I'm using screenshots here.

I set up an expensive cloud VM with:

- 96 CPU cores, 768GB RAM
- SQL Server 2019 CU6
- Stack Overflow 2018-06 database
- Cost Threshold for Parallelism = 50
- Max Degrees of Parallelism = 0

3.1 p2

What we'll cover

How SQL Server balances input across threads

How to spot when that isn't working

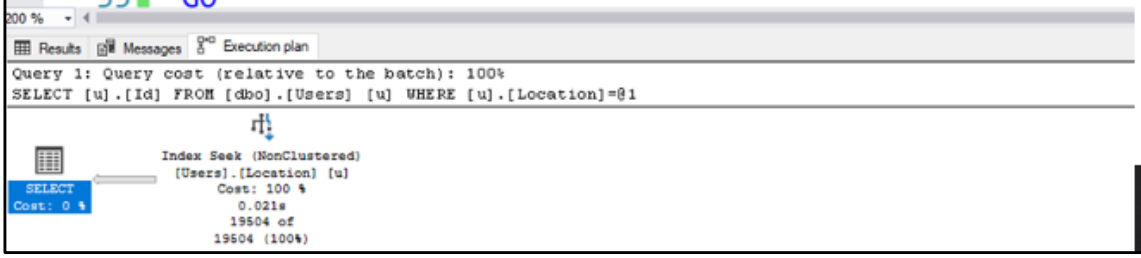
What to look for when it DOESN'T balance work:

- Unevenly skewed data
- Spills on parallelism operators
- Spools: skewed all to one thread

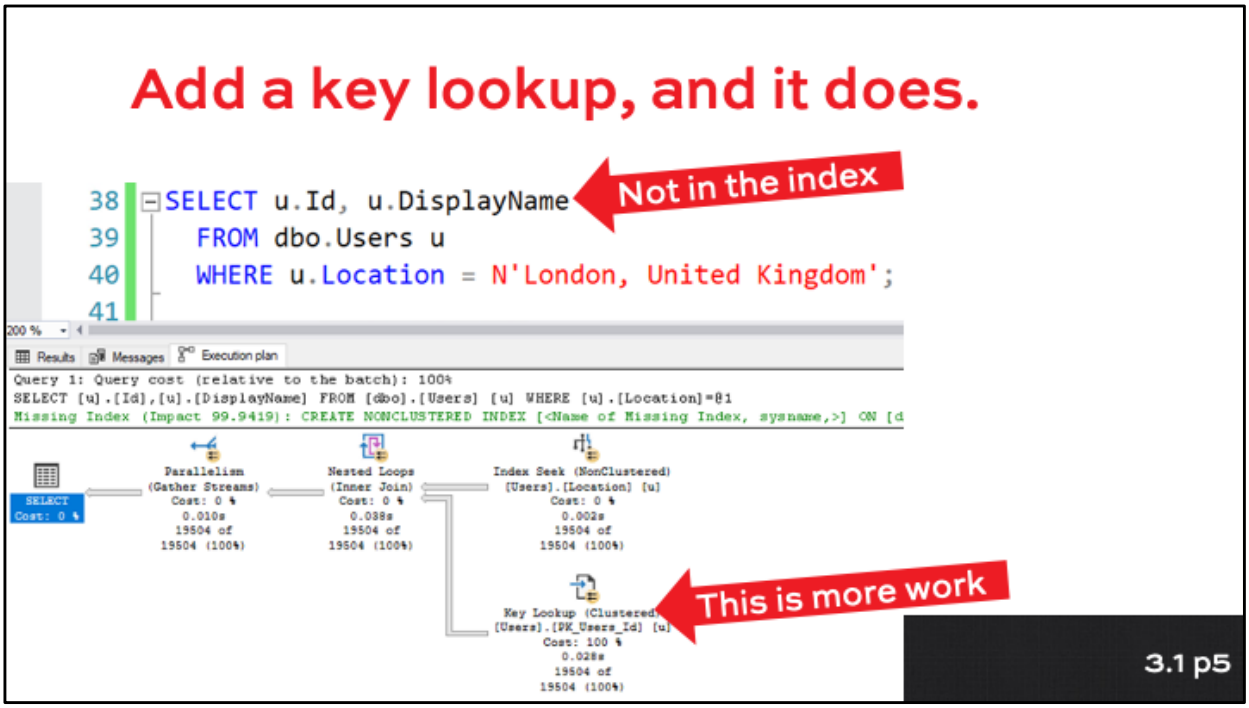
3.1 p3

Easy query plans don't go parallel.

```
28  CREATE INDEX Location ON dbo.Users(Location);
29  GO
30
31  /* If I need to find all the users in one location: */
32  SELECT u.Id
33    FROM dbo.Users u
34    WHERE u.Location = N'London, United Kingdom';
35  GO
```

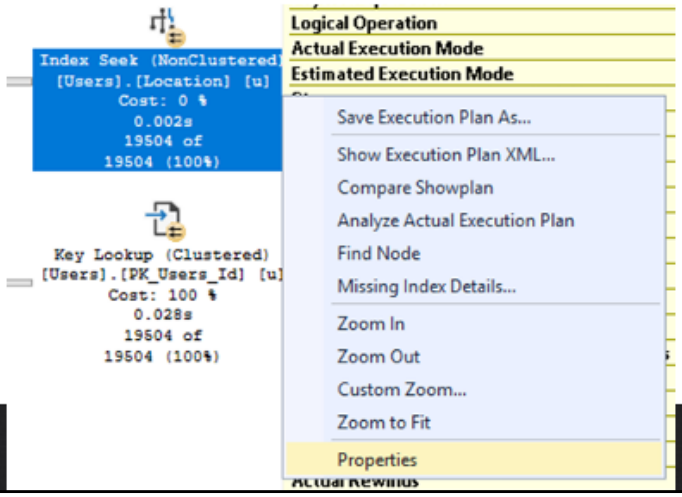


3.1 p4

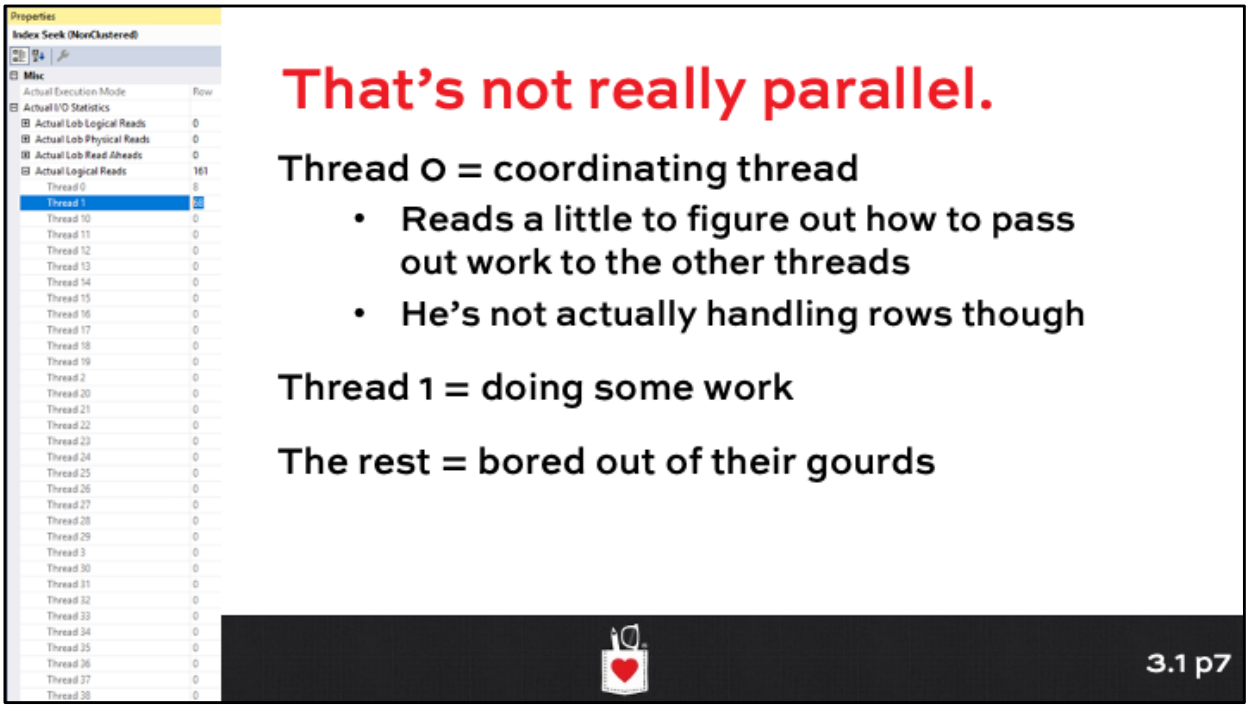


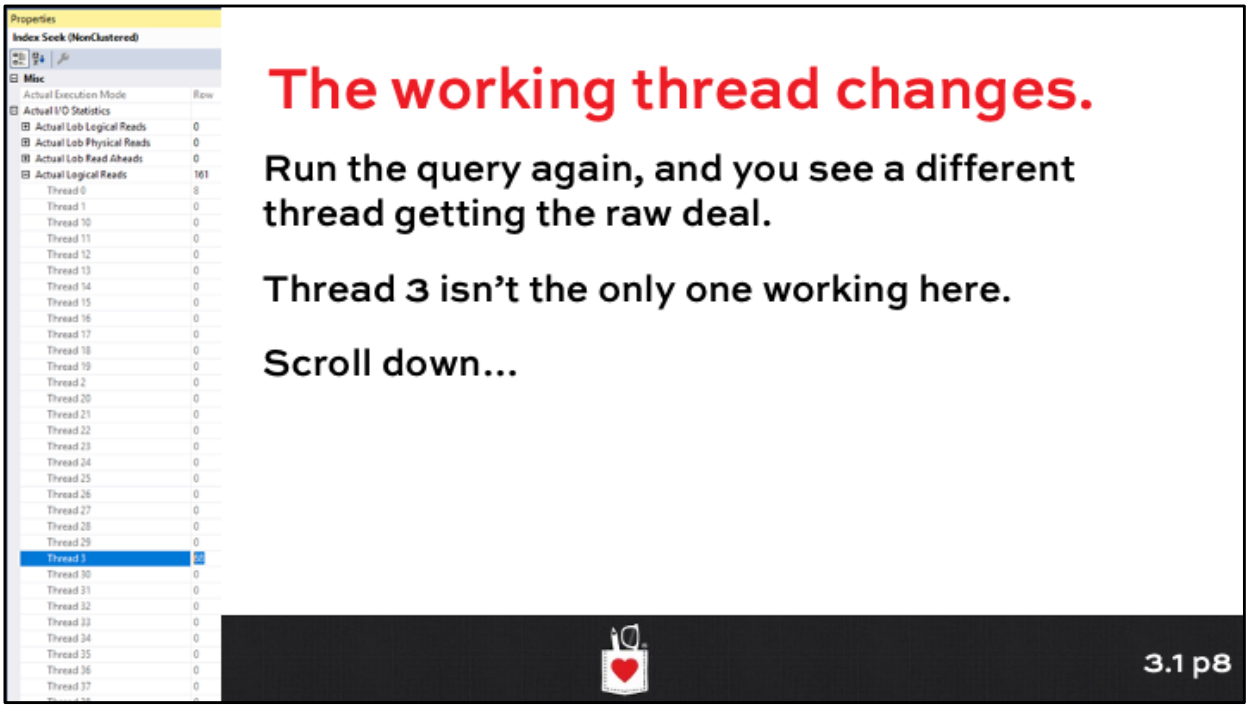
Does it really go parallel?

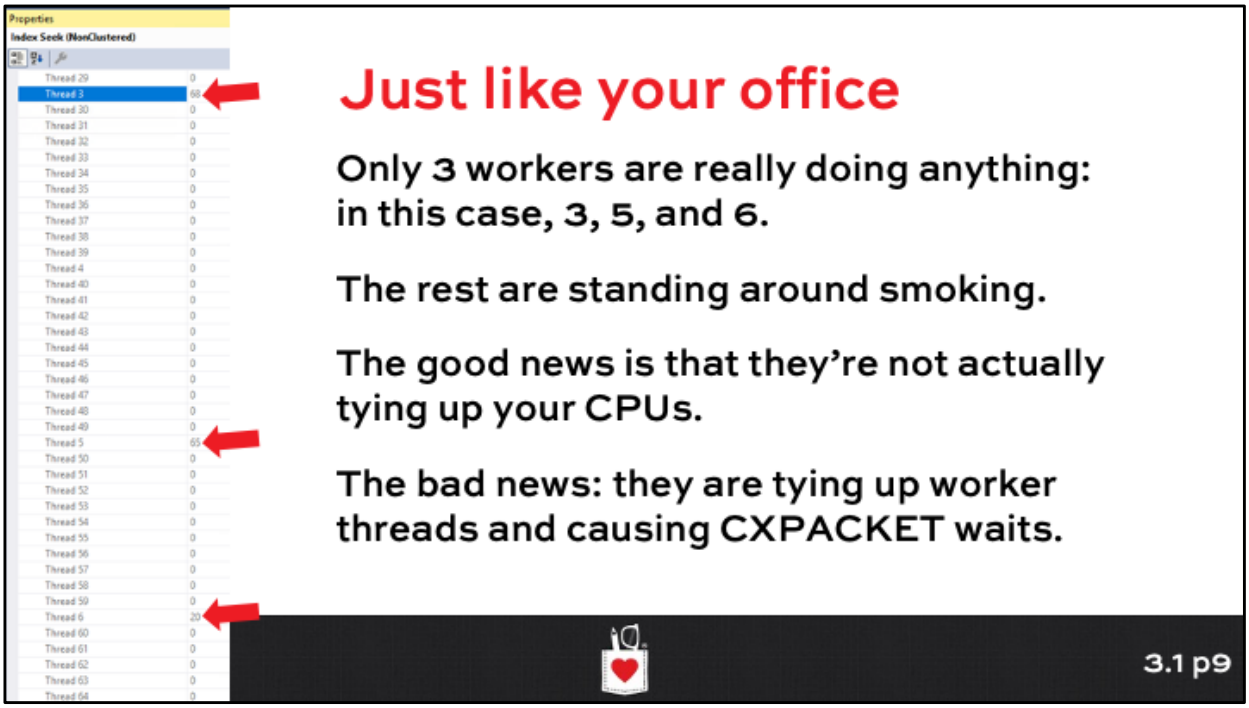
Go into the index seek's properties:



3.1 p6





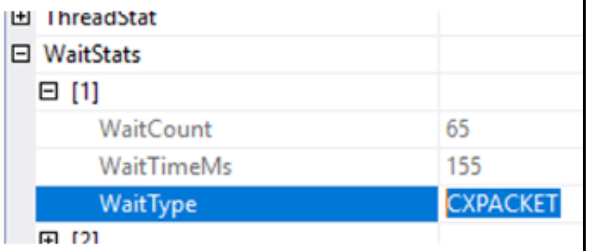


Seeing the waits

Right-click on the SELECT operator and go into its properties.

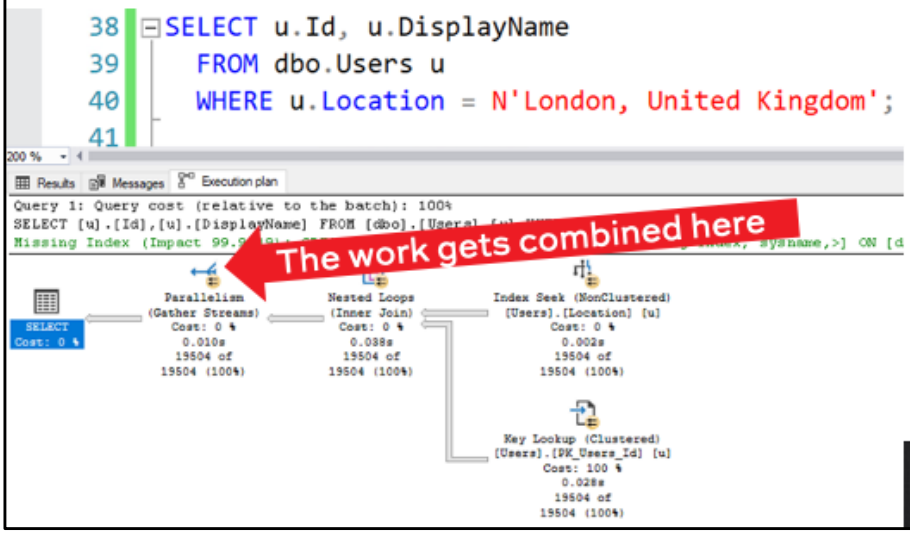
On modern versions of SQL Server, the wait stats are shown in the details.

Why 155ms? Well, the whole plan is unbalanced.



3.1 p10

The rows start out unbalanced.



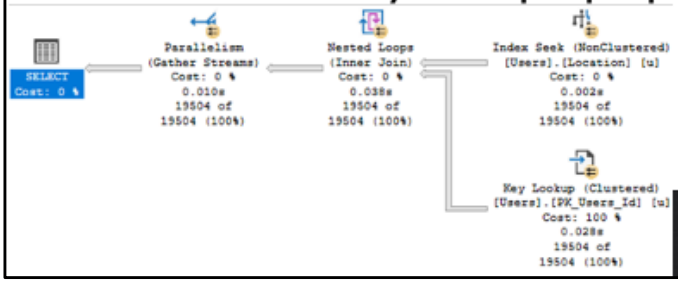
3.1 p11

This plan is really 3-dimensional.

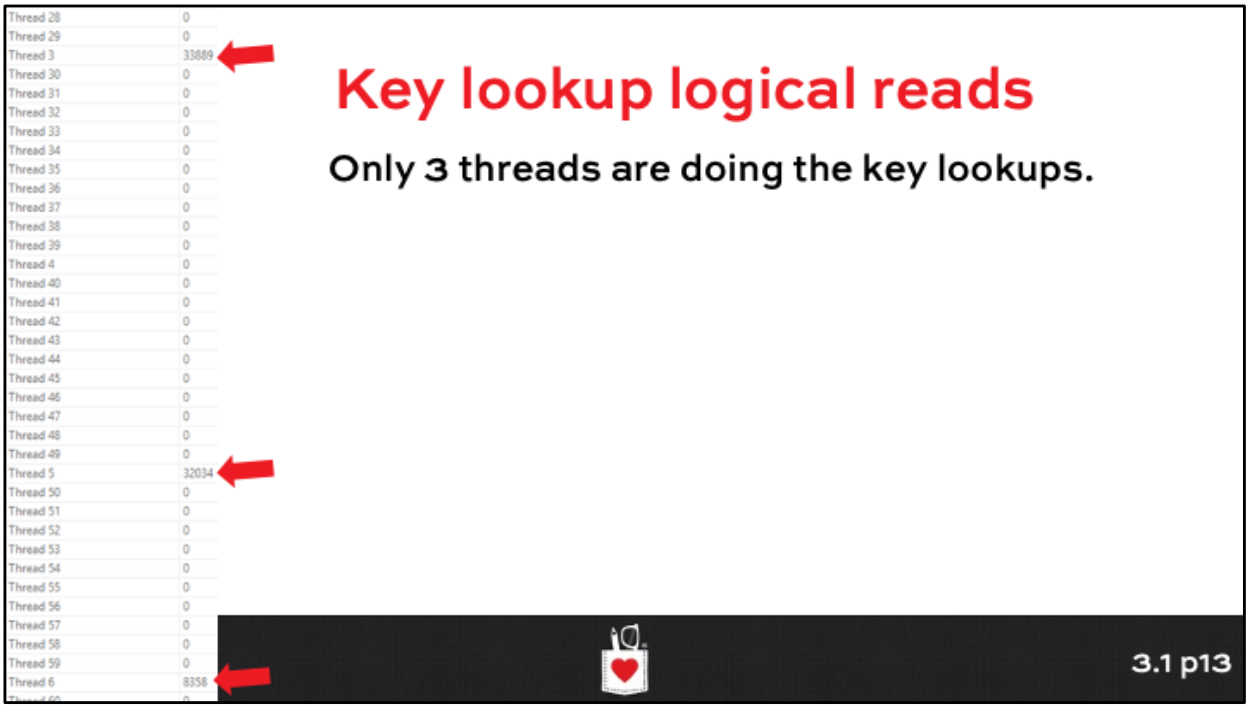
Everything to the right of the Parallelism operator was done by 64 threads.

They're not sharing the work of the join and the key lookups.

Look at the key lookup's properties...



3.1 p12

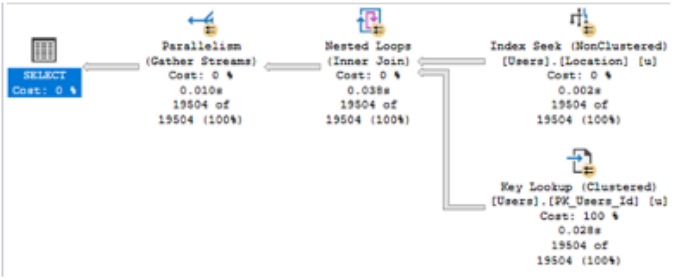


In this plan...that's fine.

It runs in 10 ms.

Sure, work isn't balanced, and there's CXPACKET.

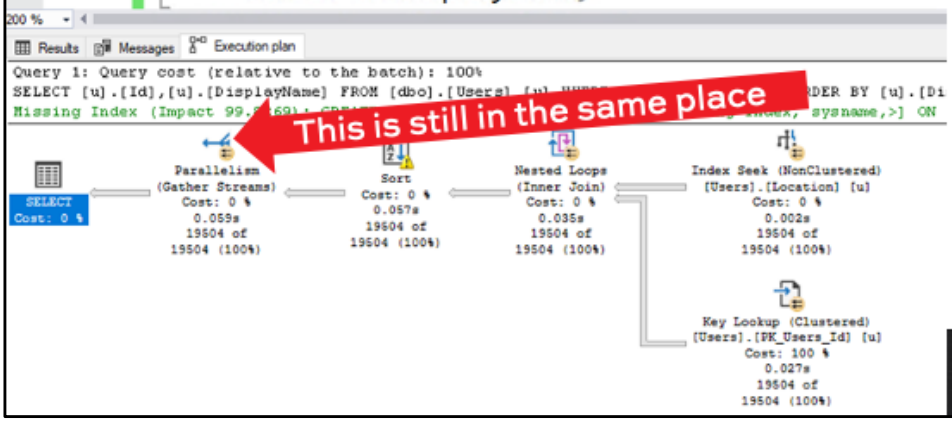
Who cares? It's fine.



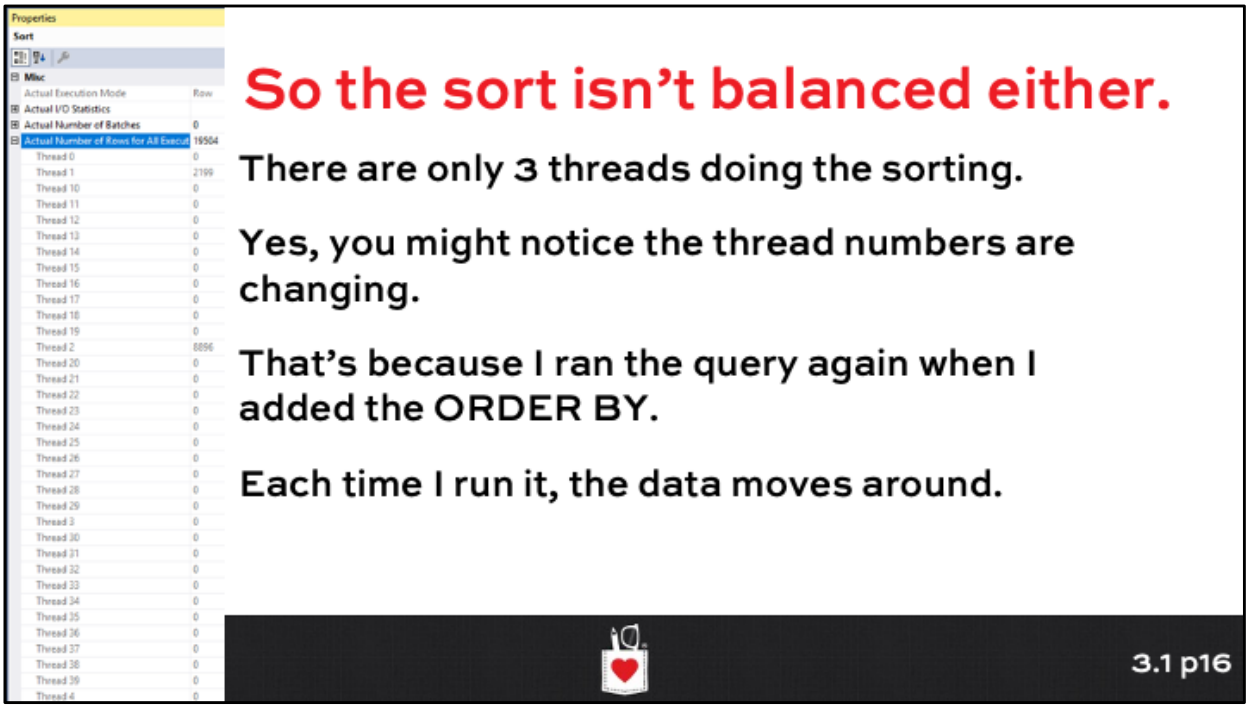
Let's add more work to the query.

Adding an ORDER BY

```
53  SELECT u.Id, u.DisplayName
54    FROM dbo.Users u
55    WHERE u.Location = N'London, United Kingdom'
56    ORDER BY u.DisplayName;
```



3.1 p15



What if we wanted to balance it?

Well, we don't, not for this query.

This query runs in 59 milliseconds.

But it's generating over 3 seconds of CXPACKET waits because we have so many cores sitting around.

Can we fix that with a hint?

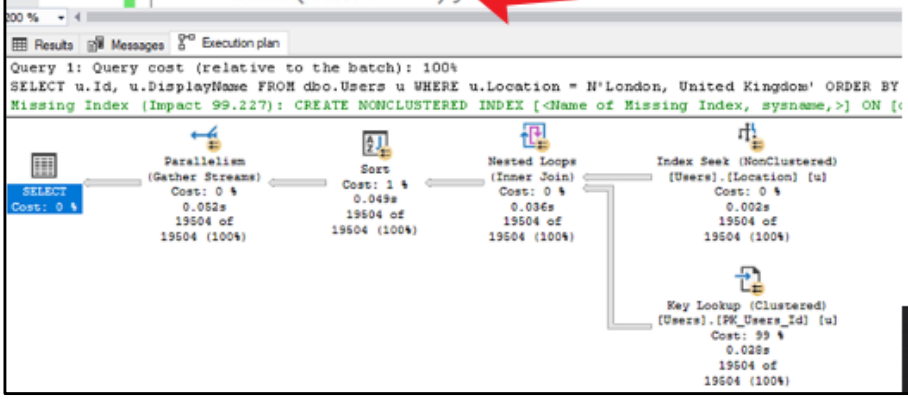
⊟ ThreadStat	
Branches	1
⊞ ThreadReservation	
UsedThreads	64
⊟ WaitStats	
⊟ [1]	
WaitCount	129
WaitTimeMs	3197
WaitType	CXPACKET

3.1 p17

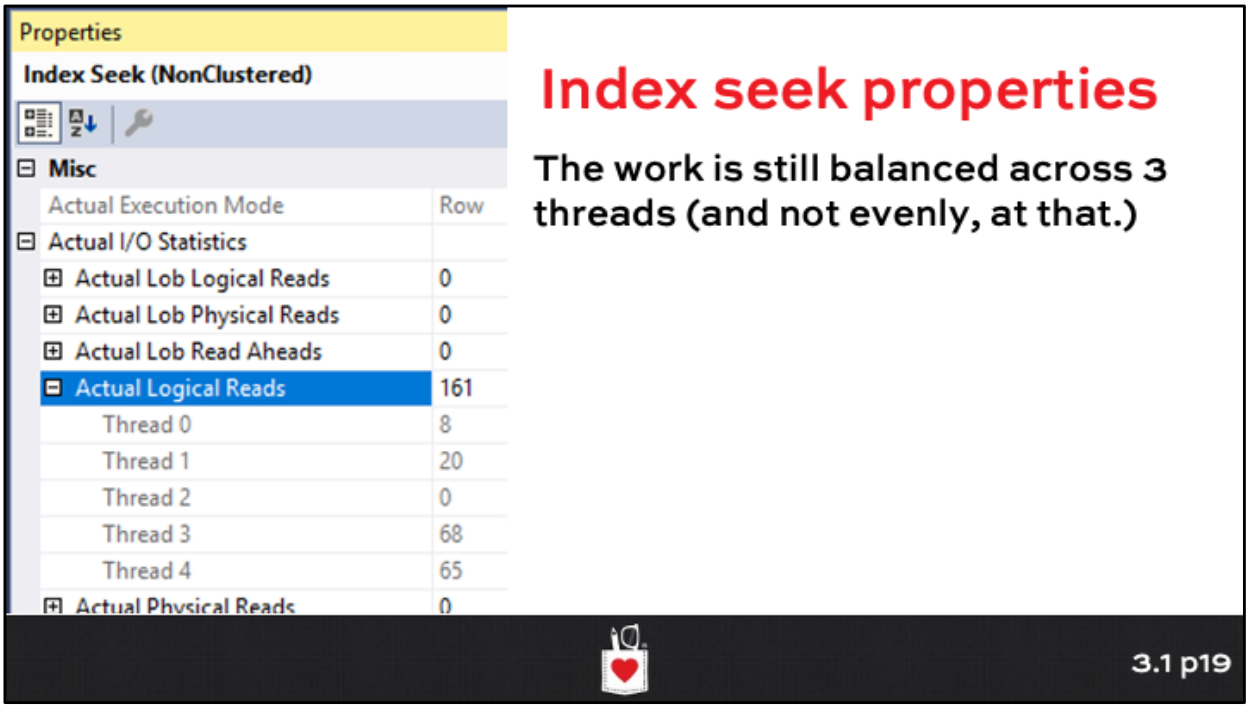
MAXDOP 4: it still looks parallel.

```
69  SELECT u.Id, u.DisplayName
70    FROM dbo.Users u
71    WHERE u.Location = N'London, United Kingdom'
72    ORDER BY u.DisplayName
73    OPTION(MAXDOP 4);
```

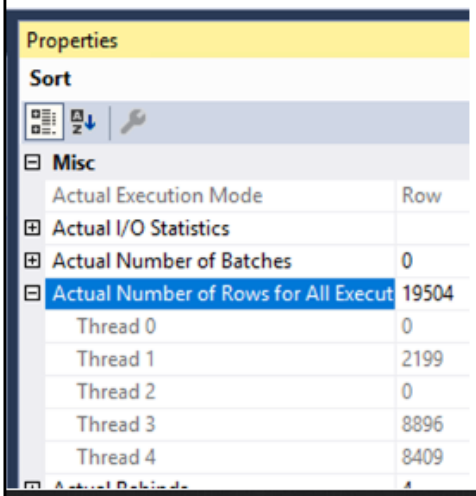
New



3.1 p18



And the sort is more balanced

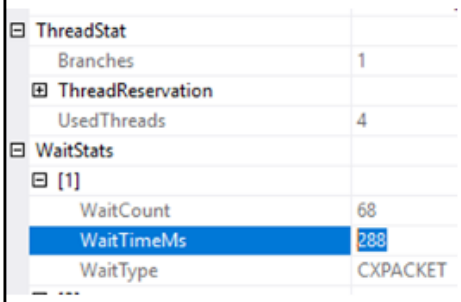


Although again, still not great.

But in each operator, there are less idle cores sitting around waiting for all this work to finish, so...

3.1 p20

There is way less CXPACKET.



The query isn't faster: it still takes around 50-60ms.

We haven't succeeded in dividing the work up more evenly across our 64 cores.

But who cares? It's a fast query.

Let's add a join.

3.1 p21

We have an index to support this:

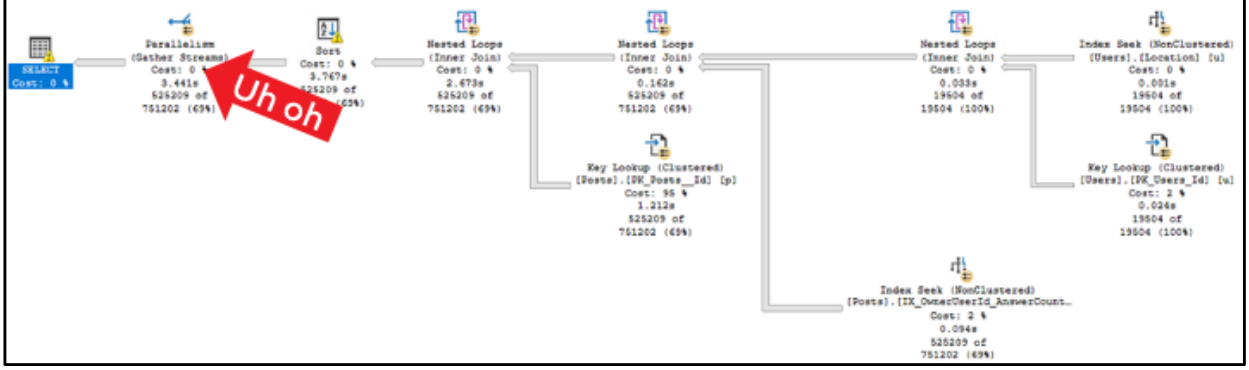
Get the posts for the people who live in London, and sort them by Posts.Score descending:

```
CREATE INDEX OwnerUserId ON dbo.Posts(OwnerUserId);
GO
SELECT u.Id, u.DisplayName, p.Id AS PostId, p.Title, p.Score
  FROM dbo.Users u
  INNER JOIN dbo.Posts p ON u.Id = p.OwnerUserId
  WHERE u.Location = N'London, United Kingdom'
  ORDER BY p.Score DESC, u.DisplayName;
```

3.1 p22

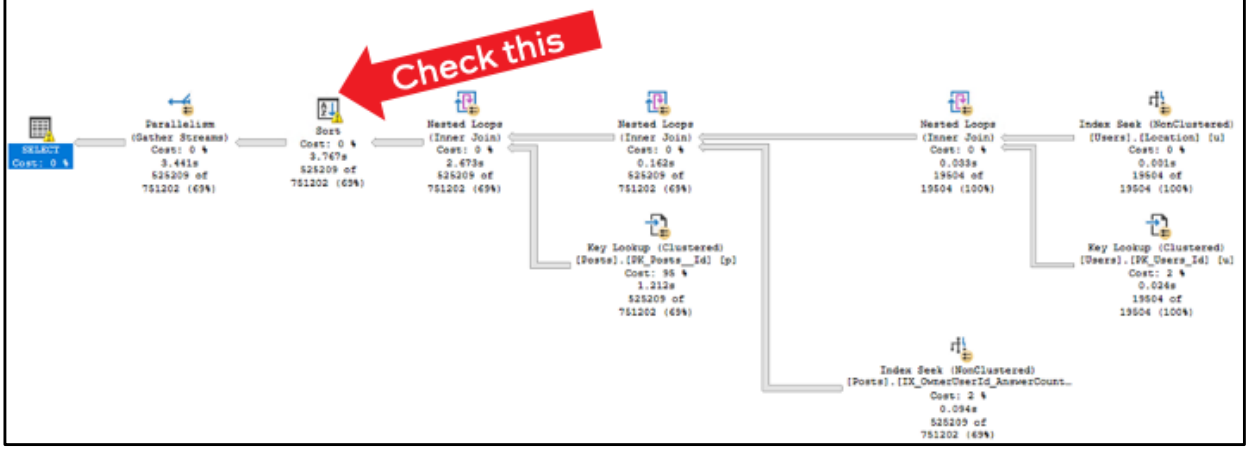
The parallelism icon is still at left

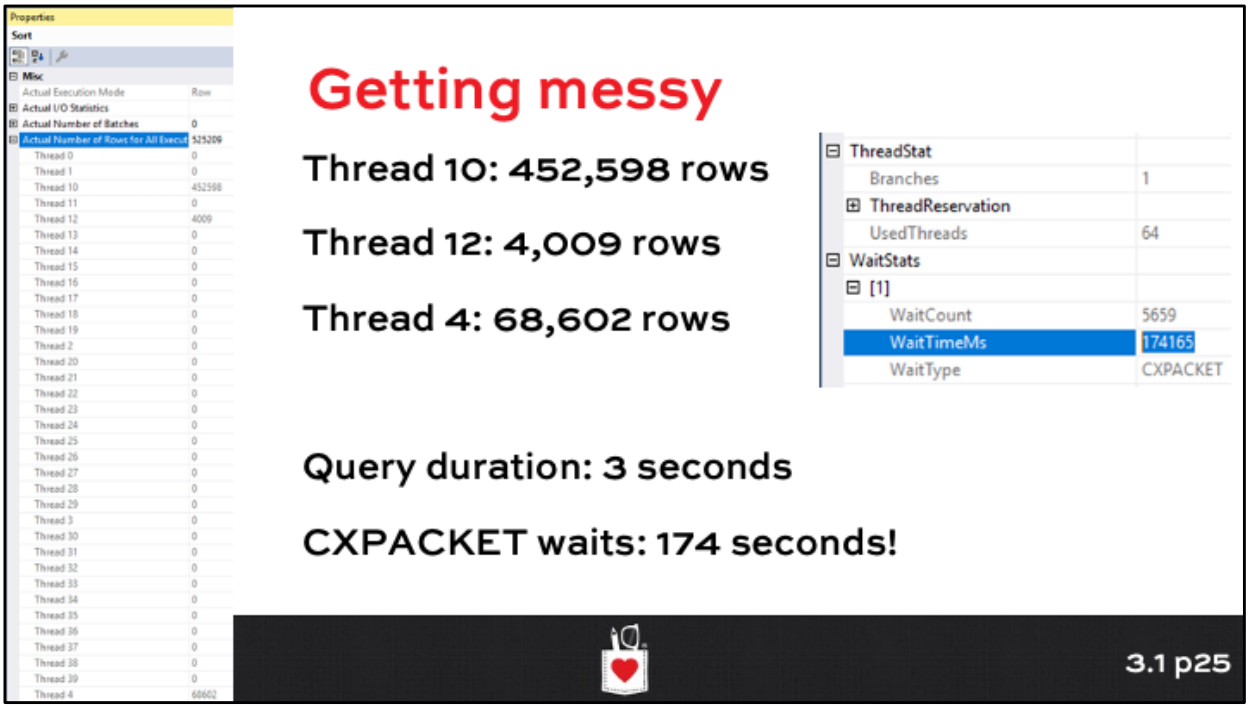
Which means EVERYTHING upstream was divided out between 64 cores, all based on how many rows each core found in the first operation:



Check the sort's properties

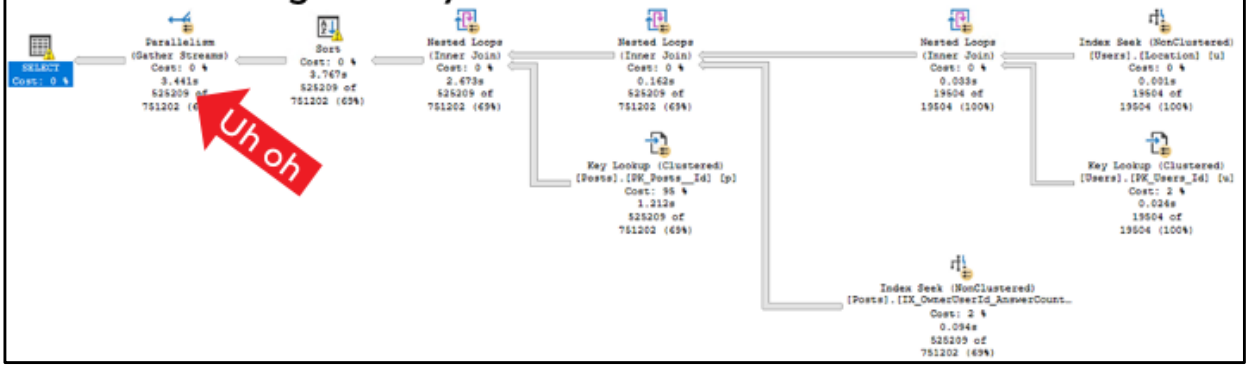
We'll look here to see how unbalanced it is right before the results are recombined:





The problem: wide areas of inbalanced operators

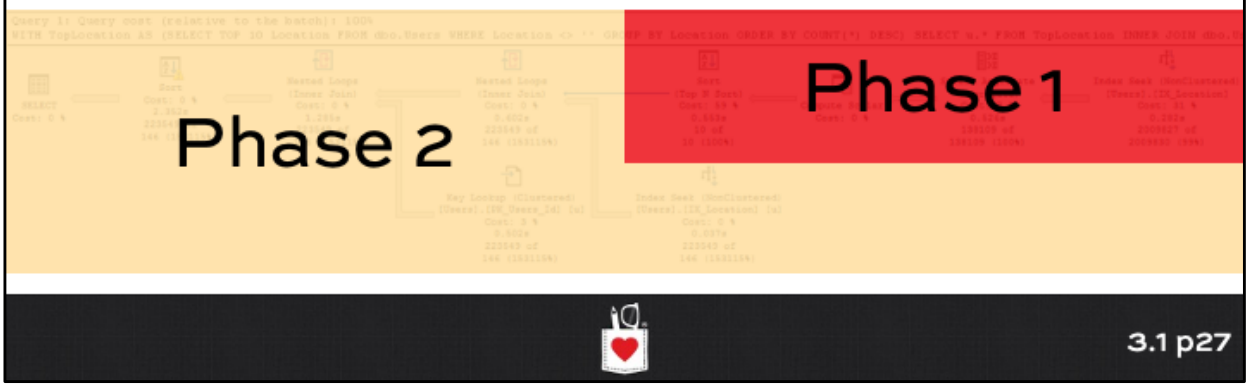
When you see a parallelism gather streams,
the more icons you have underneath it,
the more risk you have of imbalances not being able to leverage all of your CPU cores.



So how can we fix it?

Remember how I talk about estimated vs actual rows?

How I say that you look for the point where they're 10X off, and you separate out the plan into phases?



3.1 p27

It's the same thing for parallelism.

If there are a whole bunch of operators downstream from a Parallelism Gather Streams...

And you want that work to be more evenly divided across lots of CPU cores...

Then you need to break the query up into phases, and have SQL Server rebalance the work across cores.

3.1 p28

Our current query

```
SELECT u.Id, u.DisplayName, p.Id AS PostId, p.Title, p.Score
  FROM dbo.Users u
  INNER JOIN dbo.Posts p ON u.Id = p.OwnerUserId
  WHERE u.Location = N'London, United Kingdom'
  ORDER BY p.Score DESC, u.DisplayName;
```

Phase 1: go find the users in London

Phase 2: go find their posts

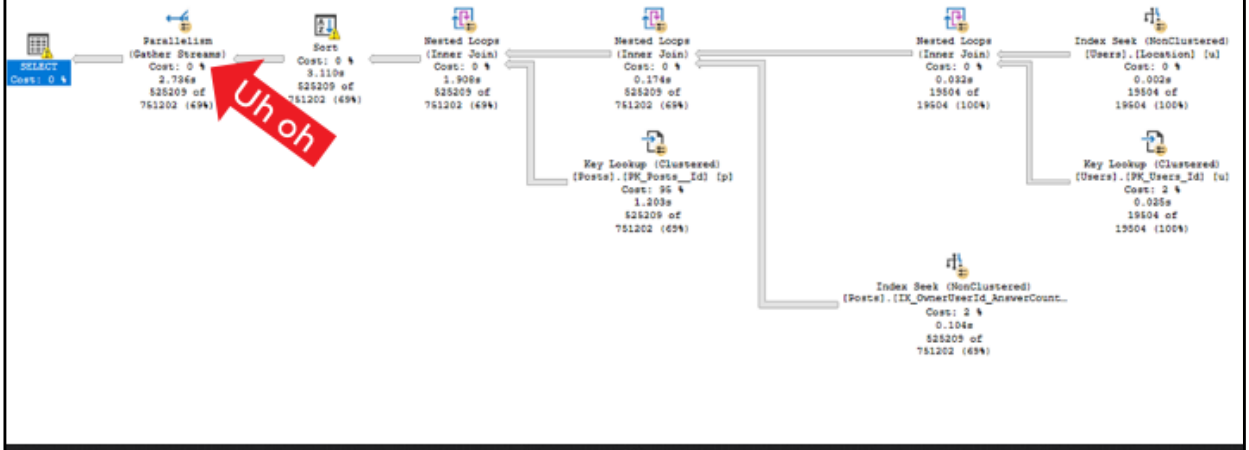
Phase 3: sort the joined result set

3.1 p29

Try separating phase 1 with a CTE

```
WITH MatchingUsers AS (
    SELECT u.Id, u.DisplayName
    FROM dbo.Users u
    WHERE u.Location = N'London, United Kingdom'
)
SELECT u.Id, u.DisplayName, p.Id AS PostId, p.Title, p.Score
  FROM MatchingUsers u
  INNER JOIN dbo.Posts p ON u.Id = p.OwnerUserId
  ORDER BY p.Score DESC, u.DisplayName;
```

It doesn't work. Same plan.



3.1 p31

Remember from Fundamentals...

T-SQL is a declarative language.

You're describing the shape of your *result set*,
not the shape of your execution plan.

There are hacks you can use, but...

Trick to force the CTE to run first

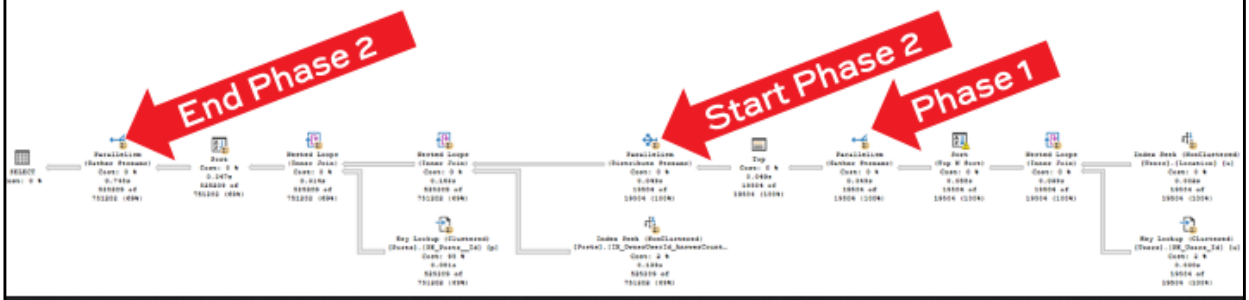
THIS IS NOT A GOOD IDEA, but you can do it:

```
WITH MatchingUsers AS (
    SELECT TOP 2147483647 u.Id, u.DisplayName
    FROM dbo.Users u
    WHERE u.Location = N'London, United Kingdom'
    ORDER BY u.DisplayName
)
SELECT u.Id, u.DisplayName, p.Id AS PostId, p.Title, p.Score
  FROM MatchingUsers u
  INNER JOIN dbo.Posts p ON u.Id = p.OwnerUserId
  ORDER BY p.Score DESC, u.DisplayName;
```

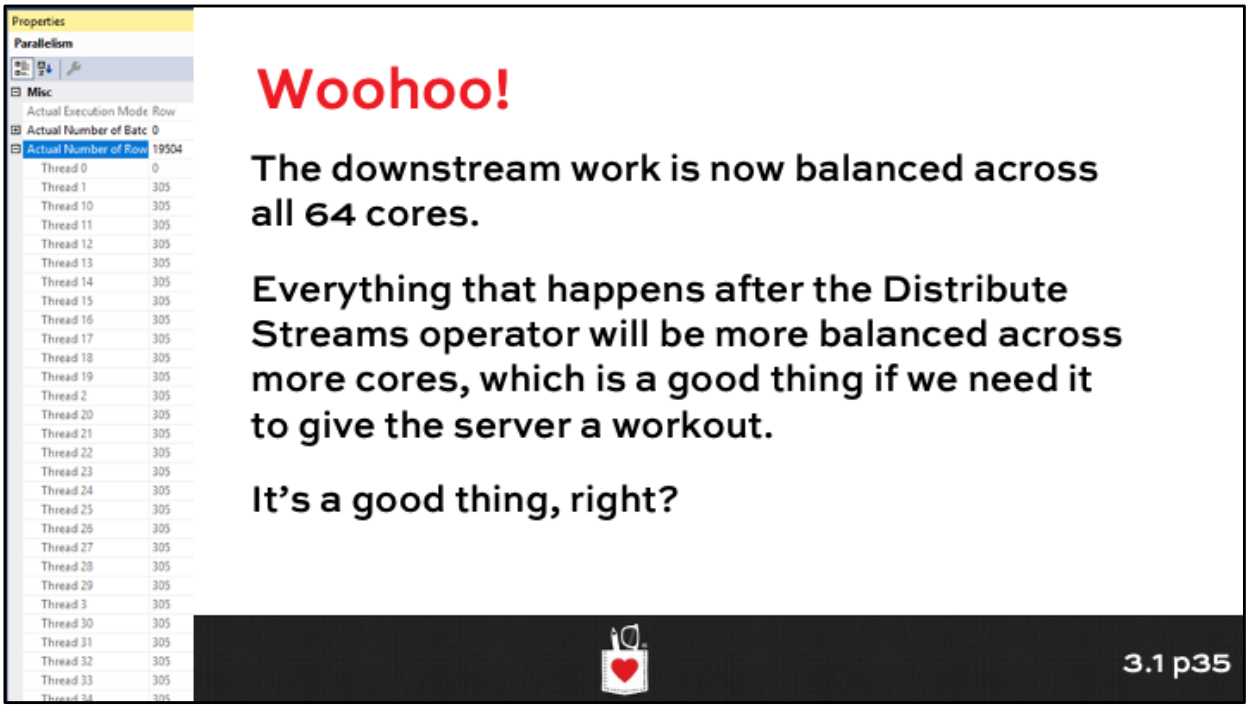
It does technically work.

The plan now has 3 parallelism operators instead of 1.

The "Distribute Streams" operator rebalances our work across multiple cores. Let's check its properties.



3.1 p34



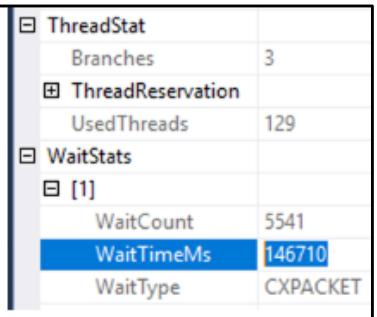
Well, here, no.

We actually end up with MORE CXPACKET waits.

Because we allocated 128 threads this time around instead of 64.

We had multiple branches in our plan that could all go parallel at the same time.

I'm not a big fan of the CTE TOP hack.
Let's use temp tables instead.



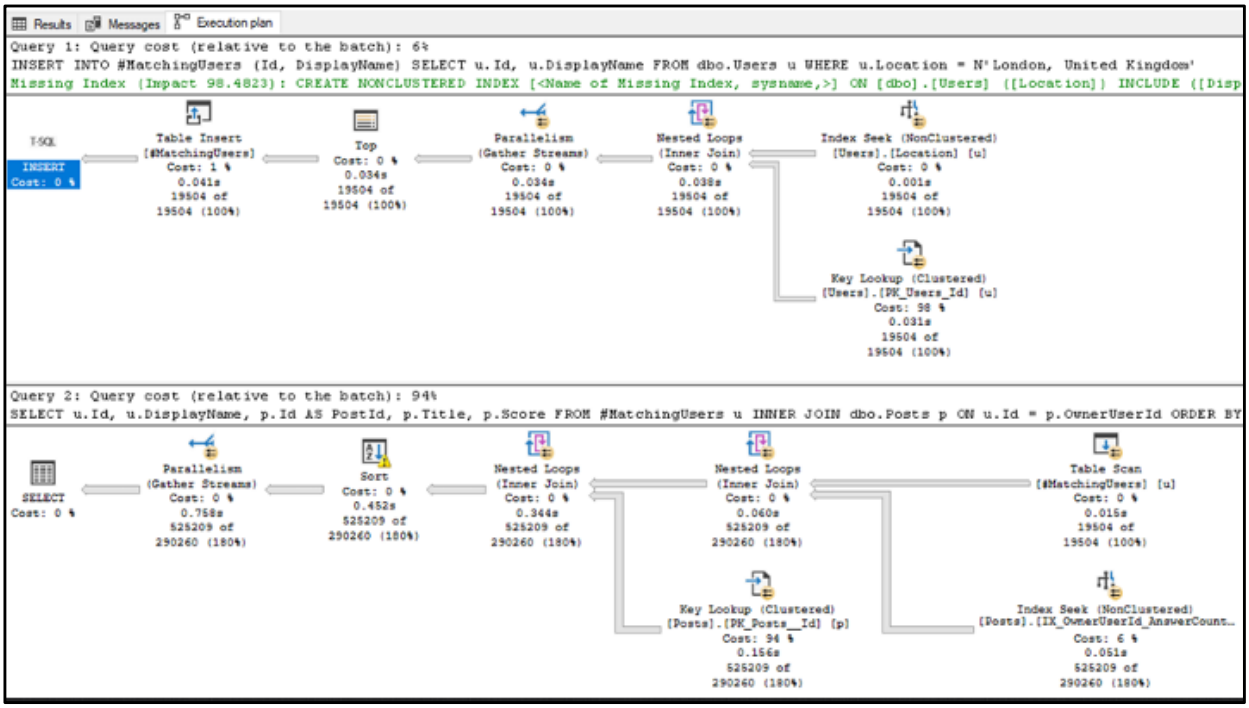
3.1 p36

Temp table version

```
CREATE TABLE #MatchingUsers (Id INT, DisplayName NVARCHAR(40));
INSERT INTO #MatchingUsers (Id, DisplayName)
    SELECT u.Id, u.DisplayName
    FROM dbo.Users u
    WHERE u.Location = N'London, United Kingdom';

SELECT u.Id, u.DisplayName, p.Id AS PostId, p.Title, p.Score
  FROM #MatchingUsers u
  INNER JOIN dbo.Posts p ON u.Id = p.OwnerUserId
  ORDER BY p.Score DESC, u.DisplayName;
```

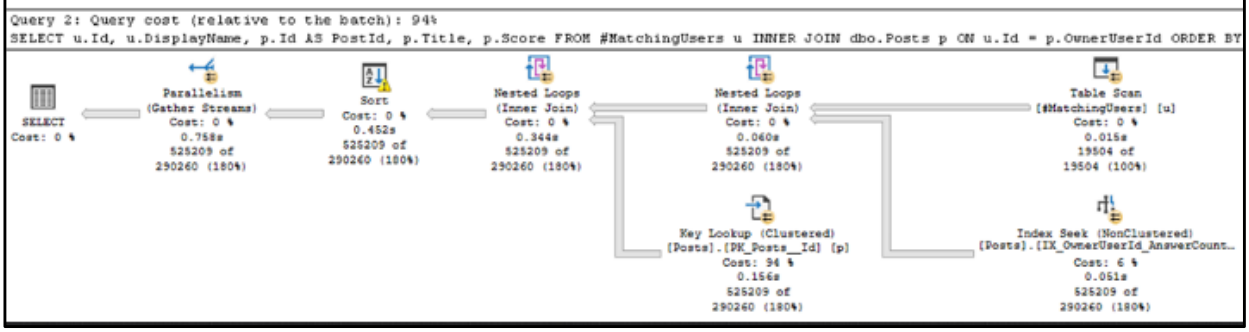
3.1 p37

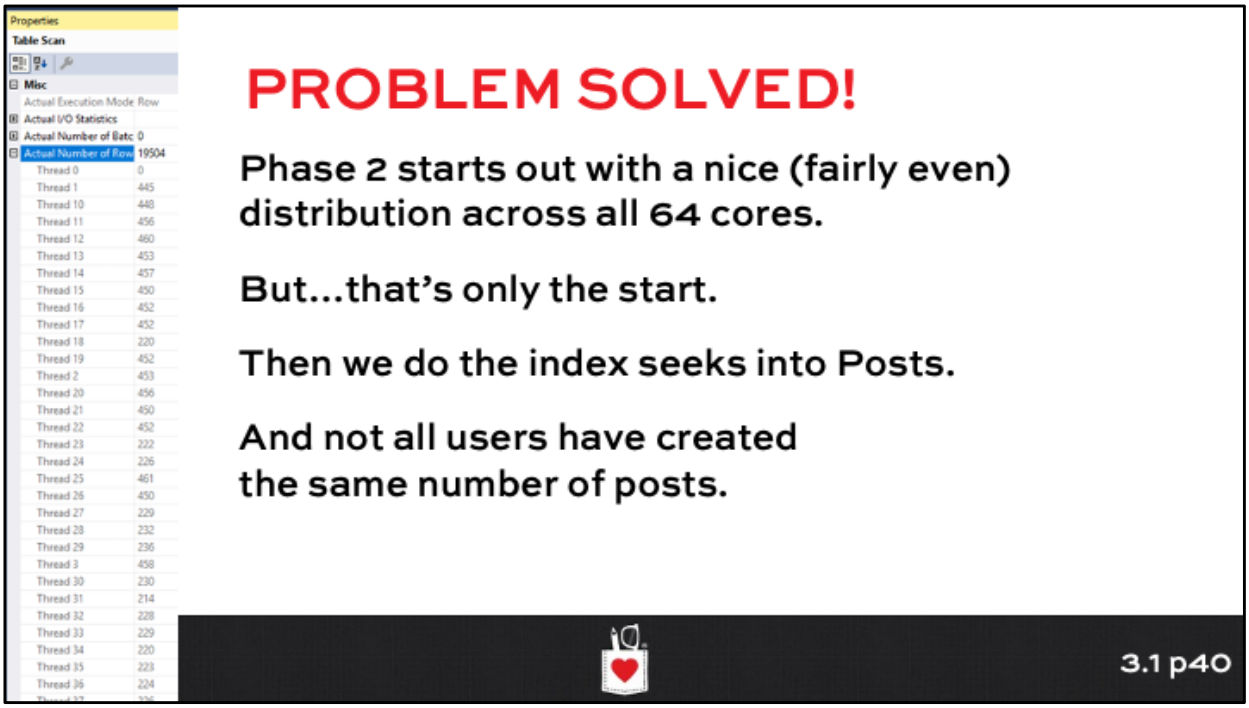


In the second query...

Phase 2 gets to start fresh.

When it scans the temp table, does it evenly divide the work across more than just 3 cores?

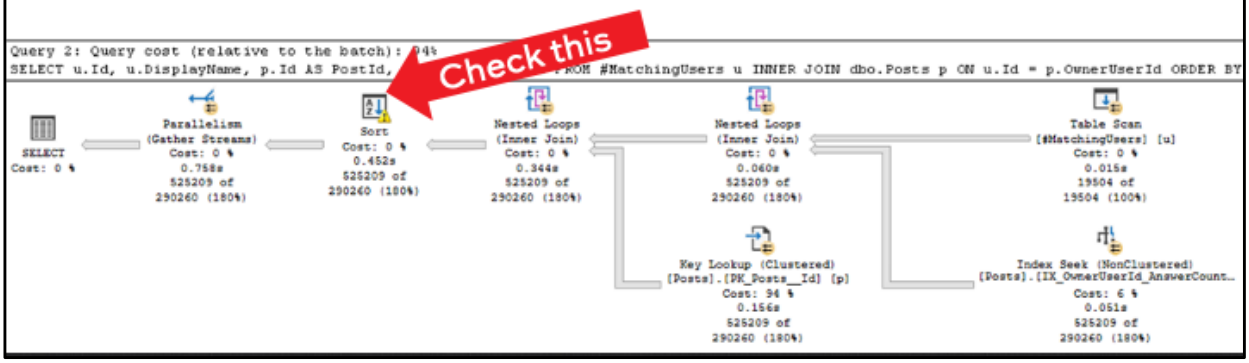


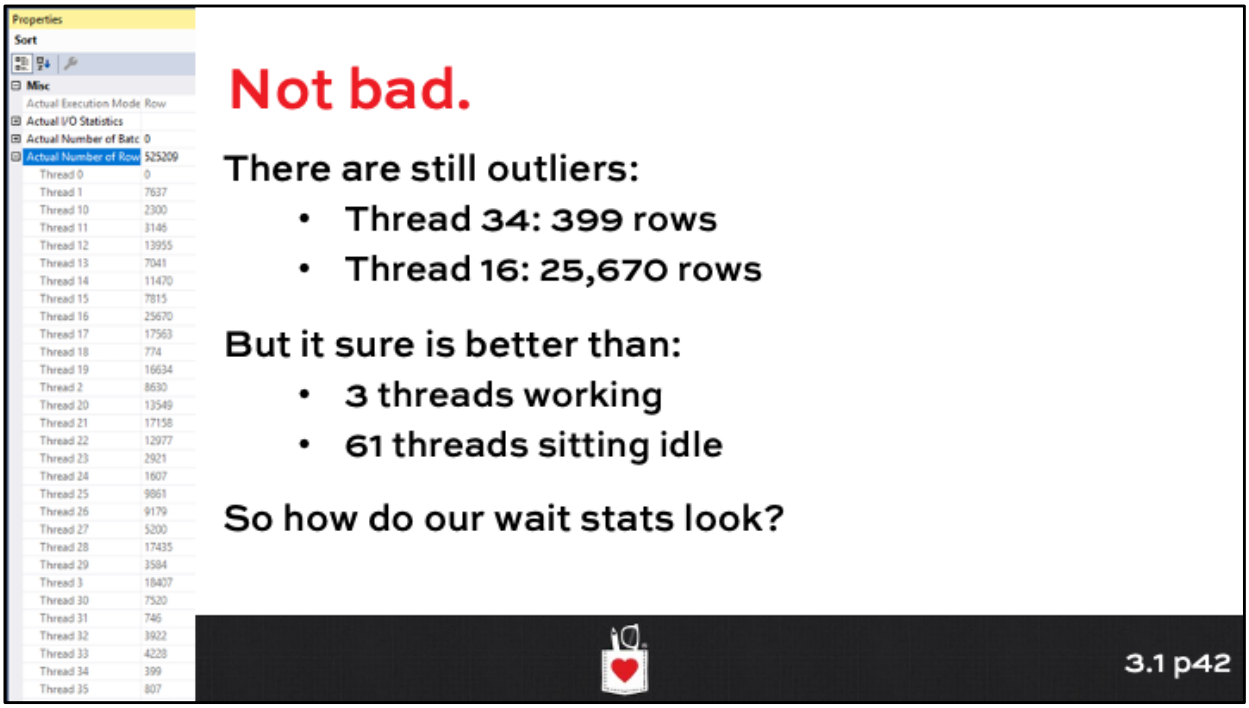


As we go upstream, balance is off

Each thread finds different numbers of Posts.

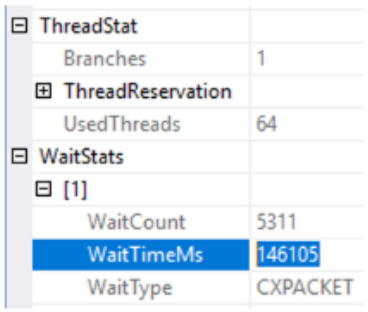
To see how bad it is, let's check the Sort properties right before we gather streams again...





Awww, dang.

The work is still unbalanced enough that we have over 2 minutes of CXPACKET waits in a 3-second query.



But remember:
we're talking about a 3-second query here.

You would never normally work this hard to tune it (and you wouldn't be throwing 64 cores at it.)

This is just a useful demo query.

Catching unbalanced parallelism

If you need to make a query go faster,

And you see a lot of operators under a
Parallelism Gather Streams operator,

Don't assume that they're all going evenly parallel.

If you want more cores to get used,
you may need to break your query into phases again.

3.1 p44

Problem #1: skewed data

3.1 p45

Ways SQL can balance the work

Most common:

- Hash: Consumers are chosen via a hash function
- Round Robin: Rows sent to consumers in order

Less common:

- Broadcast: Every row is sent to all consumers
- Demand: Row is sent to first consumer that asks
- Range: Consumers each get a unique range of rows

3.1 p46

But what happens if...

Your data skews heavily towards one value?

Your data only gets hashed on one value?

Your data just happens to all hash to a single bucket?

To find out, let's build a query
where one of the rows is way different.

3.1 p47

Jon Skeet vs 1000 of you

```
WITH hi_lo AS
(
    SELECT TOP 1 u.Id, u.DisplayName, 1 AS sort_order
    FROM dbo.Users AS u
    ORDER BY u.Reputation DESC

    UNION ALL

    SELECT TOP 1000 u.Id, u.DisplayName, 2 AS sort_order
    FROM dbo.Users AS u
    WHERE EXISTS (SELECT 1/0 FROM dbo.Posts AS p WHERE u.Id = p.OwnerUserId AND p.Score > 0)
    AND u.Reputation > 1
    ORDER BY u.Reputation ASC, u.CreationDate DESC
)
```

3.1 p48

Then after the CTE, the query goes on:

- We take data from that CTE
- We join it to other tables
  - Posts
  - Comments
  - Badges
- We aggregate some stuff
- The app tells you how long it'd take them to achieve Skeet-like status

CTE

Join

Aggregate

```
SELECT u.DisplayName,
       SUM(CONVERT(BIGINT, p.Score)) AS PostScore,
       SUM(CONVERT(BIGINT, c.Score)) AS CommentScore,
       COUNT_BIG(b.Id) AS BadgeCount
FROM hi_lo AS u
JOIN dbo.Posts AS p
ON p.OwnerUserId = u.Id
JOIN dbo.Comments AS c
ON c.UserId = u.Id
JOIN dbo.Badges AS b
ON b.UserId = u.Id
GROUP BY u.DisplayName, sort_order
ORDER BY u.sort_order
```

3.1 p49

Jon Skeet is big.

These numbers are much higher than anyone else's by far, but the real problem is that his Id – 22656 – always goes to a single thread

That leaves the other threads with nearly no work to do at all

```
SELECT COUNT(*) AS PostCount
FROM dbo.Posts AS p
WHERE p.OwnerUserId = 22656;
```

PostCount
11371

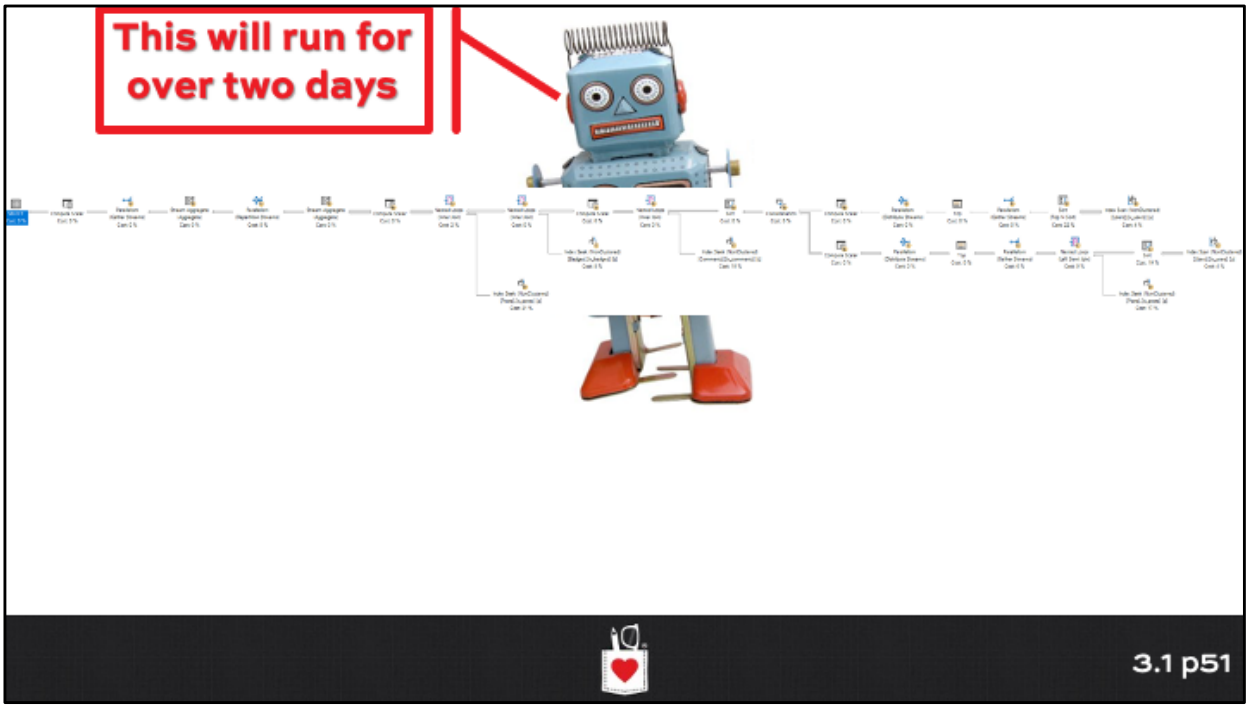
```
SELECT COUNT(*) AS CommentCount
FROM dbo.Comments AS c
WHERE c.UserId = 22656;
```

CommentCount
13425

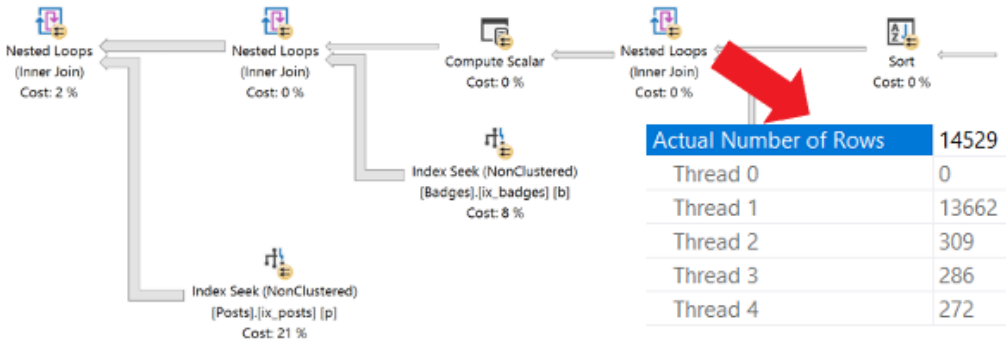
```
SELECT COUNT(*) AS BadgeCount
FROM dbo.Badges AS b
WHERE b.UserId = 22656;
```

BadgeCount
2821

3.1 p50

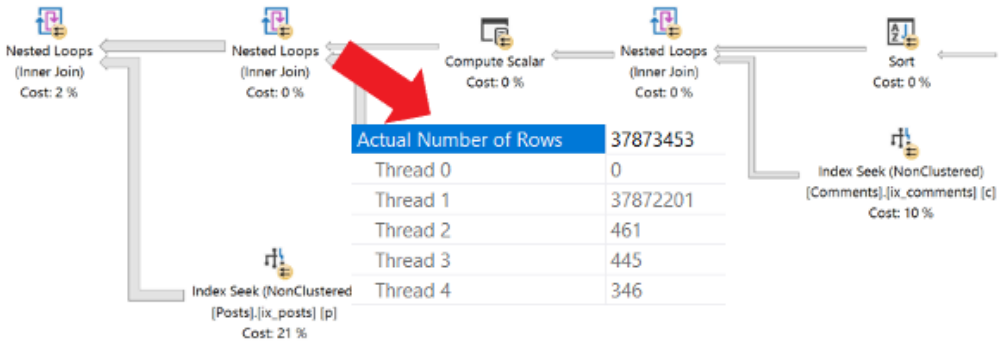


Work from right to left...



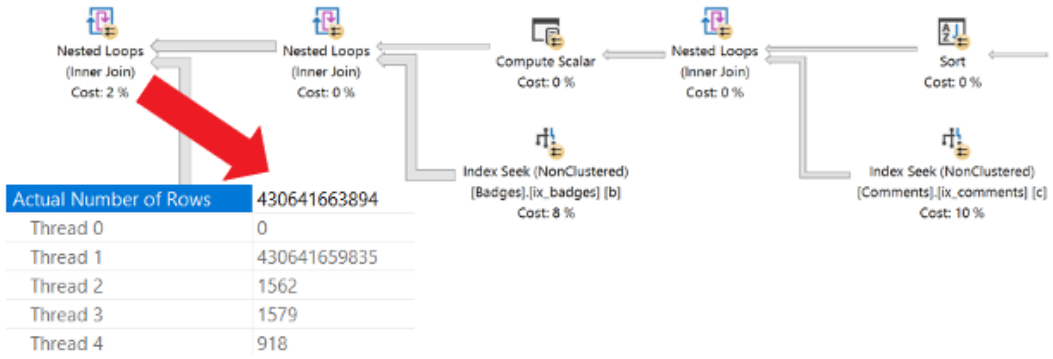
3.1 p52

And the further we go...



3.1 p53

The crazier it gets...



Actual Number of Rows	430641663894
Thread 0	0
Thread 1	430641659835
Thread 2	1562
Thread 3	1579
Thread 4	918

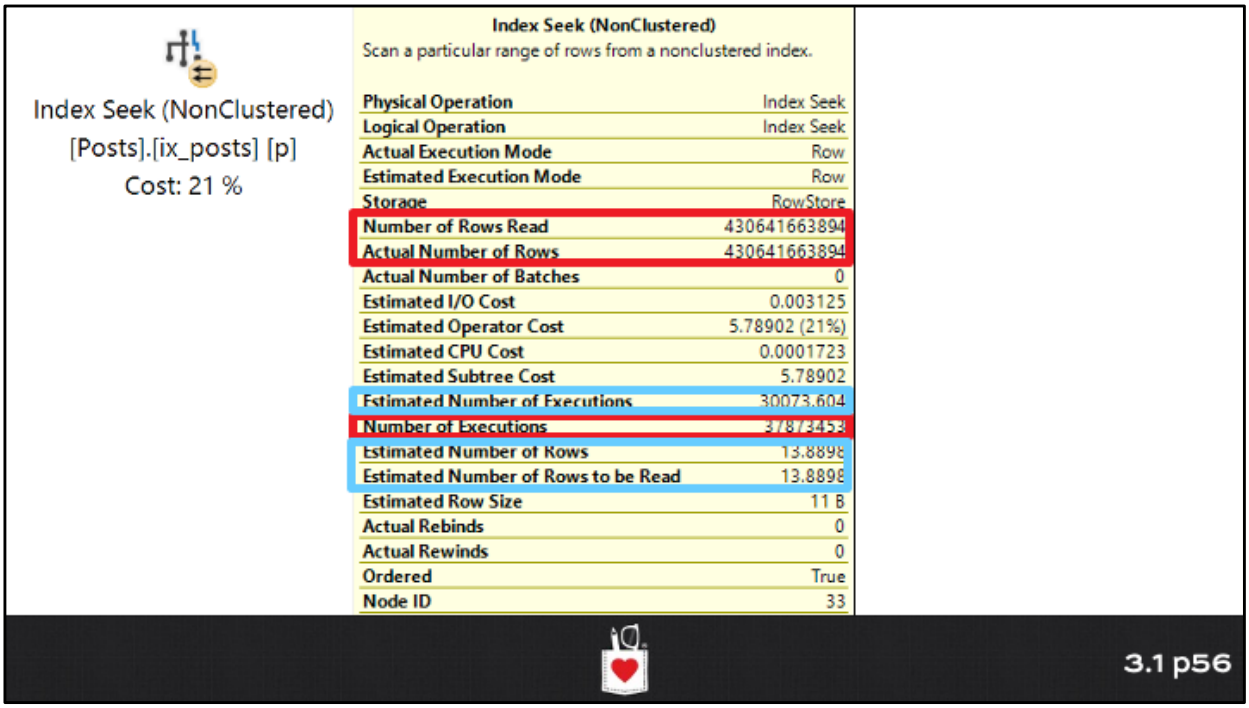
https://docs.microsoft.com/en-us/archive/blogs/craigfr/parallel-nested-loops-join

3.1 p54

Comma chameleon

Actual Number of Rows	430,641,663,894
Thread 0	0
Thread 1	430641659835
Thread 2	1562
Thread 3	1579
Thread 4	918

3.1 p55



sp_BlitzFirst warns you

But only on newer versions of SQL Server

Priority	FindingsGroup	Finding	URL	Details
0	sp_BlitzFirst 2018-10-0...	From Your Community Volunteers	http://FirstResponderKit.org/	<?ClickToSeeDetails -- We hope you found this tool useful. -- ?>
100	Query Performance	Queries with 10000x cardinality misestimations	N/A	<?ClickToSeeDetails -- The query on SPID 52 has been running for 11 seconds, with a large cardinality misestimate. -- ?>
100	Query Performance	Queries with 10000x skewed parallelism	N/A	<?ClickToSeeDetails -- The query on SPID 52 has been running for 11 seconds, with a parallel threads doing uneven work. -- ?>

Details from Live Query Stats/Lightweight Profiling

- https://docs.microsoft.com/en-us/sql/relational-databases/performance/live-query-statistics
- https://www.brentozar.com/archive/2018/10/the-new-lightweight-query-plan-profile-hint/

3.1 p57

Hints help reshape the plan

MERGE JOIN	2.5 minutes
HASH JOIN	2 minutes
MAXDOP 1	1 minute
Break the query up into 1 for Jon, 1 for others	Nearly instantly

3.1 p58

Serial killer

Don't go around throwing hints on all of your parallel queries – you have to observe skew first

Even then, this should be a last resort, and you still:

- Need to be sure it's not just a certain set of parameters that benefits
- The query wouldn't be better off with some tuning
- Stats aren't out of date, etc.

3.1 p59

Different DOPs?

Sometimes changing DOP to different numbers can change how different values distribute to threads

Usually changing from even > odd DOP

This can help if:

- Many repetitive values distribute to one thread

This won't help if:

- A single value dominates the results

https://orderbyselectnull.com/2017/09/26/a-serial-parallel-query/

Problem #2: parallelism spills

3.1 p61

Coaxing

It's hard to get the optimizer to mess up this badly

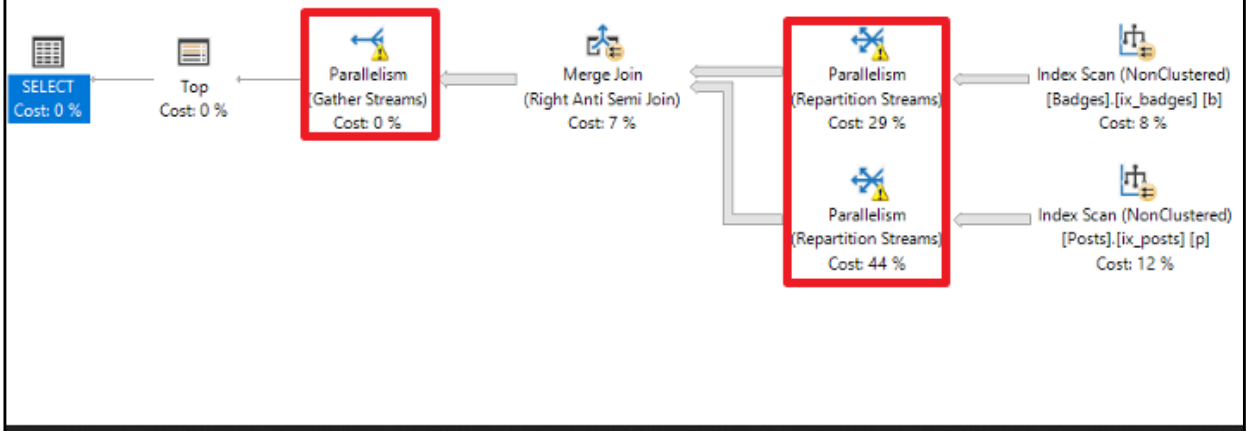
Rather than come up with a big, contrived, confusing query, I'm going to add hints to force bad behavior

```
SELECT      TOP 10 p.OwnerUserId
FROM        dbo.Posts AS p
WHERE       NOT EXISTS (   SELECT   *
                           FROM     dbo.Badges AS b
                           WHERE    p.OwnerUserId = b.UserId )
ORDER BY    p.OwnerUserId
OPTION(MAXDOP 4, USE HINT('ENABLE_PARALLEL_PLAN_PREFERENCE'), MERGE JOIN, HASH GROUP, RECOMPILE)
```

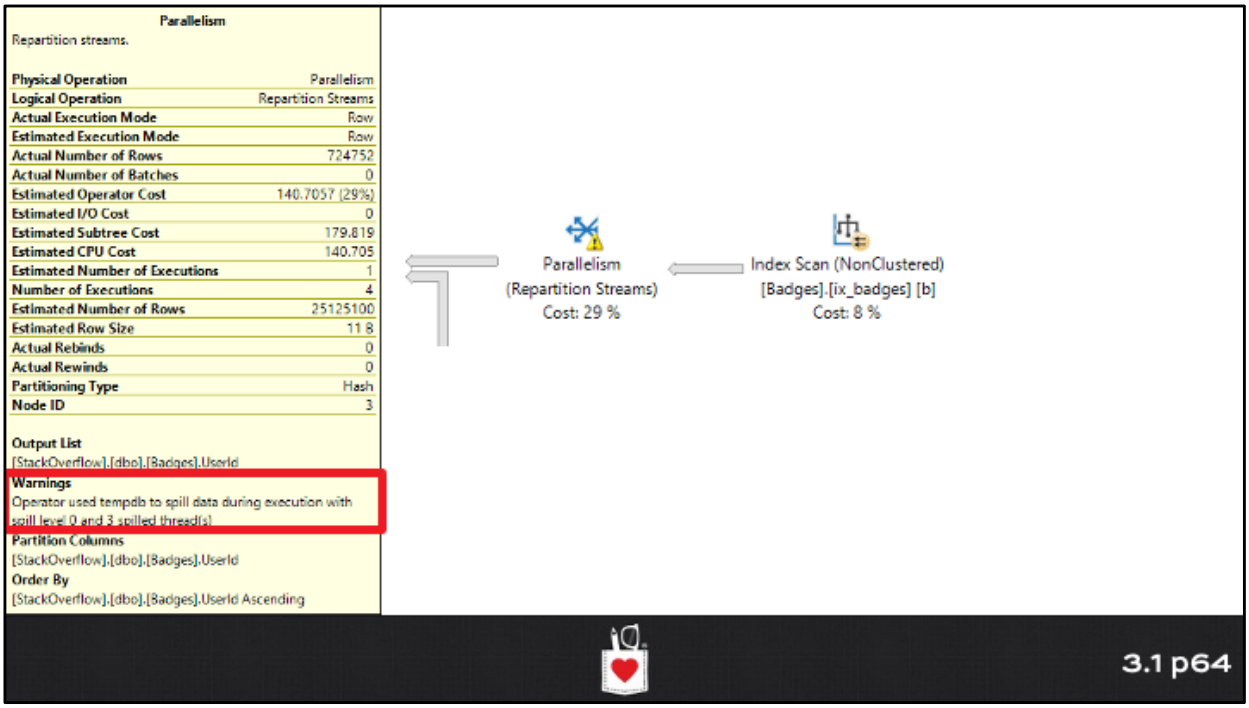
3.1 p62

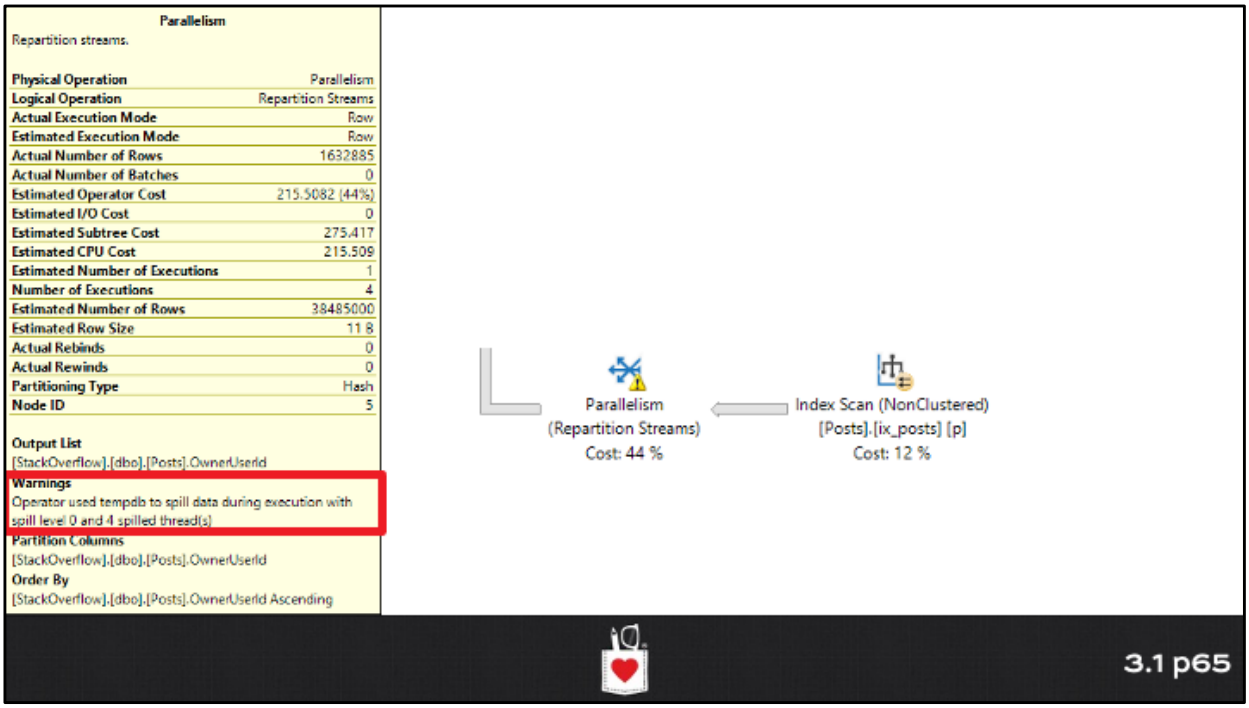
It runs for 8 minutes

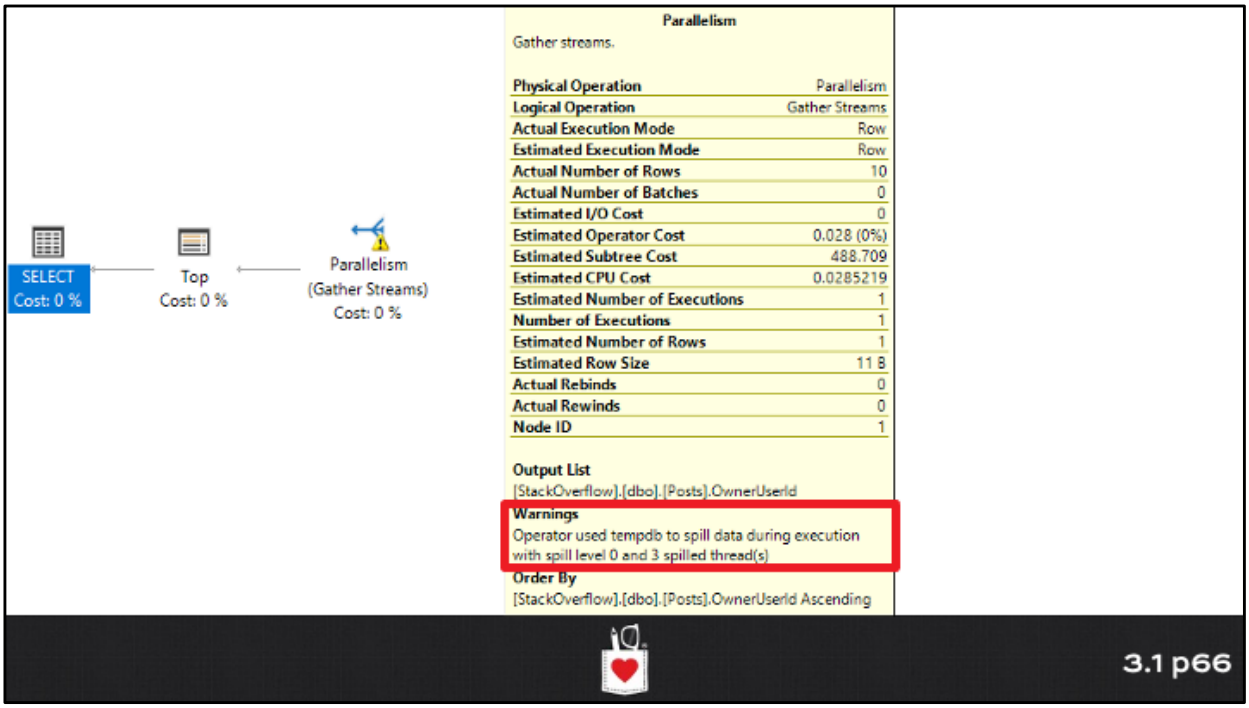
And has spills in unusual places:



3.1 p63







SQL Server tracks spills, but...

For normal tempdb spills, this works great

For exchange spills, not so much:

Min Spills	Max Spills	Total Spills
24	24	24

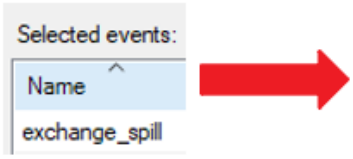
- sp_BlitzCache @SortOrder = 'spills'

Problems here:

- They're deceptively low numbers for the problems they cause in queries
- There's no way to differentiate normal tempdb spills from exchange spills

3.1 p67

You have to use XE to catch it



But the robots aren't fixing this for you, either.

name	timestamp	opcode	query_operation...	sql_text		thread_id	timestamp (UTC)	workdesc_physi...
exchange_spill	2018-07-20 14:20:34.5512371	Begin	1	SELECT	TOP...	1	2018-07-20 18:2...	0
exchange_spill	2018-07-20 14:20:42.0348718	Begin	5	SELECT	TOP...	4	2018-07-20 18:2...	0
exchange_spill	2018-07-20 14:20:47.0768691	Begin	5	SELECT	TOP...	4	2018-07-20 18:2...	0
exchange_spill	2018-07-20 14:20:47.4010734	End	5	SELECT	TOP...	4	2018-07-20 18:2...	15
exchange_spill	2018-07-20 14:20:47.4518097	End	5	SELECT	TOP...	4	2018-07-20 18:2...	16
exchange_spill	2018-07-20 14:21:03.3521992	Begin	5	SELECT	TOP...	2	2018-07-20 18:2...	0
exchange_spill	2018-07-20 14:21:08.3757635	Begin	3	SELECT	TOP...	1	2018-07-20 18:2...	0
exchange_spill	2018-07-20 14:21:12.1391992	Begin	5	SELECT	TOP...	2	2018-07-20 18:2...	0
exchange_spill	2018-07-20 14:21:14.6545044	Begin	3	SELECT	TOP...	1	2018-07-20 18:2...	0
exchange_spill	2018-07-20 14:21:15.9205639	Begin	1	SELECT	TOP...	3	2018-07-20 18:2...	0
exchange_spill	2018-07-20 14:21:20.2520655	Begin	5	SELECT	TOP...	3	2018-07-20 18:2...	0
exchange_spill	2018-07-20 14:21:31.2320963	Begin	5	SELECT	TOP...	3	2018-07-20 18:2...	0
exchange_spill	2018-07-20 14:21:37.5133691	Begin	3	SELECT	TOP...	4	2018-07-20 18:2...	0
exchange_spill	2018-07-20 14:21:42.5133622	Begin	3	SELECT	TOP...	4	2018-07-20 18:2...	0
exchange_spill	2018-07-20 14:21:48.7788670	Begin	5	SELECT	TOP...	1	2018-07-20 18:2...	0
exchange_spill	2018-07-20 14:21:53.7789698	Begin	5	SELECT	TOP...	1	2018-07-20 18:2...	0
exchange_spill	2018-07-20 14:22:10.0591886	Begin	5	SELECT	TOP...	4	2018-07-20 18:2...	0
exchange_spill	2018-07-20 14:22:15.0591936	Begin	5	SELECT	TOP...	4	2018-07-20 18:2...	0
exchange_spill	2018-07-20 14:22:41.3404692	Begin	3	SELECT	TOP...	2	2018-07-20 18:2...	0
exchange_spill	2018-07-20 14:22:46.3601007	Begin	3	SELECT	TOP...	2	2018-07-20 18:2...	0
exchange_spill	2018-07-20 14:23:57.7134693	Begin	3	SELECT	TOP...	3	2018-07-20 18:2...	0
exchange_spill	2018-07-20 14:24:02.7134716	Begin	3	SELECT	TOP...	3	2018-07-20 18:2...	0

3.1 p68

Fixing it

This can be genuinely difficult, because you have to

- Recognize a slow query
- That it's parallel
- That it's spilling
- That the spills are from the exchanges
  - Sorts and Hashes spill too
  - The warning aren't in cached/estimated plans

You may need to resort to query rewrites, hints, plan guides, or pinning a different plan via Query Store to resolve this long term

3.1 p69

Problem #3: single-threaded spools

3.1 p70

Spools aren't bad

There are lots of different Spools in execution plans

- Table spools
- Row count spools
- Window spools
- Index spools

Spools can be lazy or eager

The kind of spools we care about in parallel plans are
Eager Index Spools

3.1 p71

What's in a spool?

Spools are stored in tempdb

- They're result set caches that get destroyed when a query finishes, and they can't be shared

Lazy spools

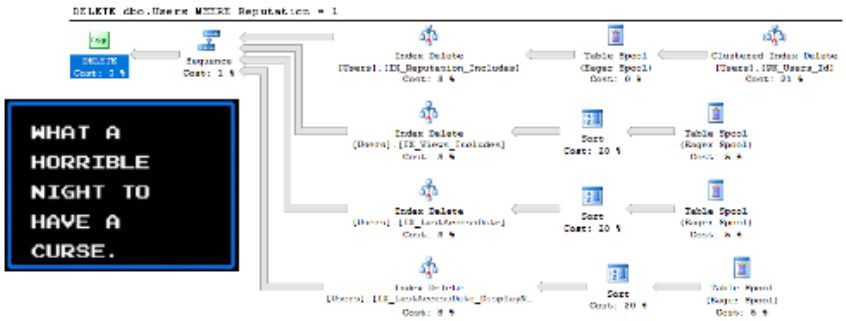
- Non-blocking, only fetch data for a particular value when requested

Eager spools

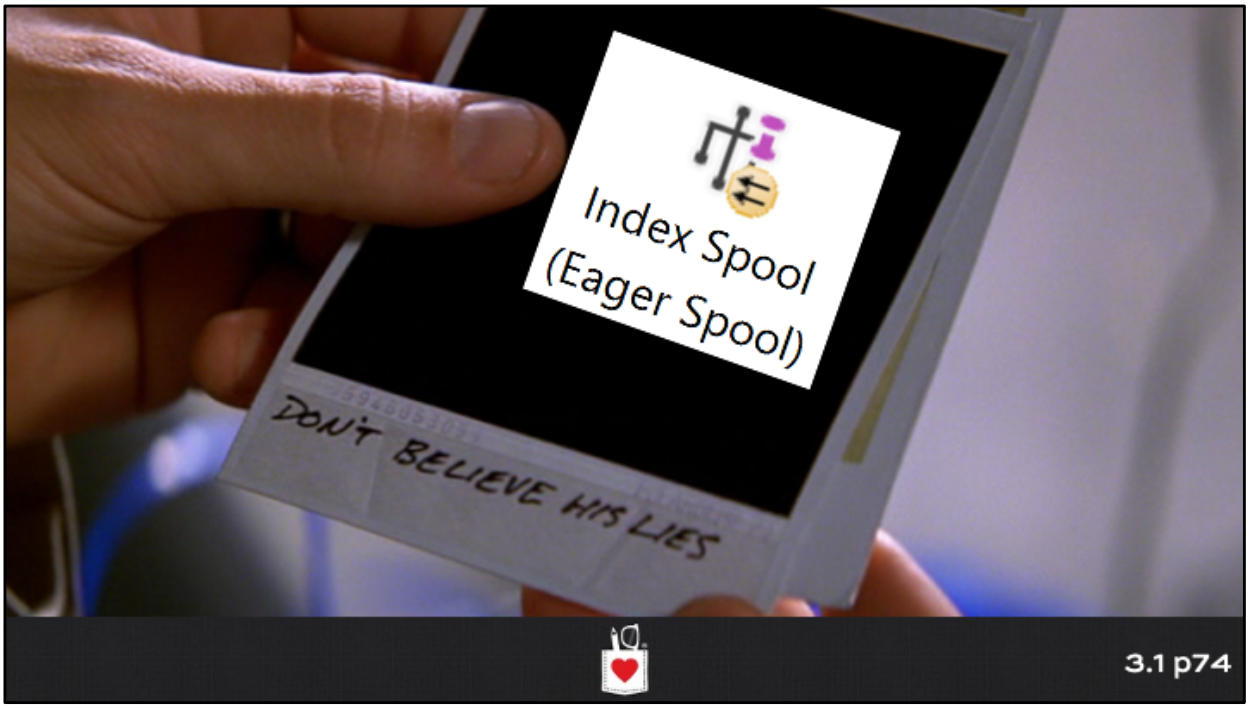
- Blocking, fetch all data needed at once

Spools in modification queries: OK

These are for Halloween Protection, and you can't really do anything to fix these except tune your indexes down to just the ones you need.



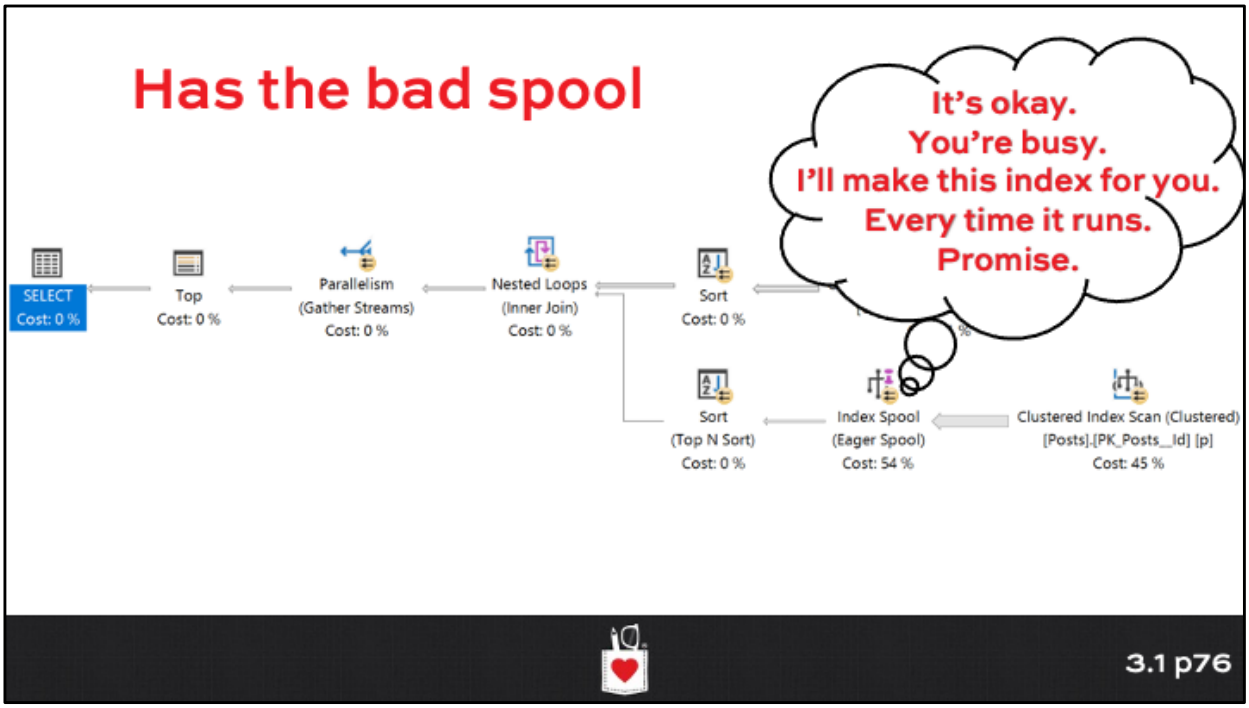
3.1 p73



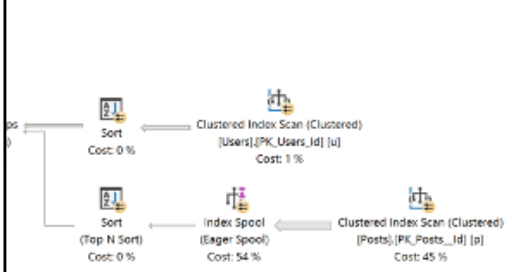
Test query

```
SELECT TOP 101 u.DisplayName, u.Reputation, ca.*
FROM dbo.Users AS u
CROSS APPLY (
    SELECT TOP 1 p.Id, p.Score, p.Title
    FROM dbo.Posts AS p
    WHERE p.OwnerUserId = u.Id
    AND p.PostTypeId = 1
    ORDER BY p.Score DESC
            ) AS ca
WHERE u.Reputation >= 100000
ORDER BY u.Reputation DESC;
```

3.1 p75



SAYS it's parallel, but...

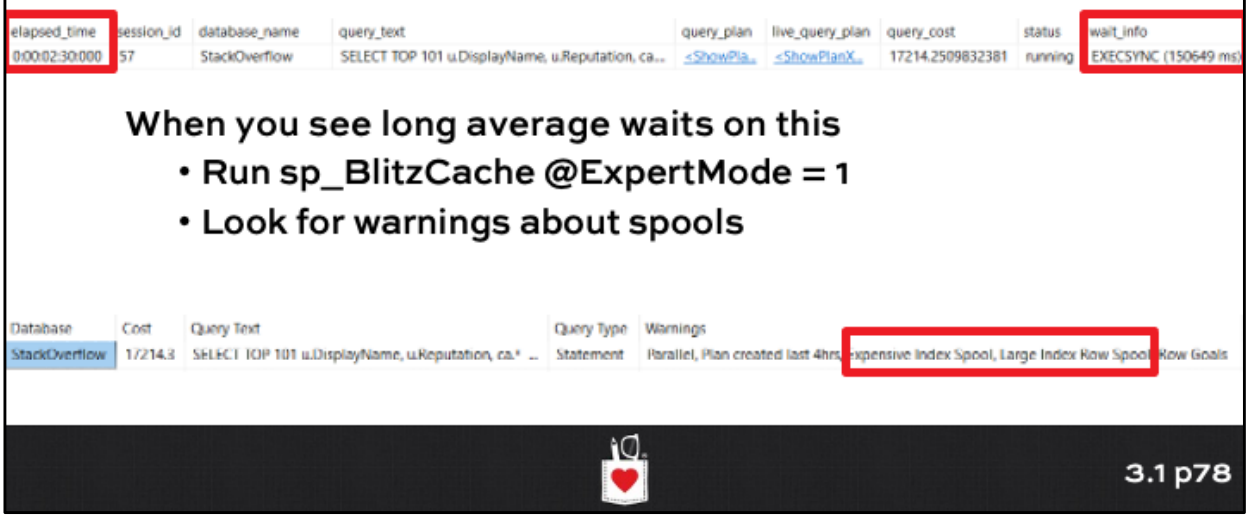


Clustered Index Scan (Clustered)

Misc	
Actual Execution Mode	Row
Actual I/O Statistics	
Actual Number of Batche	0
Actual Number of Rows	38485046
Thread 0	0
Thread 1	0
Thread 2	38485046
Thread 3	0
Thread 4	0

3.1 p77

EXECSYNC waits



elapsed_time	session_id	database_name	query_text	query_plan	live_query_plan	query_cost	status	wait_info
0:00:02:30:000	57	StackOverflow	SELECT TOP 101 u.DisplayName, u.Reputation, ca...	<ShowPla...	<ShowPlanX...	17214.2509832381	running	EXECSYNC (150649 ms)

When you see long average waits on this

- Run sp_BlitzCache @ExpertMode = 1
- Look for warnings about spools

Database	Cost	Query Text	Query Type	Warnings
StackOverflow	17214.3	SELECT TOP 101 u.DisplayName, u.Reputation, ca.* ...	Statement	Parallel, Plan created last 4hrs, Expensive Index Spool, Large Index Row Spool, Row Goals

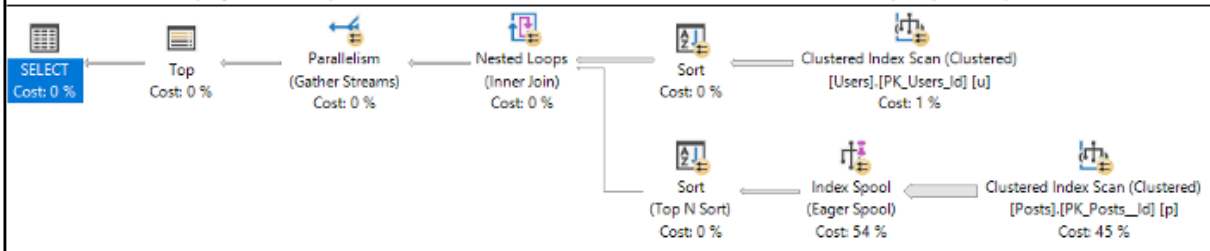
3.1 p78

No missing index request

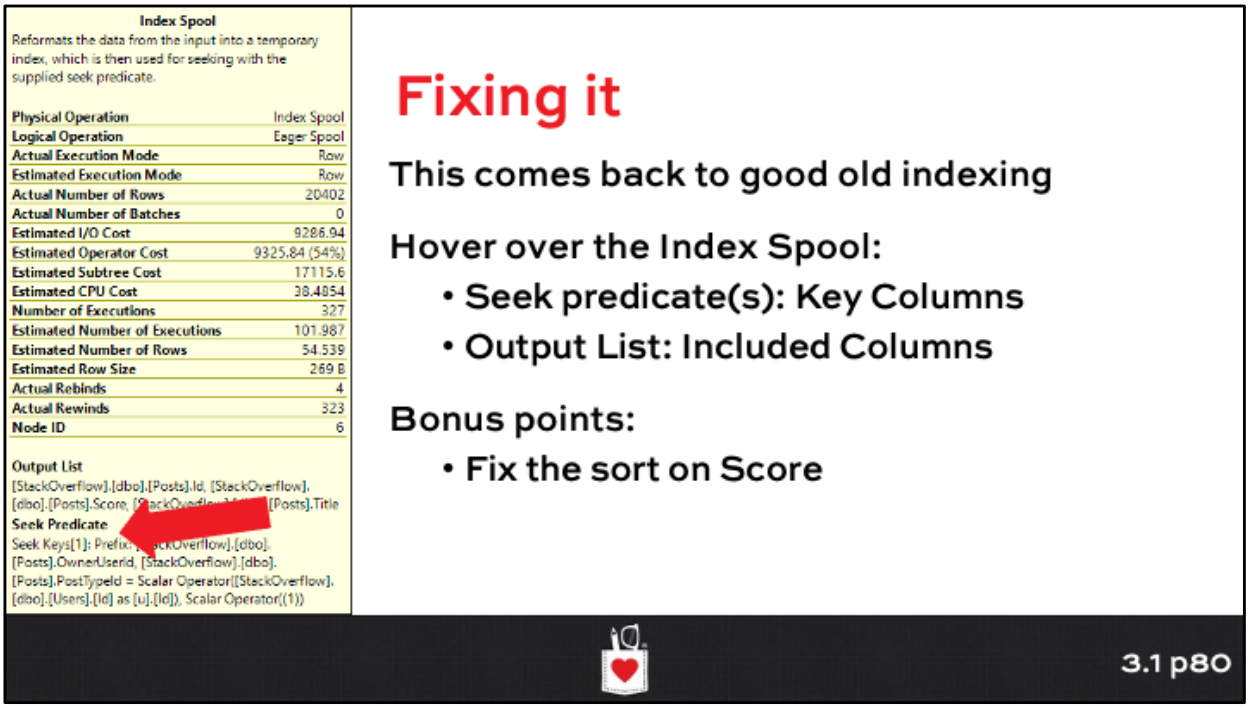
This is SQL Server at its most passive aggressive

- "Oh, no, I picked those community college brochures up for someone else... Why? Interested?"

Query 1: Query cost (relative to the batch): 100%
SELECT TOP 101 u.DisplayName, u.Reputation, ca.* FROM dbo.Users AS u CROSS APPLY (SELECT TOP 1 p.Id, p.Score, p.Title FROM dbo.Posts AS p



3.1 p79



Recap

3.1 p81

Start with the basics.

Normally, SQL Server does a good-enough job of balancing work across threads. Not great. Just OK.

But look out for:

- A single Parallelism Gather Streams operator with a LOT of operators underneath
- Spills on parallelism operators
- Index Spool – Eager Spool

Fixing it = index tuning, query rewrites.