u binarni kod
- *cargo – builder* = upravitelj paketa koji se koristi, između ostaloga, za kreiranje novih projekata, prevođenje izvornog koda programa i izradu završnu verzije, pokretanje izvršnog programa ... [2], [8].

Naznake sintakse Rust jezika se mogu vidjeti iz HelloWorld-a u Rustu koji je prikazan na slici 2.2.

```
fn main() {
    println!("Hello, world!");
}
```

Slika 2.2 HelloWorld u programskom jeziku Rust

Rust koristi module za organizaciju koda u logičke cjeline pa modul može sadržavati funkcije, strukture ili opet module – ugniježđeni moduli. Primjer definiranja i korištenja modula prikazan je na slici 2.3.

```rust
mod config {
    // items in modules by default have private visibility
    fn select() {
        println!("called config::select");
    }

    // use the `pub` keyword to override private visibility
    pub fn print() {
        println!("called config::print");
    }
}

fn main() {
    // public items inside module can be accessed outside the parent module
    // call public print function from display module
    config::print();
}
```

Slika 2.3 Definiranje i korištenje modula

Promatrajući aplikacije koje u razvoju koriste Rust dolazimo do odgovora na pitanja područja primjene Rusta:
Mozillin internet preglednik Firefox napisan je u osam programskih jezika, a prema zadnjim podacima iz kolovoza 2024. godine, 11,5% Firefoxa čini Rust. [17];
Najveće svjetske kompanije kao što su Facebook, Microsoft i ostale koriste Rust programski jezik u svojim proizvodima [18].

2.2 NoSQL baze podataka

Pojam NoSQL baza podataka se prvi put pojavio 1998. godine od strane Carla Strozzija, koji je svoju bazu nazvao „Strozzi NoSQL“, koja je tablice spremala kao ASCII datoteke, a nije poržavala SQL za pristup podacima. Motivacija autora za izradu ovakve baze bila je želja naći bolji način rada s bazama podataka bez složenih SQL upita. S obzirom da složeni SQL upiti predstavljaju problem u performansama rada i da to stvara dodatan trošak i potrebu za dodatnim resursima te da je povećanje količine podataka izazov za relacijske baze, pronalazak alternativa koje bi smanjile kompleksnost se činilo kao nužno rješenje [19].

Za razliku od relacijskih baza podataka koje pohranjuju podatke putem tabličnih relacija, nerelacijske baze pohranjuju i dohvaćaju podatke na drukčiji način. Umjesto tablica, NoSQL baze podataka su dokument-orijentirane, tj. rade s dokumentima umjesto sa strogo definiranim tablicama podataka. Baze usmjerene na dokumente prikupljaju podatke iz dokumenata i omogućavaju njihovo dohvaćanje u pretražujućem i organiziranom obliku. Na ovaj način nestrukturirani podaci kao što su slike, videozapisi ili novinski članci mogu biti spremljeni u

jedan dokument koji se lako može pronaći bez kategorizacije polja kao u relacijskim bazama. Ovakva organizacija podataka zahtijeva dodatne napore obrade te veći prostor za pohranu nego što je to kod visoko organiziranih SQL podataka, ali zato omogućava bržu obradu određenih zahtjeva [19].

Još jedna važna karakteristika NoSQL baza je pristup modeliranju podataka, koji se značajno razlikuje od dizajna tablica u relacijskim bazama. Moguće je početi s kodiranjem, pohranom i dohvaćanjem podataka bez prethodnog znanja kako baza sprema podatke ili kako radi. Prethodno nepoznavanje sheme je možda i jedna od najznačajnijih razlika između nerelacijskih i relacijskih baza podataka. Ovakav pristup omogućava jednostavnost dizajna, jednostavno vodoravno skaliranje te lako podnosi značajne promjene sheme [19].

Četiri su glavne vrste NoSQL baza podataka [14]:

- Ključ-vrijednost baze podataka (eng. Key-value databases);
- Grafovske baze podataka (eng. Graph databases);
- Dokumentne baze podataka (eng. Document databases);
- Stupčaste baze podataka (eng. Columnar databases).

2.2.1 Ključ – vrijednost baza podataka

Struktura ili arhitektura ključ-vrijednost baza podataka sastoji se od ključa i vrijednosti. Ključ predstavlja jedinstvenu vrijednost pomoću koje se razlikuje od drugih ključeva unutar baze i dodjeljuje mu se vrijednost; ova dva podatka sačinjavaju ključ-vrijednost baze podataka kao što je prikazano u tablici 2.1. Pretraživanje je moguće samo po ključu. Domena vrijednosti nije definirana, što znači da ona može biti zapis, slika, zvuk, dokument, web-adresa i sl. Ovo svojstvo omogućava veliku fleksibilnost prilikom spremanja različitih vrsta podataka. Ova vrsta NoSQL baza smatra se najjednostavnijom jer sadrži samo parove ključeva i vrijednosti koji zajedno sačinjavaju jedan zapis u bazi [20].

Tablica 2.1 Ključ-vrijednost parovi

KLJUČ	VRIJEDNOST
Key1	Slika.png
Key2	PODATAK
Key3	Pjesma.mp3
Key4	Zavrsni.zip

Primjeri ovakvih baza su BerkeleyDB i LevelDB [21].

2.2.2 Dokumentne baze padataka

Dokument-baze podataka konceptualno su slične ključ-vrijednost bazama i mogu se smatrati podvrstom ključ-vrijednost baza. Dokument-baza podataka je NoSQL baza podataka koja podatke pohranjuje u označene dokumente ključ-vrijednost parova. Za razliku od ključ-vrijednost baza gdje komponenta vrijednosti može sadržavati bilo koju vrstu podatka, dokument baze uvijek pohranjuju dokument u komponentu vrijednosti. Druga važna razlika je da ključ-vrijednost baze podataka ne pokušavaju razumjeti sadržaj vrijednosne komponente, dok dokument-baze to čine.

Pretraživanje podataka najčešće se radi na vrijednosti, a ne pomoću ključa kao u ključ-vrijednost bazama podataka. Iako je moguće pretraživanje po ključu u dokument-bazama, to nije čest slučaj. Pretraživanje po vrijednosti dokument-bazama daje dodatnu sofisticiranost za dohvaćanje podataka [20]. Primjer ovakve baze podataka je MongoDB.

2.3 Merkle stablo i blockchain

Hash funkcija je kriptografski algoritam koji za ulaz prima određene podatke, dok za izlaz vraća niz znakova s fiksnom dužinom koji se naziva hash podatak. Hashiranje je proces generiranja hash vrijednosti iz niza znakova koristeći matematičke funkcije. Predstavlja način osiguravanja ili zaštite integriteta podataka. Hash i hashiranje su pojmovi koji se koriste u strukturi Merkle stabla.

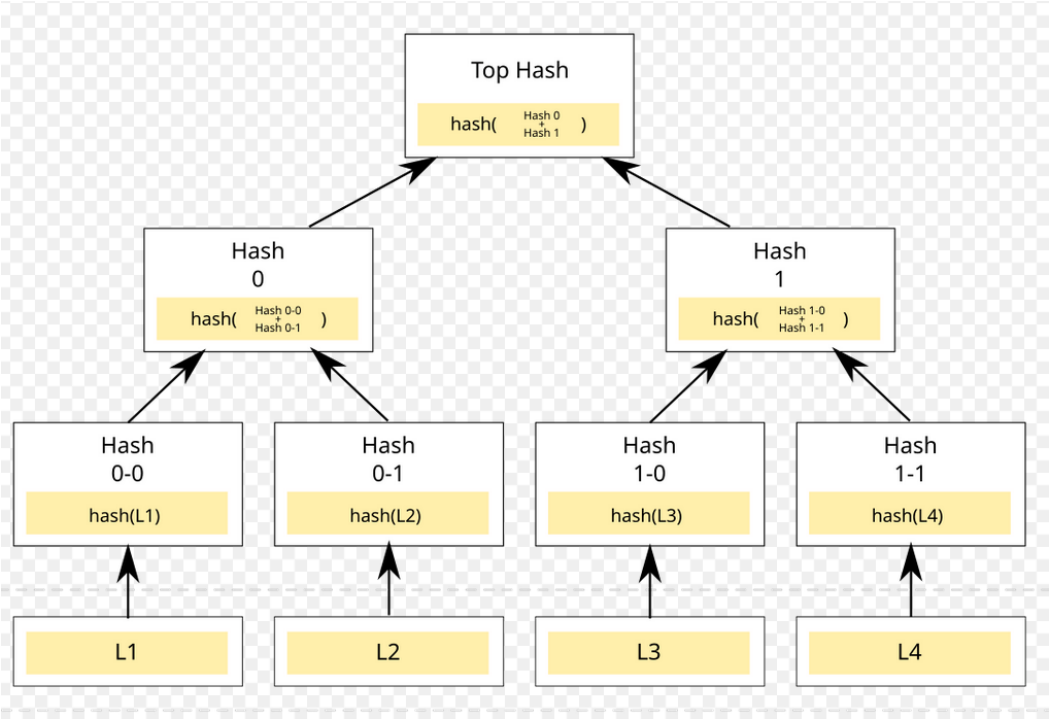
Većina hashing algoritama su javni i dostupni svakome. Svaki ulaz kroz hashing algoritam će imati jedinstveni i specifični izlaz. Čak i kada se isti ulaz ponovi više puta, hashing funkcija će uvijek dati jednaki izlaz koji je određen ulazom. Ali ako je ulaz i najmanje promijenjen, hash će biti prezentiran kao potpuno drugačiji niz znakova, što znači da mala promjena ulaza neće rezultirati maloj promjeni izlaza, već potpunom promjenom kao što se može vidjeti u tablici 2.2.

Hash funkcija je „one way function“, što znači da se može koristiti samo u jednom smjeru. Drugim riječima, hashirana poruka ne može biti „de-hashirana“.

Tablica 2.2 SHA-256 vrijednosti

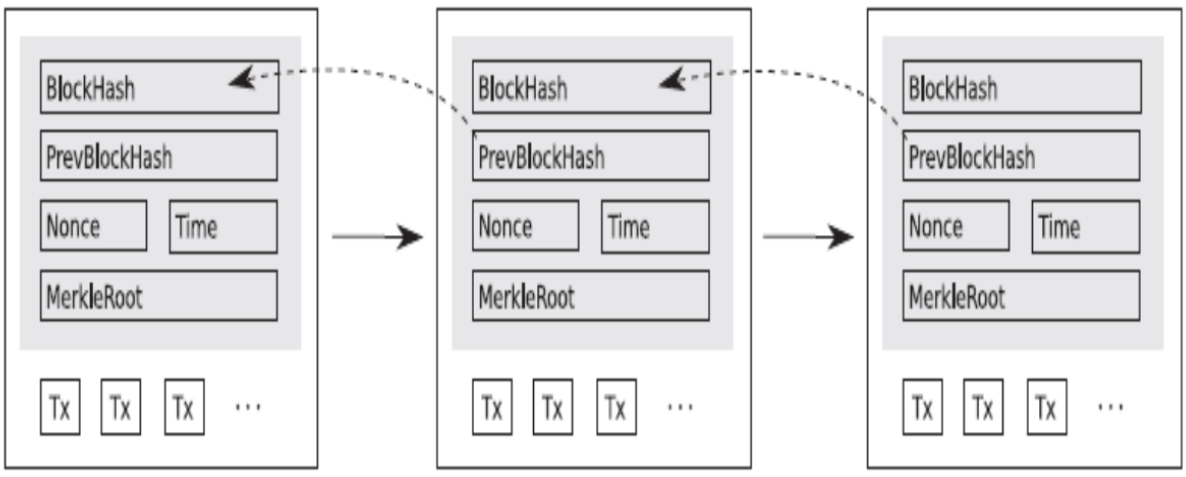
Ulaz	Izlaz
RITEH	b668cd0035f2b65c1cd42045f78af45fa6e5434645096aed59469a4d32826ed7
Riteh	3c9ec2ec1b294a673a943ac130477f62724fbc4dcfecbb8ce910c9a640dc248a

Merkle stablo je struktura podataka čiji je svaki list (podatak) povezan sa svojom hash vrijednosti te čiji svaki nadređeni čvor sadrži hash vrijednost svojih podređenih blokova - podataka kao što je prikazano na slici 2.4. Ova struktura omogućava da se na efikasan način verificira integritet cijelog skupa podataka bez potrebe za pregledom svakog pojedinog elementa.



Slika 2.4 Merkle stablo [22]

Blockchain se može direktno prevesti kao 'lanac blokova', a radi se o strukturi sastavljenoj od blokova podataka povezanih u jednosmjerni lanac gdje svaki novi blok ovisi o prethodnom bloku. Ova tehnologija se temelji na kriptografiji. Svaki blok u lancu je konačan, što znači da ima pohranjenu određenu količinu podataka ili transakcija. Sljedeći blok je povezan s prethodnim blokom i s onim koji će tek biti kreiran u budućnosti, kao što je prikazano na slici 2.5.



Slika 2.5 Blockchain

U ovakvom lancu blokova očito je da prvi blok mora imati drugačije određenu hash vrijednost prethodnog bloka od ostalih jer prethodni blok ne postoji. Prvi blok u lancu se naziva 'genesis blok' i u njemu je hash vrijednost prethodnog bloka specifična za svaki konkretni blokchain.

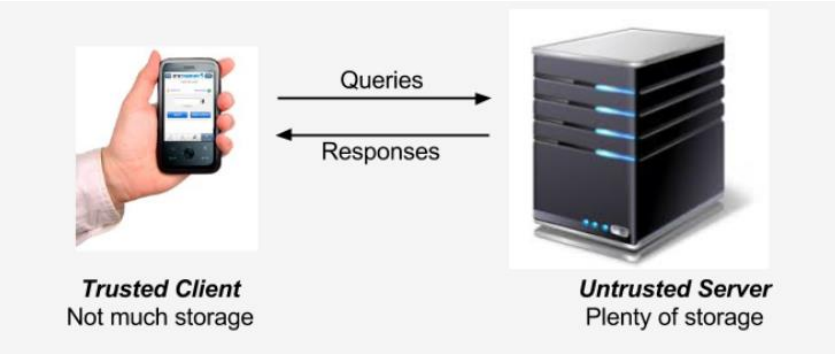
Svaka promjena stanja cijele baze se zapiše tako da se u blockchain zapiše novi root hash cijele baze. Blockchain služi kao instrument osiguravanja povjerenja da je uistinu baza ažurirana pravilno, a mehanizam kojim se to postiže oslanja se na root hash Merkle stabla. Sigurnost blockchaina se očituje u tome što ako netko želi izmijeniti podatke u jednom bloku, trebao bi izmijeniti podatke u svima, što je skoro pa nemoguće. Blok podataka se poslije trenutka zapisa više ne može mijenjati i ne može doći do narušavanja integriteta podataka u blockchainu. Ovako 'uređen' lanac blokova opisan je i definiran 2008. godine kada je jedna neidentificirana osoba (ili grupa osoba) pod pseudonimom Satoshi Nakamoto pokrenula web stranicu Bitcoin kriptovalute [24].

Ako ovako opisan blockchain, ovakvu bazu podataka stavimo u mrežno P2P okruženje u kojem se blockchain baza replicira između svih peer-ova (node-ova ili čvorova) u mreži, onda je rizik od promjene podatka u bloku još manji jer bi to značilo da bi promjenu bloka u jednom blockchainu, koja je nemoguća, trebalo napraviti na svim peerovima što je još manje vjerojatno.

Na opisanim svojstvima blockchaina se temelje sustavi kriptovaluta.

2.4 Provjerljiva struktura podataka

Vlasnici podataka često se odlučuju za spremanje privatnih ili poslovnih podataka na diskove koji nisu pod kontrolom ili nisu u vlasništvu vlasnika podataka, ali podrazumijevaju da će kod korištenja podaci imati izvornu cjelovitost ili integritet. Slika 2.6 prikazuje jednu takvu komunikaciju u kojem klijent bez diskovnog prostora koristi vanjske usluge.

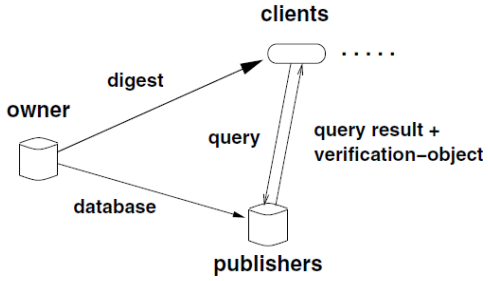


Slika 2.6 Interakcija klijenta i servera [25]

U slučaju važnih podataka, u komunikaciji klijenta i servera server mora osigurati dokaz, a klijent mora provjeriti dokaz iz kojeg se može zaključiti da su podaci koje je klijent primio:

- potpuni – kompletni, tj. nema više ili manje podataka u odnosu na podatke koje je server poslao;
- ispravni – što znači da nije bilo neautoriziranih promjena;
- ažurni ili svježi – što znači da se ne odnose na neko prethodno stanje povezano s istim upitom [13].

Uobičajene su i situacije u kojima klijent koristi podatke, ali ne s lokacije na kojoj se podaci održavaju, nego s lokacije koja je posredno došla u situaciju da vraća podatke koji izvorno ne održava kao na slici 2.7. U ovakvoj situaciji opet postoji rizik da slika ili kopija podataka nije jednaka izvornim podacima.



Slika 2.7 Korištenje slike - kopije podataka [26]

Odgovor na uvodno navedene situacije rizika korištenja podataka izvan kontrole korisnika ili klijenta jest primjena provjerljivih struktura podataka (engl. Authenticated Data Structure, skraćeno ADS) ili modela s provjerljivim strukturama podataka [13, 25].
Osim toga, na ADS se postavljaju sljedeći uvjeti kako bi se osigurala njegova funkcionalnost:
- Sigurnost sadržaja mora biti zagarantirana;
- Veličina sadržaja mora biti dovoljno mala da ne povećava znatno komunikaciju između klijenta i servera radi provjere;
- Algoritam za stvaranje mora biti efikasan kako bi se osigurala brza provjera podataka.

S obzirom na uvjete koji se postavljaju na ADS, primjena Merkle stable je prva asocijacija koja zadovoljava postavljene zahtjeve [26].

3. RASPODIJELJENA PROVJERLJIVA BAZA PODATAKA – GROVEDB

Dash je jedna od kriptovaluta koja je nastala 2014. godine na temeljima puno poznatijeg Bitcoina. Programsku podršku Dash kriptovalute razvija i održava Dash Core Group (skraćeno DCG) [3], a u nastavku će biti predstavljena programska rješenja DCG-a, namjena svakog od njih i njihovi međuodnosi.

3.1 Dash Core

Dash Core je naziv paketa (grupe) programa koji podržavaju Dash kriptovalutu, tj. transakcije Dash kriptovalute, što podrazumijeva prijenos Dash kriptovalute iz jednog u drugi novčanik. Instalacijom Dash Core paketa dobiju se tri programa [4] prema tablici 3.1 u nastavku.

Tablica 3.1 Dash Core aplikacije

Naziv programa	Kratki opis programa
dash-qt	Čvor Dash mreže; sučelje novčanika, ne servisira klijentske upite
dashd	Čvor dash mreže; nema vizualno sučelje; servisira upite od dash-cli programa
dash-cli	Omogućuje slanje naredbi (kroz terminal) koje servisira dashd program.

Podaci koje programi iz tablice 3.1 zapisuju ili čitaju pohranjuju se u blokove podataka. Blokovi su povezani u lance blokova, a lanac blokova se naziva blockchain bazom podataka.

Aplikacije iz Dash Core paketa, ali i ostale Dash aplikacije se mogu koristiti u okruženjima prema tablici 3.2.

Tablica 3.2 Dash okruženja

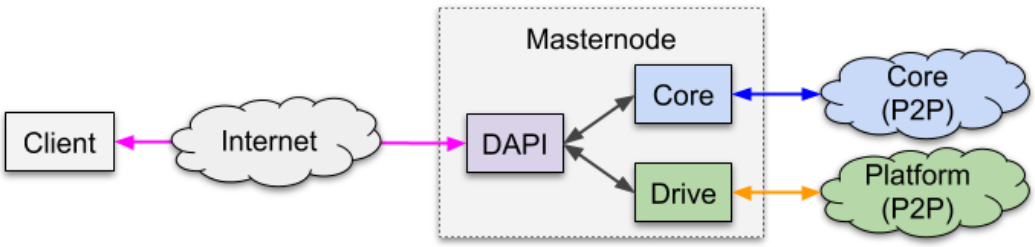
Naziv okruženja	Opis okruženja
Mainnet	Dash kriptovaluta, Internet node-ovi
Testnet	tDash kriptovaluta, Internet node-ovi
Devnet [10]	tDash kriptovaluta, intranet node-ovi
Regtest	tDash kriptovaluta

Okruženje ili Dash mreža iz tablice 3.2 se određuje preko argumenata u pozivu programa.

3.2 Dash Platform

Dash kriptovaluta nastala je grananjem, tj. odvajanjem od Bitcoina, a Dash Core aplikacije su namijenjene podršci Dash kriptovalute uz korištenje općih tehnologija blockchain baze podataka kao i funkcionalnosti P2P mreže računala u Dash mreži. Uspostavom Dash kriptovalute i Dash Core programske podrške kriptovalute ostale su nepokrivene ostale korisničke potrebe koje također traže sigurno spremanje podataka pri čemu baza podataka nije u vlasništvu jedne osobe. Ovakva ideja je motivirala Dash Core Group-u na izradu više programskih rješenja koja se zajedno zovu Dash Platform [5].

Odnos Dash Core-a i Dash Platform-e je prikazan na slici 3.1.



Slika 3.1 Dash Core i Dash Platform aplikacije

Slika 3.1 pokazuje jedan Dash čvor s programima Dash Core-a i Dash Platform-e. Dash Core čvor je povezan preko P2P mreže s ostalim Dash Core čvorovima za potrebe razmjene blokova s Dash transakcijama. Dash Platforma je također dio Dash čvora i preko P2P mreže može biti povezana s ostalim čvorovima Dash Platform mreže. Dvije glavne arhitekturalne komponente Dash Platforme su Drive i DAPI [11], a vizualna shema Dash aplikacija je prikazana na slici 3.2.

<table>
<tr><td colspan="3">DAPI</td><td rowspan="3">MASTERNODE</td></tr>
<tr><td>DASH-QT</td><td>DASHD</td><td>Drive
Platform chain
Platform state
Storage</td></tr>
<tr><td colspan="2">Blockchain baza kriptovalute</td><td>GROVEDB</td></tr>
</table>

Slika 3.2 Dash Core i Dash Platform aplikacije

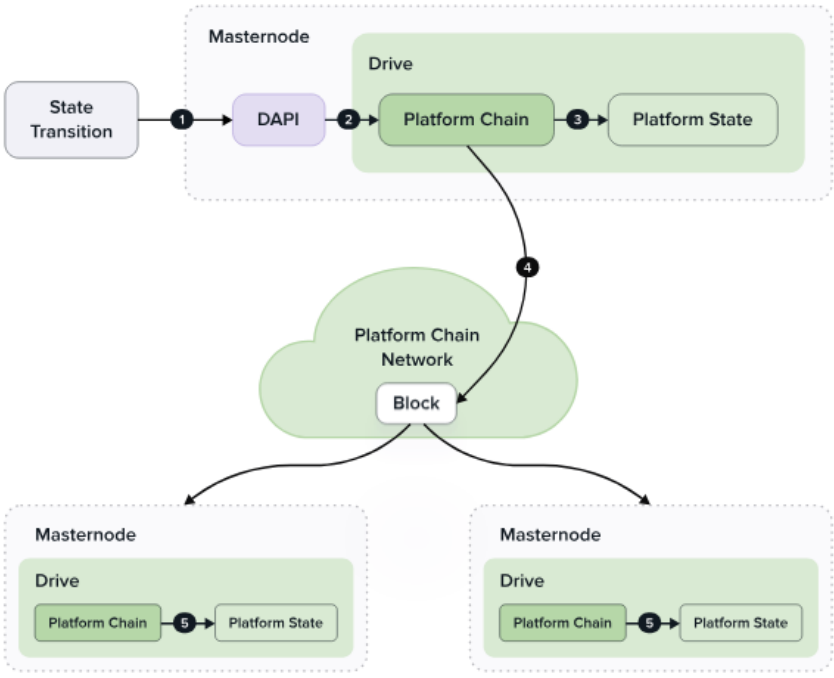
Aplikacije sa slike 3.2 navedene su u tablici 3.3 s kratkim opisom svakog od navedenih programskih rješenja.

Tablica 3.3 Dash Platform aplikacije

Komponenta	Opis
Drive	Storage komponenta, radi upis u blockchain bazu – GroveDB Platform Chain propagira blok
DAPI – decentralizirani API	Serverska strana (web servisa) koja zaprima HTTP upite prema Dash Core i Dash Platform
DAPI-client	
GroveDB	Hijerarhijska provjerljiva baza podataka

Dash Platform u uvjetima postojanja P2P mreže čvorova ima pojam stanja platforme kao skup podataka koji opisuje jedno stanje. Slika 3.3. prikazuje proces spremanja podataka u platformu kroz sljedeće aktivnosti:

1. Promjene stanja dolaze na DAPI jednog čvora
2. DAPI šalje promjene stanja na lanac platforme kako bi ih se validiralo i stavilo u blok.
3. Sve ispravne promjene stanja se primjenjuju na stanje platforme.
4. Blockchain Platform-e šalje blok svim glavnim čvorovima.
5. Glavni čvorovi ažuriraju podatke na modulu s obzirom na sve prethodno navedeno.



Slika 3.3 Proces spremanja podataka u Dash Platform-i [7].

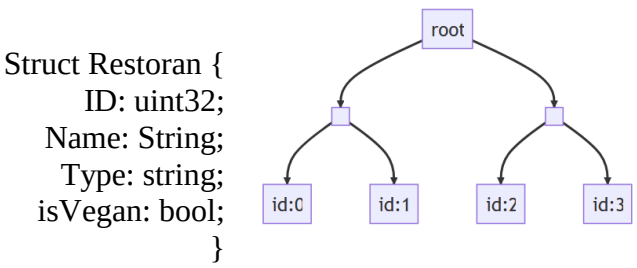
3.3 Motivacija za razvoj baze

Podaci Dash Platform-e trebaju bazu podataka, a DCG su 2023. godine objavili GroveDB kao dio Dash Platforme, ali i s mogućnosti samostalnog – odvojenog korištenja [28].

Problem provjerljive strukture podataka koja će osigurati odgovore na pitanja integriteta ili ispravnosti sadržaja distribuiranih podataka obrađuje se u [13, 25, 26, 27], a [13] i DCG navodi u razvojnoj dokumentaciji GroveDB-a, iz čega se može zaključiti da je navedeni članak bio teoretska osnova za razvoj GroveDB-a, tj. potreba za provjerljivom bazom podataka (ADS).

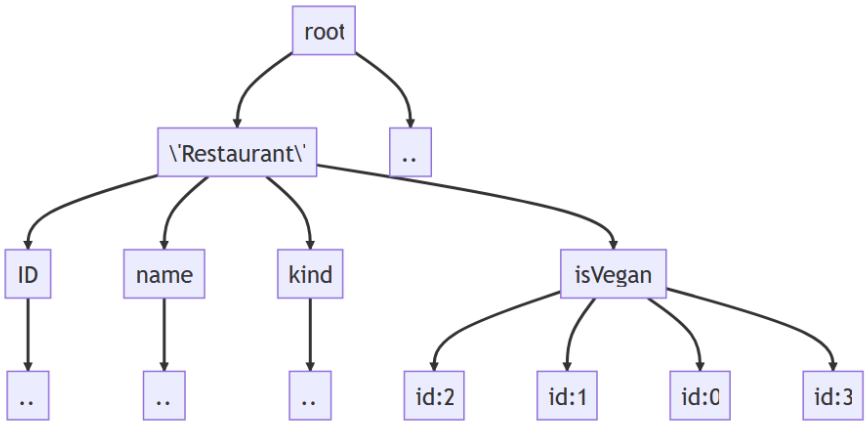
3.4 Arhitektura baze podataka (primjer)

GroveDB je razvijena na osnovu postojeće RocksDB, a RocksDB baza grananjem od LevelDB ključ-vrijednost NoSQL baze podataka. GroveDB je, za razliku od svojih prethodnika, hijerarhijska provjerljiva baza koja koristi Merkle stablo za potrebe funkcije provjerljivosti. Primjer na slici 3.4 pokazuje zapis podataka na primjeru podataka o restoranima pri čemu je restoran struktura u nastavku:



Slika 3.4 Baza s četiri podatka

Stablasti zapisi su efikasni za pretraživanje po jednom kriteriju, što bi u ovom slučaju značilo po ključu (ID) restorana, ali u realnim sustavima pretraživanje je potrebno i po ostalim atributima; u ovom slučaju, po nekom od atributa strukture Restoran. Rješenje GroveDB-a jest zadržavanje stablaste strukture s root hash vrijednosti uz primjenu podstabala za potrebe pretraživanja po ostalim atributima kao što je prikazano na slici 3.5.



Slika 3.5 Arhitektura podrške sekundarnih indeksa

3.5 Razvoj programskog rješenja za upotrebu GroveDB-a

U nastavku je objašnjeno korištenje nekoliko funkcija GroveDB baze.

3.5.1 Otvaranje ili kreiranje baze

GroveDB se smješta u nekom od direktorija na datotečnom sustavu računala i putanja do direktorija je jedina varijabla naredbe za kreiranje ili otvaranje baze. Naredba je prikazana na slici 3.6, a primjer korištenja naredbe je u prilogu 9.1., ali i na GitHub lokaciji https://github.com/antunsimic/zavrsni/blob/main/projekt/repl/src/bin/ex01.rs.

```
// Specify a path and open GroveDB at the path as db
let path = String::from("../tutorial-storage");
let db = GroveDb::open(path).unwrap();
```

Slika 3.6 Otvaranje baze

3.5.2 Dodavanje podataka u bazu

Slika 3.5 pokazuje više stabala unutar stabla GroveDB baze. Za dodavanje praznih stabala i ključ-vrijednosti u bazu GroveDB koristi se insert() funkcija na način kao na slici 3.7.

```
db.insert(
    root_path,
    b"tree1",
    Element::empty_tree(),
    None,
    None,
    grove_version,
)
.unwrap()
.expect("successful tree insert");
```

```
db.insert(
    &[b"tree1"],
    b"hello",
    Element::new_item(b"world".to_vec()),
    None,
    None,
    grove_version,
)
.unwrap()
.expect("successful key1 insert");
```

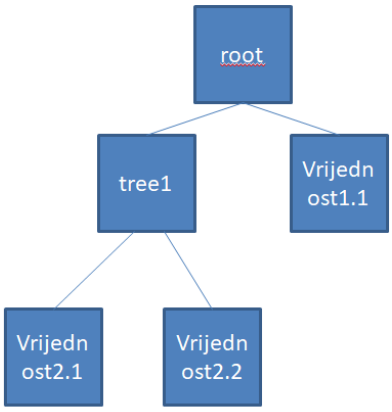
Slika 3.7 Argumenti insert funkcije

Dio argumenata funkcije insert() opisan je u tablici 3.1 u nastavku.

Tablica 3.1 Argumenti insert funkcije

Predmet dodavanja	Stablo	Par ključ-vrijednost
1. Argument insert funkcije	Stablo-odredište	Stablo-odredište
2. Argument insert funkcije	Naziv novog podstabla	Ključ
3. Argument insert funkcije	Element::empty_tree()	Vrijednost

Oba primjera dodavanja su demonstrirana u prilogu 8.1 na primjeru baze podataka na slici 3.8 u nastavku.



Slika 3.8 Korištena baza uz insert funkciju

3.5.3 Čitanje podataka iz baze – get funkcija

Jedan od načina čitanja podataka iz GroveDB je pomoću get() funkcije. Get funkcija koristi se na način prikazan na slici 3.9.

```
let elem = db
    .get(&[b"tree1"], b"hello", None, grove_version)
    .unwrap()
    .expect("successful get");
assert_eq!(elem, Element::new_item(b"WORLD".to_vec()));
```

Slika 3.9 Argumenti get funkcije

Važnija svojstva get funkcije:

- argumenti get funkcije su stablo i jedan ključ para ključ-vrijednost, stoga get funkcija vraća jednu vrijednost;
- ne podržava generiranje dokaza povezanog s vrijednosti koju vraća.

Primjer korištenja get funkcije je u prilogu 8.1.

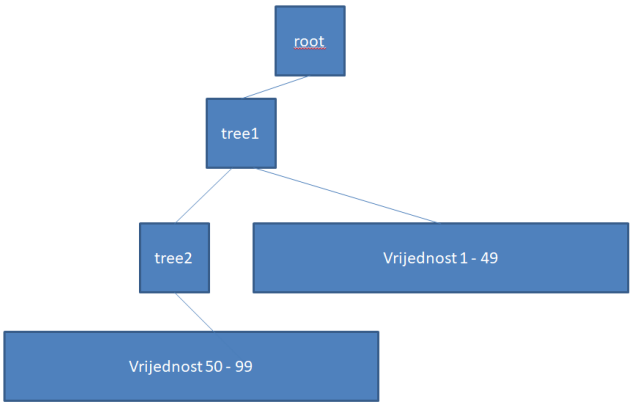
3.5.4 Čitanje podataka iz baze s kriptografskim dokazom – Query

Get funkcija iz prethodnog poglavlja čita podatke iz zadanog stabla baze podataka i za jedan zadani ključ vraća vrijednost ako zadani ključ postoji u stablu. Primjenom funkcija iz GroveDB Query modula upit se postavlja također na stablo, ali vraća sve podatke iz stabla koji zadovoljavaju ulazne uvjete. Priprema i postavljanje upita su opisani u nastavku u tablici 3.2.

Tablica 3.2 Priprema i postavljanje upita korištenjem Query modula

Sintaksa	Kratki opis
Query::new()	Inicijalizacija upita
query.insert_range	Definiranje upita (slično ključnoj riječi *where* iz SQL-a)
PathQuery::new	Povezivanje stabla i upita iz drugog koraka
db.query_item_value	Izvršavanje upita
db.prove_query	Generiranja dokaza uz rezultat upita

Korištenje opisanog Query modula demonstrirano je u prilogu 9.2, ali i na GitHub lokaciji https://github.com/antunsimic/zavrsni/blob/main/projekt/repl/src/bin/ex02.rs. Primjer koristi bazu prikazanu na slici 3.10.



Slika 3.10 Baza za potrebe demonstracije Query modula

Izvođenjem programa na bazi podataka sa slike 3.10 dobiju se rezultati navedeni u tablici 3.3.

Tablica 3.3 Rezultat izvođenja programa

Stablo	Definirani raspon u insert_range	Rezulat pretraživanja
TREE1	45 – 55	45, 46, 47, 48, 49
TREE2	45 – 55	50, 51, 52, 53, 54

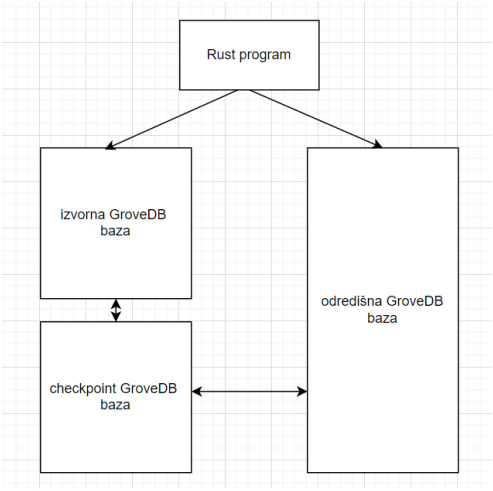
U tablici 3.2 u zadnjem retku kao opcija je navedena mogućnost generiranja dokaza uz postavljeni upit. Navedena mogućnost je također demonstirana u prilogu 9.3.

3.5.5 GroveDB replikacija

Replikacija je svojstvo sustava za upravljanje bazom podataka koje omogućuje da sadržaji (baze podataka i njihove tablice) jednog ili više poslužitelja budu dodatno dostupni na nekom drugom poslužitelju ili više njih (koje nazivamo replikama). Replikacija se koristi za različite potrebe, a važnije su:

- skalabilnost: veći broj poslužitelja omogućuje da se čitanja iz baze podataka raspodijele na veći broj poslužitelja, što smanjuje opterećenje primarnog poslužitelja i osobito je primjenjivo za sustave koji imaju velik broj čitanja u odnosu na broj pisanja,
- pomoć u izradi rezervnih kopija podataka: rezervne kopije je lakše izraditi kad se podaci ne mijenjaju pa je tipičan postupak izrada replikacije, odspajanje replike od primarnog poslužitelja (čime ona prestaje izvoditi promjene na podacima) i izrada pohrane podataka na replici [29].

Funkcionalnost replikacije GroveDB-a nije opisana na web stranicama dokumentacije ni tutorijala baze, za sada nije objavljen nijedan blog na tu temu, ali na u izvornog koda baze postoji source datoteka iz koje se može vidjeti kako se funkcionalnost koristi. Slika 3.11 pokazuje važne komponente koje sudjeluju u repliciranju podataka.



Slika 3.11 GroveDB replikacija

Priprema i izvedba replikacije uključuje sljedeće programske aktivnosti:

- Na GroveDB bazi koja će biti izvor repliciranja potrebno je primjeniti ‘create_checkpoint’ funkciju uz atribut putanje na datotečnom sustavu na kojoj će se za potrebe replikacije kreirati dodatna ‘checkpoint’ GroveDB baza – slika 3.12;

```
db_source.create_checkpoint(&path_checkpoint).expect("cannot create checkpoint");
let db_checkpoint_0 = GroveDb::open(path_checkpoint).expect("cannot open groveDB from checkpoint");
```

Slika 3.12 Kreiranje checkpoint baze

- Postavljanje MultiStateSyncInfo strukture koja upravlja replikacijom na zadane vrijednosti – slika 3.13;
- Oznaka početka transakcije na odredišnoj bazi – slika 3.13;
- Pokretanje replikacije između ‘checkpoint’ baze i odredišne baze – slika 3.13;

- Oznaka kraja transakcije na odredišnoj bazi sa spremanjem svih promjena – slika 3.13.

```
println!("\n######### db_checkpoint_0 -> db_destination state sync");
let state_info = MultiStateSyncInfo::default();
let tx = db_destination.start_transaction();
sync_db_demo(&db_checkpoint_0, &db_destination, state_info, &tx, &grove_version).unwrap();
db_destination.commit_transaction(tx).unwrap().expect("expected to commit transaction");
```

Slika 3.13 MultiStateSyncInfo, transakcija, replikacija

Korištenje opisanih funkcija replikacije s provjerama i usporedbom root hash vrijednosti je demonstrirano u sljedećem poglavlju.

4. IMPLEMENTACIJA I TESTIRANJE BAZE

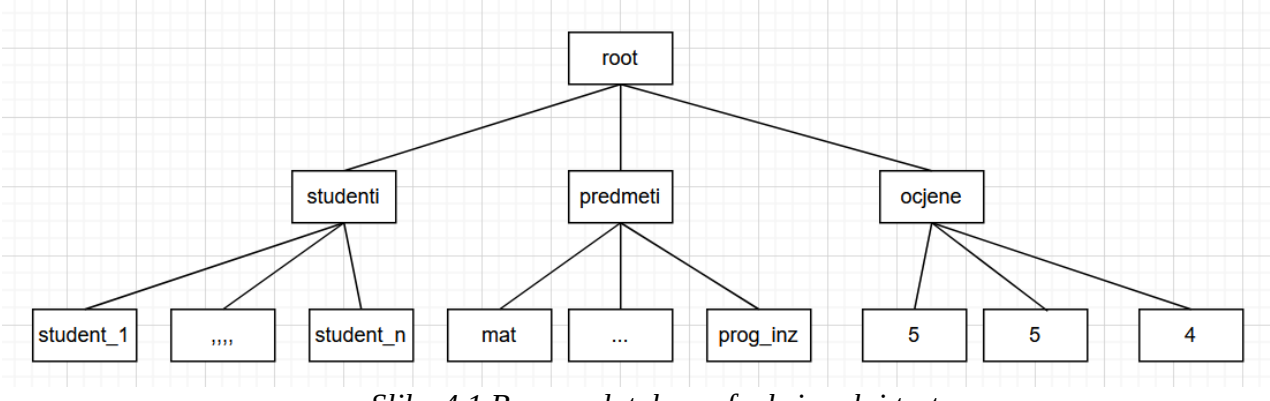
Primjenom prethodno opisanih tehnologija, u sklopu ovoga rada implementirane su dvije baze i više programa za potrebe funkcionalnog i performansnog testiranja.

4.1 Funkcionalno testiranje

Funkcionalnim testiranjem cilj je pokazati da baza podataka radi (ili ustvrditi da ne radi) u skladu s očekivanjima. Kroz provedeni test su ispitane funkcionalnosti kreiranja baze, izgradnje stablaste – hijerarhijske strukture baze, dodavanja podataka, čitanja podataka te replikacija baze podataka.

4.1.1 Priprema testa

Za funkcionalno testiranje kreirana je korisnička GroveDB baza podataka jednog fakulteta koja sadrži podatke o studentima, predmetima i položenim ispitima i replika baze na drugu lokaciju datotečnog sustava koja predstavlja bilo koju instituciju koja treba imati navedene podatke. Stablo korištene baze podataka je prikazano na slici 4.1.



Slika 4.1 Baza podataka za funkcionalni test

Primjeri ključ-vrijednost podataka navedeni su u tablici 4.1.

Tablica 4.1 Primjeri vrijednosti u bazi

Stablo	Ključ	Vrijednost	Napomena
Student	1	Ivo Ivić	
Student	2	Pero Perić	
Predmet	1	Matematika	
Predmet	2	Algoritmi	
Ocjene	1_2	5	Ključ je složen od ključa studenta i ključa predmeta
Ocjene	2_2	4	

Programsko rješenje koje kreira baza podataka sa slike 4.1 radi replikaciju baze s provjerom ispravnosti replikacije, a napravljeno je uz primjenu pseudokoda u nastavku:

Početak

kreiranje baze 'Riteh-računarstvo-predmeti, ocjene' – izvorna baza;

Kreiranje baze 'replika';

Populacija-punjenje baze 'Riteh-računarstvo-predmeti' – prema slici 4.1

Ispis root hash podatka obiju baza

Obavljanje replikacije - od izvora prema replici;

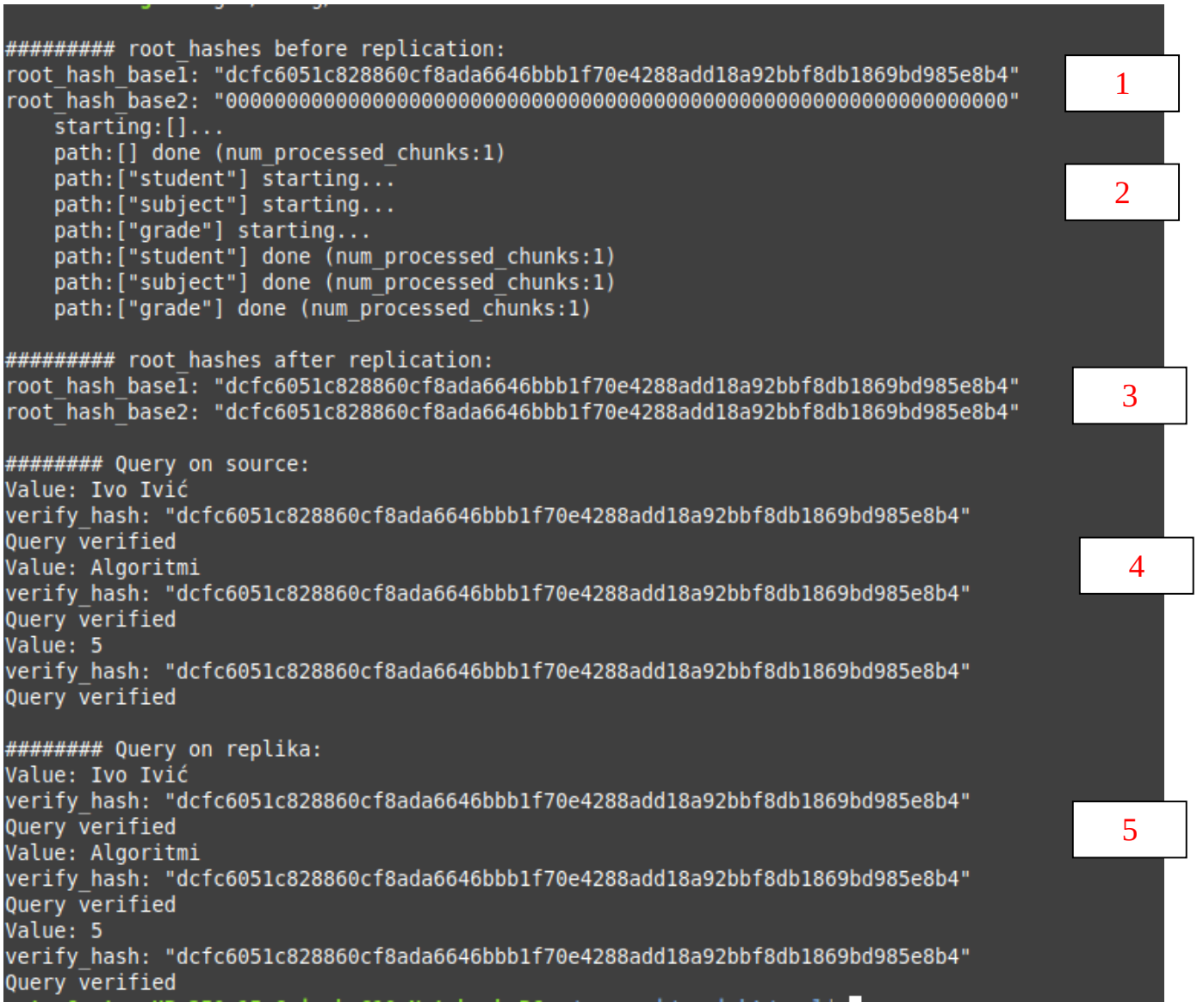
Ispis root hash podatka obiju baza;

Ispis vrijednosti za iste odabrane ključeve u objema bazama

Kraj

4.1.2 Provedba i rezultati testa

Program prema pseudokodu iz prethodnog poglavlja s Github lokacije https://github.com/antunsimic/zavrsni/blob/main/projekt/repl/src/main.rs poslije builda tijekom izvođenja ispisuje poruke prikazane na slici 4.2. Program se može naći i u prilogu 9.3.



Slika 4.2 Ispis poruka tijekom izvođenja programa

Kod izvođenja programa se može uočiti:

1. Prije ispisa teksta ‘before replication’ program je kreirao izvornu bazu (na ispisu ‘base1’) i odredišnu bazu podataka (na ispisu ‘base2’), zatim ‘base1’ napunio podacima iz tablice 4.1. Sadržaji baza u ovoj fazi su različiti, što pokazuju i prve vrijednost root hash-eva (oznaka 1 na slici 4.2);

2. Iza ispisa prvih root hash-eva slijedi replikacija baze ‘base1’ u bazu ‘base2’ kroz komade (‘chunks’) s porukama između ispisa root hash-eva (oznaka 2 na slici 4.2);
3. Poslije replikacije ponovno su ispisani root hash-evi baza podataka, koji su sada jednaki, što znači da je odredišna baza ‘base2’ sada sadržajno jednaka izvornoj bazi ‘base1’ (oznaka 3 na slici 4.2);
4. Osim preko root hash-eva koji se odnose na cijelu bazu, provjera replikacije je napravljena provjerom vrijednosti uz zadane ključeve u obje baze podataka (oznake 4 i 5 na slici 4.2), pa su ispisane vrijednosti:
    - Uz ključ = 1 iz stabla ‘Studenti’: Ivo Ivić;
    - Uz ključ = 2 iz stabla ‘Predmeti’: Algoritmi;
    - Uz ključ = 1_2 iz stabla ‘Ocjene’: 5,

    Ovime smo se uvjerili da su baze podataka uistinu jednake sadržajem.
5. Uz vrijednosti koje se dobiju iz upita na bazu, koje su ispisane uz oznake 4. i 5. na slici 4.2, baza vraća ispisane ‘proof’ vrijednosti koje su jednake hash root-u baze kao što je navedeno u [28] uz ‘proof’ funkciju.

4.2 Performansno testiranje

Osim funkcionalnosti softvera, važno je znati kojom brzinom navedene funkcionalnosti rade, tj. koliko resursa trebaju ili troše, što je za GroveDB bazu podataka opisano u nastavku.

4.2.1 Priprema testa

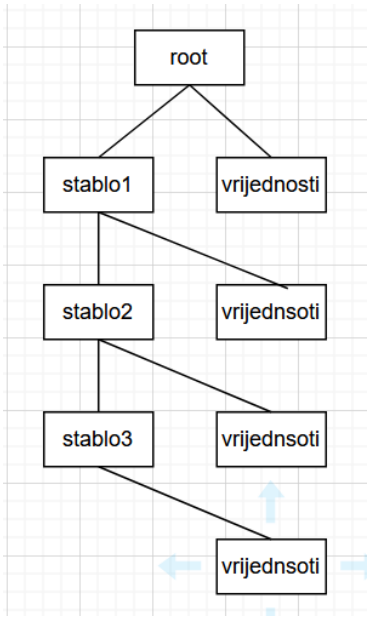
U test perfomansi GroveDB baze uključena su sljedeća mjerenja:

- Cijena – veličina baze;
- Brzina dodavanja podataka – vrijeme potrebno za dodavanje ključ-vrijednost podatka;
- Brzina dohvaćanja podataka.

Za navedena mjerenja potrebna je baza u koju će se podaci dodavati, podaci koji će se u bazu dodavati i programi koji će izvršavati potrebne radnje nad bazom podataka i povezana mjerenja.

Podatak koji je u bazu upisivan je konstanta, string duljine 500 znakova, a baza podataka u koji je upisivan jest baza s tri podstabla u hijerarhijskoj strukturi prema slici 4.3. String se dodaje ravnomjerno po stablima zajedno s ključevima također prema slici 4.3.

Ključevi po stablima	
Stablo	Ključevi
Root	[1-500], [500-1000] ...
Stablo1	[1-500], [500-1000] ...
Stablo2	[1-500], [500-1000] ...
Stablo3	[1-500], [500-1000] ...



Slika 4.3 Baza podataka za test performansi

Za testiranje su pripremljena tri programa s odvojenim funkcionalnostima:

- Kreiranje baze;
- Dodavanje podataka;
- Postavljanje upita.

Pseudokod navedenih programa kojima se testiraju performanse je u nastavku:

Kreiranje baze: početak

Otvaranje baze podataka

Dodavanje stablo1 u root

Dodavanje stablo2 u stablo1

Dodavanje stablo3 u stablo2

Kraj

Dodavanje podataka: početak

Inicijalizacija vrijednosti

Inicijalizacija ključa

Otvaranje baze podataka

Početak mjerenja vremena

Ponavljanje 500 puta:

Inkrementiracija ključa

Dodavanje para ključ-vrijednost u root

Dodavanje para ključ-vrijednost u stablo1

Dodavanje para ključ-vrijednost u stablo2

Dodavanje para ključ-vrijednost u stablo3

Ispis proteklog vremena

Kraj

Postavljanje upita: početak

Otvaranje baze podataka

Inicijalizacija ključa

Početak mjerenja vremena

Ponavljanje 500 puta:

Inkrementacija ključa

Upit na root stablo uz trenutni ključ

Upit na stablo1 uz trenutni ključ

Upit na stablo2 uz trenutni ključ

Upit na stablo3 uz trenutni ključ

Ispis vremena

Kraj

4.2.2 Provedba i rezultati testa

Programi potrebni za ovo testiranje prema pseudokodu iz prethodnog poglavlja su napisani u Rust programskom jeziku, nalaze se u p-open.rs, p-insert.rs i p-query.rs datotekama na Github lokaciji https://github.com/antunsimic/zavrsni/tree/main/projekt/repl/src/bin. Programi osim obavljanja navedenih koraka mjere i vrijeme potrebno za testiranje uz korištenje Rust Instant modula i povezanih funkcija prema slici 4.4 iz programa za kreiranje baze.

```
105         // Setup
106         let start = Instant::now();
107         insert_empty_tree_db(&db, &[], b"tree1", &grove_version);
108         insert_empty_tree_db(&db, &[b"tree1"], b"tree2", &grove_version);
109         insert_empty_tree_db(&db, &[b"tree1", b"tree2"], b"tree3", &grove_version);
110     //    let constant = "a".repeat(500);
111         println!("Setup time: {:?}", start.elapsed());
112     */
```

Slika 4.4 Modul za mjerenje vremena

GroveDB baza se može koristiti unutar Dash Platforme ili samostalno, a performansni testovi su provedeni na samostalnoj GroveDB bazi u lokalnom (Linux) okruženju, što znači da u rezultatima testiranja nema nikakvih utjecaja mreže.

Uz opisane podatke koji će se dodavati u bazu, strukturu baze, programe i specifikaciju okruženja, testiranje performansi je provedeno kroz aktivnosti iz tablice 4.2, a rezultati su zabilježeni u tablici 4.3 i dodatno analizirani u tablici 4.4:

Tablica 4.2 Aktivnosti testiranja

Redni broj	Program	Ishod	Ključevi iz skupa	Relevantni redak tablice 4.3	Relevantni stupci tablice 4.3.
1.	p-open	Kreirana baza	N/A	1	Zauzeta memorija
2.	p-insert	Dodano 2000 ključ-vrijednosti	1-500	2	Zauzeta memorija, vrijeme dodavanja
3.	p-query	Pročitano 2000 ključ-vrijednosti	1-500	2	Upita u sekundi
4.	p-insert	Dodano 2000 ključ-vrijednosti	501-1000	3	Zauzeta memorija, vrijeme dodavanja
5.	p-query	Pročitano 2000 ključ-vrijednosti	501-1000	3	Upita u sekundi
7.	p-insert	Dodano 2000 ključ-vrijednosti	1001-1500	4	Zauzeta memorija, vrijeme dodavanja
8.	p-query	Pročitano 2000 ključ-vrijednosti	1001-1500	4	Upita u sekundi

Tablica 4.3 Rezultati testiranja

Redni broj	Predmet promatranja	Zauzeta memorija	Vrijeme dodavanja	Upita u sekundi
1.	Prazna baza	80 kB		
2.	Prvih 2000 ključ-vrijednosti	26 MB	1,98s	11049
3.	Sljedećih 2000 ključ-vrijednosti	33 MB	2,4s	10869
4.	Sljedećih 2000 ključ-vrijednosti	37 MB	2,62s	10989

Tablica 4.4 Obrada rezultata testiranja

	Prosjek	Varijanca
Dodavanje	869.93 par/s	16170.83
Postavljanje upita	10969 upit/s	8400

U tablici 4.4 „par/s“ označava koliko ključ-vrijednost parova je prosječno dodano po sekundi u bazu podataka, a slično tome „upit/s“ označava koliko je upita izvršeno po sekundi.
Prema dobivenim rezultatima performansnog testiranja do 6000 ključ-vrijednosti u bazi, može se uočiti:

- Vrijeme potrebno za dodavanje sadržaja u bazu blago raste s porastom veličine baze;
- Vrijeme potrebno za izvođenje upita je neovisno o veličini baze;
- Dodavanje u bazu podataka znatno je sporije od postavljanja upita
- Varijanca dodavanja je prilično visoka, što ukazuje na utjecaj veličine baze na performanse dodavanja novih podataka
- Unatoč visokoj varijanci postavljanja upita, GroveDB pokazuje stabilne performanse u ovoj kategoriji neovisno o veličini baze podataka, s obzirom da su razlike između postignutog broja upita u testovima vrlo male u odnosu na apsolutni broj ostvarenih upita u svakom od testova
- S obzirom da 2000 ključ–vrijednosti (uz veličinu vrijednosti od 500 B) zauzima izvan baze prostor od otprilike 1 MB, baza podataka zauzima mnogo više memorije od veličine umetnutih podataka, ali povećanjem broja ključ-vrijednosti u bazi razlika između povećanja baze podataka i stvarne veličine umetnutih podataka se smanjuje.

5. ZAKLJUČAK

DCG razvija programsku podršku Dash kriptovalute kroz dvije osnovne grupe softvera: Dash Core i Dash Platform. Na osnovama postojeće RocksDB NoSQL key-value baze podataka i uz korištenje Merkle stabla razvijen je GroveDB za potrebe korištenja unutar Dash Platforme.

Izvorni kod GroveDB-a DCG objavljuje na Githubu, a na svom blogu objavljuje novosti i način korištenja baze. Prema tagovima prve verzije izvorni kod je iz 2022. godine, a prema tekstovima na blogu produkcijska verzija objavljena je 2023. godine, što je, čini se, povezano sa širinom podrške korištenja ove baze koja je ograničena isključivo na DCG.

GroveDB je prva produkcijska baza s implementiranim osobinama provjerljivosti i sekundarnih indeksa napisana u Rust programskom jeziku. Svojstvo provjerljivosti rješava izazove outsource-anja podataka, a brzi sekundarni indeksi temeljeni na arhitekturi zapisa podataka u bazi povećavaju područje primjene ovakve baze podataka. GroveDB ima implementirane mehanizme replikacije baze na proizvoljan broj lokacija, kao što je demonstrirano u prethodnom poglavlju, što omogućuje distribuciju podataka i povećava područje primjene.
Iz provedenih performansnih testova može se uočiti stabilna brzina postavljanja upita od približno 10969 po sekundi, a brzina upisivanja ključ-vrijednosti u bazu od prosječno 870 ključ-vrijednosti u sekundi za veličinu podatka vrijednost 500 B smanjuje se kako baza podataka raste.

6. LITERATURA

[1] the Rust team: „Install Rust – Rust Programming Language“, s Interneta, https://www.rust-lang.org/tools/install, 10. travnja 2024.

[2] Tutorialspoint: „Rust Tutorial“, s Interneta, https://www.tutorialspoint.com/rust/rust_package_manager.htm, 10. travnja 2024.

[3] Dash: „Dash Core Group Information“, s Interneta, https://www.dash.org/dcg, 24. svibnja 2024.

[4] Dash Core Group, Inc.: „Introduction – Dash Core latest documentation“, s Interneta, https://docs.dash.org/projects/core/en/stable/docs/examples/introduction.html, 25. svibnja 2024

[5] Dash – Digital Cash: "Dash is Becoming a Decentralized Cloud | Dash Platform #1", s Interneta, https://www.youtube.com/watch?v=9WqUMrIN58Q&list=PLiFMZOlhgsYKoWoCA-KdkTPFITWxhjDxW; 1. lipnja 2024.

[6] Dash: „Decentralized API (DAPI)“, s Interneta, https://dashplatform.readme.io/v0.19.0/docs/explanation-dapi, 1. lipnja 2024.

[7] Dash Core Group, Inc.: „Drive – Dash Platform latest documentation“, s Interneta, https://docs.dash.org/projects/platform/en/stable/docs/explanations/drive.html; 1.lipnja 2024.

[8] Monday Morning Haskell: „Cargo Package Management“, s Interneta, https://mmhaskell.com/rust/cargo, 20. lipnja 2024.

[9] DeLucia, P.: „What is GroveDB“, s Interneta, https://www.dash.org/blog/what-is-grovedb, 25. lipnja 2024.

[10] Thephez: „Dash Devnets“, s Interneta, https://www.dash.org/blog/dash-devnets/, 12. lipnja 2024.

[11] Dash Core Group, Inc.: „What is Dash Platform - Dash Platform latest documentation“, s Interneta, https://docs.dash.org/projects/platform/en/latest/docs/intro/what-is-dash-platform.html; 12. lipnja 2024.

[12] Dash Core Group, Inc.: „GroveDB: Documentation“, s Interneta, https://www.grovedb.org/documentation.html; 15. lipnja 2024.

[13] Etemad, M.; Küpçü, A,: „Database Outsourcing with Hierarchical Authenticated Data Structures“, Cryptology {ePrint} Archive, Paper 2015/351, Istanbul, Turkey, 2015.

[14] Stojanović, A.: “Osvrt na NoSQL baze podataka – Četiri osnovne tehnologije“, Polytechnic & Design, Vol. 4, No. 1, pp. 44-53, Zagreb, 2016.

[15] Answerjet: „Rust Programming Language: history, features, applications, why learn?“, s Interneta, https://www.answersjet.com/2021/06/rust-programming-language-history-features-applications-why-should-learn-rust.html, 12. lipnja 2024.

[16] Klabnik, S.: „The History of Rust“, s Interneta, https://www.youtube.com/watch?v=79PSagCD_AY, 12. lipnja 2024.

[17] mozilla/gecko-dev: „How much Rust in Firefox“, s Interneta, https://4e6.github.io/firefox-lang-stats/, 12. kolovoza 2024.

[18] Dreimanis, G.: „9 Companies That Use Rust in Production“, s Interneta, https://serokell.io/blog/rust-companies, 13. kolovoza 2024.

[19] Fowler, A.: „NoSQL For Dummies“, John Wiley & Sons, Inc., Hoboken, New Jersey, 2015.

[20] Coronel, C.; Morris, S.: „Database Systems: Design, Implementation, and Management, 12th Edition“, Cengage Learning, Boston, MA, 2017.

[21] The Upwork Team: „NoSQL vs. SQL: Important Differences & Which One Is Best for Your Project“, s Interneta, https://www.upwork.com/hiring/data/sql-vs-nosql-databases-whats-the-difference, 31. srpnja 2024.

[22] Wikipedijini suradnici: „Merkle tree“, s Interneta, https://en.wikipedia.org/wiki/Merkle_tree , 14. kolovoza 2024.

[23] Mohanta, B. K.; Panda, S. S.; Jena, D.: “An Overview of Smart Contract and Use Cases in Blockchain Technology,” 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT), Bengaluru, India, pp. 1-4, 2018.

[24] Nakamoto, S.: "Bitcoin: A Peer-to-Peer Electronic Cash System", s Interneta, https://bitcoin.org/bitcoin.pdf, 30. srpnja 2024.

[25] Hicks, M.: „Authenticated Data Structures (Generically)“, s Interneta, http://www.pl-enthusiast.net/2014/06/11/authenticated-data-structures-generically/, 20. kolovoza 2024.

[26] Martel, C.; Nuckolls, G.; Devanbu, P.: “A General Model for Authenticated Data Structures”, Algorithmica, Vol. 39, pp. 21-41, 2004.

[27] Brun, M.; Traytel, D.: “Generic Authenticated Data Structures, Formally”, Leibniz International Proceedings in Informatics (LIPIcs), Vol. 141, pp. 10:1-10:18, 2019.

[28] Dash Core Group, Inc.: „GroveDB - Cryptographically Secure Database with Advanced Functionalities“, s Interneta, https://www.grovedb.org/, 7. lipnja 2024.

[29] Miletić, V: „Replikacija sadržaja poslužitelja sustava za upravljanje bazom podataka MariaDB“, s Interneta, https://group.miletic.net/hr/nastava/materijali/mariadb-replikacija/#, 21. kolovoza 2024.

7. POPIS OZNAKA I KRATICA

P2P - peer to peer struktura je struktura računala u kojoj jedan poslužitelj nije središte preko kojeg se vrši sva izmjena podataka, već je svim članovima mreže ravnopravno omogućeno međusobno komunicirati i razmjenjivati podatke

SQL – Standard Query Language

DCG – Dash Core Group

ADS – Authentic Data Structure – provjerljiva struktura podataka

8. SAŽETAK

Jedno od rješenja izazova povezanih s outsource-anjem i integritetom podataka u bazama podataka jest primjena provjerljivih baza podataka, a to su baze podataka koje uz traženi sadržaj osiguravaju i dokaz da podaci sadržajno nisu promijenjeni.
U ovom radu je predstavljena hijerarhijska provjerljiva baza podataka GroveDB koja je razvijena u programskom okruženju Dash kriptovalute i može se koristiti unutar Dash ekosustava, ali i samostalno ili integrirana u druge sustave. Za drugu opciju u radu su pokazani funkcionalne i performansne mogućnosti baze. Baza se kreira kroz program koji također sadrži funkcije za rad s bazom podataka. Na bazi se ne može koristiti SQL sintaksa, nego se dodavanje podataka i upiti na bazu rade koristeći funkcije specifične za GroveDB. GroveDB ima implementirane mehanizme replikacije na koliko god je potrebno lokacija baze, što omogućuje distribuciju podataka i povećava područje primjene. Baza ima pojam root hasha koji je jednak dokazu koji baza osigurava uz upite na bazu. Iz performansnih testova se može uočiti blagi pad brzine dodavanja podataka proporcionalno rastu baze i stabilna brzina upita bez obzira na veličinu baze.

Ključne riječi: provjerljiva baza podataka, NoSQL, Merkle stablo, GroveDB

SUMMARY

One of the solutions to challenges associated with outsourcing and data integrity in databases is the application of verifiable databases. These databases provide not only the requested content but also proof that the content has not been altered. This paper presents GroveDB, a hierarchical verifiable database developed within the Dash cryptocurrency programming environment. It can be used within the Dash ecosystem, as well as independently. The paper demonstrates the functional and performance capabilities of the database. The database is created through a program that includes functions for database management. SQL syntax cannot be used on the database; instead, data additions and queries are conducted using a syntax more complex than SQL. GroveDB incorporates replication mechanisms at as many database locations as needed, as demonstrated in this work, enabling data distribution and expanding its application scope. The database includes a concept of a root hash, which is equivalent to the proof provided with database queries. Performance tests indicate that the speed of data addition slows as the database grows in size, while the querying speed remains consistent regardless of the database size.

Keywords: verifiable database structure, NoSQL, Merkle tree, GroveDB

9. PRILOG

9.1 Primjer korištenja *insert* i *get* funkcije

```
use grovedb::{operations::insert::InsertOptions, Element, GroveDb,
PathQuery, Query, Transaction};
use grovedb_version::version::GroveVersion;

const TREE1: &[u8] = b"tree1";
const root_path: &[&[u8]] = &[];

// Allow insertions to overwrite trees
// This is necessary so the tutorial can be rerun easily
const INSERT_OPTIONS: Option<InsertOptions> = Some(InsertOptions {
    validate_insertion_does_not_override: false,
    validate_insertion_does_not_override_tree: false,
    base_root_storage_is_free: true,
});

fn main() {
    // Specify a path and open GroveDB at the path as db
    let path = String::from("../tutorial-storage");
    let db = GroveDb::open(path).unwrap();

    let grove_version = GroveVersion::latest();


    // Insert key-value 1 into the root tree
    db.insert(
        root_path,
        b"key1",
        Element::new_item(b"vrijednost-1.1".to_vec()),
        None,
        None,
        grove_version,
    )
    .unwrap()
    .expect("successful key1 insert");

    insert_empty_tree_db(&db, root_path, TREE1, &grove_version);

    // Insert key-value 2 into the tree1
    db.insert(
        &[TREE1],
        b"key2",
        Element::new_item(b"vrijednost2_1".to_vec()),
        None,
        None,
        grove_version,
    )
    .unwrap()
```

```
    .expect("successful key2 insert");

  // Insert key-value 3 into the tree1
    db.insert(
        &[TREE1],
        b"key3",
        Element::new_item(b"vrijednost2_2".to_vec()),
        None,
        None,
        grove_version,
    )
    .unwrap()
    .expect("successful key2 insert");

    // At this point the Items are fully inserted into the database.
    // No other steps are required.

    // To show that the Items are there, we will use the get()
    // function to get them from the RocksDB backing store.

    // Get value 1
    let result1 = db.get(root_path, b"key1", None,
grove_version).unwrap();

    // Get value 2
    let result2 = db.get(&[TREE1], b"key2", None,
grove_version).unwrap();

    // Print the values to terminal
    println!("{:?}", result1);
    println!("{:?}", result2);
}

fn insert_empty_tree_db(db: &GroveDb, path: &[&[u8]], key: &[u8],
grove_version: &GroveVersion)
{
    db.insert(path, key, Element::empty_tree(), INSERT_OPTIONS, None,
grove_version)
        .unwrap()
        .expect("successfully inserted tree");
}
```

9.2 Primjer korištenja query funkcije

```
use grovedb::{operations::insert::InsertOptions, Element, GroveDb,
PathQuery, Query};
use grovedb_version::version::GroveVersion;

const TREE1: &[u8] = b"tree1";
const TREE2: &[u8] = b"tree2";

// Allow insertions to overwrite trees
// This is necessary so the tutorial can be rerun easily
const INSERT_OPTIONS: Option<InsertOptions> = Some(InsertOptions {
```

```
    validate_insertion_does_not_override: false,
    validate_insertion_does_not_override_tree: false,
    base_root_storage_is_free: true,
});

fn main() {
    // Specify the path where the GroveDB instance exists.
    let path = String::from("../tutorial-storage");

    let grove_version = GroveVersion::latest();

    // Open GroveDB at the path.
    let db = GroveDb::open(path).unwrap();

    // Populate GroveDB with values. This function is defined below.
    populate(&db);

    // Define the path to the subtree we want to query.
//    let path = vec![KEY1.to_vec(), KEY2.to_vec()];
    let path = vec![TREE1.to_vec(), TREE2.to_vec()];

    // Instantiate a new query.
    let mut query = Query::new();

    // Insert a range of keys to the query that we would like
returned.
    // In this case, we are asking for keys 30 through 34.

query.insert_range(45_u8.to_be_bytes().to_vec()..55_u8.to_be_bytes().t
o_vec());

    // Put the query into a new unsized path query.
    let path_query = PathQuery::new_unsized(path, query.clone());

    // Execute the query and collect the result items in "elements".
    let (elements, _) = db
        .query_item_value(&path_query, true, false, true,None,
&grove_version)
        .unwrap()
        .expect("expected successful get_path_query");

    // Print result items to terminal.
    println!("{:?}", elements);
}

fn populate(db: &GroveDb) {
    let root_path: &[&[u8]] = &[];

    let grove_version = GroveVersion::latest();

    // Put an empty subtree into the root tree nodes at KEY1.
    // Call this SUBTREE1.
```

```
    db.insert(root_path, TREE1, Element::empty_tree(), INSERT_OPTIONS,
None, grove_version)
        .unwrap()
        .expect("successful SUBTREE1 insert");

    // Put an empty subtree into subtree1 at KEY2.
    // Call this SUBTREE2.
    db.insert(&[TREE1], TREE2, Element::empty_tree(), INSERT_OPTIONS,
None, grove_version)
        .unwrap()
        .expect("successful SUBTREE2 insert");

    // Populate SUBTREE2 with values 0 through 49 under keys 0 through
49.
    for i in 0u8..50 {
        let i_vec = (i as u8).to_be_bytes().to_vec();
        db.insert(
            &[TREE1],
            &i_vec,
            Element::new_item(i_vec.clone()),
            INSERT_OPTIONS,
            None,
            grove_version,
        )
        .unwrap()
        .expect("successfully inserted values");
    }

   // Populate SUBTREE1 with values 50 through 99 under keys 50
through 99.
    for i in 50u8..100 {
        let i_vec = (i as u8).to_be_bytes().to_vec();
        db.insert(
            &[TREE1, TREE2],
            &i_vec,
            Element::new_item(i_vec.clone()),
            INSERT_OPTIONS,
            None,
            grove_version,
        )
        .unwrap()
        .expect("successfully inserted values");
    }

}
```

9.3 Primjer replikacije

```
use std::collections::VecDeque;
use std::path::Path;
use grovedb::{operations::insert::InsertOptions, Element, GroveDb,
PathQuery, Query, Transaction};
```

```
use grovedb::reference_path::ReferencePathType;
use rand::{distributions::Alphanumeric, Rng, };
use grovedb::element::SumValue;
use grovedb::replication::CURRENT_STATE_SYNC_VERSION;
use grovedb::replication::MultiStateSyncInfo;
use grovedb_version::version::GroveVersion;

const ROOT_PATH: &[&[u8]] = &[];

// Allow insertions to overwrite trees
// This is necessary so the tutorial can be rerun easily
const INSERT_OPTIONS: Option<InsertOptions> = Some(InsertOptions {
    validate_insertion_does_not_override: false,
    validate_insertion_does_not_override_tree: false,
    base_root_storage_is_free: true,
});



fn create_empty_db(grovedb_path: String) -> GroveDb   {
    let db = GroveDb::open(grovedb_path).unwrap();
    db
}

fn main() {
    let grove_version = GroveVersion::latest();

    // 1. Creating base 1
    let path_base1 = generate_random_path("../tutorial-storage/",
"/db_base1", 24);
    let db_base1 = create_empty_db(path_base1.clone());

    // 2. Creating base 2
    let path_base2 = generate_random_path("../tutorial-storage/",
"/db_base2", 24);
    let db_base2 = create_empty_db(path_base2.clone());

    // 3. Populate base 1 with STUDENT, SUBJECT, and GRADE tables
    // Create subtrees for each table
    insert_empty_tree_db(&db_base1, ROOT_PATH, b"student",
&grove_version);
    insert_empty_tree_db(&db_base1, ROOT_PATH, b"subject",
&grove_version);
    insert_empty_tree_db(&db_base1, ROOT_PATH, b"grade",
&grove_version);

    // Insert data into STUDENT table
    let students = vec![
        (b"1", "Ivo Ivić".as_bytes()),
        (b"2", "Pero Perić".as_bytes()),
    ];
    for (student_id, student_name) in students {
        db_base1.insert(
```

```
            &[b"student"],
            student_id,
            Element::new_item(student_name.to_vec()),
            INSERT_OPTIONS,
            None,
            &grove_version,
        ).unwrap().expect("successfully inserted student");
    }

    // Insert data into SUBJECT table
    let subjects = vec![
        (b"1", "Matematika".as_bytes()),
        (b"2", "Algoritmi".as_bytes()),
    ];
    for (subject_id, subject_name) in subjects {
        db_base1.insert(
            &[b"subject"],
            subject_id,
            Element::new_item(subject_name.to_vec()),
            INSERT_OPTIONS,
            None,
            &grove_version,
        ).unwrap().expect("successfully inserted subject");
    }

    // Insert data into GRADE table
    let grades = vec![
        (b"1_1", 4i32), // Ivo - matematika
        (b"1_2", 5i32), // Ivo - algoritmi
        (b"2_1", 3i32), // Pero - matematika
        (b"2_2", 4i32), // Pero - algoritmi
    ];
    for (grade_key, grade_value) in grades {
        db_base1.insert(
            &[b"grade"],
            grade_key,
            Element::new_item(grade_value.to_be_bytes().to_vec()),
            INSERT_OPTIONS,
            None,
            &grove_version,
        ).unwrap().expect("successfully inserted grade");
    }

    // 4. Print the root hash data of both databases
    println!("\n######### root_hashes before replication:");
    let root_hash_base1 = db_base1.root_hash(None,
grove_version).unwrap().unwrap();
    println!("root_hash_base1: {:?}", hex::encode(root_hash_base1));
    let root_hash_base2 = db_base2.root_hash(None,
grove_version).unwrap().unwrap();
    println!("root_hash_base2: {:?}", hex::encode(root_hash_base2));

    // 5. Replication of base 1 (source) to base 2 (destination)
```

```
    let state_info = MultiStateSyncInfo::default();
    let tx = db_base2.start_transaction();
    sync_db_demo(&db_base1, &db_base2, state_info, &tx,
&grove_version).unwrap();
    db_base2.commit_transaction(tx).unwrap().expect("expected to
commit transaction");

    // 6. Print the root hash data of both databases
    println!("\n######### root_hashes after replication:");
    let root_hash_base1 = db_base1.root_hash(None,
grove_version).unwrap().unwrap();
    println!("root_hash_base1: {:?}", hex::encode(root_hash_base1));
    let root_hash_base2 = db_base2.root_hash(None,
grove_version).unwrap().unwrap();
    println!("root_hash_base2: {:?}", hex::encode(root_hash_base2));

    // 7. Printing the corresponding value for the same key from both
databases
    let query_path_student: &[&[u8]] = &[b"student"];
    let query_key_student = b"1".to_vec(); // Query for John Doe
    let query_path_subject: &[&[u8]] = &[b"subject"];
    let query_key_subject = b"2".to_vec(); // Query for Math
    let query_path_grade: &[&[u8]] = &[b"grade"];
    let query_key_grade = b"1_2".to_vec(); // Query for John Doe's
grade in Math
    println!("\n######## Query on source:");
    query_db(&db_base1, query_path_student, query_key_student.clone(),
&grove_version);
    query_db(&db_base1, query_path_subject, query_key_subject.clone(),
&grove_version);
    query_db(&db_base1, query_path_grade, query_key_grade.clone(),
&grove_version);

    println!("\n######## Query on replika:");
    query_db(&db_base2, query_path_student, query_key_student.clone(),
&grove_version);
    query_db(&db_base2, query_path_subject, query_key_subject.clone(),
&grove_version);
    query_db(&db_base2, query_path_grade, query_key_grade.clone(),
&grove_version);
}

fn insert_empty_tree_db(db: &GroveDb, path: &[&[u8]], key: &[u8],
grove_version: &GroveVersion)
{
    db.insert(path, key, Element::empty_tree(), INSERT_OPTIONS, None,
grove_version)
        .unwrap()
        .expect("successfully inserted tree");
}
```

```
fn generate_random_path(prefix: &str, suffix: &str, len: usize) ->
String {
    let random_string: String = rand::thread_rng()
        .sample_iter(&Alphanumeric)
        .take(len)
        .map(char::from)
        .collect();
    format!("{}{}{}", prefix, random_string, suffix)
}

fn query_db(db: &GroveDb, path: &[&[u8]], key: Vec<u8>, grove_version:
&GroveVersion) {
    let path_vec: Vec<Vec<u8>> = path.iter()
        .map(|&slice| slice.to_vec())
        .collect();

    let mut query = Query::new();
    query.insert_key(key.clone());

    let path_query = PathQuery::new_unsized(path_vec, query.clone());

    let (elements, _) = db
        .query_item_value(&path_query, true, false, true, None,
grove_version)
        .unwrap()
        .expect("expected successful get_path_query");

    for value in elements.into_iter() {
        if path.iter().any(|&p| p == b"grade") {
            // Convert the byte array back to an integer
            let grade = i32::from_be_bytes(value.try_into().unwrap());
            println!("Value: {}", grade);
        } else {
            // Convert the byte array back to a string
            let name = String::from_utf8(value).unwrap();
            println!("Value: {}", name);
        }
    }

    let proof = db.prove_query(&path_query, None,
grove_version).unwrap().unwrap();
    // Get hash from query proof and print to terminal along with
GroveDB root hash.
    let (verify_hash, _) = GroveDb::verify_query(&proof, &path_query,
grove_version).unwrap();
    println!("verify_hash: {:?}", hex::encode(verify_hash));
    if verify_hash == db.root_hash(None,
grove_version).unwrap().unwrap() {
        println!("Query verified");
    } else {
        println!("Verification FAILED");
    }
}
```

```
fn sync_db_demo(
    source_db: &GroveDb,
    target_db: &GroveDb,
    state_sync_info: MultiStateSyncInfo,
    target_tx: &Transaction,
    grove_version: &GroveVersion,
) -> Result<(), grovedb::Error> {
    let app_hash = source_db.root_hash(None,
grove_version).value.unwrap();
    let mut state_sync_info =
target_db.start_snapshot_syncing(state_sync_info, app_hash, target_tx,
CURRENT_STATE_SYNC_VERSION, grove_version)?;

    let mut chunk_queue : VecDeque<Vec<u8>> = VecDeque::new();

    // The very first chunk to fetch is always identified by the root
app_hash
    chunk_queue.push_back(app_hash.to_vec());

    while let Some(chunk_id) = chunk_queue.pop_front() {
        let ops = source_db.fetch_chunk(chunk_id.as_slice(), None,
CURRENT_STATE_SYNC_VERSION, grove_version)?;
        let (more_chunks, new_state_sync_info) =
target_db.apply_chunk(state_sync_info, chunk_id.as_slice(), ops,
target_tx, CURRENT_STATE_SYNC_VERSION, grove_version)?;
        state_sync_info = new_state_sync_info;
        chunk_queue.extend(more_chunks);
    }

    Ok(())
}
```