BRENT OZAR
UNLIMITED®

Fundamentals of Index Tuning

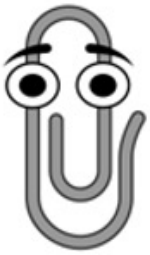
Part 7: the built-in missing index recommendations

Logistics, chat, questions, recording info:
BrentOzar.com/training/live

07 p1

Index hints are a gift.

They're a byproduct of plan compilation, but they're not the main deliverable.



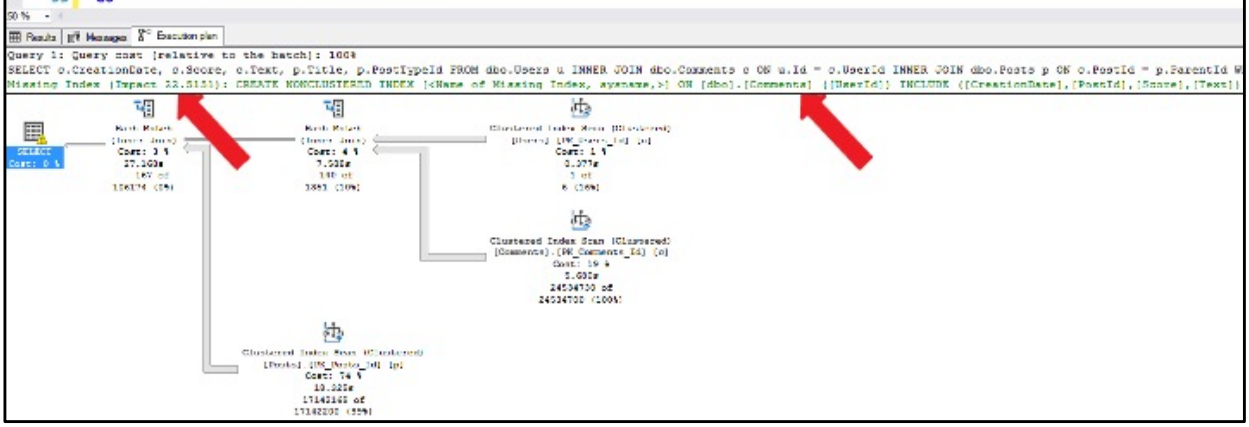
- Shown in execution plans
- Tracked over time in DMVs like sys.dm_db_missing_index_details
- Shown in tools like sp_BlitzIndex

But they're not perfect gifts.

- Suggests super wide indexes
- Doesn't de-duplicate requests
- Don't get thrown for all queries
- Get cleared at tricky times
- Doesn't recommend filtered, columnstore, indexed views, XML, spatial, in-memory OLTP

In plans, only the first one shows

```
/* What missing index does this ask for? Are you sure? */
SELECT c.CreationDate, c.Score, c.Text, p.Title, p.PostTypeId
  FROM dbo.Users u
  INNER JOIN dbo.Comments c ON u.Id = c.UserId
  INNER JOIN dbo.Posts p ON c.PostId = p.ParentId
  WHERE u.DisplayName = 'Brent Ozar';
GO
```



```
1  <?xml version="1.0" encoding="utf-16"?>
2  <ShowPlanXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XM
3    <BatchSequence>
4      <Batch>
5        <Statements>
6          <StmtSimple StatementCompId="1" StatementEstRows="106174" StatementId="1" StatementOptmLevel="F
7            <StatementSetOptions ANSI_NULLS="true" ANSI_PADDING="true" ANSI_WARNINGS="true" ARITHABORT="t
8            <QueryPlan DegreeOfParallelism="1" MemoryGrant="46240" CachedPlanSize="
9              <MissingIndexes>
10               <MissingIndexGroup Impact="22.5151">
11                 <MissingIndex Database="[StackOverflow2013]" Schema="[dbo]" Table="[Comments]">
12                   <ColumnGroup Usage="EQUALITY">
13                     <Column Name="[UserId]" ColumnId="6" />
14                   </ColumnGroup>
15                   <ColumnGroup Usage="INCLUDE">
16                     <Column Name="[CreationDate]" ColumnId="2" />
17                     <Column Name="[PostId]" ColumnId="3" />
18                     <Column Name="[Score]" ColumnId="4" />
19                     <Column Name="[Text]" ColumnId="5" />
20                   </ColumnGroup>
21                 </MissingIndex>
22               </MissingIndexGroup>
23               <MissingIndexGroup Impact="76.6896">
24                 <MissingIndex Database="[StackOverflow2013]" Schema="[dbo]" Table="[Posts]">
25                   <ColumnGroup Usage="EQUALITY">
26                     <Column Name="[ParentId]" ColumnId="15" />
27                   </ColumnGroup>
28                   <ColumnGroup Usage="INCLUDE">
29                     <Column Name="[PostTypeId]" ColumnId="16" />
30                     <Column Name="[Title]" ColumnId="19" />
31                   </ColumnGroup>
32                 </MissingIndex>
33               </MissingIndexGroup>
34             </MissingIndexes>
35             <Warnings>
```

SSMS shows the FIRST one

But not the rest

Logistics, chat, questions, recording info:
BrentOzar.com/training/live

07 p5

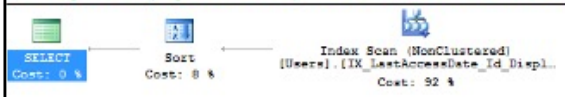
We're sorting by Age, but...

```
291 SELECT Id
292 FROM dbo.Users
293 WHERE DisplayName = 'Brent Ozar'
294 ORDER BY Age
```

100 %

Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT Id FROM dbo.Users WHERE DisplayName = 'Brent Ozar' ORDER BY Age
Missing Index (Impact 99.9502): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users] ([DisplayName])
```



Limitations of the Missing Indexes Feature

SQL Server 2008 R2 | Other Versions | This topic has not yet been rated - Rate this topic

The missing index feature has the following limitations:

- It is not intended to fine tune an indexing configuration.

07 p7

The worst gotcha by far

Limitations of the Missing Indexes Feature

SQL Server 2008 R2 | Other Versions | This topic has not yet been rated - Rate this topic

The missing index feature has the following limitations:

- It is not intended to fine tune an indexing configuration.
- It cannot gather statistics for more than 500 missing index groups.
- It does not specify an order for columns to be used in an index.
- For queries involving only inequality predicates, it returns less accurate cost information.
- It reports only include columns for some queries, so index key columns must be manually selected.
- It returns only raw information about columns on which indexes might be missing.
- It does not suggest filtered indexes.
- It can return different costs for the same missing index group that appears multiple times in XML Showplans.
- It does not consider trivial query plans.

And these apply to both the missing indexes in query plans, AND to the missing index DMVs.

The worst gotcha by far

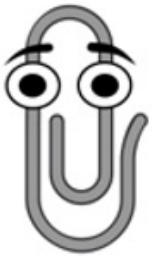
Limitations of the Missing Indexes Feature

SQL Server 2008 R2 | Other Versions | This topic has not yet been rated - Rate this topic

The missing index feature has the following limitations:

- It is not intended to fine tune an indexing configuration.
- It cannot gather statistics for more than 500 missing index groups.
- It does not specify an order for columns to be used in an index.
- For queries involving only inequality predicates, it returns less accurate cost information.
- It reports only include columns for some queries, so index key columns must be manually selected.
- It returns only raw information about columns on which indexes might be missing.
- It does not suggest filtered indexes.
- It can return different costs for the same missing index group that appears multiple times in XML Showplans.
- It does not consider trivial query plans.

And these apply to both the missing indexes in query plans, AND missing index DMVs.



Let's see how he does it.

Logistics, chat, questions, recording info:
BrentOzar.com/training/live

Create table w/10M identical rows

```sql
CREATE TABLE dbo.DiningRoom
   (FirstColumn INT,
    SecondColumn INT,
    ThirdColumn INT,
    FourthColumn INT,
    FifthColumn INT,
    SixthColumn INT
    );
INSERT INTO dbo.DiningRoom
   (FirstColumn, SecondColumn, ThirdColumn, FourthColumn, FifthColumn, SixthColumn)
   SELECT TOP 10000000 1, 1, 1, 1, 1, 1
   FROM sys.all_columns ac1
   CROSS JOIN sys.all_columns ac2
   CROSS JOIN sys.all_columns ac3;
GO
```

200 %

Results | Messages

	FirstColumn	SecondColumn	ThirdColumn	FourthColumn	FifthColumn	SixthColumn
1	1	1	1	1	1	1
2	1	1	1	1	1	1
3	1	1	1	1	1	1

Single-column equality search

```
44  /* Turn on actual execution plans, and check the missing index requests: */
45  SET STATISTICS TIME, IO ON;
46  GO
47  SELECT 'Hi Mom!'
48    FROM dbo.DiningRoom
49    WHERE FirstColumn = 0;
```

200 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT 'Hi Mom!' FROM [dbo].[DiningRoom] WHERE [FirstColumn]=@1
Missing Index (Impact 68.6423): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[DiningRoom] ([FirstColumn])
```

Logistics, chat, questions, recording info:
BrentOzar.com/training/live

07 p12

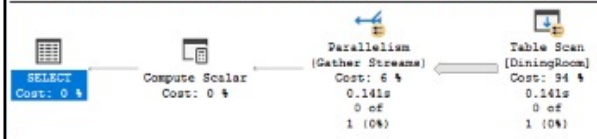
Also works if we look for column 2

```
51 SELECT 'Hi Mom!'
52     FROM dbo.DiningRoom
53     WHERE SecondColumn = 0;
54 GO
```

200 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT 'Hi Mom!' FROM [dbo].[DiningRoom] WHERE [SecondColumn]=@1
Missing Index (Impact 68.6423): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[DiningRoom] ([SecondColumn])
```

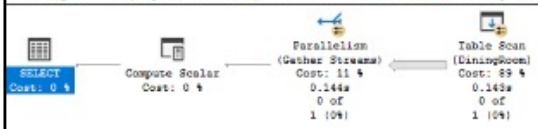


And if we look for both columns...

```
59  SELECT 'Hi Mom!'
60     FROM dbo.DiningRoom
61     WHERE FirstColumn = 0
62       AND SecondColumn = 0;
63
```

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT 'Hi Mom!' FROM [dbo].[DiningRoom] WHERE [FirstColumn]=@1 AND [SecondColumn]=@2
Missing Index (Impact 59.3985): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[DiningRoom] ([FirstColumn],[SecondColumn])
```



Logistics, chat, questions, recording info:
BrentOzar.com/training/live

07 p14

So far, not bad.

Logistics, chat, questions, recording info:
BrentOzar.com/training/live

And if we flip the WHERE clause?

What if we put SecondColumn first?

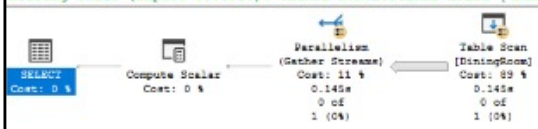
```
SELECT 'Hi Mom!'
  FROM dbo.DiningRoom
  WHERE SecondColumn = 0
    AND FirstColumn = 0;
GO
```

Hmm...what's determining order?

```
64  SELECT 'Hi Mom!'
65     FROM dbo.DiningRoom
66     WHERE SecondColumn = 0
67       AND FirstColumn = 0;
68  GO
69
```

Results | Messages | Execution plan

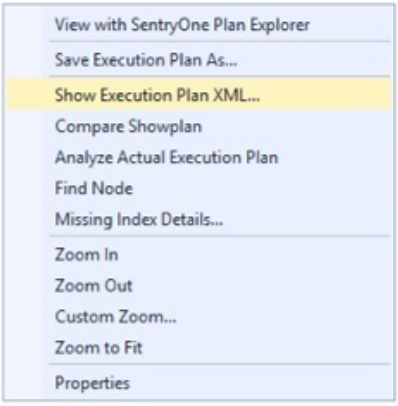
```
Query 1: Query cost (relative to the batch): 100%
SELECT 'Hi Mom!' FROM [dbo].[DiningRoom] WHERE [SecondColumn]=@1 AND [FirstColumn]=@2
Missing Index (Impact 59.3985): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[DiningRoom] ([FirstColumn],[SecondColumn])
```



Logistics, chat, questions, recording info:
BrentOzar.com/training/live

View the execution plan XML

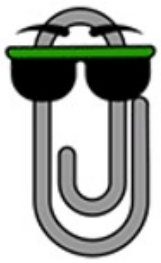
```
bo].[DiningRoom] ([FirstColumn],[SecondColumn])
```



To see how order is calculated, right-click on the plan and view the XML:

Clippy uses the table order.

The first column in the table goes first, second goes second, and so forth.



```
</ThreadStat>
<MissingIndexes>
                  roup Impact="59.3985">
                  x Database="[StackOverflow2013]" Schema="[dbo]" Table="[DiningRoom]">
                  up Usage="EQUALITY">
                   Name="[FirstColumn]" ColumnId="1" />
                   Name="[SecondColumn]" ColumnId="2" />
                  oup>
                  lex>
                  Group>
                  >
                  SerialRequiredMemory="0" SerialDesiredMemory="0" RequiredMemory="72"
```

It's just a little bit more complex...

Clippy picks key order using:

- Equality searches
  `(=, IS NULL, IN a list of 1)`
  ordered by the column they are in the table
- Inequality search columns
  `(<, >, LIKE, IS NOT NULL, IN a list of 2 or more)`
  ordered by the column they are in the table

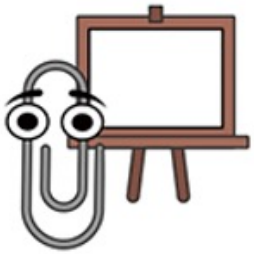
Clippy can't consider

How often you filter on a field

How selective your filter clause is

The size of the field

What you do further upstream
(joining, grouping, ordering)



Logistics, chat, questions, recording info:
BrentOzar.com/training/live

He's focused on WHERE, not GROUP BY or ORDER BY.

Order the whole table

```
SELECT Id
FROM dbo.Users
ORDER BY DisplayName;
```

133 %

Results | Messages | Execution plan

Query 1: Query cost (relative to the batch): 100%
SELECT Id FROM dbo.Users ORDER BY DisplayName

Order the whole table

```
SELECT Id
FROM dbo.Users
ORDER BY DisplayName;
```

133 %

Results | Messages | Execution plan

t (relative to the batch): 100%
.Users ORDER BY DisplayName

Logistics, chat, questions, recording info:
BrentOzar.com/training/live

07 p24

Filter, then order by

```
SELECT Id
FROM dbo.Users
WHERE Location = 'India'
ORDER BY DisplayName;
```

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT [Id] FROM [dbo].[Users] WHERE [Location]=@1 ORDER BY [DisplayName] ASC
Missing Index (Impact 90.0332): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users] ([Location]) INCLUDE ([DisplayName])
```

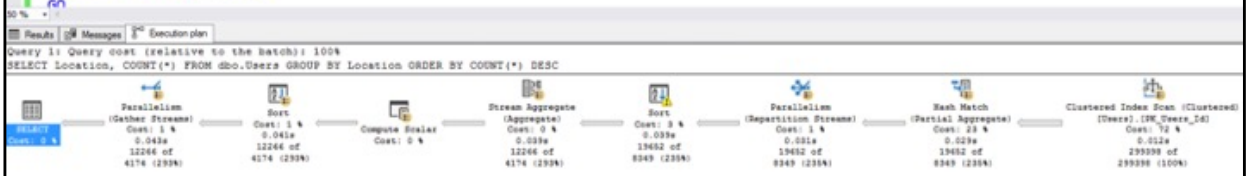
Clippy just INCLUDEs DisplayName, figuring he's going to sort all of the people in India by name, every single time this query runs. Another blind spot.

What's he suggest for this?

```
SELECT Location, COUNT(*)
  FROM dbo.Users
  GROUP BY Location
  ORDER BY COUNT(*) DESC;
```

Seems like a lot of work

```
SELECT Location, COUNT(*)
  FROM dbo.Users
  GROUP BY Location
  ORDER BY COUNT(*) DESC;
GO
```



Scan the whole table, dump locations into buckets, go parallel across threads, sort them, spill to disk...

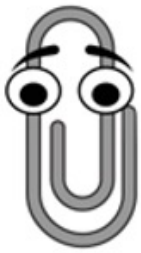
But no index recommendation?

Try creating one by hand.

```
CREATE INDEX IX_Location
ON dbo.Users(Location);
```

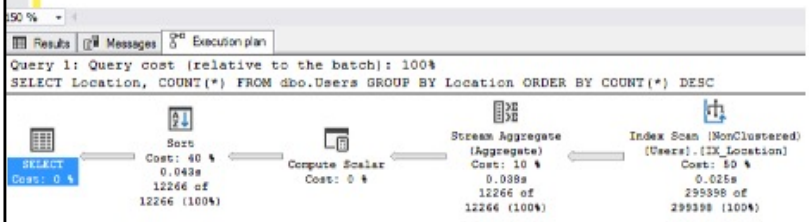
Try creating one by hand.

```
CREATE INDEX IX_Location
ON dbo.Users(Location);
```



He uses the index

```sql
CREATE INDEX IX_Location ON dbo.Users (Location);
GO
SELECT Location, COUNT(*)
  FROM dbo.Users
  GROUP BY Location
  ORDER BY COUNT(*) DESC;
GO
```



Way faster

Single-threaded

Great estimates

No spills to disk

Adding Clippy's indexes can even make things worse.

Disclaimer: reproing this is tricky.

The exact index suggestions will vary based on:

- Your Stack database size (10GB, 50GB, 300+GB)
- Your SQL Server version
- Cost Threshold for Parallelism

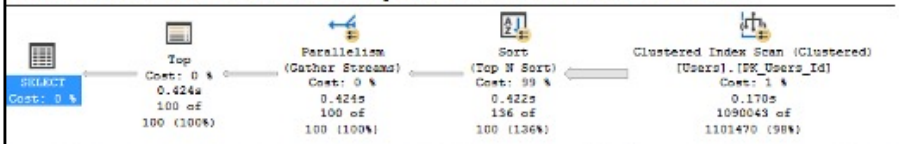
Try this query with no indexes.

```
57  DropIndexes;
58  GO
59  SELECT TOP 100 *
60    FROM dbo.Users
61    WHERE Reputation = 1
62    ORDER BY CreationDate DESC;
63  GO
64
```

50 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT TOP 100 * FROM dbo.Users WHERE Reputation = 1 ORDER BY CreationDate DESC
```



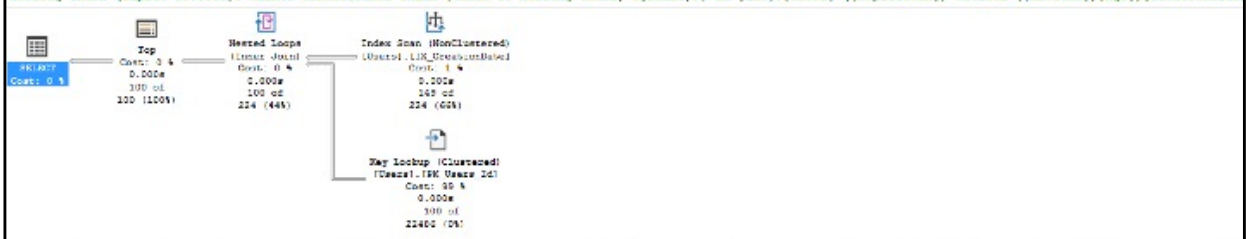
Logistics, chat, questions, recording info:
BrentOzar.com/training/live

Add an index, and it's fast!

```
65  CREATE INDEX IX_CreationDate ON dbo.Users(CreationDate);
66  GO
67  SELECT TOP 100 *
68      FROM dbo.Users
69      WHERE Reputation = 1
70      ORDER BY CreationDate DESC;
71  GO
```

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT TOP 100 * FROM dbo.Users WHERE Reputation = 1 ORDER BY CreationDate DESC
Missing Index (Impact 61.1101): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users] ([Reputation]) INCLUDE ([AboutMe],[Age],[CreationDate
```



Logistics, chat, questions, recording info:
BrentOzar.com/training/live

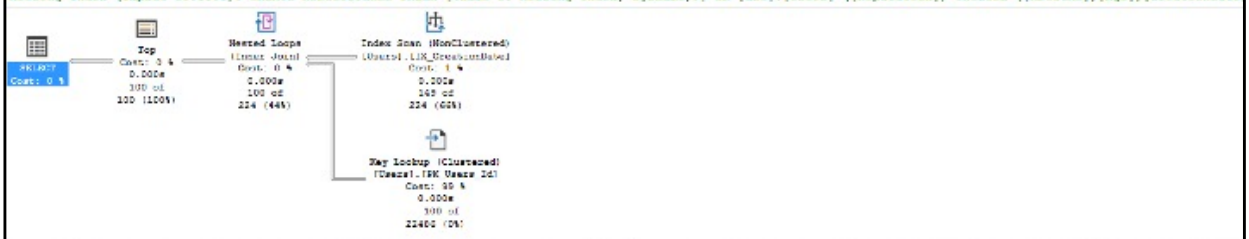
07 p34

Add an index, and it's fast, but...

```
65  CREATE INDEX IX_CreationDate ON dbo.Users(CreationDate);
66  GO
67  SELECT TOP 100 *
68    FROM dbo.Users
69    WHERE Reputation = 1
70    ORDER BY CreationDate DESC;
71  GO
```

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT TOP 100 * FROM dbo.Users WHERE Reputation = 1 ORDER BY CreationDate DESC
Missing Index (Impact 61.1101): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users] ([Reputation]) INCLUDE ([AboutMe],[Age],[CreationDate
```



Logistics, chat, questions, recording info:
BrentOzar.com/training/live

Now he's got an idea.

```
65  CREATE INDEX IX_CreationDate ON dbo.Users(CreationDate);
66  GO
67  SELECT TOP 100 *
68     FROM dbo.Users
69     WHERE Reputation = 1
70     ORDER BY CreationDate DESC;
71  GO
```

Logistics, chat, questions, recording info:
BrentOzar.com/training/live

07 p36

```sql
CREATE INDEX IX_CreationDate ON dbo.Users(CreationDate);
GO
SELECT TOP 100 *
  FROM dbo.Users
  WHERE Reputation = 1
  ORDER BY CreationDate DESC;
GO
```

150 %

Results | Messages | Execution plan

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 1 ms.

(100 rows affected)
Table 'Users'. Scan count 1, logical reads 468, physical reads 0, read

(1 row affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 136 ms.
```

That's...interesting.

The query is already really, really fast, and does pretty few logical reads.

He wants to double the table size.

```
The Query Processor estimates that implementing the following index could improve the query cost by 61.1101%.
*/

/*
USE [StackOverflow2013]
GO
CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]
ON [dbo].[Users] ([Reputation])
INCLUDE ([AboutMe],[Age],[CreationDate],[DisplayName],[DownVotes],[EmailHash],[LastAccessDate],[Location],[UpVotes],[Views],[WebsiteUrl],[AccountId])
```

But note the index's key.

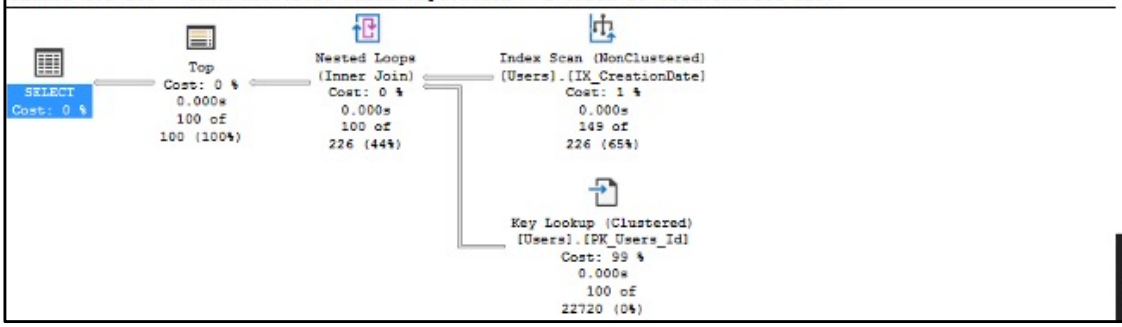
```
75  CREATE NONCLUSTERED INDEX IX_Clippy_Reputation
76  ON [dbo].[Users] ([Reputation])
77  INCLUDE ([AboutMe],[Age],[CreationDate],[DisplayName],[DownVotes],[Email
78
79  SELECT TOP 100 *
80    FROM dbo.Users
81    WHERE Reputation = 1
82    ORDER BY CreationDate DESC;
83  GO
```

We create it.
It doesn't get used!

150 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT TOP 100 * FROM dbo.Users WHERE Reputation = 1 ORDER BY CreationDate DESC
```



07 p39

```
75 CREATE NONCLUSTERED INDEX IX_Clippy_Reputation
76 ON [dbo].[Users] ([Reputation])
77 INCLUDE ([AboutMe],[Age],[CreationDate],[DisplayName],[DownVotes],[Email
78
79 SELECT TOP 100 *
80   FROM dbo.Users
81   WHERE Reputation = 1
82   ORDER BY CreationDate DESC;
83 GO
```

We create it.
It doesn't get used!

150 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT TOP 100 * FROM dbo.Users WHERE Reputation = 1 ORDER BY CreationDate DESC
```

Nested Loops
(Inner Join)
Cost: 0 %
0.000s
100 of
226 (44%)

Index Scan (NonClustered)
[Users].[IX_CreationDate]
Cost: 1 %
0.000s
149 of
226 (65%)

Key Lookup (Clustered)
[Users].[PK_Users_Id]
Cost: 99 %
0.000s
100 of
22720 (0%)

07 p40

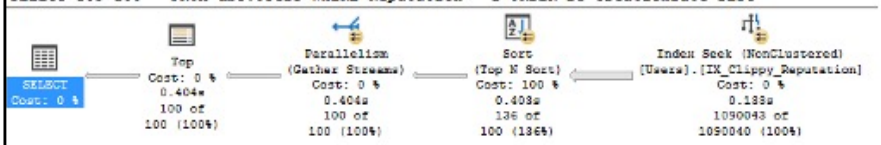
Drop the old IX_CreationDate...

```
86  DROP INDEX dbo.Users.IX_CreationDate;
87  GO
88  SELECT TOP 100 *
89    FROM dbo.Users
90    WHERE Reputation = 1
91    ORDER BY CreationDate DESC;
92  GO
```

150 %

Results | Messages | Execution plan

Query 1: Query cost (relative to the batch): 100%
SELECT TOP 100 * FROM dbo.Users WHERE Reputation = 1 ORDER BY CreationDate DESC



And the index gets used, but...that sort!

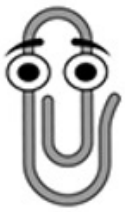
Now we're sorting 1M rows.

```sql
DROP INDEX dbo.Users.IX_CreationDate;
GO
SELECT TOP 100 *
  FROM dbo.Users
  WHERE Reputation = 1
  ORDER BY CreationDate DESC;
GO
```

Results | Messages | Execution plan

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 1 ms.

(100 rows affected)
Table 'Users'. Scan count 5, logical reads 14119, physic
Table 'Worktable'. Scan count 0, logical reads 0, physic

(1 row affected)

 SQL Server Execution Times:
   CPU time = 1577 ms,  elapsed time = 520 ms.
```

CPU time,
elapsed time,
and logical reads
are all WORSE
than the original query.



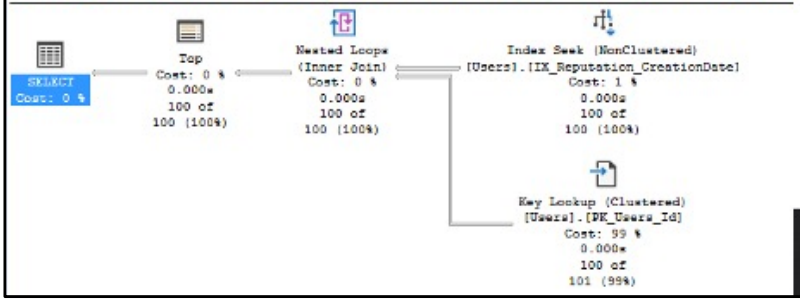
Logistics, chat, questions, recording info:
BrentOzar.com/training/live

```
94   DropIndexes;
95   GO
96   CREATE INDEX IX_Reputation_CreationDate
97     ON dbo.Users(Reputation, CreationDate);
98   GO
99   SELECT TOP 100 *
100    FROM dbo.Users
101    WHERE Reputation = 1
102    ORDER BY CreationDate DESC;
103  GO
```

150 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT TOP 100 * FROM dbo.Users WHERE Reputation = 1 ORDER BY CreationDate DESC
Missing Index (Impact 81.1673): CREATE NONCLUSTERED INDEX [<Name of Missing Index
```

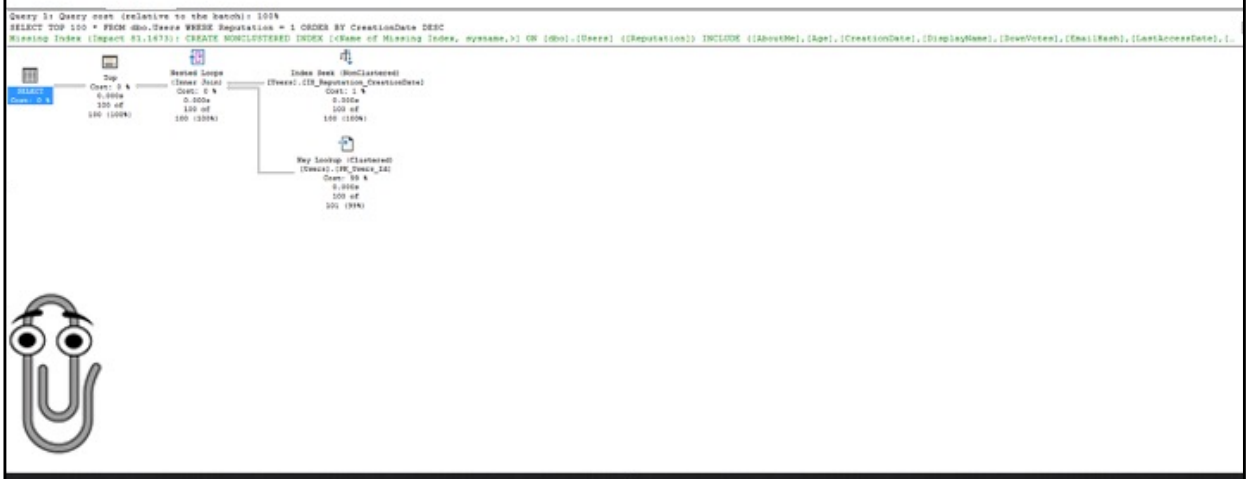


Clippy was on to something

An index tweak will help – just not the index Clippy wanted.

Key on both fields, and the sort is gone.

And he STILL wants the index.



Logistics, chat, questions, recording info:
BrentOzar.com/training/live

What we saw

A query wasn't terribly slow,
but SQL Server asked for an index

If this was a frequent query,
that index might seem attractive

But the requested index had the ORDER BY column
as an include, when it really needs to be sorted

The query was much better with that column in the key

How to identify it

Look for high average CPU and reads on top plans

Dig into every operator

In the real world on big plans, this is time consuming

You have to rule out other things that may be the issue, such as parameter sniffing and inefficient or out of date statistics

That's where tools come in.

sp_BlitzIndex

Github repository: FirstResponderKit.org

Psychiatrist-style analysis of indexes

But be warned: all its data comes from Clippy

- Index usage stats reset at odd times
- Missing index recommendations are derp
- Only really works in production

Running it at the server level

```
sp_BlitzIndex @GetAllDatabases = 1;
```

I don't tune here, but I use this to get a fast overall picture of which databases & tables to focus on.

At the table level

```
sp_BlitzIndex @SchemaName = 'dbo',
   @TableName = 'Users';
```

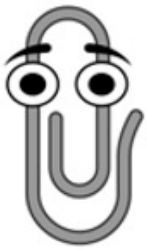
This is where I spend most of my time tuning.

Recap

You don't always get missing index requests.

Even when you do, Clippy's not putting much work in:

- Equality searches first, then inequality searches
- Fields ordered by their position in the table
- He's completely focused on the WHERE



Tools like sp_BlitzIndex get their hints from Clippy.

You can easily do better by hand.

BRENT OZAR
UNLIMITED®

Fundamentals of Index Tuning

Part 8: translating Clippy's advice into better indexes.

Logistics, chat, questions, recording info:
BrentOzar.com/training/live

This is a little different.

Restart your SQL Server instance first.

The demo script runs queries to populate Clippy's missing index recommendation DMVs. (If you have a slow VM, and it takes >10 minutes, cancel it – that's OK, you can still move on to your mission.)

Your mission: analyze the indexes with sp_BlitzIndex, see Clippy's recommendations, and build your own list of changes to make.

You can also check your production server too.