**Optimalisatie dossier: Catchem**

Dit dossier beschrijft de optimalisatie van het database in het kader van het Catchem project. We gaan in detail in op de doorgevoerde optimalisaties en de gerealiseerde efficiëntie winst. Verder lichten we enkele alternatieve optimalisaties toe die niet werden gekozen. De belangrijkste test parameters die gebruikt werden zijn: I/O kost, CPU gebruik en loop tijd van de query’s. We starten dit dossier met het toelichten van de methodologie.

Zijn er parameters voor de database engine die ik kan wijzigen? Welk type indexen gebruik ik waar? Welke datatypen zijn optimaal? Kan ik best memory optimized tables inzetten, en waar? Is het nuttig om op bepaalde tabellen partitionering toe te passen? Is het nuttig om op bepaalde tabellen compressie toe te passen? Welke optimalisaties die de docent besproken heeft zouden nuttig kunnen zijn?

**Methodologie**

* Een verandering per keer, want het kan verbonden zijn aan elkaar
* Identificeer de bottlenecks (en kwantificeer deze eerst) zo kan je je tijd efficiënt gebruiken. Bv eerst de problemen die een 80% in tijd kunnen reduceren
  + Je moet steeds een kosten baten doen: is de investering van tijd het wel waard?

De gebruikte methode bestond erin om de stroom van gegevens waarachtig te simuleren. Hiervoor werden de meest gebruikte *queries* geschreven om de testen op uit te voeren (zie queries op: github link). We hebben hier steeds gekozen om maximale variatie te simuleren. Als startpunt werden deze queries geanalyseerd door middel van de *database tuning advisor* in de SQL management studio. Dit werd eerder als startpunt gebruikt voor verder optimalisatie.

Er werd rekening gehouden met verschillende parameters.

* % gerapporteerd (op basis van wat is dit berekend?)
* I/O cost
* CPU gebruik
* Run time

Discussie: welke metrics zijn het beste om in het oog te houden? En gegeven dit welke kiezen we hier om mee te nemen. Het zou goed zijn om ook een soort van ‘overal’ improvement te kunnen uitdrukken.

Side note: wat is nu optimaal?

* Eerst de indexes
  + Je wilt een seek, geen scan (want dat zorgt voor meer logical reads)
  + Even though it is true that the amount of overhead required to maintain indexes increases for data manipulation queries, be aware that SQL Server must first find a row before it can update or delete it; therefore, indexes can be helpful for UPDATE and DELETE statements with necessary WHERE clauses. The increased efficiency in using the index to locate a row usually offsets the extra overhead needed to update the indexes, unless the table has a lot of indexes.
    - No index = een heap
    - Clustered index = zoals een dictionaire
    - Non-clustered index = zoals een index in een boek
      * Let wel, niet teveel indexen (geheugen kost)
      * En het wegschrijven van resultaten kost meer doordat er meerdere updates dienen te gebeuren
      * Voordeel: kan in ander geheugen gestoken worden
    - Composite vs non-composite: Furthermore, while creating an index on multiple columns, which is also referred to as a composite index, column order matters. In many cases, using the most selective column first will help filter the index rows more efficiently
      * Dit kan bv gedaan worden als je op basis van Gender iets opzoekt, als er maar 2 waarden zijn, halt een index niet veel uit (niet selectief genoeg), een gewone PK (clustered, scan) werkt dan ook niet efficient. Het zou dan wel beter zijn om een composiete index te gebruiken.
      * (bij composieten) Het beste is om volledige overeenstemming te hebben in de volgorde van de index en de query. Als er city + postcode in de index staat moet je in de query city + postcode zetten (in die volgorder) voor de optimale query. Omdraaien is het slechtste. Waar enkel city vragen de 2de beste optie is.
    - Indexes on integers are always better
    - Er is een relatie tussen een clustered en non clustered index. In elke non-clusterede index staat er een verwijzijng naar de clustered index (die zoekt de waarde op in de index en geeft de rowID terug).
      * Creeer de clustered indexen eerst (dan staan ze al goed in de non-clustered indexen ipv aan te passen
      * Oppassen met de grote van de clustered indexen (want worden gekopieerd in non-clustered)
    - Take into account the ordering of data (should match again the order for efficency gains)
    - Bij een clustered is de toegang direct en complete (1 lookup: alle data), dit is niet zo bij een non-clustered(tenzij het een covering of composiete non-cluster is).
    - VRAAG: is het wel zo efficent om de PK clustered te maken: zijn er andere rijen (of combinatie van rijen) die hier beter aan zouden voldoen? Onze PK’s zijn groot en hebben veel info nodig om ze te gebruiken in nonclustered indexen.
    - A nonclustered index on a frequently updatable column isn’t as costly as having a clustered index on that column. The UPDATE operation on a nonclustered index is limited to the base table and the nonclustered index. It doesn’t affect any other nonclustered indexes on the table. Similarly, a nonclustered index on a wide column (or set of columns) doesn’t increase the size of any other index, unlike that with a clustered index. However, remain cautious, even while creating a nonclustered index on a highly updatable column or a wide column (or set of columns), since this can increase the cost of action queries, as explained earlier in the chapter
* Dan de vorm van de queries zelf (er moet steeds een hoge selectiviteit zijn in de where/join clauses)
* Covering key (including) and actually joining the columns to form a key is different
* Take into account the combination of indexes (when the nonclustered is already wide, and cannot include or add or when there are dependencies), you can add a separate index on the column & sql will use both

**Check chapter 18, 19 and 24 certainly!!**

**Check:**

* **Waar zijn de indexes opgeslagen? Zetten op een aparte schijf?**
* **Check of alle indexen op 1/ al dan niet op meerdere 8kb pages zijn opgeslagen. GUID zijn lang 16 bytes, en zijn dus misschien niet het meeste geschikt (p119)**
* **Non-clustered index on PK ipv een clustered index? (omwille van grote van sleutels?) + check of een andere niet beter clustered kan zijn**
  + **Je moet de meest gebruikte access path gebruiken om de clustered index op te zetten (is dit wel de ID?)**
* **Check full tekst**
* **And check spatial indexes!**
* **Make view of number of sequences?**

**dbo. stage**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **ID** | **Container\_size** | **Description** | **Lat** | **Long** | **Sequence\_number** | **Type** | **Visibility** |
| 0x0000 | 0 | Officiis sit ad exercitationem ut et. | 34,4742887903853 | 135,584906933211 | 6 | 0 | 0 |
| **CLUSTERED** |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

Op dbo.stage worden in essentie 3 queries uitgevoerd. Ten eerste is de creatie van de stages (simpel insert statement, op basis waarvan daarna entries aan dbo.treasure\_stages worden toegevoegd). Ten tweede kan een gebruiker bij het zoeken van treasures filteren op het aantal stages (maar dit hoort eigenlijk bij dbo.treasure\_stages). Ten derde kan er een opzoek query gebeuren van de beschrijvingen van de stages die bij een bepaalde treasure horen. Dit laatste dient te gebeuren op basis van de visbility.

Index strategie. Op deze tabel dient geen verdere indexeren worden toegepast (optimialisatie van deze query gebeurde door dbo\_treasure\_stages te indexeren).

* Baseline: 12 reads on stage & cpu time 15ms & elapsed time 85ms cpu
* Niet toegepast,
  + verwisselen van clustered index op ID met een non-clustered index including the verschillende kolommen (dit is niet sneller, en niet geheugen efficient, een clustered returned ineens alles).
  + N-clustered on visibility: 12 reads &cpu time ms & elapsed time 95ms
  + N-clustered including all: 12 reads & cpu time ms & elapsed time 93ms

Vraag: is een sub-query sneller dan een join?

CHECK : filtered index op type & visibility?

**dbo.treasure**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Difficulty** | **Terrain** | **City-city\_id** | **Owner\_id** |
| 0x0000 | 3 | 1 | 0x6234 | 0xB31059 |
| **CLUSTERED** |  |  |  |  |
|  |  |  |  |  |

Er worden X aantal queries uitgevoerd op dbo.treasure. Ten eerste, het aantal beheerde treasure per user kan worden opgevraagd. Ten tweede, kunnen gebruikers zoeken op treasures waar er gefiltered kan worden op locatie (automatisch), moeilijkheid, reliëf en aantal stages. Ten derde, gebruikers kunnen treasures aanmaken.

Index strategie: (1) Een nonclustered index on owner\_ID (Base line: (total time) 138ms, 3 reads,N clustered index on owner\_id : (total time: 28ms, 6 logical reads). Verder is het mogelijk om met een indexed view de index view te gebruiken (om het aantal treasures te berekenen op voorhand). Dit zorgt voor een reductie in logical reads en parse/compile time. Maar zonder index view is het net een paar ms sneller. Het zal dus van de read/writes afhangen. (2) een non-clustered index on the city\_city\_id (3) een index view die een tabel maakt met het aantal stages al berekend

* Check whether it is possible to make a filtered index on difficulty or terrain
* Check whether you can make multiple indexes (in different order, incl)
* Check effect on write operations

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Compile time | Total time | Scan | Reads (logical) | Reads (phy) |
| Query diff (base) | 0 ms | 16/198ms | 5 | 9657 | 0 |
| Query diff (non-cl on city) | 3 ms | 0/94 ms | 1 | 915 | 0 |
| Query diff (non-cl on city, incl) | 1 ms | 0/141 ms | 1 | 5 | 0 |
| Query terrain (base) | 0 ms | 16/104ms | 5 | 9657 | 0 |
| Query terrain (non-cl on city) | 17 ms | 0/84 ms | 1 | 915 | 0 |
| Query diff (non-cl on city, incl) | 1ms | 0/93ms | 1 | 5 | 0 |
| Query stages (base) | 16 ms | 47/39ms | 294 | 1048 | 0 |
| Query stages (non-cl on city) | 8ms | 15/107ms | 294 | 973 | 0 |
| Query diff (non-cl on city, incl) | 8ms | 0/99ms | 294 | 973 | 0 |
| Quert all (base) | 0 ms | 30/160ms | 17+5 | 86+9657 | 0 |
| Quert all (non-cl on city) | 11 ms | 0/146ms | 17+1 | 74+915 | 0 |
| Query ALL (non-cl on city, incl) | 12ms | 0/149 ms | 17+1 | 74+5 | 0 |
| Query ALL (with view) | 0 | 147 ms | 1 | 6 | 0 |

* Fragmentatie strategy: eventueel deze DB fragmenteren op basis van de locatie. Een user gaat waarschijnlijk zoeken naar treasures in de buurt, het is dus niet nodig om bepaalde zaken volledig af te gaan (maar wat met internet-stages?).
* Gebruikers kunnen zoeken op treasure hunts in de buurt waarbij je kan filteren op moeilijkheid, reliëf en aantal stages
  + Je kan detail opvragen van een treasure waarbij je volgende informatie ziet:
    - de laatste X geplaatste logberichten
    - Beschrijving
    - Gegevens van de gebruiker waaronder het aantal beheerde treasures
    - Detail van de stages getoond naargelang de ingestelde zichtbaarheid van de stages.
* Uiteraard kunnen gebruikers treasures (met de daarbij horende stages) aanmaken en logs aanmaken

**dbo.treasure\_log**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **ID** | **Description** | **Log\_time** | **Log\_type** | **Session\_start** | **Hunter\_id** | **Treasure\_id** |
| 0x00000 | Laborum possimus vitae quia praesentium ut similique magni voluptatibus cum vel doloribus maxime iste commodi. | 2016-07-24 19:27:29.7400000 | 2 | 2016-07-24 19:27:13.7400000 | 0x9EA59 | 0x56A0B2C |
| **CLUSTERED** |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

Er worden X aantal queries uitgevoerd op dbo.treasure. Ten eerste, het aantal X geplaatste logberichten dat bij een bepaalde treasure hoort. Ten tweede heeft de helpdesk de mogelijkheid om logs te moderen, hiervoor moeten ze zoeken op tekst.

Index strategie. (1)Non-cluster index op treasure ID zorgt voor een sterke vermindering in zoek tijd. (2) Echter, het is ook mogelijk om de beschrijving al mee te zetten op de index, dit is iets minder reads (maar dit een write-heavy tabel, dus uitzoeken of dit wel OK is). (3) het is eventueel een idee om de clustered index te zetten op log\_time (met ID in de sleutel), dit is waarschijnlijk te groot (gezien die een write heavy db is). (4) om logs te moderen kan er een full tekst index worden gebruikt

Voor de eerste query dient de volgorde van de log\_time overeen te komen met de ‘laatst’ geplaatste berichten; we kunnen hier ook een view voor creeëren?

* Check of een view creeren bij log\_time helpt om de order by eruit te halen

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Compile time | Total time | Scan | Reads (logical) | Reads (phy) |
| Query (base) | 2 ms | 704/1599ms | 5 | 5212110 | 66 |
| Query (non-cl on treasuer\_id) | 8ms | 16/98ms | 1 | 348 | 0 |
| Query (non-cl on treasuer\_id) + incl descr | 10ms | 94ms | 1 | 293 | 3 |
| Query cluster on log\_time+ID | 1ms | 0/110ms | 1 | 8 | ! |

* **Check of het mogelijk is om alles over 1 treasure in 1 query te plaatsen (dus ook de user, logs etc).**
* **Meeste nieuwe rijen worden hier aangemaakt**
* **Partioning strategy? Deze database vraagt het X-aantal laatste op, dus we kunnen bijvoorbeeld maandelijks/jaarlijks partioneren**
  + Je kan detail opvragen van een treasure waarbij je volgende informatie ziet:
    - de laatste X geplaatste logberichten
    - Beschrijving
    - Gegevens van de gebruiker waaronder het aantal beheerde treasures
    - Detail van de stages getoond naargelang de ingestelde zichtbaarheid van de stages.
* De helpdesk heeft een scherm ter beschikking om logs te modereren en daarvoor te zoeken op de inhoud van de log beschrijvingen.

**dbo.treasure\_stages**

|  |  |
| --- | --- |
| **Tresure\_id** | **Stages\_id** |
| 0x8000 | 0x7EE8F3 |
|  | **UNIQUE non-clustered ID** |
| **CLUSTERED INDEX** |  |
|  |  |

De tabel dbo.treasure\_stages krijgt X queries te verwerken. Ten eerste, bij het aanroepen van de stages horende bij een bepaalde treasure wordt in deze tabel gezocht naar treasure\_ID.

Index strategies. Ten eerste dient er een clustered index gezet te worden op treasure\_id om snel de stages\_id te kunnen zoeken horende bij een bepaalde treasure (dit zorgt voor een drop in logical reads van X naar X en een reducering van uitvoeringstijd).

* Alternatief is een non-clustered index (omdat er maar 1 rij moet worden opgehaald), dan kan er of wel een include of een composiete index worden gemaakt).
* Letten op dependency met stages\_id (hier staat een unique constraint op)

**-** Uiteraard kunnen gebruikers treasures (met de daarbij horende stages) aanmaken en logs aanmaken

**dbo.user\_table**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Id** | **First\_name** | **Last\_name** | **Mail** | **number** | **Street** | **City\_city\_id** |
| 0x0000 | Yanis | Andre | Yanis.Andre@girard.com | 223 | Allée, Voie du Bac | 0x3DBB |
|  |  |  |  |  |  |  |

Deze tabel wordt aangeroepen met drie queries. Ten eerste wanneer de owner\_jd wordt gematched. Ten tweede wanneer de helpdesk een lijst trekt. Ten derde wanneer er een gebruiker wordt aangemaakt.

Index strategie: Als eerste regel, moet de query en de index in overeenstemming zijn. Dus de elementen die worden opgevraagd, moeten ook in de index aanwezig zijn. Er staat een clustered index op de ID, dit is ook nodig om een snelle look-up te doen voor andere tabellen. Voor een non-clustered index toe te voegen moeten er een aantal keuzes gemaakt worden. (1) een including of een composiete non-clustered index? Bij een including index is het mogelijk om een variabele mee te nemen in de index. Dit kan voorkomen dat er terug naar de fysieke tabel moet gegaan worden. Dus om voor en achternaam te verkrijgen, kan een non-clustered index op voornaam including naam, een snel resultaat geven gezien het kan weergegeven worden op basis van de tabel. Dit is dus goed voor kleine hoeveelheden data weer te geven. Bij een composiete index wordt er eerst gesorteerd op de eerste kolom en daarna op de volgende. Hier moet je rekening houden met de volgorde van het sorteren.

Het belangrijkste is dat er steeds een covering index gebruikt wordt (betekend, dat wat er in de query staat ook in de index is meegenomen). Dus moest het belangrijk zijn om op bepaalde gegevens te zoeken, dan moet hier additionele indexen op gemaakt worden. Als het enkel de naam is dan is 1 index voldoende.

* De helpdesk kan ook lijsten opvragen van gebruikers
* Bij het registreren wordt dan weer een gebruiker aangemaakt.
* **Check of de index strategy hier (ID mee in non-clust) nuttig is voor de andere.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Compile time | Total time | Scan | Reads (logical) | Reads (phy) |
| Query (base) | 25ms | 125/65ms | 5 | 25872 | 0 |
| Query (non-cl on id) incl names | 2ms | 46/129ms | 1 | 3227 | 0 |
| Query (non-cl on name) incl last name | 2ms | 0/111ms | 1 | 5 | 0 |
| Query (non-cl on name) incl first name | 2ms | 0/97ms | 1 | 5 | 0 |

* **Niet gedaan: een non-clustered index op nummer, met een query die eerst die nummer vraagt. Deze index wordt niet gebruikt.**
* **CHECK: equivalentie tussen include & composiet bij non clustered indexen.**
* **Wat met dubbele entries?**

**Optimalisatie algemeen**

De eerste algemene optimalisatie was het omvormen van de IDs.

* All ID's are stored as binary(255), meaning that the binary value takes up 255 bytes.
* This inflates the computation time (18s), I/O cost= 292.417 and CPU cost =1.97
* Solution is to store this ID as a binary, legnth 16byte. Why 16? The identifier was is an hexadecimal UUID which need 16 byte to store
* The computation time is reduced to 15s, I/O cost = 0 and CPU cost = 0.179
* This is done for all identifiers
* Remark: one possible problem with storing it binary is that different implementation may change the order of the bytes when storing. So you might also store it as a char.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Object** | **Optimalisatie** | **Metric 1** | **Metric 2** | **Metric 3** |
| IDs | ID(255) -> ID(16) |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Figuur 1: Optimalisatie algemeen

**Advantages of using a ‘narrow’ ID key**

**• Reduces I/O (by having to read fewer 8KB pages)**

**• Makes database caching more effective because SQL Server can cache fewer index pages, consequently reducing the logical reads required for the index pages in the memory • Reduces the storage space for the database**

**Optimalisatie per gebruiker**

De optimalisatie word hier gestructureerd per categorie eind gebruiker. We beginnen bij de spelers.

**Spelers**

Spelers moeten zich registeren, treasures en stages en logs aanmaken. De Tuning advisor stelt het volgende voor. Wij hebben gekozen om dit en dit mee te nemen. Maar dit en dit niet mee te nemen. Want deze redenen. Verder hebben we ook nog de volgende zaken geprobeerd waar we dit en dit hebben meegenomen want. Maar dit en dit niet want.

In conclusie zien we een totale verbetering van % in metric 1 en % in metric 2.

* Nakijken: welke join gebruikt SQL (loop, merge of hash)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Object** | **Optimalisatie** | **Metric 1** | **Metric 2** | **Metric 3** |
| Treasure ID | Index |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Figuur 2: Optimalisatie speler queries

**Help desk**

De help desk moet posts te modderen (tekst doorzoeken)**,** namen en logs van mensen opzoeken.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Object** | **Optimalisatie** | **Metric 1** | **Metric 2** | **Metric 3** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Figuur 3: Optimalisatie Helpdesk

**Software input**

* + Long/lat

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Object** | **Optimalisatie** | **Metric 1** | **Metric 2** | **Metric 3** |
| Treasure ID | Index |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Figuur 4: Optimalisatie software

**DBA**

* + onderhoud

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Object** | **Optimalisatie** | **Metric 1** | **Metric 2** | **Metric 3** |
| Treasure ID | Index |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

* Figuur 2: Optimalisatie DBA

**Rapport**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Object** | **Optimalisatie** | **Metric 1** | **Metric 2** | **Metric 3** | **Verbetering** | **Uitgevoerd?** |
| Treasure ID | Index |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| **Totale winst** |  | **+%** | **+%** | **+%** | **+%** | **#optimalisaties** |

**Conclusie**

**Tuning advisor output:**

CREATE NONCLUSTERED INDEX [\_dta\_index\_user\_table\_5\_821577965\_\_K1\_2\_3] ON [dbo].[user\_table]

(

[id] ASC

)

INCLUDE ( [first\_name],

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

Go

his topic describes how to add included (or nonkey) columns to extend the functionality of nonclustered indexes in SQL Server by using SQL Server Management Studio or Transact-SQL. By including nonkey columns, you can create nonclustered indexes that cover more queries. This is because the nonkey columns have the following benefits:

* They can be data types not allowed as index key columns.
* They are not considered by the Database Engine when calculating the number of index key columns or index key size.

An index with nonkey columns can significantly improve query performance when all columns in the query are included in the index either as key or nonkey columns. Performance gains are achieved because the query optimizer can locate all the column values within the index; table or clustered index data is not accessed resulting in fewer disk I/O operations.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Query | Index given | Used | Time | I/O cost | CPU | Reads |
| Select first\_name | Standard clustered index on ID (PK) |  | 1.894s | 18.1965 | 0.535332 | 24643 |
| Select first\_name | Non Clustered index including first\_name and last\_name |  | 2.198s | 2.38387 | 0.535332 | 3227 |
| Select id | Non Clustered index on ID including no columns |  | 2.461s | 0.982384 | 0.535332 | 1330 |
| Select id | Non Clustered index on ID including first\_name and last\_name |  | 2.199s | 2.38387 | 0.535332 | 3227 |
| Select \* | Standard clustered index on ID (PK) | **Clustered index** | 4.308s | 18.1965 | 0.535332 | 24643 |
| Select \* | Non Clustered index on ID including first\_name and last\_name | **Clustered index** | 4.311s | 18.1965 | 0.535332 | 24643 |
| Select first\_name | Non Clustered index on ID including first\_name and last\_name |  | 2.198s | 2.38387 | 0.535332 | 3227 |
| Select first\_name | Non Clustered index on ID including name, mail street and number |  | 2.195s | 6.47498 | 0.535332 | 8768 |
| Select name, mail street and number | Non Clustered index on ID including first\_name and last\_name |  | 3.445s | 18.1965 | 0.535332 | 24643 |
| Select name, mail street and number | Non Clustered index on ID including name, mail street and number |  | 3.712s | 6.47498 | 0.535332 | 8768 |

* De winst in het toevoegen van non-key kolommen (zoals naam en voornaam) in een non-clustered index hangt af van de query
  + Als de opgevraagde kolommen overeen komen met wat er in de non-clustered index mee is genomen, dan zal de query efficiënter zijn
* **Clustered keys zijn goed voor opzoeken, maar niet voor schrijven**
* **Why not put a clustered index on name/last name? or on any other?**
* **How does the combination of keys work? Cluster PK is not removed?**
  + **Doen we opzoekingen op basis van de sleutel?**

[**https://hackernoon.com/clustered-vs-nonclustered-what-index-is-right-for-my-data-717b329d042c**](https://hackernoon.com/clustered-vs-nonclustered-what-index-is-right-for-my-data-717b329d042c)

* **Tijd over: FK opnemen in treasure? Sneller**
* **-** check academic literature

**Check:**

* **https://www.sqlshack.com/query-optimization-techniques-in-sql-server-tips-and-tricks/**
* The more filters in the Where clause the better. Simply because the more filters we put in is less data that SQL Server will return. You’ve seen this in this article, but keep in mind when you see scans; you either don’t have a Where clause or the Where clause didn’t cover enough columns
* Select only columns that you need. Too often people have a complex query with a lot of Where clauses and Joins, but if the query starts with the Select (\*) everything which directly affects network, bandwidth, and SQL Server because it’s grabbing everything instead of only fetching columns that you need
* Be mindful of Joins. This is entirely another aspect of the game. SQL Server internally has three different ways to tie data from multiple tables together. Covering all of them would require another article which I have in mind for future. General rule here, always join columns that have indexes, keys on them and stay away from joining columns like character data
* Revisit indexing often. We already mentioned this when talking about indexing strategy. There is one tool, part of SQL Server, that’s called Index Tuning Wizard which I wanted to mention which can be quiet useful, but I’ll make sure to cover this in one of the next articles
* Create indexes on boolean and numeric data types. Basically, we’re looking for data with the high value of uniqueness which are great candidates for indexes
* Ensure indexes cover Where clauses. Also shown in this article with switching execution plan operation from clustered index scan to non-clustered index seek
* Move queries to stored procedures when possible because you can get a reliable performance gain from doing so
* Creating another indexes in a table can speed up the reading process, but will have a negative impact when changing the table.
* it is recommended to use the EQUALS operator (=) for indexed fields. Of course, you can’t get by with this single operator, you need to use operators BETWEEN, LIKE, <, >, <=, or >= for comparisons. But it is necessary to set a condition that would produce the smallest amount of results. And when creating clustered index, it is also necessary to put first columns on which EQUALS was used.
* It is best to join tables starting with the one that will produce the least amount of results after filtering.
* Indexes are not used when:
  + Function or operation/conversion is applied to a column. For example:
  + Range of values is too big.