Univerzitet u Nišu

Elektronski fakultet

Katedra za računarstvo

*Optimizacija upita*

*Korišćenjem Microsoft SQL Server baze podataka*

****

**Mentor:** **Student:**

Prof. dr Aleksandar Stanimirović Jelena Tošić, 1116

Niš, 2021. godine

**Sadržaj:**

[1. Uvod 1](#_Toc68804637)

[2. Definisanje parametara optimizacije 2](#_Toc68804638)

[2.1 Pregled korišćenih podataka 2](#_Toc68804639)

[2.2 Parametri optimizacije upita 3](#_Toc68804640)

[3. Optimizacija upita u MSSQL-u 4](#_Toc68804641)

[3.1 Korišćenje *OR* operatora u *Join* predikatu 4](#_Toc68804642)

[3.2 Korišćenje džoker karaktera na početku i kraju pretraživane fraze 6](#_Toc68804643)

[3.2.1. Pretraga džoker karakterima korišćenjem N-gram metode 7](#_Toc68804644)

[3.3. Ogromni upiti 10](#_Toc68804645)

[3.4. Indeksi i optimizacija 13](#_Toc68804646)

[3.2.1 Indeksi u SQL Serveru 14](#_Toc68804647)

[3.4.2. Query Optimizer u SQL Serveru 14](#_Toc68804648)

[3.4.3. Optimizacija upita dodavanjem odgovarajućih indeksa 15](#_Toc68804649)

[3.4.4. Optimizacija upita dodavanjem predloženih indeksa 18](#_Toc68804650)

[4. Zaključak 20](#_Toc68804651)

[5. Literatura 21](#_Toc68804652)

# Uvod

Cilj različitih tehnika koje se bave optimizacijom performansi prilikom keiranja upita ka različitim bazama podataka je smanjiti vreme odziva i iskoristiti sistemske resurse na najbolji mogući način. Iskorišćavanje sistemskih resursa pre svega uključuje minimizaciju CPU vremena, zatim mrežnog prometa i naravno broja čitanja i upisa u memoriju diska. Ostvarivanje ovog cilja direktno zavisi od logičke i fizičke strukture podataka, kao i zahteva koji su postavljeni nad datim sistemom, odnosno aplikacijom koja koristi podatke.

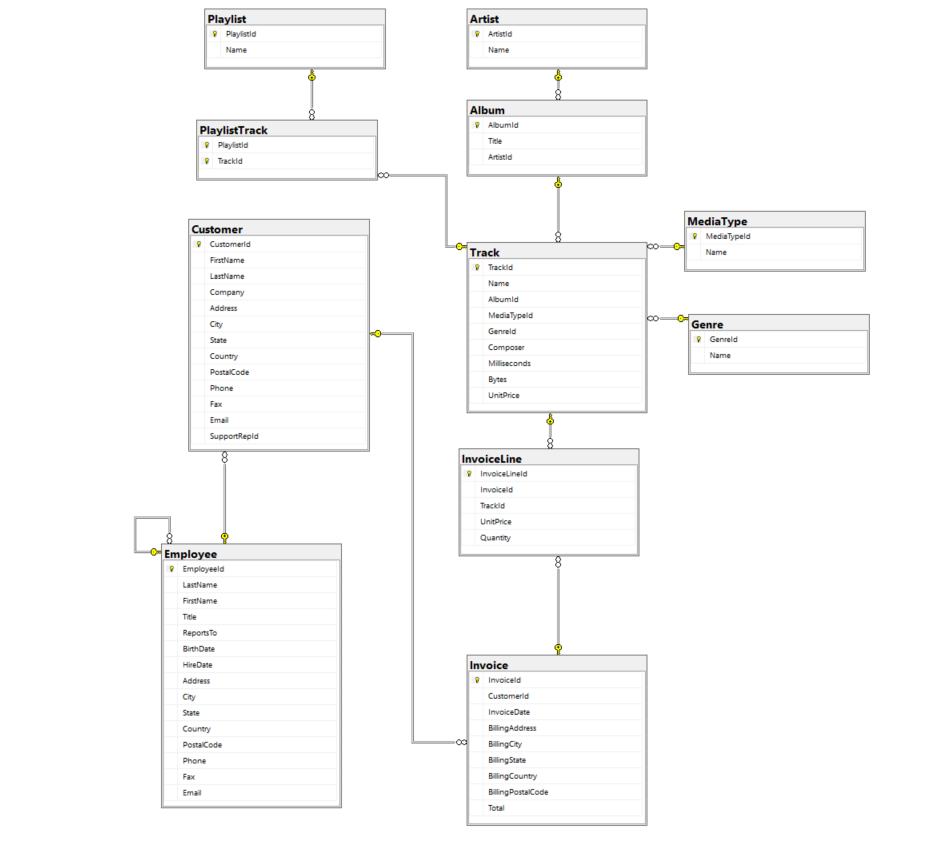
Budući da su strukture baze podataka složene, u većini slučajeva, za različite vrste upita, neophodnim podacima moguće je pristupiti na različite načine, korišćenjem različitih struktura podataka i pristupom u različitim redosledima. Ovi različiti načini za pristup podacima često zahtevaju različito vreme obrade, pri čemu mogu značajno uticati na vreme odziva ukoliko nisu adekvatno napisani. Zato je glavna težnja svih tehnika za optimizaciju upita da ponađu način za obradu datog upita tako da se on izvršava *prihvatljivo* dugo i nastavi da se tako izvršava u određenom, razumnom, vremenskom periodu u budućnosti. [1]

Ovaj rad će se u nastavku baviti različitim tehnikama koje omogućavaju optimizaciju upita kod *Microsoft Sql Server*[[1]](#footnote-1)baze podataka (*MSSQL* u nastavku teksta), pri čemu će svaka od tehnika za optimizaciju upita biti propraćena primerima korišćenjem relativno jednostavne *demo* baze.

# Definisanje parametara optimizacije

U ovom poglavlju biće definisani parametri koji na bitan način utiču na razumevanje određenog upita koji se u datom trenutku posmatra u svrhu optimizacije korišćenjem *MSSQL* baze podataka. Zato će na početku ovog poglavlja prvo biti dat kratak pregled korišćenog modela podataka.

## 2.1 Pregled korišćenih podataka

Baza podataka pod nazivom *Chinook* [2], nad kojom će se vršiti upiti i njihova optimizacija, sadrži podatke koji opisuju prodavnicu digitalnih medija. Baza sadrži tabele koje pamte podatke o umetnicima, albumima, numerama, mušterijama i fakturama. Izgled relacionog modela prikazan je na slici 1 u nastavku.

Slika 1 Relacioni model podatataka

## 2.2 Parametri optimizacije upita

* Definisanje optimizacije:

Definisanje pojma *optimalno* je bitna tačka pre početka same optimizacije i uglavnom se odnosi na pitanja koja mogu biti postavljena u svrhu preispitivanja da li je proces *dovoljno* optimizovan. Uglavnom je bitno dati odgovor, da li dati upit pruža adekvatne performanse uzevši u obzir količinu resursa koji su na raspolaganju kao i količinu novca koju je moguće utrošiti sa ciljem da se upit dalje optimizuje. Bitno razmatranje u ovom koraku tiče se i eventualnog odabira drugog načina za izvršenje datog upita, ili njegovog odbacivanja ukoliko nije neophodan, s obzirom da izvršavanje isuviše loše utiče na performanse.

* Parametri upita koji je predmet optimizacije:

Optimizacija upita na prvom mestu podrazumeva razumevanje njegove svrhe. Nakon toga potrebno je utvrditi koliko često se dati upit izvršava, s obzirom da je njegovo izvršenje jako skupo. Sa druge strane na optimizaciju može uticati i koliko je veliki rezultujući skup koji upit vraća kao i to koliko su velike tabele koje bivaju pretraživane. Takođe neki parametri u upitu mogu imati ograničene vrednosti, pri čemu je možda moguće pojednostaviti pretraživanje izbacivanjem odnosno filtriranjem određenih vrednosti koje nisu bitne.

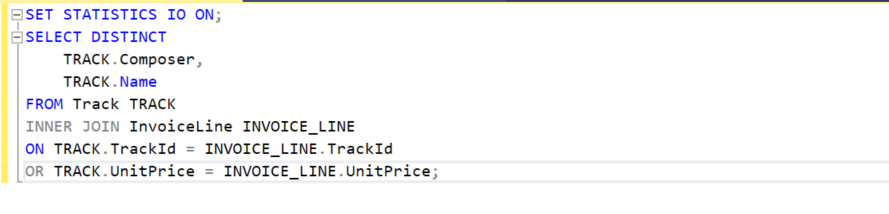
U nastavku će sledeći problemi, pitanja i parametri koji utiču na optimizaciju biti detaljnije obrađeni na primeru upita nad zadatom bazom podataka u *MSSQL*-u.

# Optimizacija upita u MSSQL-u

## Korišćenje *OR* operatora u *Join* predikatu

*OR* u *SQL Servru* predstavlja skupu operaciju s obzirom da ne može biti procesiran u sklopu jedne jedinstvene operacije. Ovaj problem je posledica činjenice da je *OR* inkluzivan operator a to znači da svaka komponenta u *OR* operatoru mora biti zasebno obrađena, pri čemu se nakon kompletiranja zasebnih rezultata, oni spajaju u jedinstveni rezultat koji se vraća nazad. Ovo nije situacija i sa *AND* operatorom koji je za razliku od *OR* operatora ekskluzivan, pri čemu rezultujuće podatke nije potrebno na kraju spajati već je moguće postepeno ih odsacati.

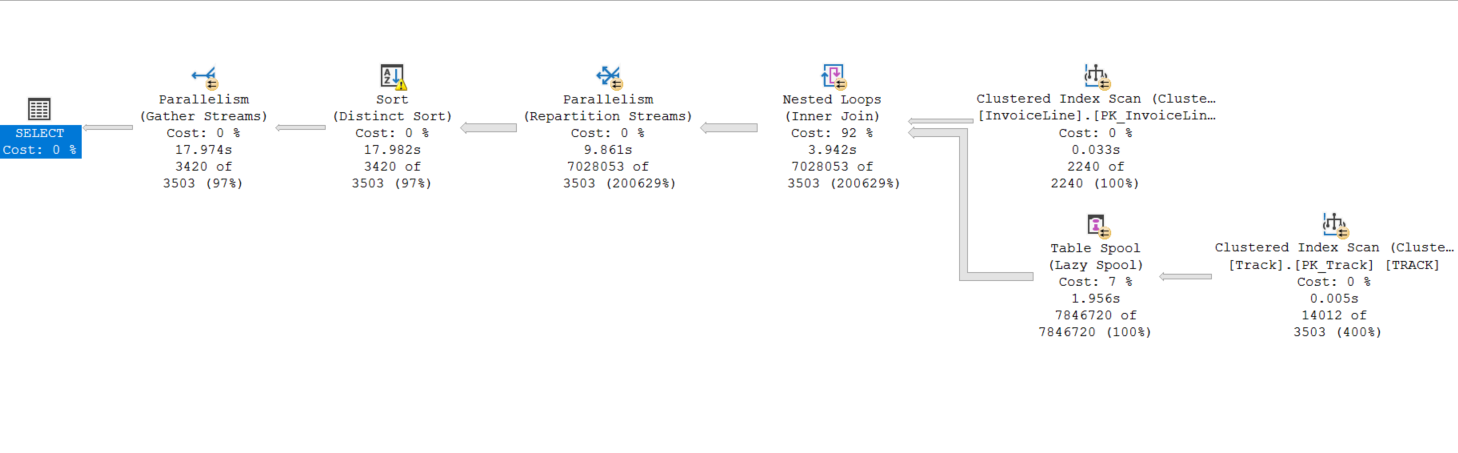
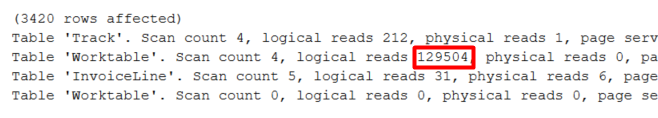
Verovatno najgori slučaj koji može nastati je kada se *OR* koristi prilikom vraćanja podataka iz više različitih tabela ili kolona, s obzirom da je sada potrebno pristupati i različitim tabelama i njihovim kolonama što može postati jako skupo.

Primer ovakvog upita prikazan je na slici u nastavku.

Slika 2 neoptimizovani OR upit u Join predikatu

 Imajući u vidu prethodno definisanu bazu podataka, dati upit ima za cilj da pribavi sve različite numere i njihove kompozitore izvršavanjem *join*-a nad tabelama koje sadrže podatke o fakturama i odgovarajućim numerama. Upit u ovom slučaju koristi *OR* klauzulu koja filtrira rezultate po tome da li se odgovarajući ID-jevi i cene po komadu poklapaju. U nastavku možemo videti i statistiku izvršenja datog upita.

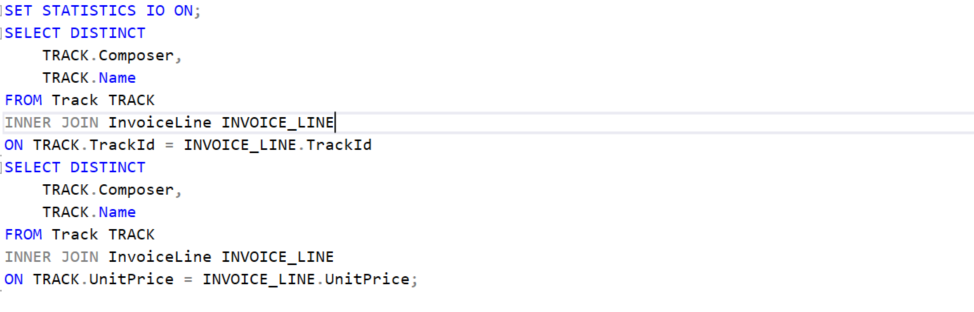
Slika 3 vreme izvršenja OR upita

Na osnovu priloženih statistika izvršenja prethodnog upita odmah je moguće zaključiti da je upit poblematičan, s obzirom da je bilo potrebno čak dvadeset sekundi za njegovo kompletrianje. Sa druge strane ako se pažljivije pogleda statistika prikazana na slici 5, moguće je odmah razumeti zašto je to tako. Oko 130 000 čitanja je obradio *OR* deo upita, pri čemu tabela *Track* poseduje 3503 redova a tabela *InvoiceLine* 2240, tako da je moguće zaključiti da je *OR* deo upita pročitao daleko više redova od onoga što je sadržano u ukupno obe tabele.

Slika 5 Statistika izvršenja OR upita

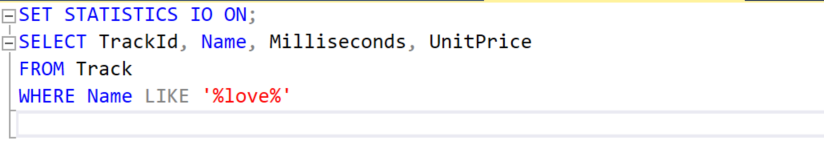
Slika 4 Vreme potrebno za izvršenje upita

Slika 4 Plan izvršenja OR upita

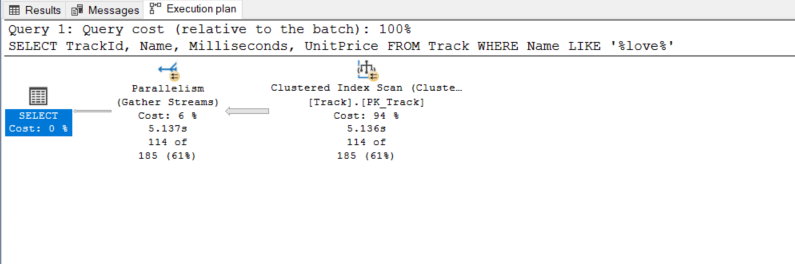
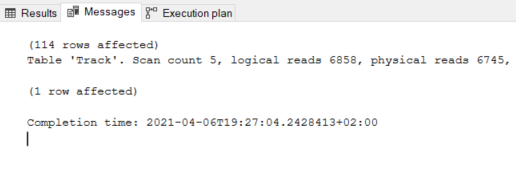
Najbolja opcija za rešenje problema bila bi da se dati upit raščlani na manje, jednostavnije upite, koji se potom mogu spojiti unijom u zajednički rezultat. Prikaz ove optimizacije prikazan je na slici 6. pri čemu se sada upit izvršava za manje od 1 milisekunde:

Slika 6 Optimizacija OR upita

## Korišćenje džoker karaktera na početku i kraju pretraživane fraze

Prilikom pretraživanja tekstualnihpodataka, odnosno stringova, džoker (engl. wildcard) karakteri kreiraju najširu moguću pretragu, pri čemu najšira pretraga uglavnom podrazumeva i najneefikasniju pretragu podataka. Pretraživanje džoker karakterom, koji se nalazi na početku i na kraju pretraživane fraze, zahteva pretraživanje svih zapisa po datom stringu na bilo kojoj poziciji u okviru selektovanog polja. Posledica ovog načina pretraživanja je, takođe, nemogućnost korišćenja prethodno kreiranih rastućih i opadajućih indeksa nad datom tabelom. Primer ovakvog upita i njegovih statistika dat je u nastavku:

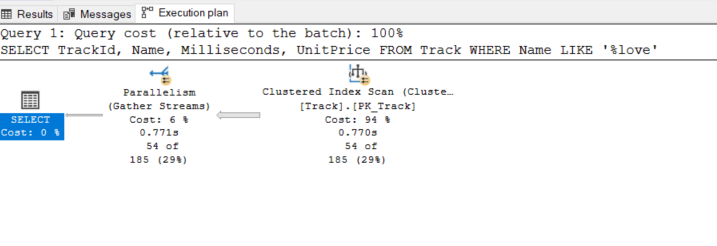
Slika 7 Pretraga korišćenjem džoker karaktera



Slika 8 Plan izvršenja upita

Slika 9 Broj čitanja iz keš bafera koje izaziva dati upit

Broj čitanja prikazan u primeru na slici 9 drastično raste, ukoliko bi se upit izvršava nad ogromnom tabelom, pri čemu u datom primeru tabeli *Track* prethodno dodato oko 650 000 novih zapisa. U ovom slučaju dolazi do drastičnog pada performansi, odnosno dati upit se sada ne može smatrati prihvatljivim, što bi bio slučaj ukoliko tabela ima par hiljada zapisa. Kako broj zapisa u tabeli raste potrebno je razmotiriti neke od predloženih rešenja problema:

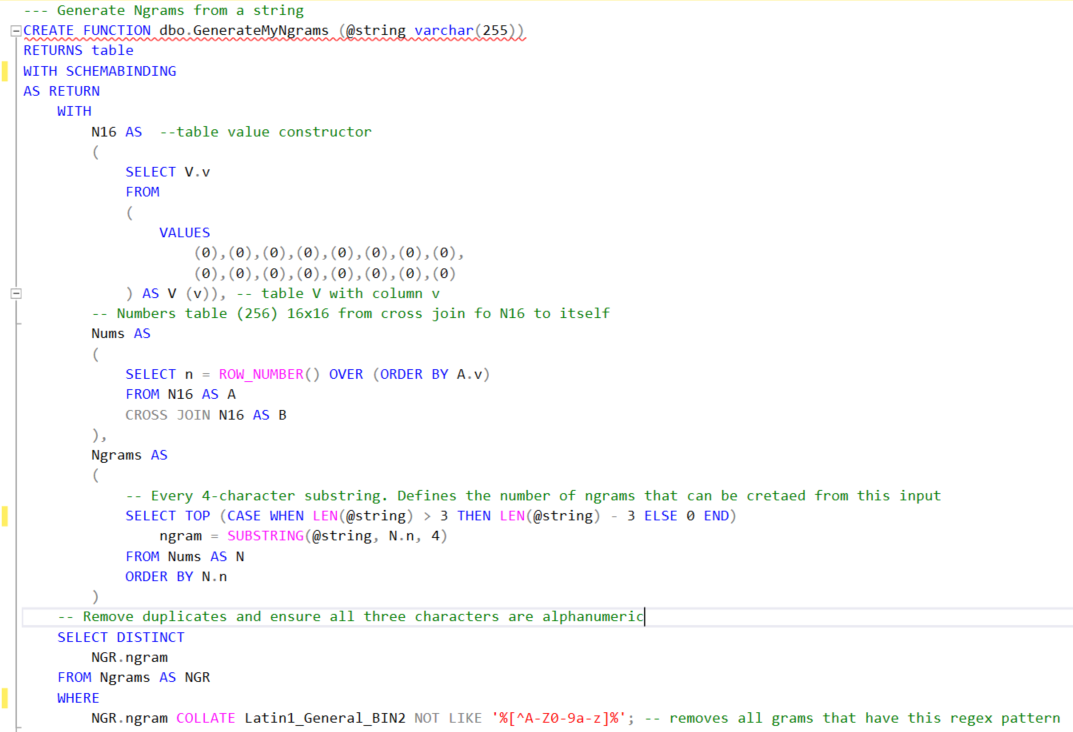
* Izbacivanje ovakvog upita ukoliko zaista nije neophodan
* Korišćenje filtriranja pre primene pretraživanja džoker karakterima sa ciljem smanjenja veličine ulaznog skupa podataka (za manje obime ulaznog skupa ovaj upit može biti prihvatljivih performansi)
* Izbacivanje jednog od data dva džoker karaktera (ukoliko je to moguće) optimizovao bi izvršenje:

Slika 10 Optimizacija pretraživanja izbacivanjem jednog džoker karaktera

Upit u kome se data fraza pretražuje korišćenjem jednog džoker karaktera (‘*%love’*), kompletira svoje izvršenje za 1 sekundu, za razliku od prethodne situacije, kada je bilo potrebno 5 sekindi za izvršenje upita.

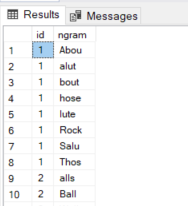
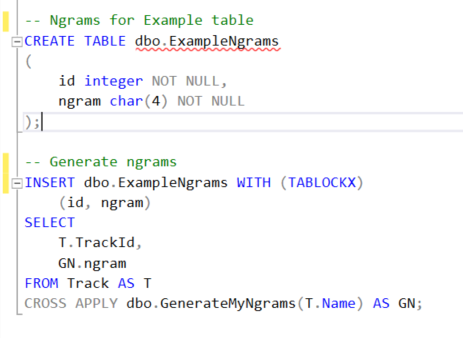
* Pretraga korišćenjem N-gram indeks metode

### Pretraga džoker karakterima korišćenjem N-gram metode

Tehnika pretraživanja stringova pod nazivom *N-gram,* kreira posebnu tabelu koja čuva podstringove poznatije kao *ngram*-ove. Ova metoda je korisna ukoliko se vrši pretraživanje relativno kraćih tekstualnih kolona pri čemu omogućava pretragu podstringova bez da pretražuje celu tabelu. Veličina N u ovoj tehinici indeksiranja odnosi se na veličinu podstringova, odnosno fragmenata koji će biti kreirani za svrhe pretraživanja. Ovu veličinu *N* treba prilagoditi aplikaciji i podacima, pri čemu veća vrednost *N* odgovara preciznijoj pretrazi. [6] U nastavku je dat primer jedne takve implementacije 4-gram-a.

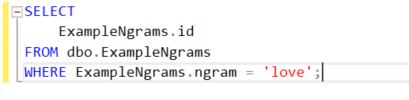
Slika 11 Funkcija za kreiranje 4-gram-a

Funkcija *GenerateMyNgrams (@string varchar(255))* vrši kreiranje četvorokarakternih podstringova ciljnih podataka. Ulazni string može biti maksimalne dužine 256 karaktera. Ova funkcija će za ulazni string prvo proveriti da li je moguće kreirati *ngram*-ove, i ukoliko jeste, kreiraće odgovarajući broj, različitih, četvorokarakternih podstringova od svake reči datog stringa (pri čemu se blanko znaci izbacuju). U svrhu izvršenja ove funkcije koriste se *table value* konstruktori koji omogućavaju da se u okviru jedne *DML* naredbe navede grupa od više redova podataka.

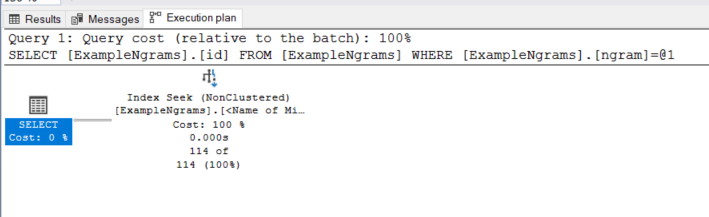
Na slici 12 prikazano je kreiranje tabele koja vrši perzistenciju *ngram*-a. Nakon toga vrši se unos ciljnih podataka koji u ovom slučaju predstavljaju kolone tabele *Track -* *TrackId* i *Name*, pri čemu se *Name* kolona odnosi na ime numere koju želimo da pretražujemo korišćenjem džoker karaktera. Izged tabele nakon kreiranja četvorokarakternih podstringova prikazan je na sledećoj slici:

Slika 12 Kreiranje ngram tabele i insertovanje ciljnih podataka

Slika 13 Četvorokarakterni podstringovi kreirani ngram metodom

Izgled i vreme izvršenja ovako optimizovanog upita:

Slika 14 Ekvivalentni optimizvovani upit za pretraživanje numera koje poseduju u svom imenu dati podstring

Vreme izvršenja ovog upita je sada manje od milisekunde*,* što se može smatrati dobro optimizovanim upitom. Treba napomenuti da je prethodno dodat i neklasterizovani indeks nad kolonom *ngram,* pri čemu će više o indeksima biti rečeno u narednim poglavljima.

Slika 15 Vreme izvršenja optimizovanog upita

Mane ovakvog načina pretraživanja podataka, sa druge strane mogu biti različite. Razumljivo je da svaki put kada se doda novi, obriše ili ažurira postojeći podatak u koloni *Track.Name,* ažurira i tabela koja pamti *ngram-*ove, s tim da treba uzeti u obzir da se ova tabela uvećava nekoliko puta brže od tabele koja čuva originalne podatke. Dakle, bitno je napomenuti da će trošak *Ngram* tehnike pretraživanja stringova uvek biti njeno održavanje. Zato je u nekim situacijama najbolje izbeći ovakvu vrstu upita koji koriste džoker karaktere i na početku i na kraju pretraživanog stringa, ukoliko je to u datoj situaciji moguće.

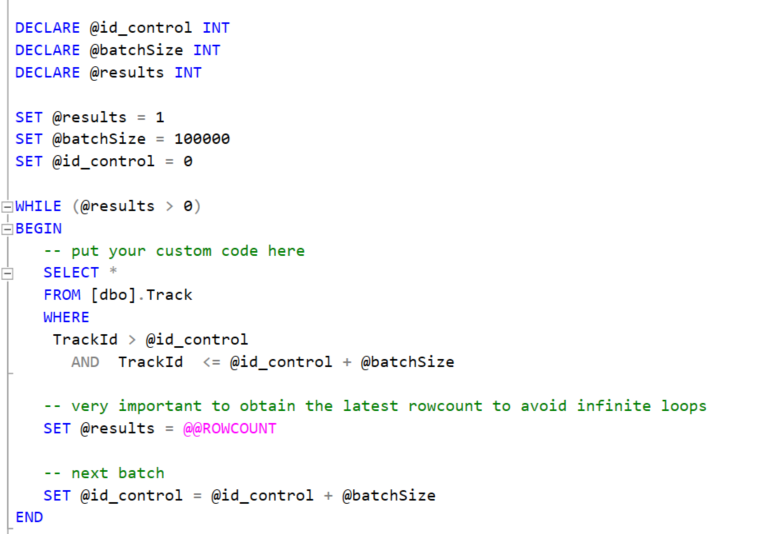
## Ogromni upiti

Operacije koje vrše upisivanje, brisanje ili ažuriranje nad velikim podacima u bazi, mogu dovesti do loših performansi celokupnog sistema, pri čemu može doći do različitih problema poput toga da će korisnici biti prinuđeni da čekaju duže vreme na podatke koje zahtevaju, ili pak da će doći do generisanja velikih log fajlova i popunjavanja dostupnog fizičkog prostora. S jedne strane, zaključavanje i blokiranje predstavljaju mehanizme koji pružaju dobru sigurnost, tako što omogućavaju da podaci u bazi budu konzistentni kada drugi upiti pokušavaju da koriste iste te podatke nad kojima se trenutno izvršava dati upit. Sa druge strane, kada govorimo o velikim operacijama, poput, na primer velikih upisa u bazu podataka, ovi mehanizmi zaključavanja mogu stvoriti probleme u performansama. Ovakve operacije će često zaključati celu tabelu kako bi mogle da unesu ili ažuriraju podatke i uz to provere ograničenja koja postoje nad tim podacima, zatim ažuriraju indekse i pokrenu *trigger*-e.

Broj redova koji biva unešen ili ažuriran može biti veći (oko milion) ili manji (10 000) a da može da se smatra *ogromnim,* zato što izaziva primetan pad performansi, pri čemu ovi brojevi zavise od konkretnih ograničenja definisanim nad datom tabelom i datim podacima. Neke od operacija koje mogu dovesti do ovih situacija mogu biti:

* Unošenje velike količine novih podataka u bazu
* Arhiviranje ili brisanje velike količine starih podataka iz baze
* Dodavanje nove kolone u postojeću tabelu i njeno popunjavanje
* Ažuriranje postojeće kolone
* Menjanje tipa podataka neke kolone

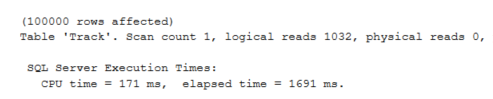
Jedan od načina za rešavanje ovog problema može obuhvatati izvršenje ovih procesa u *batch*-evima (engl. batch - određena, manja količina inicijalnih podataka koju je potrebno obraditi u jednoj iteraciji većeg procesa).

Način za kreiranje upita koji vrši selektovanje svih podataka iz tabele u *batch*-evima prikazan je na slici 16. i obuhvata sledeće stavke:

Slika 16 Primer selektovanja podataka korišćenjem batch-eva

* Na početku je moguće postaviti parametar koji definiše veličinu jednog batch-a, a ovu vrednost je potrebno prilagoditi aplikaciji i konkretnoj bazi, imajući u vidu ograničenja, indekse i trigere definisane nad datom tabelom. Često je ovaj parametar moguće odrediti testiranjem, pri čemu se utvrđuje optimalna veličina koja neće uzrokovati pogoršanje performansi sistema. Sa druge strane ažuriranje tabele u batch-evima smanjuje vreme izvršenja kao i brzinu porasta log fajla. U datoj implementaciji parametar *batchSize* postavljen je na 100 000, s obzirom da tabela *Track* poseduje oko 660 000 zapisa (nakon dodavanja novih podataka).
* U okviru *Select* naredbe može se dodati i *Top* komanda koja ima za cilj ograničavanje broja redova koji će se procesuirati u sklopu jedne iteracije. Ovo je preporučljivo koristiti ukoliko se vrši ažuriranje podataka u sklopu *batch*-eva.
* Parametrar *id\_control* služi za praćenje broja procesuiranih redova nakon izvršenja svake iteracije, čime se osigurava da će svaki podatak biti obrađen i da će biti obrađeno tačno onoliko podataka koliko postoji u tabeli.

Poruka nakon izvršenja upita, za jedan *batch* ispisuje sledeći rezultat:

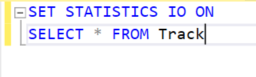


Slika 17 Poruka nakon izvršenja batch upita

Pri čemu je vreme izvršenja ovakvog upita:



Slika 18 Vreme izvršenja batch upita

Gde, za razliku od običnog, neoptmizovanog upita koji selektuje sve podatke iz tabele *Track* vreme izvršenja iznosi:

Slika 19 Upit i vreme izvršenja Select naredbe bez korišćenja batch-eva

Kao što se može primetiti, za velike *DML* procese koji se izvršavaju nad manjim *batch*-evima moguće je ubrzati vreme izvršenja upita. U ovom slučaju to ubrzanje iznosi oko 2 sekunde, uzevši u obzir da tabela *Track* ne poseduje *ogroman* broj podataka (u čijoj situaciji bi ubrzanje bilo još veće). Takođe bitno jeste i to, da ovakav pristup neće generisati ogromne log fajlove.

Jedina mana ovakvog pristupa optimizaciji jeste ta što ključ mora biti sekvencijalni broj i mora postojati makar jedan red u svakoj iteraciji obrade kako se proces ne bi završio pre nego što data operacija bude primenjena nad svim podacima nad kojima je potrebno izvršiti obradu.

## Indeksi i optimizacija

Dodavanje efektivnih indeksa predstavlja jedan od najboljih načina da se poboljšaju performanse izvršenja upita. Korišćenjem indeksa *SQL Server Engine* može izvršiti task u znatno kraćem vremenskom intervalu u poređenju sa situacijom kada indeks nedostaje. Nedostatak indeksa onda dovodi do situacije kada *SQL Server Engine* mora izvršiti skeniranje cele tabele (engl. table scan), pri čemu se vrši pregled svakog zapisa u tabeli. Uzevši u obzir da su tabele nad kojima se izvršavaju upiti velike, *table scan* ima izuzetno loš uticaj na performanse celokupnog sistema.

Sa druge strane, višestruki indeksi mogu imati negativan uticaj na performanse i mogu dodatno opteretiti sistemske resurse. Takođe, dodavanje indeksa zahteva prethodnu analizu opterećenja i SQL upita koji se najčešće izvršavaju nad datom bazom. Zato je najbolje dodati minimalan broj indeksa koji su neophodni, i to za one upite koji predstavljaju usko grlo u performansama, pri čemu se takođe izvršavaju i jako često.

### Indeksi u SQL Serveru

Indeksi u SQL Serveru (i u mnogim drugim bazama) predstavljaju strukturu podataka povezanu sa tabelom ili pogledom koja ubrzava pribavljanje redova iz date tabele ili pogleda, pri čemu sadrže polje ključa koje može biti izgrađeno od jedne ili više kolona.

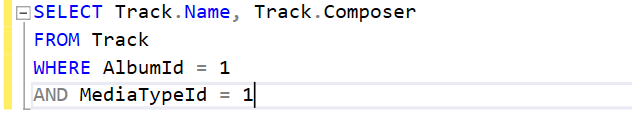
Tabela ili pogled može posedovati sledeći tip indeksa:

* *Clustered* indeksi (klasterizovani indeksi) - fizički sortiraju stranice podataka prema koloni ili kolonama koje su deo klasterizovanog indeksa. Klasterizovani indeks može se kreirati nad jednom ili više kolona, pri čemu može postojati samo jedan klasterizovani indeks po tabeli. Razlog tome je što tabela može imati samo jedan fizički redosled sortiranja na nivou stranice podataka koji se zasniva na klasterisanom indeksu. U najvećem broju slučajeva ovaj indeks se kreira nad primarnim ključem tabele, pri čemu ovo ne mora uvek biti slučaj.
* *Non clustered* indeksi – indeksne stranice su sortirane prema indeksu, pri čemu ovo ne utiče na same podatke i njihov fizički redosled. Neklasterizovani indeks poseduje pokazivače koji upućuju na stranice sa podacima.

*SQL Server Engine* će automatski kreirati *non clustered* indeks kada se kreira tabela sa *UNIQUE* ograničenjem nad kolonom te tabele. Sa druge strane, ukoliko se posatvi *PRIMARY KEY* ograničenje *SQL Server Engine* će kreirati klasterizovani indeks (ukoliko klasterizovani indeks već ne postoji).

### 3.4.2. Query Optimizer u SQL Serveru

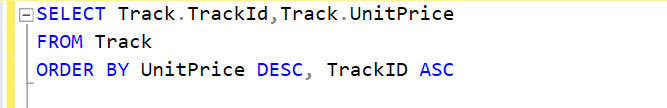
*SQL Server Query Optimizer* predstavlja optimizator zasnovan na izračunavanju troškova izvršenja nekog upita. Ova komponenta analizira određeni broj planova izvršenja nekog upita koji je predmet optimizacije i procenjuje njihove troškove, pri čemu optimizator bira onaj plan sa najnižom cenom od razmatranih kandidata.

Sledeći primer može poslužiti za ilustraciju rada optimizatora.

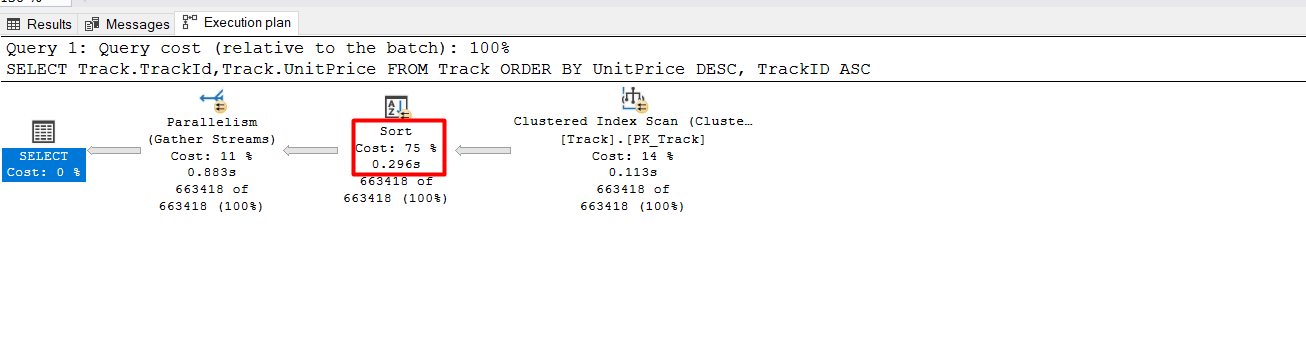
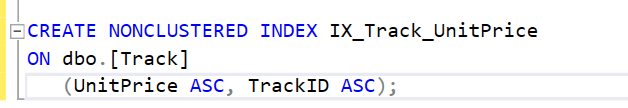
Slika 20 Upit za ilustraciju rada Query Optimizer-a

Uzevši u obzir prethodni primer, *Query Optimizer* će evaluirati više različitih planova izvršenja, dostupnih za izvršenje datog upita, pri čemu će izabrati onaj koji bude evaluirao kao najefikasniji, pri čemu metod za izvršenje upita može uključivati *table scan* ili korišćenje nekog od dostupnih indeksa (ukoliko oni uopšte postoje). Treba imati na umu da indeksi mogu biti od pomoći za različite vrste *Select, Update, Delete* naredbi. Kada optimizator koristi indeks, on pretražuje kolone ključa indeksa, pronalazi lokaciju skladišta redova potrebnih za upit i sa te lokacije izdvaja odgovarajuće redove. Na ovaj način se generiše mnogo manje I/O operacija sa diska u odnosu na *table scan* operacije. Zato optimizacija upita za cilj ima kreiranje indeksa koji su najpogodniji za datu aplikaciju i upite koje ona koristi, tako da optimizator može izabrati najefikasniji metod i skupa za optimalno izvršenje datih upita.

### 3.4.3. Optimizacija upita dodavanjem odgovarajućih indeksa

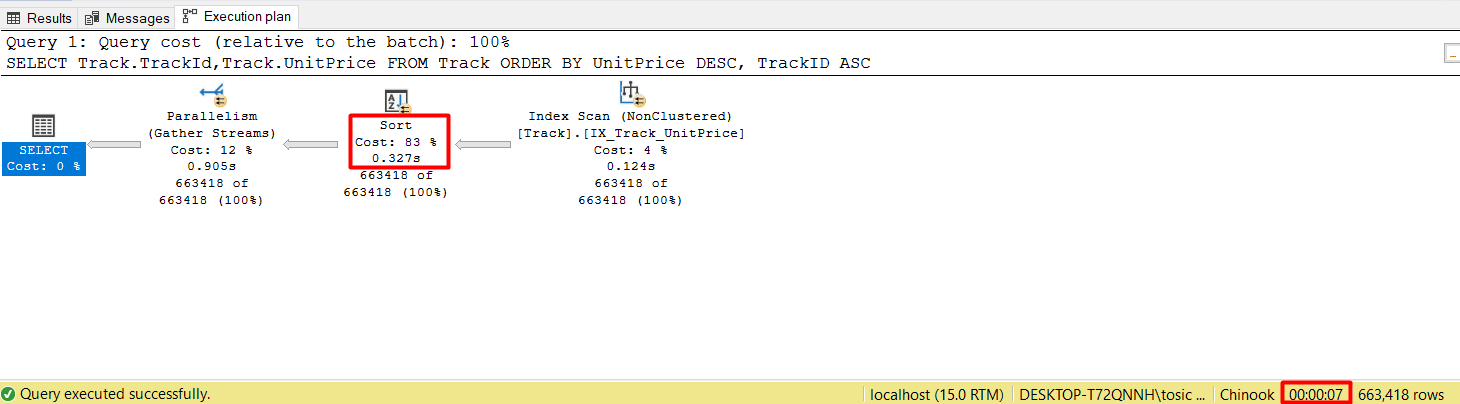
U primeru sa slike 21, dat je upit koji vrši selekciju id-ja i cene numere korišćenjem *ORDER BY* klauzule za sortiranje rezultata u opadajućem redosledu prema ceni i rastućem redosledu prema id-ju, pri čemu je za izvršenje ovog upita potrebno čak 8 sekundi. Upit i plan njegovog izvršenja prikazani su na sledećim slikama u nastavku.

Slika 21. Upit koji koristi sortiranje podataka

Kao što je sa date slike evidentno, dati upit koristi skupu radnju sortiranja sa ukupno 75% udela u ukupnoj ceni izvršenja. Kako bi se smanjilo vreme izvršenja datog upita moguće je dodati novi neklasterizovani indeks nad kolonama *UnitPrice* i *TrackId.* Indeks se može dodati nad datim kolonama na sledeći način, pritom imajući u vidu da po default-u SQL Server sortira podatke u rastućem redosledu u indeksu:

Slika 23 Dodavanje indeksa

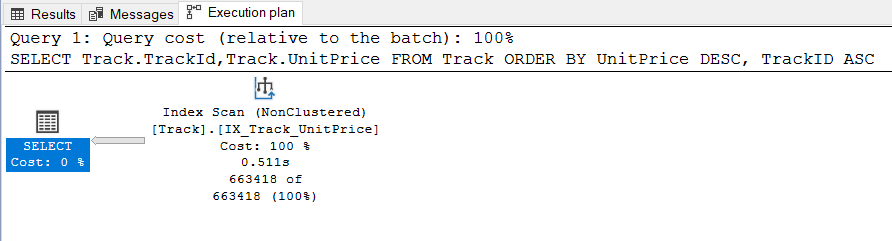
Slika 22 Plan izvršenja prethodnog upita

 Do poboljšanja u izvršenju upita u ovom slučaju neće doći s obzirom da indeks koristi podrazumevani redosled sortiranja za obe kolone u indeksu. Plan izvršenja upita prikazuje sledeće, pri čemu je evidentno da upit i dalje koristi *Sort* operator prilikom svog izvršenja:

Slika 24 Plan izvršenja upita korišćenjem indeksa sa defaultnim načinom sortiranja

Nakon izmene prethodno kreiranog indeksa, tako da sada koristi opadajući redosled sortiranja po koloni *UnitPrice*, plan izvršenja prikazuje da se Sort operator neće koristiti prilikom izvršenja datog upita, već da će *SQL Server Optimizer* za izvršenje iskoristiti indeks koji zadovoljava upit:

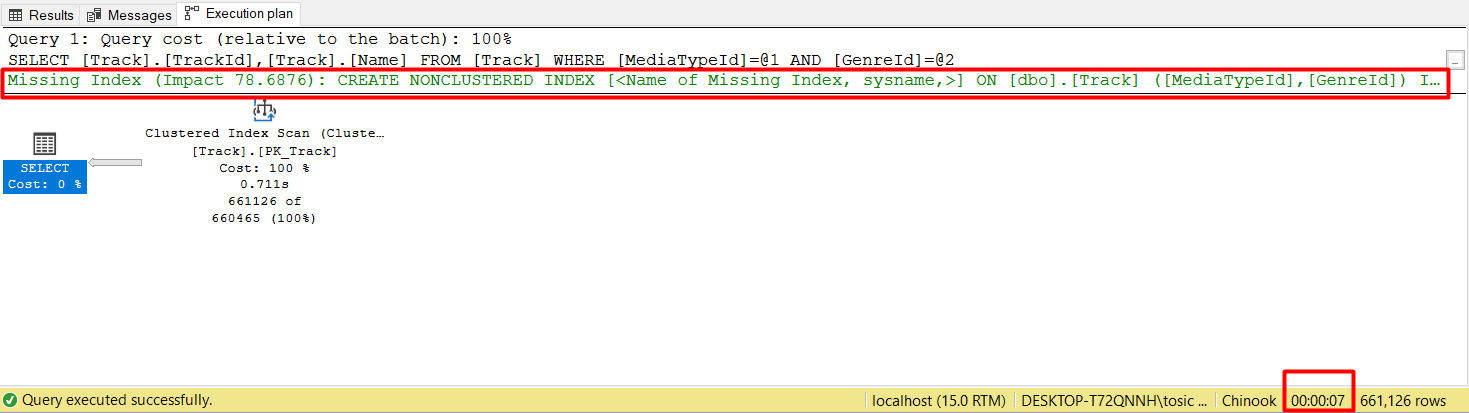
Slika 25 Izvršenje upita korišćenjem novog indeksa

Treba napomenuti da se prilikom izvršenja datog upita korišćenjem indeksa koji vrši sortiranje u odgovarajućim redosledima dobija poboljšanje od samo 14.3%. Zato je prilikom dodavanja indeksa potrebno postaviti sledeća pitanja:

* Koliko ubrzanje se može postići dodavanjem datog indeksa?
* Koliko često će se dati upit izvršavati?
* Da li je dato ubrzanje predstavlja značajno poboljšanje u performansama u datoj aplikaciji?

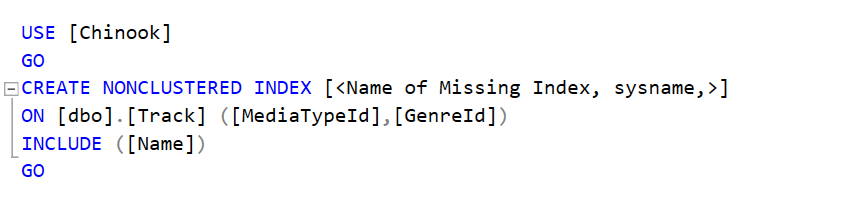
### 3.4.4. Optimizacija upita dodavanjem predloženih indeksa

Slika 26 Upit nad tabelom Track – demonstracija nedostajućih indeksa

Uzevši u obzir prethodno navedeni upit *SQL Server Optimizer* će dati sledeće upozorenje koje može predstavljati jednostavno i potencijalno korisno poboljšanje datog upita, ukoliko bi se dodao nedostajući indeks:

Slika 27 Upozorenje MSSQL Optimizatora za indeksom koji nedostaje u slučaju izvršenja gorenavedenog upita

U obaveštenju koje šalje *SQL Server Optimizer,* navedeno je da poboljšanje dodavanjem indeksa može popraviti performanse upita od 78.7%, što bi moglo da predstavlja značajno poboljšanje performansi u nekoj konkretnoj situaciji korišćenja i ukoliko se zahteva pribavljanje velikog broja podataka iz date tabele. Dodavanje predloženog indeksa prikazano je u nastavku:

Ukoliko je ovo tip upita koji će veoma često biti izvršavan od strane *SQL Server-a,* njegovo dodavanje će predstavljati značajnu optimizaciju. Ukoliko to nije slučaj, treba razmisliti i o negativnim uticajima indeksa na performanse sistema. Jedno od takvih jeste i činjenica da indeksi zahtevaju dodatno održavanje (i dodatni prostor), što nosi za posledicu da će upisi u ovu tabelu biti dodatno usporeni. Ukoliko se kreira previše indeksa nad nekom tabelom, oni mogu imati negativne posledice po performanse sistema i to svaki put kada bi se vršila neka od operacija koja treba da izmeni stanje podataka u datoj tabeli, poput brisanja, ažuriranja ili dodavanja. Sa druge strane posledice definisanja premalo indeksa, koji bi trebalo da omoguće brže izvršavanje drugih upita mogu narušiti druge performanse. Data pitanja zavise od konkretne situacije i opterećenja same aplikacije koja koristi podatke (da li postoji više čitanja ili upisa) i na taj način definišu smer u kome optimizacija treba da se kreće. A na samom kraju verovatno je najbolje naći balans – niti preterati sa korišćenjem indeksa, niti skroz zaboraviti na njih.

Slika 28 Dodavanje predloženog indeksa

# Zaključak

Optimizacija upita u MSSQL-u predstavlja veoma široku temu, sa velikim brojem predloženih rešenja i poboljšanja koja mogu biti primenljiva u različitim problemima optimizacije, a da pritom data poboljšanja uvek zavise od konkretne situacije i same aplikacije, koja koristi i manipuliše podacima u datoj bazi. Kako bi performanse nekog ovakvog sistema bile poboljšane, verovatno je najbolje, za početak, dobro razumeti same podatke, njihovu strukturu, povezanost, zatim njihovu brojnost i ograničenja koja definišu njihovo ponašanje, kao i onu drugu stranu – koja zahteva njihovo korišćenje. Nakon toga moguće je uvideti potencijalni smer u kome bi optimizacija mogla da se kreće, kako bi stvarna poboljšanja bila ostvarena. I na kraju, ovaj rad je imao za cilj da predstavi neke od ključnih tačaka koje je potrebno sagledati na samom početku procesa optimizacije kritičnih upita – procesa s obzirom da optimizacija nije svršena radnja već gotovo uvek ima prostora da napreduje i razvija se, zajedno sa sistemom i njegovim podacima.

# Literatura

[1] Optimizacija upita <https://en.wikipedia.org/wiki/Query_optimization>

[2] Korišćena baza podataka Chinook https://github.com/cwoodruff/ChinookDatabase/blob/master/Scripts/Chinook\_SqlServer.sql

[3] Optimizacione tehnike u Sql serveru 1 https://www.sqlshack.com/query-optimization-techniques-in-sql-server-the-basics/

[4] Dopuna SQL upita za produkcione baze podataka https://www.sisense.com/blog/8-ways-fine-tune-sql-queries-production-databases/

[5] Ngram tehnika <https://social.technet.microsoft.com/wiki/contents/articles/33419.sql-server-implementation-of-n-gram-search-index.aspx>

[6] Trigram Wildcard String Search in SQL Server <https://sqlperformance.com/2017/09/sql-performance/sql-server-trigram-wildcard-search>

[7] Optimizacione tehnike u Sql serveru 2 <https://www.sqlshack.com/query-optimization-techniques-in-sql-server-tips-and-tricks/>

[8] Optimizacija velikih upita u SQL Serveru <https://www.mssqltips.com/sqlservertip/5783/optimize-large-sql-server-dml-processes-by-using-batches/>

[9] Klasterovani i neklasterovani indeksi u SQL Serveru <https://docs.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver15>

[10] SQL Server indeksi i najbolja rešenja za podešavanje performansi https://www.quest.com/community/blogs/b/database-management/posts/11-sql-server-index-best-practices-for-improved-performance-tuning

1. U trenutku pisanja ovog rada najnovija verzija SQL Servera je Microsoft SQL Server 2019, RTM 15.0.2000.5. [↑](#footnote-ref-1)