Indexing guidelines

Last updated by | Vitor Tomaz | Feb 24, 2023 at 3:32 AM PST

Contents

Prerequisites

First make sure that you are familiar with index concepts - https://supportability.visualstudio.com/AzureSQLDB/_wiki/wikis/AzureSQLDB.wiki/724458/Clustered-Non-Clustered-indexes-and-Heaps-General-guidelines

Looking at the structure of a query and determine an index

For example, we have the table and query below on the Adventure Works database:

```
Select CustomerID from [SalesLT].[Customer] where SalesPerson = 'adventure-works\josé1'
```

Since CustomerID is the PK (so the will be no Key Lookup), the index required for this query would be:

```
CREATE NONCLUSTERED INDEX [IDX01_SalesLT] ON [SalesLT].[Customer] ([SalesPerson])
```

Looking at another query:

```
Select FirstName, LastName from [SalesLT].[Customer] where SalesPerson = 'adventure-works\josé1'
```

On this case, since FirstName and LastName are not part of the index that we have above or part of the Clustered Index the SELECT query will have a Key Lookup operation. To avoid it the index would have to be rewritten to:

```
CREATE NONCLUSTERED INDEX [IDX01_SalesLT] ON [SalesLT].[Customer] ([SalesPerson]) INCLUDE ([FirstName],[LastName])
```

Note that after running the last SELECT statement, this index would be suggested (index with included columns) as a missing index. It will suggest has a new index. In other words, it will not suggest to redesign the first index.

Thats why that we should not blindly create an index that is reported on the missing index DMV (*dm_db_missing_index_details* and *dm_db_missing_index_group_stats*).

Remember that there is a write overhead associated with indexes. Like so, try redesigning indexes so you can avoid redundancy.

Finding the order of the key columns of an index

Like we have seen above, in general we will place as key comlumns the query predicates (where clauses). This will be easy if we have just one column. If we have more we have to take into account what will be the column order on the index key.

Note that the indexes suggested by the missing index DMVs, when there is more than one column, will simply follow the column order on the table, that most of the cases is not ideal. This limitations are documented here - https://learn.microsoft.com/en-us/sql/relational-databases/indexes/tune-nonclustered-missing-index-suggestions?view=sql-server-ver16#limitations-of-the-missing-index-feature .

The general guidelines for determining the key column order:

- Cardinality - columns with higher number of distinct values should come first. It would not be interesting to have on the first position of the index a column that returns most of the table rows.
- Inequality predicates at the end, equality columns first. Example of an inequality: (...) WHERE ID > 15. Example of an equality: (...) WHERE ID = 8.
- Try to put first columns that are used by other queries. Example:

Imagine that I the following index created (only index, besides of the Clustered index)

```
CREATE NONCLUSTERED INDEX [IDX01_SalesLT] ON [SalesLT].[Customer] ([FirstName],[SalesPerson])
```

If I run the query below, since the column SalesPerson is the second column of the index, it will perform a SCAN instead of a Seek.

```
Select CustomerID from [SalesLT].[Customer] where SalesPerson = 'adventure-works\josé1'
```

So the most apropriate action would be to rewrite the index to

```
CREATE NONCLUSTERED INDEX [IDX01_SalesLT] ON [SalesLT].[Customer] ([SalesPerson],[FirstName])
```

This way the index will be used efficiently by queries that uses just [SalesPerson] or both columns. Note that a query uses only [FirstName] will now be not optimal, so changing indexes needs an understanding of the workload and testing.

Another example. Lets take a few query examples:

```
Select CustomerID from [SalesLT].[Customer] where SalesPerson = 'adventure-works\josé1'
```

```
Select FirstName, LastName from [SalesLT].[Customer] where CompanyName = 'Sharp Bikes' and SalesPerson = 'adventure-works\josé1'
```

```
Select FirstName, LastName from [SalesLT].[Customer] where SalesPerson = 'adventure-works\josé1'
```

```
Select EmailAddress from [SalesLT].[Customer] where SalesPerson = 'adventure-works\josé1'
```

Each of this queries will result on four different suggestions for an index (table with only the clustered index on the CustomerID table).

```
CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [SalesLT].[Customer] ([SalesPerson])
```

> CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [SalesLT].[Customer] ([CompanyName],[SalesPerson])

> CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [SalesLT].[Customer] ([SalesPerson]) INCLUDE ([FirstName],[LastName])

> CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [SalesLT].[Customer] ([SalesPerson]) INCLUDE ([EmailAddress])

We can actually combine all of them in just one. This will reduce the write overhead and reduce the number of indexes to maintain (note that for this example I`m not following the cardinality rule).

> Create index idx01 on [SalesLT].[Customer] (SalesPerson,CompanyName) include (FirstName, LastName, EmailAddress)

Some indexing guidelines - conclusion

- Choose a narrow clustered index key - less space used
- Clustered index key with fixed-width
- Understand search arguments . key columns should follow cardinaity of each column . if cardinality is close and there is a possibility of creating less indexes by grouping columns, follow that path (less overhead on writes and less indexes to maintain)

Improving Sorting operations with indexes

The data can be already sorted on an index. One good example are Clustered indexes - check this TSG.

But we can have sorted data on nonclustered indexes and avoid Sort operations. Example:

First create a table, using the data from AdventureWorksLT:

```
select * into table1 from SalesLT.SalesOrderDetail
```

Create a clustered index on the new table:

```
Create clustered index ci1 on table1 (SalesOrderID)
```

Now let`s run the query:

```
select  UnitPrice from table1
where UnitPrice >= 1466.01
order by UnitPrice desc
```

This query will perform a Custered Index scan and a Sort by the UnitPrice in descending order.

Now let`s create the index below:

```
create index idx01 on table1 (UnitPrice desc)
```

Note that I`m using the DESC (descending order. The default is ASC - ascending).

Running the query again:

```
select  UnitPrice from table1
where UnitPrice >= 1466.01
order by UnitPrice desc
```

The query now performs an index seek, and since the data is already sorted in descending order it will not have any sort operator.

How good have you found this content?