**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. We starten dit dossier met het toelichten van de methodologie. Daarna lichten we de verschillende optimalisaties toe. Het overzicht van alle optimalisaties is te vinden op pagina X.

# Methodologie

De stroom van gegevens werd waarachtig gesimuleerd door de verscheidene optimalisaties te testen op de meest gebruikte query’s (zie github link). Om zo efficiënt mogelijk te werken werden eerst de voornaamste bottlenecks geïdentificeerd en aangepakt. Elke optimalisatie werd apart getest. Na afloop werd de volledige workload ook geanalyseerd door de *database tuning advisor* (DTA) in de SQL management studio. Echter met de voorstellen van de DTA werd voorzichtig omgesprongen. De database werd als ‘geoptimaliseerd’ beschouwd als alle query’s adequaat presteerden, tot aan het punt waar extra investeringen slechts afnemende opbrengsten zouden realiseren.

Voor de keuze van de doorgevoerde optimalisaties werd er rekening gehouden met verschillende paramaters; het aantal fysieke/logische *reads*/*writes*, het CPU gebruik en de totale looptijd van de query. Zo is het aantal *reads* een zeer informatieve parameter die weergeeft hoeveel rijen de databank heeft moeten aanspreken en ophalen. Dit geeft onder andere weer hoe efficiënt de indexering strategie was. Het CPU gebruik en de totale looptijd van een query, hoewel deze systeem maar ook tijdsafhankelijk kunnen zijn, geven ze een goede en relevante maatstaf om in acht te nemen. Om te controleren voor de tijdsafhankelijkheid zijn deze maatstaven een gemiddelde van vijf metingen. De meting van de globale efficiëntie winst werd uiteindelijk gerealiseerd door …

Er werd niet alleen rekening gehouden met de efficiëntie waarmee data kon worden opgehaald, maar ook de efficiëntie waarmee data kan worden weggeschreven. Verder stond het comfort van de klant/user centraal.

# Optimalisatie

Tot de verscheidene optimalisaties hier doorgevoerd behoren: een transformatie van de ID’s, verscheidene indexering strategieën, fragmentatie, het creëren van *views*, het gebruik van compressie, *data optimized tables* en het aanpassen van de onderliggende database instellingen.

## Transformatie ID’s

Het gebruikte sleutel format in de operationele databank had een lengte van 255 bytes. Door deze sleutels op te slagen als datatype binary(16), werd er niet alleen opslagruimte vrijgemaakt, maar verlopen nagenoeg alle query’s die deze sleutels ophalen of wegschrijven efficiënter. Verder vermindert dit ook de I/O cost bij indexering aangezien de databank minder achterliggende pages (die 8KB groot zijn), moet lezen of wegschrijven. Er werd gekozen voor het data type binary(16) omdat dit het gangbare type is om GUID’s op te slagen.

Om de winst van deze optimalisatie te illustreren werd er een query uitgevoerd die (alle) data ophaalt uit de treasure\_log, user\_table en treasure\_table. Het systeem had hiervoor in totaal **XXX ms** voor nodig, met een **CPU cost van XX**. Door alle Ids te vervangen kon deze query uitgevoerd worden in **XXX ms** en met een **CPU cost van XX**, of een efficiëntie winst **van XX%.**

Onder sommige omstandigheden kan het gebruik van het binaire data type problematisch zijn. Zo kan een verschillende implementatie de volgorde van de bytes veranderen. Echter in het huidige operationele systeem werden er geen enkele conflicten waargenomen.

## Indexering strategie

Buiten de geclusterde indexen op de primaire sleutels, was er initieel geen enkele index toegepast op de operationele databank. Hieronder wordt er per tabel in de databank de voorgestelde indexering strategie besproken. De indexering strategie omhelst meer dan enkel indexen toe te voegen op tabellen. Er werd ook steeds gekeken dat de query’s zo efficiënt mogelijk geschreven werden en dat ze in overeenstemming zijn met de geïmplementeerde index.

### dbo.treasure

Er worden drie 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, kunnen gebruikers ook treasures aanmaken.

De index strategie bestaat er ten eerste uit om een non-clustered index te plaatsen op owner\_ID. Om het aantal treasures te zoeken bij een gegeven gebruiker werd er in de initiële set-up 9660 read operaties uitgevoerd die 80ms CPU tijd vergde. Dit kon verminderd worden naar 9 read operaties en <0 ms CPU tijd. Het is ook mogelijk om een view te creëren die het aantal schatten al op voorhand berekend. Zo moet dit niet steeds opnieuw gebeuren, wat parse en compile tijd reduceert. Er werd een clustered index geplaatst op de ID van de treasure in deze view. Het gebruiken van de view verminderd het aantal read operaties naar 2, en de CPU tijd blijft hetzelfde. Er moet echter wel rekening gehouden worden dat write operaties nu (iets) zwaarder zullen zijn.

Ten behoeve van de tweede query is het mogelijk om een non-clustered index te plaatsten op de ID van city. Dit verminderde het aantal reads drastisch (van 47983 tot 39703) alsook de CPU tijd (391ms tot 374ms). Echter, deze index kan verder verfijnd worden door het toevoegen (include) van terrain en difficulty bij deze index. Zo moet er betreft het opzoeken van terrain en difficulty niet meer naar de fysieke databank gegaan worden, maar wordt deze zoektocht beperkt tot de ‘pages’ van de index. Tenslotte werd het ook getest of het mogelijk is om een view te creëren voor het aantal stages per treasure. Hier werd een clustered index geplaatst op de ID van treasure en ook een non-clustered index op city ID samen met terrain, difficulty en aantal stages. Hier, zoals hierboven, werd uitdrukkelijk rekening gehouden met de volgorde van de elementen van de index en de elementen uit de query. Dit reduceerde het aantal reads tot 6 en een totale CPU tijd van <0ms.

### dbo.stage/dbo.treasure\_stages

Op **dbo.stage/dbo.treasure\_stages** worden in essentie 3 queries uitgevoerd. Ten eerste wordt deze tabel aangeroepen om de beschrijvingen op te zoeken die horen bij een bepaalde treasure. Dit dient te gebeuren op basis van de visbility. Ten tweede kan een gebruiker bij het zoeken van treasures filteren op het aantal stages, echter dit werd reeds bovenaan behandelt. Ten derde worden er stages toegevoegd.

De indexering strategie van deze tabel bestond eruit om een clustered index te creëren op de treasure ID van de tabel treasure\_stages. Waar er eerst 7679 reads nodig waren met een CPU tijd van 703ms, werd dit gereduceerd tot 15 reads en <0ms CPU tijd. Deze index heeft geen impact op het gekozen query plan van de test-query uit de bovestaande sub-sectie (over treasure). Het query plan prefereerde nog steeds om de view te gebruiken (met een verschil van 54 reads). Er werd getest of het mogelijk is om deze clustered index te vervangen met een non-clustered index

### dbo.treasure\_log

Er worden 3 queries uitgevoerd op dbo.treasure\_log. Ten eerste, het laatste X aantal geplaatste logberichten dat bij een bepaalde treasure hoort. Ten tweede heeft de helpdesk de mogelijkheid om logs te modereren, hiervoor moeten ze zoeken op tekst. Tot slot kunnen er uiteraard ook nieuwe rijen worden toegevoegd in de tabel.

Index strategie. Non-clustered index op treasure ID zorgt voor een sterke vermindering in zoek tijd. Voor enkele queries zou het interessant zijn om extra indexen op deze tabel te plaatsen, maar aangezien deze tabel één is waar veel inserts op moeten gebeuren opteren we ervoor om het aantal indexen tot het minimum te beperken.

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 creëren?

* Check of een view creëren 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.

**PARTITIONING**

1. Filegroup creëren voor elke partitie, namelijk 9 (zie verder). We laten de PRIMARY filegroup buiten beschouwing, hier gaan we niets van de partitie inzetten.
2. Maak voor elke filegroup een file van het type rows aan en assign deze file aan de filegroup. Je kan geen partitionering doen en de partitie toewijzen aan een lege filegroup
3. Partition wizard volgen. We gaan partitioneren op basis van de log\_time. De partition functie en scheme geven we een nuttige naam (e.g. partition\_fun\_treasure\_log en partition\_scheme\_treasure\_log)
4. Bij map partitions kiezen we voor Set boundaries. Hier stellen we in wat onze range is. De range die we nemen is onze van de eerste log (september 2015) tot de meeste recente log (september 2018). Deze eerste partitie heeft in dit geval een beperkte hoeveelheid data, maar we stellen deze partitie hier even voor als al wat er in een ‘echte’ productiedatabase voor deze datum zou komen, wat op zijn beurt ook verder gepartitioneerd zou worden. Wanneer we kiezen voor een halfjaarlijkse partitionering, dan resulteert dit in 8 verschillende partities + 1 extra voor de nieuwe data.
5. Genereer script
6. Voer uit

### Dbo.user\_table

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?

## Creëren van views

## Compressie

Compressie staat toe om de on-disk footprint sterk te verminderen en mogelijk ook CPU kost te besparen. Hoewel de compressie en decompressie CPU intensief is, zal er minder I/O kost zijn omdat de gebruikte schijfruimte een stuk kleiner is geworden. De eerste tabel die hierbij in het hoofd springt is de tabel met het grootste aantal rijen, namelijk de treasure\_log tabel. Wanneer we via onderstaande script kijken hoeveel plaats er bespaard kan worden, dan is dit op het eerste zicht aanzienlijk:

EXEC sp\_estimate\_data\_compression\_savings

@schema\_name = 'dbo',

@object\_name = 'treasure\_log',

@index\_id = NULL,

@partition\_number = NULL,

@data\_compression = 'ROW'

EXEC sp\_estimate\_data\_compression\_savings

@schema\_name = 'dbo',

@object\_name = 'treasure\_log',

@index\_id = NULL,

@partition\_number = NULL,

@data\_compression = 'PAGE'

Compressie op de treasure\_log tabel uitvoeren zou de tabel tot 40% in grootte doen reduceren. Er is echter nog een andere factor die we in rekening dienen te brengen, namelijk het feit dat op deze tabel heel veel writes gebeuren. Er is heel wat discussie over compressie toepassen op write-heavy databases, nl. of dit het process zou vertragen of niet.

## Memory optimized tables

Memory optimized tabellen zijn veelbelovend omwille van hun snelheid, ze zitten immers in het veel snellere RAM geheugen in plaats van on disk. Tot voor SQL Server 2016 was dit echter vaak geen interessante optie aangezien het (onder andere) niet mogelijk was om foreign keys te definieren op deze tabellen. Met de komst van SQL Server 2016 valt onder meer deze belemmering weg, wat het aantal use cases voor deze feature verhoogt.

Er is echter nog een probleem. Hoewel foreign keys zijn toegestaan sinds SQL Server 2016, beperkt zich dit enkel tot in-memory tabellen onderling. Het is dus niet mogelijk om foreign keys te leggen naar disk-based tabellen. De structuur van catchem is zo dat de tabellen dependencies hebben op elkaar (city heeft een foreign key naar country, treasure\_stages naar treasure en stages tabellen, ..). Uiteindelijk zouden we alle tabellen dus in memory moeten maken. Dit lijkt ons niet wenselijk, aangezien dit heel wat kostbaar RAM geheugen zou kosten. Een andere optie is om geen foreign keys te leggen, maar dit zorgt ervoor dat de integriteit van onze databank niet kan gegarandeerd worden, wat net het grote voordeel is van relationele databases.

Gezien de architectuur van de database en de beperkingen van SQL Server 2016 lijkt het ons dan ook geen goed idee om in-memory tabellen te gebruiken.

# Rapport

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

# Conclusie

**INFORMATION (to remove afterwards):**

* 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).
      * Creër 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?**
* **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?**
* 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?).

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.