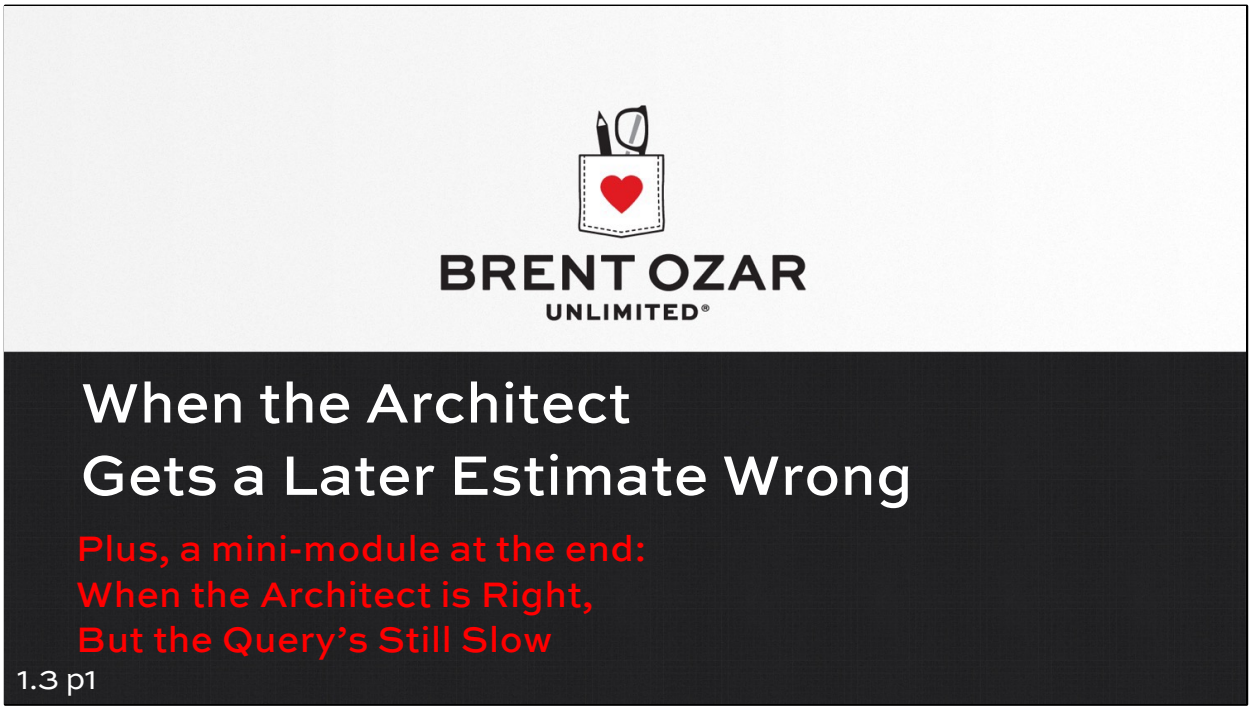


2 kinds of estimates

Early:

- Your query has a filter (typically WHERE, but can be elsewhere)
- SQL Server uses statistics to guess how many rows will match the filter

Late:

- Your query joins to other tables
- There's no direct filter on those tables
- What we're looking for is based off the early filters on other tables

1.3 p2

```
CREATE OR ALTER PROC dbo.usp_SearchPostsByLocation
    @Location VARCHAR(100) AS

    /* Find the most recent posts from an area */
    SELECT TOP 200 u.DisplayName, p.Title, p.Id, p.CreationDate
      FROM dbo.Users u
      INNER JOIN dbo.Posts p ON u.Id = p.OwnerUserId
      WHERE u.Location = @Location
      ORDER BY p.CreationDate DESC;
GO
```

1.3 p3

```
CREATE OR ALTER PROC dbo.usp_SearchPostsByLocation
    @Location VARCHAR(100) AS

    /* Find the most recent posts from an area */
    SELECT TOP 200 u.DisplayName, p.Title, p.Id, p.CreationDate
      FROM dbo.Users u
      INNER JO      dbo.Posts p ON u.Id = p.OwnerUserId
      WHERE u.Location = @Location
      ORDER BY p.CreationDate DESC;
GO
```

Later

1.3 p4

Late filter on Posts

```
FROM dbo.Users u
INNER JOIN dbo.Posts p ON u.Id = p.OwnerUserId
WHERE u.Location = @Location
```

Late

To predict how many Posts we'll find, we have to find the right Users first.

That doesn't mean we have to execute Users first: if we think our query will find a lot of Users, SQL Server might decide to scan the whole Posts table first.

Generally speaking

Early estimates are:

- Driven by the WHERE clause
- Based on statistics on tables
- Executed first in the query plan

Late estimates are:

- A multiplier of the early estimates (more users = more comments)
- Based on density vectors, averages
- If early estimates are wrong, these are screwed

1.3 p6

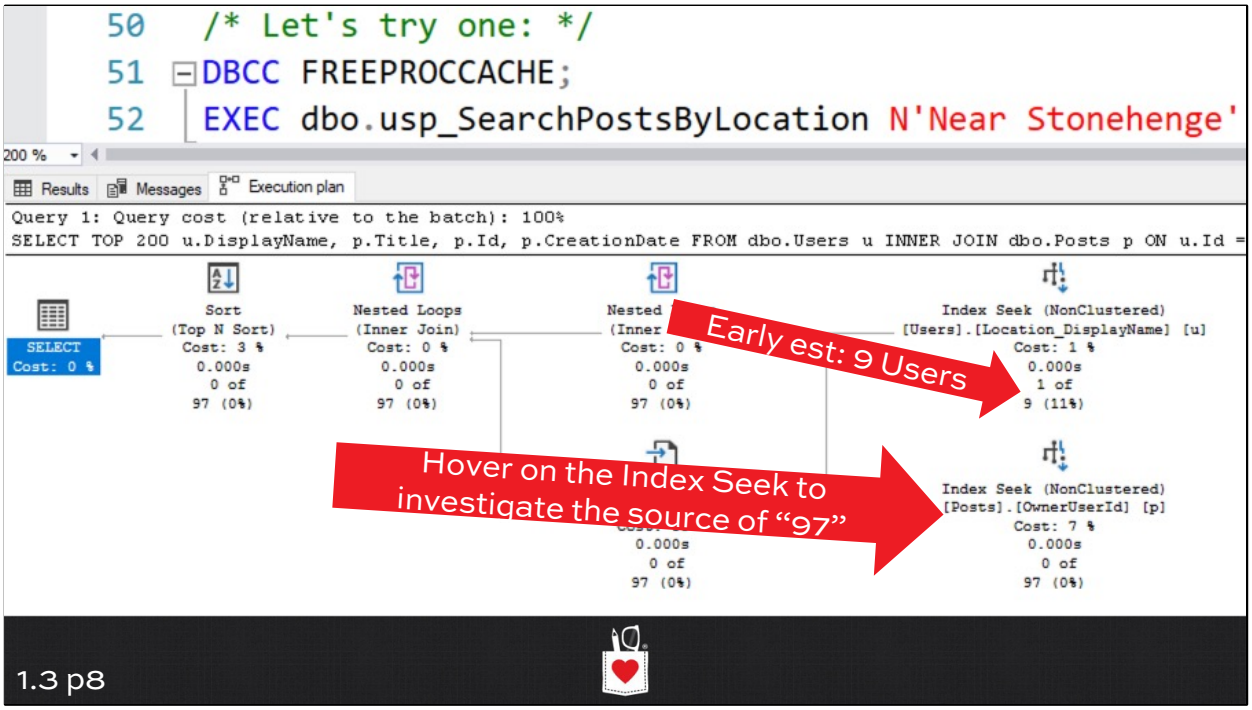
Architect has to do it all at once

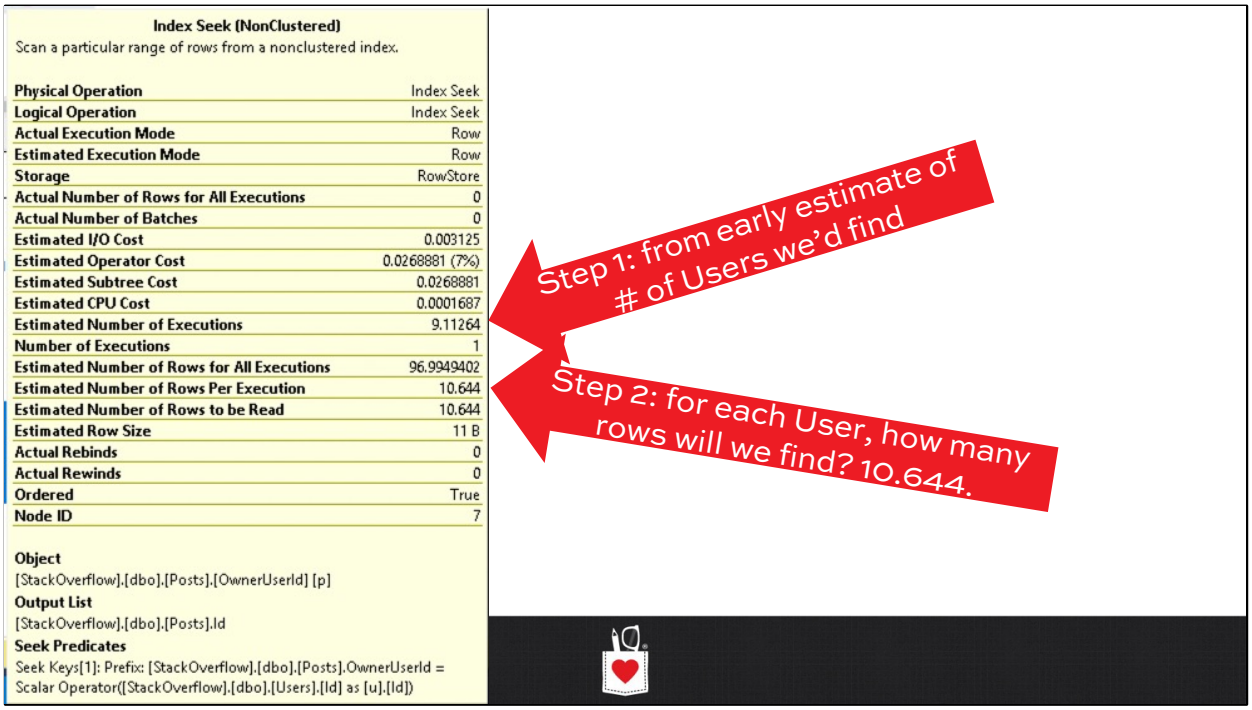
Two separate phases:

- **Architect: design** a query plan
- **Builder: execute** the query plan

And for any one statement (SELECT),

- Plan design must finish before execution starts
- While a query is being executed, that statement's plan can't be redesigned in flight
- Only subsequent statements can be redesigned AFTER our statement finishes executing





1 User = 10.644 Posts.

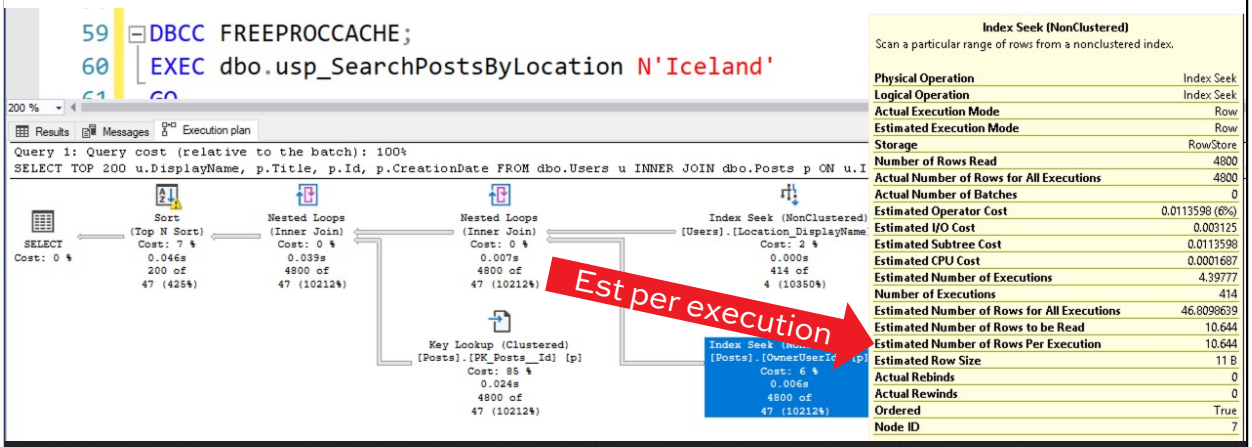
That's not specific to individual users: on average, SQL Server is guessing 10.644 Posts.

SQL Server doesn't know which users live in which Locations.

It's just making one guess for any location.

To prove it, let's try another location in another stats bucket.

Iceland? Same 10,644 rows.



1.3 p11

Where's the 10.644 come from?

You might remember Density Vector:
the average number of rows for any given value.

```
69  DBCC SHOW_STATISTICS('dbo.Posts', 'OwnerUserId');
```

200 %

Results | Messages

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows	Persisted Sample Percent
1	OwnerUserId	Jul 16 2018 3:19PM	40700647	40700647	161	0.09686216	8	NO	NULL	40700647	0

	All density	Average Length	Columns
1	2.615199E-07		OwnerUserId
2	2.456963E-08		wnerUserId, Id

This

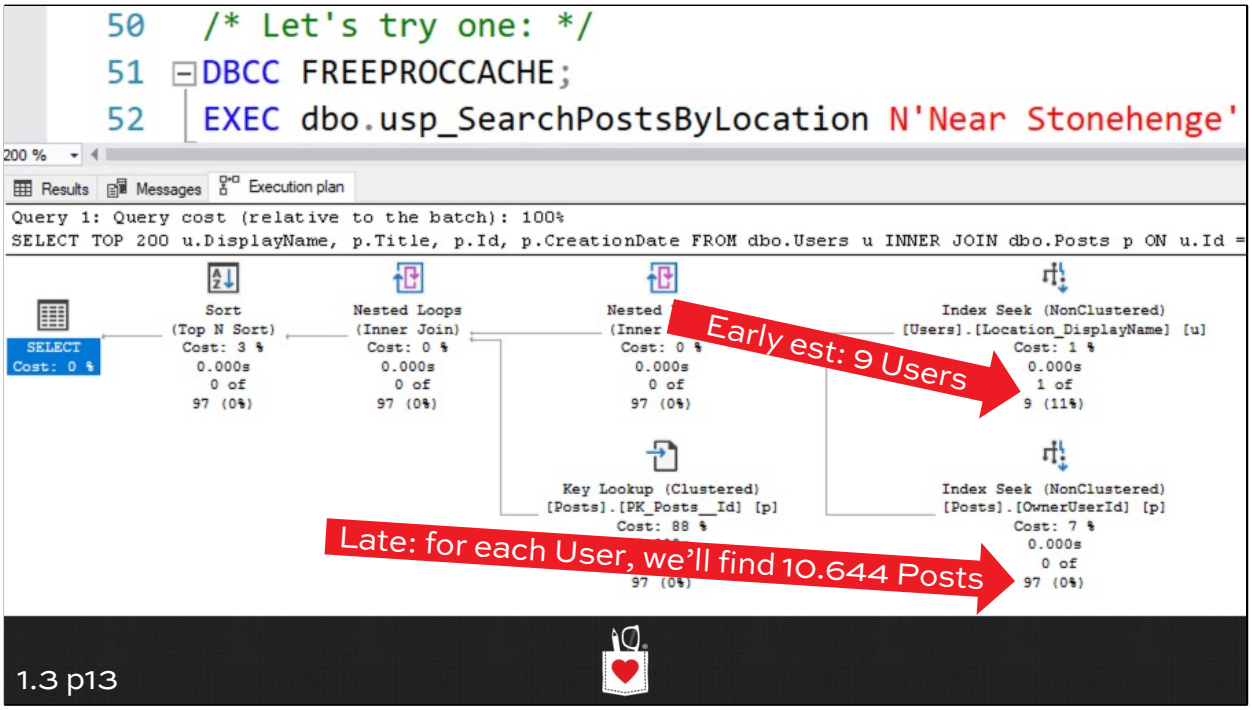
For any given OwnerUserId

```
70  SELECT 2.615199E-07 * 40700647;
```

200 %

Results | Messages

	(No column name)
1	10.6440291333753



Late estimates are just guesses.

There is a stats histogram on Posts by OwnerUserId, but that doesn't do us any good.

Until we actually execute the query, we just don't know which specific UserIds live Near Stonehenge.

So SQL Server uses averages.

How late estimates go wrong

SQL Server makes a bad early estimate, OR

The early estimate is accurate, but Posts has outliers:

- Not a lot of people live Near Stonehenge, but they write a LOT of Posts
- Or, a whole lot of people live Near Stonehenge, but they hardly write any Posts at all

Late estimation error sources

In order of how often I see 'em:

1. The early estimation is wrong
   (always work right to left, top to bottom)
2. There are outliers in the later tables, like:
   - Few people live Near Stonehenge,
     but they write a LOT of Posts
   - Or, many people live Near Stonehenge,
     but they hardly write any Posts at all

1.3 p16

Outlier example

Query Locations with few people, but many posts:

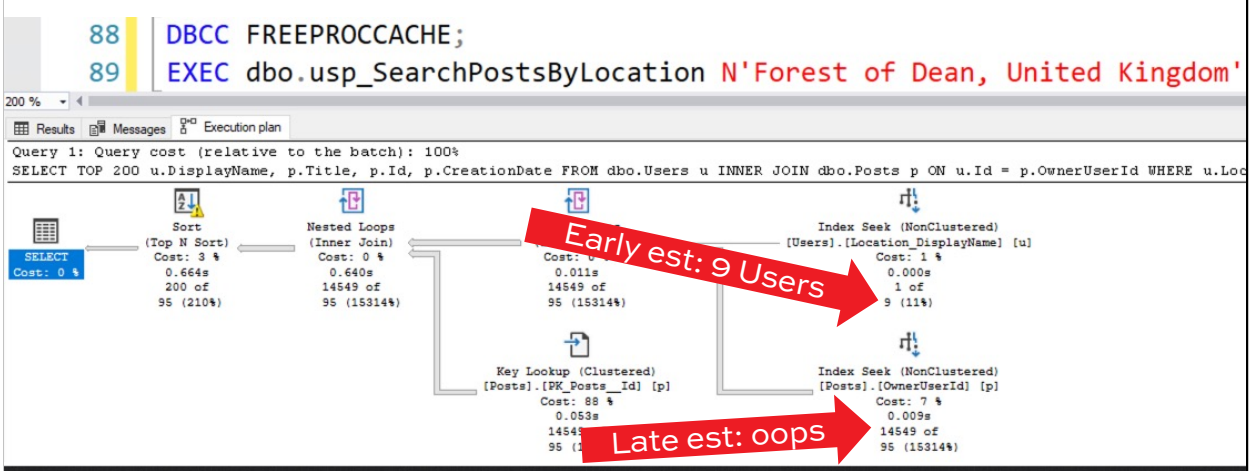
```
76  SELECT TOP 100 u.Location,
77      (SUM(1) / COUNT(DISTINCT u.Id)) AS PostsPerUser,
78      COUNT(DISTINCT u.Id) AS Residents, SUM(1) AS TotalPosts
79  FROM dbo.Users u
80  INNER JOIN dbo.Posts p ON u.Id = p.OwnerUserId
81  GROUP BY u.Location
82  ORDER BY (SUM(1) / COUNT(DISTINCT u.Id)) DESC;
```

200 %

Results | Messages | Execution plan

	Location	PostsPerUser	Residents	TotalPosts
1	Forest of Dean, United Kingdom	14549	1	14549
2	St Kilda West	13864	1	13864
3	Västerväla, Sweden	10643	1	10643
4	on the server farm	7323	2	14646
5	Tabayesco, Lanzarote, Canary Islands	7261	1	7261

The early estimate is close, but...



And we can't fix that.

There are 2 phases to querying:

1. The architect (query optimizer) designs a query plan, then
2. The builder (query processor) executes the plan

By the time we start executing,
it's too late: we can't fix later estimates.

The problem we're up against

Early estimation error sources:

1. WHERE clause obfuscation
2. 201 bucket problem
3. Statistics with low sampling rates
4. Statistics out of date

Late estimation error sources:

1. Early estimation errors
2. Outlier values during joins

1.3 p21

We have two options

- **Either fix the root cause** of the bad estimate
  (the right way, but can be hard, time-consuming)
- **OR mitigate the effects:**
  break the query up into two parts,
  and go back to the architect (query optimizer)
  after the first part is done,
  so it knows how many rows are really involved

And very often, the second one is easier.

X-Acto Knife Technique

1.3 p23

The X-Acto Knife Technique

```
88  DBCC FREEPROCCACHE;
89  EXEC dbo.usp_SearchPostsByLocation N'Forest of Dean, United Kingdom'
```

200 %

Results | Messages | Execution plan

Query 1: Query cost (relative to the batch): 100%
SELECT TOP 200 u.DisplayName, p.Title, p.Id, p.CreationDate FROM dbo.Users u INNER JOIN dbo.Posts p ON u.Id = p.OwnerUserId WHERE u.Loc

SELECT Cost: 0 %
Sort (Top N Sort) Cost: 3 % 0.664s 200 of 95 (210%)
Nested Loops (Inner Join) Cost: 0 % 0.640s 14549 of 95 (15314%)
Nest (Inn Cos 0.011s 14549 of 95 (15314%)
Index Seek (NonClustered) [Users].[Location_DisplayName] [u] Cost: 1 % 0.000s 1 of 9 (11%)
Key Lookup (Clustered) [Posts].[PK_Posts__Id] [p] Cost: 88 % 0.053s 14549 of
Index Seek (NonClustered) [Posts].[OwnerUserId] [p] Cost: 7 % 0.009s 14549 of 95 (15314%)

Early est: OK

Late est: bad

1.3 p24

The X-Acto Knife Technique

Read the plan right to left, top to bottom

Find the place where estimates vs actual suddenly went >10X high or >10X low

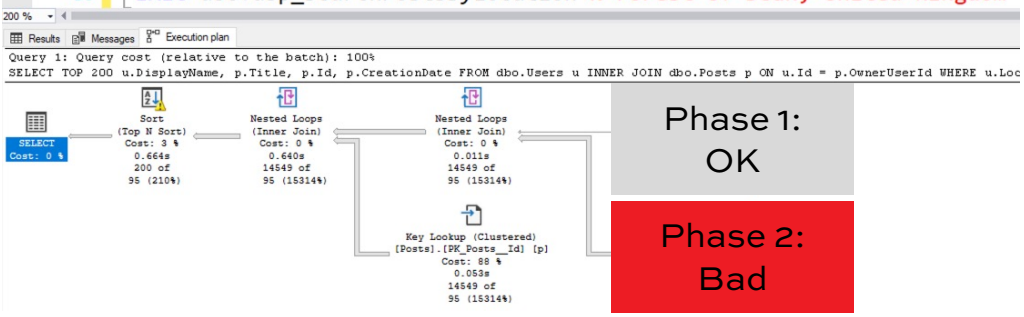
Break the query up into two parts:

- Phase 1 runs, insert into temp table
- Phase 2 runs, SQL Server builds stats on the temp table, understands how many rows came out of Phase 1

1.3 p25

The X-Acto Knife Technique

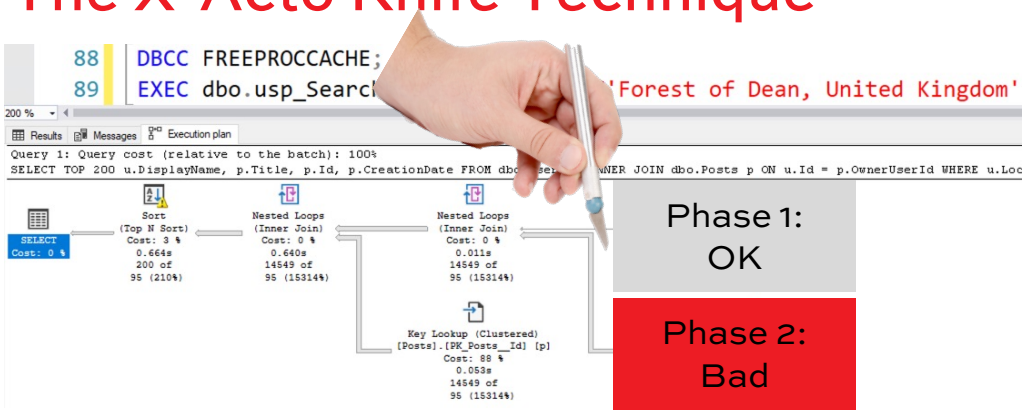
```
88  DBCC FREEPROCCACHE;
89  EXEC dbo.usp_SearchPostsByLocation N'Forest of Dean, United Kingdom'
```



Phase 1: OK

Phase 2: Bad

The X-Acto Knife Technique



Phase 1: OK

Phase 2: Bad

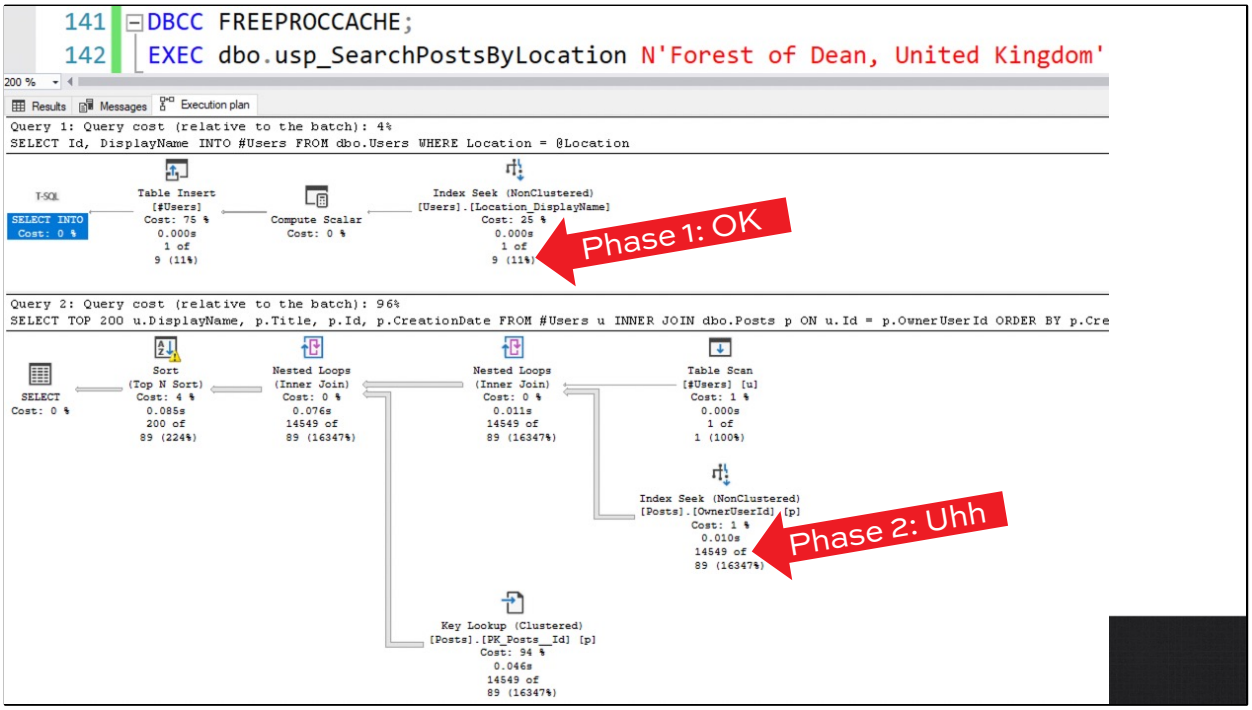
Original version

```
CREATE OR ALTER PROC dbo.usp_SearchPostsByLocation
    @Location VARCHAR(100) AS

    /* Find the most recent posts from an area */
    SELECT TOP 200 u.DisplayName, p.Title, p.Id, p.CreationDate
      FROM dbo.Users u
      INNER JOIN dbo.Posts p ON u.Id = p.OwnerUserId
      WHERE u.Location = @Location
      ORDER BY p.CreationDate DESC;
GO
```

1.3 p28

```
CREATE OR ALTER PROC dbo.usp_SearchPostsByLocation
    @Location VARCHAR(100) AS
BEGIN
    SELECT Id, DisplayName
        INTO #Users
        FROM dbo.Users
        WHERE Location = @Location;

    SELECT TOP 200 u.DisplayName, p.Title, p.Id, p.CreationDate
      FROM #Users u
      INNER JOIN dbo.Posts p ON u.Id = p.OwnerUserId
      ORDER BY p.CreationDate;
END
```

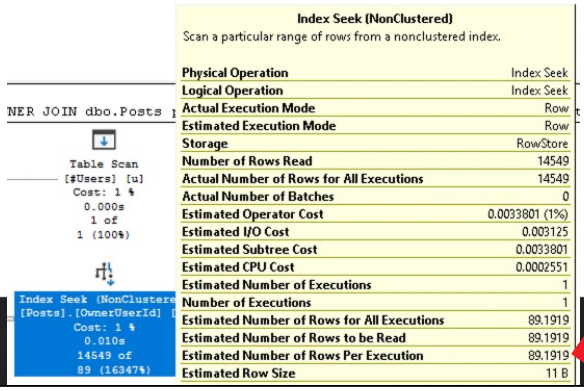
1.3 p29



“Better”, but, uh, not by much

When SQL Server was just estimating an unknown average UserId, it guessed 10.644.

Now, it’s estimating 89.1919



Index Seek (NonClustered)

Scan a particular range of rows from a nonclustered index.

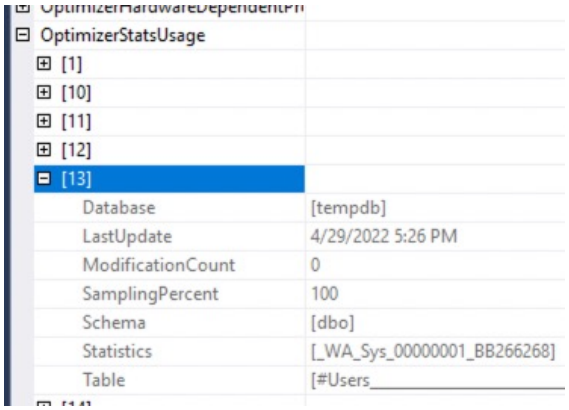
Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	14549
Actual Number of Rows for All Executions	14549
Actual Number of Batches	0
Estimated Operator Cost	0.0033801 (1%)
Estimated I/O Cost	0.003125
Estimated Subtree Cost	0.0033801
Estimated CPU Cost	0.0002551
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows for All Executions	89.1919
Estimated Number of Rows to be Read	89.1919
Estimated Number of Rows Per Execution	89.1919
Estimated Row Size	11 B

Different, but…

1.3 p31

Why 89.1919?

SQL Server did build statistics on the temp table:

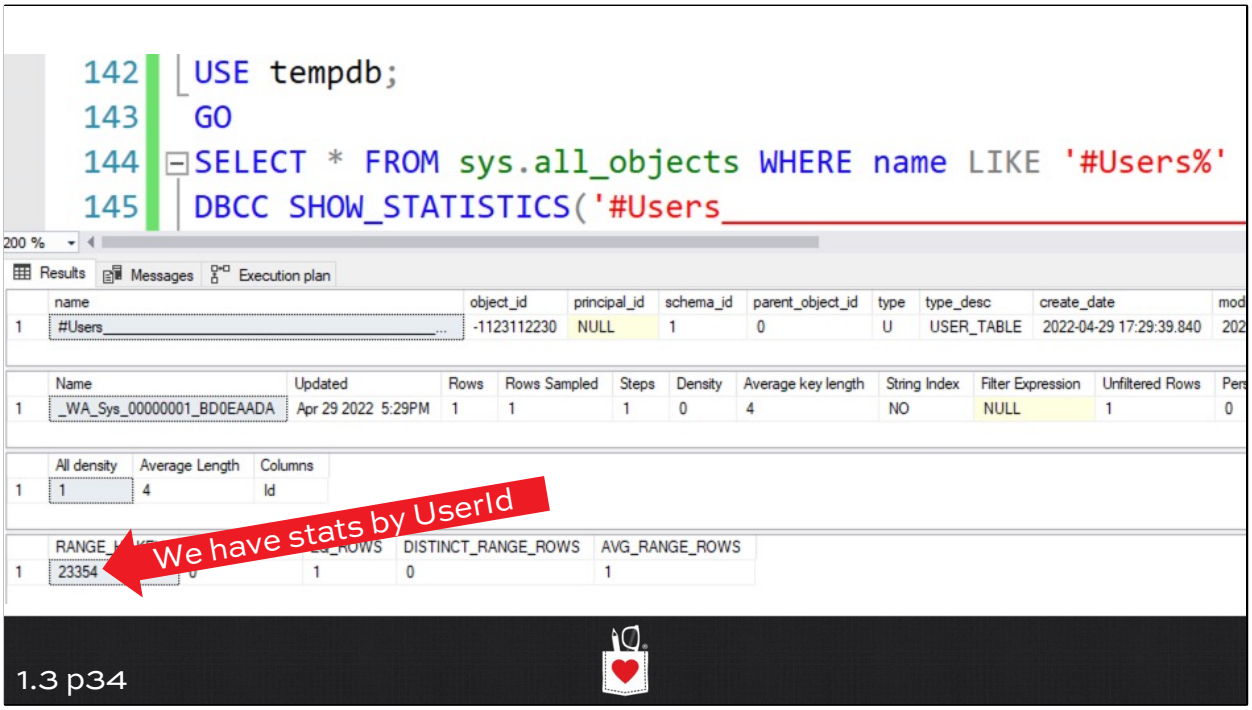


1.3 p32

Digging into the stats

To see temp table stats, we'll need to run the code interactively (so the #table doesn't get destroyed):

```
SELECT Id, DisplayName
    INTO #Users
    FROM dbo.Users
    WHERE Location = N'Forest of Dean, United Kingdom'

/* Read the plan on this, get the temp table's statistics name */
SELECT TOP 200 u.DisplayName, p.Title, p.Id, p.CreationDate
    FROM #Users u
    INNER JOIN dbo.Posts p ON u.Id = p.OwnerUserId
    ORDER BY p.CreationDate;

USE tempdb;
GO
SELECT * FROM sys.all_objects WHERE name LIKE '#Users%'
DBCC SHOW_STATISTICS('#Users
```



```
147  /* Go back to Stack, check the stats for that OwnerUserId */
148  USE StackOverflow;
149  GO
150  DBCC SHOW_STATISTICS('dbo.Posts', 'OwnerUserId')
```

200 %

Results | Messages

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows	Persisted Sample Percent
1	OwnerUserId	Jul 16 2018 3:19PM	40700647	40700647	161	0.09686216	8	NO	NULL	40700647	0

	All density	Average Length	Columns
1	2.615199E-07	4	OwnerUserId
2	2.456963E-08	8	OwnerUserId, Id

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
7	15168	211111	11911	1508	139.994
8	19068	194583		68	116.6565
9	22656			1488	103.1042
10	29407	227707	21531	2553	89.19193
11	37213	220343	7970	2502	88.06675

23354 is here

Diabolical

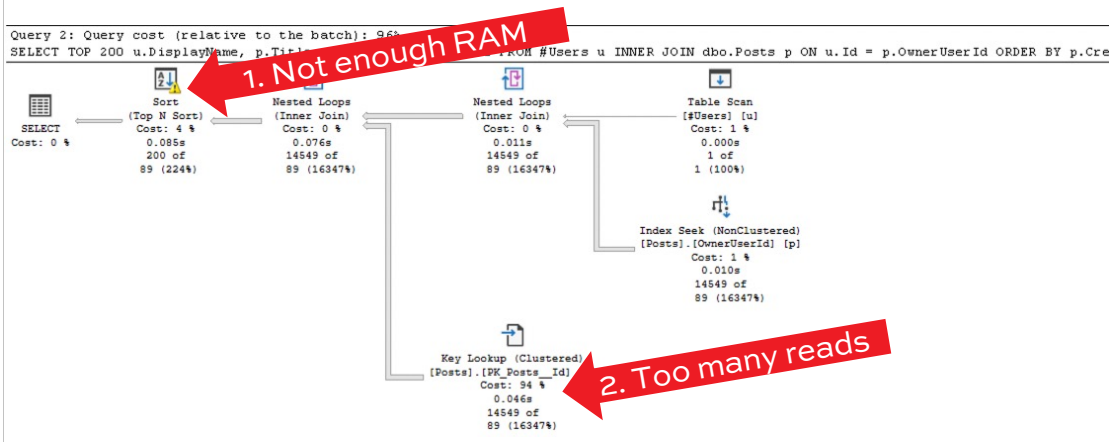
1.3 p35

Fix one problem, find another

Problem #1: SQL Server didn't know that a prolific user lived in Forest of Dean

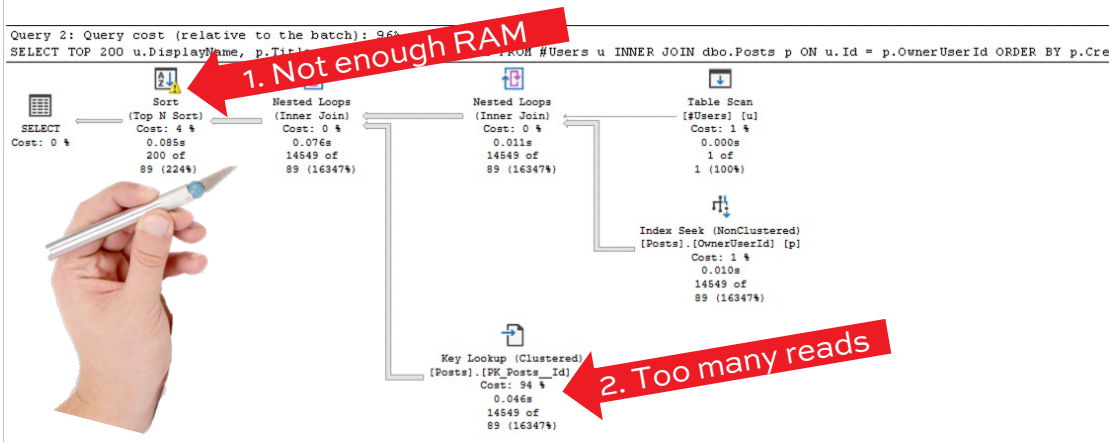
Problem #2: even when SQL Server knew the UserId, that Id still didn't have his own bucket, so we hit the 201 buckets problem

What issues do we need to solve?



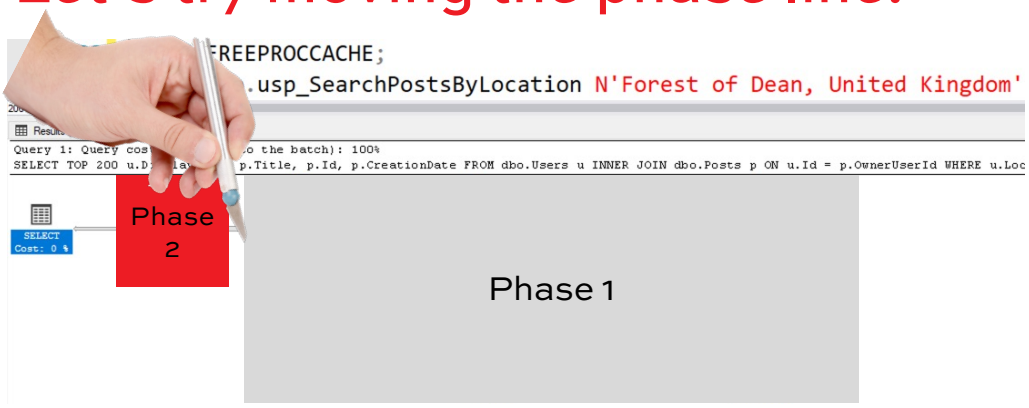
1.3 p37

Let’s try cutting later.



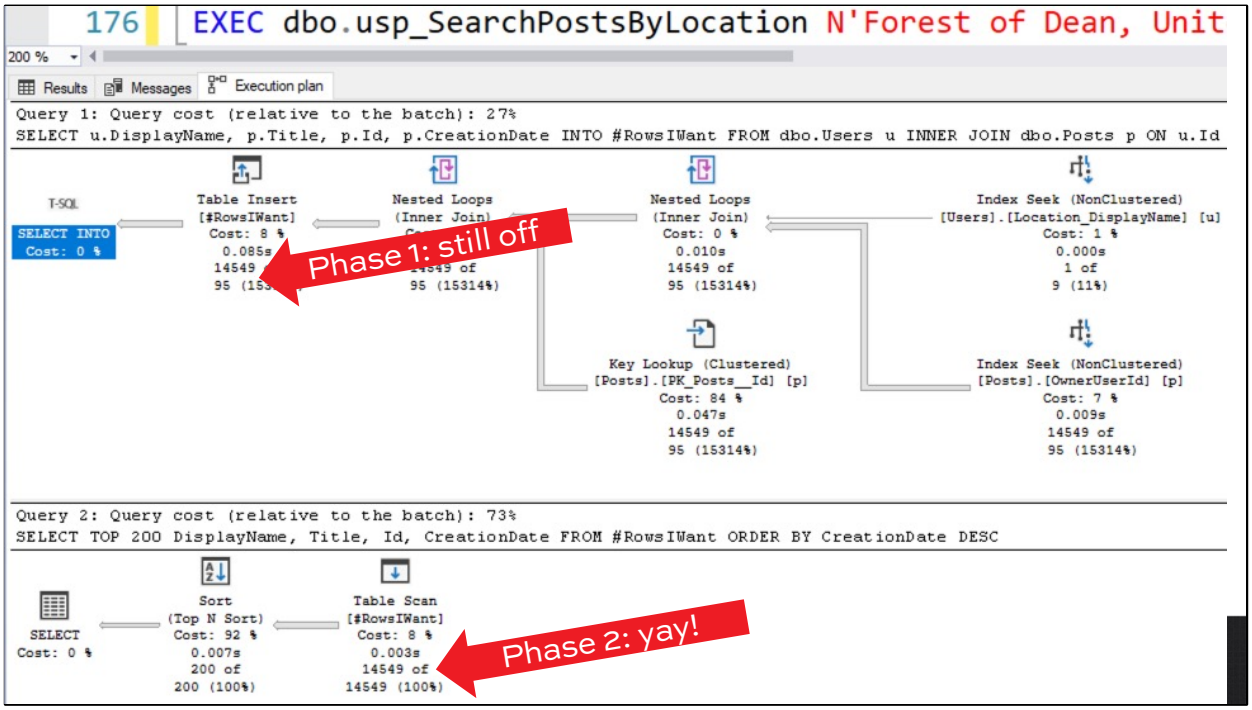
1.3 p38

Let's try moving the phase line.



1.3 p39

```
CREATE OR ALTER PROC dbo.usp_SearchPostsByLocation
    @Location VARCHAR(100) AS
BEGIN
    SELECT u.DisplayName, p.Title, p.Id, p.CreationDate
      INTO #RowsIWant
      FROM dbo.Users u
      INNER JOIN dbo.Posts p ON u.Id = p.OwnerUserId
      WHERE u.Location = @Location;

    SELECT TOP 200 DisplayName, Title, Id, CreationDate
    FROM #RowsIWant
    ORDER BY CreationDate DESC;
END
```



Here, this is overkill.

This query ran in under a second anyway.

The idea is to teach you the techniques quickly.

Ideally, best to worst:

1. Get the early estimates right
2. You'll only be left with later estimate issues
3. Mitigate those with the X-Acto knife technique
4. Even when you do that, you can still hit edge case issues like the 201 buckets problem

1.3 p42

There's a catch.

Temp tables are reused across sessions.

You can inherit someone else's stats, even if you drop the temp tables.

Learn more in the temp tables module in Fundamentals of TempDB.

For now, think of this technique as OPTION (RANDOM RECOMPILE)

1.3 p43

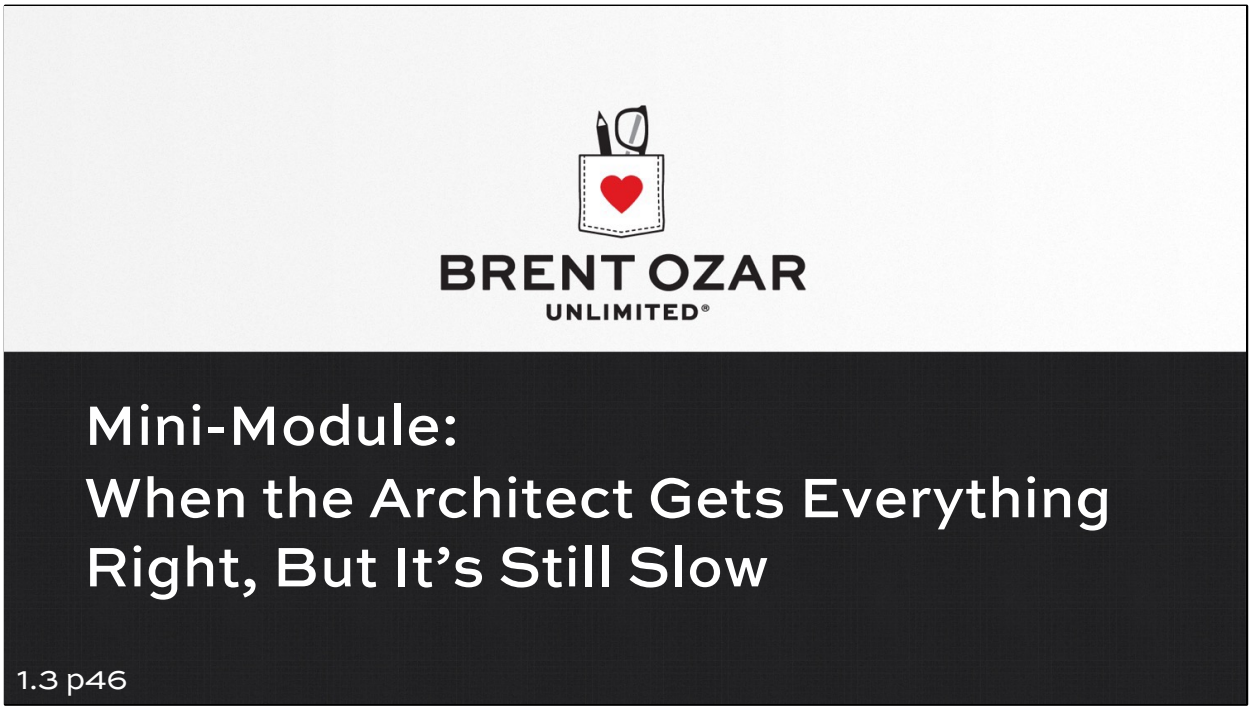
Right to left, top to bottom

Early estimation errors = late estimation errors, too. Fix the early ones first.

Late estimations are just based on averages.

If you have outlier values (like Forest of Dean), you need the X-Acto Knife technique with temp tables.

Just be aware that it's OPTION (RANDOM RECOMPILE).



“My estimates are all close!”

3 possible fixes:

- Add or tune indexes so there’s less work to do
  (we cover that in Mastering Index Tuning)
- Reorder operations in the query
- Add hardware
  (we cover that in Mastering Server Tuning)

1.3 p47

I'm going to use a dumb example.

You would never write a query this terribad:

1. Total up all the Votes by UserId
2. Show the top voters in a specific location

But bear with me. It'll tell the story.

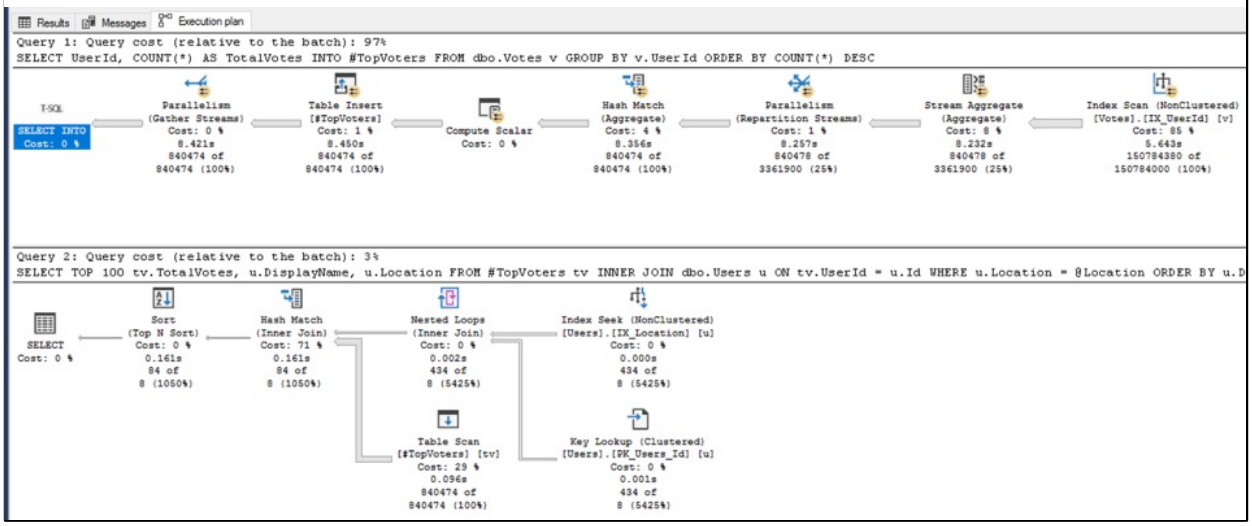
```sql
CREATE OR ALTER PROC dbo.usp_TopVotersInCity @Location NVARCHAR(40) AS
BEGIN
/* Find the users who have left the most votes */
SELECT UserId, COUNT(*) AS TotalVotes
   INTO #TopVoters
   FROM dbo.Votes v
   GROUP BY v.UserId
   ORDER BY COUNT(*) DESC;

SELECT TOP 100 tv.TotalVotes, u.DisplayName, u.Location
   FROM #TopVoters tv
   INNER JOIN dbo.Users u ON tv.UserId = u.Id
   WHERE u.Location = @Location
   ORDER BY u.DisplayName;
END
GO
```

Scan all the votes

Do some filtering

1.3 p49

Runs, but takes 8-10 seconds



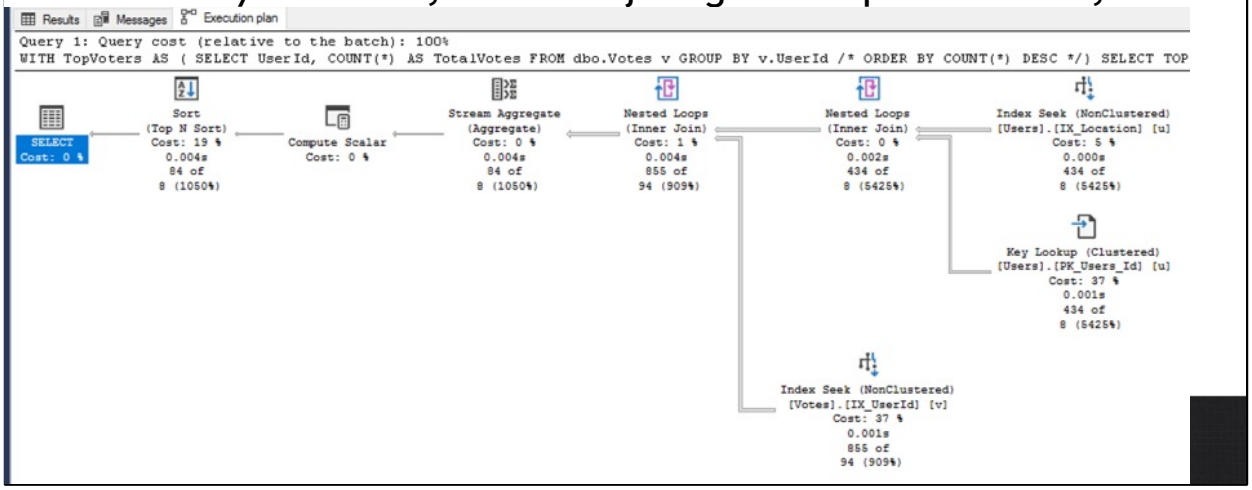
Common Table Expression (CTE)

Clamp the queries together into one,
and let SQL Server reorder operations.

```
CREATE OR ALTER PROC dbo.usp_TopVotersInCity_CTE @Location NVARCHAR(40) AS
BEGIN
WITH TopVoters AS (
    SELECT UserId, COUNT(*) AS TotalVotes
      FROM dbo.Votes v
      GROUP BY v.UserId
      /* ORDER BY COUNT(*) DESC */)
SELECT TOP 100 tv.TotalVotes, u.DisplayName, u.Location
  FROM TopVoters tv
  INNER JOIN dbo.Users u ON tv.UserId = u.Id
  WHERE u.Location = @Location
  ORDER BY u.DisplayName;
END
```

Runs instantly

Because SQL Server says, lemme find the right Users first by Location, THEN I'll just go add up their votes,



CTEs are great when...

You have a long query to tune (like a proc)

You're not sure which parts should be done first

When the row estimates for each query look fine

Change temp tables & child procs into CTEs,
let SQL Server reorder the processing,
and see if it goes faster.

	Temp tables & table variables	CTEs & APPLY
Scope (lasts for)	1 session, with gotchas	1 statement
Materialized to disk*	Yes*	No*
Executed	Just once, when you insert	Can be multiple times even in 1 statement
Has statistics (which can be good or bad)	Temp tables: yes Table variables: no, til 2019	No
Good when	The initial estimates suck and we need to go back to the architect repeatedly	The initial estimates are accurate

1.3 p55

* Not every temp object gets written to disk, and CTE/APPLY can cause TempDB spills just like any other query operation