

Elektronski fakultet, Niš

Smer: Računarstvo i informatika, Softversko inženjerstvo

**Predmet: Sistemi za upravljanje bazama podatka**

Tema:

**Optimizacija upita kod Cassandra baze podataka**

Student: Dušan Dimitrov 1428

Mentor: Doc. dr Aleksandar Stanimirović

Niš, Novembar 2022

**Sadržaj**

[Uvod u NOSQL i Cassandra bazu podataka 3](#_Toc119578205)

[Optimizacija upita kod Cassandra baze podataka 5](#_Toc119578206)

[Query driven pristup 5](#_Toc119578207)

[ALLOW FILTERING 7](#_Toc119578208)

[Prepared statements-pripremljene naredbe 7](#_Toc119578209)

[Batch 8](#_Toc119578210)

[Implementacija Demo aplikacije za demonstraciju batch naredbe 9](#_Toc119578211)

[Indeksi 13](#_Toc119578212)

[Tombstone 18](#_Toc119578213)

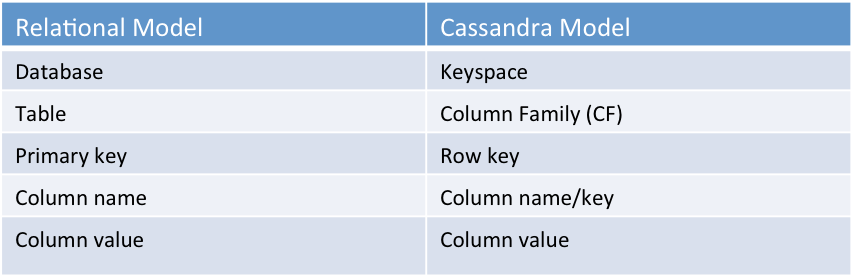
[Zaključak 21](#_Toc119578214)

[Literatura 22](#_Toc119578215)

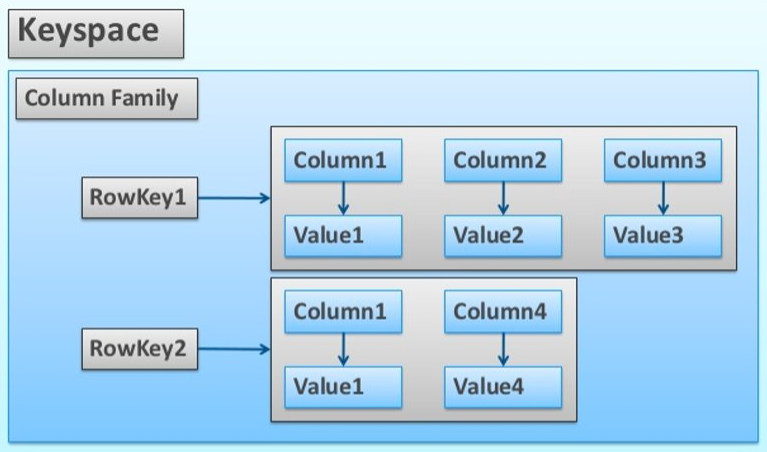
# Uvod u NOSQL i Cassandra bazu podataka

U današnje vreme se sve više kreiraju i koriste mobilne i web aplikacije, što je dovelo do određenih promena. U svakom trenutku puno aktivnih korisnika generiše sadržaj, a tradicionalne relacione baze podataka postaju usko grlo za moderne aplikacije. One se teško skaliraju i ne idu u korak sa određenim aplikacijama, njihovim funkcionalnim i nefunkcionalnim zahtevima. Kao alternativa, pojavljuju se NoSQL baze podataka. One nastoje da reše neke od problema sa kojima se susreću relacione baze. Cassandra baza podataka predstavlja primer jako popularne NoSQL baze podataka, a CQL Cassandra Query Language predstavlja jezik za komunikaciju sa Cassandra bazom podataka. CQL je kreiran kako bi Cassandra ličila na relacione baze podataka.

Cassandra baza podataka predstavlja wide column bazu podataka. Arhitekturu je preuzela od DynamoDB-a, a model podataka, koji je od velikog značaja za ovaj rad, od Google Bigtable baze podataka. Model podataka je takav da podseca na model podataka kod relacionih baza podataka, a za to je zaslužan CQL. CQL pruža sličan interfejs kao i SQL, ali baš zato i sakriva prave detalje skladištenja podataka. Dok kod relacione baze postoji termin **baza podataka**, on je kod Cassandra baze podataka poznat kao **keyspace**. **Tabela** odgovara **familiji kolona (column family)**. **Primarni ključ** iz relacionog sveta predstavlja **ključ vrste (row key)** kod Cassandra baze podataka. **Naziv kolone** i **vrednost kolone** su zadržali isti naziv, što je moguće videti na narednoj slici:



Cassandra se može posmatrati kao četvorodimenzionalna heš mapa. Da bi se došlo do podatka, potrebno je znati keyspace, familiju kolona, ključ vrste i ključ/naziv kolone, a sam model podataka je moguće videti na sledećoj slici:



Glavna prednost Cassandra modela podataka je postojanje fleksibilnih vrsta. Svaka vrsta može da ima različite kolone. Kod relacionih baza je moguće da ne postoji neka vrednost u koloni, ali se zapravo ta kolona čuva u bazi, samo ima null vrednost. Kod Cassandra baze podataka to nije slučaj. Ukoliko vrsta nema kolonu, neće se čuvati null vrednost za nju. Cassandra omogućava da svaka vrsta ima totalno različite kolone. Međutim, sa time treba biti jako pažljiv. Kroz aplikaciju je to tešto ispratiti i omogućiti mapiranje između struktura podataka korišćenih u samoj aplikaciji i bazi podataka. Treba jako smisleno, pažljivo i povezano dodavati nove kolone.

Cassandra baza podataka sadrži primarni ključ, odnosno jedinstveni identifikator svake vrste. Sastoji se od 2 dela:

* particioni ključ
* klastering ključ.

**Particioni ključ** je obavezan. Služi kako bi se identifikovala particija, odnosno čvor na kome će biti sačuvan podatak. **Clustering key** nije obavezan, ali je neophodan za efikasnost Cassandra baze podataka. Služi kako bi po njemu podaci bili sortirani u okviru jedne particije.

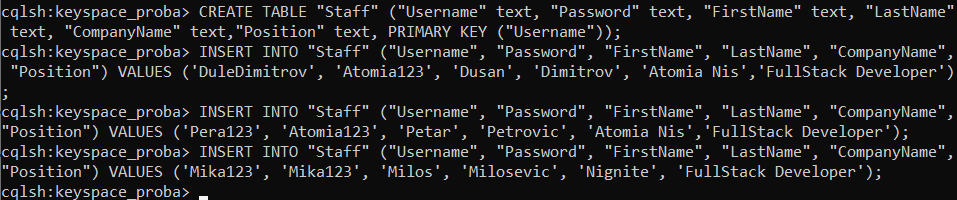
# Optimizacija upita kod Cassandra baze podataka

## Query driven pristup

Kada se priča o upitima, a i optimizaciji upita kod Cassandra baze podataka, treba se uzeti u obzir takozvani **query driven pristup**. Naime, neophodno je na samom početku projektovanja baze podataka razmišljati o upitima aplikacije. To se razlikuje od načina razmišljanja prilikom projektovanja relacionih baza podataka, kod kojih je uvek moguće doći do željenog podatka i informacije pomoću nekog više ili manje kompleksnog upita i join-a. Cassandra baza podataka ne poseduje mogućnost join-a. Prilikom razmišljanja o upitima kod Cassandra baze podataka, treba razmišljati o njenoj prednosti koja se odnosi na veliku brzinu upisa. Dakle, ne teži se normalizaciji podataka, već se može i neki podatak duplirati. Podaci se dupliraju zato što Cassandra baza podataka jako efikasno pristupa podacima po primarnom ključu, pa se za svaki upit kreira takav column family da mu se pristupa po primarnom ključu. Ukoliko je potrebno nekom entitetu pristupati po više različitih kolona, potrebno je kreirati više različitih column family-a. One će imati različit partition key, dok će im se ostale kolone poklapati i tako se postiže najosnovniji nivo optimizovanog upita.

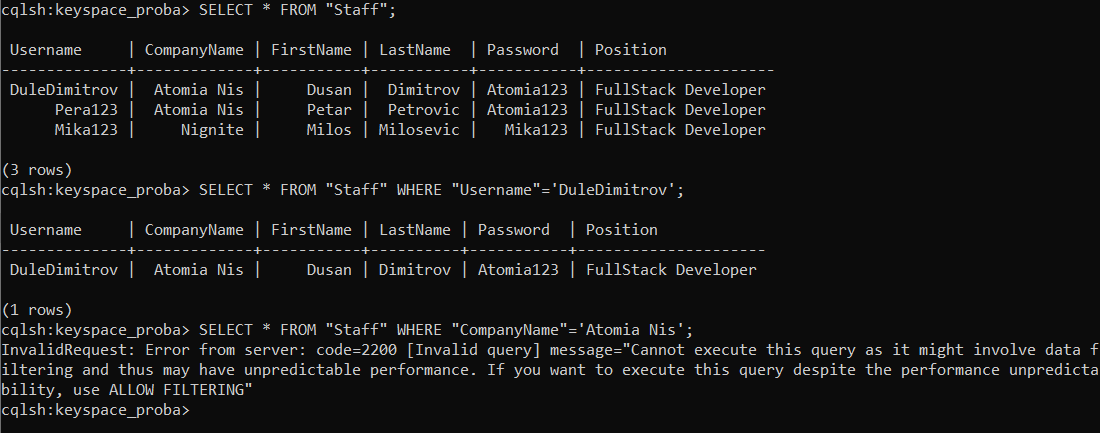
Posmatrajmo jednu aplikaciju za praćenje zaposlenih i kompanijama u kojima oni rade.

Za praćenje osnovnih informacija o zaposlenom, potrebno je kreirati jednu familiju kolona sa podacima o username-u-primarni ključ, password-u, imenu (FirstName), prezimenu (LastName), imenu kompanije (CompanyName), kao i poziciji na kojoj zaposleni trenutno radi. Kreiranje familije kolona, kao i par insert naredbi, prikazano je na sledećoj slici:



Pomoću ove familije kolona je pogodno naći najosnovnije informacije o zaposlenom i izvršiti najosnovnije SELECT upite. Moguće je izlistati sve zaposlene, pronaći informacije o zaposlenom na osnovu njegovog username-a-primarnog ključa. Međutim, ukoliko želimo pronaći sve zaposlene koji rade u jednoj kompaniji, to neće predstavljati najoptimizovaniji upit kod Cassandra baze podataka. Nešto kasnije u odeljku **ALLOW FILTERING** će biti detaljno objašnjeno zašto.

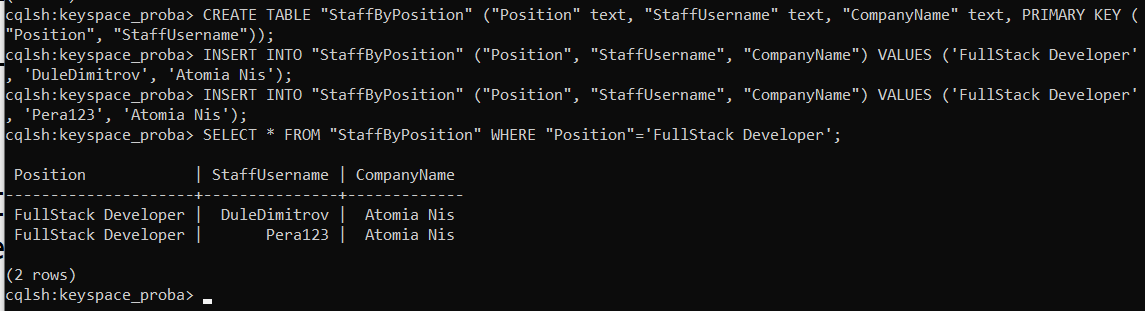
Na sledećoj slici je moguće videti osnovne upite koji rade, ali i pomenuti upit koji neće raditi:



Za dobavljanje informacija o zaposlenima neke kompanije, pogodno je kreirati još jednu familiju kolona: **StaffByCompany** koja će od kolona imati ime kompanije (CompanyName) , username zaposlenog (StaffUsername) i poziciju na kojoj zaposleni radi (Position). Naredbe za kreiranje familije kolona, kao i par osnovnih INSERT naredbi, moguće je videti na sledećoj slici, kao i upit za dobavljanje svih zaposlenih neke kompanije:



Sa druge strane, ukoliko je potrebno pribaviti informacije o zaposlenom na osnovu pozicije na kojoj radi, ove dve familije kolona neće biti pogodne. Potrebno je kreirati novu familiju kolona **StaffByPosition** koja će od kolona imati poziciju zaposlenog (Position), username zaposlenog (StaffUsername), kao i kompanija u kojoj radi (CompanyName). Naredbe za kreiranje ove familije kolona, par INSERT naredbi, kao i upit za dobavljanje informacija o zaposlenima po poziciji na kojoj rade je moguće videti na sledećoj slici:



## ALLOW FILTERING

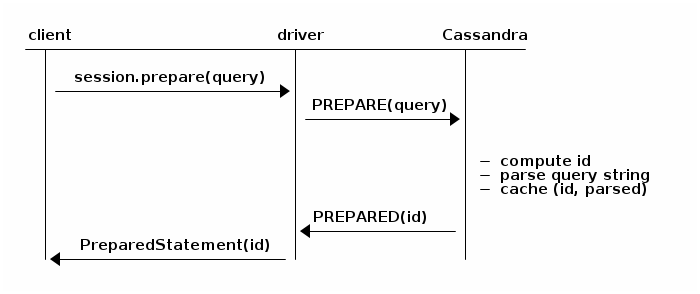
Cassandra baza podataka podrzava efikasno izvrsavanje upita pretrage podataka samo po primarnom ključu. Ukoliko se pokuša pretraga podataka na osnovu neke druge kolone dobiće se poruka:

***Bad Request: Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING.***

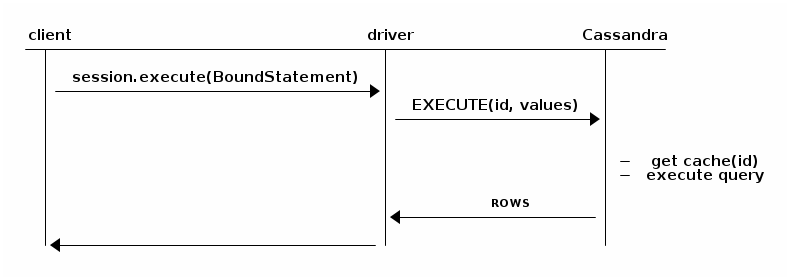
Za objašnjenje ove poruke treba posmatrati sam model podataka Cassandra baze podataka i kako su podaci skladišteni. Naime, u jednom realnom distribuiranom peer-to-peer Cassandra sistemu, podaci se nalaze na različitim particijama/čvorovima, a za određivanje particije je zadužen particioni ključ. Cassandra baza podataka omogućava pretragu po particionom ključu, zato što tako zna na kojoj particiji/čvoru da traži podatak. Nakon toga, moguće je još više suziti pretragu, dodavanjem i klastering ključa. Međutim, ukoliko Cassandra baza podataka ne zna na kojoj je particiji podatak, ona ih mora sve pretražiti, a to se zapravo dešava kada se zahteva pretraga po nekoj drugoj koloni. Cassandra baza podataka je svesna da će takva pretraga dugo trajati, pa zato i ispisuje adekvatnu poruku. Međutim, Cassandra baza podatak ostavlja mogućnost pribavljanja podatka na osnovu kolone koja nije primarni ključ uz **ALLOW FILTERING**. Tada će Cassandra baza podataka pronaći sve vrste koje odgovaraju tabeli, pa će od njih izvršiti filtriranje i vratiti adekvatne podatke. Ukoliko familija kolona sadrži milion vrsta, a cak 95% zadovoljava uslov pretrage, upit neće biti totalno neefikasan. Međutim, ukoliko familija kolona sadrži milion vrsta, a samo 2 vrste zadovoljavaju uslov pretrage, upit će biti totalno neefikasan. U tom slučaju bi Cassandra baza podataka učitala 999 998 vrsta ni za šta. Ukoliko je takav upit čest, uočava se znatno smanjivanje performansi same baze podataka. [1] Kako prosečan korisnik ne zna sve/većinu podataka u familiji kolona, tako on ni ne treba da koristi ALLOW FILTERING, već treba da izvrši **bolje planiranje familija kolona i upita** kao što je opisano u sekciji **Query driven pristup**.

## Prepared statements-pripremljene naredbe

Česta je situacija da se određene CQL naredbe (upiti) ponavljavljaju. Postavlja se pitanje da li je na neki način moguće iskoristiti što se naredbe ponavljaju i ubrzati njihovo izvršavanje. Odgovor je definitivno da. Cassandra baza podataka podržava nešto što se zovu pripremljene naredbe, ili u originalu **Prepared statements**. Naime, one se preporučuju za korišćenje kada se interaguje sa Cassandra bazom podataka, zato što znatno ubrzavaju izvršenje. Pripremljene naredbe se najpre prekompaljiraju i keširaju. Ovo je optimizacija koja omogućava parsiranje upita samo jednom, a njegovo izvršavanje više puta sa različitim vrednostima. Na sledećoj slici je moguće videti kako sam proces parsiranja i keširanja izgleda:



Bitno je napomenuti da Cassandra baza podataka nakon parsiranja i keširanja upita, vraća jedinstveni identifikator koji će kasnije biti korišćen. Kada se izvršava pripremljena naredba, drajver će samo poslati jedinstveni identifikator. Njegovo slanje omogućava preskakanje faze parsiranja. Parsirani upit se pronalazi u kešu na osnovu identifikatora i sam upit se izvršava kao što je moguće videti na sledećoj slici:



Pripremanje naredbe je potrebno izvršiti samo jednom. Međutim, ukoliko se pripremljeni upit izvršava samo jednom, sama priprema naredbe je neefikasna, zato što je bilo neophodno 2 puta kontaktirati Cassandra bazu podataka. Ukoliko se zna da će se određena naredba izvršavati više puta, definitivno treba razmišljati o pripremi naredbi koje gotovo svi drajveri Cassandra baze podataka podržavaju. [2]

## Batch

**BATCH** predstavlja DML naredbu kojom je moguće grupisati više INSERT, UPDATE i DELETE naredbi. Podseća na SQL transakciju, ali nije. Ukoliko je u BATCH-u reč o naredbama koje se odnose na 1 istu particiju, omogućena je **atomičnost** i **izolacija**. Ako je reč o naredbama koje manipulišu podacima na više particija, BATCH je **samo atomična** operacija. Atomičnost predstavlja princip sve ili ništa. Ili će se izvršiti sve naredbe iz BATCH-a, ili neće nijedna. Izolacija znači da nijedna BATCH naredba nema pristup promenama iz neke druge BATCH naredbe dok se ona kompletno ne izvrši. **Logovanje** omogućava atomičnost, tako što kada je uklljučeno, podatak se upisuje najpre u batchlog sistemsku tabelu. Ukoliko je više particija uključeno u BATCH, logovanje je uključeno (LOGGED) po defaultu. Logovanje je isključeno (UNLOGGED) po defaultu kada je reč o jednoj particiji. Cassandra prvo upisuje naredbe iz BATCH-a u batchlog tabelu. Odatle ih briše tek kada su sve operacije uspešno završene.

Struktura jedne BATCH naredbe izgleda [3]:

**BEGIN** [**UNLOGGED** | **LOGGED**] **BATCH**

[**USING** **TIMESTAMP** [***epoch\_microseconds***]]

***dml\_statement*** [**USING** **TIMESTAMP** [***epoch\_microseconds***]];

[***dml\_statement***; ...]

**APPLY** **BATCH**;

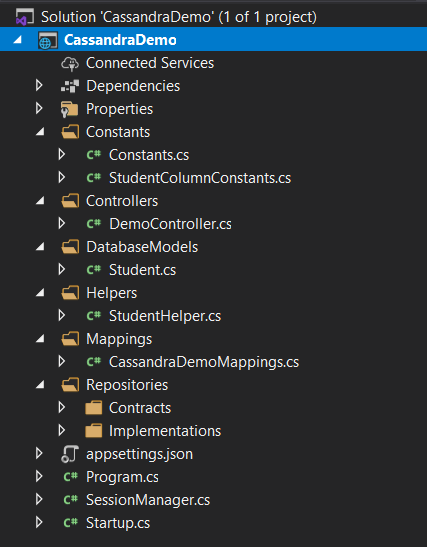
Definitivno se ne preporučuje korišćenje batch naredbe nad velikim brojem particija (sa velikim brojem particionih ključeva) zato što će se tako obavljati skeniranje na više nodova i to može znatno smanjiti performanse upita kod Cassandra baze podataka.

Takođe, preveliki broj naredbi u okviru jedne batch naredbe nije preporučljiv. To će takođe negativno uticati na performanse. Umesto jedne velike batch naredbe, preporučuje se korišćenje više izdeljenih batch naredbi. Mnogi drajveri Cassandra baze podataka čak izazivaju i runtime exception kada je batch naredba previše velika.

Batch naredba nije ni zamišljena da poboljša performanse naredbi u odnosu na njihovo izvršavanje van batcha. Batch služi kako bi uveo atomičnost i izolaciju i kako bi ubrzao grupno izvršavanje naredbi.

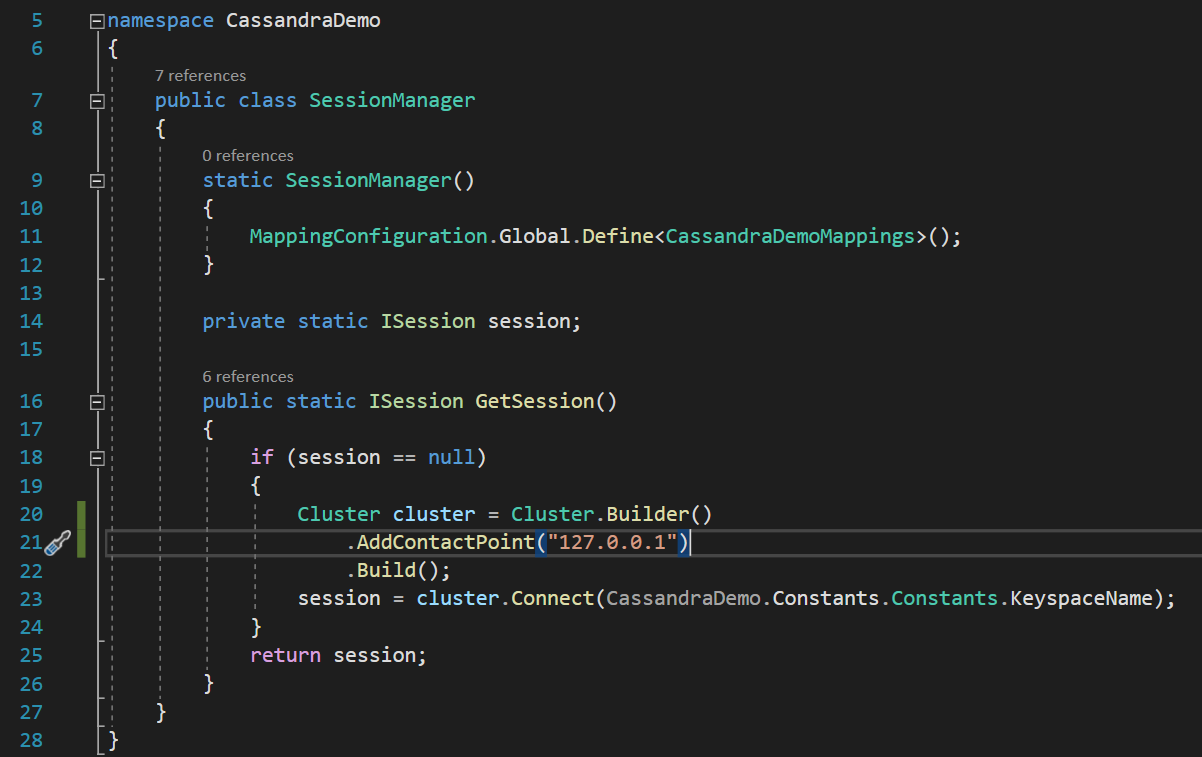
### Implementacija Demo aplikacije za demonstraciju batch naredbe

Kako bi se demonstrirao rad batch naredbe, kao i koja se optimizacija u performansama izvršenja upita postiže, implementirana je jedna .NET Core WebApi aplikacija pod nazivom **CassandraDemo**.



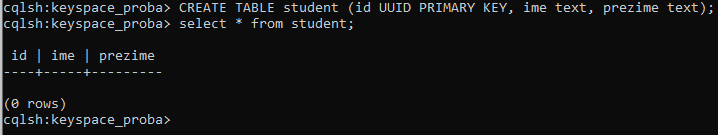
Za kreiranje konekcije prema Cassandra bazi podataka je iskorišćen **Singleton** projektni obrazac. Pomoću **SessionManager** statičke klase i statičke metode **GetSession** omogućeno je:

* kreiranje cluster instance,
* povezivanje na bazu,
* vraćanje sesije koja će biti korišćena u repozitorijumu.



Session Manager je takođe iskorišćen kako bi se u njegovom statičkom konstruktoru definisala odgovarajuća mapiranja između familija kolona i POCO klasa. Statički konstruktor je konstruktor koji će se pozvati samo jednom prilikom pokretanja aplikacije. Za mapiranja, ali i za kreiranje konekcije je iskorišćen NuGet paket **CassandraCSharpDriver Data Stax-a**.

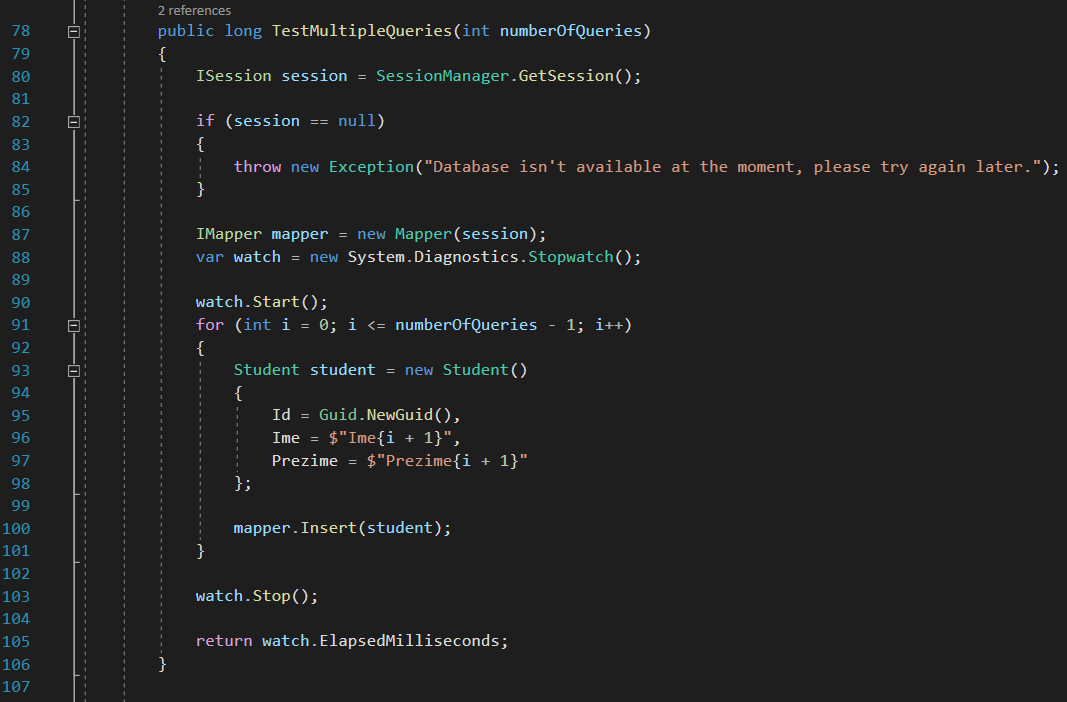
Za ovu demo aplikaciju je korišćena familija kolona student. Ona je kreirana putem osnovne CQL naredbe:



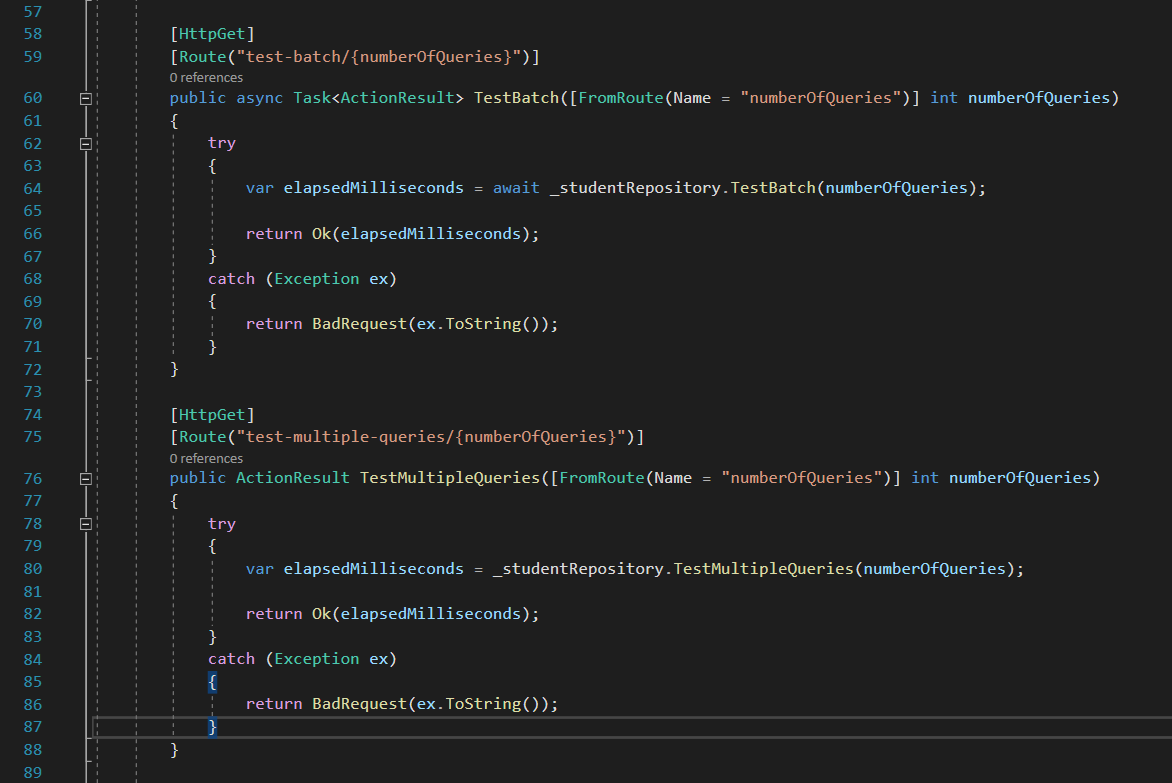
Za demonstraciju rada **batch-a**, najbitnije su metode **TestBatch** i **TestMultipleQueries** StudentRepository-a. **TestBatch** najpre kreira sesiju ka Cassandra bazi podataka pomoću SessionManager-a i nakon toga startuje jednu štopericu (watch). Zatim se kreira instanca batch-a, a kao tip batch-a se bira **BatchType.Logged**. Ovaj tip obezbeđuje atomičnost svih naredbi u batch-u. Nakon toga se dodaju naredbe u batch, a to su u ovom slučaju insert naredbe random generisanih studenata. Koliko će insert naredbi biti, zavisi od parametra metode numberOfQueries. Pozivom ***await mapper.ExecuteAsync(batch).ConfigureAwait(false);*** će se zapravo izvršiti sama batch naredba. Na kraju se zaustavlja štoperica (watch) i vraća vreme u milisekundama koje je prošlo od početka izvršavanja batch naredbe. Sve ovo je moguće videti na narednoj slici:



**TestMultipleQueries** je slična kao TestBatch metoda, međutim, ne izvršavaju se u batch-u insert naredbe, već nad mapper-om. To će se u CQL-u prevesti kao obična INSERT naredba. Kod koji ovo detaljno prikazuje je na sledećoj slici:

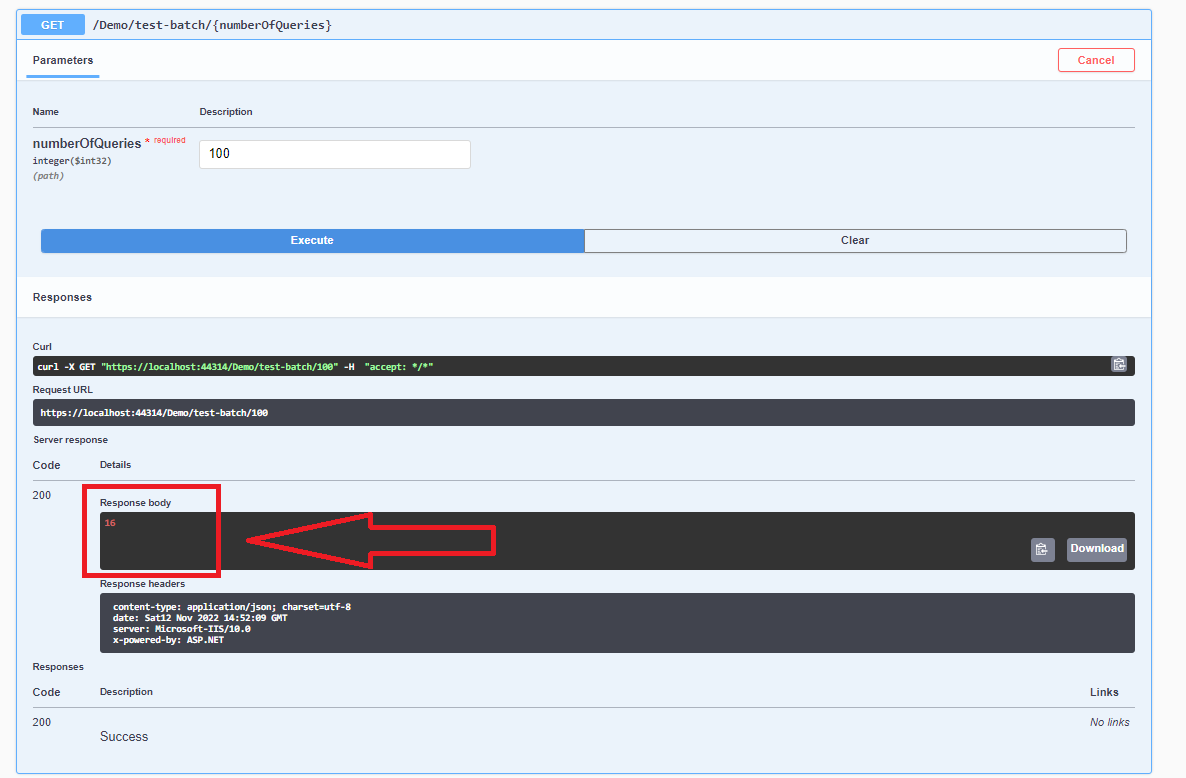


U DemoController-u su definisane 2 GET rute kako bi se testirale TestBatch i TestMultipleQueries metode koje su upravo objašnjene.

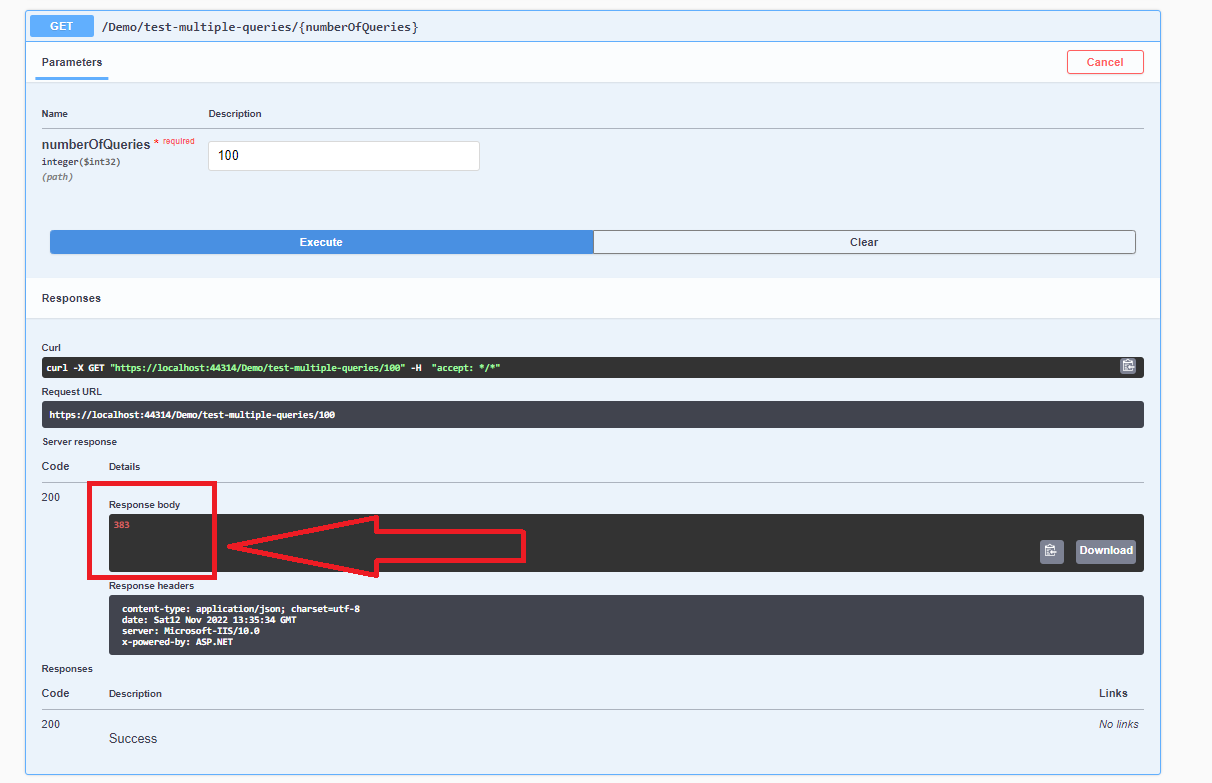


Dotnet sam konfiguriše Swagger za svaku definisanu rutu kontrolera, pa je ovo iskorišćeno za testiranje i zaključak koliko se zapravo ubrzava izvršenje naredbi kada su one u okviru jednog batch-a.

Test je (više puta) sproveden za **100 (insert) naredbi** i rezultat je da je za izvršenje naredbi u **batch-u** potrebno oko **16ms**, što je moguće videti na sledećoj slici:



Sa druge strane, **bez batch-a**, je potrebno oko **383ms**. Ovo je moguće videti na narednoj slici i time se uveriti u optimizaciju upita kroz batch:



## Indeksi

Indeksi i generalno indeksiranje kod Cassandra baze podataka mogu značajno uticati na performanse i optimizovati upite. Indeksi predstavljaju način pristupanja podacima kod Cassandra baze podataka po kolonama koje ne predstavljaju primarni ključ (**bez upotrebe ALLOW FILTERING).** Pravi benefit indeksa je brza i efikasna pretraga podataka koji zadovoljavaju određeni uslov. Indeks indeksira vrednosti kolone u novoj, posebnoj, skrivenoj tabeli. Indeksi se mogu koristiti za kolekcije i bilo koje kolone, **osim za brojače i statičke kolone**. Statička kolona predstavlja kolonu koju dele sve vrste jedne particije. [4]

Potrebno je lepo promisliti kada je potrebno koristiti indekse, a kada ne. Bolje je koristiti indekse kada vrednosti jedne kolone nemaju mnogo različitih vrednosti. Naime, ukoliko se kreira indeks nad kolonom koja ima puno različitih vrednosti, upit će zahtevati puno pretraga za malo rezultata. Na primer, posmatrajmo jedan sistem za praćenje pesama, njihovih autora i pevača. U familiji kolona koja ima milijardu pesama, pretraga pesme po autoru pesme može biti neefikasno. To je tako zato što je autor pesme nešto jedinstveno i specifično za pesmu. Sa druge strane, pevači uglavnom imaju više pesama koje pevaju, pa je bolje pretraživati po pevačima. Dakle, u ovom slučaju bi bilo pogodnije kreirati indeks za kolonu pevač. A ako nam je potrebna pretraga po autoru pesme, to je moguće ručno održavati dupliranjem podataka u novoj familiji kolona, gde je primarni ključ autor pesme. [5]

Naredba za kreiranje indeksa kod Cassandra baze podataka je **CREATE INDEX**. Ukoliko podaci u familiji kolona već postoje, Cassandra baza podataka će indeksirati podatke tokom naredbe. Nakon kreiranja indeksa, Cassandra baza podataka automatski indeksira podatke odmah nakon njihovog dodavanja u bazu. Sintaksa naredbe CREATE INDEX izgleda:

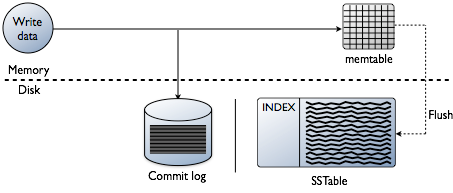
**CREATE INDEX IF NOT EXISTS index\_name**

**ON keyspace\_name.table\_name ( KEYS ( column\_name ) )**

* index\_name predstavlja identifikator samog indeksa. Nije obavezan, a ukoliko se ne navede, Cassandra baza podataka će mu dodeliti ime: **table\_name\_column\_name\_idx**
* column\_name predstavlja samu kolonu koje će postati indeks

Ovaj opisani indeks predstavlja osnovni, odnosno **regularni sekundarni indeks**, poznatiji kao i **2i**. Osim njega postoji i SSTable povezani sekundarni indeks, tj u originalu: **SSTable-Attached Secondary Index (SASI)**.

Da bi se bolje shvatila razlika ova 2 indeksa, potrebno je posmatrati upis podataka kod Cassandra baze podataka bez indeksa. Na nivou svakog pojedinačnog čvora upis počinje beleženjem aktivnosti o upisu u strukturi na disku koja se zove **commit log**.Ova struktura služi kako bi usled nestanka struje bila zabeležena informacija o upisu. Nakon dolaska struje se može proveriti da li postoji nešto što nije propagirano na disk i propagirati ga. Nakon toga se podatak skladišti u strukturi u memoriji koja se zove **memtable**. Memtable predstavlja keš kome će se kasnije pristupati prilikom čitanja podatka. Čuva podatke sortirano sve dok ima mesta u memtable-u i commit log-u. Zatim se podaci upisuju na disk (flush) u strukturu podataka koja je poznata kao **SSTable**. Nakon toga se commit log i memtable prazne.



Kod 2i se prilikom upisa upisuje podatak i u indeks i u Memtable. Nakon toga se istovremeno upisuje u SSTable, pa tako indeks ima poseban životni vek u odnosu na sam podatak. SASI indeks koristi novu logiku povezivanja životnog ciklusa SSTable-a i indeksa. Indeksiranje se zapravo obavlja i u memoriji, pa se posle određenog vremena upisuje zajedno indeks sa sve SSTable-om na disk.

Sintaksa SASI indeksa je:

**CREATE CUSTOM INDEX IF NOT EXISTS ime\_indeksa ON ime\_familije\_kolona (ime\_kolone) USING 'org.apache.cassandra.index.sasi.SASIIndex';**

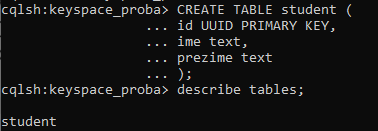
Prednosti SASI indeksa su tokenizacija pretrage teksta, brzo pretraživanje intervala vrednosti i takozvanog in-memory indeksiranja. Mana je definitivno to što se generišu veliki indeks fajlovi, pogotovo kada je podržana tokenizacija teksta. Međutim, treba napomenuti da su SASI indeksi eksperimentalni. Podržava ih DataStax, međutim, ne u produkciji.

SASI implementira 3 tipa indeksa:

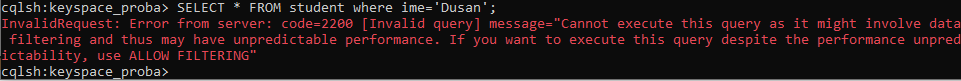
* PREFIX
* CONTAINS
* SPARSE.[6]

**PREFIX** tip indeksa je osnovni/defaultni mode rada i nije potrebno ništa dodatno specificirati, što nije slučaj sa druga 2.

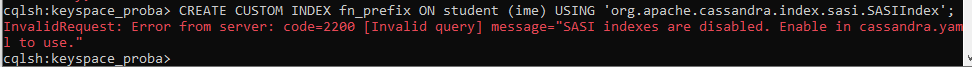
Na primer, ukoliko želimo kreirati familiju kolona studenata, sa njihovim id-em kao primarnim ključem i imenom i prezimenom, možemo kreirati PREFIX indeks za kolonu ime. CQL za kreiranje familije kolona izgleda ovako:



Ukoliko bismo odmah izvršili naredbu pretrage po koloni ime, dobili bismo adekvatnu poruku o ALLOW FILTERING:



Ukoliko bismo kreirali indeks, ovaj problem bismo resili. Naime, nakon prvog pokušaja kreiranja indeksa, se dešava sledeća situacija:



Neophodno je u cassandra.yaml fajlu izmeniti vrednosti 2 propertija na true:

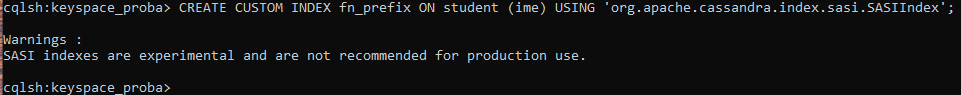
* enable\_sasi\_indexes
* enable\_transient\_replication.[7]

Nakon toga je neophodno sačuvati izmene. Kako je ovo rađeno sa verzijom DataStax Cassandra implementacije u docker kontejneru, potrebno je izvršiti sledeće naredbe u okviru samog kontejnera:

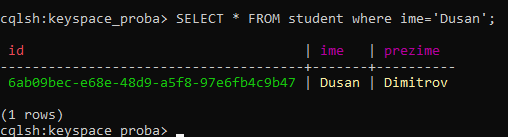
* nodetool flush system
* docker restart 1d3a8e7e6aa8 – gde je 1d3a8e7e6aa8 indeks kontejnera.

Ukoliko bi se radilo bez docker-a, neophodno je samo restartovati servis Cassandra instance.

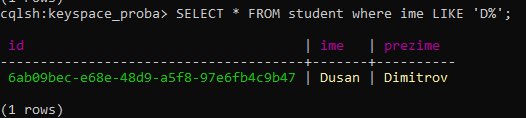
Nakon ovoga, moguće je izvršiti komandu za kreiranje SASI indeksa, uz adekvatnu komandu da su SASI indeksi još ekseprimentalni i da se ne preporučuju za produkciju:



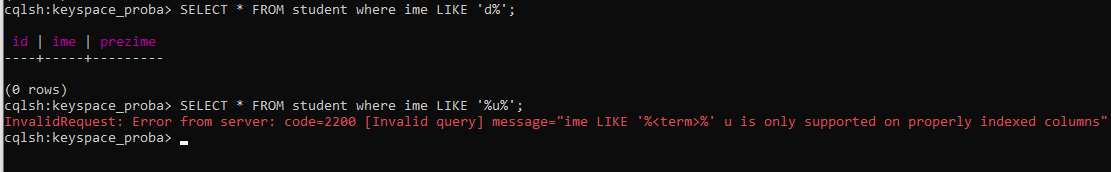
Nakon ovoga je moguće pretražiti familiju kolona student po koloni ime:



Sa ovim prefix SASI indeksom je moguće koristiti i ključnu reč LIKE, pa pronaći i imena studenata koja počinju sa D:



Prefiks indeks, ovako konfigurisan, nije case-insensitive, što je moguće videti i na sledećoj slici. Takođe je moguće videti da prefix index podržava LIKE pretragu samo ako se pretražuje početak kolone:

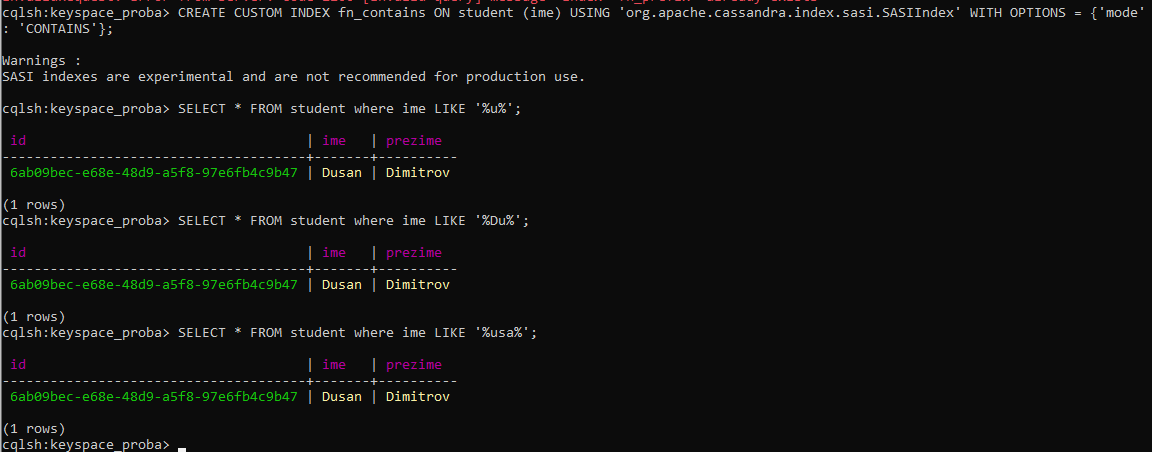


**CONTAINS SASI** indeks može rešiti pomenuti problem pretrage. Njegova struktura je:

**CREATE CUSTOM INDEX fn\_contains ON ime\_familije\_kolona (ime\_kolone) USING 'org.apache.cassandra.index.sasi.SASIIndex'**

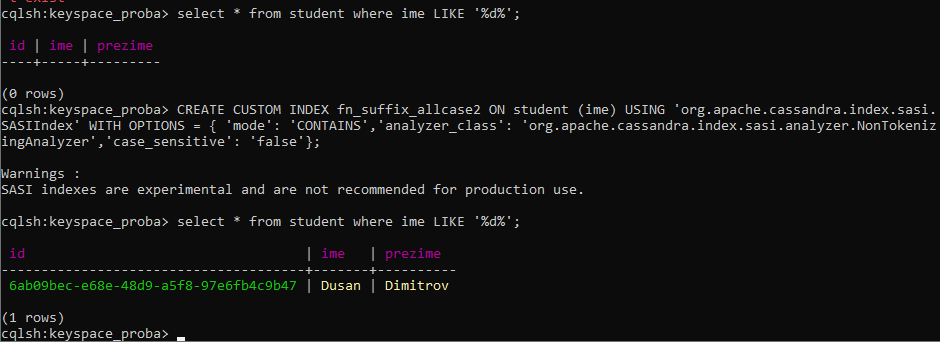
**WITH OPTIONS = { 'mode': 'CONTAINS' };**

Na sledećoj slici je moguće videti konkretnu naredbu za kreiranje CONTAINS indeksa, kao i rešenje problema pretrage podatka koji PREFIX indeks nije mogao da reši:



CONTAINS indeks takođe omogućava pretragu po vrednosti, ne samo LIKE pretragu. Međutim, u tom slučaju je neophodno dodati ALLOW FILTERING u naredbu, iako sama Cassandra baza podataka neće koristiti pravi mehanizam filtriranja. Dakle ovo je samo zbog sintakse.

I PREFIX i CONTAINS indeksi mogu da budu case-isensitive, ali ih je potrebno tako i napraviti. Kreiranje indeksa, koji može obavljati upite takve da rezultat ne zavisi od veličine slova (mala/velika) podatka:

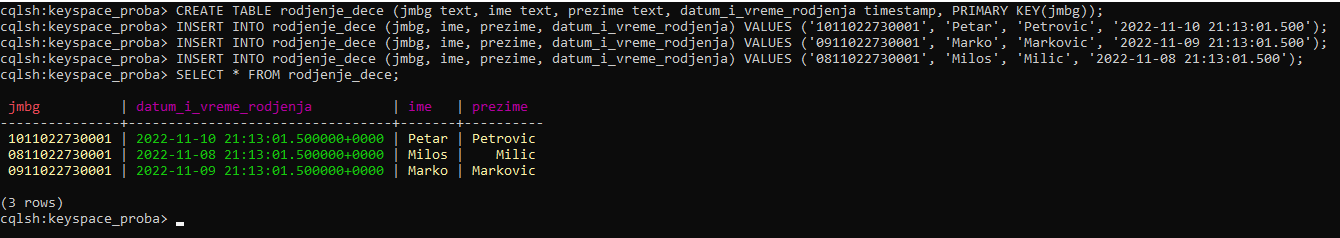


**SPARSE** indeks služi za poboljšavanje performansi upita koji se odnose na kolone sa velikim intervalom vrednosti, kao na primer timestamp-ovi. Struktura SPARSE indeksa je:

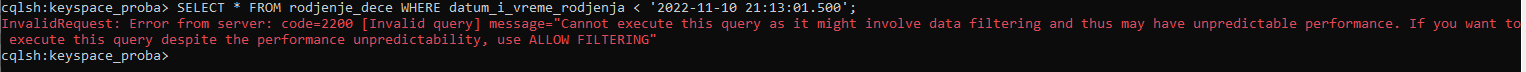
**CREATE CUSTOM INDEX fn\_sparse ON ime\_familije\_kolona (ime\_kolone) USING 'org.apache.cassandra.index.sasi.SASIIndex'**

**WITH OPTIONS = { 'mode': 'SPARSE' };**

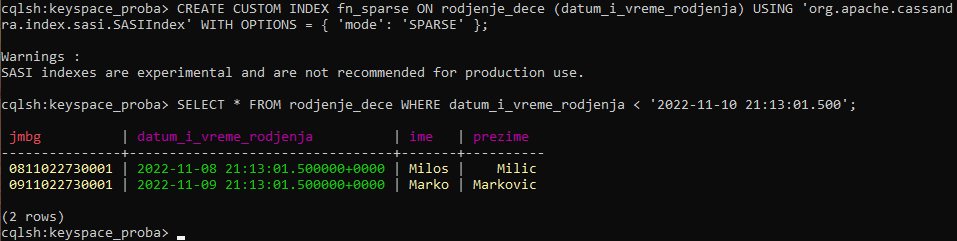
Posmatrajmo jedan primer familije kolona rodjenje\_dece, koja prati datum i vreme rodjenja dece. Na sledećoj slici je moguće videti kako se tačno kreira familija kolona, kao i kakvi podaci se unose:



Ukoliko bismo nad takvom familijom kolona, bez indeksa, pokušali da nađemo svu decu rođenu pre 10.novembra 2022. i 21:13:01.500, ne bismo mogli. Tačnije, dobili bismo poruku o neefikasnom upitu i kako se mora koristiti ALLOW FILTERING:



Međutim, nakon kreiranja SPARSE indeksa, ovakav upit postaje moguć:



Takođe, svi operatori poređenja: >, <, >=, <=, su podržani i validni. Treba napomenuti da SPARSE indeksi mogu da se koriste isključivo za numeričke podatke, tako da LIKE upiti ne mogu da se koriste nad poljem koje je indeksirano SPARSE indeksom.

## Tombstone

Tombstone-ovi mogu uticati na performanse kod Cassandra baze podataka. **Neki naizgled obični upiti mogu postati jako neefikasni.** U tom slučaju se ne obavlja optimizacija samog upita kroz njegovo menjanje, već na malo drugačiji način. Da bi se to objasnilo, neophodno je opisati tombstone.

Operacija **brisanja** je kod Cassandra baze podataka implementirana u osnovi isto kao i operacija upisa. Međutim, brisanje nije efikasno kao pisanje. Vrsta koju je potrebno obrisati prolazi kroz gotovo identičan proces kao i upis. Prvo se beleži informacija o upisu u commit log-u. Zatim se podatak upisuje u memtable i na kraju flush-uje u SSTable. Međutim, ono što razlikuje brisanje je to što se dodatno pamti i jedan fleg koji se zove **tombstone**. Ovaj fleg ima unapred definisan rok trajanja i na njegovom kraju dolazi do brisanja samog podatka.

Kod Cassandra baze podataka, tombstone se može generisati prilikom:

* DELETE naredbe
* Setovanja TTL-Time To Live
* INSERT-ovanja null vrednosti.

Podatak se briše kroz klasičan proces spajanja više SSTable-a, tako što se neće upisati u novi SSTable. Moguće je vrstu, pa čak i kolonu obeležiti ovim tombstone markerom. Time se podatku dodeljuje vreme posle koga će biti obrisan iz baze [8].

Kako bi se **tombstone obrisao**, moraju biti zadovoljeni sledeći kriterijumi:

* Tombstone mora biti kreiran pre gc\_grace\_seconds (definiše se u konfiguracionim fajlovima).
* Tabela koja sadrži tombstone je uključena u proces kompakcije.
* Svi SSTable-ovi koji sadrže relevantne podatke su uključeni u kompakciju. [9]

Međutim, tombstone-ovi mogu napraviti puno problema kod Cassandra baze podataka.

Prvi problem na koji se nailazi je da se prostor na disku ne oslobadja nakon brisanja makar gc\_grace\_seconds. Ovo definitivno može postati problem, ukoliko se cluster brzo puni. U određenim situacijama se može desiti da se sam prostor nikada ni ne oslobodi i tada je definitivno potrebna manualna intervencija.

Još veći problem mogu postati performanse čitanja. Poznato je da je Cassandra baza podataka pogodna za korišćenje kada ima više upisa, nego čitanja, ali da i čitanje može biti relativno efikasno, ukoliko se pridržavamo određenih pravila za dobavljanje podataka po primarnom ključu. Međutim, ukoliko postoji puno tombstone-ova, to može uticati na performanse čitanja. Problemi sa performansom se mogu desiti kod sledećih upita:

* Upiti koji se izvršavaju nad svim particijama familije kolona (npr. “SELECT \* FROM keyspace.table;”).
* Upiti koji su zasnovani na intervalu/range-u. U sekciji sa indeksima su obrađeni mehanizmi gde range upiti mogu biti efikasni, međutim, to se sve menja ukoliko ima puno tombstone-ova.
* Upiti koji sadrže ALLOW FILTERING. Više puta je pomenuto koliko ALLOW FILTERING može usporiti sam upit, ali ovde su usporenja još veća ukoliko ima puno tombstone-ova.

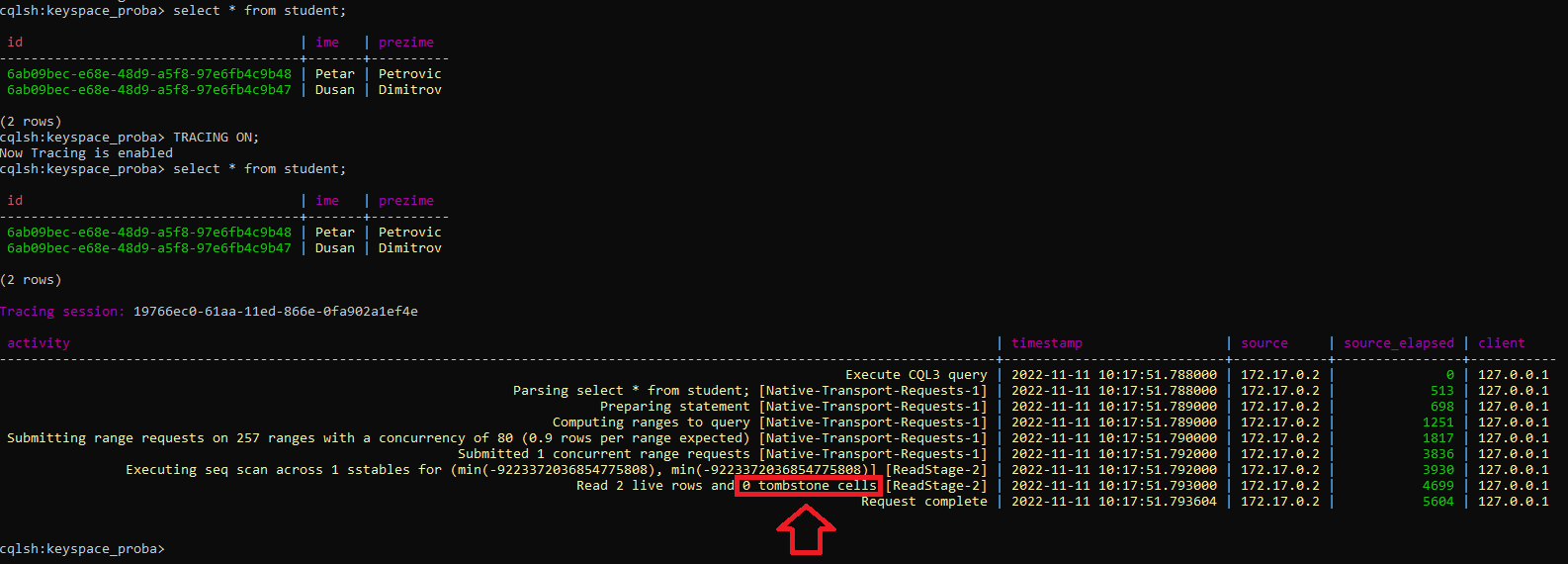
Postavlja se pitanje zašto ovi upiti postaju neefikasni ukoliko ima puno tombstone-ova. Odgovor je da će prosto Cassandra baza podataka onda morati da pre vraćanja podataka da fitrira vrednosti tombstone-ova i tako gubi na performansama. Svaki čvor treba da vrati koordinatoru sve svoje tombstone-ove za slučaj da ostale replike nisu svesne postojanja tog tombstone-a.

Postoje različiti mehanizmi za otkrivanje previše tombstone-ova. Naime, ukoliko postoji više od 1,000 tombstone-ova, generisaće se warning u log fajlovima za određeni upit:

*WARN org.apache.cassandra.db.ReadCommand Read 0 live rows and 87051 tombstone cells for query SELECT \* FROM example.table*

Takođe, upit se neće izvršiti ukoliko postoji više od 100,000 tombstone-ova i javiće se exception: **TombstoneOverwhelmingException**.

Osnovni način detekcije tombstone-ova je putem uključivanja tracing-a, naredbom: **TRACING ON**. Osim najosnovnijih informacija o svakom upitu, dobija se i broj tombstone-ova. Na sledećoj slici je to moguće videti:



Postoje načini za izbegavanje tombstone-ova. Najosnovnije pravilo je izbegavati ALLOW FILTERING. Ukoliko se koriste range upiti, voditi računa da se ne pretražuju brisani podaci i gledati da se range radi nad što užem intervalu vrednosti. Takođe, voditi računa o tome da je Cassandra baza podataka pogodna za aplikacije sa puno upisa i smislenim i promišljenim čitanjima. Nema efikasna ažuriranja i brisanja, pa samim tim ih treba i izbegavati i generalno dizajnirati sistem da nema puno brisanja. Ukoliko ima puno brisanja, definitivno nije pogodna Cassandra baza podataka. Preporuka za brisanje je da se zapravo brišu svi podaci iz tabele, putem truncate ili drop naredbe. Na taj način se brišu podaci, ali se ne generišu tombstone-ovi.

U najvećem broju slučajeva, najbolji pristup je prosto sačekati da se tombstone-ovi automatski izgube spajanjem SSTable-ova. Međutim, ponekad je potrebno i ranije reagovati, ukoliko određeni upiti postaju jako spori. Komanda **nodetool compact** forsira spajanje svih SSTable-ova. Ova komanda zahteva puno slobodnog prostora na disku. Od verzije 3.0 Cassandra baze podataka, postoji i korisna komanda **nodetool garbagecollect**. Ova komanda radi na principu više manjih spajanja SSTable-ova. Zahteva više CPU-a od nodetool compact komande, međutim, zahteva manje slobodnog prostora na disku. Napomenimo još jednom, tombstone-ovi će se automatski obrisati samo ako prođe vreme od gc\_grace\_seconds od trenutka kreiranja tombstone-ova. Svrha postojanja gc\_grace\_seconds je obezbeđivanje dovoljno vremena kako bi se ispravila konzistentnost u klasteru. Tako da definitivno treba pažljivo i promišljeno izvršavati komande za forsiranje brisanja tombstone-ova zato što to može dovesti do **“oživljavanja podataka”**.Odnosno, zbog postojanja tombstone-a na jednoj replici, inicijalno se sam podatak obriše. Međutim, kako on ne postoji na nekoj drugoj replici, ponovo će se upisati.

# Zaključak

U današnje vreme se sve više i više pojavljuju moderne arhitekture aplikacija koje su rezultat naprednijih funkcionalnih i nefunkcionalnih zahteva nekog sistema. Relacione baze podataka postaju usko grlo za skaliranje i ispunjavanje svih ostalih zahteva. Tu se pojavljuju NoSQL baze podataka. Cassandra, kao jedan od najpoznatijih predstavnika NoSQL baza podataka, je obrađen u ovom radu. Fokus je bio ompitimazacija upita kod Cassandra baze podataka.

Cassandra baza podataka poseduje CQL-Cassandra Query Language, koji itekako ima prostora za optimizaciju. U ovom radu su obrađene najpoznatije i najkorišćenije metode optimizacija upita, a sve je to propraćeno adekvatnim primerima.

Najpre je obrađen najosnovniji način optimizacije upita kod Cassandra baze podataka i generalno prva stvar sa kojom se moramo upoznati kada radimo upite u CQL-u. Naime, Cassandra podržava efikasne upite (bez korišćenja indeksa) samo po primarnom ključu. **Query driven princip** govori o tome da se na početku dizajniranja Cassandra sistema treba razmišljati o upitima. Za svaki upit treba kreirati adekvatnu tabelu, odnosno familiju kolona, kako je to preciznije reći kod Cassandra baze podataka. Tako da je za pribavljanje podataka potrebno koristiti odgovarajuću familiju kolona i njen primarni ključ. U suprotnom će Cassandra dati predlog korišćenja **ALLOW FILTERING**, a u ovom radu je objašnjeno zašto ga ne treba koristiti. (Napomena: kod korišćenja određenih indeksa je ALLOW FILTERING deo sintakse i tada se zapravo neće obaviti filtriranje podataka i tako smanjiti performanse Cassandra baze podataka.) Ukoliko je uočeno da se neki upiti često izvršavaju, moguće je optimizovati ih korišćenjem **prepare mehanizma**. Ovo pripremanje naredbe/upita ubrzava izvršenje upita na taj način što će se određene faze pripreme obaviti samo jednom, a ne prilikom svakog izvršenja upita. **Batch** predstavlja još jedan mehanizam optimizacije upita. Ubrzava se grupno izvršenje naredbi, a moguće je postići i atomičnost i izolaciju. U jednoj **demo aplikaciji** je demonstrirano korišćenje batch naredbe iz dotnet WebApi-a uz korišćenje DataStax-ovog drajvera. **Indeksi** predstavljaju optimizaciju upita kod gotovo svake baze podataka, pa su ovde obrađeni kod Cassandra baze podataka. Određeni napredni indeksi su samo eksperimentalno dostupni kod DataStax najnovije verzije Cassandra baze podataka (verzija 4.0), pa je pomoću docker-a sve to isprobano i demonstrirano u ovom radu.

# Literatura

[1] <https://www.datastax.com/blog/allow-filtering-explained>

[2] <https://docs.datastax.com/en/developer/java-driver/3.0/manual/statements/prepared/>

[3] <https://docs.datastax.com/en/cql-oss/3.x/cql/cql_reference/cqlBatch.html>

[4]<https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/usePrimaryIndex.html#:~:text=Indexing-,An%20index%20provides%20a%20means%20to%20access%20data%20in%20Cassandra,other%20than%20the%20partition%20key>

[5] <https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useWhenIndex.html>

[6] <https://docs.datastax.com/en/dse/5.1/cql/cql/cql_using/useSASIIndex.html>

[7] <https://medium.com/featurepreneur/enabling-sasi-index-86604c37d31b>

[8]<https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/dml/dmlAboutDeletes.html>

[9]https://www.instaclustr.com/support/documentation/cassandra/using-cassandra/managing-tombstones-in-cassandra/