Logo

Description automatically generatedLogo, company name

Description automatically generatedUNIVERZITET U NIŠU

ELEKTRONSKI FAKULTET

KATEDRA ZA RAČUNARSTVO

**Sistemi za upravljanje bazama podataka – Seminarski rad**

**Interna struktura i organizacija indeksa kod MS SQL Server-a**

Student:

Nemanja Raković, 590

Profesor:

Doc. dr Aleksandar Stanimirović

Sadržaj

[Sadržaj 2](#_Toc102113692)

[1. Uvod 3](#_Toc102113693)

[2. Microsoft SQL Server 4](#_Toc102113694)

[3. Indeksi u SQL Serveru 5](#_Toc102113695)

[3.1 Osnove internog skladištenja podataka 5](#_Toc102113696)

[3.2 Klasterovani indeksi 6](#_Toc102113697)

[3.3 Neklasterovani indeksi 9](#_Toc102113698)

[3.4 Pokrivajući i filtrirani indeksi 11](#_Toc102113699)

[3.5 Fragmentacija i održavanje indeksa 14](#_Toc102113700)

[4. In-Memory OLTP 17](#_Toc102113701)

[4.1 In-Memory OLTP sistemi 17](#_Toc102113702)

[4.2 In-Memory Hash indeksi 17](#_Toc102113703)

[4.2 Bw-Stablo indeksi 20](#_Toc102113704)

[5. Columnstore 22](#_Toc102113705)

[5.1 Sistemi skladišta podataka 22](#_Toc102113706)

[5.2 Columnstore indeksi 22](#_Toc102113707)

[6. Ostali tipovi indeksa 26](#_Toc102113708)

[6.1 XML indeksi 26](#_Toc102113709)

[6.2 Prostorni indeksi 26](#_Toc102113710)

[6.3 Tekstualni indeksi 28](#_Toc102113711)

[7. Literatura 30](#_Toc102113712)

1. Uvod

U današnje doba je prepoznat značaj podataka kao potencijalno najvrednijeg resursa neke organizacije. S obzirom da se količina informacija koja je dostupna eksponencijalno uvećava, potrebno je imati alate koji mogu da na efikasni i korisniku efikasni način upravljaju masivnim, kompleksnim skupovima podataka i iz njih izvuku korisne informacije. U suprotnom, ti isti podaci mogu postati smetnja, s obzirom da cena njihovog skladištenja i održavanja može nadmašiti vrednost koju pružaju.

Gotovo sve organizacije, od lokalnih prodavnica do multinacionalnih korporacija i vlada, poseduju informacioni sistem, u čijoj srži se nalaze baze podataka, i sistemi za njihovo upravljanje. Baze podataka predstavljaju kolekciju informacija vezanih za određeni subjekat, namenu ili pojavu unutar jedne ili više povezanih organizacija. Sistem za upravljanje bazama podataka (*DBMS*) je softver namenjen da pomogne u održavanju i korišćenju velikih količina podataka.

DBMS ima nekoliko ključnih prednosti u odnosu na naivni pristup čuvanja podataka u datotekama i pisanja specifičnog aplikacionog koda za njihovo korišćenje. DBMS garantuje apstraktni pogled na podatke koji je nezavisan od stvarnog mehanizma skladištenja, sofisticirane algoritme i strukture za efikasno skladištenje i pristup podataka, integritet smeštenih podataka, kontrolu pristupa, korisničke interfejse za administraciju, konkurentni pristup bazi, konzistentni oporavak od pada sistema i funkcije koje su zajedničke velikom broju aplikacija koje koriste bazu te time omogućava brži razvoj klijentskog softvera.

Do skoro je većina DBMS rešenja bila relacione prirode. U svom osnovnom radu iz 1970. godine „Relacioni model podataka za velike zajedničke banke podataka“, E. F. Codd je definisao relacioni model zasnovan na matematičkoj teoriji skupova gde se podaci predstavljeni korisniku u vidu tabele i gde svaki red je različit i predstavlja uređenu n-torku relacije R, gde je relacija ništa drugo nego tabela koja se sastoji iz kolona i redova. Vrlo je bitno reći da svaka tabela ima isti skup kolona. Relaciona baza podataka je baza podataka koja čuva podatke u relacijama (tabelama).

Danas su deo tržišta preuzela NoSQL rešenja koja ne koriste relacioni model, u pokušaju da se suoče sa masivnim rastom količine podataka drugačijim pristupima. I pored toga, relacione sistemi su i dalje itekako relevantni jer su najčešće najbliži ljudskom shvatanju odnosa među podacima. Međutim, danas je bitnije nego ikad voditi računa o performansama baze podataka da bi se odgovorilo na izazove koje je doneo Internet, IoT tehnologije i generalno sveprisutnost digitalizacije u današnjem svetu. Performanse baze podataka se ogledaju u brzini obrade upita i vraćanja rezultata korisniku, kao i brzini izvršavanja komandi skladištenja, brisanja, itd.

Jedna od osnovnih tehnika kojom se DBMS-ovi koriste u rešavanju problema performansi su indeksi. Indeksne strukture se prvenstveno koriste da se smanji broj pristupa sporijoj memoriji (disku, u nastavku *I/O pristupi/operacije*). Koriste se prilikom evaluacije i optimizacije upita i kod samog pribavljanja podataka.

U nastavku rada biće prikazana interna struktura i organizacija indeksa kod Microsoft SQL Server rešenja.

2. Microsoft SQL Server

SQL Server je sistem za upravljanje relacionim bazama podataka koji razvija i održava kompanija Microsoft. Jedno je od popularnijih rešenja dostupnih na današnjem tržištu iz ponude relacionih sistema. Primarna funkcija mu je skladištenje i pribavljanje podataka koje koriste druge aplikacije, kako na istoj mašini tako i preko mreže. Komunikacija se obavlja preko Tabular Data Stream (TDS) protokola, čiji se paketi dalje mogu slati raznovrsnim protokolima kao što su TCP/IP, poruke, web servisi, itd.

Microsoft koristi sopstvenu implementaciju glavne celine ANSI SQL 92 standarda pod nazivom Transact-SQL (T-SQL). Takođe ima mnoštvo proširenja specifičnih za SQL Server, uglavnom vezanih za funkcije i proceduralne strukture.

Microsoft u ponudi ima gomilu edicija sa različitim skupovima funkcionalnosti. Pored standardnog interfejsa ka bazi podataka, u ponudi usluga se takođe mogu naći servisi za broker poruka, servisi za mašinsko učenje, replikaciju, generisanje izveštaja (poslovna inteligencija), obaveštenja, integracije sa sistemima za masovno skladištenje, tekstualnu pretragu…

Takođe dolazi u paru sa grafičkim alatom SQL Server Management Studio (SSMS) koji omogućava konfigurisanje, administraciju i upravljanjem svim komponentama SQL Servera kroz grafički i skriptni korisnički interfejs. Postoji i integracija sa Visual Studiom IDE-om. Azure SQL Server je PaaS u sklopu Azure platforme za računarstvo u oblaku, koja je uglavnom kompatibilna sa on-premise verzijom kojoj se u ovom radu posvećuje pažnja.

Verzija sistema koja se koristi u ovom radu je Microsoft SQL Server 2019 Developer Edition.

3. Indeksi u SQL Serveru

3.1 Osnove internog skladištenja podataka

SQL Server skladišti podatke u tabelama i indeksima koje predstavljaju grupu *data* stranica; redovi stranice pripadaju određenom objektu ili entitetu. Stranica je osnovna logička jedinica skladištenja podataka veličine 8 KB i adresira se pomoću identifikatora datoteke u kojoj se nalazi i broja (identifikatora) stranice. Svaki logički slog unutar baze ima jedinstveni identifikator, *rid*, koji omogućava lociranje stranice na disku, osim u jednom specijalnom slučaju vezanom za indekse. Treba napomenuti da skladištenje zasnovano na redovima nije jedini način, skladištenje zasnovano na kolonama i *in-memory* tehnologije će biti obrađene u kasnijim poglavljima.

Stranica se sastoji od zaglavlja veličine 96 bajtova, prostora u kome se skladište redovi podataka i na kraju stranice se nalazi struktura koja predstavlja niz *slotova*. Svaki slot sadrži pomak u bajtovima za red na koji se odnosi; redovi se upisuju u slotove na osnovu pravila sortiranja određenog indeksom; prednost ovog pristupa je da redovi ne moraju da budu fizički sortirani unutar same stranice.

SQL Server logički grupiše stranice u jedinice veličine 64 KB koje se nazivaju *extents*. Postoji nekoliko specijalnih vrsta stranica koje se nazivaju *allocation maps*, i koje prate korišćenje *extent*-ova i stranica unutar određenog fajla. Takođe postoje PFS (*page free space*) stranice koje prate stanje i zauzetost drugih stranica, svaka pokriva oko 64 MB prostora.

Stranice se ne modifikuju direktno na disku, već svaki SQL Server učitava stranicu u radnu memoriju prilikom svakog korišćenja. Stranice se učitavaju u *buffer pool* odnosno memorijski keš. Upisivanje stranica na disk koristi lenji (*lazy*) pristup. Kada je potrebno zameniti stranicu zbog manjka memorije ili ako nema više meta u kešu, SQL Server bira *najdavnije korišćene* stranice. Ako stranice nije identična kopiji na disku (ili je nova ili obrisana), upisuje se na disk dodavanjem u transakcioni log.

Podaci u tabelama nisu sortirani ako se drugačije ne naglasi. Ovakve tabele se zovu *heap* tabele. Stranice za dodavanje u ove tabele se biraju pomoću PFS stranica. Prilikom izvršavanja upita se koristi IAM (*index allocation map*) da pronađe sve potrebne stranice za skeniranje.

Ako na stranici nema dovoljno mesta prilikom izmene nekog reda, SQL Server pomera red na novu stranicu i menja stari red redom koji pokazuje na novu lokaciju (*forwarding* pointer). Ovo sprečava duplo čitanje reda i optimizuje neklasterovane indekse. Tabela se može manuelno regenerisati da bi se sprečile loše performanse prilikom nagomilavanja forwarding pokazivača.

Heap tabele mogu biti korisne prilikom testiranja, npr. da se što brže upiše velika količina podataka u bazu. U realističnim uslovima je teško zamisliti slučaj u kome nije poželjno sortirane podatke korišćenjem klasterovanih indeksa.

Indeksi su strukture podataka čiji elementi sadrže ključ indeksa i slog tabele, ili ključ i logički *rid* sloga. U nastavku poglavlja će biti obrađene implementacije ovih struktura u SQL Serveru.

Svi „regularni“ indeksi u SQL Serveru se zasnivaju na strukturi B-stabla (tehnički je u pitanju B+ stablo, ali će se koristiti termin B-stablo radi jednostavnosti). U pitanju je struktura podataka u vidu stabla pretrage sa proizvoljnim brojem čvorova, koja garantuje balansiranost po pitanju visine. Svi putevi od korena do listova stabla su uvek iste dužine. Prosečan broj dece čvora (*fan-out*) je u praksi bar 100, što znači da je stablo retko dublje od 3-4 nivoa. Listovi stabla sadrže ključeve i same podatke, dok se koren i čvorovi između sastoje od ključeva i pokazivača na niže čvorove. Svaki čvor koji nije list pokazuje na *k – 1* čvorova za *k* ključeva koje sadrži, gde se praćenjem „leve“ putanje od ključa *k* dolazi do vrednosti koje su isključivo manje od *k*. Nivo listova takođe može biti ulančan, i kod SQL Servera je to slučaj, korišćenjem dvostruke lančane liste, gde svaki čvor pokazuje na čvor koji me prethodi i na naredni čvor. Elementi B-stabla u SQL Serveru su stranice, a sami podaci sadržani u čvorovima listovima zavise od tipa indeksa.

U narednim primerima će se koristi sledeća jednostavna tabela za demonstraciju osnova indeksa:

create table dbo.Customers

(

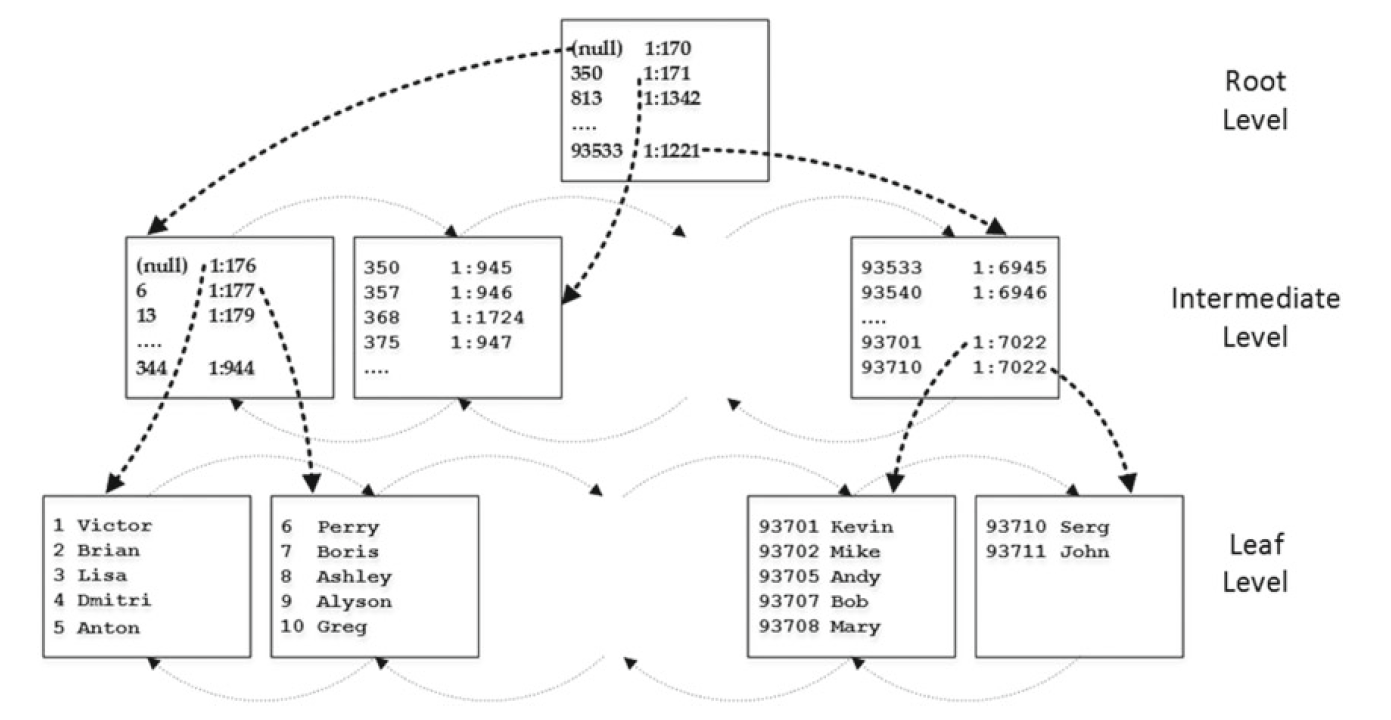
CustomerId int not null,

Name varchar(64) not null,

)

3.2 Klasterovani indeksi

Klasterovani indeksi definišu fizičko uređenje podataka unutar tabele. Slogovi su sortirani u odnosu na ključ indeksa koji se sastoji od jedne ili više kolona tabele. Nad tabelom se može definisati samo jedan klasterovani indeks. Indeks ma strukturu B-stabla čiji listovi sadrže slogove:

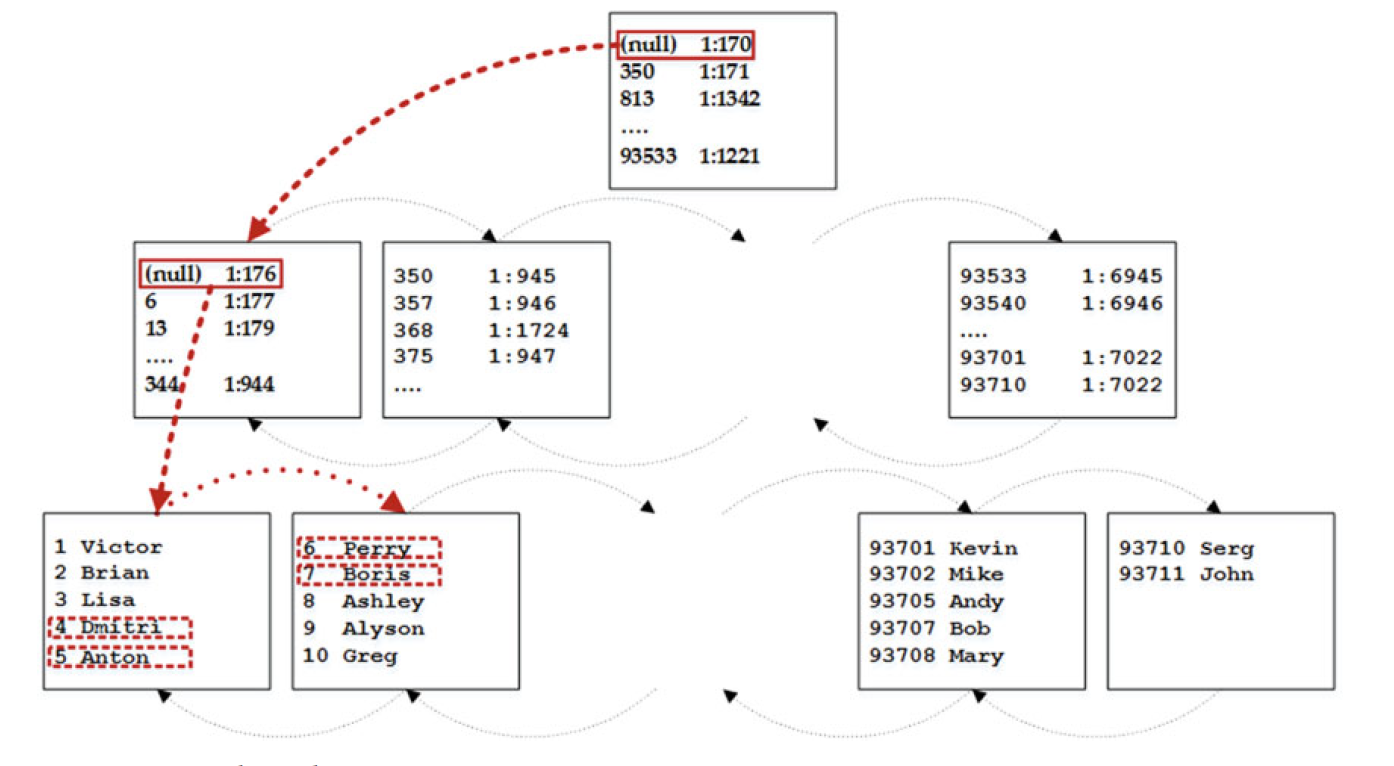


Slika 1 - Klasterovani indeks nad tabelom Customers

Koren i međuslojevi nisu obavezni ako nema dovoljno stranica unutar indeksa da se stablo tako rebalansira. Jedina razlika između korenog čvora i čvora ispod koji nisu listovi je što koren sadrži NULL umesto minimalne vrednosti ključa indeksa. Čvorovi međuslojeva su takođe dvostruko ulančani.

Ako se za primer uzme upit “select Name from dbo.Customers”, SQL Server zna 3 načina za čitanje podataka iz indeksa:

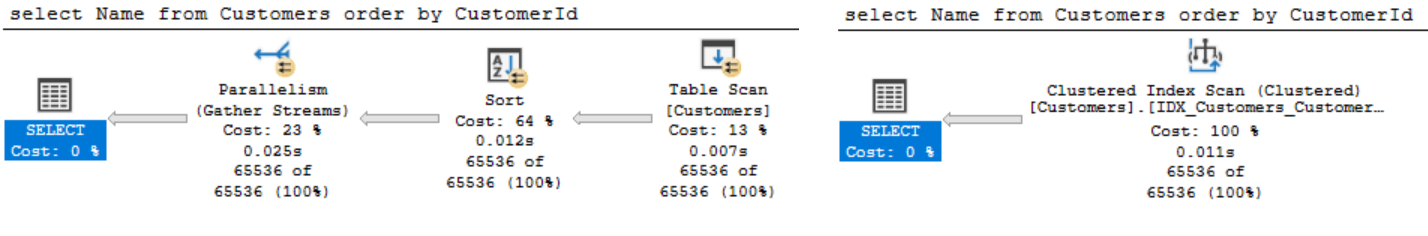
* *Index scan* – SQL Server kreće od korenog čvora i čita prvi red da bi našao dete sa najmanjim vrednostima ključa, i ponavlja taj proces dok ne dođe do kranjeg „levog“ lista. Ceo nivo lista se zatim skenira pomoću dvostruke lančane liste. S obzirom da SQL Server poseduje određene optimizacije, dobro je naglasiti redosled order by CustomerId klauzulom.
* *Allocation order scan* – SQL Server pristupa podacima iz tabele pomoću IAM stranica, kao kod *heap* tabela. SQL Server obično izbegava ovakvo ponašanje, osim u slučaju litanja podataka sa readuncommitted ili serializable nivoa izolacije u transakcijama. U T-SQL se to može postići konstrukcijom with (nolock).
* *Index seek* – SQL Server mora da pronađe stranicu sa određenom vrednošću ključa indeksa. U zavisnosti od specificiranog uslova, može doći do poklapanja sa jednim ili više slogova.
  + *Point-lookup* – Uslovi tipa where CustomerId = 228
  + *Range scan* – Uslovi tipa where CustomerId > 228



Slika 2 - Primer Index Seek upita za vrednost ključa u rasponu [4, 7]

Sledećom naredbom se definiše klasterovani indeks, a zatim slede rezultati pozivanja istog upita pre i posle naredbe kreiranja indeksa:

create clustered index IDX\_Customers\_CustomerId on dbo.Customers(CustomerId)



Slika 3 - Poređenje izvršavanja upita sa i bez indeksa

Gotovo uvek je korisno definisati klasterovani indeks kao *unique*. Budući da neklasterovani indeksi mogu da ukazuju na vrednosti ključa klasterovanog indeksa, problem jedinstvenosti mora biti rešen na neki način. Ako klasterovani indeks nije jedinstven, SQL Server će proširiti redove dodatnom kolonom koja ima jedinstveni redni broj za svaki duplikat, čak i kada nema realnih duplikata.

Još jedna dobra praksa je da vrednosti ključa indeksa budu što manje kako bi se smanjilo vreme obrade i prostor koji zauzimaju stranice u kešu. Takođe, s obzirom da sve reference na ključ indeksa (uglavnom se misli na neklasterovane indekse) moraju biti izmenjene prilikom izmene vrednosti ključa, poželjno je da indeks bude statičan, u smislu da se vrednosti ključa ne menjaju.

Najčešće je dobro koristi klasterovane indekse čija se vrednost ključa kontinualno uvećava. Ovaj pristup smanjuje fragmentaciju indeksa ali potencijalno dovodi do uskog grla ako veliki broj konkurentnih transakcija pokušava da doda novi slog jer se sve bore za istu stranicu. Ovaj problem se može ublažiti korišćenjem OPTIMIZE\_FOR\_SEQUENTIAL\_KEY opcije prilikom kreiranja indeksa.

Iz prethodnih (uglavnom) dobrih praksi da ključ klasterovanog indeksa bude jedinstvena, mala, statična vrednost koja se uvećava za svaki novi red, iz čega sleduje da su *identity* i *sequence* kolone idealni kandidati, osim kada se često javlja veliki broj upisa u isto vreme, kao u primeru iznad.

Sve gorenavedene osobine čine klasterovane indekse idealnim kandidatima za primarni ključ tabele. Ono što se mora uzeti u obzir je da su indeksi fizički koncept dok je primarni ključ logički koncept, i za početak uopšte nije neophodno definisati ga da bi se kreirala tabela. SQL Server podrazumevano kreira klasterovani indeks prilikom definisanja primarnog ključa, ali jedini uslov da indeksna struktura bude primarni ključ je da bude jedinstvena, te se u svrhu primarnog ključa mogu koristiti i neklasterovani indeksi.

Naredba kojom se tabela iz prethodnih primera kreira sa primarnim ključem:

create table dbo.Customers

(

CustomerId int not null identity(1,1),

Name varchar(64) not null,

constraint PK\_Customers

primary key clustered(CustomerId asc)

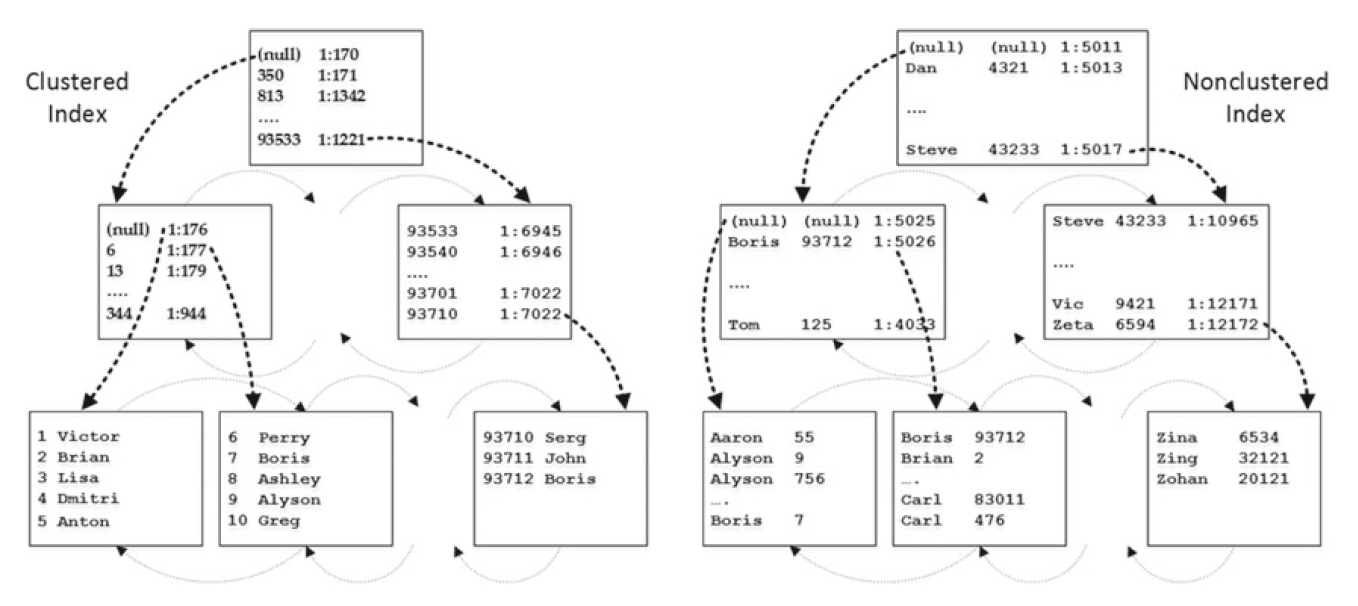
with (OPTIMIZE\_FOR\_SEQUENTIAL\_KEY = on)

)

Postoje izuzeci kada je za klasterovani indeks bolje odabrati određenu kombinaciju kolona sa podacima umesto namenske kolone kao u primeru iznad. To su uglavnom situacije kada se nesrazmerno veliki deo operacija nad tabelom svodi na upit(e) u kojima se sve te kolone poklapaju, što može eliminisati skupe operacije pretraživanja. Takođe, ako se kolone često menjaju, nekad bolje je zameniti skup neklasterovanih indeksa kompozitnim klasterovanim indeksom nad tim kolonama.

3.3 Neklasterovani indeksi

Najbolja analogija za neklasterovane indekse je klasičan indeks pojmova na kraju knjige. Ako posmatramo knjigu kao skladište podataka, brojevi stranica bi predstavljali klasterovani indeks. Sama knjiga je sortirana po brojevima stranica, ali indeks na kraju knjige je sortiran leksikografski u odnosu na reči koje se javljaju u knjizi, i uz svaku reč ima pridružene brojeve stranica na kojima se ta reč javlja. Tim putem se može brzo i efikasno pronaći tražena stranica.



Slika 4 - Neklasterovani indeks za isi primer

Sama struktura neklasterovanih indeksa je jako slična klasterovanim. Suštinska razlika je što listovi B-stabla umesto slogova sadrže *rid*, identifikator sloga putem kojeg se može locirati stranica koja ga sadrži. U zavisnosti od toga da li je sama tabela uskladištena korišćenjem klasterovanog indeksa, postoje 2 moguće vrednosti za format *rid*-a:

* Ako je u pitanju heap tabela, *rid* je u formi *fajl:stranica:slot*. Ovo je i glavni razlog za postojanje *forwarding pointera* pomenutih na početku poglavlja, jer sprečavaju osvežavanje neklasterovanih indeksa u slučaju premeštanja sloga u drugu stranicu. Identifikator u tom slučaju referencira pokazivač.
* Ako je u pitanju tabela sa klasterovanim indeksom, *rid* skladišti vrednost ključa klasterovanog indeksa. Kao i kod heap tabela, ova osobina pomaže u sprečavanju nepotrebnog osvežavanja indeksa.

Na Slici 5 je prikazan primer pretrage za sledeći indeks i upit:

create nonclustered index IDX\_CustomersID\_NCR

on dbo.Customers(CustomerId);

select \* from dbo.Customers where Name = 'Boris'



Slika 5 - Upotreba neklasterovanog indeksa

U situaciji kada indeksi nisu jedinstveni i redovi za iste vrednosti ključa nisu na istim stranicama, SQL Server mora da proveri i naredne stranice dok ne naiđe na nepoklapanje.

Pored operacija na Slici 5, SQL Server će pročitati narednu stranicu i ponovo pristupiti klasterovanom indeksu, ovog puta tražeći krajnju desnu stranicu na slici. Za potrebe ovog upita koji vraća pronalazi 2 sloga je bilo potrebno učitati 10 stranica, odnosno za svaki pronađeni slog je potrebno proći celu dubinu klasterovanog indeksa. Očigledno je da je za iole veći broj pronađenih slogova potreban veliki broj I/O operacija. Dodatna loša okolnost je da iako su koren i međučvorovi najverovatnije u kešu, listovi nisu, što dovodi do nasumičnog čitanja diska za pristup čvorovima klasterovanog indeksa koje je jako sporo zbog fizičke organizacije diskova. U tom slučaju bi indeksno skeniranje klasterovanog indeksa bilo efikasnije zbog mogućnosti blok-čitanja sa diska. SQL Server je zbog ovih nedostataka izrazito konzervativan kada je u pitanju korišćenje neklasterovanih indeksa, tj. bitno je da je selektivnost indeksa veoma visoka, odnosno kada je očekivani broj pronađenih slogova manji.

SQL Server implementira *unique constraint* na prozvoljnoj koloni kreiranjem jedinstvenog neklasterovanog indeksa na toj koloni.

SQL Server ima mogućnost preseka neklasterovanih indeksa, odnosno kada su uslovi upita definisani nad više kolona od kojih svaka ima svoj indeks, SQL Server će izvšiti *join* po *rid*-u da bi kreirao torke koje odgovaraju upitu. Postavlja se pitanje kako odrediti da je efikasnije koristiti presek a kada pretragu na jednom od indeksa. Odgovor uglavnom zavisi od selektivnosti indeksa, tj. visoko selektivni indeksi će i pored velikog broja operacija prilikom pristupanja klasterovanom indeksu ipak biti performantniji. Presek indeksa takođe zahteva više procesorskog vremena i uglavnom su pogodniji za rešenja gde je potrebno skenirati i agregirati veće količine podataka. U opštem slučaju je bolje imati manji broj širih indeksa sa više pokrivenih kolona nego veliki broj uskih indeksa, recimo na pojedinačnim kolonama. Krajnji dizajn skupa indeksa uvek zavisi od prirode podataka i upita koji se nad njima najčešće vrše.

3.4 Pokrivajući i filtrirani indeksi

SQL Server izbegava korišćenje neklasterovanih indeksa kada planer upita predvidi veliki broj pretraga po ključu ili *rid*-u, odnosno kada je selektivnost indeksa niska. Ove operacije obično dovode do velikog broja čitanja, kako iz keša tako i sa diska. Bitno zapažanje je da je pretraga po ključu/identifikatoru glavni problem u ovoj situaciji, odnosno da će i nisko-selektivni indeks biti upotrebljen ako se izbegne pretraga.

Kada se svi podaci koji su upitu potrebni nalaze unutar indeksa, tada imamo *index-only* upite koji su izrazito efikasni. Indeksi koji omogućavaju ovakvo ponašanje za određeni upit se nazivaju *covering* odnosno pokrivajući. Standardni način za omogućavanje ovakvog ponašanja podrazumeva dodavanje kolone koju je potrebno pribaviti na kraj indeksa, iako se ona efektivno ne koristi za pretragu.

SQL Server ima posebno rešenje za efektivnije kreiranje pokrivajućih indeksa – indeksi mogu sadržati vrednosti nekih kolona uz sam ključ, koji ne mora sadržati te iste kolone. Ovi podaci se smeštaju samo u nivou lista i ne utiču na sortiranje indeksa.

U narednom primeru će se tabela Customers modifikovati da pored identifikatora sadrži ime, prezime, rođendan i kratki opis mušterije. Koristiće se klasterovani indeks na koloni CustomerId i kompozitni neklasterovani indeks na kolonama (LastName, FirstName). Biće izvršena dva upitau kojima se pribavljaju prezime, ime i neindeksirana kolona rođendan. U prvom upitu SQL Server sam bira plan izvršenja a u drugom je forsirano korišćenje neklasterovanog kompozitnog indeksa:

create table dbo.Customers

(

CustomerId int not null identity(1,1),

FirstName nvarchar(64) not null,

LastName nvarchar(128) not null,

Birthday datetime not null,

ShortBio char(200) null

)

create unique clustered index IDX\_Customers\_CustomerId

on dbo.Customers(CustomerId)

create nonclustered index IDX\_Customers\_LastName\_FirstName

on dbo.Customers(LastName, FirstName)

-- Queries

select CustomerId, LastName, FirstName, Birthday

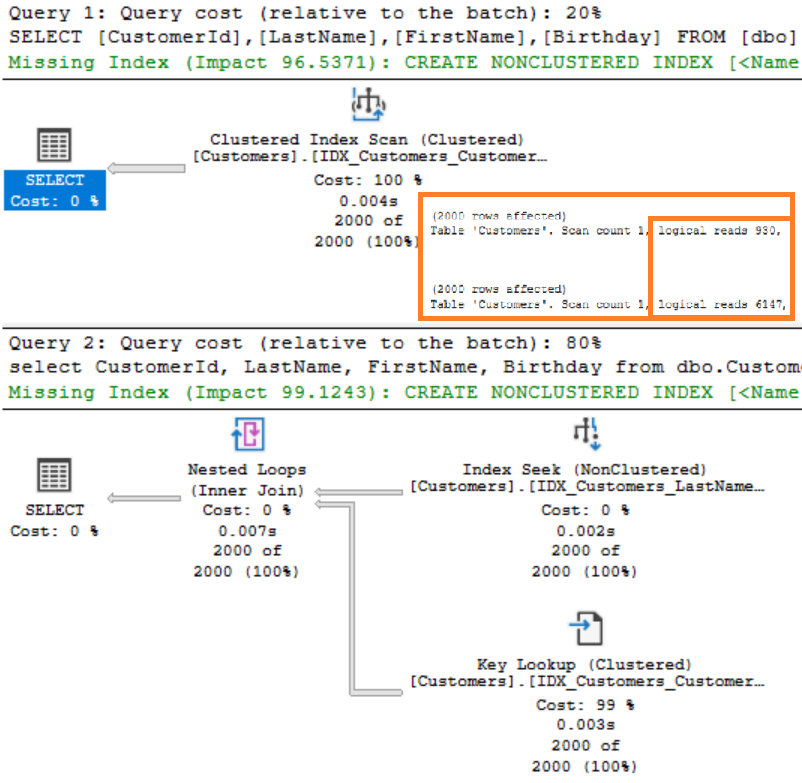
from dbo.Customers

where LastName = 'Anderson'

select CustomerId, LastName, FirstName, Birthday

from dbo.Customers with (Index=IDX\_Customers\_LastName\_FirstName)

where LastName = 'Anderson'



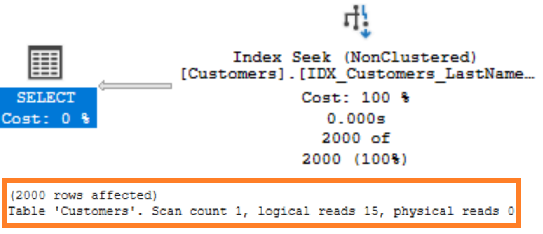
Slika 6 - Upotreba neklasterovanog indeksa bez potpune pokrivenosti

SQL server se u prvom upitu odlučio za korišćenje klasterovanog indeksa, što je bila dobra odluka s obzirom da drugi upit uzrokuje 6.6 puta više logičkih čitanja. Takođe je predložio kreiranje indeksa koji nedostaje. S obzirom da upit selektuje ime i prezime, a identifikator se već nalazi u indeksu u vidu pokazivača na klasterovani indeks, jedina kolona koja nedostaje je rođendan. U sledećoj naredbi je prikazano kreiranje novog neklasterovanog indeksa koji unapređuje prethodni dodavanjem opcije za skladištenje kolone koja nedostaje i time postaje pokrivajući indeks za ovaj upit, čime se automatski kvalifikuje u idealni plan izvršavanja, kao što se vidi na slici:

create nonclustered index IDX\_Customers\_LastName\_FirstName\_BirthdayIncluded

on dbo.Customers(LastName, FirstName)

include(Birthday)



Slika 7 - SQL Server sam bira novi neklasterovani pokrivajući indeks

Još jedna funkcionalnost zajednička za oba tipa indeksa je mogućnost filtriranja. Za primer ispod, polje Internal nema smisla uključivati u sam ključ indeksa zbog niske selektivnosti. Zato filtrirani indeksi omogućavaju da se redovi koji se ne poklapaju uopšte ne indeksiraju:

create nonclustered index IDX\_Customers\_Internal

on dbo.Customers(CustomerId)

include(Internal)

where Internal = 1

Filtrirani indeksi podržavaju samo jednostavne filtere i imaju problema sa keširanjem planova izvršavanja i održavanjem statistika, koje zahtevaju često osvežavanje.

Idealna primena filtriranih indeksa je rešavanje problema jedinstvenih vrednosti kod kolona u koje je dozvoljeno upisivanje null vrednosti, kao na primeru Customers tabele ispod:

create unique index IDX\_Customers\_SSN

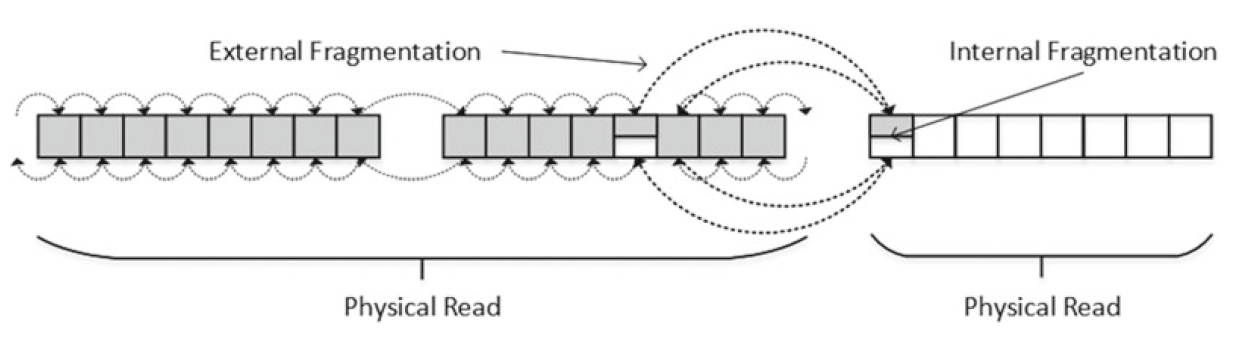
on dbo.Customers(SSN)

where SSN is not null

3.5 Fragmentacija i održavanje indeksa

SQL Server pristupe stranicama u memoriji naziva logičko čitanje (*logical read*) a čitanje sa diska fizičko čitanje (*physical read*). Obe vrste čitanja imaju uticaja na performanse upita. Jedna od optimizacija koje SQL Server primenjuje da smanji broj I/O (fizičkih) čitanja je *read-ahead*, odnosno *prefetching*. Analizom informacija smeštenih u međunivou indeksa se može odrediti da li se stranice na nivou listova nalaze na kontinualnim lokacijama na disku. U tom slučaju se ovom tehnikom može učitati više stranica u memorijski keš i smanjiti broj I/O poziva, za verovatni slučaj da su te stranice potrebne. Ova operacija blok-čitanja sa diska nije mnogo skuplja od čitanja pojedinačne stranice je vreme potrebno za pronalaženje same stranice/bloka uglavnom dominira cenom I/O operacije (*seek time* + *rotational delay*).

Sledeći primer podrazumeva postojanje 2 susedna potpuno popunjena extenta, i situaciju gde je u stranicu unutar jednog od njih potrebno dodati novi red u indeks. S obzirom da je extent pun, potrebno je alocirati novi ekstent i stranicu, premestiti deo podataka u novu stranicu i ulančati novu stranici unutar nivoa listova da bi se održao ispravni redosled čvorova. Ova procedura se naziva podela stranica (*page split*) i ovom primeru dovodi do obe moguće vrste fragmentacije indeksa: interne i eksterne.



Slika 8 - Primer fragmentacije prilikom podele stranice

Eksterna fragmentacija se odnosi na to da se logički redosled stranica razlikuje od fizičkog redosleda na disku. Ovo onemogućava prednosti gorepomenutog blok-čitanja i povećava ukupni broj potrebnih I/O čitanja.

Interna fragmentacija je problem prevelikog praznog prostora unutar stranica. Ova pojava automatski dovodi do postojanja većeg broja stranica što povećava broj logičkih čitanja i zauzeće keš memorije.

Za razliku od eksterne fragmentacija, određeni stepen interne fragmentacije nije uvek loša pojava. Malim stepenom interne fragmentacije se može umanjiti rizik eksterne podele stranica i pojave eksterne fragmentacije prilikom dodavanja i izmene podataka jer stranica može imati dovoljno mesta da obezbedi nove podatke. Kod indeksa kod kojih se vrednost ključa kontinualno uvećava, kao što je pomenuto kod primarnih ključeva, interna fragmentacija specijalni nije poželjna jer se novi podaci uvek dodaju na kraj indeksa.

Jedna od opcija koja se može podesiti na svim indeksima u SQL Serveru je FILLFACTOR. To je broj koji predstavlja ciljanu popunjenost stranice u procentima, podrazumevana vrednost je 100 (ne ostavlja se ciljano prazno mesto u stranicama indeksa). Ovaj faktor se može konfigurisati na nivou individualnih indeksa kao i na nivou celog servera, gde se globalna vrednost koristi ako specifična nije eksplicitno definisana. SQL Server koristi ovu vrednost samo prilikom kreiranja ili rekreiranja indeksa, u toku normalnog rada se stranica svakako može napuniti do kraja i podeliti.

Dobra je praksa konfigurasati faktor blizu maksimalne vrednosti za indekse koji kontinualno rastu. Za ostale slučajeve jedna opšta vrednost koju treba primeniti. Najbolji pristup je gradacijsko smanjivanje vrednosti dok se ne dostigne optimalna vrednost sa najmanjim stepenom kako interne tako i eksterne fragmentacije. U ovu svrhu se može koristiti manuelno ispitivanje pomoću sys.dm\_db\_index\_physical\_stats funkcije kao i praćenje operacija podele stranica pomoću *Extended Events* mehanizma.

Gorepomenuta funkcija kao parametre prihvata bazu, tabelu, indeks, particiju i nivo detaljnosti informacija; korisniku su na raspolaganju detaljne informacije o fizičkom skladištenju indeksa unutar DBMS-a. U narednom primeru se pribavlja neki od najbitnijih kolona:

select

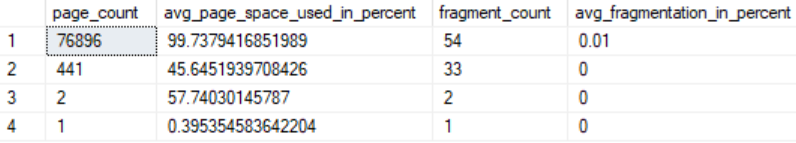
page\_count, avg\_page\_space\_used\_in\_percent,

fragment\_count, avg\_fragmentation\_in\_percent

from sys.dm\_db\_index\_physical\_stats(

db\_id(), object\_id(N'dbo.FactSales'),

1, null, 'DETAILED')

**

Slika 9 - Rezultat upita fizičkih osobina indeksa

SQL Server podržava 2 metode održavanja indeksa:

* Reorganizacija - alter index reorganize
* Rekreiranje - alter index rebuild

Reorganizacija vrši logičko sortiranje stranica na nivou lista i vrši zbijanje podataka unutar stranica da se dostigne nivo definisan faktorom popunjenosti. Izvršava se u realnom vremenu i u slučaju prekida se zadržavaju rezultati operacije. Reorganizacija ne osvežava statistike, što se mora uzeti u obzir ako na automatsko ažuriranje statistika nije omogućeno inače.

Rekreiranje indeksa podrazumeva kreiranje novog indeksa koji će zameniti stari fragmentirani indeks. SQL Server zaključava tabelu tokom trajanja ove operacije. Ova operacija postiže bolje rezultate ali se zato mora obezbediti dovoljno prostora za novi indeks u toku trajanja operacije i mora se predvideti nedostupnost tabele, je u slučaju prekida nema parcijalnih rezultata. Takođe, ova operacija vrši osvežavanje statistika. Enterprise verzija SQL Servera omogućava korišćenje tabele tokom rekreiranja indeksa korišćenjem *row versioning* tehnike.

4. In-Memory OLTP

4.1 In-Memory OLTP sistemi

SQL Server je, kao i većina RDBMS sistema, dizajniran u vreme kad je hardver bio veoma skup, i bilo je moguće raditi samo sa malom količinom memorije u poređenju sa količinom podataka koja bi bila skladištena na disku i predstavljala celu bazu podataka. Sa današnjim hardverom je često moguće smestiti sve aktivne operativne podatke u radnu memoriju, ali i pored toga što će podaci sada uvek biti prisutni u *buffer­*-u, *latch* mehanizmi koje štite integritet strukture stranica u bazi su usko grlo koje se ne može zaobići. Serijalizacija postaje glavni problem: samo jedna jedinica izvršavanja može istovremeno pisati u stranicu.

*Online transaction processing* (OLTP) je tip obrade podataka koji se sastoji od izvršavanja velikog broja transakcija u realnom vremenu sa mnoštvom korisnika. Inženjeri u Microsoft-u su uvideli ove probleme i od nule kreirali novu arhitekturu koja bi donela velika unapređenja performansi, pogotovo u domenima gde je ključno obrađivati transakcije u realnom vremenu. Glavni fokusi u razvoju su bili optimizacija za skladištenje u radnoj memoriji (eliminisanje *buffer* *pool*-a, kreiranje indeksa u memoriji…), eliminacija *latch* i *lock* struktura i kompilacije T-SQL direktno u mašinski kod sistema na kome se izvršava.

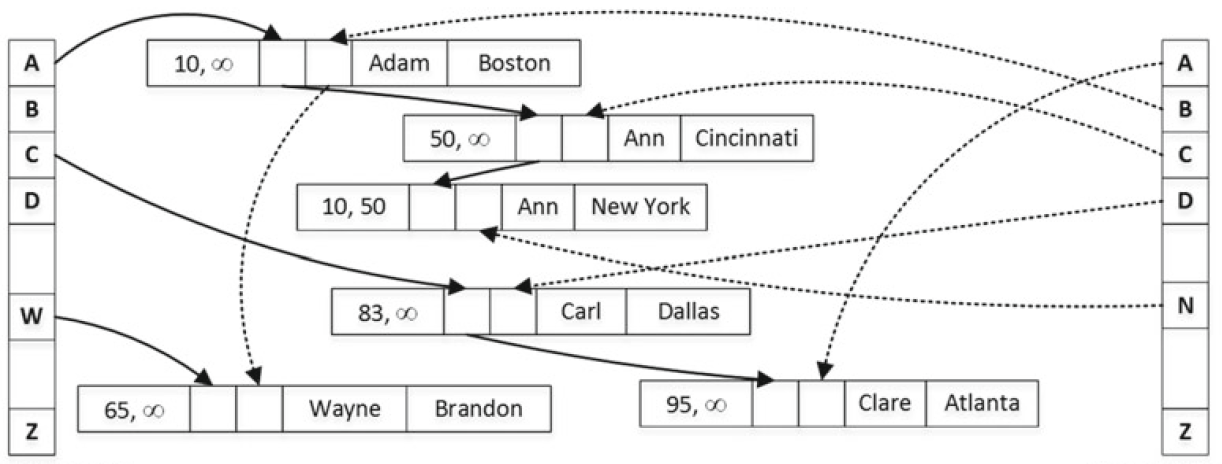
Tabele optimizovane za memoriju (*memory-optimized table*) su nov tip tabela (u daljem tekstu *in-memory* tabele). Kreiraju se regularnom create table naredbom sa tim da se u tom trenutku generiše DLL sa svim neophodnim funkcijama. Takođe je neophodno kreirati sve indekse zajedno sa tabelom. Tabele mogu biti skladištiti podatke ili samo šemu na disk, operacije nad tabelama koje ne skladište podatke se ne upisuju u log transakcija ali su zato ekstremno brze. Struktura ovih tabela se uglavnom ne može menjati, izmena zapravo kreira novu tabelu i kopira podatke.

In-Memory tabele podržavaju 2 glavna tipa indeksa:

1. Hash indeksi (klasterovani i neklasterovani)
2. Bw-stablo indeksi (samo neklasterovani)

4.2 In-Memory Hash indeksi

Hash indeksi nad in-memory tabelama su jedina implementacija hash indeksa u SQL Serveru, s obzirom da hash indeksi generalno nisu primenljivi kada je u pitanju bilo kakva operacija osim upita koji specificiraju sve kolone na koje se primenjuje hash funkcija. Primenom hash funkcije na kolone indeksnog ključa se određuje u koji segment (*bucket*) će slog biti smešten. Svi slogovi koji imaju istu vrednost hash funkcije se smeštaju u isti segment i povezani su lancem pokazivača na svoju lokaciju u in-memory tabeli. Hash funkcija je ista za ceo server i balansirana je, tj. prati Poasonovu ili zvonastu distribucije, nije ravnomerno raspoređena. Realistično je da oko 30% segmenata sadrži duplikate.



Slika 10 - Levo: hash indeks nad poljem Name, desno nad poljem City

Dva glavna faktora određuju dužinu lanca kod kolizije. Prvi je selektivnost indeksa, gde će duplikati negativno uticati performanse. Drugi je broj segmenata u samom indeksu, koji se specificira prilikom kreiranja indeksa. Generalno je dobro definisati broj segmenata da bude 150-200% veći u odnosu na kardinalnost indeksa, tj. broj jedinstvenih vrednosti ključa u indeksu. S obzirom da nije uvek moguće predvideti rast podataka u bazi, postoji mogućnost pozivanja alter table komande koja rekreira strukturu tabele sa novim brojem segmenata.

create table dbo.CustomersOnDisk

(

CustomerId int not null identity(1,1),

FirstName varchar(64),

LastName varchar(64),

Placeholder char(100) null,

constraint PK\_CustomersOnDisk primary key clustered(CustomerId)

);

create nonclustered index IDX\_CustomersOnDisk\_LastName\_FirstName

on dbo.CustomersOnDisk(LastName, FirstName);

create table dbo.CustomersMemoryOptimized

(

CustomerId int not null identity(1,1)

constraint PK\_CustomersMemoryOptimized

primary key nonclustered

hash with (bucket\_count = 30000),

FirstName varchar(64),

LastName varchar(64),

Placeholder char(100) null,

index IDX\_CustomersMemoryOptimized\_LastName\_FirstName

nonclustered hash(LastName, FirstName)

with (bucket\_count = 1024),

) with (memory\_optimized = on, durability = schema\_only)

Nad obema tabelama će biti izvršeni sledeći upiti, čiji su planovi izvršenja prikazani ispod:

select CustomerId, FirstName, LastName

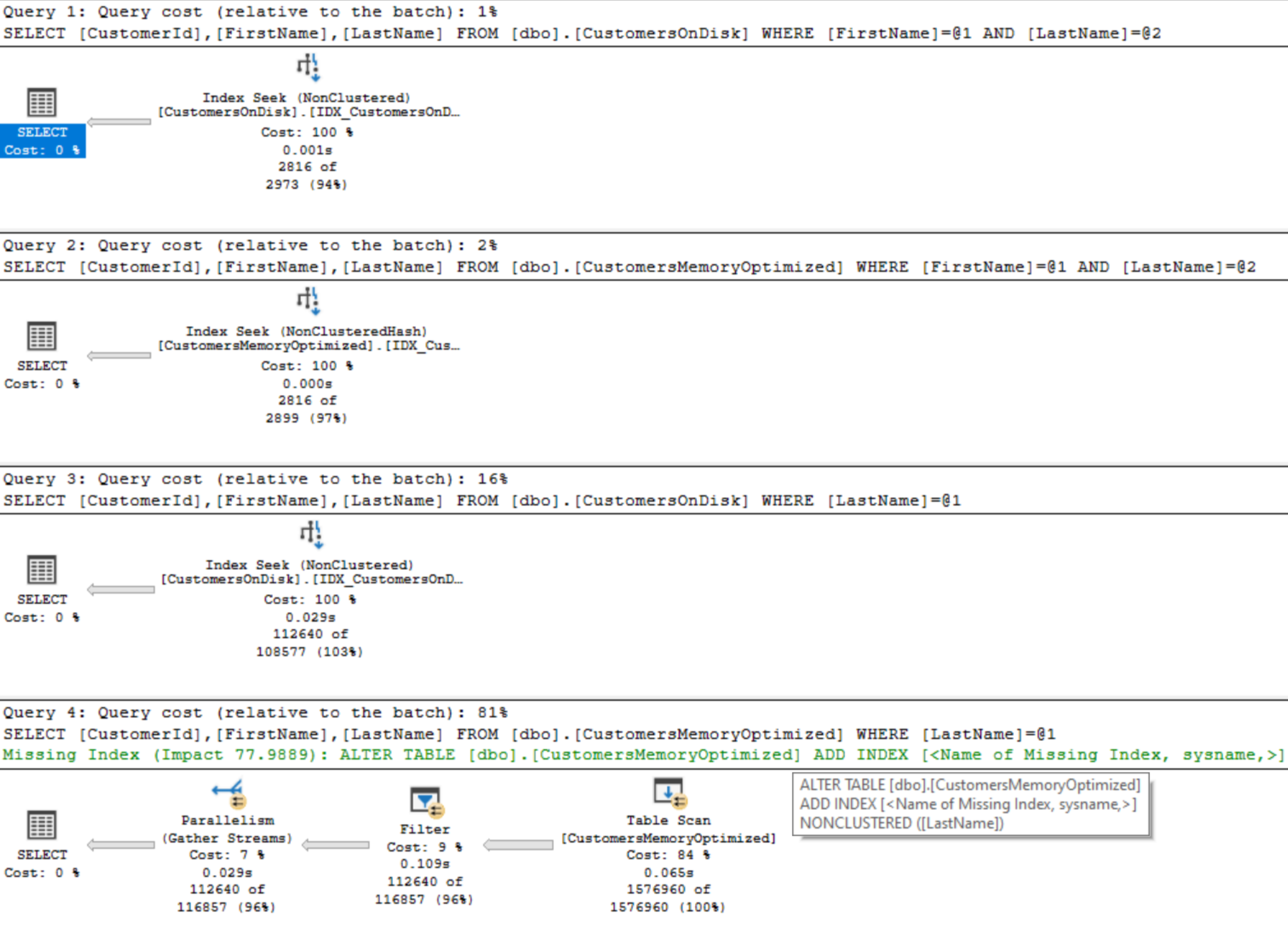
from CustomersOnDisk/CustomersMemoryOptimized

where FirstName = 'Paul' and LastName = 'White';

select CustomerId, FirstName, LastName

from CustomersOnDisk/CustomersMemoryOptimized

where LastName = 'White';



Slika 11 - Poređenje upita koji koristi tabelu na disku i in-memory tabelu sa hash indeksom

Može se primetiti da se upiti koji uključuju jednakost nad oba polja specificirana u hash indeksu izvršavaju veoma brzo. Ali ono što je ključno je da kod druga dva upita, za indeks na disku (upit #3) SQL Server i dalje ima mogućnost da koristi Index Seek mehanizam dok se za poslednji upit mora izvršiti skeniranje cele tabele, koje je na sreću brže jer je u pitanju in-memory struktura. SQL Server je primenio Table Scan s obzirom da bi Index Scan bio jako neefikasan.

Za slučajeve kad ne postoji poklapanje sa svim poljima ključa indeksa ili pretraživanje opsega podataka, neophodno je imati indeks koji se bazira na strukturi B-stabla ili hash sa potpunim poklapanjem. To je ono što SQL Server i preporučuje kod poslednjeg upita, kreiranje neklasterovanog indeksa na polju LastName preko koga je traženo.

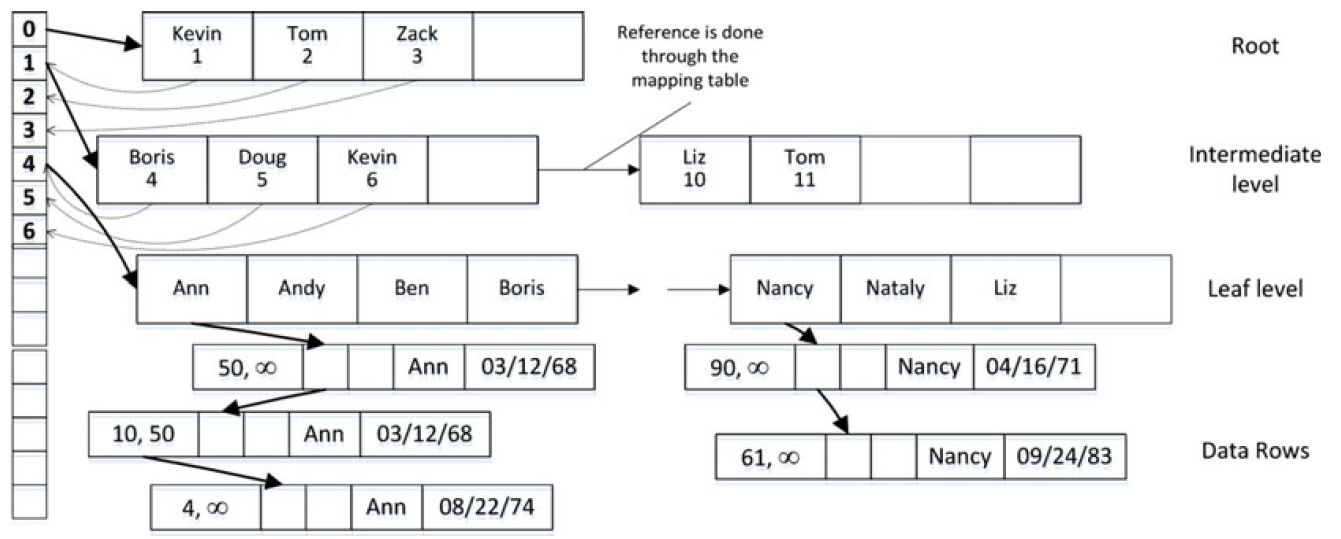
4.2 Bw-Stablo indeksi

Neklasterovani indeksi koji se koriste kod in-memory tabela se takođe nazivaju i *range* indeksi jer rešavaju problem hash indeksa za uslove koji se odnose na opsege podataka. Ovde imamo primere konfuznog imenovanja od strane Microsoft-a s obzirom da su hash indeksi takođe neklasterovani, tj. pojam klasterovanog indeksa nema smisla kada je u pitanju in-memory tabela.

*Bw-stablo* je varijanta B-Stabla dizajnirana osmišljena od strane Microsofta 2011. Stranice indeksa sadrže skup uređenih vrednosti ključa indeksa kao kod regularnih indeksa i maksimalna veličina stranica je takođe 8 KB. Međutim, ove stranice same nemaju fiksnu veličinu i ne mogu da se menjaju nakon što su kreirane.

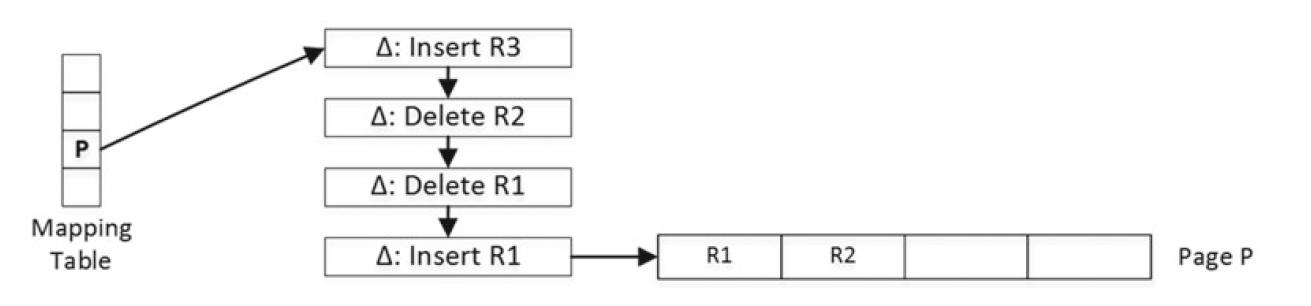
Slogovi iz nivoa lista ovog indeksa sadrže pokazivače na lanac slogova koji se imaju istu vrednost ključa indeksa, slično kao kod hash indeksa gde su više slogova povezani. Svaki indeks u tabeli dodaje pokazivač na niz indeksnih pokazivača za slog, nevezano da li je u pitanju hash ili range indeks.

Koren i međunivoi se nazivaju internim stranicama i kao kod B-stabla upućuju na naredni dublji nivo indeksa, sa tim da umesto da upućuju na samu stranicu oni koriste logički identifikator stranice (PID), koji predstavlja poziciju (pomak) u posebnoj strukturi nalik nizu koja se zove *mapping table*. Svaki element ove strukture sadrži pokazivač na stvarnu indeksnu stranicu.



Slika 12 - Mapping table

Najbitnija konceptualna razlika između B i Bw stabala je u tome što kod B-stabla na disku za svaki slog podataka postoji koji ima istu vrednost ključa indeksa postoji poseban slog u indeksu, dok se kod Bw-stabla čuva samo pokazivač na prvi slog podataka u lancu.



Slika 13 - Ulančavanje delta slogova

Budući da su stranice stabla nepromenljive, kada je potrebno izvršiti neku promenu SQL Server kreira novu stranicu i premešta pokazivač *mapping* tabele na nju. Budući da Bw-stablo nema *lock* i *latch* mogućnosti, uvedeni su *delta* slogovi da bi se ostvario integritet podataka. Delta slogovi predstavljaju promene nad stranicom, INSERT i DELETE naredbe kreiraju po jedan delta slog, a UPDATE dva, po jedan za DELETE i INSERT. Delta slogovi se ulančavaju između mapping tabele i same stranice, i SQL Server koristi *InterlockedCompareExchange* mehanizam da garantuje atomičnost i serijabilnost operacije dodavanja delta slogova. Budući da nakupljanje delta slogova loše utiče na performanse, SQL Server vrši konsolidaciju nakon što njihov broj premaši određeni prag.

Sledeća naredba dodaje indeks koji je predložio SQL Server i ponovo izvršava upit:

-- Add missing index on in-memory table

alter table [dbo].[CustomersMemoryOptimized]

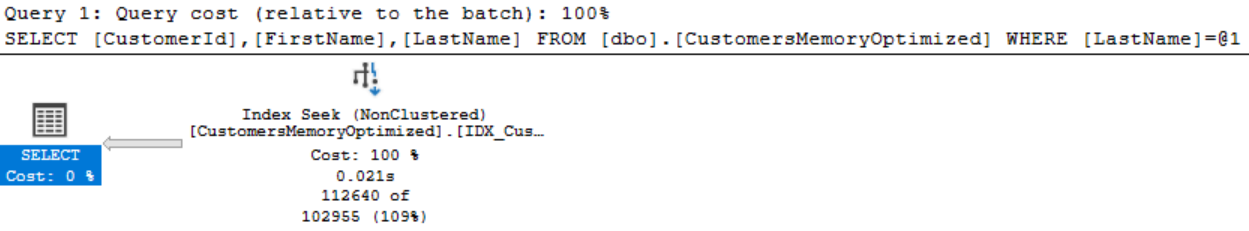
add index IDX\_CustomersMemoryOptimized\_LastName

nonclustered ([LastName])

select CustomerId, FirstName, LastName

from dbo.CustomersMemoryOptimized

where LastName = 'White';



Slika 14 - Plan izvršenja upita sa novim indeksom

5. Columnstore

5.1 Sistemi skladišta podataka

Skladišta podataka (Data Warehouse, DW), takođe OLAP (*online analytical processing*), su sistemi koji se koriste za analizu, izveštavanje i podršku u donošenju odluka, uglavnom unutar velikih organizacija. U kontrastu sa OLTP sistemima pomenutim u prethodnom poglavlju koji su dizajnirani za operativne aktivnosti i procesiranje jednostavnih upita putem kratkih transakcija, DW sistemi se koriste za obradu kompleksnih upita koji obično vrše agregaciju i obrađivanje velikih količina podataka.

Kod njih su upiti čiji je trajanje izvršenja reda veličine sekundi, pa čak i minuta, prihvatljivo; nasuprot nestalnim podacima u OLTP sistemima, ovde je reč o statičnijim podacima namenjenim manjem broju korisnika, i obrada informacija se uglavnom izvršava periodično, uobičajeno noću ili neradnim danima.

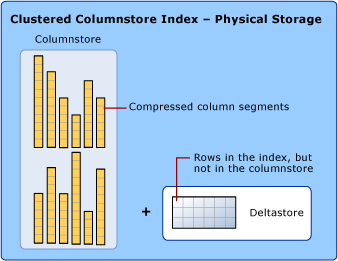
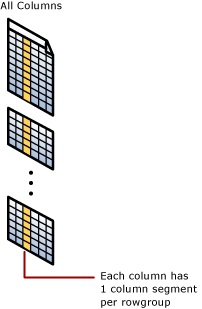
OLTP sistemi uglavnom nisu dobri za generisanje izveštaja; štaviše, oni obično narušavaju operativne performanse sistema. Zato se putem ETL (Extract Transform and Load) procesa podaci učitavaju u DW sisteme u prilagođenom obliku. To su uglavnom nekoliko *tabela činjenica* i veći broj *tabela dimenzija*, koje sadrže neka svojstva vezana za činjenice. Ove tabele činjenica mogu da sadrže milione, pa čak i milijarde slogova, dok su tabele dimenzija znatno manje.

I pored optimizacionih tehnika sličnim onima koje su primenjuju u OLTP sistemima, performanse prestaju da budu zadovoljavajuće nakon određene količine podataka. Jedan način rešavanja ovog problema je korišćenje columnstore indeksa.

5.2 Columnstore indeksi

Tipičan upit DW sistema koristi manji podskup kolona tabele činjenica, vršeći spojeve sa više tabela dimenzija i usput obavljajući dodatne računice i agregacije. U standardnom slogovno-orijentisanom pristupu (rowstore), SQL Serve će pristupati slogovima jedan po jedan, učitavajući ceo slog u memoriju, iako nisu sve kolone neophodne za upit. Columnstore indeksi su idealni kandidati za ovakve upite koji ne pretražuju tabele radi određene vrednosti već vrše analitičke upite, često nad celom tabelom.

Columnstore indeks u SQL Serveru funkcionišu tako što prvo dele tabelu u *grupe slogova (rowgroup)*, koje sadrže najviše 220(1.048.576) slogova. Neophodno je da budu dovoljno velike da podrže stepen kompresije koji ova tehnika pruža. Svaka grupa sadrži po jedan *kolona-segment (column segment)* za svaku kolonu u tabeli, i svaki od njih je kompresovan i smešten na disku. S obzirom da su svi podaci unutar kolona iz istog domena i da je moguće često ponavljanje vrednosti, ova tehnika pruža neverovatnu kompresiju, koja često može da smanji zauzeće na disku i do 10 puta. Takođe se ublažava usko grlo I/O operacija i smanjuje količina memorije koja se koristi, što ostavlja višak memorije za druge upite i operacije.

Slika 15 - Interna organicija columnstore skladištenja

Da bi izbegla fragmentacija kolona-segmenata, često se koristi privremeni klasterovani indeks koji se naziva *deltastore*, kao i B-stablo primarnih ključa obrisanih slogova. Ovaj indeks se kombinuje sa columnstore indeksom prilikom obrade upita. Sastoji se iz više *delta grupa slogova (delta rowgroup)*. Svaki je klasterovani B-stablo indeks koji se koristi da skladišti slogove dok ne dosegnu gorenavedeni limit, nakon čega se od njih kreira nova kompresovana grupa slogova.

Poput rowstore indeksa, columnstore indeks može biti klasterovani ili neklasterovani. Neklasterovani columnstore indeksi sadrže kopiju dela ili celih slogova kojih se nalaze u rowstore skladištu i može sadržati opcioni filter, dok klasterovani indeks služi kao primarni skladišni mehanizam za samu tabelu. Filtrirani indeks interno koristi bitmapu da označi (ne)validne slogove. Moguća je kombinacija rowstore i columnstore pristupa, gde po ceni održavanja oba indeksa korisnik dobija mogućnost da ima brzu pretragu u realnom vremenu i efikasne analitičke upite. Naravno, samo jedan od ovih tipova indeksa može biti klasterovani.

Još jedna velika prednost columnstore indeksa je *batch* *processing*, koje je usko integrisano sa i optimizovano za korišćenje columnstore indeksa. Ova tehnika obrađuje više slogova odjednom te se zato naziva i vektorsko izvršenje.

U kodu koji sledi je prikazan deo inicijalizacije tabela za testiranje columnstore indeksa. Primeri tabela su preuzeti iz „Pro SQL Server Internals“. Na tabeli FactSales koja je skladištena pomoću rowstore klasterovanog indeksa je kreiran i dodatni neklasterovani columnstore indeks. DBMS pri test upitu bira plan koji koristi columnstore indeks, te je zato u prvom izvršavanju upita isforsirano korišćenje klasterovanog indeksa.

create table dbo.DimArticles

(

ArticleId int not null primary key,

ArticleCode nvarchar(32) not null,

ArticleCategory nvarchar(32) not null

);

create table dbo.FactSales

(

DateId int not null

foreign key references dbo.DimDates(DateId),

ArticleId int not null

foreign key references dbo.DimArticles(ArticleId),

BranchId int not null

foreign key references dbo.DimBranches(BranchId),

OrderId int not null,

Quantity decimal(9,3) not null,

UnitPrice money not null,

Amount money not null,

DiscountPcnt decimal (6,3) not null,

DiscountAmt money not null,

TaxAmt money not null,

constraint PK\_FactSales primary key (DateId, ArticleId, BranchId, OrderId)

with (data\_compression = page)

);

-- Create nonclustered index for comparison based on workload (test query)

create nonclustered columnstore index IDX\_FactSales\_ColumnStore

on dbo.FactSales(DateId, ArticleId, BranchId, Quantity, UnitPrice, Amount);

-- Test queries

select a.ArticleCode, sum(s.Amount) as [TotalAmount]

from dbo.FactSales s with (index = 1)

join dbo.DimArticles a on s.ArticleId = a.ArticleId

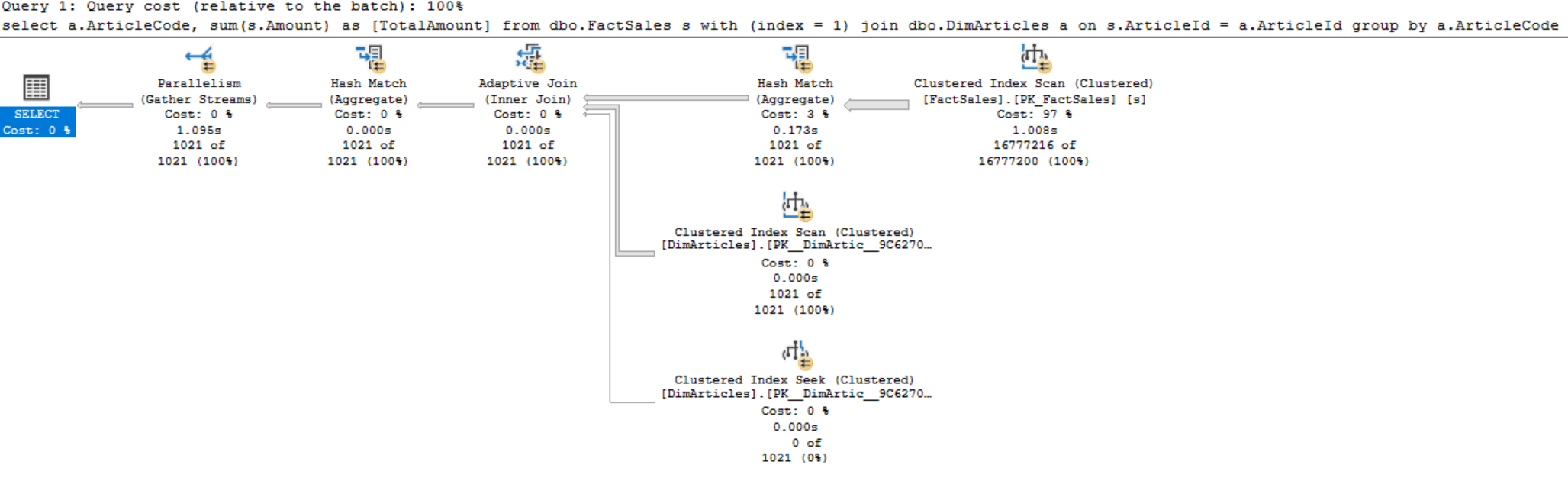
group by a.ArticleCode

select a.ArticleCode, sum(s.Amount) as [TotalAmount]

from dbo.FactSales s

join dbo.DimArticles a on s.ArticleId = a.ArticleId

group by a.ArticleCode



Slika 16 - Prinuđeno korišćenje rowstore indeksa



Slika 17 - Korišćenje columnstore indeksa

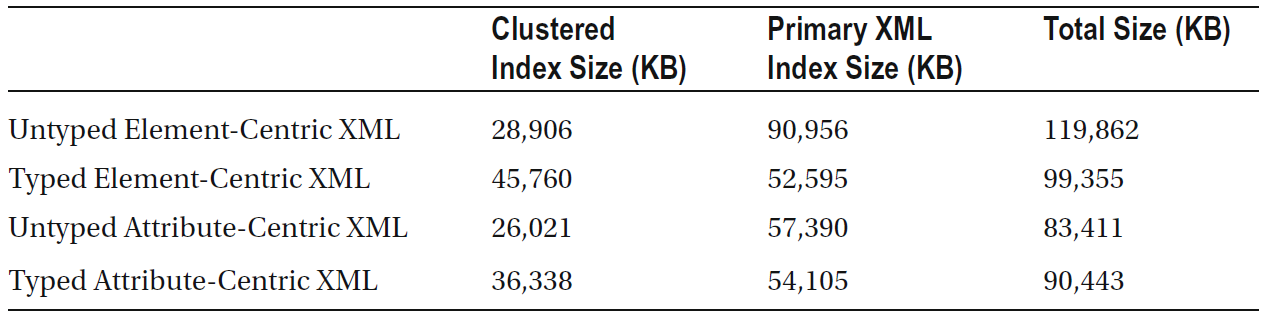
6. Ostali tipovi indeksa

6.1 XML indeksi

SQL Server ima mogućnost skladištenja XML podataka korišćenjem internog formata baziranog na UTF-16 kodiranju sa kompresijom. Originalni XML dokument se ne čuva (nepogodno za slučaj kada je neophodno zadržati formatiranje). XML podaci mogu biti tipizirani ili netipizirani. Netipizirani mogu skladištiti bilo kakve podatke dokle god su u validnom formatu dok tipizirani zahtevaju poklapanje sa šemom koja se može kreirati uz pomoć T-SQL naredbi i dodeliti koloni tabele. Tipizirani podaci mogu unaprediti brzinu upita ali takođe unose dodatnu obradu kod dodavanja ili menjanja podataka te ih treba koristiti promišljeno. Sama XML šema se skladišti u internom formatu unutar sistemskih tabela SQL Servera.

Na XML podacima je moguće kreirati primarne i sekundarne indekse. Primarni XML indeksi transformišu XML podatke u relacioni format i sadrže 1-2 sloga za svaki XML čvor, dok su sekundarni XML indeksi neklasterovani indeksi definisani nad relacionom tabelom koja skladišti primarni indeks. Primarni indeks zahteva da tabela ima klasterovani primarni ključ, nije dovoljno definisati jedinstveni klasterovani indeks ili neklasterovani primarni ključ.

Skladišni prostor koji zauzimaju XML indeksi može varirati u odnosu na to da li su podaci unutar dokumenta primarno smešteni u elementima ili atributima, kao i od toga da li su podaci tipizirani. Na narednoj slici je prikazano poređenje zauzeća prostora u zavisnosti od korišćenog tipa. Sva četiri pristupa se odnose na tabele sa identičnim (semantički) podacima koje sadrže 214 slogova.



Slika 18 - Poređenje XML indeksa

6.2 Prostorni indeksi

SQL Server podržava dva tipa prostornih (*spatial*) tipova podataka: *geometry* i *geography*. Geometry podržava planarne ili Euklidove koordinate koji se odnose na posmatranje površine kao ravne projekcije, dok geography podržava elipsoidalne površine, poput stvarnog izgleda planete. Oba tipa mogu da skladište informacije o lokacijama, poput GPS podataka o geografskoj širini i dužini. Geography nudi nešto veću preciznost ali zahteva i striktnije podatke, koje klijentske aplikacije moraju uzeti u obzir. Sa druge strane, iako često ima bolje performanse za probleme poput preseka površina, geometry tip nije u mogućnosti da izračuna rastojanje između dve tačke zbog ignorisanja zakrivljenosti stvarne površine.

Za primer je kreirana jednostavna tabela koja sadrži primarni ključ i geography kolonu. Uneto je oko 160k slogova koji se odnose na geografske tačke okvirno unutar pravouganika oko teritorije Srbije. Upit koji se koristi za test nalazi identifikatore svih tački koje se nalaze do 5km od Elektronskog Fakulteta u Nišu.

create table dbo.LocationsGeo

(

Id int not null identity(1,1),

Location geography not null,

constraint PK\_LocationsGeo primary key clustered(Id)

);

-- Elfak

declare @g geography;

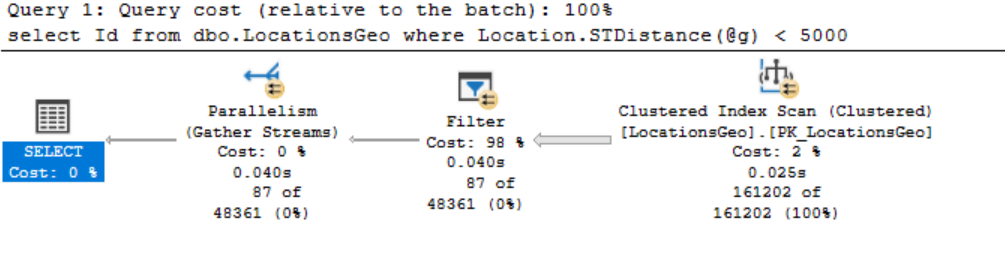
set @g = geography::Point(43.3267515, 21.8933164, 4326);

-- 5km within

select Id

from dbo.LocationsGeo

where Location.STDistance(@g) < 5000;



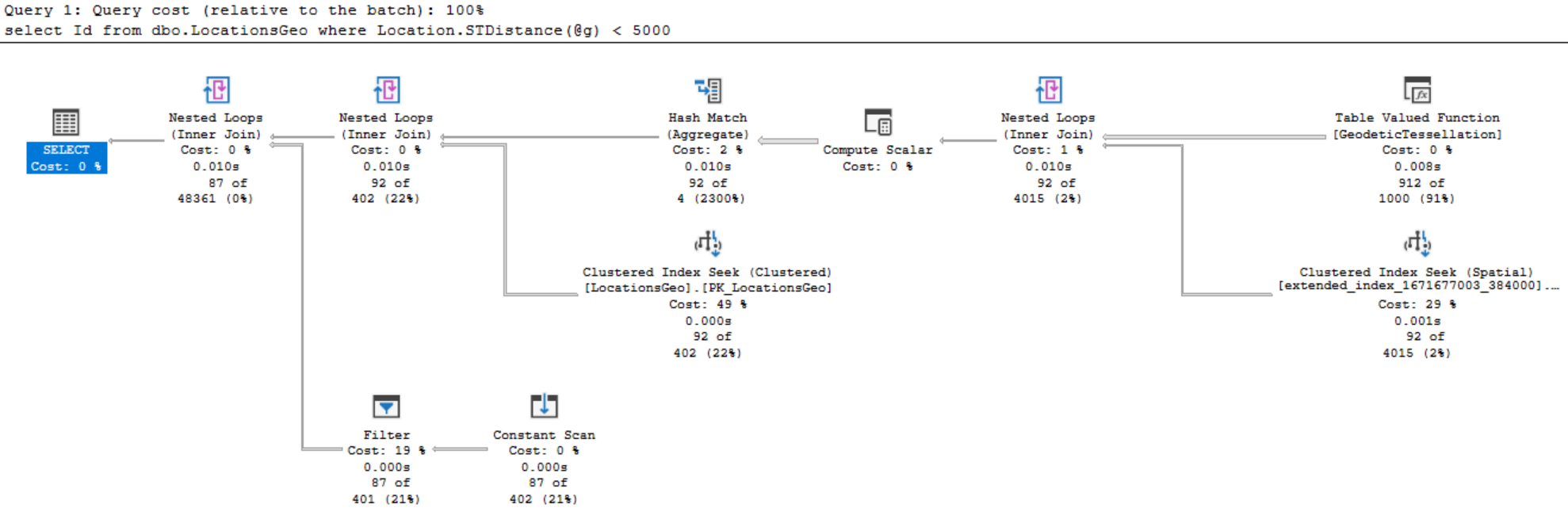
Slika 19 - Pretraga lokacija bez korišćenja indeksa

Ovaj upit se može optimizovati jednostavnim dodavanjem prostornog indeksa na geography kolonu u tabeli sledećom naredbom:

create spatial index Idx\_LocationsGeo\_Spatial

on dbo.LocationsGeo(Location);

Ako nakon dodavanja indeksa ponovimo upit, dobićemo 400% povećanje brzine za uzeti primer.



Slika 20 - Pretraga lokacija sa korišćenjem indeksa

6.3 Tekstualni indeksi

Tekstualni (*full-text*) indeks u SQL Serveru mogu uključivati jedno ili više kolona u tabeli čiji domen mora biti jedan od sledećih tipova: char, varchar, nchar, nvarchar, text, ntext, image, xml, ili varbinary(max) i FILESTREAM; svaka kolona se može odnositi na zasebni jezik. Binarni podaci zahtevaju da im se specifira i tip, npr. PDF.

Tekstualni upiti izvršavaju lingvističku pretragu tekstualnih podataka koristeći reči i fraze koja se zasniva na pravilima jezika određenog jezika. Može se odnositi na jednostavne reči i fraze ili na više formi jezičkih konstrukcija. Tekstualna pretraga kao rezultat podrazumeva svaki dokument koji sadrži bar jedno poklapanje („pogodak“). Poklapanje se odnosi na nalaženje svih zadatih termina i ispunjenje dodatnih uslova kao što je rastojanje između reči.

Tekstualni indeksi se razlikuju po strukturi od regularnih indeksa zasnovanih na B-stablima. Umesto strukture stabla zasnovane na vrednostima unutar sloga, tekstualni mehanizam SQL Servera gradi invertovani kompresovani indeks zasnovan na individualnim tokenima pronađenim unutar teksta koji se indeksira. Veličina samog indeksa je ograničena samo realnim ograničenjima memorije računara. Da bi se kreirao tekstualni indeks, kolona mora da ima definisani regularni jedinstveni indeks.

Bitna stavka za naglasiti je da SQL jezik sam po sebi poseduje delimične mogućnosti tekstualne pretrage korišćenjem LIKE operator. Ono što razlikuje LIKE operator od prave tekstualne pretrage je to što on pretražuje samo na nivou znakova, bez šireg konteksta. Takođe se njime ne mogu pretraživati binarni fajlovi. Na kraju, sa dovoljno velikom količinom nestruktuiranih tekstualnih podatka, LIKE operatoru mogu biti potrebni minuti za upit koji se korišćenjem tekstualnih indeksa izvršava u par sekundi.

Klučne između tekstualnih i regularnih indeksa u SQL Serveru su navedene u sledećoj tabeli:

|  |  |
| --- | --- |
| **Tekstualni** | **Regularni** |
| Dozvoljen je samo jedan indeks po tabeli. | Dozovljeno je više indeksa po tabeli. |
| Populacija tekstualnog indeksa (dodavanje podataka) je može biti zakazana unapred, može se izvršiti na specijalan zahtev ili se može automatski izvršavati prilikom dodavanja novih podataka. | Automatski se osvežavaju svaki put kada se podaci na kojima su zasnovani promene (prilikom dodavanja, izmene ili brisanja). |
| Grupisani su unutar iste baze podataka u jedan ili više tekstualnih kataloga (*full-text catalog*). | Nisu grupisani. |

Kreiranje jedinstvenog indeksa za podršku, tekstualnog kataloga i tekstualnog indeksa čija se populacija momentalno izvršava jer nije naveden uslov CHANGE\_TRACKING OFF, NO POPULATION u okviru naredbe; indeks koristi predefinisanu listu *stop* termina:

CREATE UNIQUE INDEX ui\_us\_Customer

ON dbo.Customer(CustomerId);

CREATE FULLTEXT CATALOG ft AS DEFAULT;

CREATE FULLTEXT INDEX ON dbo.Customer (ShortBio)

KEY INDEX ui\_us\_Customer

WITH STOPLIST = SYSTEM;

7. Literatura

1. Raghu Ramakrishnan and Johannes Gehrke, 2012, Database Management Systems (3rd. ed.), McGraw-Hill, Inc., USA.
2. Dmitri Korotkevich, 2016, Pro SQL Server Internals (2nd ed), Apress, USA,
3. <https://en.wikipedia.org/wiki/Microsoft_SQL_Server>
4. https://en.wikipedia.org/wiki/Data\_warehouse
5. https://docs.microsoft.com/en-us/sql/t-sql/language-reference?view=sql-server-ver15
6. <https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-index-design-guide>
7. <https://docs.microsoft.com/en-us/sql/relational-databases/spatial/spatial-indexes-overview>
8. <https://docs.microsoft.com/en-us/sql/relational-databases/search/full-text-search>
9. https://docs.microsoft.com/en-us/sql/t-sql/statements/create-index-transact-sql