# Osnovni koncepti indeksa u SQL Server bazi podataka

Seminarski rad iz predmeta

*Sistemi za upravljanje bazama podataka*

na master akadademskim studijama


Mentor:

dr Aleksandar Stanimirović

Student:

Arsenije Arsenijević

br. ind. 793


Maj 2020.

# Sadržaj:

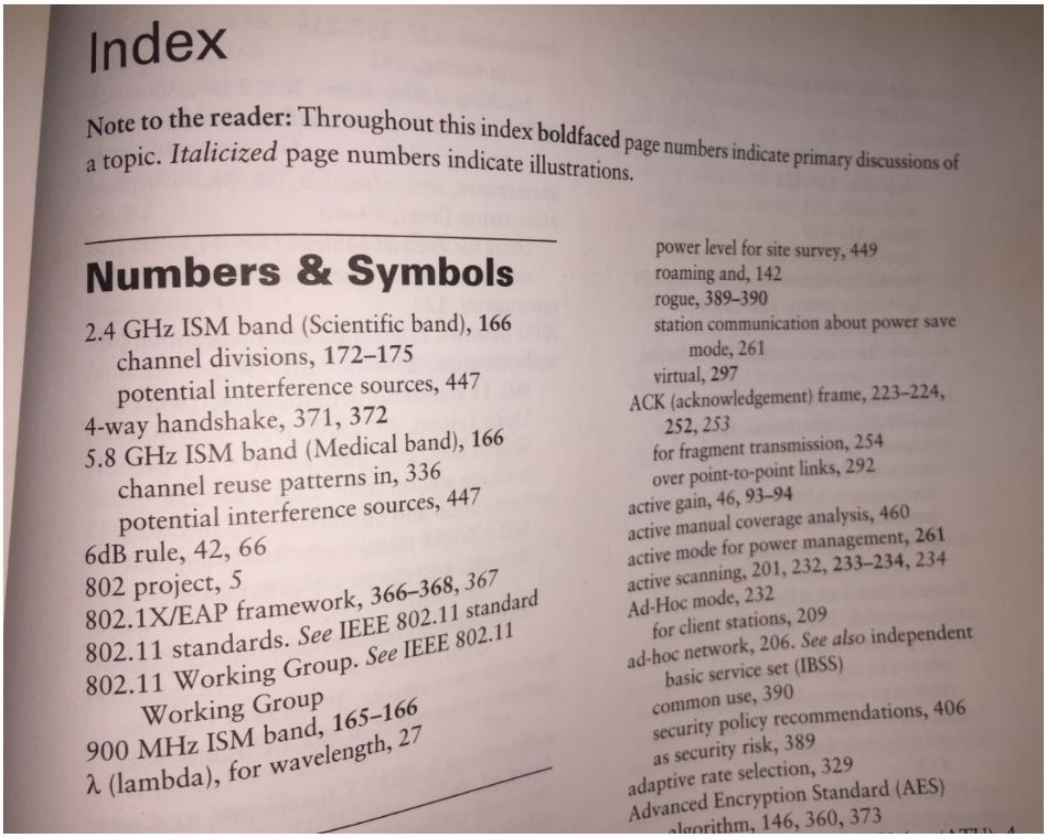
# Uvod

U MS SQL Server bazi podataka, ako ne postoje nikakavi indeksi nad tabelom, rekordi iz tabela se čuvaju u strukturi podataka zvanom gomila (eng. Heap). Kada se pretražuju podaci smešteni u takvoj tabeli, DBMS mora da prođe kroz apsolutno svaki red/rekord po koloni (ili kolonama) na osnovu koje pretražuje podatke. Ovo je izuzetno skupo sa strane sistemskih resursa, posebno kako broj rekoda u tabeli raste jer čitanje svih podataka znači sekvencijalno čitanje kroz disk masovne memorije (jer su velike tabele obično prevelike da budu keširane u RAM memoriji) što definitivno nije dobro za performanse sistema, jer dolazi do grkljanca (eng. Bottleneck). Kada se dodje do takve situacije, neophodno je korišćenje indeksa.

U radu koji sledi, pokušaću da izdvojim najbitnije i najzanimljivije delove ove pozamašne teme.

# Opšte o indeksima

Indeksi u bazama podataka predstavljaju jedan od najvažnijih faktora procesa ,,tuninga'' performansi, a kreirani su u cilju ubrzavanja iščitavanja podataka i operacija procesiranja upita iz tabele ili pogleda (eng. View) baze podataka. Pruža brz pristup redovima tabela, bez potrebe skeniranja (eng. scan) svih podataka u tabeli, u cilju prevlačenja traženih podataka. Indeks u bazi podataka može se poistovetiti sa indeksom kod knjiga, jer omogućava pronalaženje tačno određene teme, informacije u knjizi, umesto iščitavanja cele knjige do stranice gde se nalazi tražena informacija. Primer indeksa knjige može se videti na sledećoj slici:



Index

**Note to the reader:** Throughout this index **boldfaced** page numbers indicate primary discussions of a topic. *Italicized* page numbers indicate illustrations.

**Numbers & Symbols**

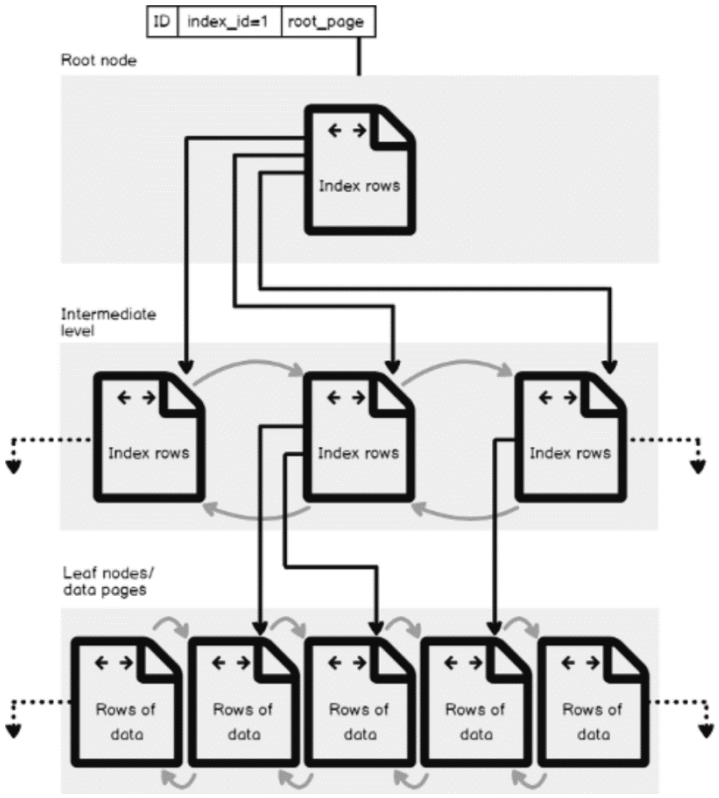
2.4 GHz ISM band (Scientific band), **166**
  channel divisions, 172–175
  potential interference sources, 447
4-way handshake, 371, **372**
5.8 GHz ISM band (Medical band), **166**
  channel reuse patterns in, 336
  potential interference sources, 447
6dB rule, 42, 66
802 project, 5
802.1X/EAP framework, 366–368, *367*
802.11 standards. *See* IEEE 802.11 standard
802.11 Working Group. *See* IEEE 802.11 Working Group
900 MHz ISM band, **165–166**
λ (lambda), for wavelength, 27

  power level for site survey, 449
  roaming and, 142
  rogue, 389–390
  station communication about power save mode, 261
  virtual, **297**
ACK (acknowledgement) frame, 223–224, 252, *253*
  for fragment transmission, 254
  over point-to-point links, 292
active gain, 46, 93–94
active manual coverage analysis, 460
active mode for power management, **261**
active scanning, 201, 232, 233–234, *234*
Ad-Hoc mode, 232
  for client stations, 209
ad-hoc network, 206. *See also* independent basic service set (IBSS)
  common use, 390
  security policy recommendations, 406
  as security risk, 389
adaptive rate selection, 329
Advanced Encryption Standard (AES) algorithm, 146, 360, 373

Pretpostavimo da imamo upit koji treba izvući podatke o zaposlenom iz tabele Employee na osnovu EmployeeID kolone. Bez postojanja indeksa nad kolonom EmployeeID, SQL Server će morati da skenira sve rekorde u tabeli da bi se izvukli potrebni podaci. Ako se kreira indeks nad kolonom EmployeeID tabele Employee, pa zatim odradi pretraga bazirana na vrednosti EmployeeID, SQL Server Engine će tražiti (eng. seek) podatke za traženi EmployeeID u indeksu i iskoristiti indeks da bi pronašao ostatak informacija o zaposlenom koje čuvamo u tabeli, pružajući značajno poboljšanje performansi i smanjivanju napora potrebnog za lociranje traženih podataka, kao što je prikazano na ilustraciji ispod:

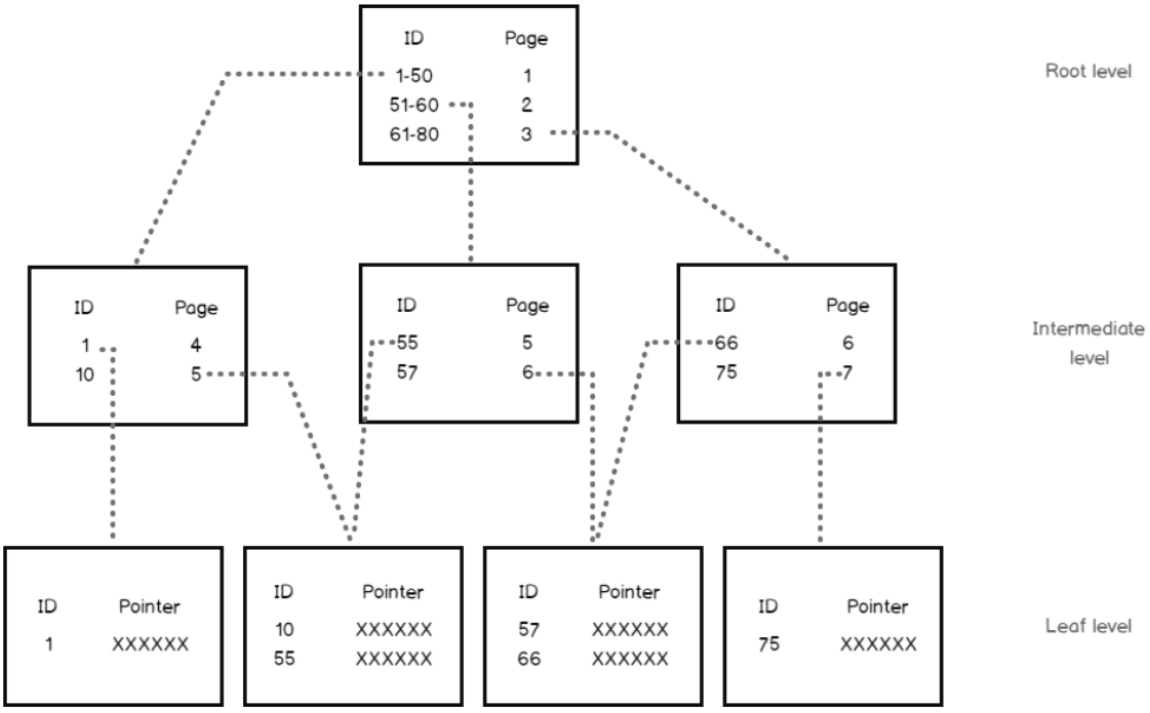

Sposobnosti brzog traženja koje pruža indeks su ostvarene uz pomoć činjenice da je indeks kod SQL Server-a kreiran koristeći oblik B-stabla strukture podataka, koja je sastavljena od stranica (eng. pages) od 8kB, s tim da se svaka stranica u toj strukturi naziva čvor indeksa (eng. index node). Struktura B-stabla pruža SQL Server Engine-u brz način prolaženja kroz redove tabele baziranom ključu indeksa (eng. index key), koji navodi navigaziju levo ili desno kroz grane stabla, da bi se pribavila tražena vrednost direktno, bez skeniranja svih podataka u tabeli. Degradacija performansi je izuzetno velika kada se vrši skeniranje velike tabele u bazi podataka.

Kao što se vidi na slici sa strane, struktura B-stabla indeksa se sastoji od 3 glavna nivoa: **nivo korena** (eng. Root Level), vršni čvor koji sadrži jednu jedinu stranicu indeksa, od koje SQL Server počinje pretragu podataka, zatim **nivo lista** (eng. Leaf Level), najniži nivo čvora koji sadrži stranice sa podacima koje tražimo, sa tim da broj lisova zavisi od količine podataka koja je smeštena u indeksu, i konačno **srednji nivo** (eng. Intermediate Level), jedan ili više nivoa između nivoa korena i listova stabla koji sadrži vrednosti ključeva indeksa i pokazivače (eng. pointer) na sledeće srednje nivoe ili listove sa stranicama podataka. Broj srednjih nivoa zavisi od količine podataka koja se čuva u indeksu.




Pretpostavimo da kreiramo indeks u nekoj tabeli baze podataka nad ID (identifikator) kolonom. Kada se pokrene upit koji traži podatke specifičnog reda u toj tabeli, baziran na vrednosti ID kolone ovih redova, SQL Server Engine će početi navigaciju od čvora korena, tu saznaje stranicu vršnog srednjeg čvora, zatim nastavlja istom logikom kroz niže srednje čvorove, sve dok ne dodje do krajnjeg lista koji sadrži podatke koji su traženi ili pokazivač na taj red u glavnoj tabeli, zavisi od tipa indeksa.

Na primer, ako se pokrene upit koji traži red sa ID kolonom vrednosti jednakoj 57, SQL Server Engine će početi pretragu u čvoru korena, gde saznaje da ID sa vrednošću 57 egzistira u drugom srednjem čvoru. U drugom srednjem, saznaje da je ID vrednosti 57 lociran u list čvoru broj 6, gde je rekord sa vrednošću ID polja 57, ili pokazivač gde taj red može biti nadjen, kao što je ilustrovano ispod:




Indeksi SQL Server-a mogu imati veliki broj čvorova u svakom nivou. Ovo pomaže poboljšanju performansi kreiranja indeksa tako što se izbegava potreba za prevelikom dubinom indeksa. **Dubina indeksa** (eng. Index Depth) je broj nivoa između čvora korena do čvorova lista. Indeks koji je previše dubok će patiti od problema degradacije performansi. Sa druge strane, indeks sa velikim brojem čvorova u svakom nivou može proizvesti previše ravnu strukturu indeksa. Veoma česta pojava je indeks sa samo 3 ili 4 nivoa.

Dodatno dubini indeksa, postoje druge dve vrlo bitne mere indeksa koje kontrolišu efektivnost istog. Prvo je osobina **gustine indeksa** (eng. Index Density) koja predstavlja meru nedostatka unikatnosti podataka u tabeli. Gusta kolona je ona koja ima veliki broj duplikata. Druga osobina je **selektivnost indeksa** (eng. Index Selectivity), koja predstavlja meru broja redova koju je potrebno skenirati prema ukupnom broju redova u tabeli.

SQL Server pruža dva osnovna tipa indeksa, **klasterovani indeks** (eng. Clustered Index) koji čuva originalne redove podataka u listovima indeksa i dodatno kontrolišekriterijum sortiranja podataka u stranicama podataka i sam redosled stranica, baziranom na ključu klasterovanog indeksa. Ovo je razlog zašto je moguće kreirati samo jedan klasterovani indeks nad tabelom. **Neklasterovani indeks** (eng. Non-clustered Index) sadrži samo vrednosti ključa indeksa kolona sa pokazivačem na prave redove podataka čuvane u klasterovanom indeksu ili tabeli, bez ikakve kontrole redosleda podataka unutar stranica i redosled stranica indeksa. SQL Server dozvoljava kreiranje do 999 neklasterovanih indeksa nad svakom tabelom. Tabela bez klasterovanog indeksa se zove tabela gomila (eng. Heap Table), koja nema kriterijum kontrole podataka i redosleda stranica, a tabela koja je sortirana korišćenjem klasterovanog indeksa se naziva klasterovana tabela (eng. Clustered Table). Klasterovani

indeks će biti automatski kreiran kada se definiše primarni ključ tabele, ako već ne postoji predefinisan klasterovani indeks nad tabelom.

Drugi tipovi indeksa u SQL Server-u su **unikatan indeks** (eng. Unique Index) koji primorava unikatnost, jednoznačnost kolone i kreira se automatski kada se definiše ograničenje unikatnosti (eng. Unique Constraint), **kompozitni indeks** (eng. Composite Index) koji sadrži više od jedne kolone i **indeks prekrivanja** (eng. Covering Index) koji sadrži sve podatke koji su potrebni određenom upitu.

Sa svim dobrim stvarima koje indeksi donose, moraju se uzeti u obzir i loše strane, jer indeksi su u nekom smislu mač sa dve oštrice. Dobro dizajniran indeks će poboljšati performanse sistema i ubrzati pretragu i prevlačenje podataka. Sa druge strane, loše dizajniran indeks će proizvesti degradaciju performansi sistema i koštati dodatni prostor na disku i takodje uneti određeno kašnjenje pri unosu/izmeni podataka u tabeli. Uvek je bolje testirati performanse indeksa pre i posle uvođenja indeksa, kako bi se donela odluka o prihvatljivosti indeksa u postojećem sistemu.

[1]

# Gustina i selektivnost

Gustina i selektivnost su neki on bitnijih faktora koji utiču na SQL Server Engine pri odlučivanju kakav plan izvršenja izraditi za određeni upit, odnosno statističke mere koje utiču na optimizer upita (eng. Query Optimizer).

**Selektivnost** meri koja porcija tabele zadovoljava određeni predikatni uslov.

*The fraction of rows from the input set of the predicate that satisfy the predicate. More sophisticated selectivity measures are also used to estimate the number of rows produced by joins, DISTINCT, and other operators* [3].

Objašnjenje razlike između gustine i selektivnosti sledi:

Indeksi imaju **gustinu**, meru jednoznačnosti ulevo-baziranih podsetova kolona u kojima su; predikati imaju **selektivnost**, meru na koju porcije tabele utiču.

Indeksi zapravo ne mogu biti selektivni ili neselektivni; za njih se samo može reći da imaju visok ili nizak nivo gustine. Moguće je da predikat nad kolonom sa veoma malom gustinom (unikatna kolona) ima veoma nizak nivo selektivnosti (velik procenat tabele je podrazumevan). Zamislimo samo predikat ID > 0 nad INTEGER kolonom identiteta (eng. Identity Column). Kolona je unikatna, ali predikat utiče na celu tabelu. Nizak nivo gustine (što je dobro), ali loša selektivnost.

Bitno je objasniti razliku između **traženja** (eng. Seek) i **skeniranja** (eng. Scan). Traženje je operacija koja, uslovno rečeno, ide kroz B-stablo tražeći red ili početak-kraj opsega redova. Skeniranje zahteva predikat i taj predikat mora biti u obliku koji može biti upotrebljen kao argument pretrage.

Traženje je čitanje dela ili svih list čvorova indeksa.

## Indeksi visoke gustine

Šta je tačno problem sa indeksom sa visokim nivoem gustine? Ukratko, vraća previše redova za bilo koji predikatni filter koji se upotrebi „protiv'' njega (osim ako TOP nije u opticaju; ignorisaćemo taj slučaj ovde). Ako indeks ima visoku gustinu, bilo koji predikat koji koristi taj indeks automatski ima lošu selektivnost, jer vraća veliku porciju tabele.

Ako za primer uzmemo tabelu sa 100 000 redova, sa kolonom nazvanom Status koja ima samo 4 vrednosti, onda, pod pretpostavkom da je distribucija tih vrednosti jednaka, upit sad predikatom koji traži jednu od tih vrednosti će pročitati 25 000 redova. Ako imamo neklasterovani indeks nad tom INTEGER kolonom, ispada da neklasterovani indeks ima 225 stranice na nivou lista i da je dubok 2 nivoa. Obzirom da četiri vrednosti imaju jednaku distribuciju, traženje po indeksu za vraćanje redova jedne od 4 vrednosti će zahtevati otprilike 60 čitanja stranica.

Da li je skeniranje indeksa bolje?

Skeniranje indeksa će pročitati sve stranice listove, to je ono što skeniranje radi (ne uzimajući u obzir slučajeve kao MIN, MAX i TOP gde je potrebno skenirati i pročitati samo deo indeksa). Tako da ako SQL odluči da koristi skeniranje indeksa zbog velike gustine indeksa moraće da pročita svih svih 100 000 redova iz svih 225 stranica (plus stranicu korena indeksa).

Traženje po indeksu od 60 stranica protivu skeniranja indeksa po 226 stranica. Izgleda prilično očigledno šta je bolje.

Ogled:

```
CREATE TABLE TestingIndexSeeks (
   Status INT,
   Filler CHAR(795) DEFAULT ''
);

INSERT INTO TestingIndexSeeks (Status)
SELECT NTILE(4) OVER (ORDER BY (SELECT 1)) AS Status FROM (
    SELECT TOP (100000) 1 AS Number FROM sys.columns a CROSS JOIN sys.columns b
) sub

CREATE NONCLUSTERED INDEX idx_Testing_Status ON dbo.TestingIndexSeeks (Status)
```

Set upita koji vidimo iznad stvara upravo situaciju upisanu u prethodnom primeru. 100 000 INTEGER vrednosti upisano u *TestingIndexSeeks* tabelu. Filler CHAR polje služi samo kao pomeraj kako bi svaki red zauzimao veći prostor u memoriji; inače bi 100 000 redova stalo u mnogo manje memorije i ogled bi bio neadekvatan.

```
SELECT status FROM dbo.TestingIndexSeeks WITH (FORCESEEK) WHERE Status = 3

SELECT status FROM dbo.TestingIndexSeeks WITH (FORCESCAN) WHERE Status = 3
```

Statistika poslednjih SELECT upita:

***Seek (traženje):***

```
Table 'TestingIndexSeeks'. Scan count 1, logical reads 60, physical reads 0,
page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob
logical reads 0, lob physical reads 0, lob page server reads 0, lob read-ahead
reads 0, lob page server read-ahead reads 0.
```

***Scan (skeniranje):***

```
Table 'TestingIndexSeeks'. Scan count 1, logical reads 226, physical reads 0,
page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob
logical reads 0, lob physical reads 0, lob page server reads 0, lob read-ahead
reads 0, lob page server read-ahead reads 0.
```

Upravo kao što se vidi, traženje po indeksu je bolje i optimizer bira baš taj ako mu je dozvoljeno da bira.

## Indeks visoke gustine i klasterovani indeks

Ako neklasterovani indeks koji SQL može da koristi nije pokrivajući, onda za svaki red u rezultujućem setu mora da se odradi *lookup* nazad u klaster/gomilu za ostatak kolona. Ti *lookups* po ključu (iliti RID) su skupe operacije. Ako je previše njih potrebno, onda će optimizer preći na skeniranje, ne neklasterovanog indeksa (to bi bilo beskorisno, jer i dalje indeks ne pokriva zahteve upita), već klasterovanog indeksa koji barem ima sve kolone koje su potrebne u upitu (takođe bi mogao da pređe na skeniranje nekog drugog neklasterovanog indeksa ako postoji neki koji pokriva upit, ali sa pogrešnim redosledom kolona za pretraživanje).

Ukratko, imati visoku gustinu neklasterovanog indeksa rezultira skeniranjem tog indeksa? Ne (osim ako predikat nije pretražujuć (*SARGable*)), ipak može rezultirati skeniranjem drugog indeksa (verovatno klasterovanog) ako indeks ne pokriva upit i taj indeks sa visokom gustinom ostaje neiskorišćen.

[2]

# Klasterovani indeks

Klasterovani indeks (eng. Clustered Index) definiše redosled u kom se podaci fizički smeštraju u tabeli. Podaci iz tabele mogu biti sortirani samo u jednom pravcu, tako da može postojati samo jedan klasterovani indeks po tabeli. U SQL Server-u, ograničenje primarnog ključa (eng. Primary Key Constraint) automatski kreira klasterovani indeks nad tom određenom kolonom.

Pogledajmo sledeći primer. Prvo, kreiramo tabelu *student* unutar baze *schooldb* izvršavanjem sledeće skripte:

```
CREATE DATABASE schooldb;

USE schooldb;

CREATE TABLE student
(
   id INT PRIMARY KEY,
   name VARCHAR(50) NOT NULL,
   gender VARCHAR(50) NOT NULL,
   DOB DATETIME NOT NULL,
   total_score INT NOT NULL,
   city VARCHAR(50) NOT NULL
);
```

Primetimo ovde da tabela *student* ima postavljeno ograničenje primarnog ključa nad kolonom *id*. Ovo automatski kreira i klasterovani indeks nad *id* kolonom. Da bismo videli sve indekse nad određenom tabelom možemo izvršiti *sp_helpindex* proceduru. Ova procedura prihvata naziv tabele kao parametar i pribavlja sve indekse te tabele. Sledeći upit pribavlja indekse kreirane nad tabelom *student*:

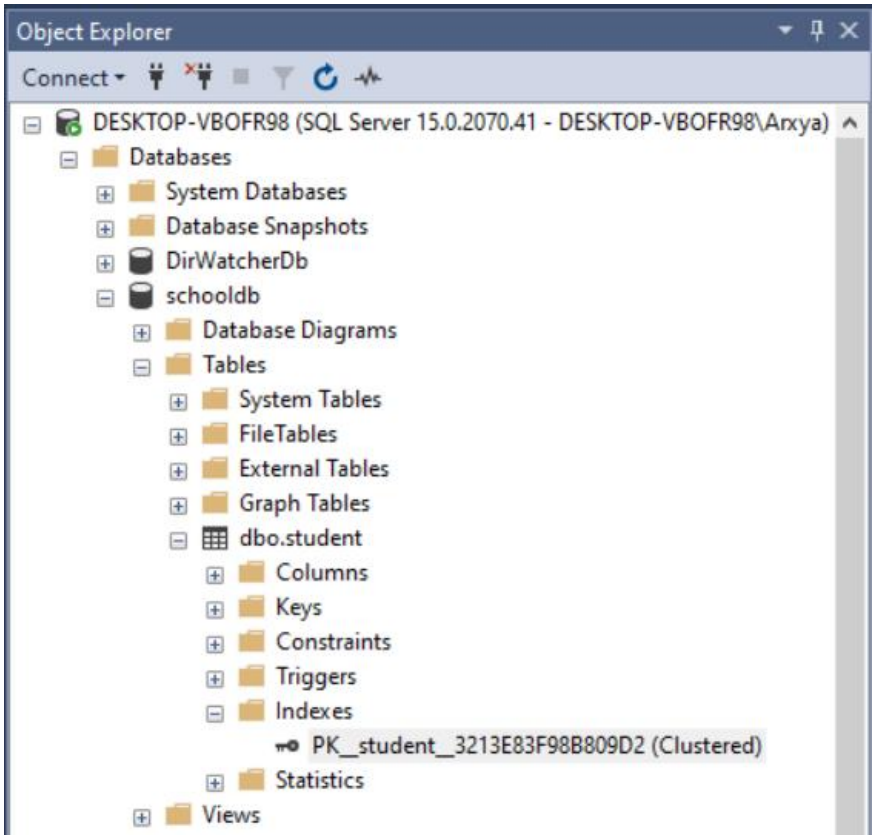
```
USE schooldb

EXECUTE sp_helpindex student
```

Prethodni upit će vratiti rezultat sličan ovome:

| index_name | index_description | index_keys |
|---|---|---|
| PK__student__3213E83F98B809D2 | clustered, unique, primary key located on PRIMARY | id |

U izlazu možemo videti samo jedan indeks. Ovo je indeks koji je automatski kreiran zbog primarnog ključa nad *id* kolonom.

Još jedan način da vidimo indekse je kroz *Object Explorer* prozor unutar *Microsoft SQL Server Management Studio* aplikacije:




Ovaj klasterovani indeks čuva podatke tabele *student* u rastućem redosledu vrednosti kolone *id*. Iz toga zaključujemo, ako se ubaci rekord sa *id* vrednošću 5, rekord će biti ubačen kao 5. red umesto kao prvi red (pretpostavljajući da imamo rekorde 1, 2, 3 i 4). Slično, ako ubacimo četvrti rekord sa *id* vrednošću 3, biće ubačen na 3. mesto. Ovo je tako zato što klasteroani indeks mora da održi fizički redosled sačuvanih podataka u skladu sa indeksnom kolonom (*id*). Kako bismo videli ovo u akciji, izvršićemo sledeću skriptu:

```
USE schooldb;

INSERT INTO student
VALUES
(6, 'Kate', 'Female', '03-JAN-1985', 500, 'Liverpool'),
(2, 'Jon', 'Male', '02-FEB-1974', 545, 'Manchester'),
(9, 'Wise', 'Male', '11-NOV-1987', 499, 'Manchester'),
(3, 'Sara', 'Female', '07-MAR-1988', 600, 'Leeds'),
(1, 'Jolly', 'Female', '12-JUN-1989', 500, 'London'),
(4, 'Laura', 'Female', '22-DEC-1981', 400, 'Liverpool'),
(7, 'Joseph', 'Male', '09-APR-1982', 643, 'London'),
(5, 'Alan', 'Male', '29-JUL-1993', 500, 'London'),
(8, 'Mice', 'Male', '16-AUG-1974', 543, 'Liverpool'),
(10, 'Elis', 'Female', '28-OCT-1990', 400, 'Leeds');
```

Skripta iznad ubacuje (eng. Inserts) 10 rekorda u tabelu *student*. Primetimo kako su rekordi ubacivani u nasumičnom redosledu u odnosu na *id* kolonu. Ali zato što klasterovani indeks postoji nad kolonom *id*, rekordi su fizički sačuvani u rastućem redosledu vrednosti *id* kolone. Izvršimo sledeći *SELECT* upit kako bismo pribavili podatke iz *student* tabele:

```
USE schooldb;

SELECT * FROM student;
```

Rekordi će biti pribavljeni u sledećem redosledu:

| id | name | gender | DOB | total_score | city |
|---|---|---|---|---|---|
| 1 | Jolly | Female | 1989-06-12 00:00:00.000 | 500 | London |
| 2 | Jon | Male | 1974-02-02 00:00:00.000 | 545 | Manchester |
| 3 | Sara | Female | 1988-03-07 00:00:00.000 | 600 | Leeds |
| 4 | Laura | Female | 1981-12-22 00:00:00.000 | 400 | Liverpool |
| 5 | Alan | Male | 1993-07-29 00:00:00.000 | 500 | London |
| 6 | Kate | Female | 1985-01-03 00:00:00.000 | 500 | Liverpool |
| 7 | Joseph | Male | 1982-04-09 00:00:00.000 | 643 | London |
| 8 | Mice | Male | 1974-08-16 00:00:00.000 | 543 | Liverpool |
| 9 | Wise | Male | 1987-11-11 00:00:00.000 | 499 | Manchester |
| 10 | Elis | Female | 1990-10-28 00:00:00.000 | 400 | Leeds |

**NAPOMENA: Realnost je ovde da će plan izvršenja biti napravljen takav da se podaci traže po ključu koji postoji i zadovoljava uslov pretrage i sadrži podatke koji se traže, odnosno podaci mogu biti smešteni i u drugačijem redosledu nego što prikaz prikazuje; činjenica u našem slučaju je da je jedini takav ključ upravo primarni ključ po kojem se podaci fizički smeštaju i zato poistovećujem prikaz sa redosledom smeštanja unutar memorije*

## Kreiranje personalizovanog klasterovanog indeksa

Možemo kreirati klasterovani indeks isto kao i podrazumevani. Da bismo kreirali novi klasterovani indeks, moramo prvo obrisati klasterovani indeks koji je kreiran uz ograničenje primarnog ključa.

Kako bismo obrisali indeks možemo izvršiti sledeći upit ili to odraditi unutar *GUI*-a *SSMS*-a (*Object Explorer – Databases - :ime baze: - Tables - :ime tabele: - Indexes - :desni klik na indeks: - Delete - Ok*):

```
USE schooldb;

CREATE CLUSTERED INDEX IX_tblStudent_Gender_Score
ON student(gender ASC, total_score DESC);
```

Proces kreiranja klasterovanog indeksa je sličan normanom kreiranju indeksa sa jednim izuzetkom: potrebno je navesti ključnu reč *CLUSTERED* pre *INDEX*.

Skripta iznad kreira indeks sa nazivom *IX_tblStudent_Gender_Score* nad tabelom *student*. Ovaj indeks je kreiran nad kolonama *gender* i *total_score*. Indeks kreiran nad više kolona se naziva **kompozitni indeks**.

Gore pomenuti indeks prvo sortira u rastućem redosledu kolone *gender*. Ako je *gender* isti za dva ili više redova, rekordi se sortiraju u opadajućem redosledu vrednosti *total_score* kolone. Može se kreirati klasterovani i nad jednom kolonom. Ako bismo sada potražili sve rekorde tabele *student*, podaci će biti vraćeni u sledećem redosledu:

| id | name | gender | DOB | total_score | city |
|---|---|---|---|---|---|
| 3 | Sara | Female | 1988-03-07 00:00:00.000 | 600 | Leeds |
| 1 | Jolly | Female | 1989-06-12 00:00:00.000 | 500 | London |
| 6 | Kate | Female | 1985-01-03 00:00:00.000 | 500 | Liverpool |
| 4 | Laura | Female | 1981-12-22 00:00:00.000 | 400 | Liverpool |
| 10 | Elis | Female | 1990-10-28 00:00:00.000 | 400 | Leeds |
| 7 | Joseph | Male | 1982-04-09 00:00:00.000 | 643 | London |
| 2 | Jon | Male | 1974-02-02 00:00:00.000 | 545 | Manchester |
| 8 | Mice | Male | 1974-08-16 00:00:00.000 | 543 | Liverpool |
| 5 | Alan | Male | 1993-07-29 00:00:00.000 | 500 | London |
| 9 | Wise | Male | 1987-11-11 00:00:00.000 | 499 | Manchester |

# Neklasterovani indeks

**Neklasterovani indeks** ne sortira fizičke podatke unutar tabele. Ustvari, neklasterovani indeks se čuva na jednom, a sama tabela sa podacima na drugom, odvojem mestu. Ovo je slično tome što se u knjizi indeks nalazi obično na kraju knjige, a sam sadržaj svuda po knjizi. Iz ovoga proističe mogućnost postojanja više od jednog neklasterovanog indeksa.

Važno je pomenuti da će podaci unutar tabele biti sortirani po klasterovanom indeksu, dok će redosled istih tih redova biti drugačiji u neklasterovanom indeksu. Indeks sadrži vrednosti kolona nad kojima je kreiran i adresu rekorda kojem te kolone pripadaju.

Kada se se izda upit nad kolonom nad kojom je kreiran indeks, SQL Server Engine će prvo proći indeks i tražiti adresu odgovarajućeg reda u tabeli. Onda će otići na tu adresu i pribaviti vrednosti ostalih kolona. Zbog ovog dodatnog koraka je neklasterovani indeks sporiji od klasterovanog indeksa.

## Kreiranje neklasterovanog indeksa

Sintaksa za kreiranje neklasterovanog indeksa je veoma slična sintaksi upita za kreiranje klasterovanog indeksa. U slučaju neklasterovanog indeksa koristi se ključna reč *NONCLUSTERED* umesto *CLUSTERED*. Pogledajmo sledeću skriptu:

```
USE schooldb;

CREATE NONCLUSTERED INDEX IX_tblStudent_Name ON student(name ASC);
```

Skripta iznad kreira neklasterovani indeks nad kolonom *name* tabele *student*. Indeks sortira po *name* u rastućem redosledu. Kako smo ranije rekli, podaci tabele i indeksa se čuvaju na odvojenim mestima. Podaci tabele će biti sortirani po klasterovanom indeksu ako takav postoji. Indeks će biti sortiran po svojoj definiciji i biti skladišten odvojeno od tabele.

Podaci tabele *student* (redosled u memoriji po klasterovanom indeksu):

| id | name | gender | DOB | total_score | City |
|---|---|---|---|---|---|
| 1 | Jolly | Female | 1989-06-12 00:00:00.000 | 500 | London |
| 2 | Jon | Male | 1974-02-02 00:00:00.000 | 545 | Manchester |
| 3 | Sara | Female | 1988-03-07 00:00:00.000 | 600 | Leeds |
| 4 | Laura | Female | 1981-12-22 00:00:00.000 | 400 | Liverpool |
| 5 | Alan | Male | 1993-07-29 00:00:00.000 | 500 | London |
| 6 | Kate | Female | 1985-01-03 00:00:00.000 | 500 | Liverpool |
| 7 | Joseph | Male | 1982-04-09 00:00:00.000 | 643 | London |
| 8 | Mice | Male | 1974-08-16 00:00:00.000 | 543 | Liverpool |
| 9 | Wise | Male | 1987-11-11 00:00:00.000 | 499 | Manchester |
| 10 | Elis | Female | 1990-10-28 00:00:00.000 | 400 | Leeds |

Podaci *IX_tblStudent_Name* indeksa:

| name | Row Address |
|---|---|
| Alan | Row Address |
| Elis | Row Address |
| Jolly | Row Address |
| Jon | Row Address |
| Joseph | Row Address |
| Kate | Row Address |
| Laura | Row Address |
| Mice | Row Address |
| Sara | Row Address |
| Wise | Row Address |

Primetimo, ovde u indeksu svaki red ima kolonu koja sadrži adresu reda kome vrednost kolone *name* pripada. tako da ako jer izdat upit koji treba pribaviti *gender* i *DOB* studenta sa *name* vrednošću „Jon'', SQL Server Engine će prvo potražiti indeks po vrednosti *name*, zatim će iščitati adresu reda koji ima tu vrednost i iz tabele *student* izvući *gender* i *DOB*.

**NAPOMENA: U zavisnosti od statistike tabele ili postojanja boljeg indeksa, SQL Server Engine može odabrati drugu opciju traženja podatka; ovo je samo ilustracija trenutnog primera.*

Prethodni primer neklasterovanog indeksa je bio najprostiji primer neklasterovanog nad jednom kolonom i bez priključenih (eng. Included) drugih kolona. U sekciji klasterizovanog indeksa smo videli da se jedan indeks može kreirati nad više kolona i da se podaci redom tako sortiraju u indeksu. Ovo je istinito i za neklasterovane indekse.

Ako nadogradimo prethodni indeks i dodamo još jednu kolonu u *ON* delu indeksa dobićemo sledeći upit:

```
USE schooldb;

CREATE NONCLUSTERED INDEX IX_tblStudent_Name ON student (
        name ASC,
        gender DESC
);
```

Ovo će efektivno dodati i kolonu *gender* unutar indeksa i kopirati sve podatke te kolone takodje u indeks. Takodje će sve redove sa istom vrednošću *name* kolone dodatno sortirati u opadajućem redosledu *gender* kolone (isto kao i kod klasterovanog) unutar indeksa.

Još jedan dodatak koji možemo ubaciti je tzv. priključena kolona (ili više njih). Sledi upit:

```
USE schooldb;
```

```
CREATE NONCLUSTERED INDEX IX_tblStudent_Name ON student (
        name ASC,
        gender DESC
)
INCLUDE (total_score);
```

Primetimo poslednji red upita. Ova *INCLUDE* sekcija upita će reći SQL Server-u da ubaci i kolonu *total_score* u indeks zajedno sa podacima *name* i *gender*, sa razlikom da neće uzimati u obzir tu kolonu za sortiranje (i interno kreiranje strukture B-stabla). Ta kolona je tu za čisto iščitavanje vrednosti direktno iz indeksa (ova osobina je ključna za **prekrivajuće indekse**).

| name | gender | total_score | Row Address |
|---|---|---|---|
| Alan | Male | 500 | Row Address |
| Elis | Female | 400 | Row Address |
| Jolly | Female | 500 | Row Address |
| Jon | Male | 545 | Row Address |
| Joseph | Male | 643 | Row Address |
| Kate | Female | 500 | Row Address |
| Laura | Female | 400 | Row Address |
| Mice | Male | 543 | Row Address |
| Sara | Female | 600 | Row Address |
| Wise | Male | 499 | Row Address |

Što više kolona uključimo u *ON* sekciji neklasterovanog indeksa, raste potreba za memorijom i za same vrednosti kolone i za strukturu B-stabla u pozadini. Što se tiče *INCLUDE* sekciji, te informacije se ne koriste u kreiranju strukture B-stabla, tako da se potreba za dodatnom memorijom za takve kolone odnosi samo na same podatke iz kolone.

[4]

[5]

## Posledice korišćenja neklasterovanih indeksa

Neklasterovani indeksi donose ogromna ubrzanja kada se pravilno projektuju. Prelazak sa *Scan* na *Seek* operaciju (kada je to moguće) umnogo smanjuje izvršenje upita, nebitno da li su u pitanju *WHERE* klauzule ili uslovi *JOIN*-ova.

S tim na umu, postoje i loše strane korišćenja neklasterovanih indeksa.

Jedna od loših strana korišćenja neklasterovanih indeksa je dodatno **zauzeće memorije**. Naime, kako bi se kreirali neklasterovani indeksi, zauzima se prostor za svaku kolonu koja je uključena u indeks (u *ON* i *INCLUDE* sekciji) za svaki red u tabeli. Odnosno imamo bukvalno dupliranje podataka tih kolona. Pored samih podataka tu je i prostor koji zauzima struktura podataka B-stabla koje je srž svakog indeksa. Što više kolona uključimo u *ON* sekciju indeksa, kompleksnost strukture B-stabla raste, a samim tim i memorija koju ta struktura zauzima. I na kraju tu su adrese redova u tabeli, koje, koliko god male bile, ipak zauzimaju memorijski prostor.

Druga loša strana je **potreba za ažuriranjem** svakog indeksa pri svakom dodavanju ili brisanju novog reda u tabeli, ili čak pri ažuriranju vrednosti u koloni koja je uključena u indeks. Ako je izmenjena kolona uključena samo u *INCLUDE* sekciju indeksa, potrebno je ,,samo'' ažurirati i tu vrednost. Ako je pak kolona u *ON* sekciji, odnosno ako je indeks kreiran nad tom kolonu, potrebno je ažurirati kompletan deo B-stabla koji se odnosi na tu kolonu. Što više indeksa,više je ovakvih situacija i u nekom trenutku indeksi nad tabelom mogu toliko opteretiti sistem da je bazi potrebno previše vremena da ažurira podatke. Ovo dovodi do zaključavanja tabele i indeksa i do zastoja sistema koji možda zavisi od te tabele.

Postoji opcija *FILLFACTOR* koja se može navesti prilikom kreiranja ili menjanja indeksa. Vrednost ove opcije označava koji je procenat memorije koju zauzima indeks u početnom stanju kada se memorija za indeks rezerviše. Kada se navede vrednost drugačija, manja od 100 (100 je max, min je 0, ali 0 ima isto značenje kao 100) to znači da se rezerviše dodatni prostor koji početno neće biti zauzet već samo rezervisan. Ovo ostavlja mogućnost da se lakše/brže indeks ažurira kada se ažuriraju podaci u tabeli, jer nije potrebno rezervisati prostor u listovima B-stabla. Dobra stvar ovog pristupa je upravo to, DBMS ne mora slati zahtev operativnom sistemu za rezervaciju memorije, jer je već ima, i ubrzava se ažuriranje indeksa, pa čak i smanjuje fragmentacija u određenom procentu. Loša strana ovoga je naravno inicijalna veća potreba za memorijom za indeks, što, kao što smo već rekli, dovodi do degradacije brzine sistema jer je potrebno pročitati više memorije kada se indeks referencira. Sledi primer definicije indeksa sa definisanim *FILLFACTOR*-om na vrednost 80, što znači da će biti zauzeto otprilike 125% memorije od potrebnih 100% za ovaj indeks:

```
CREATE NONCLUSTERED INDEX IX_tblStudent_Name ON student (
        name ASC,
        gender DESC
)
INCLUDE (total_score)
WITH (FILLFACTOR = 80);
```

[6]

Još jedna od loših strana je način smeštanja indeksa u memoriju. Obzirom da se u bazi redovno ubacuju podaci i zatim brišu, ostaju rupe u masovnoj memoriji koje će baza iskoristiti da ,,podmetne'' neke druge podatke kada dovoljno mali podaci naiđu. Ti podaci mogu biti i baš podaci iz indeksa, odnosno moguće je da se celokupan indeks ne može smestiti u sukcesivnim lokacijama na disku, posebno ako je indeks glomazan, nad glomaznom tabelom koja je sklona ažuriranju. Ovo dovodi do pojave **fragmentacije indeksa**. Fragmentacija je čest pojam u fizičkom skladištenju podataka iz

razloga koje sam naveo. Fragmentacija je loša iz razloga što je potrebno čitati podatke koji su logički sukcesivni, ali fizički odvojeni, što dovodi do potrebe čitanja sa više mesta u masovnoj memoriji, što je kod klasičnih hard diskova skupa operacija sa stanovišta vremena izvršenja. Fragmentacija indeksa je čest problem i zato je potrebno s vremena na vreme ponovo izgraditi, obnoviti (eng. Rebuild) indeks kako bi se smanjila fragmentacija i kako bi čitanje podataka indeksa iz memorije bilo brže. Sledi primer fragmentacije indeksa pre i posle obnavljanja indeksa.

(*pre*)

| index_id | name | avg_fragmentation_in_percent |
|---|---|---|
| 1 | PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID | 36.13581245 |
| 2 | AK_SalesOrderDetail_rowguid | 2.643171806 |
| 3 | IX_SalesOrderDetail_ProductID | 25.83892617 |

(*posle*)

| index_id | name | avg_fragmentation_in_percent |
|---|---|---|
| 1 | PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID | 0.24291498 |
| 2 | AK_SalesOrderDetail_rowguid | 0 |
| 3 | IX_SalesOrderDetail_ProductID | 0 |

Indekse možemo obnoviti izdavanjem *REORGANIZE* ili *REBUILD* naredbe nad indeksom. *REORGANIZE* se koristi kada je procenat fragmentacija između 10 i 30 %. *REBUILD* se koristi kada se je procenat fragmentacije preko 30%. Primeri:

*Rebuild samo jednog indeksa:*

```
ALTER INDEX IX_SalesOrderDetail_ProductID ON Sales.SalesOrderDetail REORGANIZE
```

*Reorganizacija svih indeksa nad tabelom (konkretno ova je iskorišćena u prethodnom primeru):*

```
ALTER INDEX ALL ON Sales.SalesOrderDetail REBUILD
```

Sledi upit kojim su dobijene tabele koje su date u prošlom primeru:

```
SELECT
        a.index_id,
        name,
        avg_fragmentation_in_percent
FROM sys.dm_db_index_physical_stats (DB_ID('AdventureWorks2012'),
OBJECT_ID('Sales.SalesOrderDetail'), NULL, NULL, NULL) AS a
JOIN sys.indexes AS b ON a.object_id = b.object_id AND a.index_id = b.index_id
```

# Unikatni, kompozitni i prekrivajući indeksi

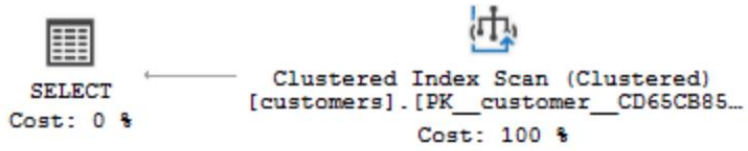
**Unikatni** (eng. Unique) indeksi služe da bi se osiguralo da kolona (ili kolone) nad kojom se definiše indeks ne može imati dva reda sa istom vrednošću. Ovaj indeks se obično poistovećuje sa ograničenjem unikata (eng. Unique Constraint), a to je zato što se prilikom kreiranja ograničenja unikata nad kolonom automatski pravi unikatni indeks. Mada se unikatni indeks može nezavisno definisati, ograničenje unikata jasnije prikazuje nameru.

Radi demonstracije rada unikatnog indeksa iskoristićemo sledeći primer.

Uzmimo postojanje tabele *customers* unutar *sales* šeme. Tabela ima dve kolone: *customer_id* i *email*. Nad kolonom *customer_id* definisan je klasterovani indeks. Želimo da pretražimo *customers* tabelu i izvučemo određen rekord sa određenim email-om:

```
SELECT
   customer_id,
   email
FROM
   sales.customers
WHERE
   email = 'caren.stephens@msn.com';
```

U trenutnoj situaciji DBMS će izgraditi sledeći plan izvršenja upita (eng. Query Execution Plan):

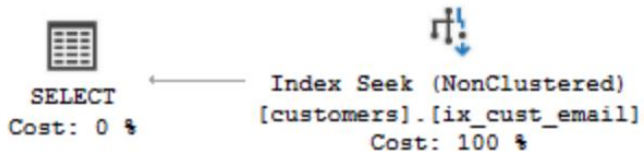



Sa slike vidimo da će Engine morati da skenira celu bazu po klasterovanom indeksu jer nema bolju alternativu. Kada kažem celu, mislim celu tabelu, jer nije poznato da li postoji samo jedan takav red ili više njih.

Sada, ako pretpostavimo da nema duplikata u koloni *email* (to možemo da proverimo i izbacimo duplikate pre sledećeg koraka), možemo da kreiramo unikatni indeks nad kolonom *email*.

```
CREATE UNIQUE INDEX ix_cust_email
ON sales.customers(email);
```

Ako ponovo pokrenemo isti *SELECT* upit, plan izvršenja upita biće:




Sada se radi traženje (eng. Seek) po našem unikatnom neklasterovanom indeksu i prolaskom kroz B-stablo u indeksu dolazimo do rezultata veoma brže i sigurno smo da ovakav upit sa operatorom jednakosti u *WHERE* klauzuli vraća maksimalno jedan rezultat, sa dodatnim ubrzanjem prestanka traženja nakon što se nađe traženi rezultat. Takodje poboljšava situaciju po pitanju brzine i kod *JOIN*-

ova u određenim situacijama (kada se *JOIN* radi samo na unikatnim kolonama, jer nije potrebno dalje traženje poklapanja jer je unapred poznato da ih neće biti).

Bitno je pomenuti da se *NULL* vrednost gleda kao unikatna vrednost, stoga u koloni koja ima definisan unikatni indeks, samo jedan red može biti sa vrednošću *NULL* (pod uslovom da je sama kolona definisana kao *NULLABLE*).

[7]

Unikatni indeks zapravo nije vrsta indeksa, već nadogradnja na indeks, pa se tako može kreirati i unikatni neklasterovani i unikatni klasterovani indeks (ovaj se zapravo kreira automatski kada se kreira ograničenje primarnog ključa).

Unikatni indeks, kako je to samo dopunjen klasterovani ili neklasterovani indeks, može biti definisan i nad više kolona, čime dolazimo do kompozitnog indeksa.

Već smo više puta pomenuli, objasnili i dali primer kompozitnog indeksa, ali ponovićemo zbog celosti segmeta rada.

**Kompozitni indeks** (eng. Composite Index) je indeks koji se kreira nad više od jedne kolone, odnosno indeks čije se B-stablo kreira na osnovu sortiranja vrednosti više kolona.

Kompozitni indeks može biti i klasterovani i neklasterovani indeks. Primer kompozitnog indeksa koji smo već koristili:

```
CREATE NONCLUSTERED INDEX IX_tblStudent_Name ON student (
        name ASC,
        gender DESC
)
INCLUDE (total_score)
WITH (FILLFACTOR = 80);
```

Kompozitan je jer su i *name* i *gender* kolone (više od jedne) u definiciji indeksa. Kolona *total_score* ne utiče na kreiranje B-stabla tako da nije deo kompozita ovog indeksa. Takodje, *FILLFACTOR* nema nikakav uticaj na činjenicu da li je ovo kompozitni indeks ili nije.

Prednost kompozitnog indeksa je činjenica da se ponaša kao indeks unutar indeksa zato što se redovi sortiraju po prvoj navedenoj koloni, pa duplikati prve kolone po drugoj, itd. Ovo doprinosi upotrebljivosti indeksa kada se pretraga vrži u tom ili sličnom redosledu. Mane su veće zauzeće memorije i duže trajanje ažuriranja i kreiranja indeksa.

Za kreiranje kompozitnog indeksa ne postoji nikakava specijalna ključna reč u upitu, već je samo neophodno smestiti više od jedne kolone unutar liste kolona definicije indeksa.

I na kraju dolazimo do pokrivajućih indeksa. **Pokrivajući indeks** (eng. Covering Index) je indeks koji sadrži sve informacije koje su potrebne upitu. U nekom smislu, apsolutno svaki indeks je pokrivajući indeks; sve zavisi od upita. Odnosno, osobina pokrivanja indeksa zavisi od upita do upita.
Objašnjenje:

Uzmimo najprostiji mogući indeks nad jednom kolonom:

```
CREATE NONCLUSTERED INDEX IX_tblStudent_Name ON student (
        name ASC
);
```

Ako pokrenemo sledeći upit nad tabelom student koja ima samo ovaj indeks:

```
SELECT
        name,
        gender
FROM student
WHERE name = 'Joe';
```

indeks će se koristiti u upitu, ali će i dalje morati da se referencira tabela pomoću adrese iz indeksa, jer je potrebna i *gender* informacija osim *name* po kojoj se filtrira izlaz.

Sa druge strane ako pustimo sledeći upit:

```
SELECT
        name
FROM student
WHERE name = 'Joe';
```

indeks sadrži sve informacije potrebne upitu.

Dobro, možda se prethodni *SELECT* upit čini kao nonsens, mada proverava postojanje takvog rekorda. Umesto toga moguće je koristiti postojeći indeks kao pokrivajući za upit:

```
SELECT
        name
FROM student
ORDER BY name;
```

Ovaj upit će koristiti samo postojeći indeks tako što će samo vratiti direktno vrednosti iz listova indeksa, jer su podaci kolone *name* već sortirani u indeksu.

Ako pak ipak želimo pokrivajući indeks za upit sa dve kolone:

```
SELECT
        name,
        gender
FROM student
WHERE name = 'Joe';
```

možemo ga napraviti na više načina, od kojih su neki pravilni neki nepravilni:

```
CREATE NONCLUSTERED INDEX IX_tblStudent_Name ON student (
        name ASC,
        gender DESC
);
```

```
CREATE NONCLUSTERED INDEX IX_tblStudent_Name ON student (
        name ASC
)
INCLUDE (
        gender
);
```

Oba prethodna indeksa su pokrivajući indeksi za poslednji *SELECT* upit.

Razmotrimo sada sledeći indeks za isti *SELECT* upit:

```
CREATE NONCLUSTERED INDEX IX_tblStudent_Name ON student (
        gender DESC
)
INCLUDE (
        name
);
```

ovaj indeks nije pokrivajući za upit, čisto iz razloga što se filtriranje vrši po koloni *name*, a ta kolona je samo priključena kolona indeksu, indeks nije od koristi pri pretrazi po datom uslovu.

Osim na nivou upita, indeks može biti pokrivajući i na nivou tabele, ali ponovo od upita zavisi da li će se koristiti on ili će se koristiti skeniranje tabele ili drugog indeksa. Potsetimo se definicije tabele:

```
CREATE TABLE student
(
   id INT PRIMARY KEY,
   name VARCHAR(50) NOT NULL,
   gender VARCHAR(50) NOT NULL,
   DOB DATETIME NOT NULL,
   total_score INT NOT NULL,
   city VARCHAR(50) NOT NULL
 );
```

Bilo kakva pretraga samo po koloni *id* će se tražiti po klasterovanom indeksu koji je kreiran uz primarni ključ. Takva pretraga će biti izuzetno brza i dostupne su odmah sve kolone. Ali šta ako iz nekog razloga želimo ,,istu'' brzinu pretrage po nekoj drugoj koloni kada izvlačimo sve kolone?

**NAPOMENA: Naglašeno je ,,**istu**'' iz razloga što će uvek neklasterovan indeks biti veći od klasterovanog, a samim tim i čitanje indeksa, samim tim i sporiji odziv na upit, ali može biti približno brzina ista.*

U tom slučaju, ako nemamo potrebe za brigom o memoriji i drugim resursima, a želimo slično ponašanje nad kolonom name, možemo definisati indeks kao što je npr.:

```
CREATE NONCLUSTERED INDEX IX_tblStudent_Name_COVER ON student (
        name ASC
)
INCLUDE (
        id,
        gender,
        DOB,
        total_score,
        city
);
```

Upit:

```
SELECT * FROM student WHERE name = 'Joe';
```

će koristiti indeks *IX_tblStudent_Name_COVER* u celosti bez ijednog obraćanja tabeli.

Ovo bukvalno dovodi do kreiranja kopije celokupne tabele na disku sa stanovišta podataka. Prednost od jednostavnog kopiranja tabela je što se ne mora voditi računa o sinhronizaciji podataka između dve kopije tabele. Nedostataka je previše: prevelika upotreba memorije (ovakav indeks će „pojesti'' više memorije od same tabele), održavanje indeksa pri ažuriranju tabele preskupo sa strane procesorskog vremena, više verovatna pojava fragmentacije indeksa, itd. itd.

U suštini, u praksi je ovakve indekse najbolje izbegavati.

Kao što se vidi u prethodnim primerima, kompozitni indeksi su pokrivajući indeksi za određene upite. Npr:

```
CREATE NONCLUSTERED INDEX IX_tblStudent_COVER2 ON student (
        name ASC ,
        gender DESC,
        id ASC
);
```

```
SELECT name, gender, id FROM student ORDER BY name;
```

# Literatura

[1] https://www.sqlshack.com/sql-server-index-structure-and-concepts/

[2] https://sqlinthewild.co.za/index.php/2015/10/06/index-selectivity-and-index-scans/

[3] https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2005/administrator/cc966419

[4] https://www.sqlshack.com/what-is-the-difference-between-clustered-and-non-clustered-indexes-in-sql-server/

[5] https://docs.microsoft.com/en-us/sql/relational-databases/indexes/create-nonclustered-indexes

[6] https://docs.microsoft.com/en-us/sql/relational-databases/indexes/specify-fill-factor-for-an-index

[7] https://www.sqlservertutorial.net/sql-server-indexes/sql-server-unique-index/