BRENT OZAR UNLIMITED®

Fundamentals of Index Tuning

The built-in missing index recommendations

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

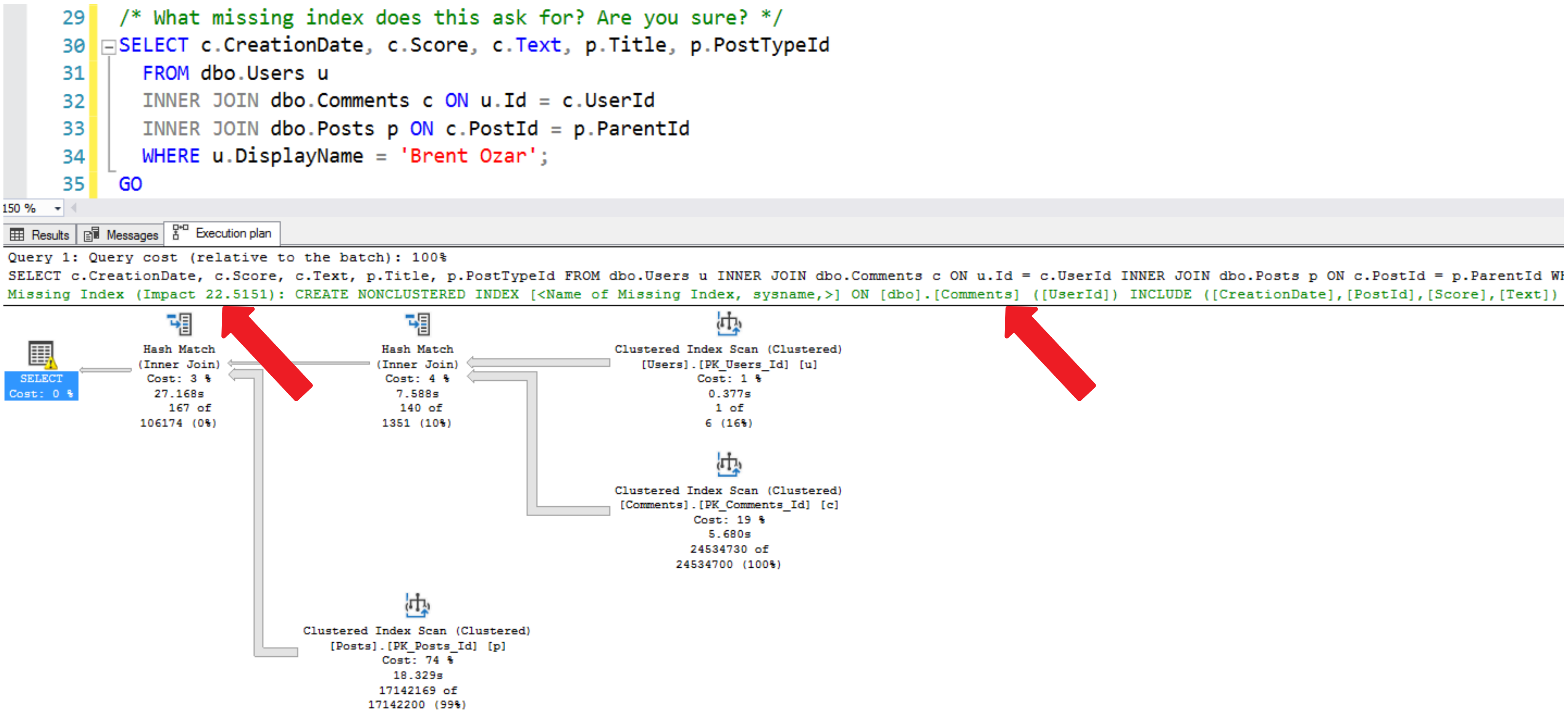
Doesn’t de-duplicate requests

Don’t get thrown for all queries

Get cleared at tricky times

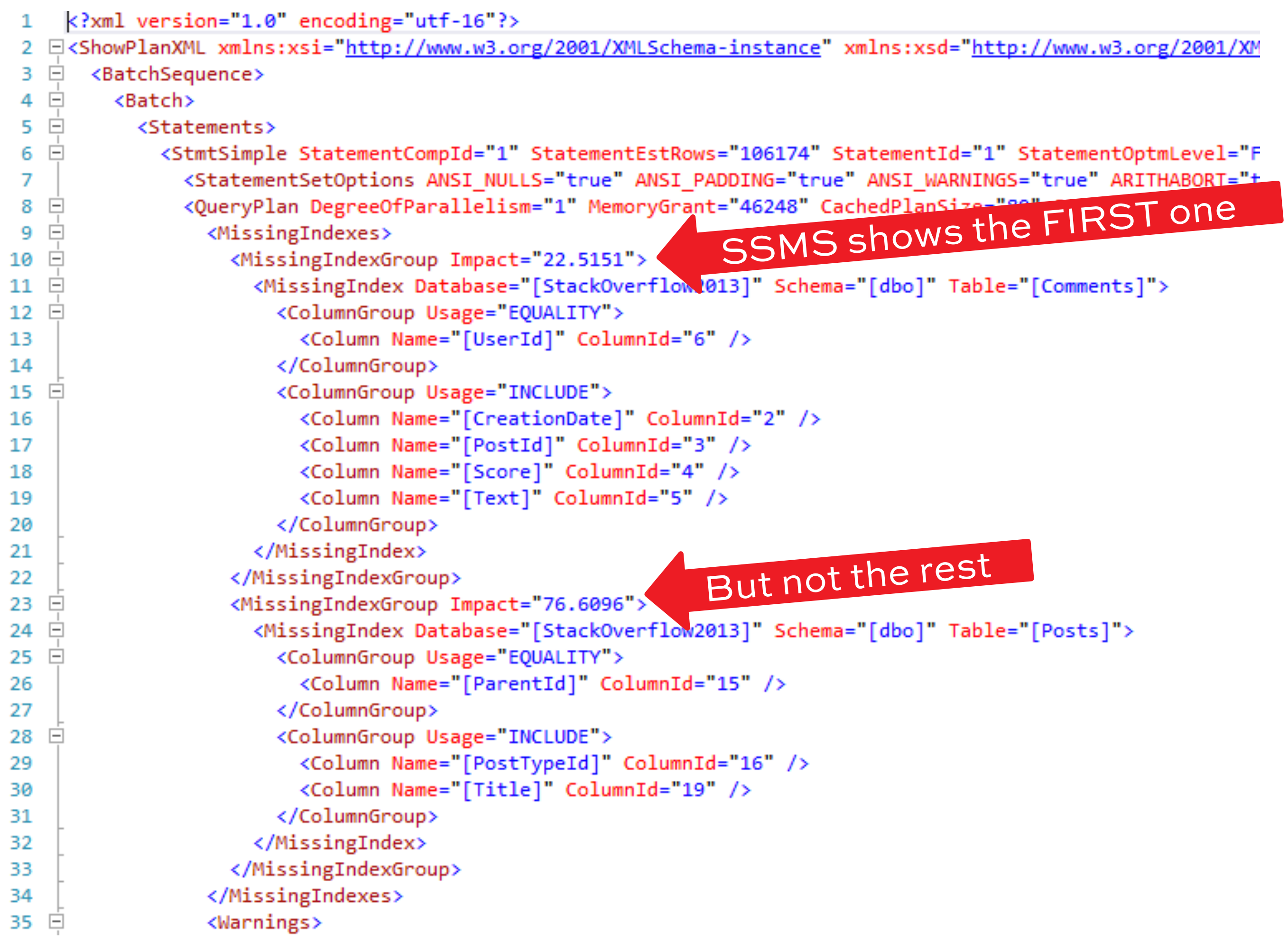
Doesn’t recommend filtered, columnstore, indexed views, XML, spatial, in-memory OLTP

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
```

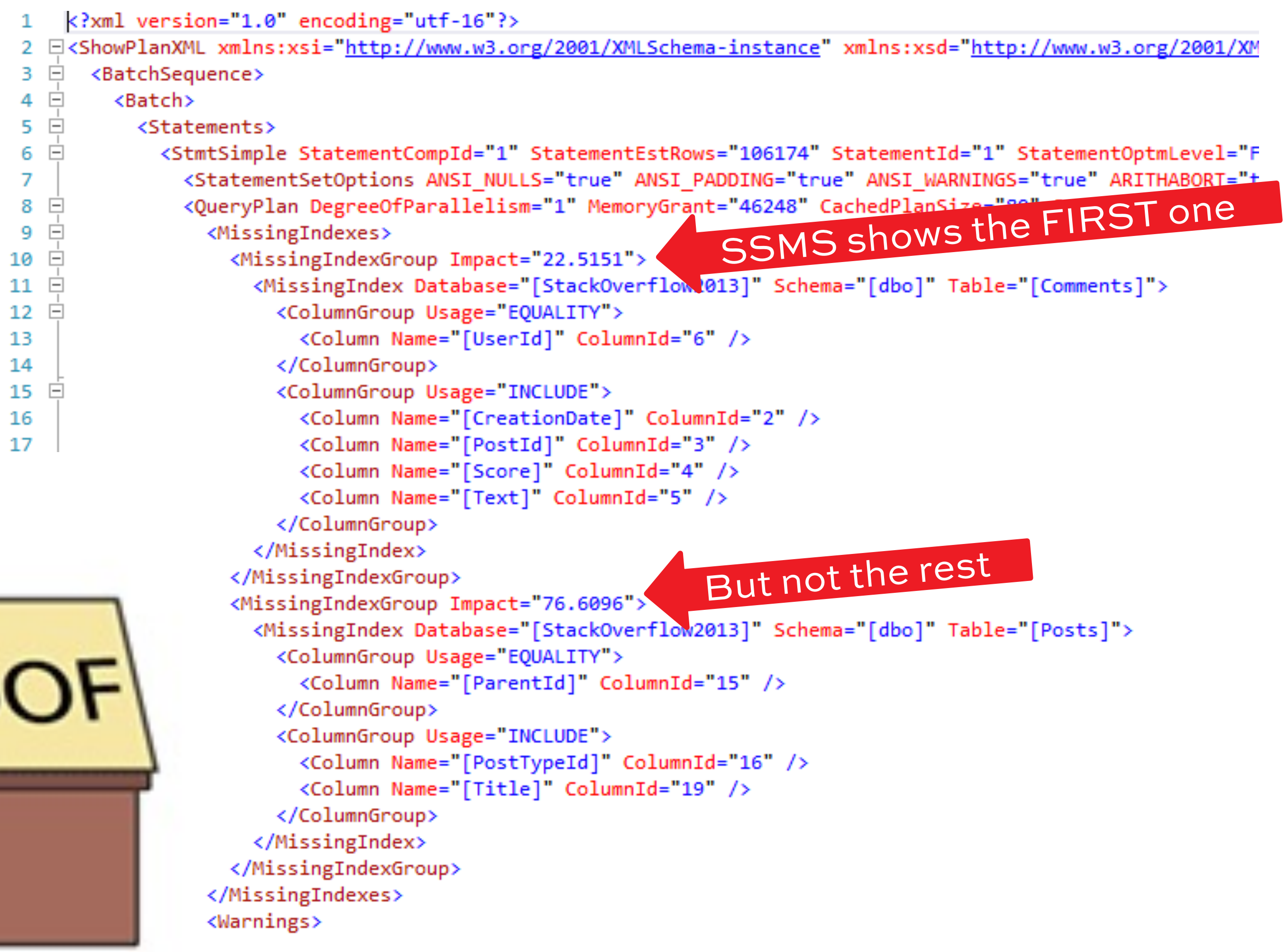
```
Query 1: Query cost (relative to the batch): 100%
SELECT c.CreationDate, c.Score, c.Text, p.Title, p.PostTypeId FROM dbo.Users u INNER JOIN dbo.Comments c ON u.Id = c.UserId INNER JOIN dbo.Posts p ON c.PostId = p.ParentId WH
Missing Index (Impact 22.5151): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Comments] ([UserId]) INCLUDE ([CreationDate],[PostId],[Score],[Text])
```



```
<?xml version="1.0" encoding="utf-16"?>
<ShowPlanXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XM
  <BatchSequence>
    <Batch>
      <Statements>
        <StmtSimple StatementCompId="1" StatementEstRows="106174" StatementId="1" StatementOptmLevel="F
          <StatementSetOptions ANSI_NULLS="true" ANSI_PADDING="true" ANSI_WARNINGS="true" ARITHABORT="t
          <QueryPlan DegreeOfParallelism="1" MemoryGrant="46248" CachedPlanSize="80"
            <MissingIndexes>
              <MissingIndexGroup Impact="22.5151">
                <MissingIndex Database="[StackOverflow2013]" Schema="[dbo]" Table="[Comments]">
                  <ColumnGroup Usage="EQUALITY">
                    <Column Name="[UserId]" ColumnId="6" />
                  </ColumnGroup>
                  <ColumnGroup Usage="INCLUDE">
                    <Column Name="[CreationDate]" ColumnId="2" />
                    <Column Name="[PostId]" ColumnId="3" />
                    <Column Name="[Score]" ColumnId="4" />
                    <Column Name="[Text]" ColumnId="5" />
                  </ColumnGroup>
                </MissingIndex>
              </MissingIndexGroup>
              <MissingIndexGroup Impact="76.6096">
                <MissingIndex Database="[StackOverflow2013]" Schema="[dbo]" Table="[Posts]">
                  <ColumnGroup Usage="EQUALITY">
                    <Column Name="[ParentId]" ColumnId="15" />
                  </ColumnGroup>
                  <ColumnGroup Usage="INCLUDE">
                    <Column Name="[PostTypeId]" ColumnId="16" />
                    <Column Name="[Title]" ColumnId="19" />
                  </ColumnGroup>
                </MissingIndex>
              </MissingIndexGroup>
            </MissingIndexes>
            <Warnings>
```

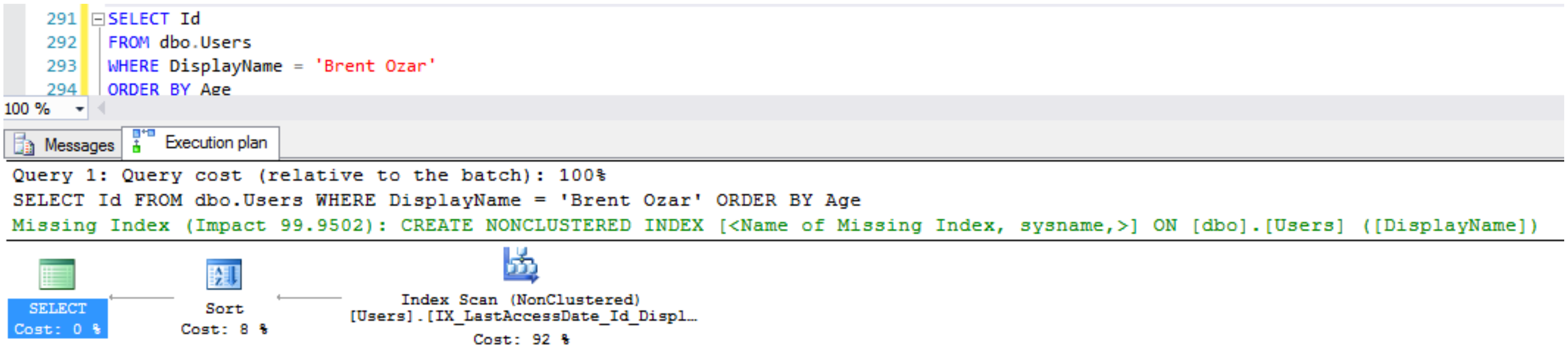
SSMS shows the FIRST one

But not the rest



```xml
<?xml version="1.0" encoding="utf-16"?>
<ShowPlanXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XM
  <BatchSequence>
    <Batch>
      <Statements>
        <StmtSimple StatementCompId="1" StatementEstRows="106174" StatementId="1" StatementOptmLevel="F
          <StatementSetOptions ANSI_NULLS="true" ANSI_PADDING="true" ANSI_WARNINGS="true" ARITHABORT="t
          <QueryPlan DegreeOfParallelism="1" MemoryGrant="46248" CachedPlanSi
            <MissingIndexes>
              <MissingIndexGroup Impact="22.5151">
                <MissingIndex Database="[StackOverflow2013]" Schema="[dbo]" Table="[Comments]">
                  <ColumnGroup Usage="EQUALITY">
                    <Column Name="[UserId]" ColumnId="6" />
                  </ColumnGroup>
                  <ColumnGroup Usage="INCLUDE">
                    <Column Name="[CreationDate]" ColumnId="2" />
                    <Column Name="[PostId]" ColumnId="3" />
                    <Column Name="[Score]" ColumnId="4" />
                    <Column Name="[Text]" ColumnId="5" />
                  </ColumnGroup>
                </MissingIndex>
              </MissingIndexGroup>
              <MissingIndexGroup Impact="76.6096">
                <MissingIndex Database="[StackOverflow2013]" Schema="[dbo]" Table="[Posts]">
                  <ColumnGroup Usage="EQUALITY">
                    <Column Name="[ParentId]" ColumnId="15" />
                  </ColumnGroup>
                  <ColumnGroup Usage="INCLUDE">
                    <Column Name="[PostTypeId]" ColumnId="16" />
                    <Column Name="[Title]" ColumnId="19" />
                  </ColumnGroup>
                </MissingIndex>
              </MissingIndexGroup>
            </MissingIndexes>
            <Warnings>
```

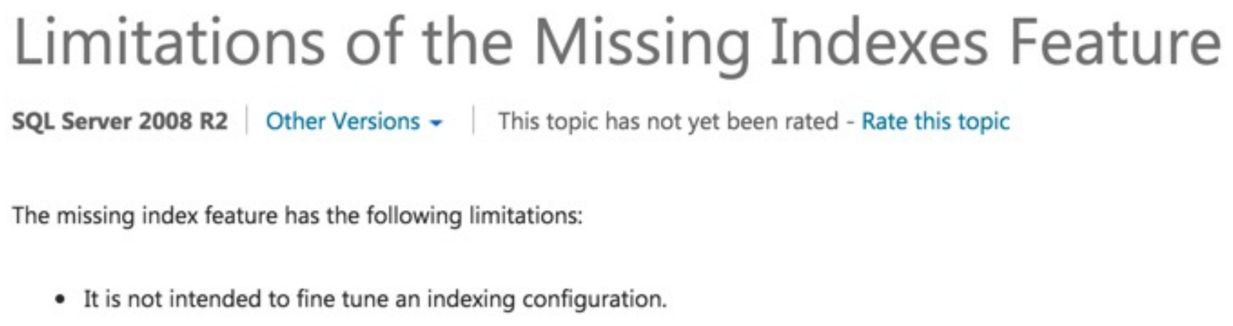
We’re sorting by Age, but...



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

```
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])
```

SELECT
Cost: 0 %
Sort
Cost: 8 %
Index Scan (NonClustered)
[Users].[IX_LastAccessDate_Id_Displ...
Cost: 92 %

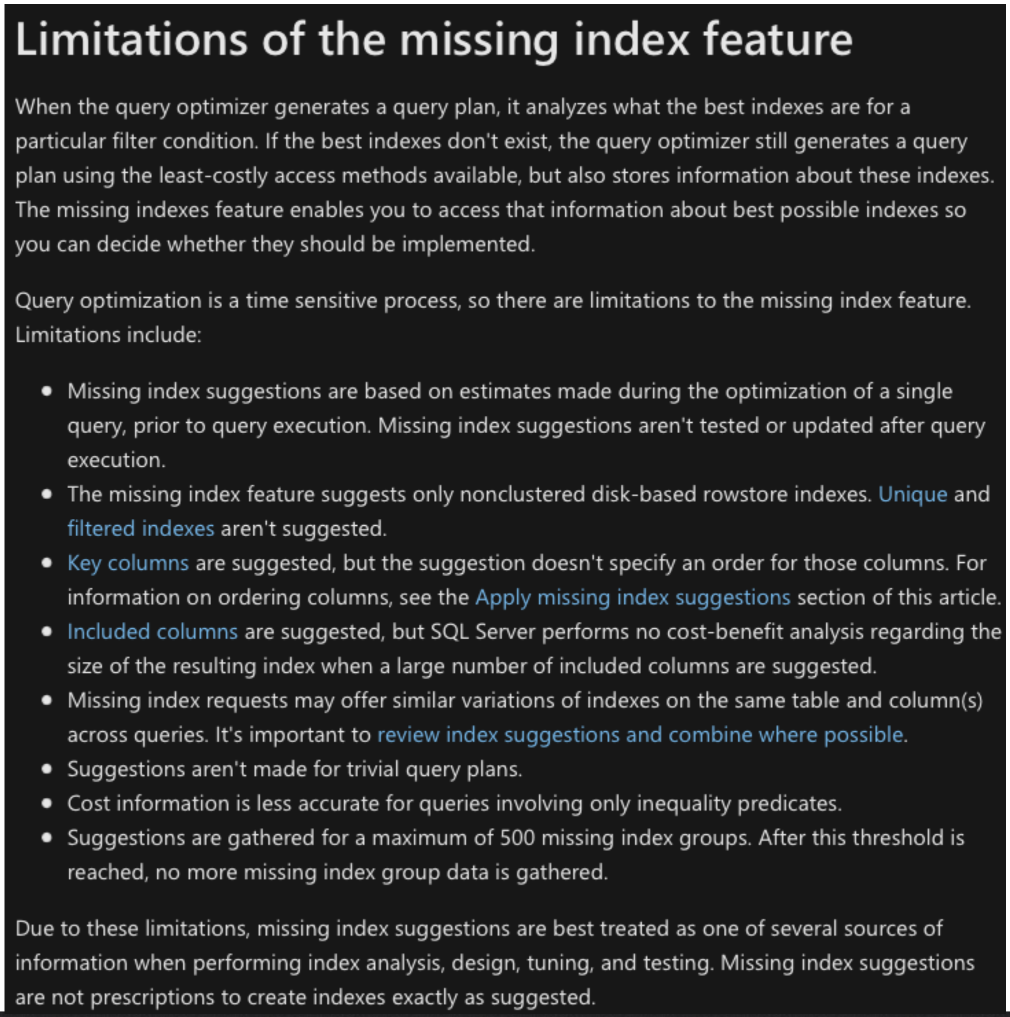


Limitations of the Missing Indexes Feature

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.



Limitations of the missing index feature

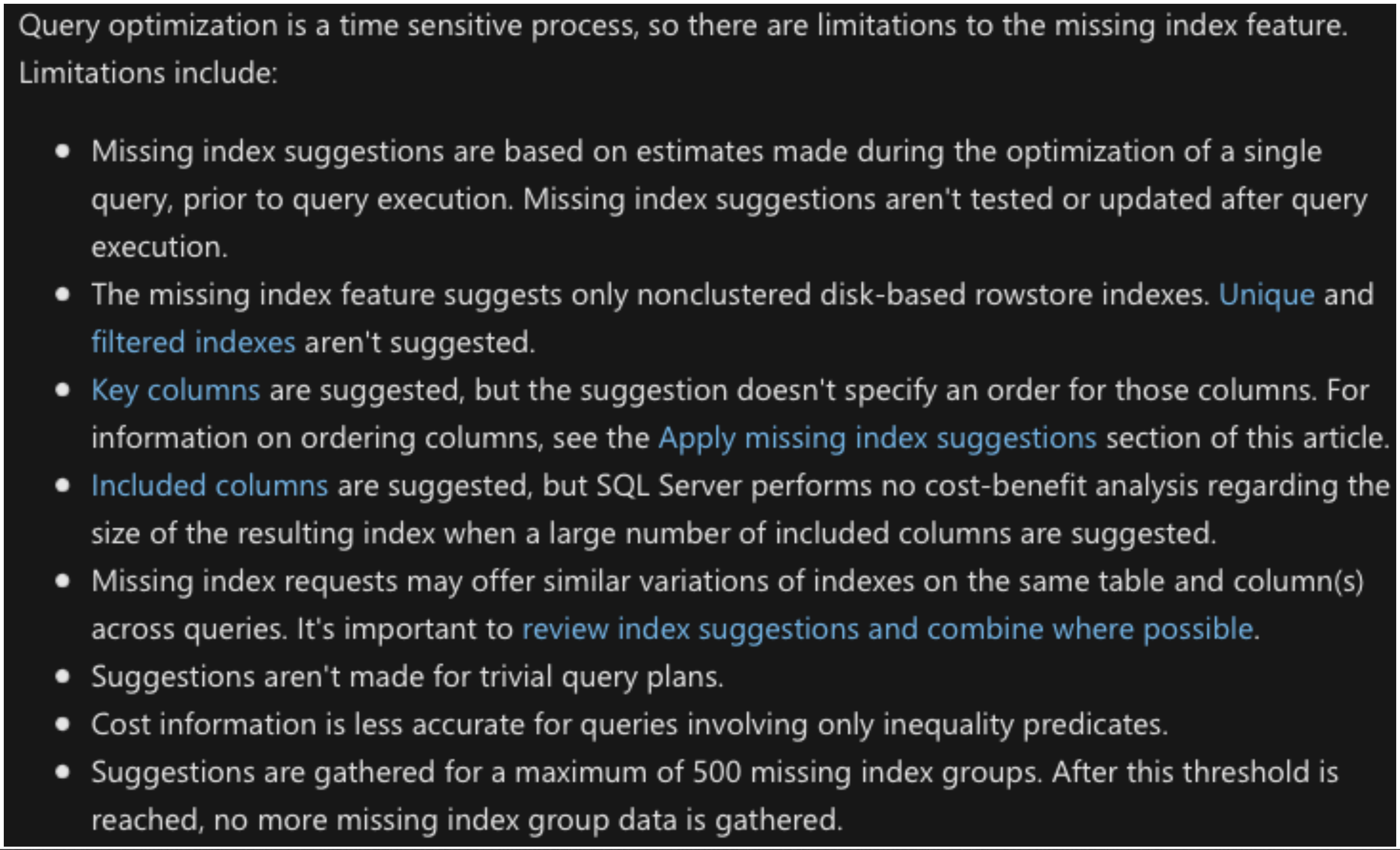
When the query optimizer generates a query plan, it analyzes what the best indexes are for a particular filter condition. If the best indexes don't exist, the query optimizer still generates a query plan using the least-costly access methods available, but also stores information about these indexes. The missing indexes feature enables you to access that information about best possible indexes so you can decide whether they should be implemented.

Query optimization is a time sensitive process, so there are limitations to the missing index feature. Limitations include:

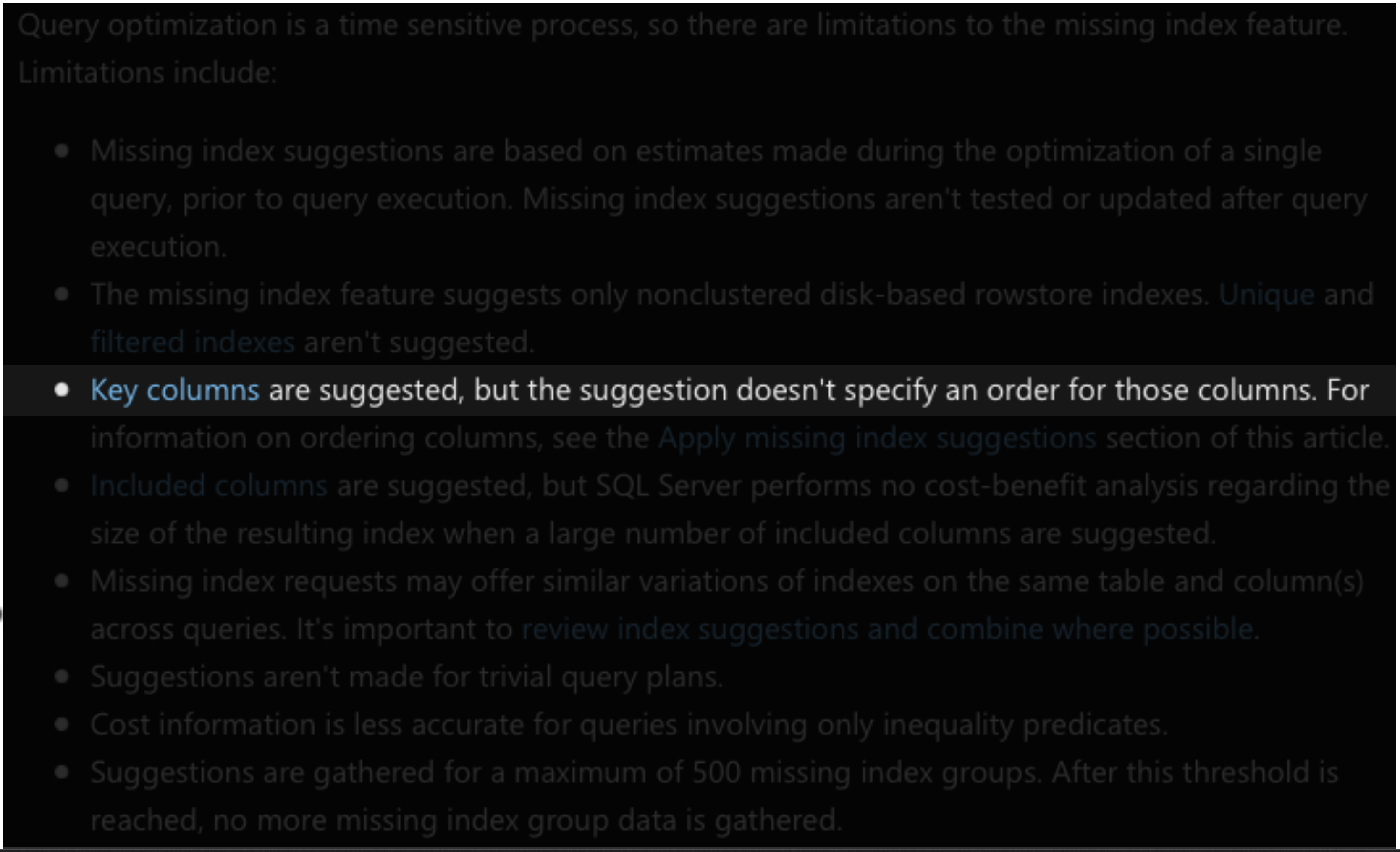
- Missing index suggestions are based on estimates made during the optimization of a single query, prior to query execution. Missing index suggestions aren't tested or updated after query execution.
- The missing index feature suggests only nonclustered disk-based rowstore indexes. Unique and filtered indexes aren't suggested.
- Key columns are suggested, but the suggestion doesn't specify an order for those columns. For information on ordering columns, see the Apply missing index suggestions section of this article.
- Included columns are suggested, but SQL Server performs no cost-benefit analysis regarding the size of the resulting index when a large number of included columns are suggested.
- Missing index requests may offer similar variations of indexes on the same table and column(s) across queries. It's important to review index suggestions and combine where possible.
- Suggestions aren't made for trivial query plans.
- Cost information is less accurate for queries involving only inequality predicates.
- Suggestions are gathered for a maximum of 500 missing index groups. After this threshold is reached, no more missing index group data is gathered.

Due to these limitations, missing index suggestions are best treated as one of several sources of information when performing index analysis, design, tuning, and testing. Missing index suggestions are not prescriptions to create indexes exactly as suggested.

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


Query optimization is a time sensitive process, so there are limitations to the missing index feature. Limitations include:

- Missing index suggestions are based on estimates made during the optimization of a single query, prior to query execution. Missing index suggestions aren't tested or updated after query execution.
- The missing index feature suggests only nonclustered disk-based rowstore indexes. Unique and filtered indexes aren't suggested.
- Key columns are suggested, but the suggestion doesn't specify an order for those columns. For information on ordering columns, see the Apply missing index suggestions section of this article.
- Included columns are suggested, but SQL Server performs no cost-benefit analysis regarding the size of the resulting index when a large number of included columns are suggested.
- Missing index requests may offer similar variations of indexes on the same table and column(s) across queries. It's important to review index suggestions and combine where possible.
- Suggestions aren't made for trivial query plans.
- Cost information is less accurate for queries involving only inequality predicates.
- Suggestions are gathered for a maximum of 500 missing index groups. After this threshold is reached, no more missing index group data is gathered.

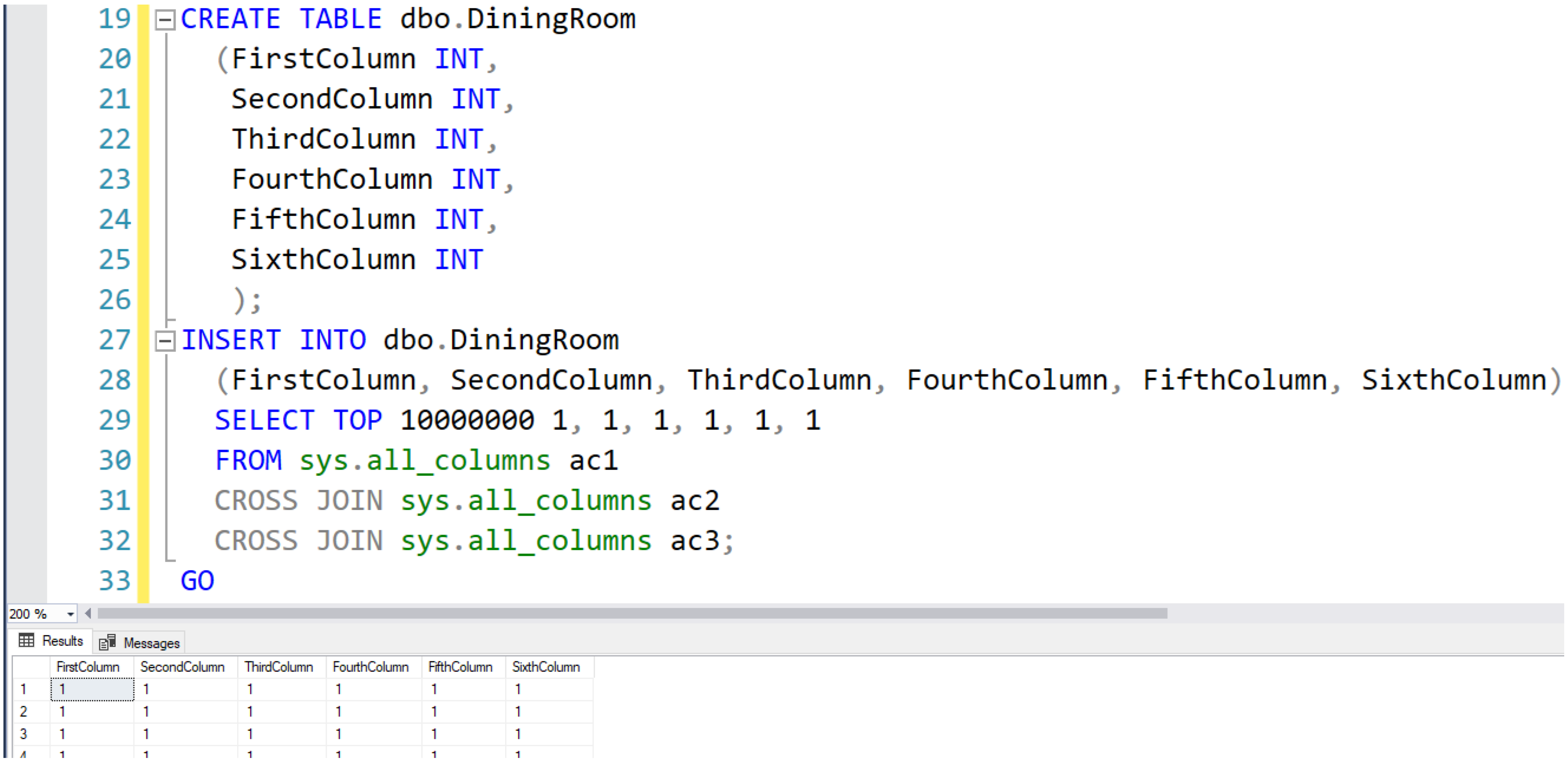


Query optimization is a time sensitive process, so there are limitations to the missing index feature. Limitations include:

- Missing index suggestions are based on estimates made during the optimization of a single query, prior to query execution. Missing index suggestions aren't tested or updated after query execution.
- The missing index feature suggests only nonclustered disk-based rowstore indexes. Unique and filtered indexes aren't suggested.
- Key columns are suggested, but the suggestion doesn't specify an order for those columns. For information on ordering columns, see the Apply missing index suggestions section of this article.
- Included columns are suggested, but SQL Server performs no cost-benefit analysis regarding the size of the resulting index when a large number of included columns are suggested.
- Missing index requests may offer similar variations of indexes on the same table and column(s) across queries. It's important to review index suggestions and combine where possible.
- Suggestions aren't made for trivial query plans.
- Cost information is less accurate for queries involving only inequality predicates.
- Suggestions are gathered for a maximum of 500 missing index groups. After this threshold is reached, no more missing index group data is gathered.

Let's see how he does it.

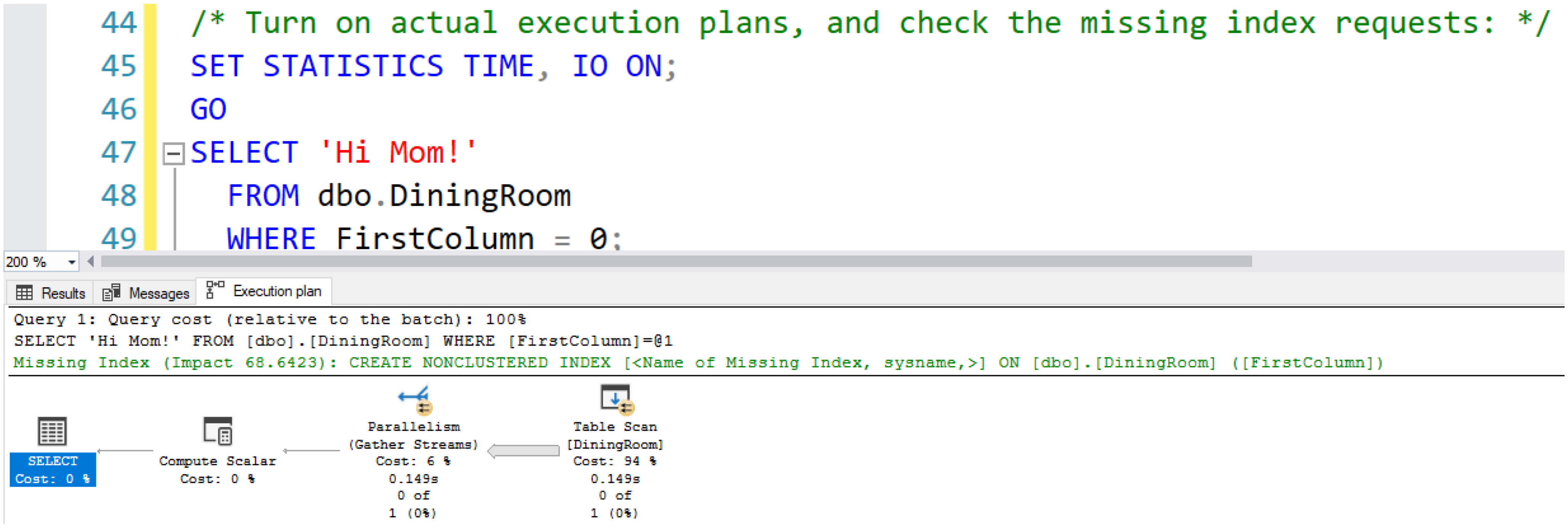
Create table w/10M identical rows



```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
```

	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
4	1	1	1	1	1	1

Single-column equality search



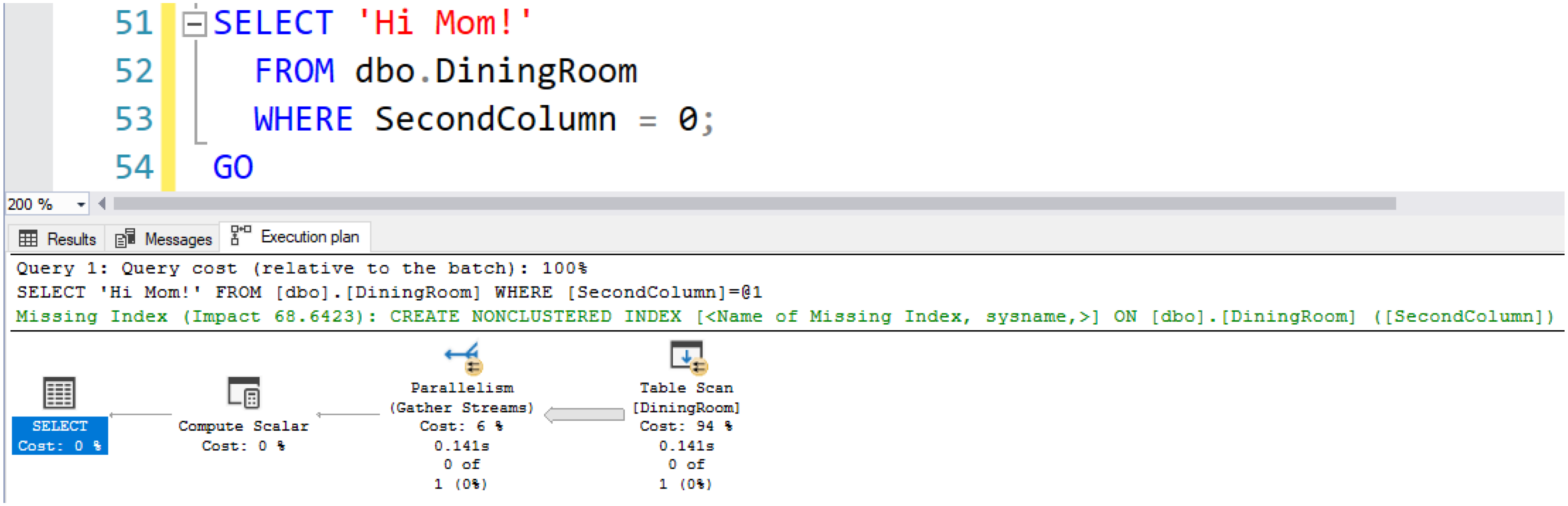
```sql
/* Turn on actual execution plans, and check the missing index requests: */
SET STATISTICS TIME, IO ON;
GO
SELECT 'Hi Mom!'
  FROM dbo.DiningRoom
  WHERE FirstColumn = 0;
```

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])
```

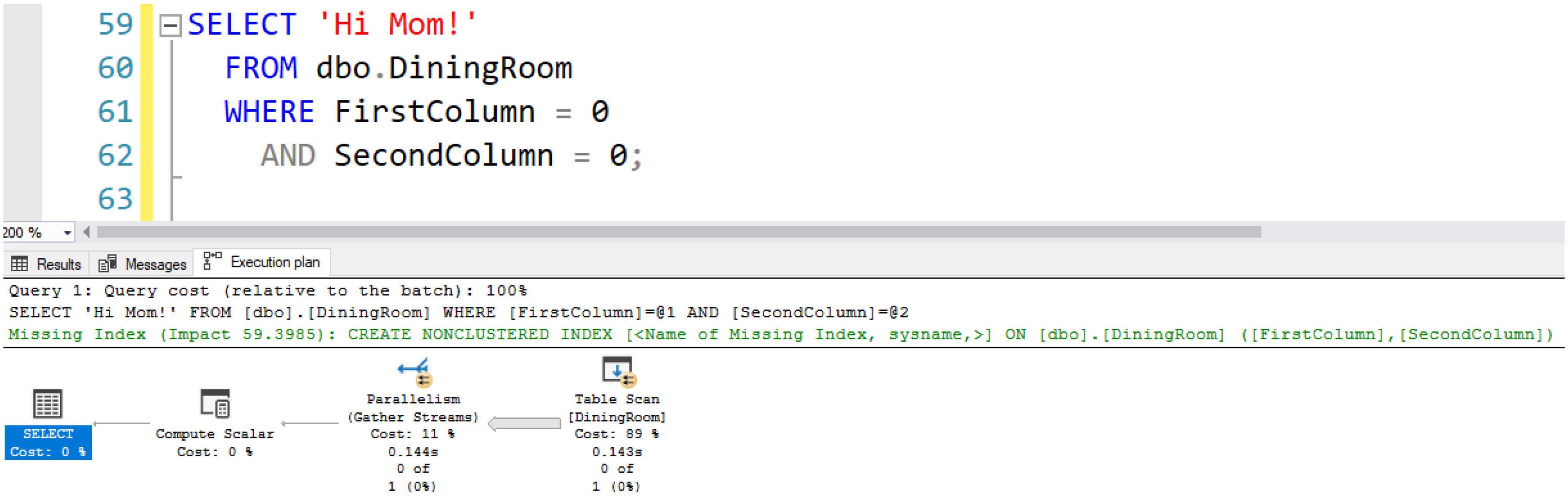
Also works if we look for column 2



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

```
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…



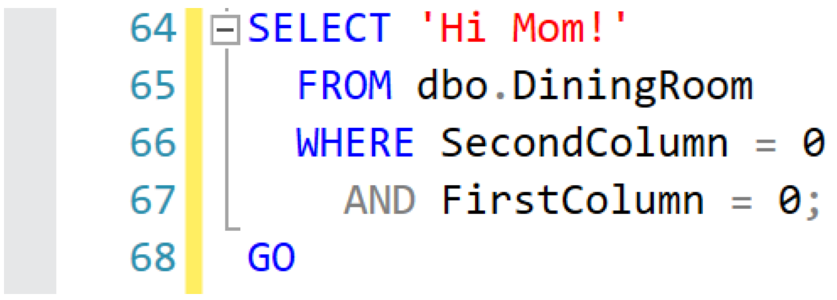
```
SELECT 'Hi Mom!'
  FROM dbo.DiningRoom
  WHERE FirstColumn = 0
    AND SecondColumn = 0;
```

```
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])
```

So far, not bad.

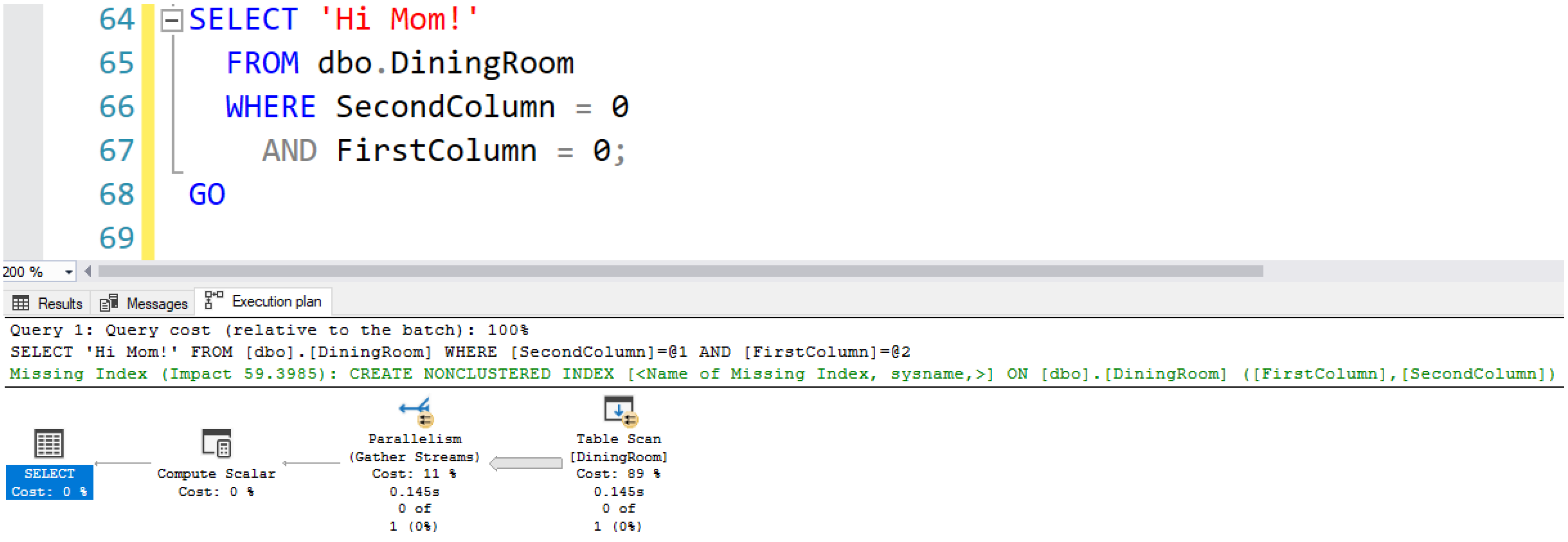
And if we flip the WHERE clause?

What if we put SecondColumn first?



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

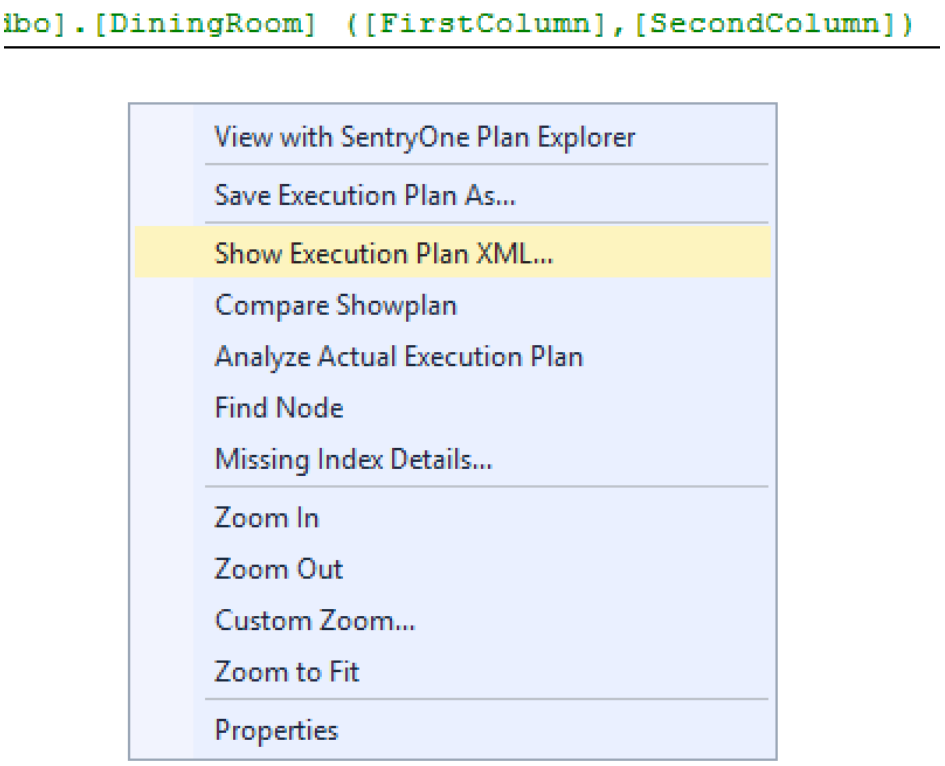
Hmm…what's determining order?



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

```
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])
```

View the execution plan XML



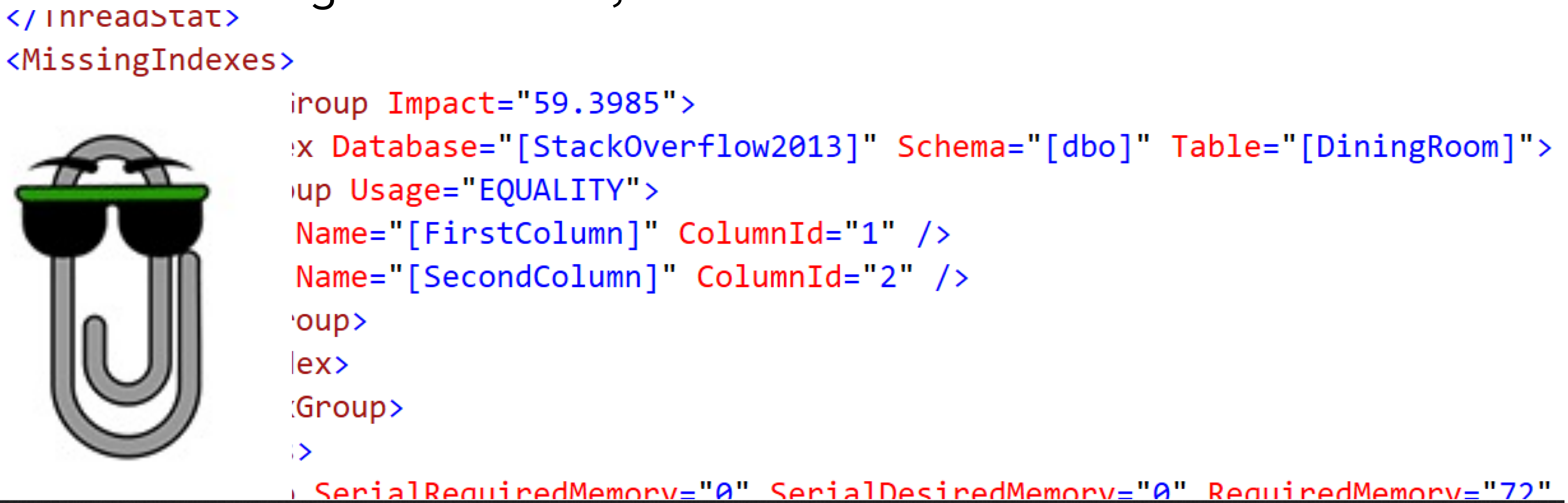
```
ibo].[DiningRoom] ([FirstColumn],[SecondColumn])
```

View with SentryOne Plan Explorer
Save Execution Plan As...
Show Execution Plan XML...
Compare Showplan
Analyze Actual Execution Plan
Find Node
Missing Index Details...
Zoom In
Zoom Out
Custom Zoom...
Zoom to Fit
Properties

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>
          ex>
          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

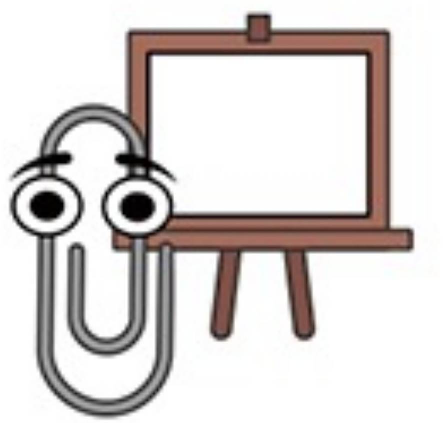
Clippy can’t consider

How often you filter on a field

How selective your filter clause is

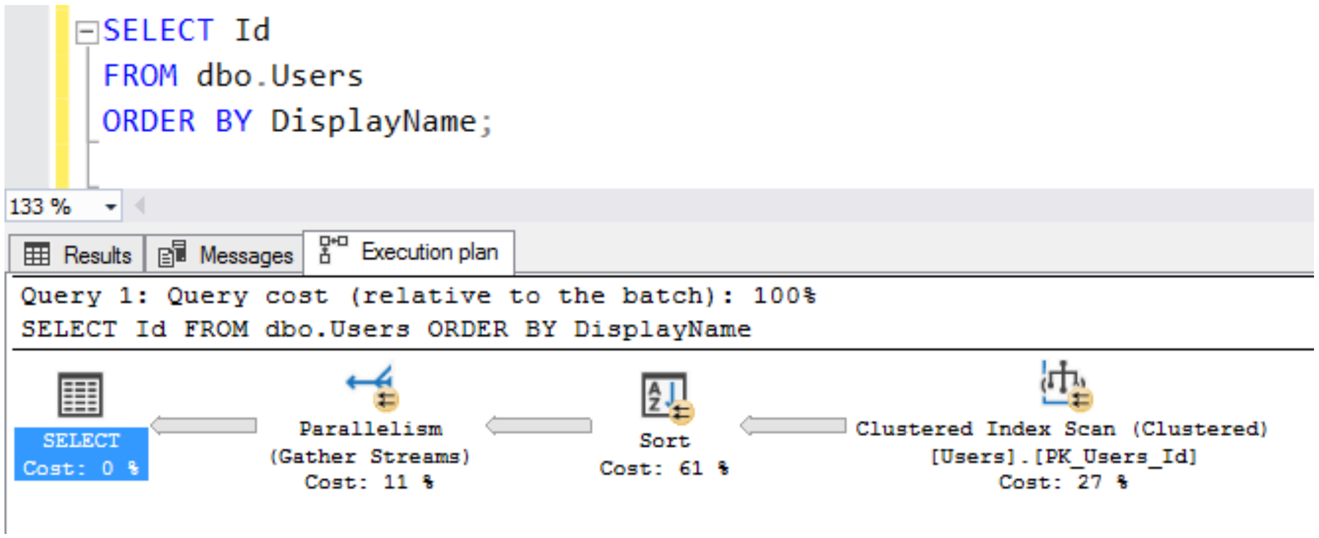
The size of the field

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



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

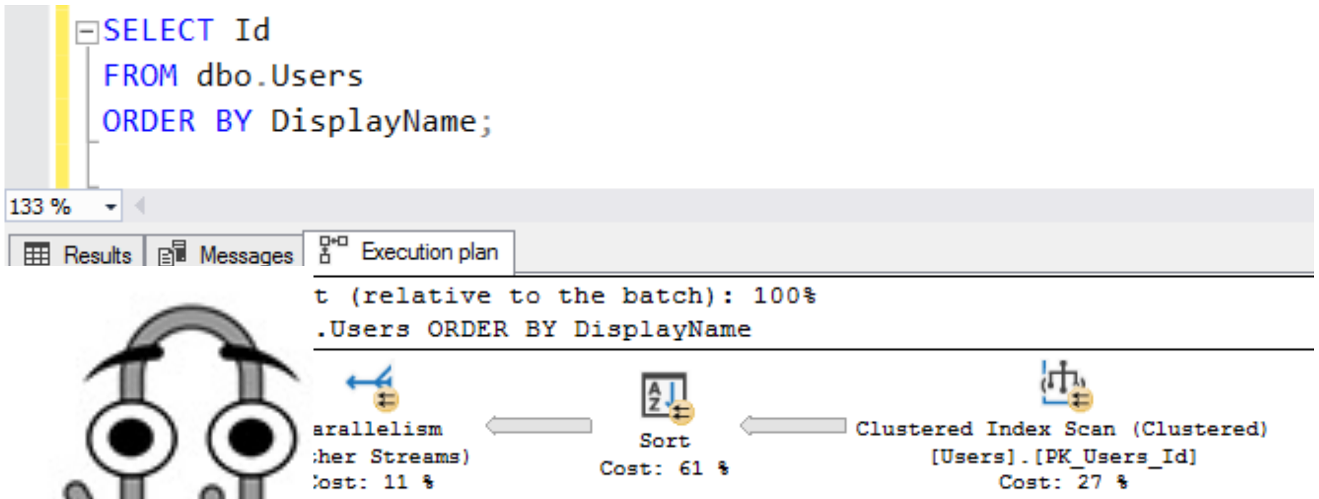
Order the whole table



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

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

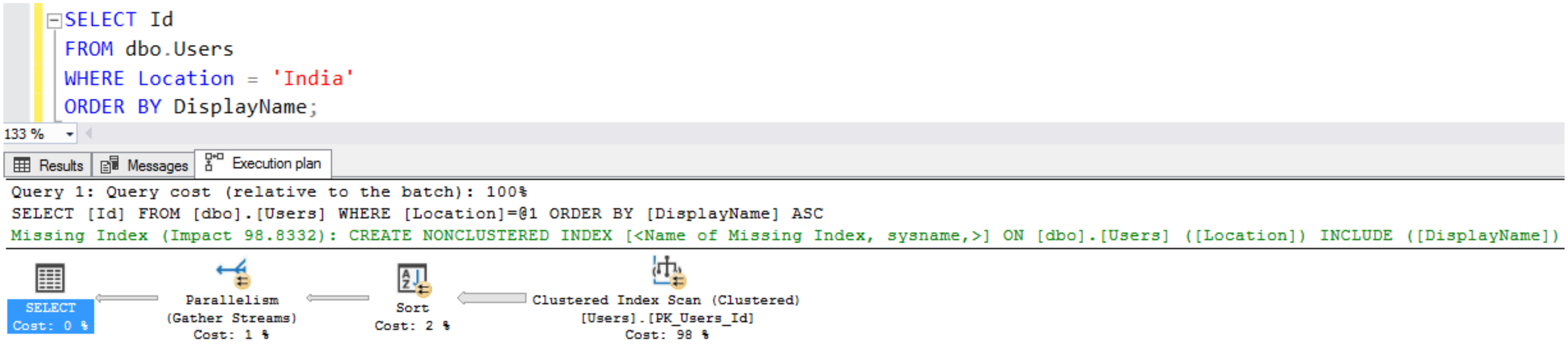
Order the whole table



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

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

Filter, then order by



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

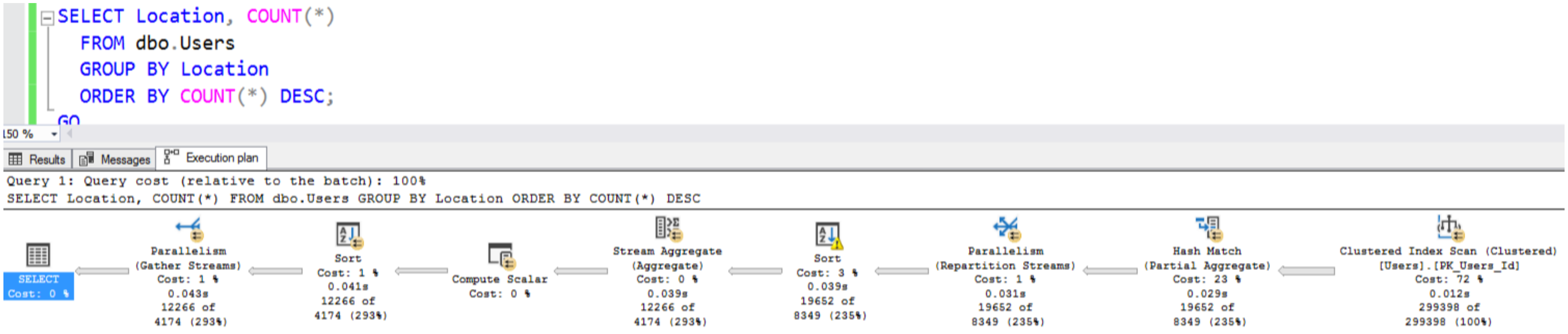
```
Query 1: Query cost (relative to the batch): 100%
SELECT [Id] FROM [dbo].[Users] WHERE [Location]=@1 ORDER BY [DisplayName] ASC
Missing Index (Impact 98.8332): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users] ([Location]) INCLUDE ([DisplayName])
```

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
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT Location, COUNT(*) FROM dbo.Users GROUP BY Location ORDER BY COUNT(*) DESC
```

```
SELECT        Parallelism        Sort        Compute Scalar   Stream Aggregate   Sort         Parallelism             Hash Match            Clustered Index Scan (Clustered)
Cost: 0 %     (Gather Streams)   Cost: 1 %   Cost: 0 %        (Aggregate)        Cost: 3 %    (Repartition Streams)   (Partial Aggregate)   [Users].[PK_Users_Id]
              Cost: 1 %          0.041s                       Cost: 0 %          0.039s       Cost: 1 %               Cost: 23 %            Cost: 72 %
              0.043s             12266 of                     0.039s             19652 of     0.031s                  0.029s                0.012s
              12266 of           4174 (293%)                  12266 of           8349 (235%)  19652 of                19652 of              299398 of
              4174 (293%)                                     4174 (293%)                     8349 (235%)             8349 (235%)           299398 (100%)
```

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

But no index recommendation?

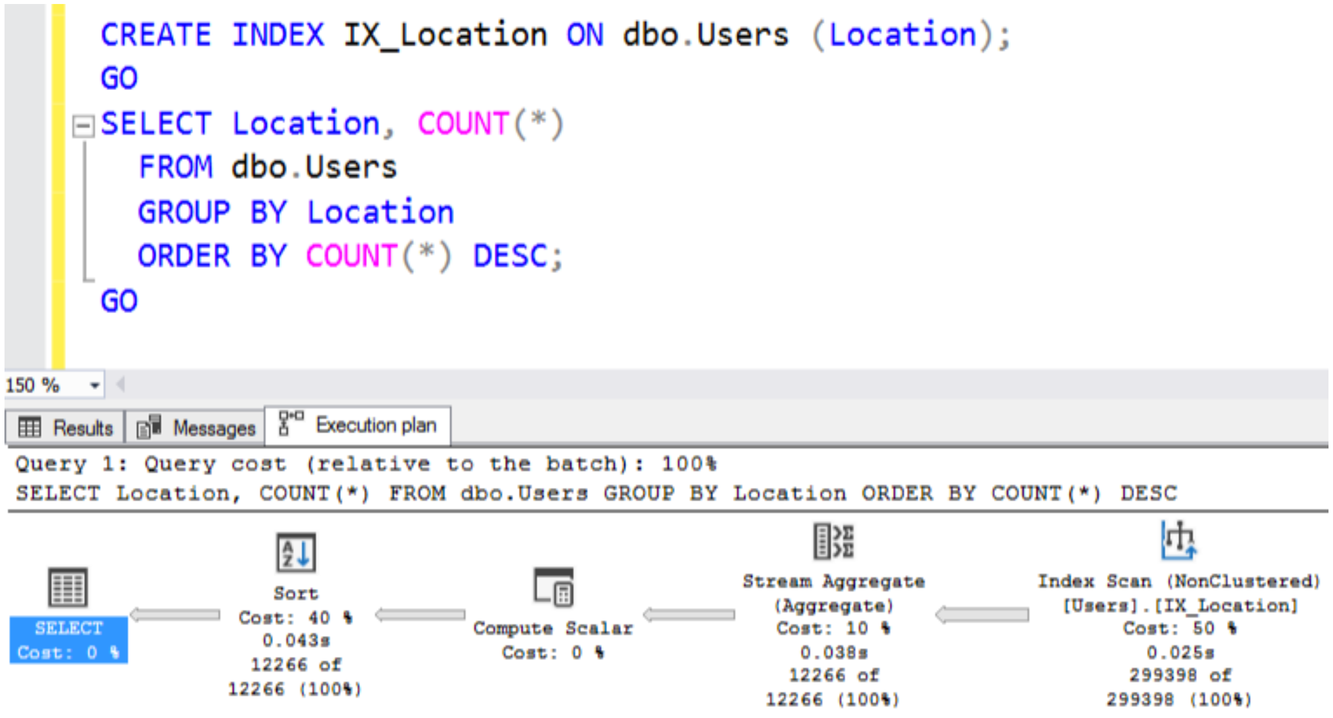
Try creating one by hand.

```
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



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

```
Query 1: Query cost (relative to the batch): 100%
SELECT Location, COUNT(*) FROM dbo.Users GROUP BY Location ORDER BY COUNT(*) DESC
```

```
SELECT        Sort            Compute Scalar   Stream Aggregate   Index Scan (NonClustered)
Cost: 0 %     Cost: 40 %      Cost: 0 %        (Aggregate)        [Users].[IX_Location]
              0.043s                           Cost: 10 %         Cost: 50 %
              12266 of                         0.038s             0.025s
              12266 (100%)                     12266 of           299398 of
                                               12266 (100%)       299398 (100%)
```

Way faster

Single-threaded

Great estimates

No spills to disk

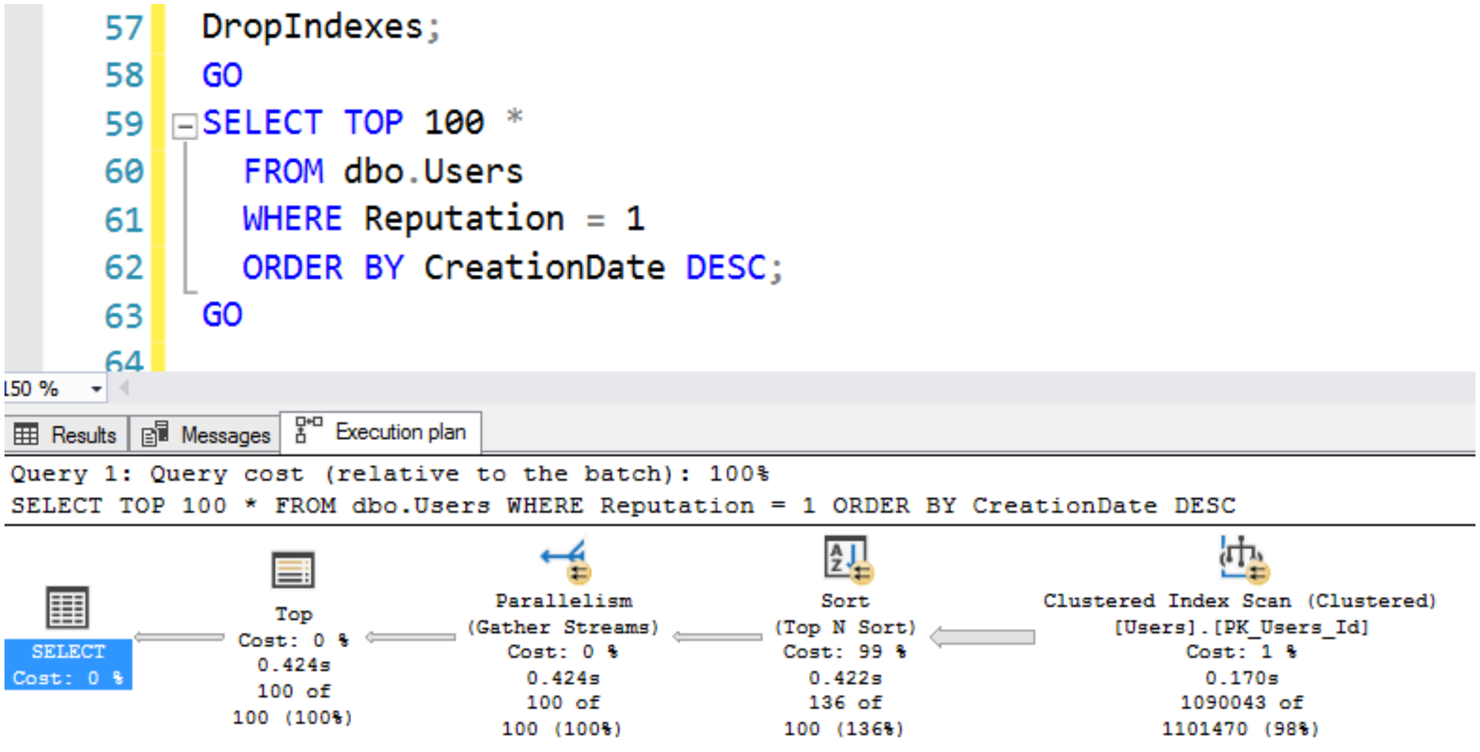
Adding Clippy's indexes can even make things worse.

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

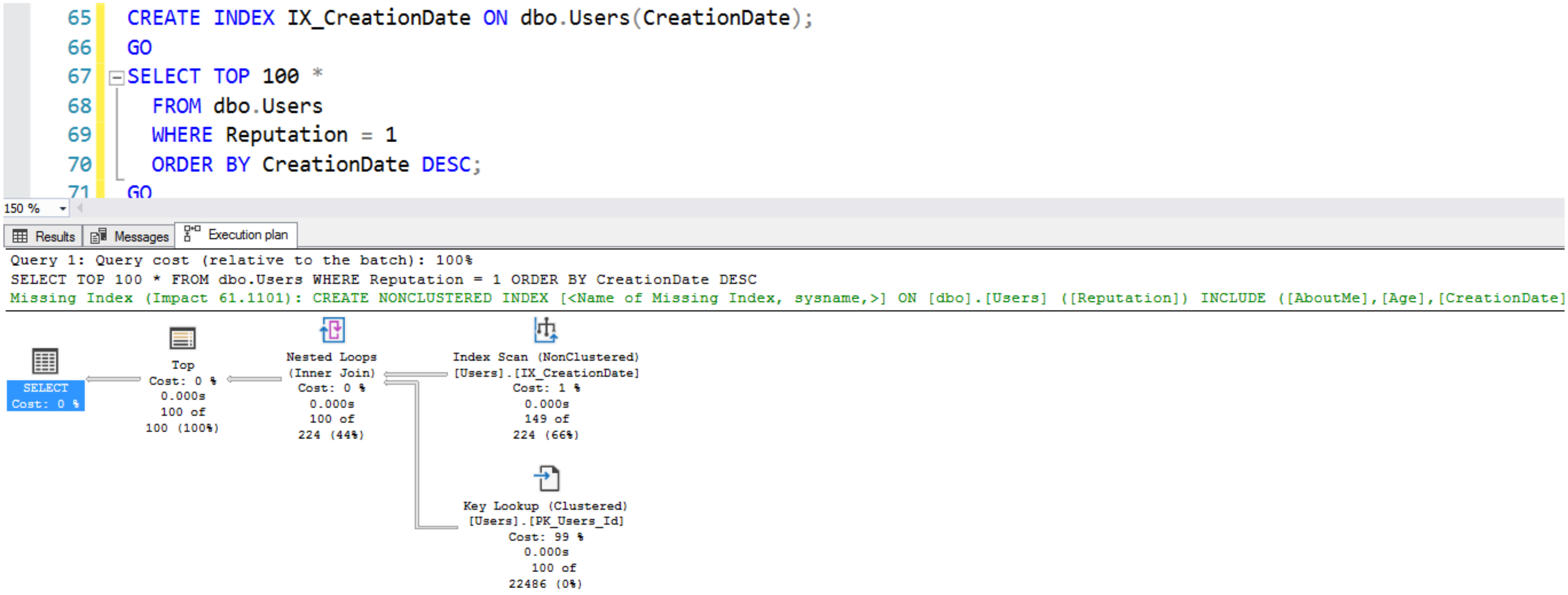
Try this query with no indexes.



```
DropIndexes;
GO
SELECT TOP 100 *
  FROM dbo.Users
  WHERE Reputation = 1
  ORDER BY CreationDate DESC;
GO
```

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

Add an index, and it's fast!

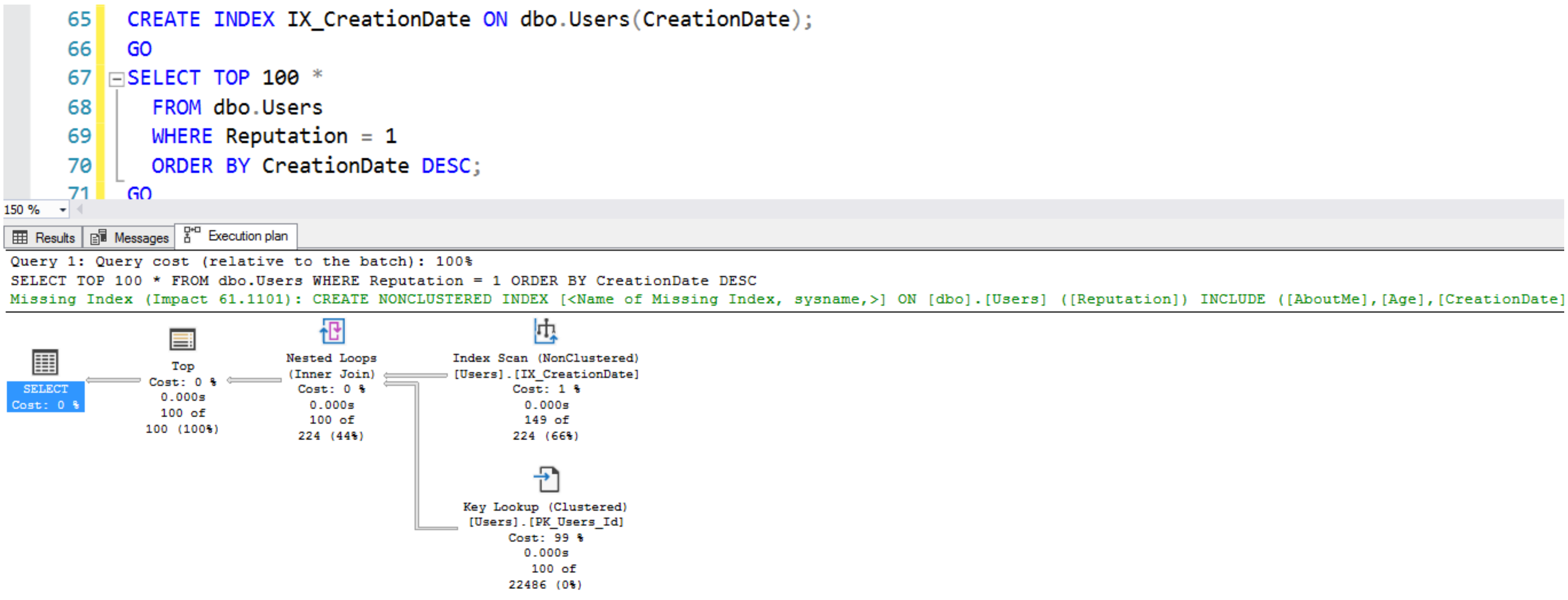


```sql
CREATE INDEX IX_CreationDate ON dbo.Users(CreationDate);
GO
SELECT TOP 100 *
  FROM dbo.Users
  WHERE Reputation = 1
  ORDER BY CreationDate DESC;
GO
```

```
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]
```

```
SELECT       Top            Nested Loops     Index Scan (NonClustered)
Cost: 0 %    Cost: 0 %      (Inner Join)     [Users].[IX_CreationDate]
             0.000s         Cost: 0 %        Cost: 1 %
             100 of         0.000s           0.000s
             100 (100%)     100 of           149 of
                            224 (44%)        224 (66%)

                                             Key Lookup (Clustered)
                                             [Users].[PK_Users_Id]
                                             Cost: 99 %
                                             0.000s
                                             100 of
                                             22486 (0%)
```

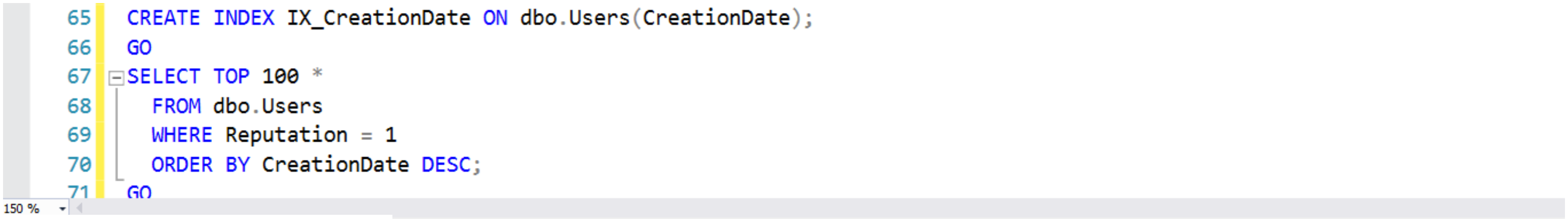
Add an index, and it’s fast, but…



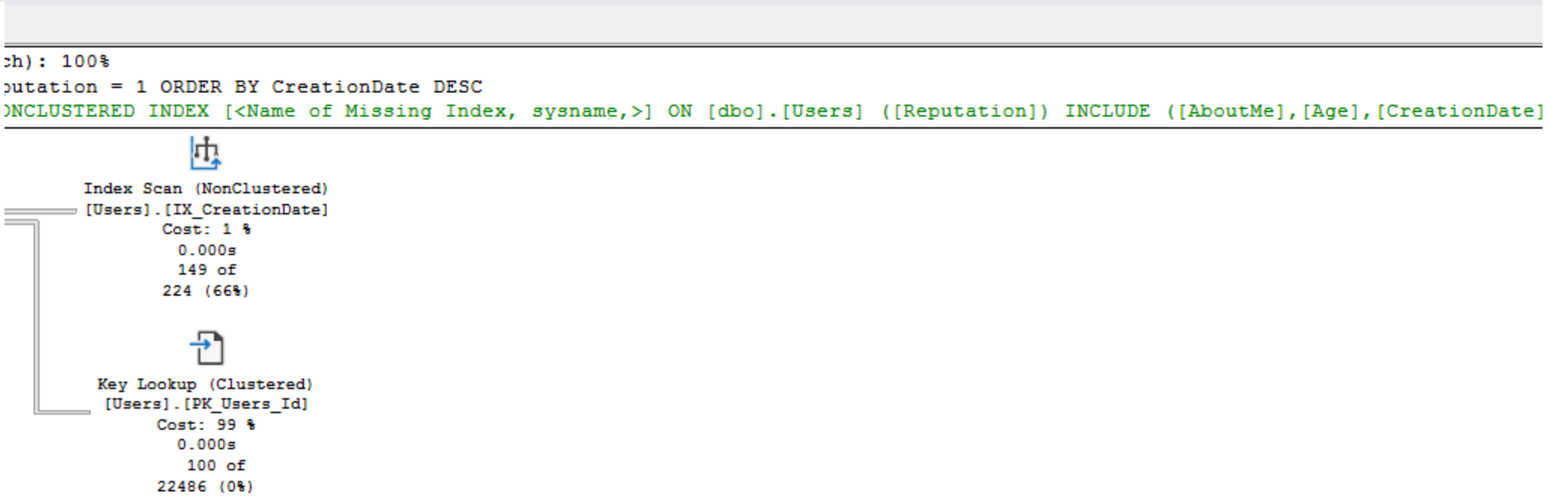
```sql
CREATE INDEX IX_CreationDate ON dbo.Users(CreationDate);
GO
SELECT TOP 100 *
  FROM dbo.Users
  WHERE Reputation = 1
  ORDER BY CreationDate DESC;
GO
```

```
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]
```

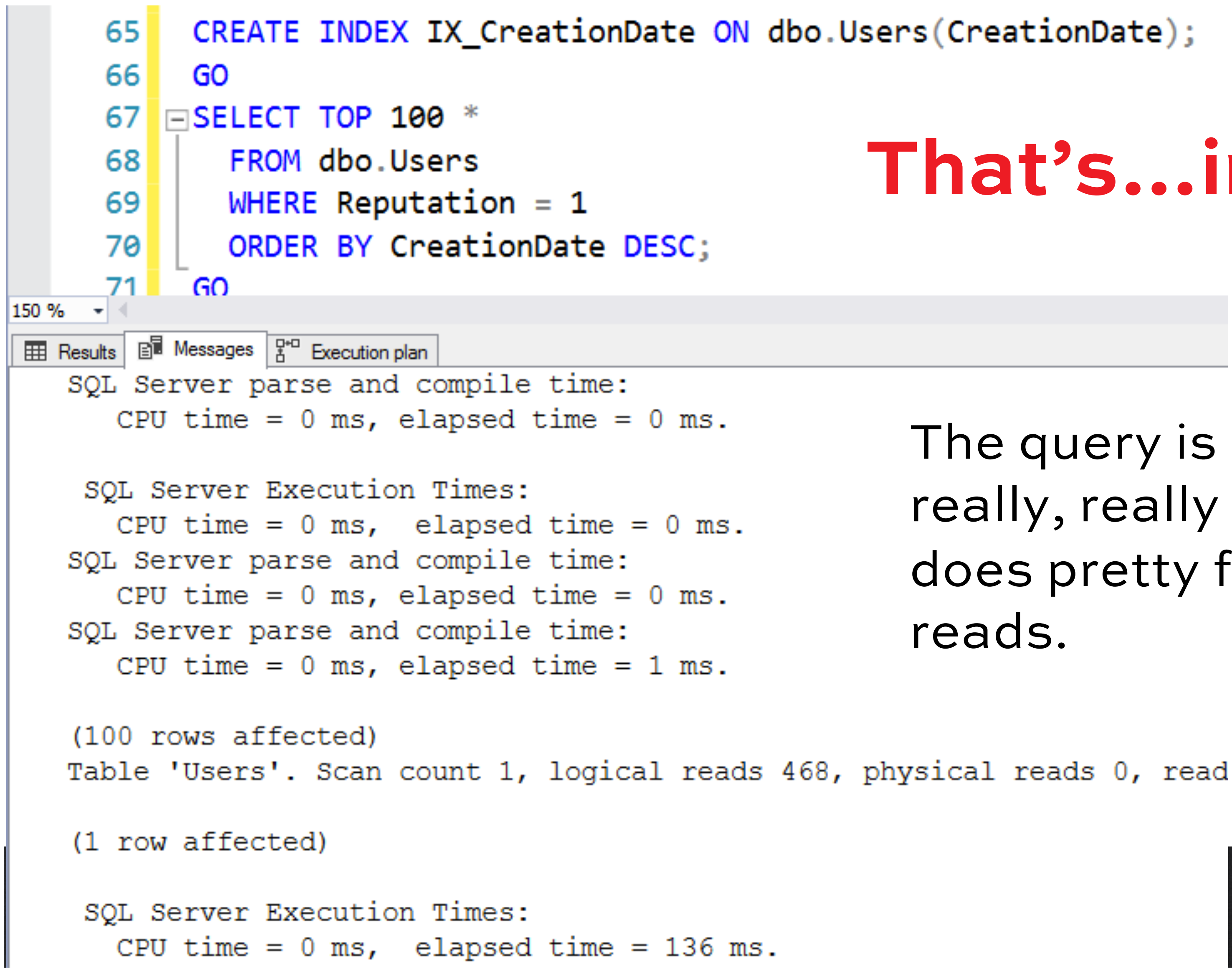
Now he's got an idea.



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



```
ch): 100%
putation = 1 ORDER BY CreationDate DESC
ONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users] ([Reputation]) INCLUDE ([AboutMe],[Age],[CreationDate]
```



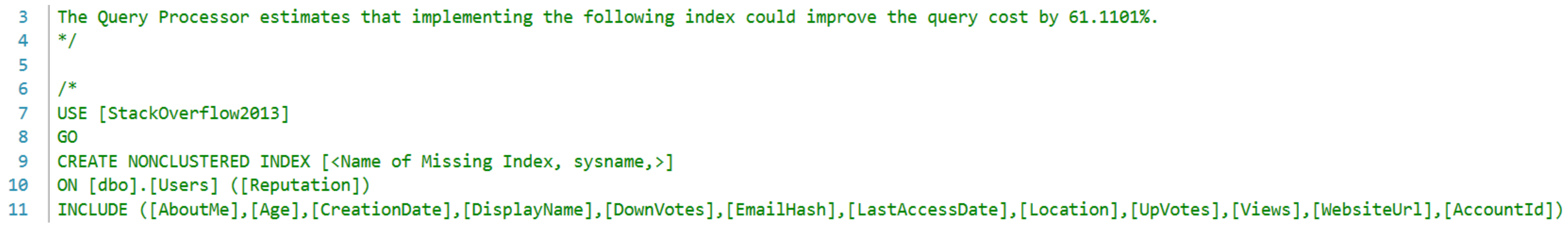
```sql
CREATE INDEX IX_CreationDate ON dbo.Users(CreationDate);
GO
SELECT TOP 100 *
  FROM dbo.Users
  WHERE Reputation = 1
  ORDER BY CreationDate DESC;
GO
```

```
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.
```

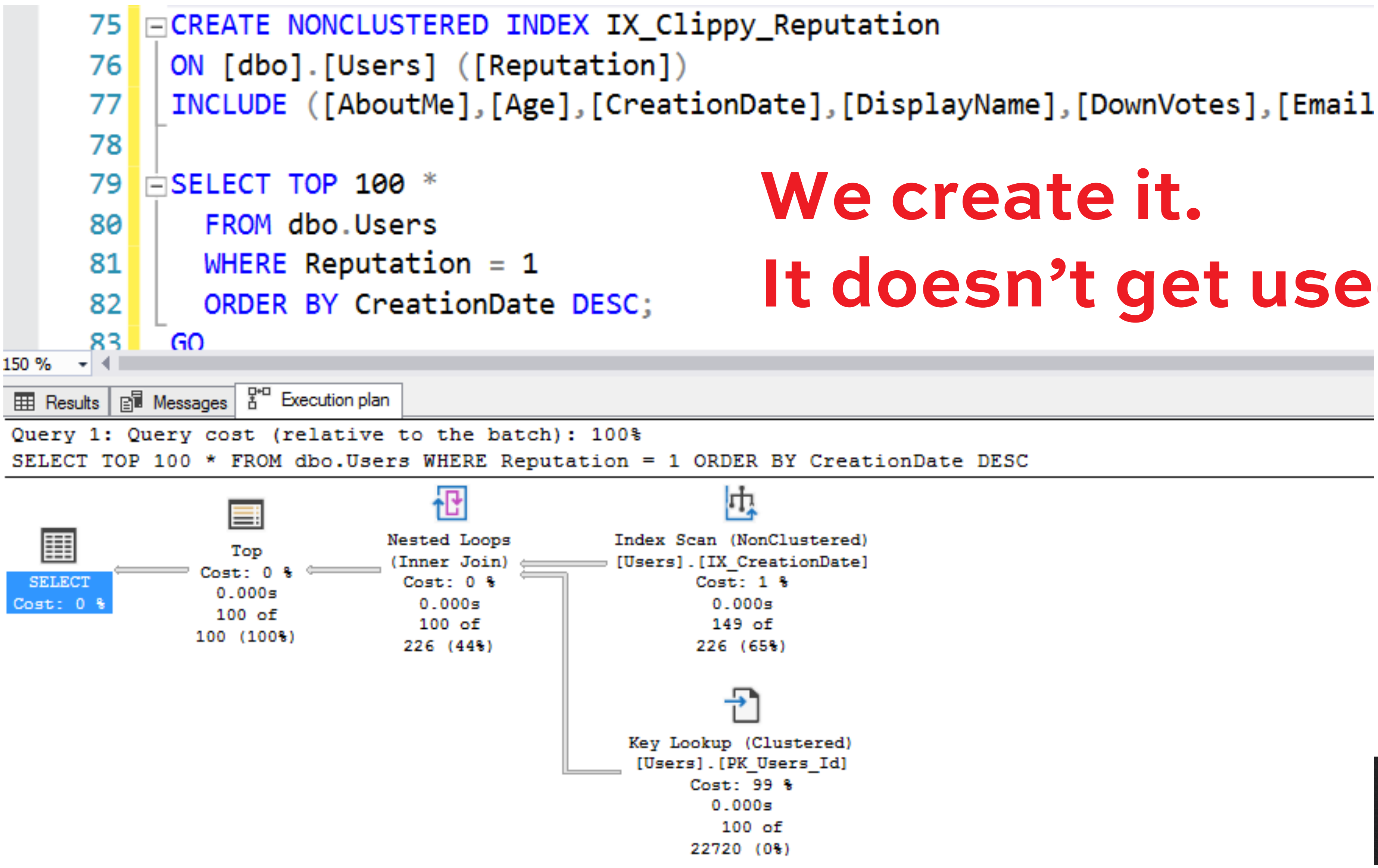
That's...interesting.

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.

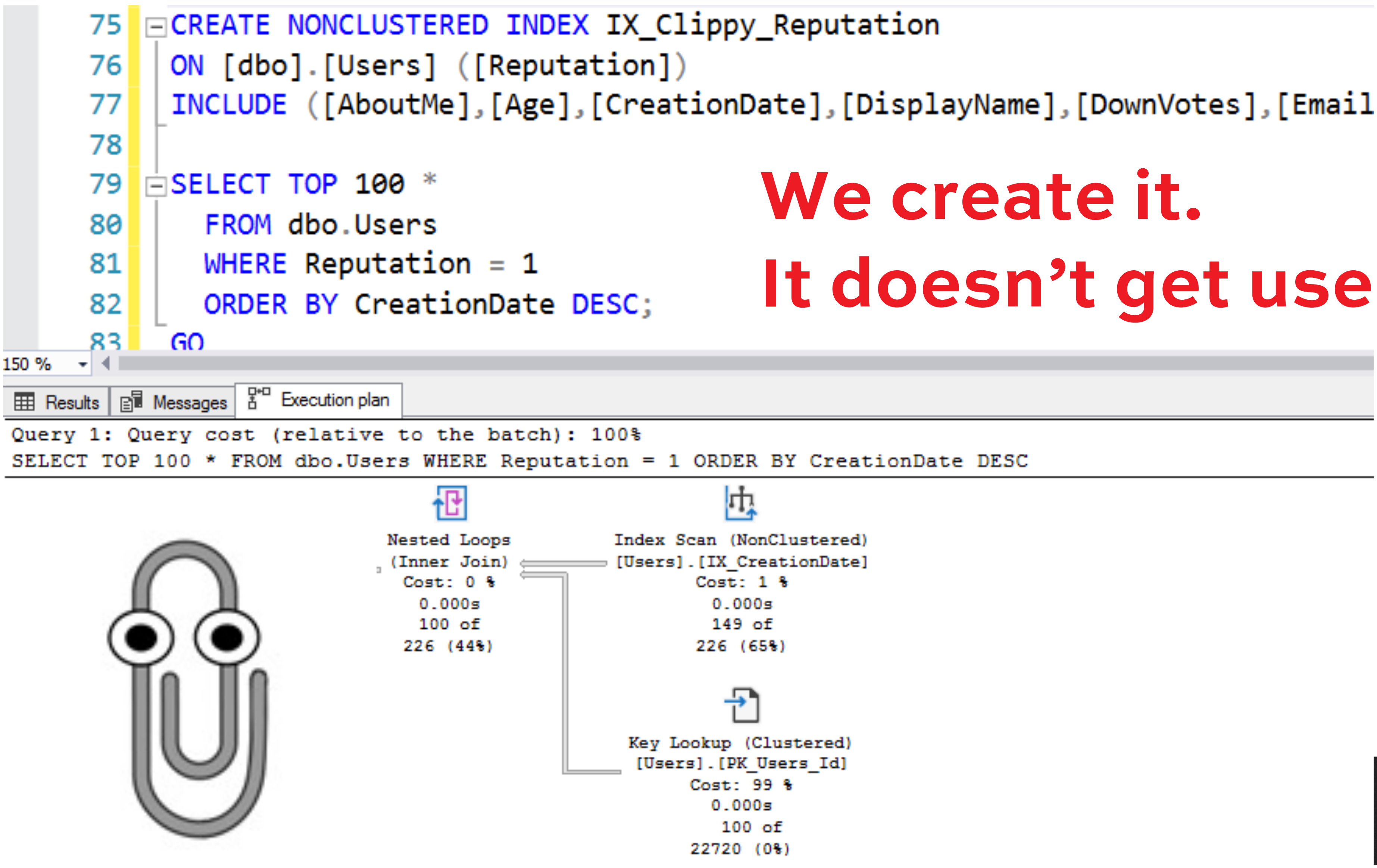


```
CREATE NONCLUSTERED INDEX IX_Clippy_Reputation
ON [dbo].[Users] ([Reputation])
INCLUDE ([AboutMe],[Age],[CreationDate],[DisplayName],[DownVotes],[Email

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

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

We create it.

It doesn't get used!



```
CREATE NONCLUSTERED INDEX IX_Clippy_Reputation
ON [dbo].[Users] ([Reputation])
INCLUDE ([AboutMe],[Age],[CreationDate],[DisplayName],[DownVotes],[Email

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

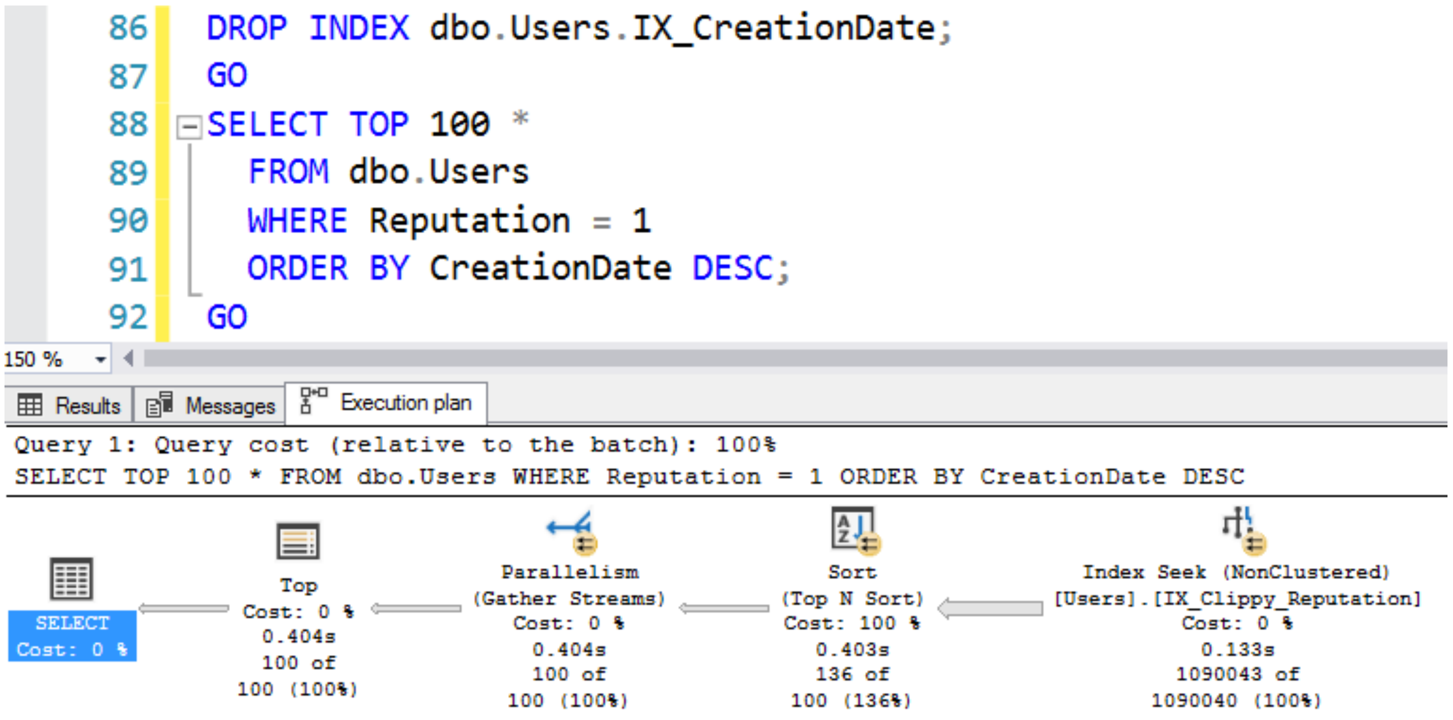
```
Query 1: Query cost (relative to the batch): 100%
SELECT TOP 100 * FROM dbo.Users WHERE Reputation = 1 ORDER BY CreationDate DESC
```

```
Nested Loops        Index Scan (NonClustered)
(Inner Join)        [Users].[IX_CreationDate]
Cost: 0 %           Cost: 1 %
0.000s              0.000s
100 of              149 of
226 (44%)           226 (65%)

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

We create it.

It doesn't get used!

Drop the old IX_CreationDate...



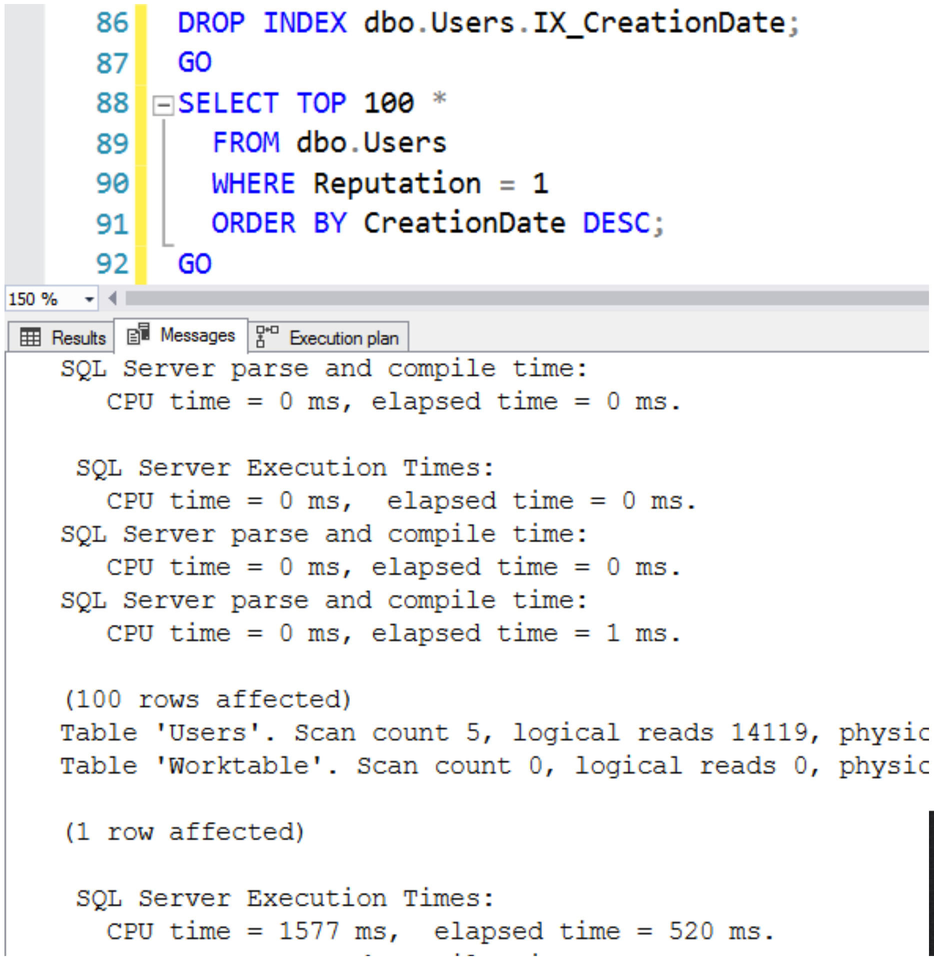
```sql
DROP INDEX dbo.Users.IX_CreationDate;
GO
SELECT TOP 100 *
  FROM dbo.Users
  WHERE Reputation = 1
  ORDER BY CreationDate DESC;
GO
```

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

```
SELECT      Top           Parallelism          Sort              Index Seek (NonClustered)
Cost: 0 %   Cost: 0 %     (Gather Streams)     (Top N Sort)      [Users].[IX_Clippy_Reputation]
            0.404s        Cost: 0 %            Cost: 100 %       Cost: 0 %
            100 of        0.404s               0.403s            0.133s
            100 (100%)    100 of               136 of            1090043 of
                          100 (100%)           100 (136%)        1090040 (100%)
```

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

Now we're sorting 1M rows.

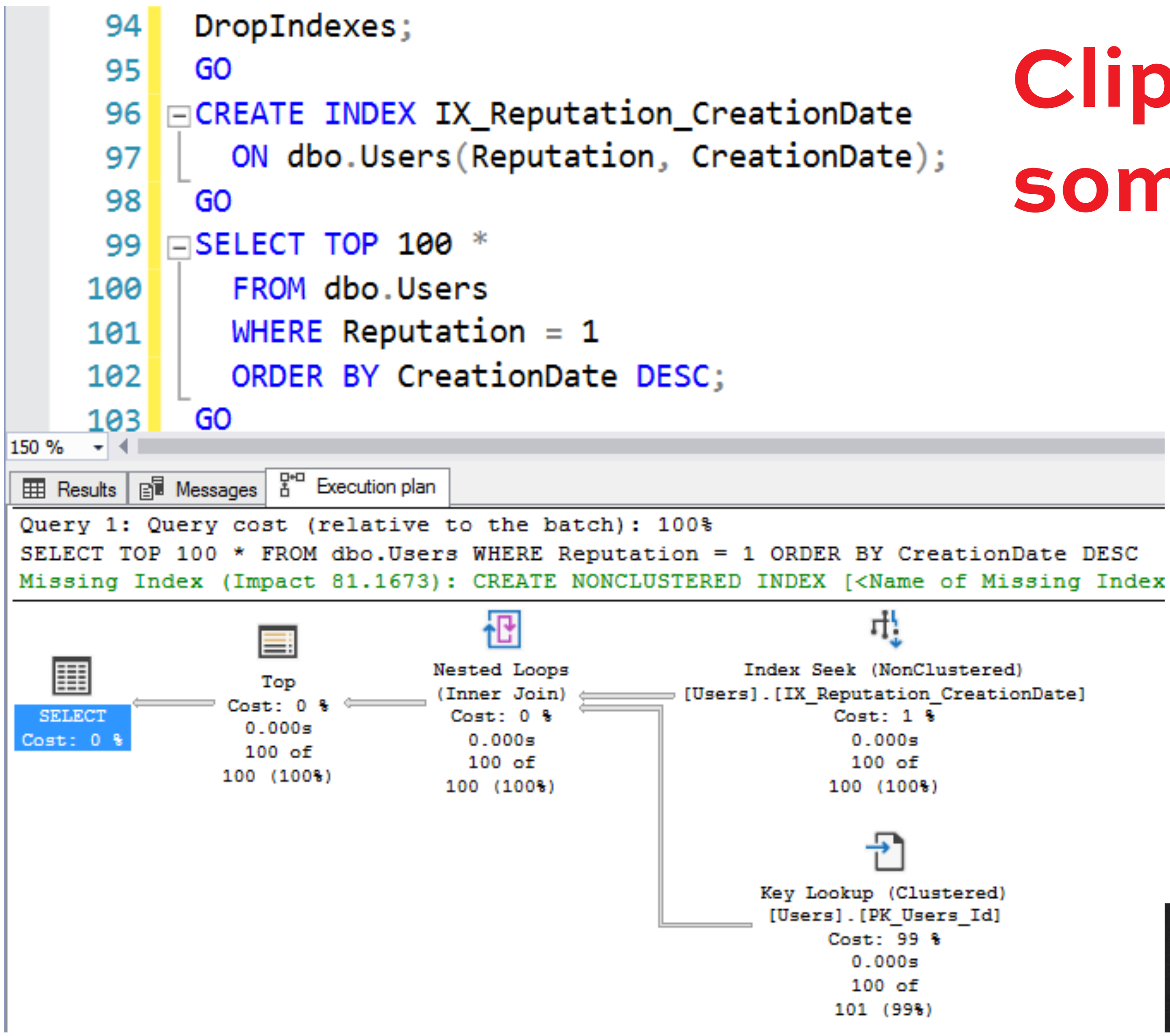


```
DROP INDEX dbo.Users.IX_CreationDate;
GO
SELECT TOP 100 *
  FROM dbo.Users
  WHERE Reputation = 1
  ORDER BY CreationDate DESC;
GO
```

```
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.

Clippy was on to something



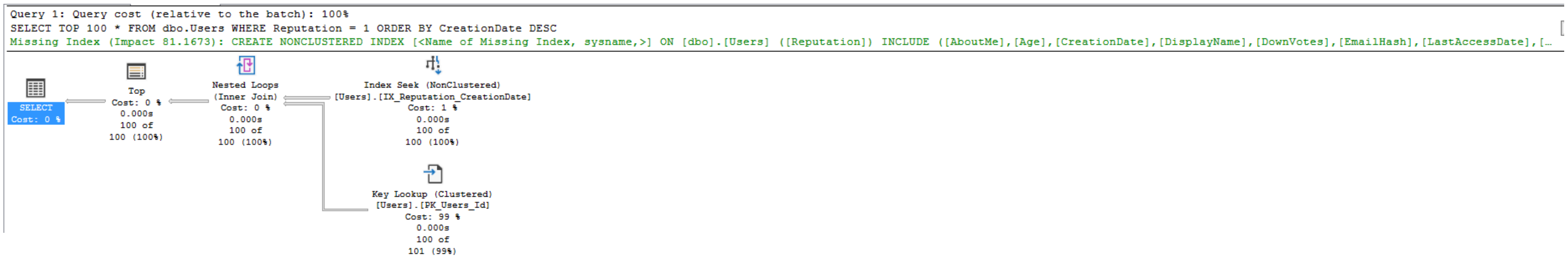
```
DropIndexes;
GO
CREATE INDEX IX_Reputation_CreationDate
  ON dbo.Users(Reputation, CreationDate);
GO
SELECT TOP 100 *
  FROM dbo.Users
  WHERE Reputation = 1
  ORDER BY CreationDate DESC;
GO
```

```
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
```

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

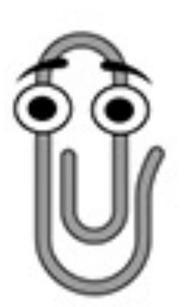
Key on both fields, and the sort is gone.

And he STILL wants the index.



```
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, sysname,>] ON [dbo].[Users] ([Reputation]) INCLUDE ([AboutMe],[Age],[CreationDate],[DisplayName],[DownVotes],[EmailHash],[LastAccessDate],[…
```

```
SELECT        Top           Nested Loops     Index Seek (NonClustered)
Cost: 0 %     Cost: 0 %     (Inner Join)     [Users].[IX_Reputation_CreationDate]
              0.000s        Cost: 0 %        Cost: 1 %
              100 of        0.000s           0.000s
              100 (100%)    100 of           100 of
                            100 (100%)       100 (100%)

                                             Key Lookup (Clustered)
                                             [Users].[PK_Users_Id]
                                             Cost: 99 %
                                             0.000s
                                             100 of
                                             101 (99%)
```



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

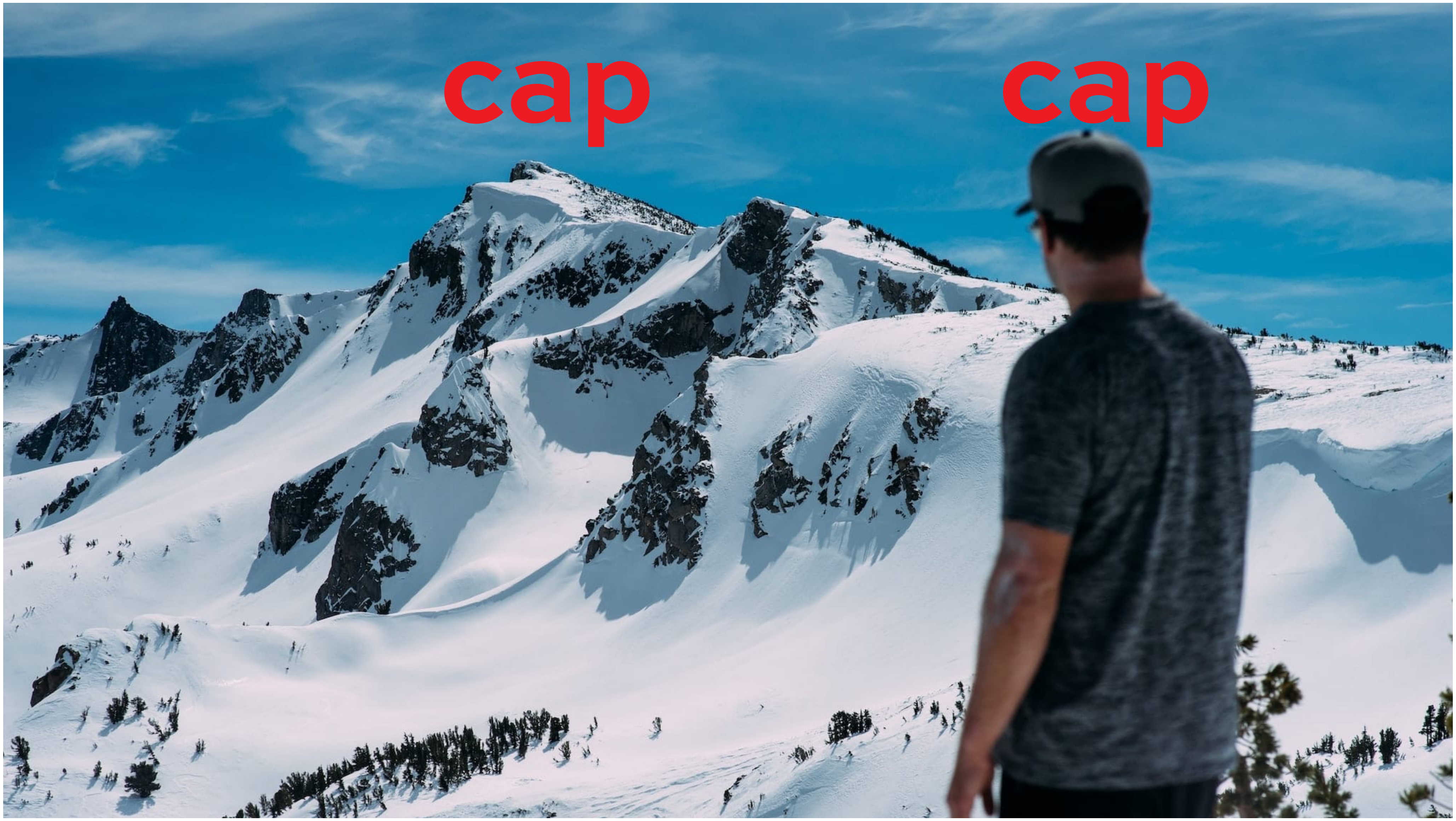
```
sp_BlitzIndex @GetAllDatabases = 1;
```

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.


cap
cap

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.