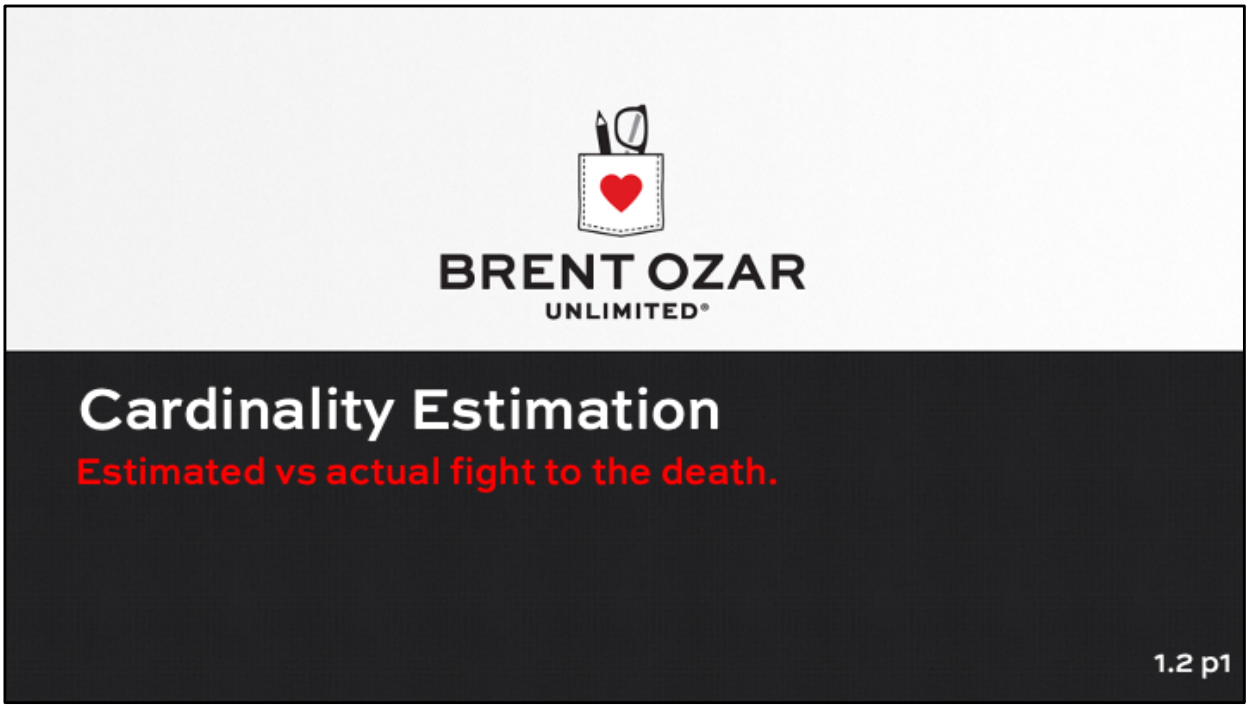
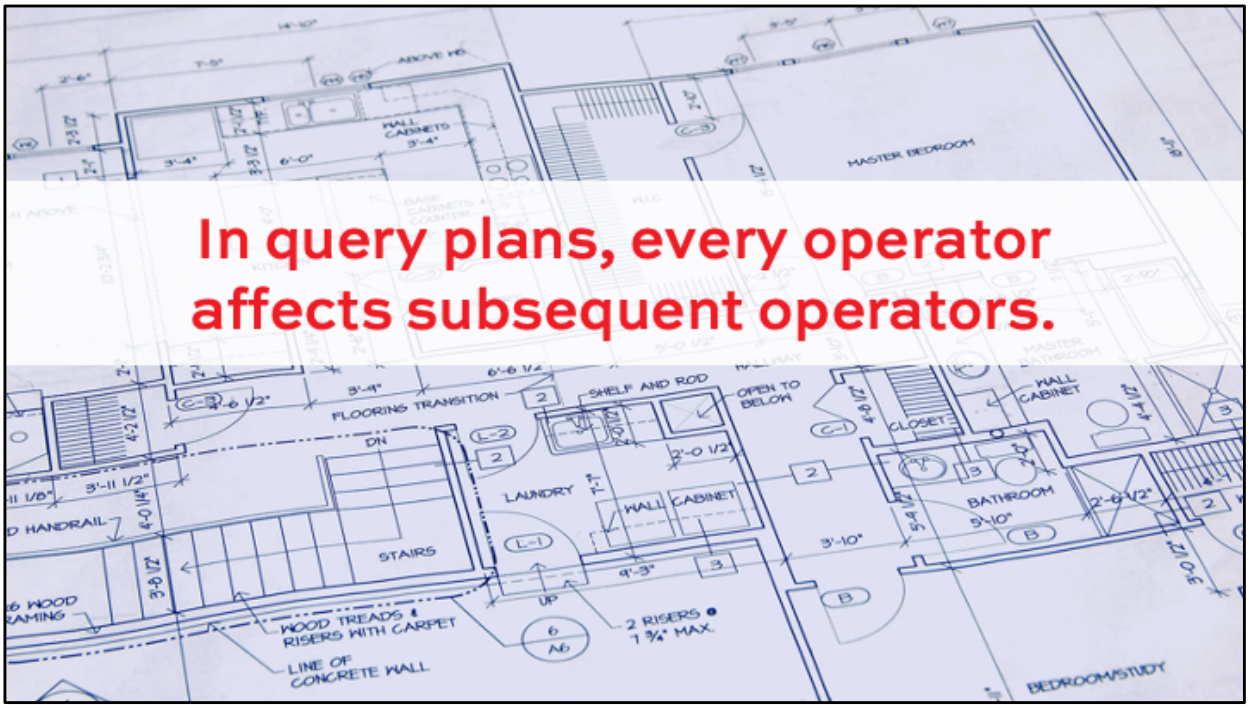




Cardinality estimation is hard.

Actual Number of Rows	110658
Estimated Number of Rows	7478

1.2 p4

Cardinality estimation is 2 things

1. How many rows will come back?
2. What will the contents of those rows be?

- Mistakes in either one can be devastating
- Even when #1 is right, #2 can be wrong

1.2 p5

When SQL Server's estimates
are reasonably close to actuals,
you're getting a good plan.

It may not be a fast plan,
but it'll accurately reflect the amount of work.
(You could reduce work by
changing the query or the indexes.)

1.2 p6

When SQL Server's estimates
are nowhere near actuals (like 10x-100x-1,000x off)
you're usually getting a bad plan.

1.2 p7

Sometimes, it's not a big deal at all.

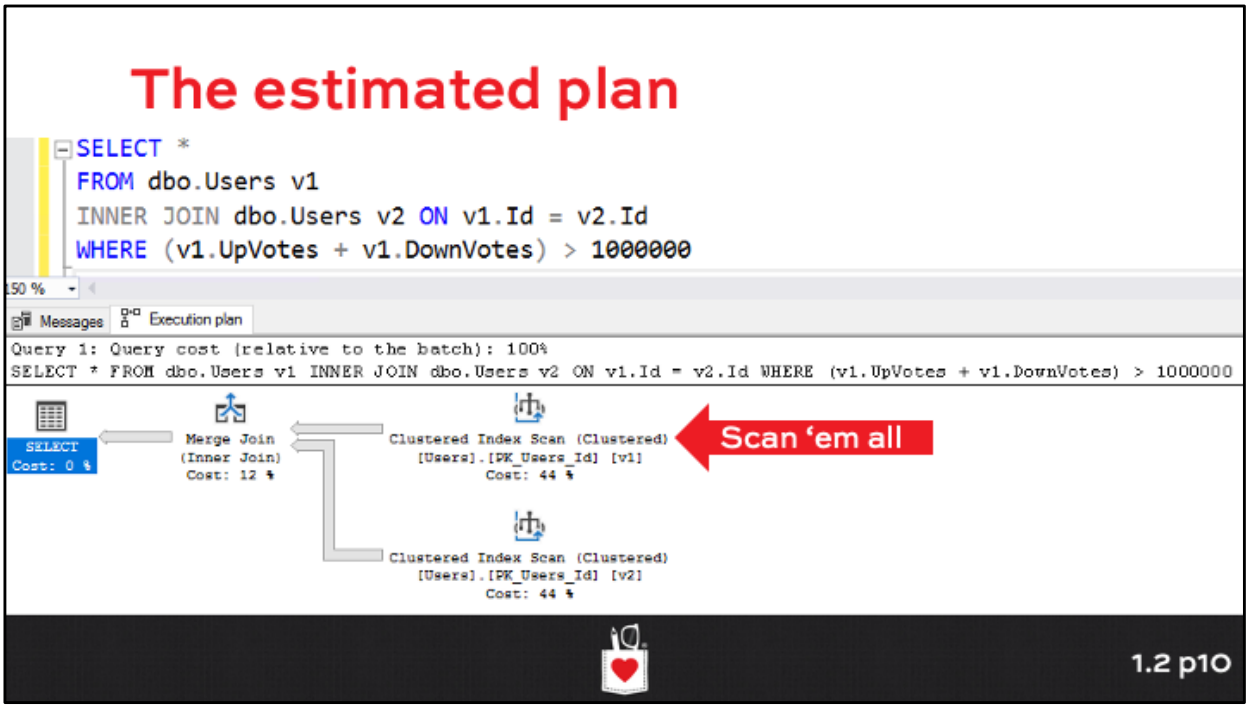
Harmless estimation errors

1.2 p8

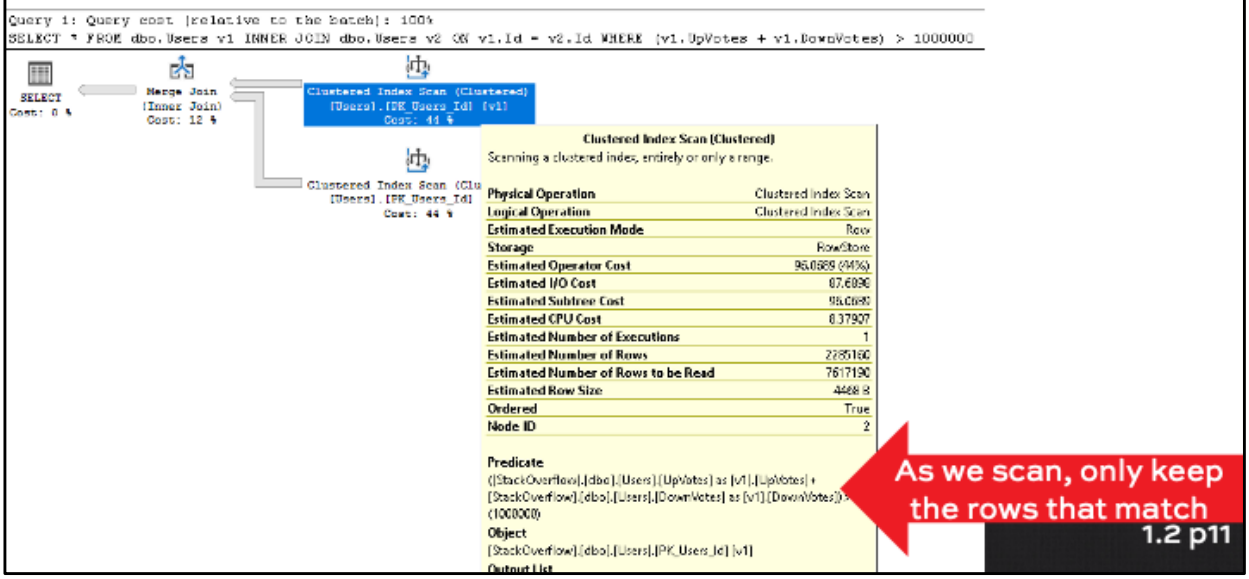
usp_UsersWithManyVotes

```
SELECT *
FROM dbo.Users v1
INNER JOIN dbo.Users v2 ON v1.Id = v2.Id
WHERE (v1.UpVotes + v1.DownVotes) > 1000000
```

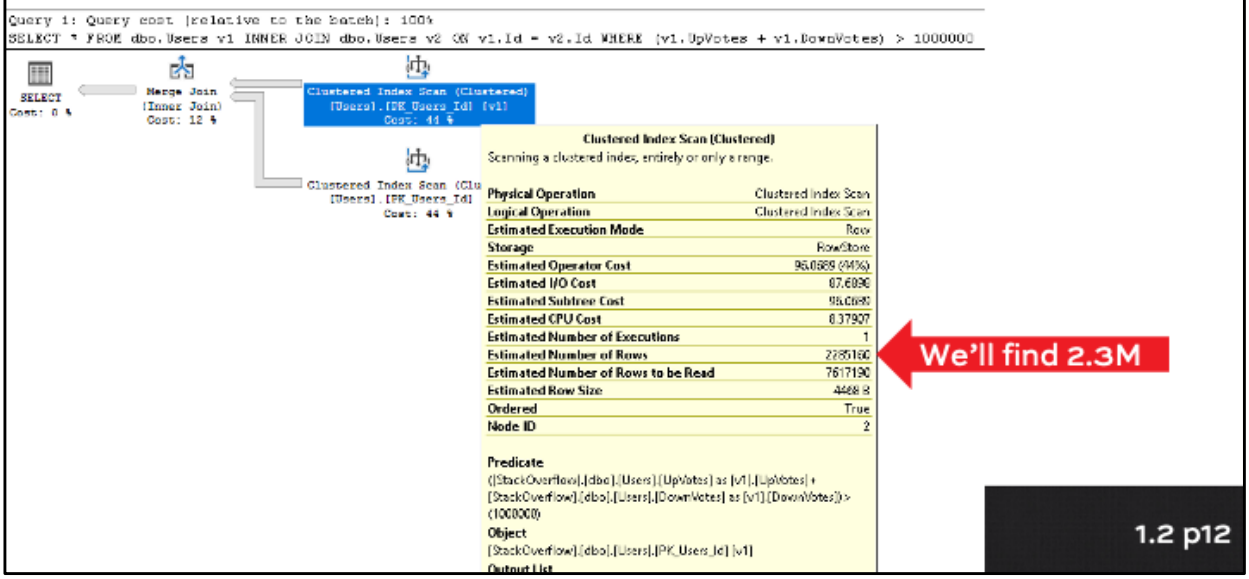
The WHERE clause is on a couple of fields.
SQL Server doesn't have stats on that combination.



This estimate is...large.



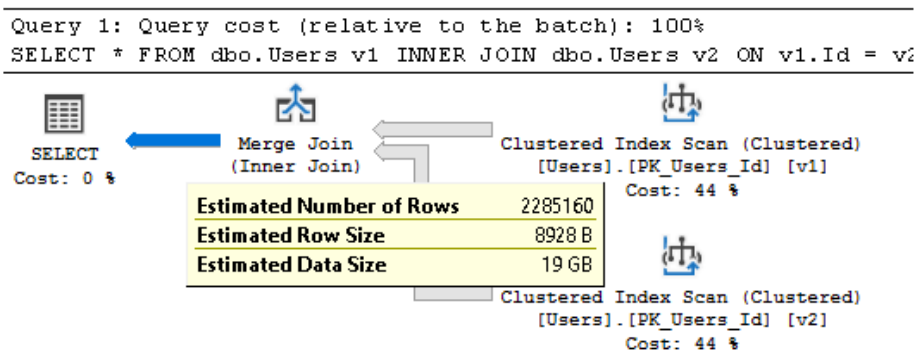
This estimate is...large.



We'll find 2.3M

1.2 p12

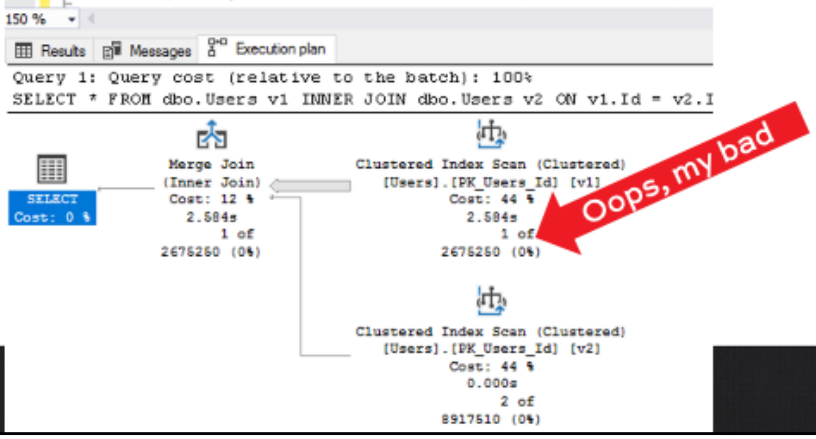
The join would produce 2.3M rows



1.2 p13

The actual plan looks different

```
SELECT *
FROM dbo.Users v1
INNER JOIN dbo.Users v2 ON v1.Id = v2.Id
WHERE (v1.UpVotes + v1.DownVotes) > 1000000;
```



1.2 p14

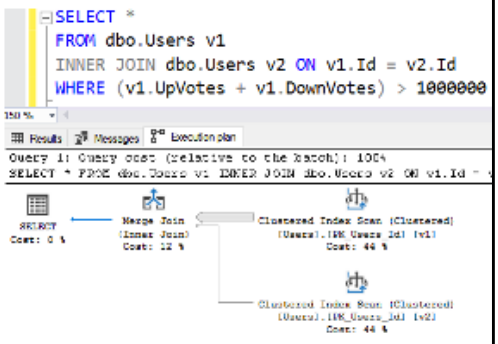
The estimate doesn't matter

Scanning the 1GB table goes really fast.

There's no big memory grant.

If you wanted to, you could tune the query or add an index.

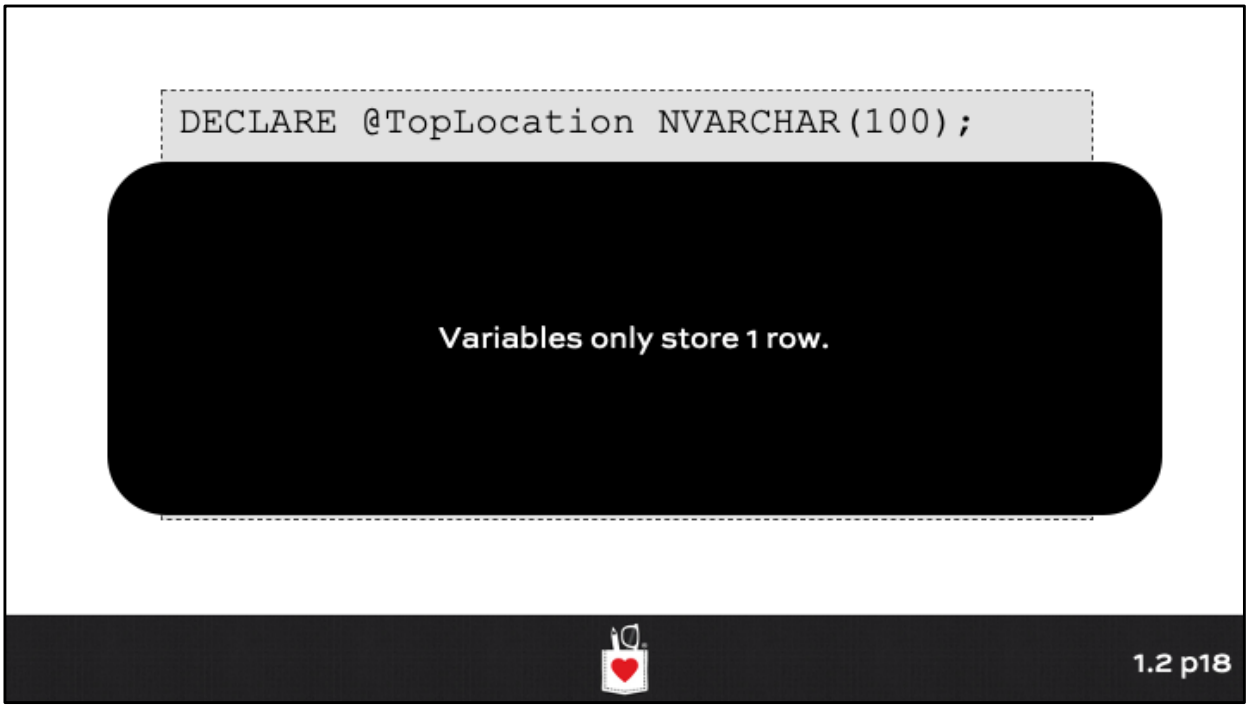
But nobody's going to notice if it isn't one of your heaviest hitters anyway.



1.2 p15

A slightly bigger firecracker

```
DECLARE @TopLocation NVARCHAR(100);

SELECT TOP 1 @TopLocation = Location
  FROM dbo.Users
  GROUP BY Location
  ORDER BY COUNT(*) DESC;

SELECT * FROM dbo.Users
WHERE Location = @TopLocation;
```



```
DECLARE @TopLocation NVARCHAR(100);

SELECT TOP 1 @TopLocation = Location
  FROM dbo.Users
  GROUP BY Location
  ORDER BY COUNT(*) DESC;
```

We're only going to put 1 value in @TopLocation.
If we thought really hard, we could possibly even
use statistics to predict what that value might be.

1.2 p19

```
DECLARE @TopLocation NVARCHAR(100);
```

But now the SELECT runs.
How can it predict how many rows will return?

```
SELECT * FROM dbo.Users
WHERE Location = @TopLocation;
```

This whole plan is built at once.

```
DECLARE @TopLocation NVARCHAR(100);

SELECT TOP 1 @TopLocation = Location
  FROM dbo.Users
  GROUP BY Location
  ORDER BY COUNT(*) DESC;

SELECT * FROM dbo.Users
WHERE Location = @TopLocation;
```

1.2 p21

usp_UsersInTopLocation

```
CREATE OR ALTER PROC dbo.usp_UsersInTopLocation AS
BEGIN
DECLARE @TopLocation NVARCHAR(100);

SELECT TOP 1 @TopLocation = Location
  FROM dbo.Users
  WHERE Location <> ''
  GROUP BY Location
  ORDER BY COUNT(*) DESC;

SELECT *
  FROM dbo.Users
  WHERE Location = @TopLocation
  ORDER BY DisplayName;
END
```

I added just a little complexity for tuning.

Avoids NULLs, empty strings

Requires a memory grant

1.2 p22

SQL Server compiles batches.

It has to build an execution plan for the entire batch all at the same time.

A stored procedure is a big batch.

You can put OPTION (RECOMPILE) on statements, forcing SQL Server to build a new execution plan given what it knows so far.

1.2 p23

Possible fix:
Combine queries

1.2 p24

I got rid of the variable.

Former query #1

Former query #2

```
CREATE OR ALTER PROC dbo.usp_UsersInTopLocation_CTE AS
BEGIN
WITH TopLocation AS (SELECT TOP 1 Location
  FROM dbo.Users
  WHERE Location <> ''
  GROUP BY Location
  ORDER BY COUNT(*) DESC)
SELECT u.*
  FROM TopLocation
   INNER JOIN dbo.Users u ON TopLocation.Location = u.Location
  ORDER BY DisplayName;
END
GO
```

1.2 p25

But the plan is built all at once.

SQL Server knows how many rows the CTE will produce: just 1, SELECT TOP 1.

But it has no idea what the location will be.

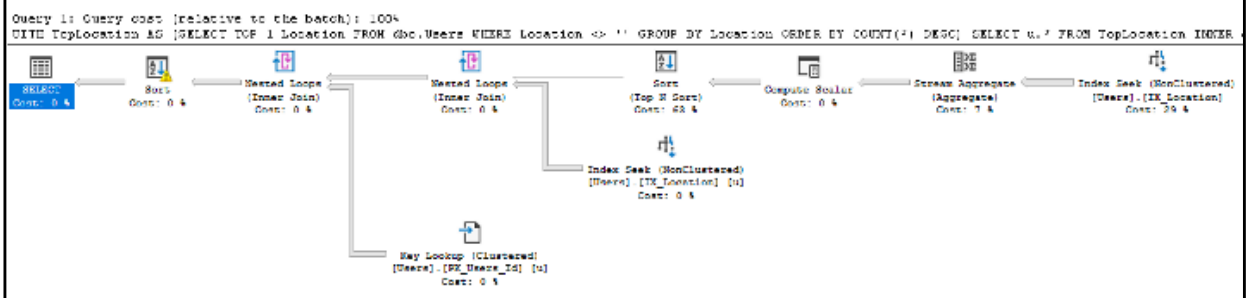
It doesn't execute the CTE first, get the location, and then execute the query.

This whole thing is done at once.

```
CREATE OR ALTER PROC dbo.usp_Users
BEGIN
WITH TopLocation AS (SELECT TOP 1
  FROM dbo.Users
  WHERE Location <> ''
  GROUP BY Location
  ORDER BY COUNT(*) DESC)
SELECT u.*
  FROM TopLocation
    INNER JOIN dbo.Users u ON TopL
  ORDER BY DisplayName;
END
GO
```

1.2 p26

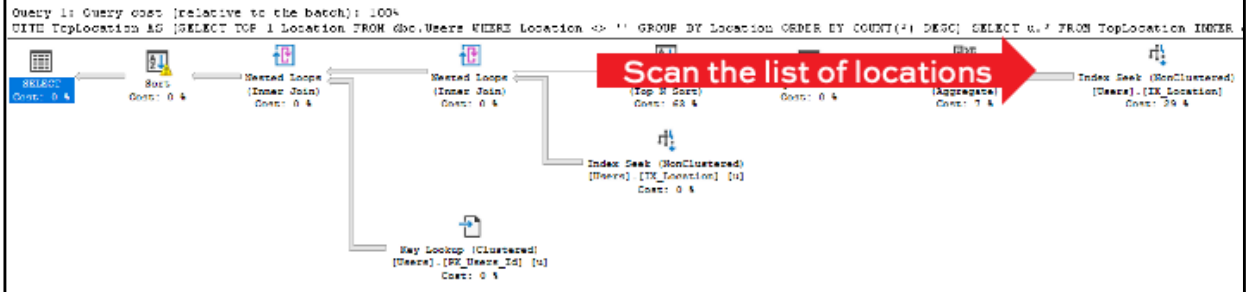
Read the plan right to left.



SQL Server's doing the same thing it did before in the 2-query process: first, it builds the list of most popular locations, takes the top 1, and then looks up the users in that location.

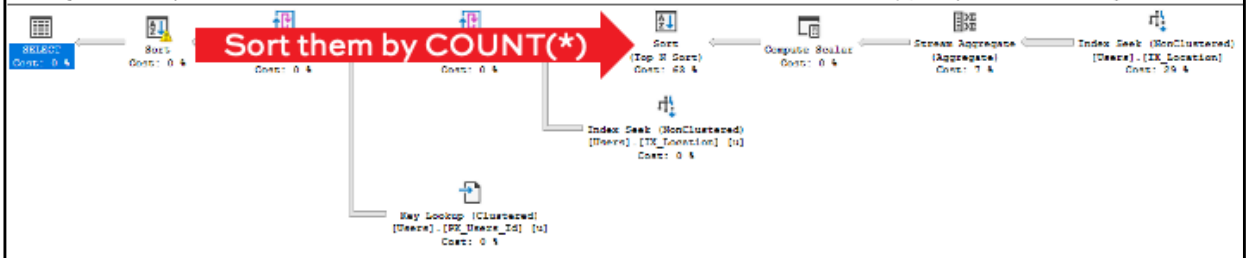
1.2 p27

Is this estimate going to be right?



Do we know how many locations we'll find?

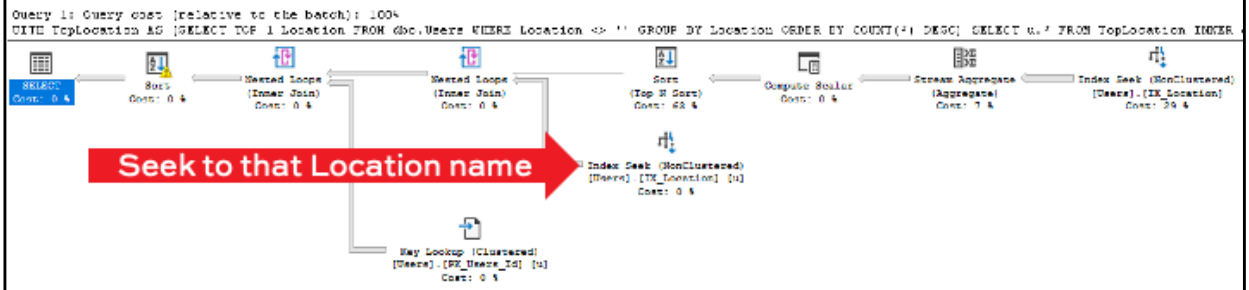
Is this estimate going to be right?



Do we know how many locations the Sort will push out?

1.2 p29

This one is a little trickier.



We're going to seek to that Location name. But 2 questions:

1. How many times are we going to seek to a Location?
2. How many rows are going to come out of the seek?

1.2 p30

From “Think Like the Engine”

A “seek” sounds like it’s only going to return 1 row.

A “scan” sounds like it returns the whole table.

But here’s what they really mean:

Seek = start reading at one specific location

Scan = start reading at either end of the object

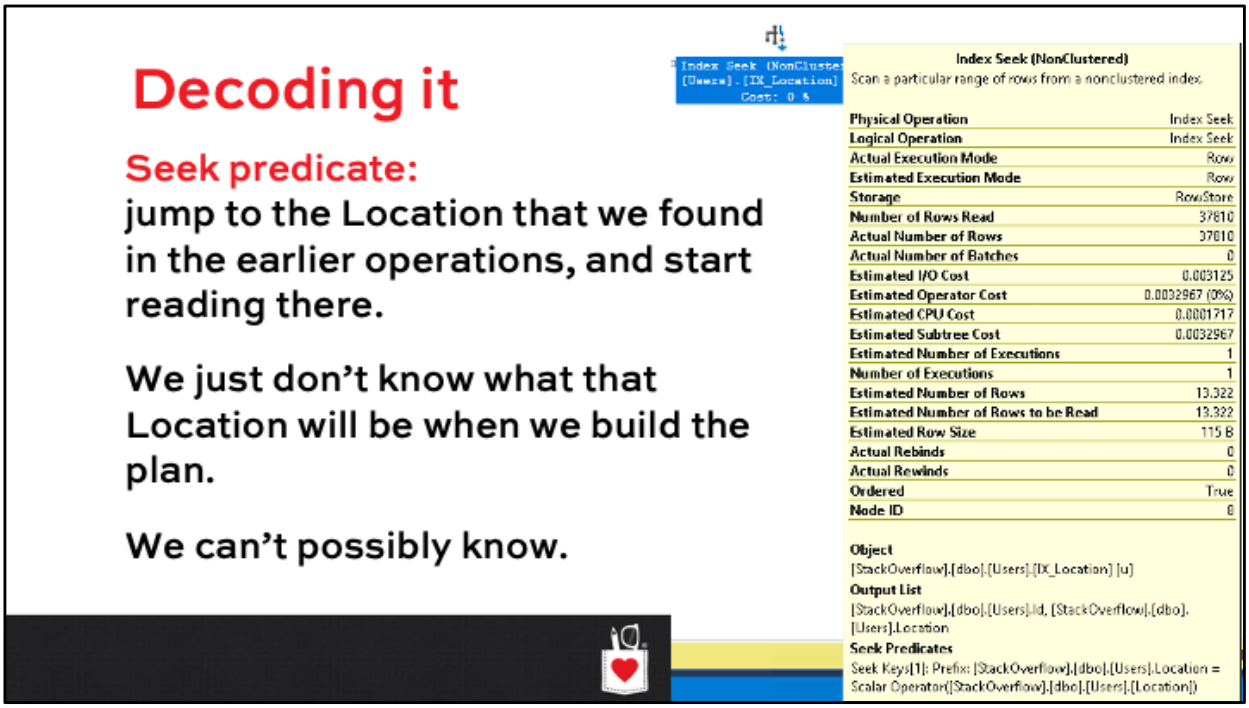
1.2 p31

Decoding it

Seek predicate:
jump to the Location that we found in the earlier operations, and start reading there.

We just don't know what that Location will be when we build the plan.

We can't possibly know.



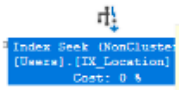
Estimated rows

The estimate is just garbage.

SQL Server's using the density vector (more on that in Think Like the Engine and Mastering Index Tuning.)

It's guessing based on how many rows the average Location has.

But we're asking for the biggest one.



Index Seek (NonClustered)

Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	37810
Actual Number of Rows	37810
Actual Number of Batches	0
Estimated I/O Cost	0.003125
Estimated Operator Cost	0.0032967 (0%)
Estimated CPU Cost	0.0001717
Estimated Subtree Cost	0.0032967
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows	13.322
Estimated Number of Rows to be Read	13.322
Estimated Row Size	115 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	8

Object
[StackOverflow].[dbo].[Users].[IX_Location] [u]

Output List
[StackOverflow].[dbo].[Users].Id, [StackOverflow].[dbo].[Users].Location

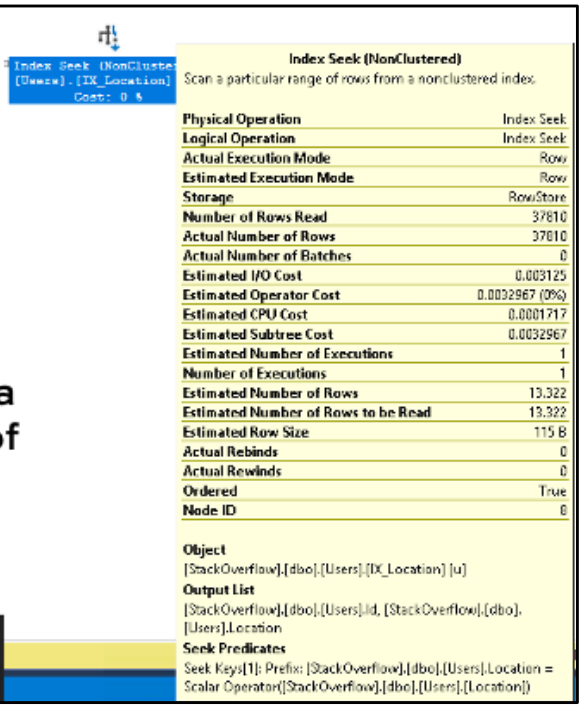
Seek Predicates
Seek Keys[1]: Prefix: [StackOverflow].[dbo].[Users].Location = Scalar Operator([StackOverflow].[dbo].[Users].[Location])

Actual rows

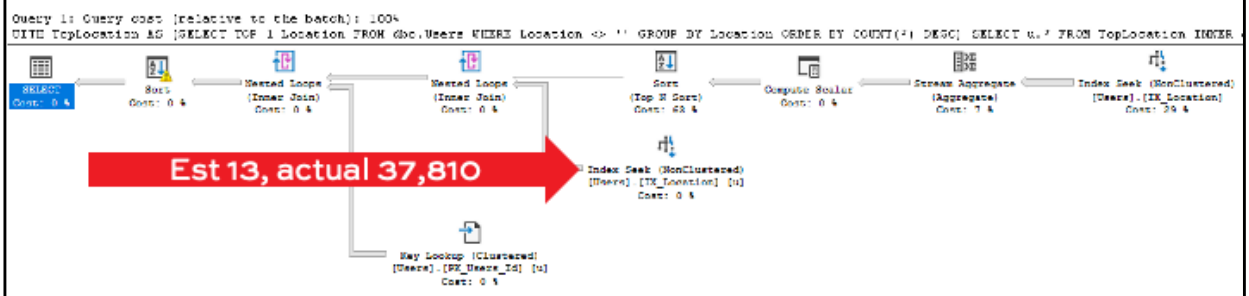
Estimates: 13

Actual: 37,810

These are way off, and it has a cascading effect on the rest of the plan.



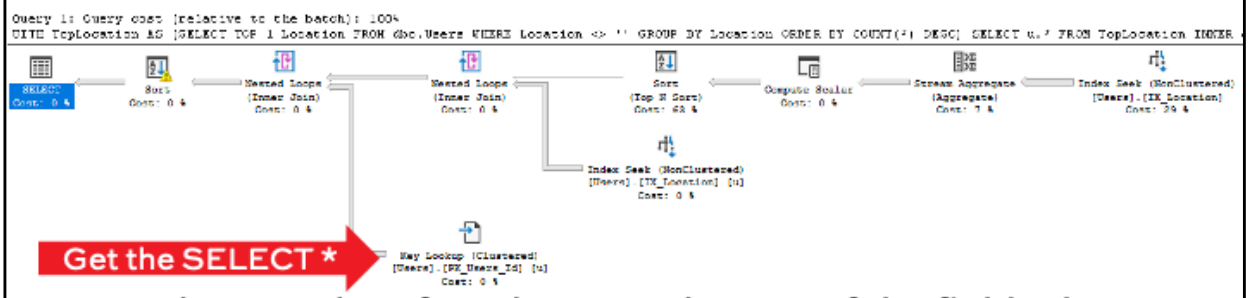
This one is a little trickier.



So now, as we move through the rest of the plan, we're going to have issues.

1.2 p35

This one is a little trickier.

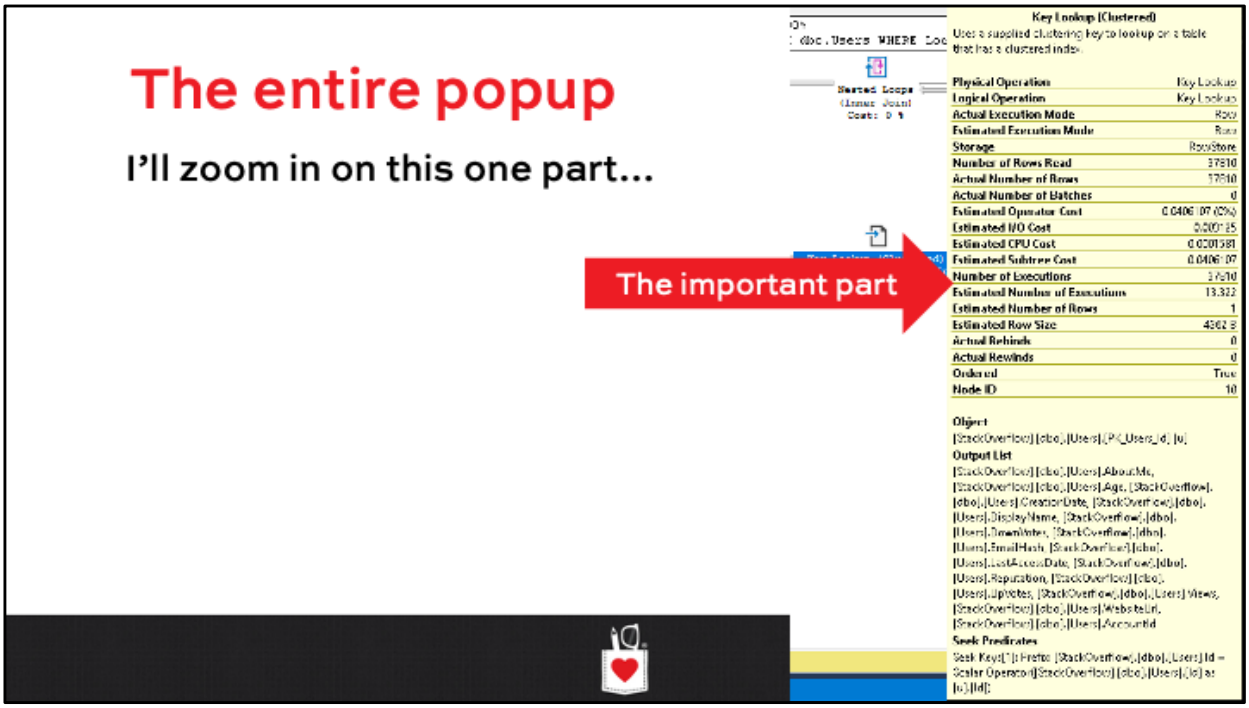


Get the SELECT *

For each User.Id we found, go get the rest of the fields that weren't included in our Location index. Again, 2 questions:

1. How many times are we going to do this key lookup?
2. How many rows are going to come out of each one?

1.2 p36



Dammit, Beavis

Actual Number of Rows	37810
Actual Number of Batches	0
Estimated Operator Cost	0.0406107 (0%)
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001581
Estimated Subtree Cost	0.0406107
Number of Executions	37810
Estimated Number of Executions	13.322
Estimated Number of Rows	1
Estimated Row Size	4262 B

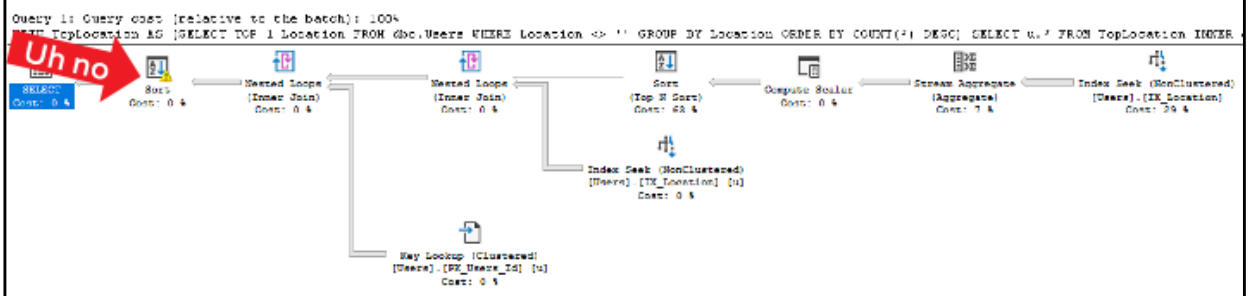
FOR THE LOVE OF ALL THAT'S HOLY CAN YOU PLEASE PUT THE FIELDS IN SOME KIND OF ORDER AND BE CONSISTENT ABOUT WHETHER YOU PREFIX THINGS WITH ACTUAL OR NOT

1.2 p38

Decoding it

The label	The value	What it means
Estimated Number of Executions	13	
Number of Executions	37,810	Which is bad because each one produces a few logical reads, so we read more pages than the entire table
Estimated Number of Rows	1	The number of rows that will come out of EACH OPERATION, which is really misleading
Actual Number of Rows	37,810	The total number of rows that came out of ALL THE OPERATIONS

Is this estimate going to be right?



The Sort needed to estimate how many rows it'd be dealing with, which affects our overall memory grant.

1.2 p40

This CTE is a great example.

Even in a small query like this:

- SQL Server processes data in order, in steps
- Can kinda be thought of as a stored procedure
- Estimates can go wrong in any step

Our job:

- Read the plan right to left, top to bottom
- Understand where estimates are going wrong
- Help SQL Server make better estimations

Possible fix:
Subquery

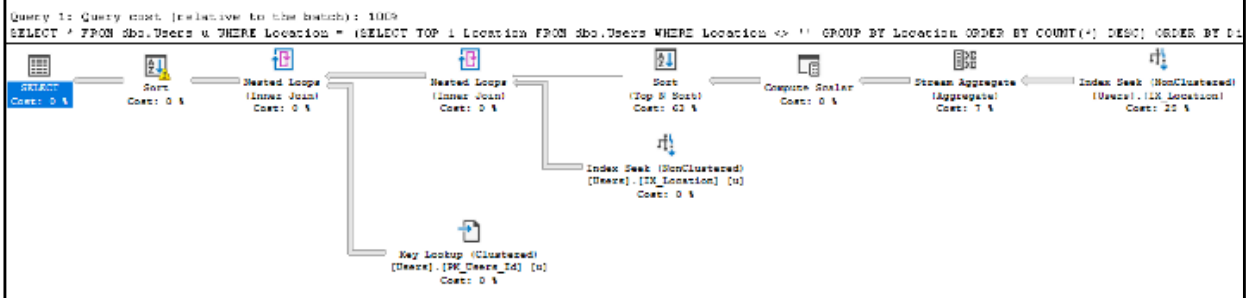
Same basic idea as the CTE

```
CREATE OR ALTER PROC dbo.usp_UsersInTopLocation_Subquery AS
BEGIN
SELECT *
  FROM dbo.Users u
  WHERE Location = (SELECT TOP 1 Location
                    FROM dbo.Users
                    WHERE Location <> ''
                    GROUP BY Location
                    ORDER BY COUNT(*) DESC)
  ORDER BY DisplayName;
END
GO
```

1.2 p43

usp_UsersInTopLocation_Subquery

- Read the execution plan right to left, top to bottom
- At each step, guess whether the estimates are OK



1.2 p44

Same problems as the CTE.

If we force SQL Server to build the entire plan at once, there's simply no way it can get the estimates right.

When you have a query like this, consider:

- Breaking it up into separate parts
- Find the point where cardinality estimation fails, and right up to that, that's what you need to break into a separate query

1.2 p45

Possible fix:
OPTION RECOMPILE

1.2 p46

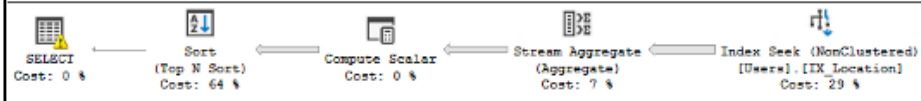
```sql
CREATE OR ALTER PROC dbo.usp_UsersInTopLocation AS
BEGIN
DECLARE @TopLocation NVARCHAR(100);

SELECT TOP 1 @TopLocation = Location
  FROM dbo.Users
  WHERE Location <> ''
  GROUP BY Location
  ORDER BY COUNT(*) DESC;

SELECT *
  FROM dbo.Users
  WHERE Location = @TopLocation
  ORDER BY DisplayName OPTION (RECOMPILE);
END
```

New plan for this one statement

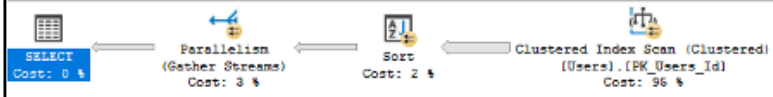
1.2 p47

A better plan? A better plan.

```
Query 1: Query cost (relative to the batch): 100%
SELECT TOP 1 @TopLocation = Location FROM dbo.Users WHERE Location <> '' GROUP BY Location ORDER BY COUNT(*) DESC
```




```
Query 2: Query cost (relative to the batch): 88%
SELECT * FROM dbo.Users WHERE Location = @TopLocation ORDER BY DisplayName OPTION (RECOMPILE)
Missing Index (Impact 95.9373): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users] ([Location])
```



At first, this might worse: a clustered index scan of the entire Users table, ignoring the Locations index. But it actually does less logical reads because we're dealing with a lot of Users. And note – no spill on the sort.

1.2 p48

It's asking for a missing index.

```
,>] ON [dbo].[Users] ([Location]) INCLUDE ([AboutMe],[Age],[CreationDate],…
```

But it includes EVERY SINGLE FIELD.

Over here in reality, we can't usually create indexes like that. (If you can, great, do it – but notice AboutMe and its horrible datatype.)

What OPTION (RECOMPILE) does

Forces SQL Server to stop and build a new plan

Takes effect at the level where you put it

Here, we're recompiling a single statement because what comes out of the prior query changes EVERYTHING we do:

- The index we use, and the way we use it
- How much memory we need

Possible fix:
OPTION RECOMPILE
at the proc level

1.2 p51

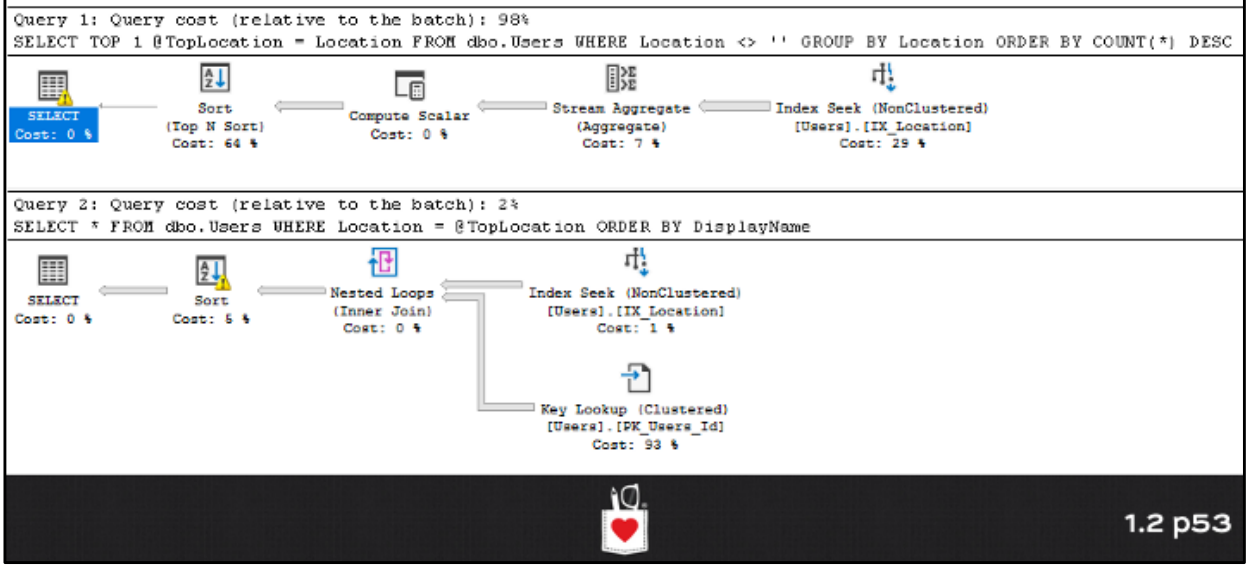
```
CREATE OR ALTER PROC dbo.usp_UsersInTopLocation WITH RECOMPILE AS
BEGIN
DECLARE @TopLocation NV
```

What if we put it up here?

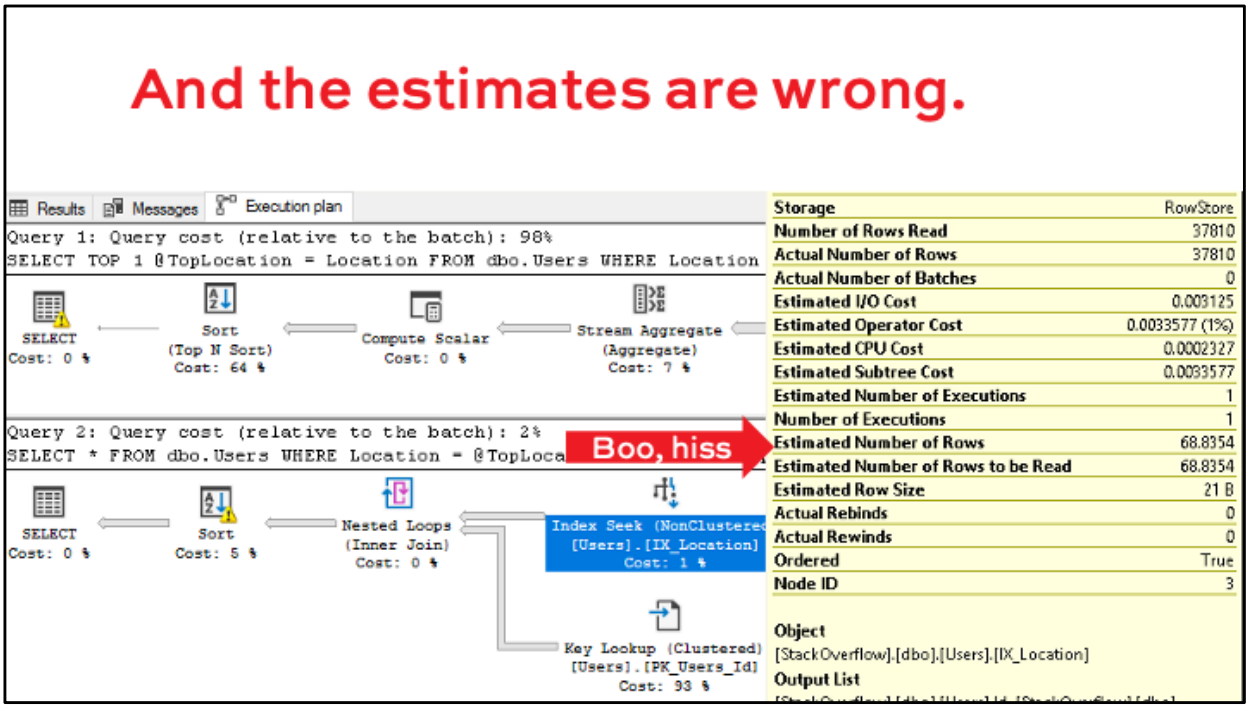
```
SELECT TOP 1 @TopLocation = Location
  FROM dbo.Users
  WHERE Location <> ''
  GROUP BY Location
  ORDER BY COUNT(*) DESC;

SELECT *
  FROM dbo.Users
  WHERE Location = @TopLocation
  ORDER BY DisplayName;
END
```

1.2 p52

We build the whole plan at once...



And the estimates are wrong.



Using recompile hints

They help when SQL Server needs to reset expectations about what it's about to do

But ***you*** have to know where to put them

Typically best used when:

- You're doing multi-step processing
- The amount of data varies WIDELY
- You can identify the point where things change dramatically, and stick the hint there

1.2 p55

Whew.

1.2 p56

What we learned

Cardinality estimation involves:

- Predicting how many rows will come back
- Guessing the contents of those rows to predict how many rows will come back from *other* operations

Lots of ways to accomplish it with varying success:

- Updating the existing statistics
- OPTION (RECOMPILE)
- Switching Cardinality Estimators (compat levels)
- Changing the database (indexes, stats)
- Rewriting the query

1.2 p57

Setting up for the lab

1. Restart your SQL Server service (clears all stats)
2. Restore your StackOverflow database (Agent job)
3. Copy & run the setup script for Lab 1
4. (No SQLQueryStress for this lab)

1.2 p58