**Optimalisatie dossier: Catchem**

Dit dossier beschrijft de optimalisatie van de 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 gesimuleerd door de optimalisaties te testen op de meest gebruikte query’s (voor het bron materiaal zie: [hier](https://github.com/BigDataKdG/Catchem/tree/master/Module%201%20Optimalisatie)). 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, ophalen of wegschrijven. Dit geeft onder andere weer hoe efficiënt een bepaalde indexering strategie is geweest. 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.

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, het creëren van *views*, fragmentatie, 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 data ophaalt uit alle tabellen. Het systeem had hiervoor in totaal ongeveer **1452ms** voor nodig. Door alle Ids te vervangen kon deze query uitgevoerd worden in ongeveer **487ms**. Tussenin werd de cache gecleared om te vermijden dat het gecached plan zou gebruikt worden.

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 query’s 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 door het implementeren van de index verminderd worden naar 9 read operaties en <0 ms CPU tijd. Verder is het is ook mogelijk om een view te creëren die het aantal schatten per user al op voorhand berekent. Zo moet dit niet steeds opnieuw gebeuren, wat ook parse en compile tijd reduceert. Er werd op deze view 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 terwijl 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 (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. Op deze view 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. Dezelfde data ophalen maar via de view 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 test query bestond eruit om in één query alle details van de stages op te halen die bij bepaalde treasures horen, waaronder hun beschrijving, de eigenaar, het aantal treasurs van de eigenaar, de moeilijkheidsgraad het terrein en de visibiliteit.

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, maar dit leverde geen winsten op.

### dbo.user\_table

Deze tabel wordt aangeroepen met drie queries. Ten eerste wanneer de details van een eigenaar van een treasure worden opgevraagd. Echter gebeurt dit op dit moment via de view dus laten we dit buiten beschouwing. Ten tweede moet er geoptimaliseerd worden voor de query’s gebruikt door de user desk. Ten slotte worden er uiteraard ook nieuwe users aangemaakt.

De index strategie bestond eruit om, als eerste regel de query en de index in overeenstemming te brengen. Met name, de elementen die worden opgevraagd, moeten in de index aanwezig zijn. We kiezen ervoor om een non-clustered index te zetten op de city ID, dit wil zeggen dat de helpdesk de zoektocht van de query a priori af zal bakenen door aan de locatie van een gebruiker mee te geven. Voor een non-clustered index zijn er nog twee keuzes. (1) ofwel een index op city ID én de gegevens van de gebruiker, de zogenaamde composieten index of (2) een index op city ID. De laatste kent de meeste efficiëntie winst en werd daarom gekozen. Dit werd ook bevestigd door de keuze van de query optimizer. Waar er initieel 25872 reads en 203 cpu tijd nodig was, kon dit terug gebracht worden tot 3 reads en 0ms CPU tijd. Als de stad ID niet wordt meegegeven, wordt er een index voorzien op de voornaam van de gebruiker, samengenomen (including) met de gegevens van de gebruiker. Dit zorgt voor 10263 minder reads en 92ms minder CPU tijd.

### 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.

De index strategie bestaat er uit om een full-text index te creëren op de tabel. Dit zorgt voor een zeer drastische vermindering van maar liefst 148555 reads en 41391ms CPU tijd.

* Query voor het laatst aantal log-berichten? Non-clustered index op treasure\_id
  + Van 209830 reads en 2265ms
  + Naar 2392 en 15ms

**Paritionering**

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

## 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.

# Samenvatting

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Table** | **Baseline reads** | **Baseline CPU** | **Optimalisatie** | **Reads** | **CPU tijd** | **Verbetering** | **Doorgevoerd** |
| Treasure | 9660 | 80ms | Index 1 | 9 | 0ms | 99.0% | Nee |
|  |  |  | VIEW 1 | 2 | 0ms | 99.9% | Ja |
|  | 47983 | 391ms | Index 2 | 39703 | 374ms | 17.2% | Nee |
|  |  |  | View 2 | 6 | 0ms | >99.9% | Ja |
| Tr/Stage(s) | 7679 | 703ms | Index 3 | 14 | 0ms | 99.8% | Ja |
| User | 25 872 | 203ms | Index 4 | 3 | 0ms | >99.9% | Ja |
| Treasure\_log | 209 775 | 41735 | Index 5 | 61220 | 344 | >99.9% | Ja |
|  |  |  | Partitioning |  |  |  | Ja |
|  |  |  | Compressie |  |  |  | Ja |
|  |  |  |  |  | 718 |  |  |

**Conclusie**

In dit document werd de optimalisatie beschreven van de catchem database. De indexeringstrategie, het aanmaken van views, het doorvoeren van partionering en compressie werden toegelicht alsook een aantal optimalisatie die niet werden doorgevoerd. Elke optimalisatie leidde tot minimum 99% meer efficiënte query’s, met voor de gesimuleerde query’s een totale CPU run-time reductie van maar liefst 42412ms.