**DBMS**

**Optimizacija SQL upita na Microsoft SQL Server-u**

Studenti:

Filip Stamenković, 342

Aleksa Trajković, 370

Sadržaj

[Uvod 3](#_Toc476688632)

[Inicijalno stanje 5](#_Toc476688633)

[Modifikovanje SQL upita 5](#_Toc476688634)

[SQL Server Query Optimizer 5](#_Toc476688635)

[Kako optimizator upita radi 5](#_Toc476688636)

[Generisanje i procena planova izvršenja (kandidata) 6](#_Toc476688637)

[Keširanje test plana 7](#_Toc476688638)

[Nagoveštaji 7](#_Toc476688639)

[Plan izvršenja 7](#_Toc476688640)

[Redosled spojeva 8](#_Toc476688641)

[Dodatna optimizacija upita kod paging-a 10](#_Toc476688642)

[Views 11](#_Toc476688643)

[Standardni View 11](#_Toc476688644)

[Indexed View 11](#_Toc476688645)

[Indeksi 12](#_Toc476688646)

[Predlog Tuning Advisor-a 14](#_Toc476688647)

[Prilog A 17](#_Toc476688648)

[Prilog B 19](#_Toc476688649)

[Prilog C 20](#_Toc476688650)

[Prilog D 22](#_Toc476688651)

[Prilog E 23](#_Toc476688652)

[Entity Framework 23](#_Toc476688653)

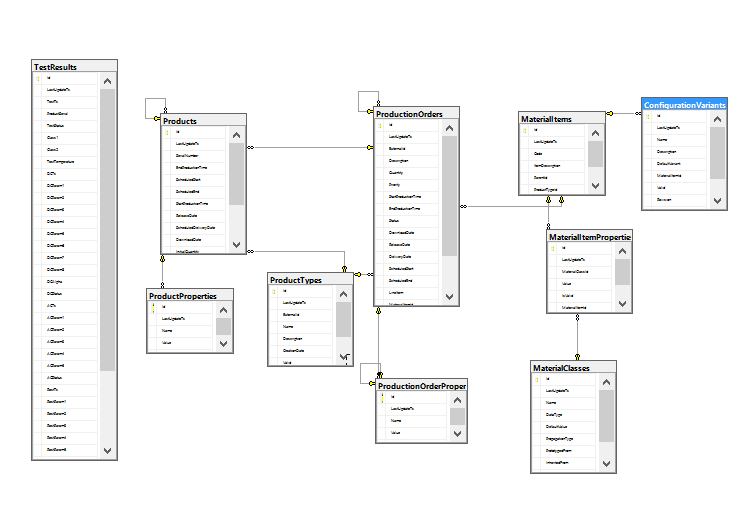
[Literatura 24](#_Toc476688654)

# Uvod

Najkorišćeniji upiti su oni koji pribavljaju podatke preko SELECT naredbe. Povećanjem količine podataka postaje bitno kako je upit napisan. Pored poznavanja SQL sintakse, programer mora poznavati specifičnosti, prednosti i mane DBMS-a nad kojim se izvršava upit. Pored prepravljanja upita, na raspolaganju su tehnike dodavanje indeksa, pogleda, statistika itd.

Ovaj rad se bavi analizom jednog kompleksnog SQL upita, koji sadrži preko 10 spojeva, sa tabelama koje sadrže nekoliko stotina hiljada slogova. Ovaj SQL upit prikazuje rezultate testiranja proizvoda u okviru fabrike, kao i neke njegove osnovne karakteristike.

ER dijagram baze je dat na sledećoj slici.



Iako ne postoji strani ključ između tabela *TestResults* i *Products* i tabela *ConfigurationVariantItems* i *ProductionOrderProperties*, postoji veza koja se koristi u analiziranom SQL upitu.

Baza sadrži 10 tabela:

**TestResults** – 712.718 podataka

**Products** – 697.625 podataka

**ProductProperties** – 3.486.173 podatka

**ProductionOrders** – 642 podatka

**ProductionOrderProperties** – 11.151 podatka

**ProductTypes** – 6 podataka

**MaterialItems** – 105 podataka

**MaterialItemProperties** – 337 podataka

**MaterialClasses** – 5 podataka

**ConfigurationVariants** – 219 podataka

Zbog velike količine podataka u upitu je iskorišćena tehnika straničenja (*paging*).

Projekat koji prati ovaj rad se nalazi na adresi: <https://github.com/FilipStamenkovic/DBMS>

# Inicijalno stanje

Inicijalni SQL upit je imao podršku za *paging*, ali zbog slabih performansi i nemogućnosti filtriranja i sortiranja, morao je biti prepravljen.

U **prilogu A** je dat prikaz inicijalnog SQL upita. Izvršavanje ovog upita za prvu straincu, veličine 400 reda, traje 20 sekundi, što nije zadovoljilo očekivanja korisnika.

## Modifikovanje SQL upita

Inicijalni SQL upit je morao biti prepravljen kako bi filtriranje i sortiranje bilo podržano. Umesto 4 ugnježdena SELECT upita, modifikovani SQL upit ima samo jedan SELECT.

U **prilogu B** je dat prikaz modifikovanog SQL upita.

Izvršavanje modifikovanog upita traje isto kao i izvršavanje inicijalnog, 20 sekunde.

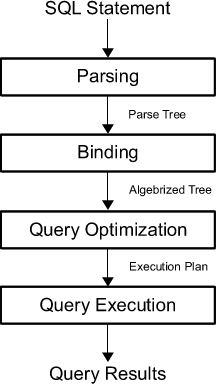
# SQL Server Query Optimizer

SQL Server Query Optimizer je optimizator na bazi predviđene cene (*cost-based*). Optimizator analizira više mogućih planova izvršenja upita (kandidati), procenjuje cenu izvršenja svakog od plana i na kraju bira plan sa najnižom cenom među razmatranim kandidatima. Optimizator ne može da razmotri svaki mogući plan izvršenja upita, jer mora da vodi računa i o ceni same analize planova, pored cene izvršenja.

## Kako optimizator upita radi

U jezgru SQL Server DBMS-a su dve glavne komponente, ***Storage Engine*** i ***Query Processor***. *Storage Engine* je odgovoran za razmenu podataka između diska i memorije, tako da maksimizuje konkurentnost, dok održava integritet podataka. *Query Processor* prihvata upite koji su prosleđeni SQL Serveru na izvršenje, osmišlja plan „optimalnog“ izvršenja, izvršava ga i vraća rezultate. Upiti se SQL Serveru prosleđuju korišćenjem SQL jezika (ili T-SQL jezika, Microsoft-ovog procedularnog proširenja SQL jezika). Pošto je SQL deklarativni jezik visokog nivoa, upitom definišemo samo koje podatke želimo, ne i neophodne korake za pribavljanje tih podataka, niti algoritme kojima se ovaj zahtev obrađuje. Stoga, za svaki upit koji primi, prvi zadatak *Query Processor*-a je da formira plan, što brže, koji opisuje najbolji način za izvršenje zadatog upita (ili barem efikasan način). Drugi zadatak je izvršenje upita prema formiranom planu.

Za svaki od ovih zadataka, zadužena je posebna komponenta *Query Processor*-a. Optimizator upita (***Query Optimizer***) formira plan i prosleđuje ga ***Execution Engine***-u, koji zapravo izršava plan i vraća rezultat iz baze.



Parsiranje proverava da li (T-)SQL ima validnu sintaksu, i prevodi SQL upit u stablo logičkih operacija visokog nivoa, koje su bliske operacijama iz originalnog SQL upita.

*Binding* se bavi proverom imena. Tokom ove operacije, SQL Server proverava da li svi objekti postoje, i vezuje svako ime tabele i kolone, iz stabla dobijenog parsiranjem, za odgovarajući objekat iz sistemskog kataloga.

Sledeći korak je generisanje određenog broja planova izvršenja, kao kandidata i odabir „najboljeg“ procenom cene izvršenja. Takođe, u ovom koraku se logičke operacije iz stabla mapiraju u fizičke operacije, koje *Execution Engine* ume da izvrši. Neke logičke operacije je moguće mapirati u više fizičkih, pa je posao optimizatora da izabere najefikasniju.

## Generisanje i procena planova izvršenja (kandidata)

Generisani i odabrani planovi kandidati se čuvaju u memoriji, u komponenti koja se naziva *Memo*. Za procenu cene plana, optimizator procenjuje cenu svakog fizičkog operatora iz toga plana, pomoću formula koje uzimaju u obzir korišćenje I/O, procesora i memorije. Ova cena najviše zavisi od algoritma koji fizički operator koristi, kao i od procenjenog broja slogova koje treba obraditi. Ova procena broja slogova se još naziva i **procena kardinalnosti**.

Za procenu kardinalnosti SQL Server koristi i održava **statistiku optimizatora**, koja sadrži statističke informacije koje opisuju raspodelu vrednosti u okviru neke kolone. Kada se izračuna cena svakog od operatora, njihovim sumiranjem se dobija cena celokupnog plana.

## Keširanje test plana

Generisani plan izvršenja može biti sačuvan u memoriji, u kešu planova, za slučaj ponovnog iskorišćenja, ako se zahteva izvršenje istog upita. Ako je validan plan dostupan u kešu planova, onda se proces optimizacije i procene cena može preskočiti.

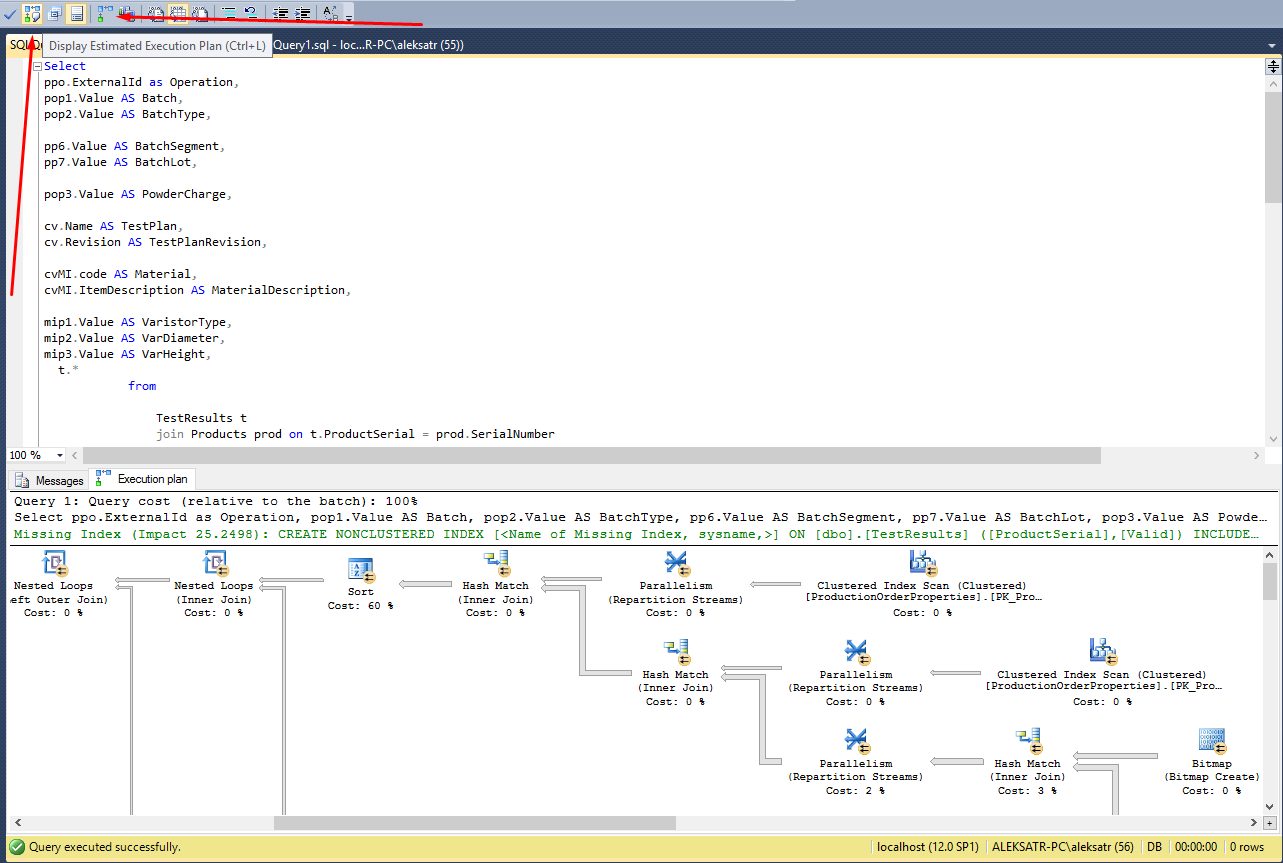
Ponovno iskorišćenje postojećeg plana ne mora uvek biti najbolja opcija. U zavisnosti od raspodele u okviru neke tabele, optimalnost plana može u velikoj meri zavisiti od parametara koji se prosleđuju upitu. Pored ovoga, neki metapodaci mogu biti promenjeni, brisanjem/dodavanjem indeksa, ograničenja, ili čak velikom promenom u sadržaju baze (menja se statistička raspodela vrednosti), čime keširani plan izvršenja postaje suboptimalan. Ovakve test planove SQL Server briše iz keša, kada proceni da postoji mogućnost da više nisu optimalni.

## Nagoveštaji

Mogu postojati slučajevi kada izabrani plan izvršenja nije dovoljno efikasan. U ovim slučajevima mogu se upotrebiti nagoveštaji (*hints*). Pošto nagoveštaji *override*-uju operacije optimizatora, moraju se koristiti oprezno, i samo kao zadnja opcija. Možemo nagovestiti optimizatoru da iskoristi određeni indeks, ili određeni algoritam spoja. Takođe, možemo proslediti kompletan plan izvršenja optimizatoru u vidu XML-a (za slučaj kada imamo plan koji je generisan na nekoj drugoj verziji SQL Servera, a ima bolje performanse).

## Plan izvršenja

Možemo zahtevati pravi plan izvršenja ili procenjeni, koji mogu biti prikazani grafički (u okviru Microsoft SQL Server Management Studio alata) ili u XML formatu. Kada se zahteva pravi plan izvršenja, potrevno je izvršiti upit, dok za procenjeni plan nije potrevno izvršavati upit i on predstavlja plan koji bi SQL Server najverovatnije odabrao, da ga je potrebno izvršiti.



Predviđeni plan izvršenja

Pravi plan izvršenja

Predlog optimizatora upita



## Redosled spojeva

Određivanje optimalnog redosleda spojeva je jedan od najkompleksinijih koraka u optimizaciji upita. Skup mogućih planova raste veoma brzo sa porastom broja spojeva tabela. Spoj kombinuje slogove iz dve tabele na osnovu predikata spoja. Spoj *n* tabela zahteva *n-1* spoj, jer je spoj operacija koja se izvršava nad dve tabele. Redosled spojeva utiče na količinu podataka koja se razmenjuje između operatora u planu izvršenja. Optimizator mora da napravi dve važne odluke u vezi spojeva:

1. Izbor redosleda spojeva
2. Izbor algoritma spoja

Kao posledica komutativne i asocijativne osobine operacije spoja, čak i jednostavni upiti nude puno različitih redosleda spojeva, i ovaj broj raste eksponencijalno sa porastom broja tabela koje se spajaju.

Ako pogledamo plan izvršenja inicijalnog upita, možemo videti da se redosled spojeva razlikuje od specificiranog u upitu, što znači da je optimizator našao efikasniji redosled.

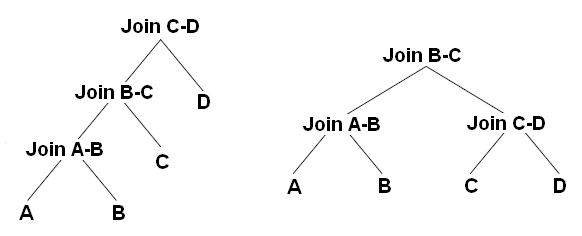
Možemo naterati optimizator upita da prati redosled spojeva koji je specificiran u upitu. Ovo se obavlja korišćenjem nagoveštaja (*hint*):

OPTION (FORCE ORDER)

Proverom plana izvršenja možemo se uveriti da redosled spojeva prati redosled spojeva u upitu. Pošto opitimizator ne može da razmotri sve moguće redoslede spojeva, već samo određeni podskup, nagoveštajem možemo nekada postići bolje performanse.

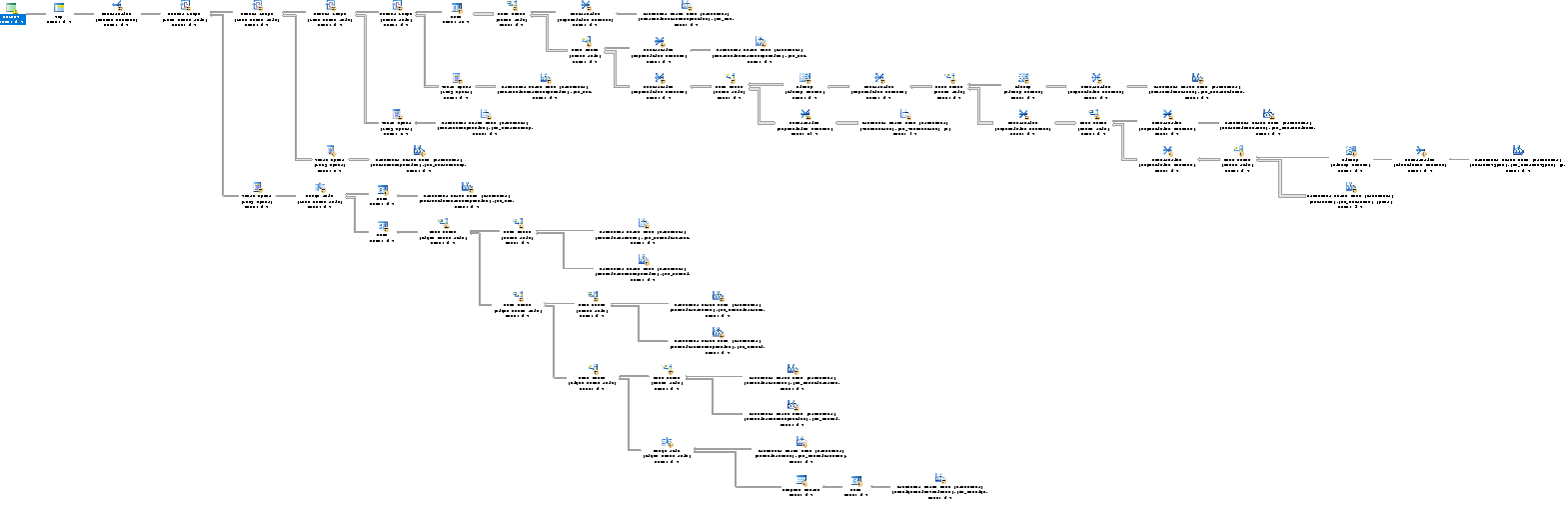
Za slučaj analiziranog SQL upita, dodavanjem opcije FORCE ORDER za pribavljanje prve stranice podataka dobija se ubrzanje od 15%.

Pošto optimizator ne može da razmotri svaki redosled spojeva, on koristi razne heuristike, kao što je oblik stabla upita, da suzi skup redosleda. Neka od ovih stabala imaju imena, kao što su *left-deep*, *right-deep* i *bushy* stabla.



|  |  |  |
| --- | --- | --- |
| **Broj tabela u spoju** | **Left-Deep Stabla** | **Bushy Stabla** |
| 1 | 1 | 1 |
| 2 | 2 | 2 |
| 3 | 6 | 12 |
| 4 | 24 | 120 |
| 5 | 120 | 1,680 |
| 6 | 720 | 30,240 |
| 7 | 5040 | 665,280 |
| 8 | 40,320 | 17,297,280 |
| 9 | 362,880 | 518,918,400 |
| 10 | 3,628,800 | 17,643,225,600 |
| 11 | 39,916,800 | 670,442,572,800 |
| 12 | 479,001,600 | 28,158,588,057,600 |

Očigledno je da u našem slučaju optimizator upita ne uzima sve redoslede spojeva u razmatranje, pa mora da nađe balans između vremena za optimizaciju i kvaliteta plana izvršenja.

Test plan modifikovanog SQL upita je dat na sledećoj slici:

# Dodatna optimizacija upita kod *paging*-a

Kod *paging*-a se može iskoristiti tehnika optimizacija upita, gde se u prvom delu upita pribavljaju primarni ključevi željenih tabela, dok se u drugom delu upita pribavljaju konkretni podaci na osnovu izvučenih primarnih ključeva.

Ova tehnika najviše povećava performance upita kada se upit koristi prilikom *paging*-a i kada upit selektuje mnogo kolona.

Pribavljanje prve stranice, pomoću SQL upita opisanog u prilogu B, se ovom tehnikom može ubrzati za 30%.

Ubrzanje ovom tehnikom je sve veće kako raste redni broj zahtevane stranice. Tako npr. kod 26. stranice dobija se ubrzanje od 34%.

A ubrzanje kod 1251. stranice je 200%, tj. ovom tehnikom se upit izvršava 3 puta brže.

Ova tehnika se može primeniti i kada se podaci prikupljaju preko *view*-a, mada su ubrzanja manja nego kod klasičnog SQL upita, ubrzanja i dalje postoje.

Ubrzanje kod 1251. stranice korišćenjem *view*-a je 113%.

Tehnika je detaljnije opisana na sajtu:

<https://sqlperformance.com/2015/01/t-sql-queries/pagination-with-offset-fetch>

U **prilogu C** se nalazi SQL upit koji koristi ovu tehniku, dok se u **prilogu D** nalazi upit koji koristi *view*.

# Views

Pored klasičnih *view*-eva, SQL server podržava jos 3 različite vrste *view*-a:

* Indexed Views
* Partitioned Views
* System Views

*System view* se koristi za meta podatke SQL servera, a *partitioned view* se koristi kod distribuiranih baza, gde se različiti delovi podataka nalaze na različitim SQL serverima. Stoga, ova dva tipa *view*-a se ne mogu koristiti za dati upit.

## Standardni View

Pribavljanje podataka pomoću klasičnog SELECT upita nad *view*-om ima slabije performance nego izvršavanje modifikovanog SQL upita. Kod modifikovanog SQL upita imamo razne tehnike ubrzanja koje nisu iskorišćene kod View-a (npr. pribavljanje prvo primarnih ključeva pa kasniji spoj za pribavljanje podataka, *force*-ovanje rasporeda *Query* *Optimizer*-u itd.).

Kada se ove tehnike primene i nad *view*-om, tada su razlike u performansama zanemarljive.

To je zato što standardni *view* koristi SQL upit u pozadini (kojim je i generisan) prilikom pribavljanja podataka, a taj upit se ne može dalje ubrzati bez indeksa.

## Indexed View

*Indexed View* se kreira tako što se standardnom *view*-u dodaje jedan jedinstveni klasterovani indeks. Takođe, *view* mora biti kreiran pomoću „WITH SCHEMABINDING“ opcije.

*Indexed View* ima velika ograničenja prilikom generisanja, neka od tih ograničenja su da SELECT upit ne sme sadržati:

* COUNT, DISTINCT, ROWSET, MIN, MAX, ORDER BY, TOP, OFFSET
* *Join*-ovanje istih tabela više puta (*self-joins*)
* OUTER *join*-ove (LEFT, RIGHT, FULL)
* *Common table expressions*
* Itd.

Detaljnije informacije o tome kako se kreira *Indexed View* kod SQL Servera se mogu naći na sajtu:

<https://msdn.microsoft.com/en-us/library/ms191432.aspx>

# Indeksi

Loše projektovani indeksi, ili nedostatak istih, je obično glavni uzrok loših performansi. U sledećoj tabeli (preuzetoj sa sajta <https://msdn.microsoft.com/en-us/library/ms175049.aspx>), dat je pregled podržanih tipova indeksa u SQL Serveru:

|  |  |
| --- | --- |
| **Tip Indeksa** | **Opis** |
| Hash | With a hash index, data is accessed through an in-memory hash table. Hash indexes consume a fixed amount of memory, which is a function of the bucket count. |
| memory-optimized nonclustered indexes | For memory-optimized nonclustered indexes, memory consumption is a function of the row count and the size of the index key columns |
| Clustered | A clustered index sorts and stores the data rows of the table or view in order based on the clustered index key. The clustered index is implemented as a B-tree index structure that supports fast retrieval of the rows, based on their clustered index key values. |
| Nonclustered | A nonclustered index can be defined on a table or view with a clustered index or on a heap. Each index row in the nonclustered index contains the nonclustered key value and a row locator. This locator points to the data row in the clustered index or heap having the key value. The rows in the index are stored in the order of the index key values, but the data rows are not guaranteed to be in any particular order unless a clustered index is created on the table. |
| Unique | A unique index ensures that the index key contains no duplicate values and therefore every row in the table or view is in some way unique.  Uniqueness can be a property of both clustered and nonclustered indexes. |
| Columnstore | An in-memory columnstore index stores and manages data by using column-based data storage and column-based query processing.  Columnstore indexes work well for data warehousing workloads that primarily perform bulk loads and read-only queries. Use the columnstore index to achieve up to **10x query performance** gains over traditional row-oriented storage, and up to **7x data compression** over the uncompressed data size. |
| Index with included columns | A nonclustered index that is extended to include nonkey columns in addition to the key columns. |
| Index on computed columns | An index on a column that is derived from the value of one or more other columns, or certain deterministic inputs. |
| Filtered | An optimized nonclustered index, especially suited to cover queries that select from a well-defined subset of data. It uses a filter predicate to index a portion of rows in the table. A well-designed filtered index can improve query performance, reduce index maintenance costs, and reduce index storage costs compared with full-table indexes. |
| Spatial | A spatial index provides the ability to perform certain operations more efficiently on spatial objects (*spatial data*) in a column of the **geometry** data type. The spatial index reduces the number of objects on which relatively costly spatial operations need to be applied. |
| XML | A shredded, and persisted, representation of the XML binary large objects (BLOBs) in the **xml** data type column. |
| Full-text | A special type of token-based functional index that is built and maintained by the Microsoft Full-Text Engine for SQL Server. It provides efficient support for sophisticated word searches in character string data. |

Indeks je struktura na disku, koja je vezana za neku tabelu ili *view* i ubrzava pribavljanje podataka iz te tabele. Indeks sadrži ključeve koji predstavljaju vrednosti iz jedne ili više kolona u tabeli ili *view*-u. Ovi ključevi se čuvaju u strukturi podataka B-stablo.

Odabir pravog indeksa za bazu, i upite na koje treba da odgovori, predstavlja složeni kompromis između brzine izvršenja upita i cene *update* operacije. Generalno, treba pratiti sledeće savete:

* Veliki broj indeksa nad nekom tabelom utiče na performanse INSERT, UPDATE, DELETE i MERGE operacija, jer svi indeksi moraju biti prilagođeni novonastalom stanju u tabeli.
* Kreiranje indeksa nad malom tabelom ne mora biti optimalno, jer može biti potrebno više vremena da se obiđe indeks prilikom traženja podatka, nego prostim skeniranjem.
* Indeksi nad *view*-evima mogu značajno povećati performanse kada *view* sadrži agregacije, spojeve ili njihovu kombinaciju. *View* ne mora biti eksplicitno referenciran u upitu, da bi ga optimizator iskoristio.
* Dobri kandidati za neklasterovane indekse su kolone koje učestvuju u spojevima i predikatima.
* Treba iskoristi **Database Engine Tuning Advisor**, koji daje predlog indeksa koje treba kreirati.

Pošto smo ograničeni na jedan klasterovani indeks po tabeli, a svaka tabela već ima kreiran klasterovani indeks nad primarnim ključem, odlučili smo da dodamo neklasterovani indeks nad kolonama koje učestvuju u spojevima problematičnog upita.

CREATE INDEX IX\_Product\_SerialNumber ON Products (SerialNumber);

CREATE INDEX IX\_TestResult\_ProductSerial on TestResults (ProductSerial);

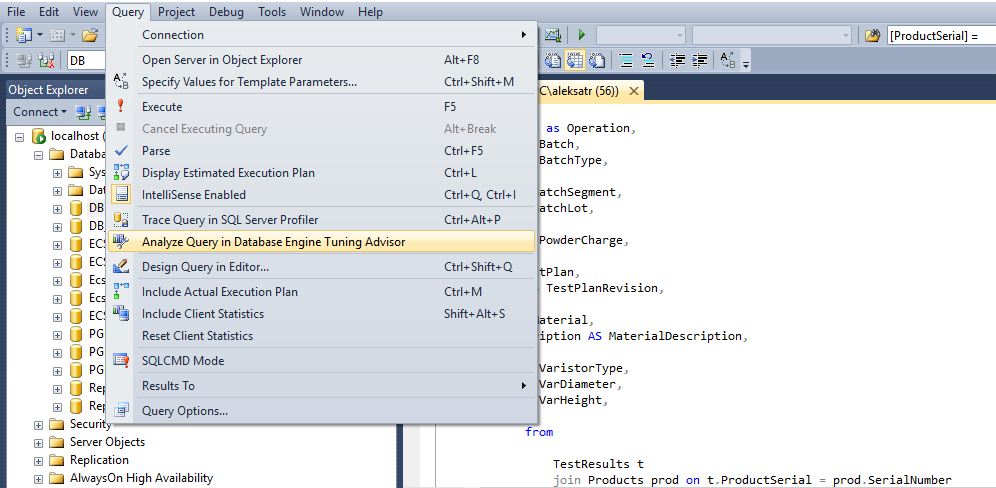
CREATE INDEX IX\_ProductProperty\_Product ON ProductProperties (ProductId);

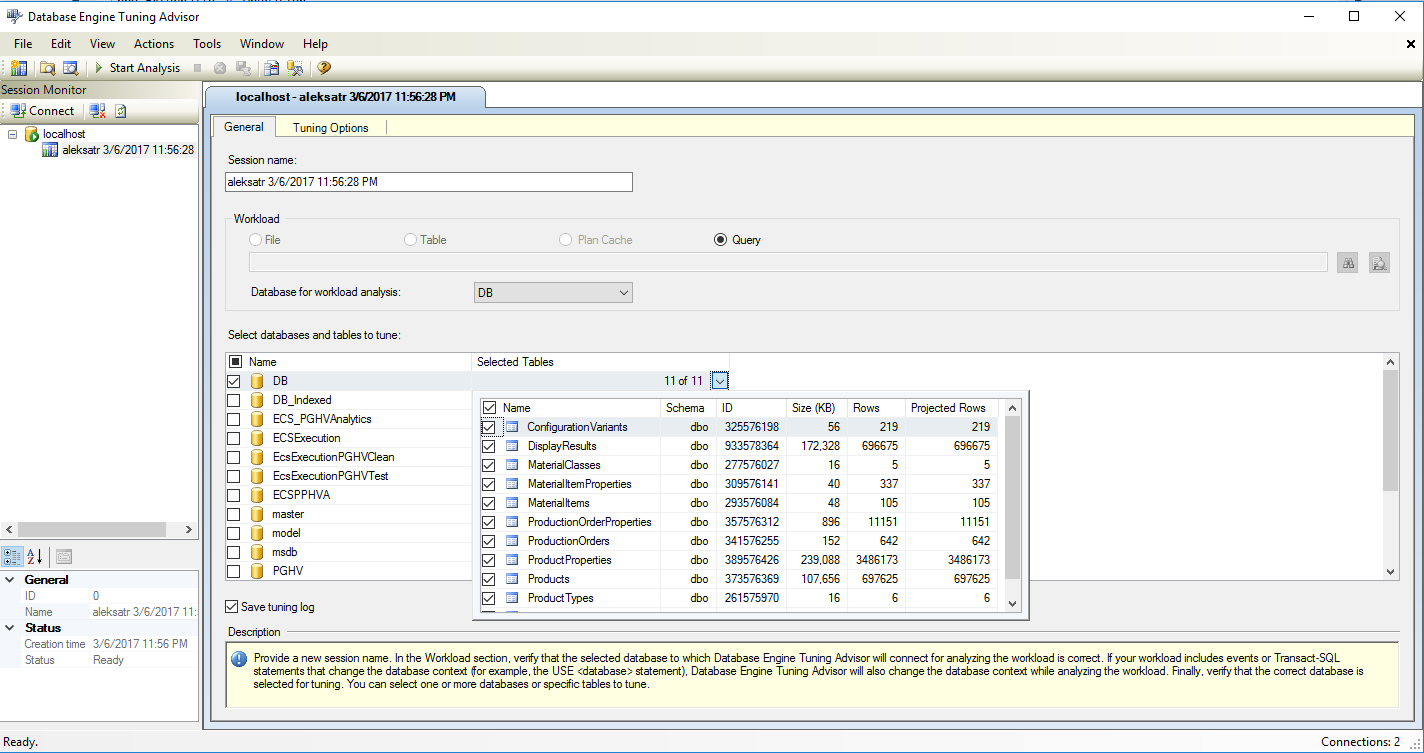
CREATE INDEX IX\_ProductionOrderProperty\_ProductionOrder ON ProductionOrderProperties (ProductionOrderId);

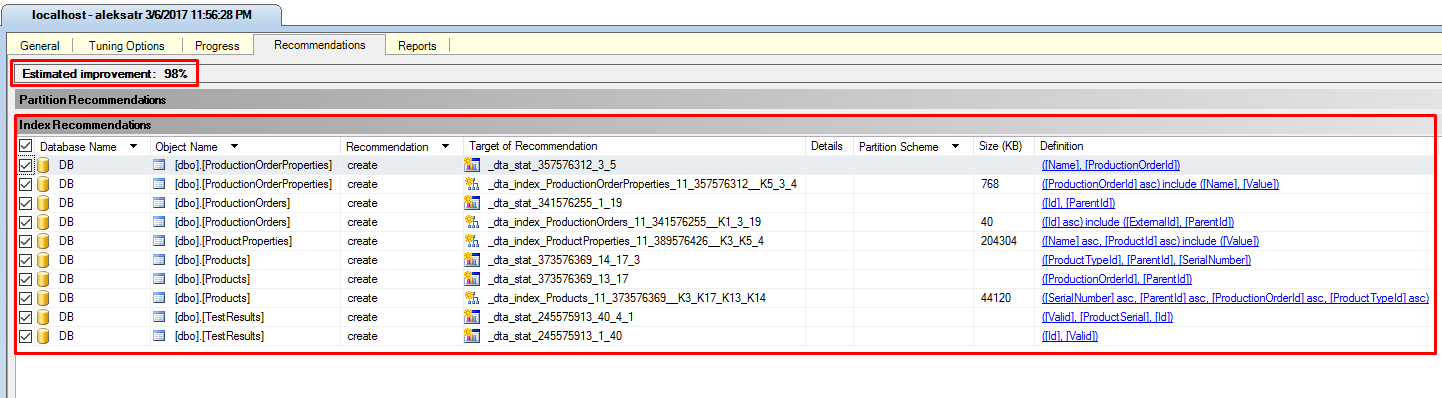
CREATE INDEX IX\_DisplayResult\_TestResult ON DisplayResults (TestResultId);

Sa kreiranih indeksima za pribavljanje prve stranice imamo ubrzanje od 1300%, za pribavljanje 26. stranice imamo ubrzanje 275%, dok su performanse za pribavljanje 1251. stranice identične.

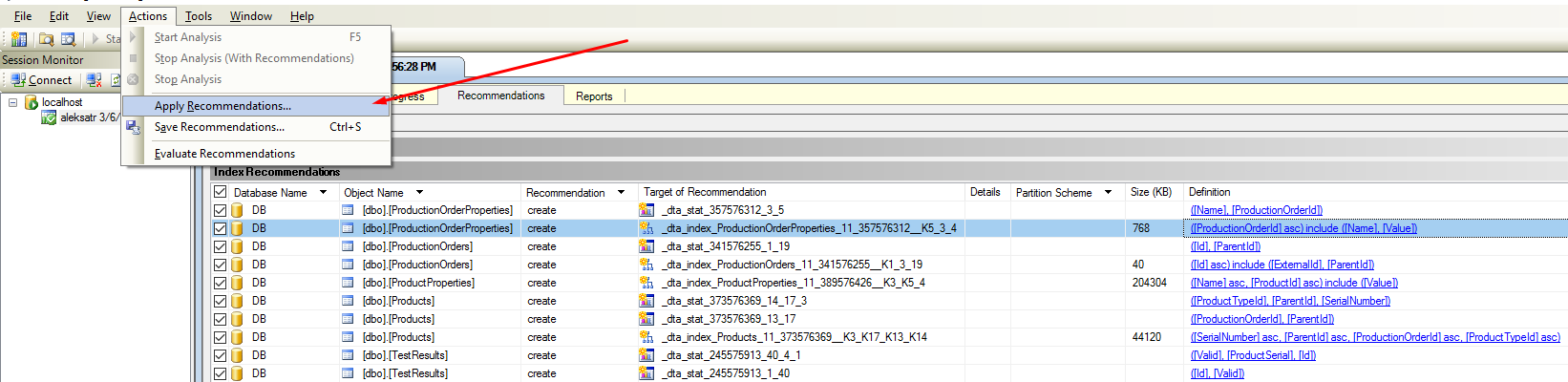
# Predlog Tuning Advisor-a







Tuning advisor je predložio više indeksa i statistika za kreiranje, koje možemo jednostavno primeniti:



Primer kompozitnog neklasterovanog indeksa koji *Tuning Advisor* predlaže uključuje dodatnu ne-ključnu kolonu (*Value*):

CREATE NONCLUSTERED INDEX [\_dta\_index\_ProductProperties\_11\_389576426\_\_K3\_K5\_4] ON [dbo].[ProductProperties]

(

[Name] ASC,

[ProductId] ASC

)

INCLUDE ([Value]) WITH (SORT\_IN\_TEMPDB = OFF, DROP\_EXISTING = OFF, ONLINE = OFF) ON [PRIMARY]

# Prilog A

SELECT distinct

ppo.ExternalId as Operation,

pop1.Value AS Batch,

pop2.Value AS BatchType,

pp6.Value AS BatchSegment,

pp7.Value AS BatchLot,

pop3.Value AS PowderCharge,

cv.Name AS TestPlan,

cv.Revision AS TestPlanRevision,

cvMI.code AS Material,

cvMI.ItemDescription AS MaterialDescription,

mip1.Value AS VaristorType,

mip2.Value AS VarDiameter,

mip3.Value AS VarHeight,

TestResult.\*

FROM

(

select \*

FROM

(

SELECT ROW\_NUMBER() OVER (order by temp.ProductSerial) AS rn, temp.\*

FROM

(

Select distinct top 100 percent TestResults.\*

from

Products

join ProductTypes on Products.ProductTypeId = ProductTypes.Id and ProductTypes.Name = 'Varistor'

join TestResults on Products.SerialNumber = TestResults.ProductSerial and TestResults.Valid = 1,

ProductionOrders,

ProductionOrderProperties

WHERE

ProductionOrderProperties.Name = 'MOBatch'

and

Products.ProductionOrderId = ProductionOrders.Id

and

ProductionOrders.Id = ProductionOrderProperties.ProductionOrderId

order by TestResults.ProductSerial asc

) as temp

) as temp2

where temp2.rn between {0} and {1}

) TestResult

left join Products p on TestResult.ProductSerial = p.SerialNumber

left join ProductionOrders po on po.Id = p.ProductionOrderId

left join ProductionOrders ppo on ppo.Id = po.ParentId

-- GET TESTPLAN ID

left join ProductionOrderProperties popTestPlan on po.Id = popTestPlan.ProductionOrderId and popTestPlan.Name = 'TestPlanId'

-- GET DATA FROM PRODUCTION ORDER PROPERTY TABLE

left join ProductionOrderProperties pop1 on po.ParentId = pop1.ProductionOrderId and pop1.Name = 'MOBatch'

left join ProductionOrderProperties pop2 on po.ParentId = pop2.ProductionOrderId and pop2.Name = 'ProductionVersion'

left join ProductionOrderProperties pop3 on po.ParentId = pop3.ProductionOrderId and pop3.Name = 'PowderCharge'

-- GET DATA FROM PRODUCT PROPERTY TABLE

left join ProductProperties pp6 on p.ParentId = pp6.ProductId and pp6.Name = 'SegmentName'

left join ProductProperties pp7 on p.ParentId = pp7.ProductId and pp7.Name = 'LayerName'

-- GET CONFIGURATION VARIANT BASED ON FETCHED TESTPLAN ID

left join ConfigurationVariants cv on CAST (popTestPlan.Value as bigint) = cv.Id

left join MaterialItems cvMI on cv.MaterialItemId = cvMI.id

-- GET DATA FROM MATERIAL ITEM PROPERTY TABLE

left join MaterialItemProperties mip1 on cvMI.id = mip1.MaterialItemId and mip1.MaterialClassId in (select Id from MaterialClasses where Name = 'Var\_Typ')

left join MaterialItemProperties mip2 on cvMI.id = mip2.MaterialItemId and mip2.MaterialClassId in (select Id from MaterialClasses where Name = 'Diameter')

left join MaterialItemProperties mip3 on cvMI.id = mip3.MaterialItemId and mip3.MaterialClassId in (select Id from MaterialClasses where Name = 'Height')

# Prilog B

Select

ppo.ExternalId as Operation,

pop1.Value AS Batch,

pop2.Value AS BatchType,

pp6.Value AS BatchSegment,

pp7.Value AS BatchLot,

pop3.Value AS PowderCharge,

cv.Name AS TestPlan,

cv.Revision AS TestPlanRevision,

cvMI.code AS Material,

cvMI.ItemDescription AS MaterialDescription,

mip1.Value AS VaristorType,

mip2.Value AS VarDiameter,

mip3.Value AS VarHeight, t.\*

from

TestResults t

join Products prod on t.ProductSerial = prod.SerialNumber

join ProductTypes pt on pt.Id = prod.ProductTypeId and pt.Name = 'Varistor'

join ProductionOrders prodOrder on prodOrder.Id = prod.ProductionOrderId

join ProductionOrders ppo on ppo.Id = prodOrder.ParentId

join ProductionOrderProperties pop1 on pop1.ProductionOrderId = prodOrder.Id and pop1.Name = 'MOBatch'

join ProductionOrderProperties pop2 on pop2.ProductionOrderId = prodOrder.Id and pop2.Name = 'ProductionVersion'

join ProductionOrderProperties pop3 on pop3.ProductionOrderId = prodOrder.Id and pop3.Name = 'PowderCharge'

left join ProductProperties pp6 on (pp6.ProductId = prod.ParentId and pp6.Name = 'SegmentName')

left join ProductProperties pp7 on (pp7.ProductId = prod.ParentId and pp7.Name = 'LayerName')

left join ProductionOrderProperties TestPlan on prodOrder.Id = TestPlan.ProductionOrderId and TestPlan.Name = 'TestPlanId'

left join ConfigurationVariants cv on CAST (TestPlan.Value as bigint) = cv.Id

left join MaterialItems cvMI on cv.MaterialItemId = cvMI.id

left join MaterialItemPropertIes mip1 on cvMI.id = mip1.MaterialItemId and mip1.MaterialClassId in (select Id from MaterialClasses where Name = 'Var\_Typ')

left join MaterialItemPropertIes mip2 on cvMI.id = mip2.MaterialItemId and mip2.MaterialClassId in (select Id from MaterialClasses where Name = 'Diameter')

left join MaterialItemProperties mip3 on cvMI.id = mip3.MaterialItemId and mip3.MaterialClassId in (select Id from MaterialClasses where Name = 'Height')

WHERE

t.valid = 1

{2}

order by {3} t.Id

OFFSET {0} ROWS

FETCH NEXT {1} ROWS ONLY;

# Prilog C

with TestIds as (Select

t.Id as TestResultId,

ppo.Id as OperationId,

pop1.Id AS BatchId,

pop2.Id AS BatchTypeId,

pp6.Id AS BatchSegmentId,

pp7.Id AS BatchLotId,

pop3.Id AS PowderChargeId,

cv.Id AS TestPlanId,

cvMI.Id AS MaterialId,

mip1.Id AS VaristorTypeId,

mip2.Id AS VarDiameterId,

mip3.Id AS VarHeightId

from

TestResults t

join Products prod on t.ProductSerial = prod.SerialNumber

-- join ProductTypes pt on pt.Id = prod.ProductTypeId and pt.Name = 'Varistor'

join ProductionOrders prodOrder on prodOrder.Id = prod.ProductionOrderId

join ProductionOrders ppo on ppo.Id = prodOrder.ParentId

left join ProductionOrderProperties TestPlan on prodOrder.Id = TestPlan.ProductionOrderId and TestPlan.Name = 'TestPlanId'

-- GET CONFIGURATION VARIANT BASED ON FETCHED TESTPLAN ID

left join ConfigurationVariants cv on CAST (TestPlan.Value as bigint) = cv.Id

left join MaterialItems cvMI on cv.MaterialItemId = cvMI.id

-- GET DATA FROM MATERIAL ITEM PROPERTY TABLE

left join MaterialItemPropertIes mip1 on cvMI.id = mip1.MaterialItemId and mip1.MaterialClassId in (select Id from MaterialClasses where Name = 'Var\_Typ')

left join MaterialItemPropertIes mip2 on cvMI.id = mip2.MaterialItemId and mip2.MaterialClassId in (select Id from MaterialClasses where Name = 'Diameter')

left join MaterialItemProperties mip3 on cvMI.id = mip3.MaterialItemId and mip3.MaterialClassId in (select Id from MaterialClasses where Name = 'Height')

join ProductionOrderProperties pop1 on pop1.ProductionOrderId = prodOrder.Id and pop1.Name = 'MOBatch'

join ProductionOrderProperties pop2 on pop2.ProductionOrderId = prodOrder.Id and pop2.Name = 'ProductionVersion'

join ProductionOrderProperties pop3 on pop3.ProductionOrderId = prodOrder.Id and pop3.Name = 'PowderCharge'

left join ProductProperties pp6 on (pp6.ProductId = prod.ParentId and pp6.Name = 'SegmentName')

left join ProductProperties pp7 on (pp7.ProductId = prod.ParentId and pp7.Name = 'LayerName')

WHERE

t.valid = 1

{2}

order by {3} t.Id

OFFSET {0} ROWS

FETCH NEXT {1} ROWS ONLY)

Select

ppo.ExternalId as Operation,

pop1.Value AS Batch,

pop2.Value AS BatchType,

pp6.Value AS BatchSegment,

pp7.Value AS BatchLot,

pop3.Value AS PowderCharge,

cv.Name AS TestPlan,

cv.Revision AS TestPlanRevision,

cvMI.code AS Material,

cvMI.ItemDescription AS MaterialDescription,

mip1.Value AS VaristorType,

mip2.Value AS VarDiameter,

mip3.Value AS VarHeight,

t.\*

from

TestIds ids

join TestResults t on t.Id = ids.TestResultId

join ProductionOrders ppo on ppo.Id = ids.OperationId

join ProductionOrderProperties pop1 on pop1.Id = ids.BatchId

join ProductionOrderProperties pop2 on pop2.Id = ids.BatchTypeId

join ProductionOrderProperties pop3 on pop3.Id = ids.PowderChargeId

left join ProductProperties pp6 on (pp6.Id = ids.BatchSegmentId)

left join ProductProperties pp7 on (pp7.Id = ids.BatchLotId )

left join ConfigurationVariants cv on cv.Id = ids.TestPlanId

left join MaterialItems cvMI on cvMI.Id = ids.MaterialId

left join MaterialItemPropertIes mip1 on mip1.Id = ids.VaristorTypeId

left join MaterialItemPropertIes mip2 on mip2.Id = ids.VarDiameterId

left join MaterialItemProperties mip3 on mip3.Id = ids.VarHeightId

order {3} by t.Id

OPTION (FORCE ORDER)

# Prilog D

with pg as (SELECT

p.id

FROM PaggingView p

{2}

order by {3} p.id

offset {0} rows fetch next {1} rows only)

select

p.\*

from PaggingView p

inner join pg on pg.id = p.id

order by {3} p.Id;

# Prilog E

## *Entity Framework*

Isti upit korišćenjem ORM-a i LINQ to SQL sintakse:

var validTestResults = dbModel.TestResults.Where(tr => tr.Valid);

var varistorProducts = dbModel.Products.Where(p => p.ProductType.Name == "Varistor");

var q1 = varistorProducts.Join(validTestResults, p => p.SerialNumber, tr => tr.ProductSerial

, (p, tr) => new { Product = p, TestResult = tr, TestPlanId = p.ProductionOrder.ProductionOrderProperties.FirstOrDefault(pop => pop.Name == "TestPlanId") });

var q2 = q1.Select(a => new {

Product = a.Product

, TestResult = a.TestResult

, TestPlanId = a.TestPlanId

, ConfigurationVariant = (a.TestPlanId == null) ? null : dbModel.ConfigurationVariants.FirstOrDefault(c => c.Id.ToString() == a.TestPlanId.Value)

, Batch = a.Product.ProductionOrder.ProductionOrderProperties.FirstOrDefault(pop => pop.Name == "MOBatch").Value

, BatchType = a.Product.ProductionOrder.ProductionOrderProperties.FirstOrDefault(pop => pop.Name == "ProductionVersion").Value

, BatchSegment = a.Product.Parent.ProductProperties.FirstOrDefault(pp => pp.Name == "SegmentName").Value

, BatchLot = a.Product.Parent.ProductProperties.FirstOrDefault(pp => pp.Name == "LayerName").Value

, PowderCharge = a.Product.ProductionOrder.ProductionOrderProperties.FirstOrDefault(pop => pop.Name == "PowderCharge").Value

});

var q3 = q2.Select(a => new {

Product = a.Product

, TestResult = a.TestResult

, TestPlanId = a.TestPlanId

, ConfigurationVariant = a.ConfigurationVariant

, Batch = a.Batch

, BatchType = a.BatchType

, BatchSegment = a.BatchSegment

, BatchLot = a.BatchLot

, PowderCharge = a.PowderCharge

, mip1 = (a.ConfigurationVariant == null || a.ConfigurationVariant.MaterialItem == null) ? null : a.ConfigurationVariant.MaterialItem.MaterialItemProperties.FirstOrDefault(p => p.MaterialClass.Name == "Var\_Typ")

, mip2 = (a.ConfigurationVariant == null || a.ConfigurationVariant.MaterialItem == null) ? null : a.ConfigurationVariant.MaterialItem.MaterialItemProperties.FirstOrDefault(p => p.MaterialClass.Name == "Diameter")

, mip3 = (a.ConfigurationVariant == null || a.ConfigurationVariant.MaterialItem == null) ? null : a.ConfigurationVariant.MaterialItem.MaterialItemProperties.FirstOrDefault(p => p.MaterialClass.Name == "Height")

});

query = q3.OrderBy(a => a.Product.Id).Select(a => new

{

Operation = a.Product.ProductionOrder.Parent.ExternalId

, Batch = a.Batch

, BatchType = a.BatchType

, BatchSegment = a.BatchSegment

, BatchLot = a.BatchLot

, PowderCharge = a.PowderCharge

, TestPlan = a.ConfigurationVariant == null ? null : a.ConfigurationVariant.Name

, TestPlanRevision = a.ConfigurationVariant == null ? null : a.ConfigurationVariant.Revision

, Material = a.ConfigurationVariant == null ? null : a.ConfigurationVariant.MaterialItem.Code

, MaterialDescription = a.ConfigurationVariant == null ? null : a.ConfigurationVariant.MaterialItem.ItemDescription

, VaristorType = a.mip1 == null ? null : a.mip1.Value

, VarDiameter = a.mip2 == null ? null : a.mip2.Value

, VarHeight = a.mip3 == null ? null : a.mip2.Value

, TestResult = a.TestResult

});

var page = query.Skip(requestedPageNumber \* pageSize).Take(pageSize).ToArray();

# Literatura

* <https://msdn.microsoft.com/en-us/library/ff650689.aspx>
* <https://msdn.microsoft.com/en-us/library/ms190174.aspx>
* <https://msdn.microsoft.com/en-us/library/ms191432.aspx>
* <https://technet.microsoft.com/en-us/library/dd171921(v=sql.100).aspx>
* <https://sqlperformance.com/2015/01/t-sql-queries/pagination-with-offset-fetch>
* <https://msdn.microsoft.com/en-us/library/ms181714.aspx>
* <https://msdn.microsoft.com/en-us/library/ms188709.aspx>
* <https://technet.microsoft.com/en-us/library/ms176005(v=sql.105).aspx>
* <http://www.sql-server-performance.com/2013/sql-server-query-optimization-part-1/>
* <https://blog.sqlauthority.com/2013/08/28/sql-server-tips-for-sql-query-optimization-by-analyzing-query-plan/>
* <https://www.simple-talk.com/sql/sql-training/the-sql-server-query-optimizer/>
* <https://technet.microsoft.com/en-us/library/jj835095(v=sql.110).aspx>