**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.

**Methodologie**

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?

**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

**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**

**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.