



BRENT OZAR
UNLIMITED®

How Parallelism Balances Work Across Threads

3.1 p1

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:

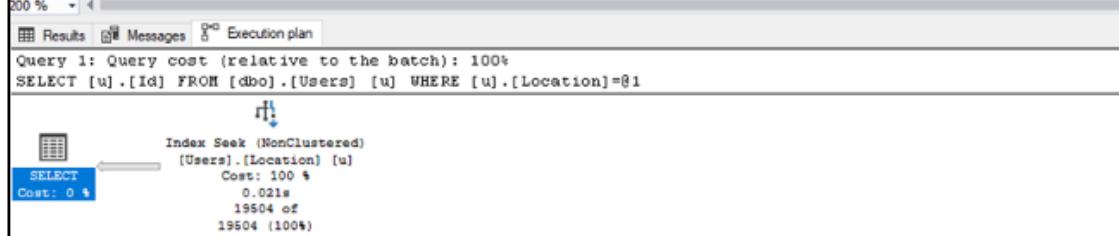
- Unevenly skewed data
- Spills on parallelism operators
- Spools: skewed all to one thread



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
```

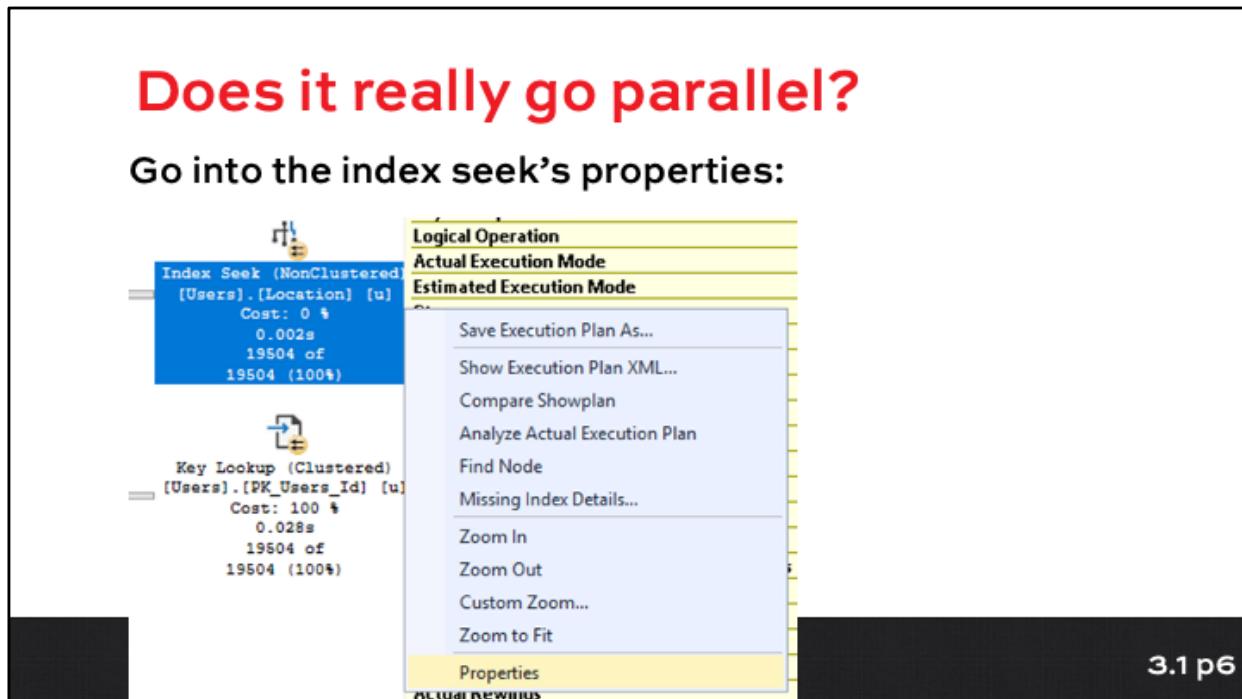


Add a key lookup, and it does.



Does it really go parallel?

Go into the index seek's properties:



3.1 p6

Properties	
Index Seek (NonClustered)	
<input type="checkbox"/>	Minc
Actual Execution Mode	Row
Actual I/O Statistics	
Actual Log Logical Reads	0
Actual Log Physical Reads	0
Actual Log Read Aheads	0
Actual Logical Reads	161
Thread 0	8
Thread 1	161
Thread 10	0
Thread 11	0
Thread 12	0
Thread 13	0
Thread 14	0
Thread 15	0
Thread 16	0
Thread 17	0
Thread 18	0
Thread 19	0
Thread 2	0
Thread 20	0
Thread 21	0
Thread 22	0
Thread 23	0
Thread 24	0
Thread 25	0
Thread 26	0
Thread 27	0
Thread 28	0
Thread 29	0
Thread 3	0
Thread 30	0
Thread 31	0
Thread 32	0
Thread 33	0
Thread 34	0
Thread 35	0
Thread 36	0
Thread 37	0
Thread 38	0

That's not really parallel.

Thread 0 = coordinating thread

- **Reads a little to figure out how to pass out work to the other threads**
- **He's not actually handling rows though**

Thread 1 = doing some work

The rest = bored out of their gourds



3.1 p7

Properties	
Index Seek (NonClustered)	
Misc	
Actual Execution Mode	Row
Actual I/O Statistics	
Actual Log Logical Reads	0
Actual Log Physical Reads	0
Actual Log Read Aheads	0
Actual Logical Reads	161
Thread 0	8
Thread 1	0
Thread 10	0
Thread 11	0
Thread 12	0
Thread 13	0
Thread 14	0
Thread 15	0
Thread 16	0
Thread 17	0
Thread 18	0
Thread 19	0
Thread 2	0
Thread 20	0
Thread 21	0
Thread 22	0
Thread 23	0
Thread 24	0
Thread 25	0
Thread 26	0
Thread 27	0
Thread 28	0
Thread 29	0
Thread 3	25
Thread 30	0
Thread 31	0
Thread 32	0
Thread 33	0
Thread 34	0
Thread 35	0
Thread 36	0
Thread 37	0

The working thread changes.

Run the query again, and you see a different thread getting the raw deal.

Thread 3 isn't the only one working here.

Scroll down...



3.1 p8

Properties	
Index Seek (NonClustered)	
Thread 29	0
Thread 3	68
Thread 30	0
Thread 31	0
Thread 32	0
Thread 33	0
Thread 34	0
Thread 35	0
Thread 36	0
Thread 37	0
Thread 38	0
Thread 39	0
Thread 4	0
Thread 40	0
Thread 41	0
Thread 42	0
Thread 43	0
Thread 44	0
Thread 45	0
Thread 46	0
Thread 47	0
Thread 48	0
Thread 49	0
Thread 5	65
Thread 50	0
Thread 51	0
Thread 52	0
Thread 53	0
Thread 54	0
Thread 55	0
Thread 56	0
Thread 57	0
Thread 58	0
Thread 59	0
Thread 6	20
Thread 60	0
Thread 61	0
Thread 62	0
Thread 63	0
Thread 64	0

Just like your office

Only 3 workers are really doing anything:
in this case, 3, 5, and 6.

The rest are standing around smoking.

The good news is that they're not actually
tying up your CPUs.

The bad news: they are tying up worker
threads and causing CXPACKET waits.



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.

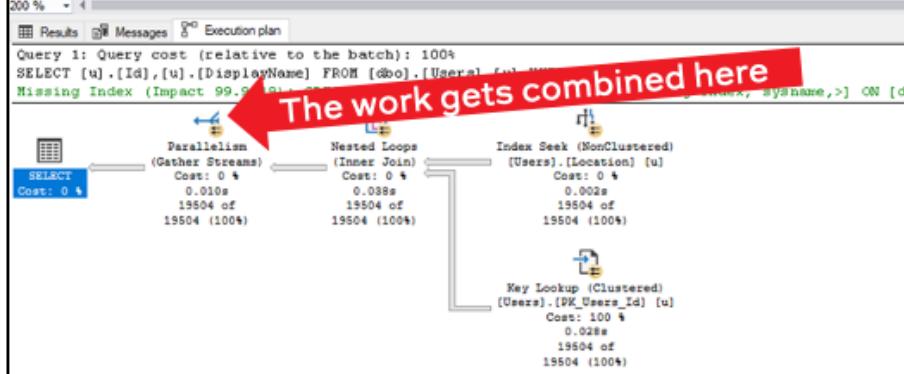
Why 155ms? Well, the whole plan is unbalanced.

WaitCount	65
WaitTimeMs	155
WaitType	CXPACKET



3.1 p10

The rows start out unbalanced.

```
38 |  SELECT u.Id, u.DisplayName  
39 |  FROM dbo.Users u  
40 |  WHERE u.Location = N'London, United Kingdom';  
41 |  


The execution plan shows a query cost of 100% relative to the batch. It starts with a 'SELECT' node (Cost: 0) which branches into a 'Parallelism' node (Gather Streams) and a 'Nested Loops' node (Inner Join). The 'Parallelism' node has two parallel streams. The top stream leads to an 'Index Seek (NonClustered)' node for the 'Users'.[Location] column (Cost: 0), which then leads to a 'Key Lookup (Clustered)' node for the 'Users].[PK_Users_Id]' column (Cost: 100%). The bottom stream of the 'Parallelism' node leads to a 'Nested Loops' node (Inner Join) with a cost of 0. The 'Nested Loops' node has two inputs: one from the 'Parallelism' node and one from the 'Index Seek' node. Both inputs have a cost of 0.4. The output of the 'Nested Loops' node has a cost of 0.038s and 19504 of rows. The final 'SELECT' node (Cost: 0) has a total cost of 0.002s and 19504 of rows.


```

The work gets combined here

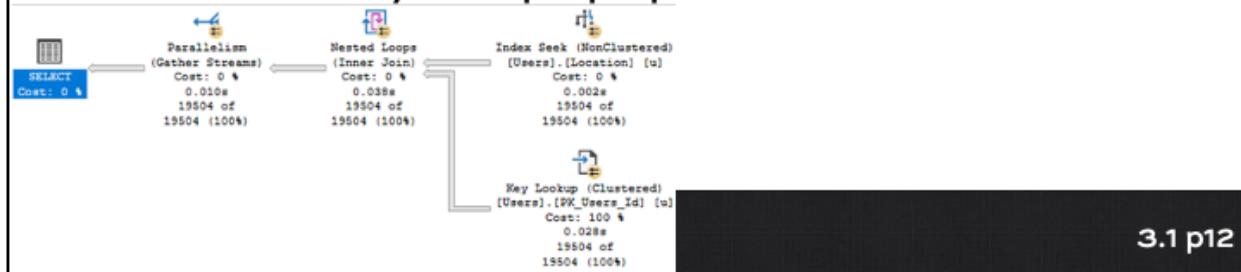
3.1 p11

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...



Thread 28	0
Thread 29	0
Thread 3	33889 ↙
Thread 30	0
Thread 31	0
Thread 32	0
Thread 33	0
Thread 34	0
Thread 35	0
Thread 36	0
Thread 37	0
Thread 38	0
Thread 39	0
Thread 4	0
Thread 40	0
Thread 41	0
Thread 42	0
Thread 43	0
Thread 44	0
Thread 45	0
Thread 46	0
Thread 47	0
Thread 48	0
Thread 49	0
Thread 5	32034 ↙
Thread 50	0
Thread 51	0
Thread 52	0
Thread 53	0
Thread 54	0
Thread 55	0
Thread 56	0
Thread 57	0
Thread 58	0
Thread 59	0
Thread 6	8358 ↙
...	...

Key lookup logical reads

Only 3 threads are doing the key lookups.



3.1 p13

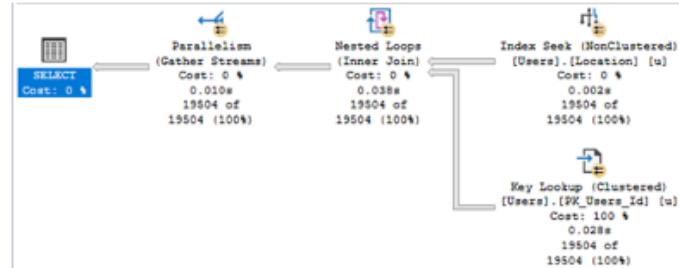
In this plan...that's fine.

It runs in 10 ms.

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

Who cares? It's fine.

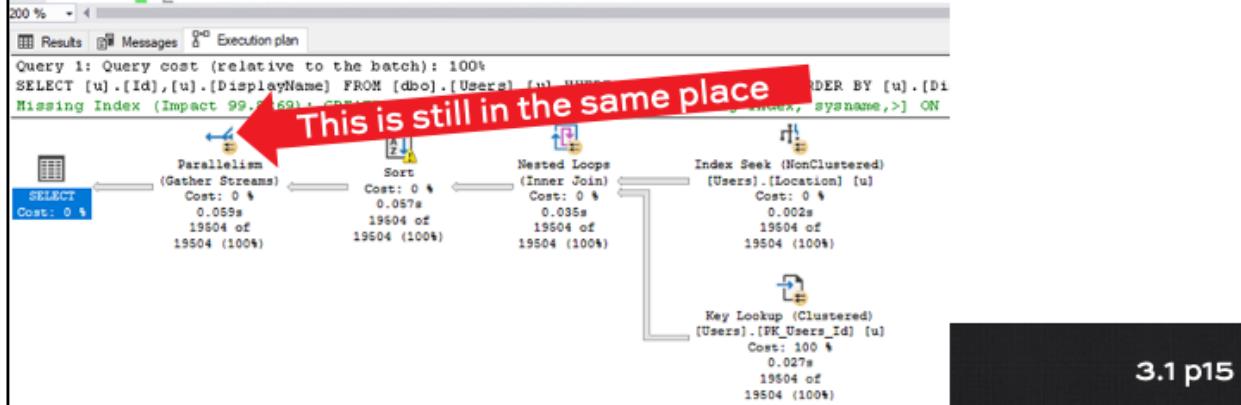
Let's add more work to the query.



3.1 p14

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;
```



Properties	
Sort	
Misc	
Actual Execution Mode	Row
Actual I/O Statistics	
Actual Number of Batches	0
Actual Number of Rows for All Execut	19504
Thread 0	0
Thread 1	2189
Thread 10	0
Thread 11	0
Thread 12	0
Thread 13	0
Thread 14	0
Thread 15	0
Thread 16	0
Thread 17	0
Thread 18	0
Thread 19	0
Thread 2	8856
Thread 20	0
Thread 21	0
Thread 22	0
Thread 23	0
Thread 24	0
Thread 25	0
Thread 26	0
Thread 27	0
Thread 28	0
Thread 29	0
Thread 3	0
Thread 30	0
Thread 31	0
Thread 32	0
Thread 33	0
Thread 34	0
Thread 35	0
Thread 36	0
Thread 37	0
Thread 38	0
Thread 39	0
Thread 4	0

So the sort isn't balanced either.

There are only 3 threads doing the sorting.

Yes, you might notice the thread numbers are changing.

That's because I ran the query again when I added the ORDER BY.

Each time I run it, the data moves around.



3.1 p16

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.

The screenshot shows a SQL query window and its corresponding execution plan. The query is:

```
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);
```

A red arrow points to the 'OPTION(MAXDOP 4)' clause with the label 'New'. Below the query, the execution plan is displayed. It shows a 'Parallelism (Gather Streams)' operator at the top, followed by a 'Sort' operator, a 'Nested Loops (Inner Join)' operator, and finally an 'Index Seek (NonClustered)' operator for the 'Location' index on the 'Users' table. A 'Key Lookup (Clustered)' operator is shown at the bottom, indicating a seek operation on the clustered index. The execution plan details the cost and execution time for each step.

Execution plan details:

- Parallelism (Gather Streams)**: Cost: 0 \$, 0.052s, 19504 of, 19504 (100%)
- Sort**: Cost: 1 \$, 0.049s, 19504 of, 19504 (100%)
- Nested Loops (Inner Join)**: Cost: 0 \$, 0.036s, 19504 of, 19504 (100%)
- Index Seek (NonClustered) [Users].[Location] (u)**: Cost: 0 \$, 0.002s, 19504 of, 19504 (100%)
- Key Lookup (Clustered) [Users].[PK_Users_Id] (u)**: Cost: 99 \$, 0.028s, 19504 of, 19504 (100%)

3.1 p18

Properties	
Index Seek (NonClustered)	
<input type="checkbox"/> Misc	
Actual Execution Mode	Row
<input type="checkbox"/> Actual I/O Statistics	
<input type="checkbox"/> Actual Lob Logical Reads	0
<input type="checkbox"/> Actual Lob Physical Reads	0
<input type="checkbox"/> Actual Lob Read Aheads	0
<input checked="" type="checkbox"/> Actual Logical Reads	161
Thread 0	8
Thread 1	20
Thread 2	0
Thread 3	68
Thread 4	65
<input type="checkbox"/> Actual Physical Reads	0

And the sort is more balanced

Properties

Sort

Misc

	Row
Actual Execution Mode	0
Actual I/O Statistics	0
Actual Number of Batches	0
Actual Number of Rows for All Execut	19504
Thread 0	0
Thread 1	2199
Thread 2	0
Thread 3	8896
Thread 4	8409

Actual Duration

3.1 p20

Although again, still not great.

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

There is way less CXPACKET.

ThreadStat	
Branches	1
ThreadReservation	
UsedThreads	4
WaitStats	
[1]	
WaitCount	68
WaitTimeMs	288
WaitType	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:

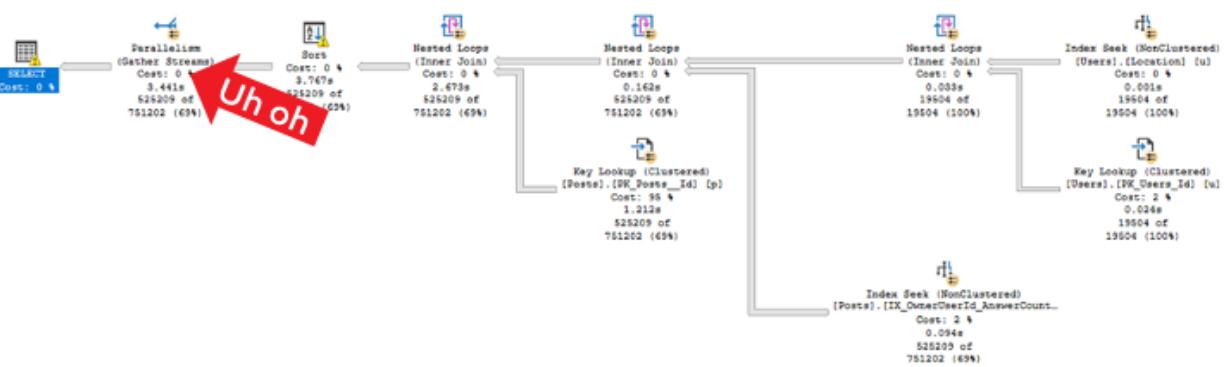
```
CREATE INDEX OwnerUserId ON dbo.Posts(OwnerUserId);
GO
SELECT u.Id, u.DisplayName, p.Id AS PostId, p.Title, p.Score
FROM dbo.Users u
INNER JOIN dbo.Posts p ON u.Id = p.OwnerUserId
WHERE u.Location = N'London, United Kingdom'
ORDER BY p.Score DESC, u.DisplayName;
```



3.1 p22

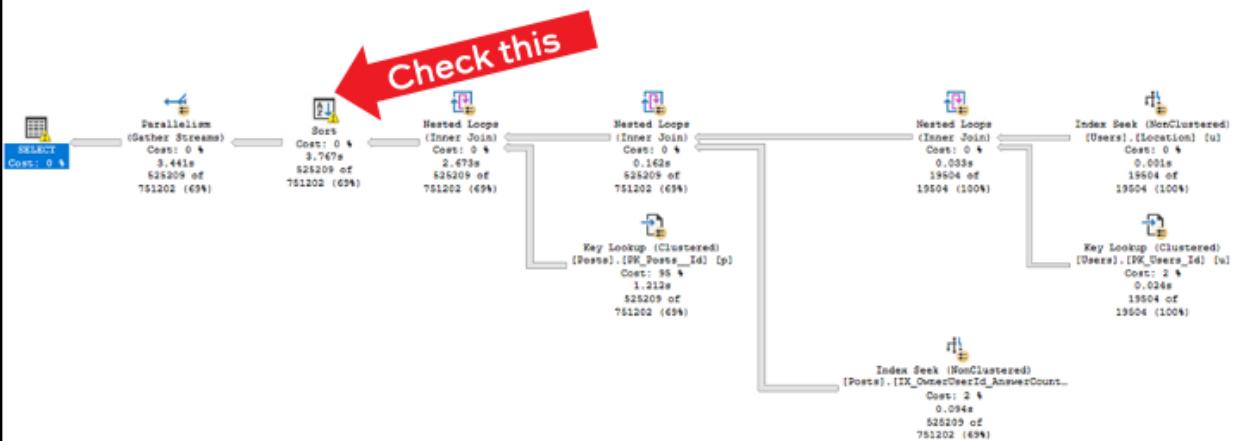
The parallelism icon is still at left

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:



Properties	
Sort	Row
Actual Execution Mode	Row
Actual I/O Statistics	
Actual Number of Batches	0
Actual Number of Rows for All Execut	525209
Thread 0	0
Thread 1	0
Thread 10	452598
Thread 11	0
Thread 12	4009
Thread 13	0
Thread 14	0
Thread 15	0
Thread 16	0
Thread 17	0
Thread 18	0
Thread 19	0
Thread 2	0
Thread 20	0
Thread 21	0
Thread 22	0
Thread 23	0
Thread 24	0
Thread 25	0
Thread 26	0
Thread 27	0
Thread 28	0
Thread 29	0
Thread 3	0
Thread 30	0
Thread 31	0
Thread 32	0
Thread 33	0
Thread 34	0
Thread 35	0
Thread 36	0
Thread 37	0
Thread 38	0
Thread 39	0
Thread 4	606602

Getting messy

Thread 10: 452,598 rows

Thread 12: 4,009 rows

Thread 4: 68,602 rows

ThreadStat	
Branches	1
ThreadReservation	
UsedThreads	64
WaitStats	
[1]	
WaitCount	5659
WaitTimeMs	174165
WaitType	CXPACKET

Query duration: 3 seconds

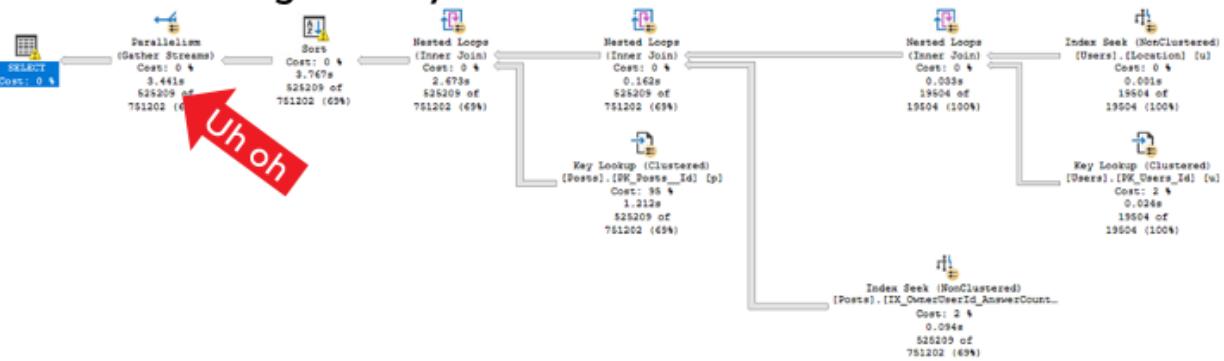
CXPACKET waits: 174 seconds!



3.1 p25

The problem: wide areas of imbalanced 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?



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



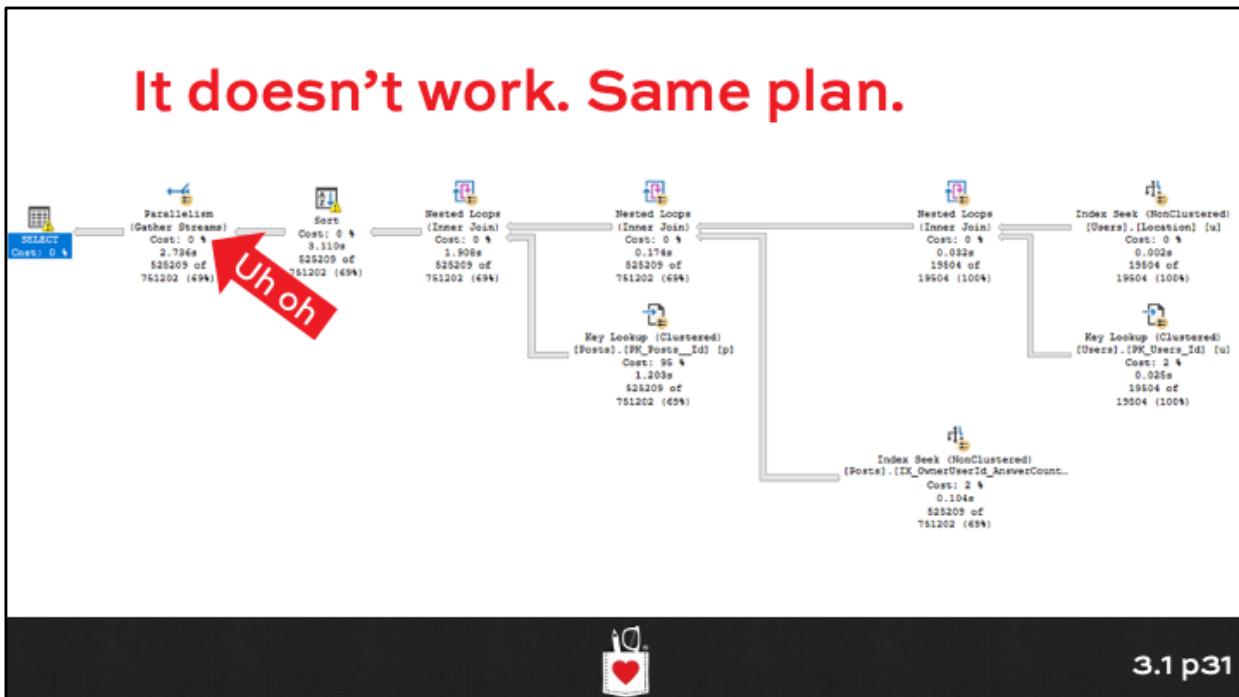
3.1 p29

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.



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...



3.1 p32

Trick to force the CTE to run first

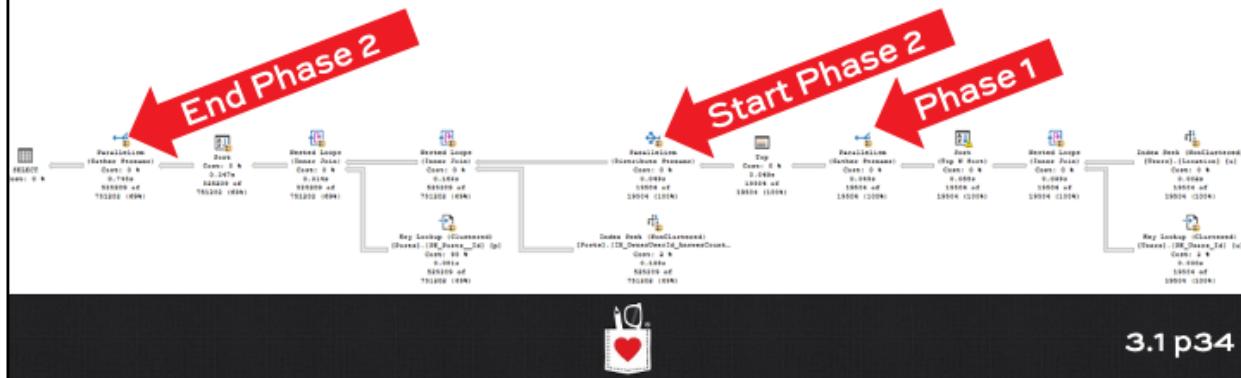
THIS IS NOT A GOOD IDEA, but you can do it:

```
WITH MatchingUsers AS ( Uh oh
    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

Properties	
Parallelism	
Misc	
Actual Execution Mode	Row
Actual Number of Batch	0
Actual Number of Row	19504
Thread 0	0
Thread 1	305
Thread 10	305
Thread 11	305
Thread 12	305
Thread 13	305
Thread 14	305
Thread 15	305
Thread 16	305
Thread 17	305
Thread 18	305
Thread 19	305
Thread 2	305
Thread 20	305
Thread 21	305
Thread 22	305
Thread 23	305
Thread 24	305
Thread 25	305
Thread 26	305
Thread 27	305
Thread 28	305
Thread 29	305
Thread 3	305
Thread 30	305
Thread 31	305
Thread 32	305
Thread 33	305
Thread 34	305

Woohoo!

The downstream work is now balanced across all 64 cores.

Everything that happens after the Distribute Streams operator will be more balanced across more cores, which is a good thing if we need it to give the server a workout.

It's a good thing, right?



3.1 p35

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.

ThreadStat	
Branches	3
ThreadReservation	
UsedThreads	129
WaitStats	
[1]	
WaitCount	5541
WaitTimeMs	146710
WaitType	CXPACKET



3.1 p36

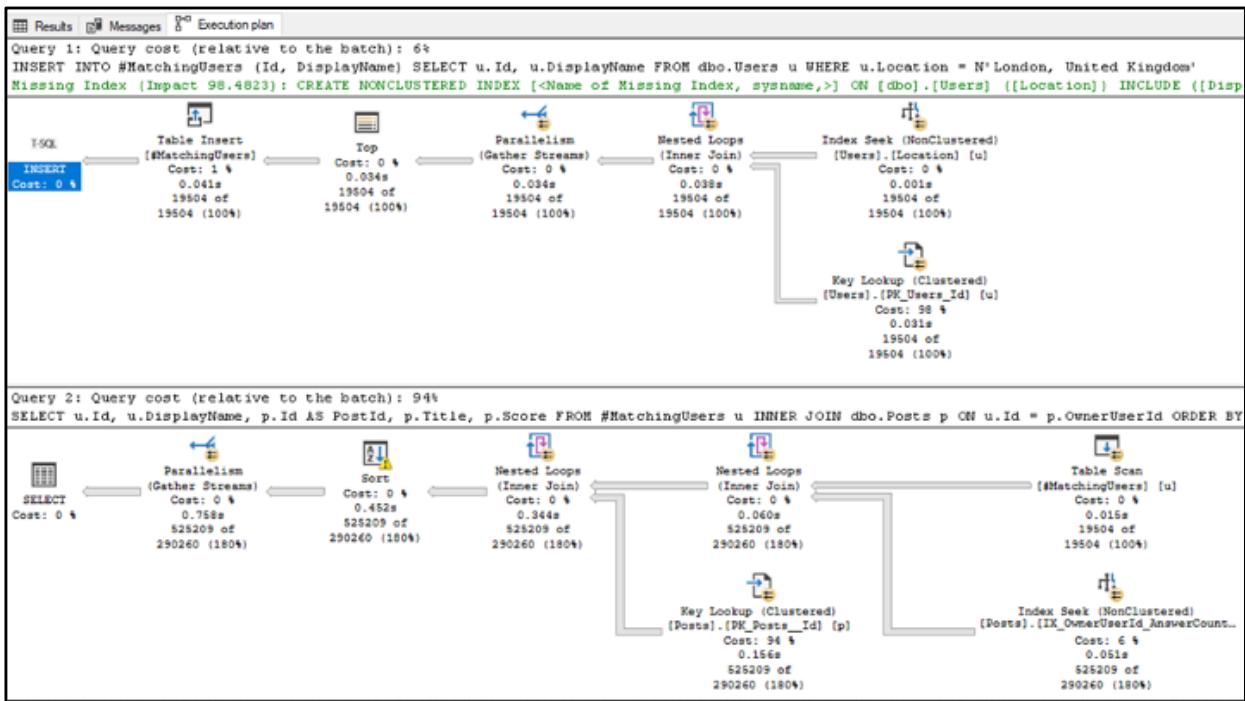
Temp table version

```
|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;
```



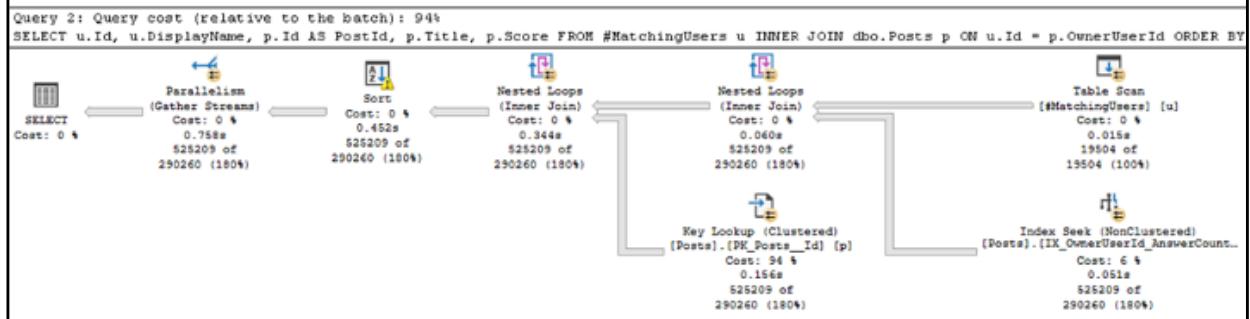
3.1 p37



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?



Properties	
Table Scan	
Misc	
Actual Execution Mode	Row
Actual I/O Statistics	
Actual Number of Batches	0
Actual Number of Rows	19504
Thread 0	0
Thread 1	445
Thread 10	448
Thread 11	456
Thread 12	460
Thread 13	453
Thread 14	457
Thread 15	450
Thread 16	452
Thread 17	452
Thread 18	220
Thread 19	452
Thread 2	453
Thread 20	456
Thread 21	450
Thread 22	452
Thread 23	222
Thread 24	226
Thread 25	461
Thread 26	450
Thread 27	229
Thread 28	232
Thread 29	236
Thread 3	458
Thread 30	230
Thread 31	214
Thread 32	228
Thread 33	229
Thread 34	220
Thread 35	223
Thread 36	224

PROBLEM SOLVED!

Phase 2 starts out with a nice (fairly even) distribution across all 64 cores.

But...that's only the start.

Then we do the index seeks into Posts.

And not all users have created the same number of posts.

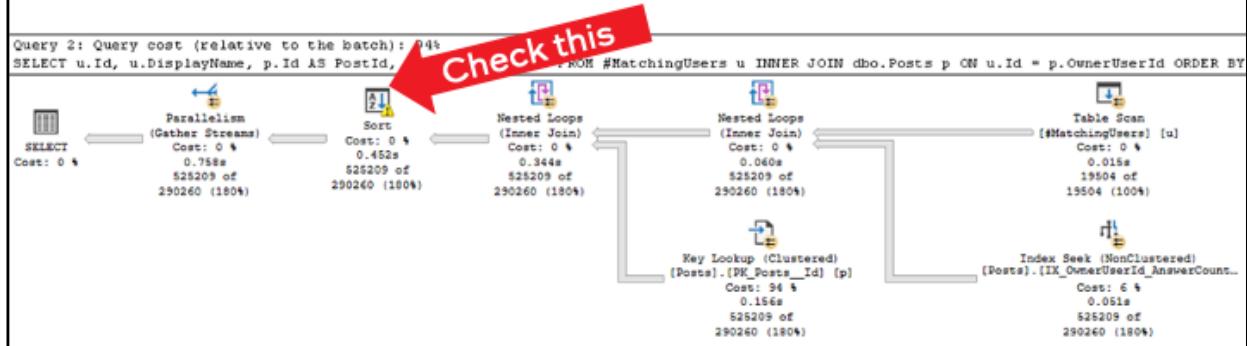


3.1 p40

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...



Properties	
Sort	
<input type="checkbox"/>	Actual Execution Mode Row
<input type="checkbox"/>	Actual I/O Statistics
<input type="checkbox"/>	Actual Number of Batch 0
<input checked="" type="checkbox"/>	Actual Number of Row 525209
Thread 0	0
Thread 1	7637
Thread 10	2300
Thread 11	3146
Thread 12	13955
Thread 13	7041
Thread 14	11470
Thread 15	7815
Thread 16	25670
Thread 17	17563
Thread 18	774
Thread 19	16634
Thread 2	8630
Thread 20	13549
Thread 21	17158
Thread 22	12977
Thread 23	2921
Thread 24	1607
Thread 25	9861
Thread 26	9179
Thread 27	5200
Thread 28	17435
Thread 29	3584
Thread 3	18407
Thread 30	7520
Thread 31	746
Thread 32	3922
Thread 33	4228
Thread 34	399
Thread 35	807

Not bad.

There are still outliers:

- **Thread 34: 399 rows**
- **Thread 16: 25,670 rows**

But it sure is better than:

- **3 threads working**
- **61 threads sitting idle**

So how do our wait stats look?



3.1 p42

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.

ThreadStat	
Branches	1
ThreadReservation	
UsedThreads	64
WaitStats	
[1]	
WaitCount	5311
WaitTimeMs	146105
WaitType	CXPACKET



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

```
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
```

Aggregate



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

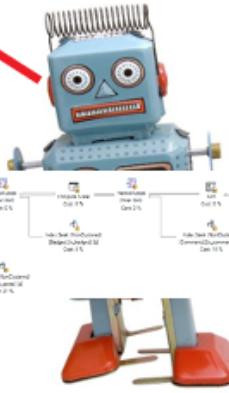
```
SELECT COUNT(*) AS BadgeCount  
FROM dbo.Badges AS b  
WHERE b.UserId = 22656;
```

BadgeCount
2821



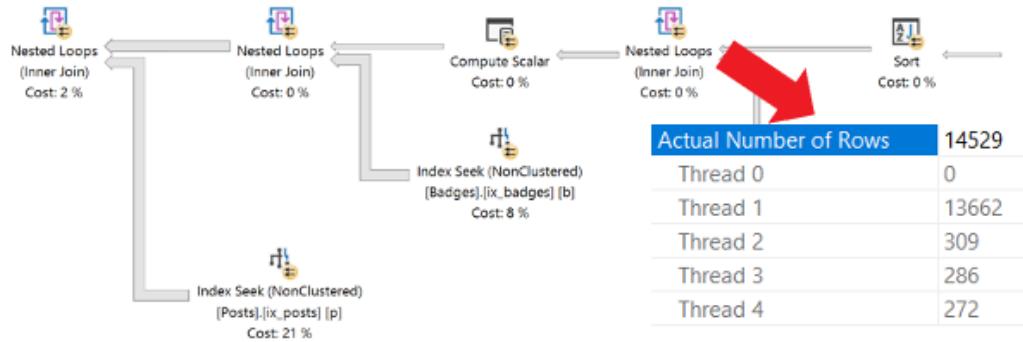
3.1 p50

This will run for
over two days



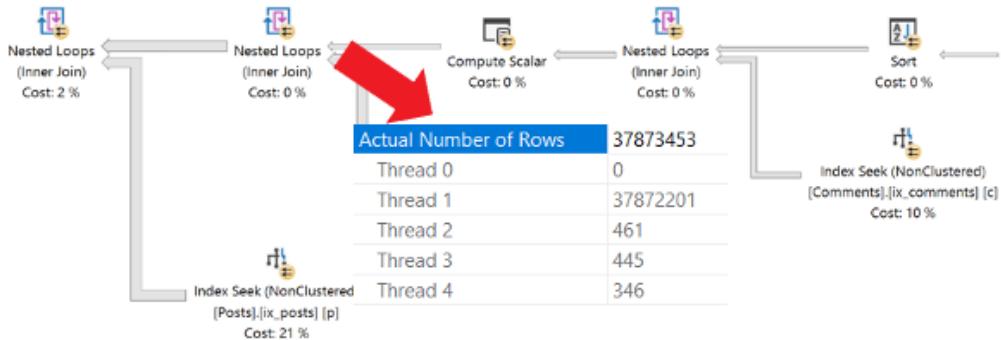
3.1 p51

Work from right to left...



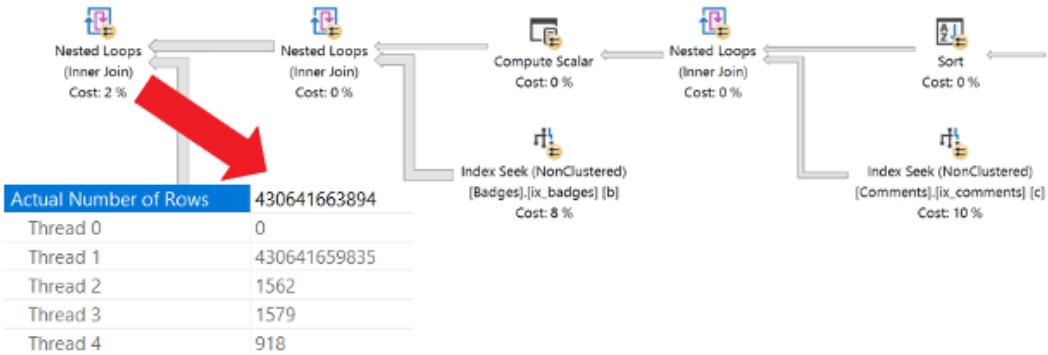
3.1 p52

And the further we go...



3.1 p53

The crazier it gets...



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



3.1 p54

Comma chameleon

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



3.1 p55



Index Seek (NonClustered)
[Posts].[ix_posts] [p]
Cost: 21 %

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	430641663894
Actual Number of Rows	430641663894
Actual Number of Batches	0
Estimated I/O Cost	0.003125
Estimated Operator Cost	5.78902 (21%)
Estimated CPU Cost	0.0001723
Estimated Subtree Cost	5.78902
Estimated Number of Executions	30073.604
Number of Executions	37873453
Estimated Number of Rows	13.8898
Estimated Number of Rows to be Read	13.8898
Estimated Row Size	11 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	33



3.1 p56

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/	xClickToSeeDetails -- We hope you found this tool useful. ... ?>
100	Query Performance	Queries with 10000x cardinality missetimations	N/A	xClickToSeeDetails -- The query on SPID 52 has been running for 11 seconds, with a large cardinality missetimate ... ?>
100	Query Performance	Queries with 10000x skewed parallelism	N/A	xClickToSeeDetails -- 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/>



3.1 p60

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

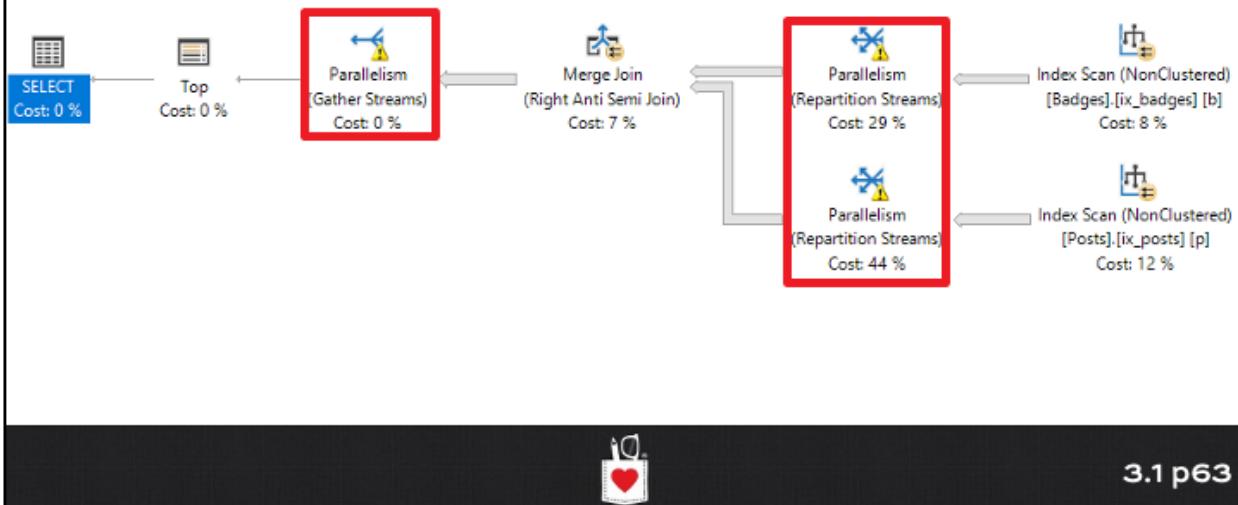
```
SELECT      TOP 10 p.OwnerUserId
FROM        dbo.Posts AS p
WHERE       NOT EXISTS (   SELECT    *
                           FROM      dbo.Badges AS b
                           WHERE     p.OwnerUserId = b.UserId )
ORDER BY    p.OwnerUserId
OPTION(MAXDOP 4, USE HINT('ENABLE_PARALLEL_PLAN_PREFERENCE'),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



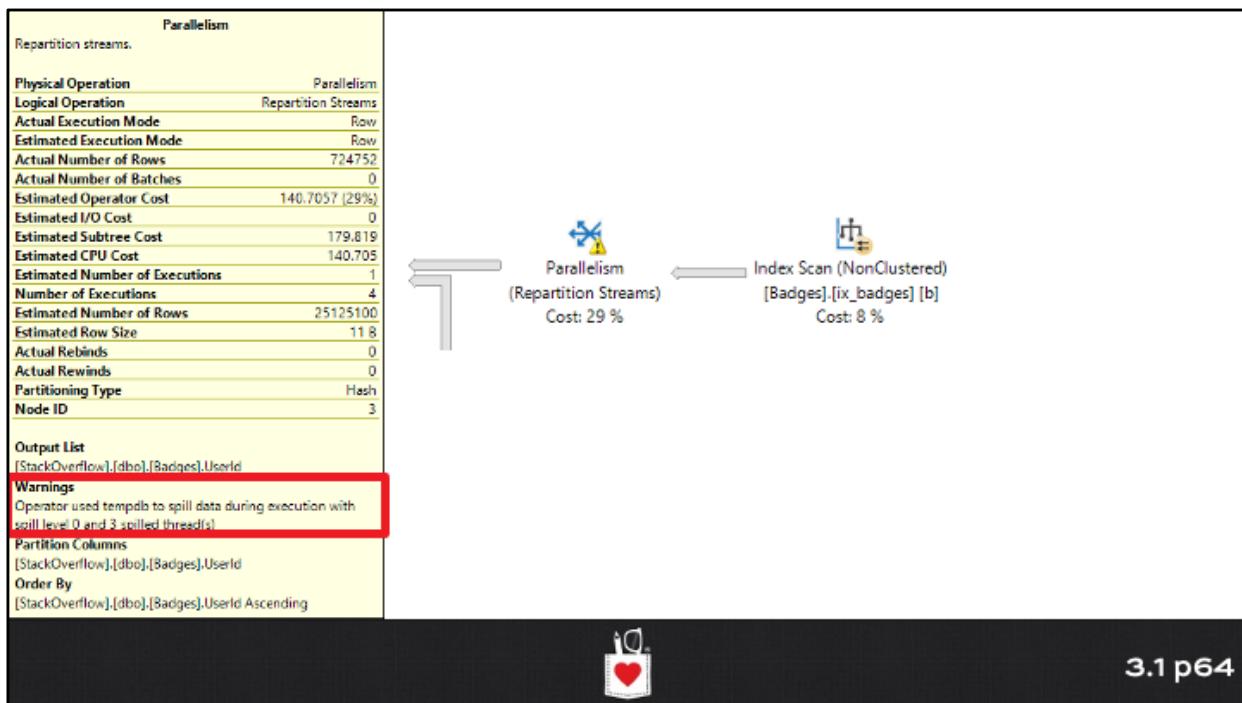
3.1 p62

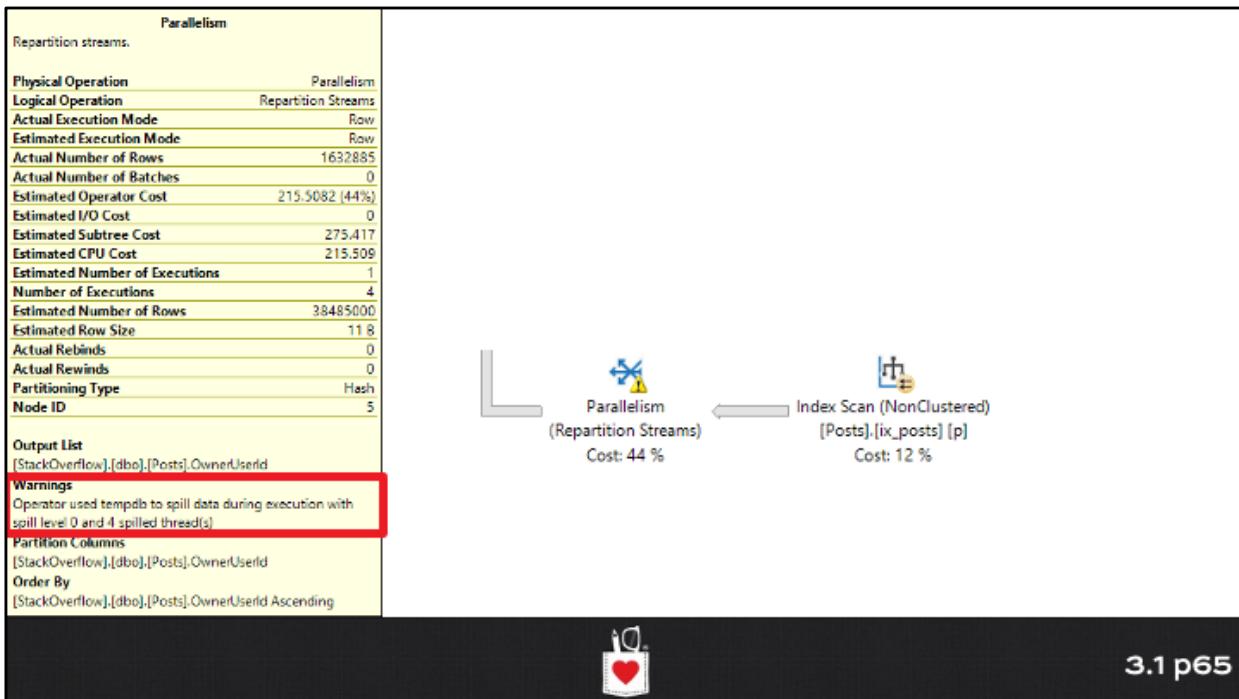
It runs for 8 minutes

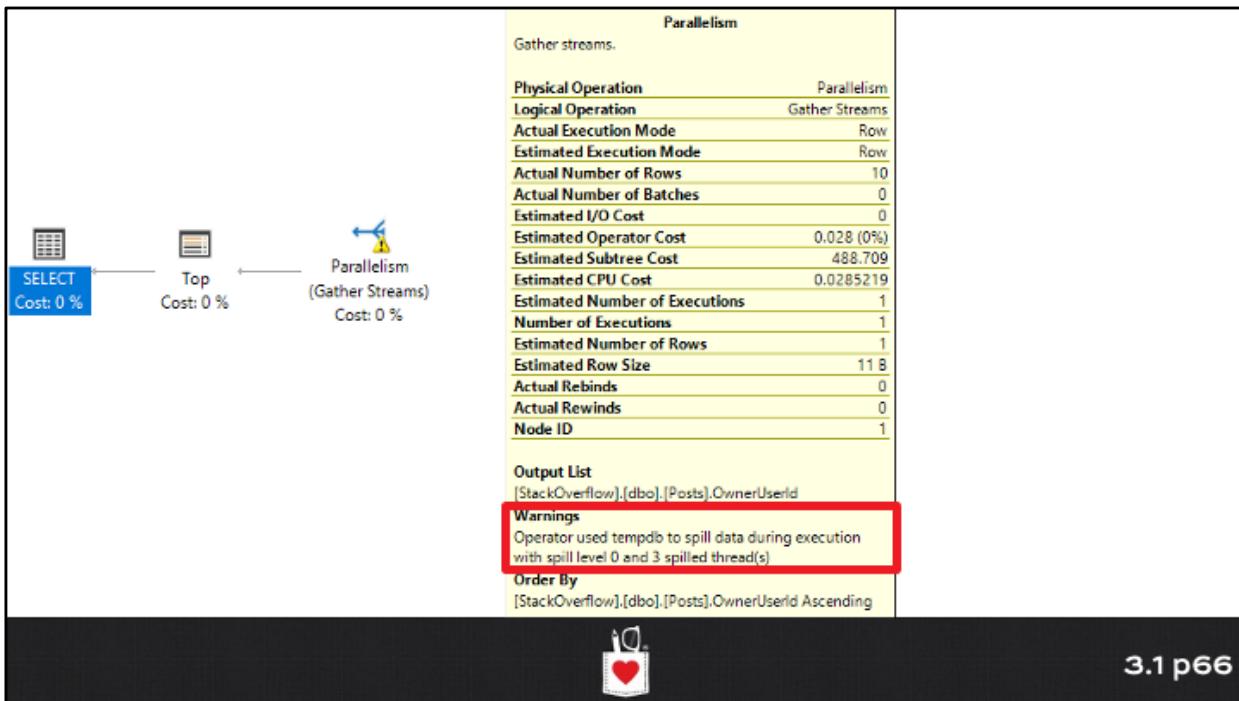
And has spills in unusual places:



3.1 p63







3.1 p66

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

Selected events:
Name
exchange_spill



name	timestamp	spid	query_id	sql_id	thread_id	blocking (UTC)	waitable_flag
exchange_spill	2018-07-23 14:20:34.5512371	Begin		1	SELECT	TCP..	1 2018-07-20 18:2.. 0
exchange_spill	2018-07-23 14:20:42.0438513	Begin		5	SELECT	TCP..	4 2018-07-20 18:2.. 0
exchange_spill	2018-07-23 14:20:47.0768891	Begin		5	SELECT	TCP..	4 2018-07-20 18:2.. 0
exchange_spill	2018-07-23 14:20:47.4013734	End		5	SELECT	TCP..	4 2018-07-20 18:2.. 15
exchange_spill	2018-07-23 14:20:47.4518097	End		5	SELECT	TCP..	4 2018-07-20 18:2.. 16
exchange_spill	2018-07-23 14:21:03.3621952	Begin		5	SELECT	TCP..	2 2018-07-20 18:2.. 0
exchange_spill	2018-07-23 14:21:06.3757635	Begin		3	SELECT	TCP..	1 2018-07-20 18:2.. 0
exchange_spill	2018-07-23 14:21:12.1381693	Begin		5	SELECT	TCP..	2 2018-07-20 18:2.. 0
exchange_spill	2018-07-23 14:21:14.6546944	Begin		3	SELECT	TCP..	1 2018-07-20 18:2.. 0
exchange_spill	2018-07-23 14:21:15.9205639	Begin		1	SELECT	TCP..	3 2018-07-20 18:2.. 0
exchange_spill	2018-07-23 14:21:26.2520855	Begin		5	SELECT	TCP..	0 2018-07-20 18:2.. 0
exchange_spill	2018-07-23 14:21:31.2320963	Begin		5	SELECT	TCP..	3 2018-07-20 18:2.. 0
exchange_spill	2018-07-23 14:21:37.6103491	Begin		3	SELECT	TCP..	4 2018-07-20 18:2.. 0
exchange_spill	2018-07-23 14:21:42.51533822	Begin		3	SELECT	TCP..	4 2018-07-20 18:2.. 0
exchange_spill	2018-07-23 14:21:48.776570	Begin		5	SELECT	TCP..	1 2018-07-20 18:2.. 0
exchange_spill	2018-07-23 14:21:53.7769533	Begin		5	SELECT	TCP..	1 2018-07-20 18:2.. 0
exchange_spill	2018-07-23 14:22:10.0091895	Begin		5	SELECT	TCP..	4 2018-07-20 18:2.. 0
exchange_spill	2018-07-23 14:22:15.0561935	Begin		5	SELECT	TCP..	4 2018-07-20 18:2.. 0
exchange_spill	2018-07-23 14:22:41.3404692	Begin		3	SELECT	TCP..	2 2018-07-20 18:2.. 0
exchange_spill	2018-07-23 14:22:46.3561107	Begin		3	SELECT	TCP..	2 2018-07-20 18:2.. 0
exchange_spill	2018-07-23 14:23:57.7134693	Begin		3	SELECT	TCP..	3 2018-07-20 18:2.. 0
exchange_spill	2018-07-23 14:28:02.7134016	Begin		3	SELECT	TCP..	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

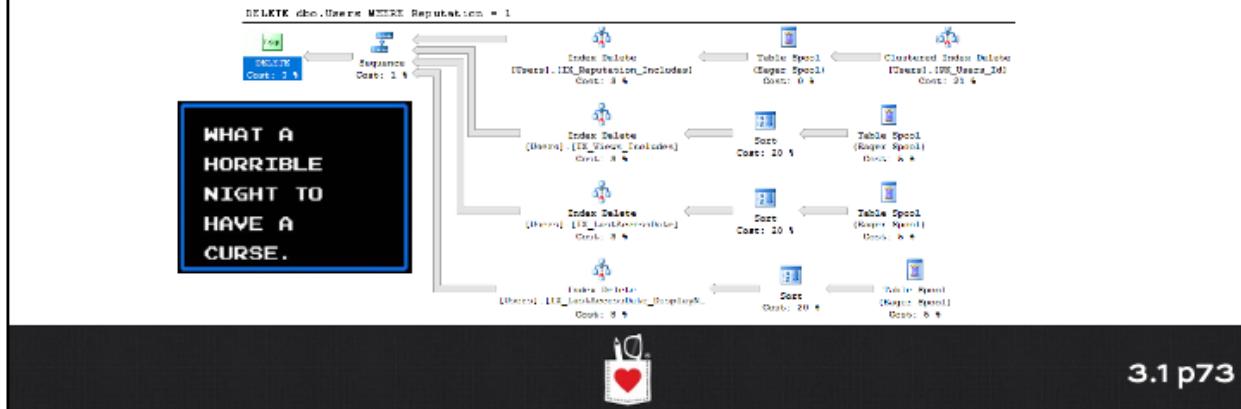
- Blocking, fetch all data needed at once



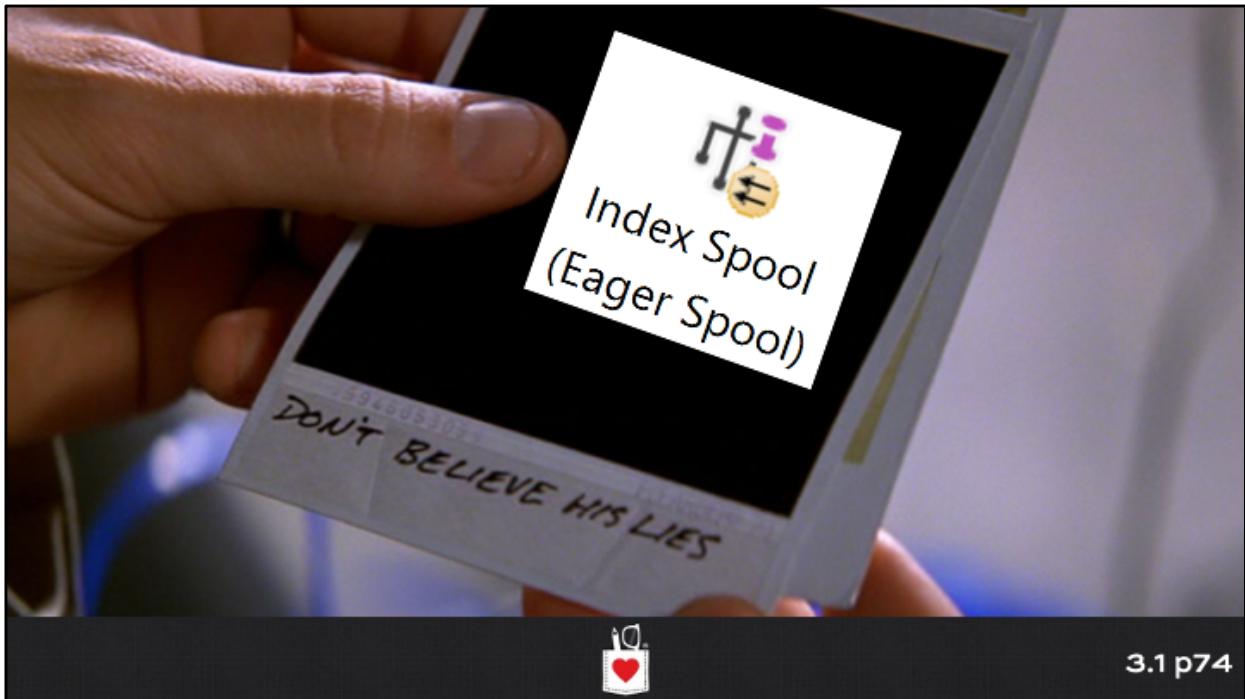
3.1 p72

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



3.1 p74

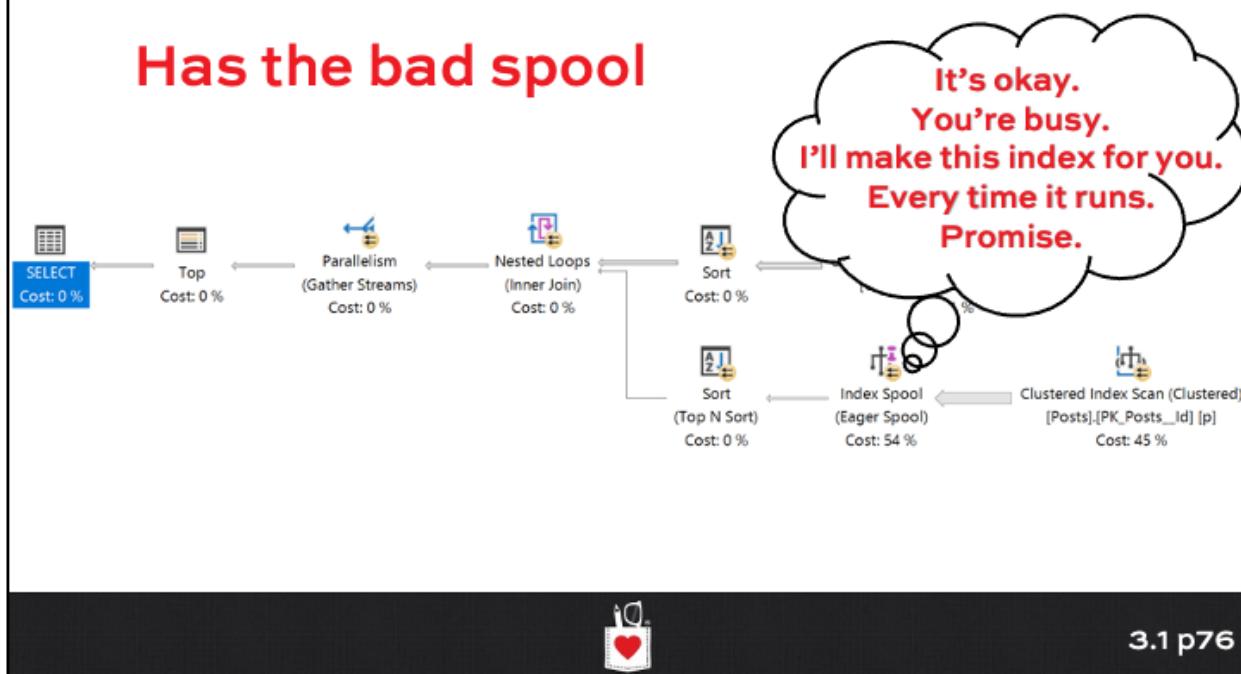
Test query

```
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;
```

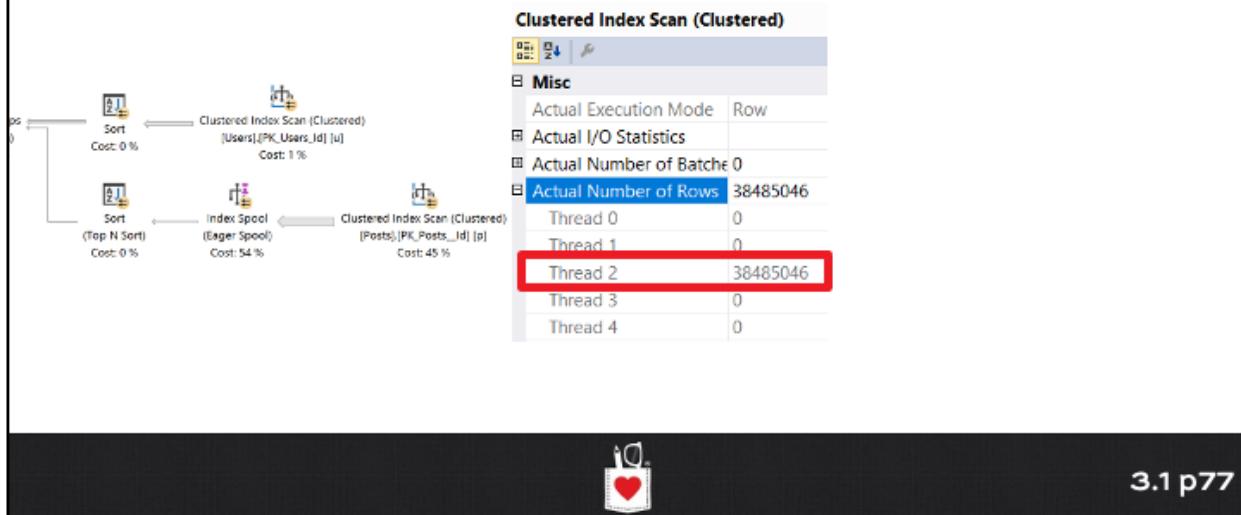


3.1 p75

Has the bad spool



SAYS it's parallel, but...



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...	ShowPlan	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

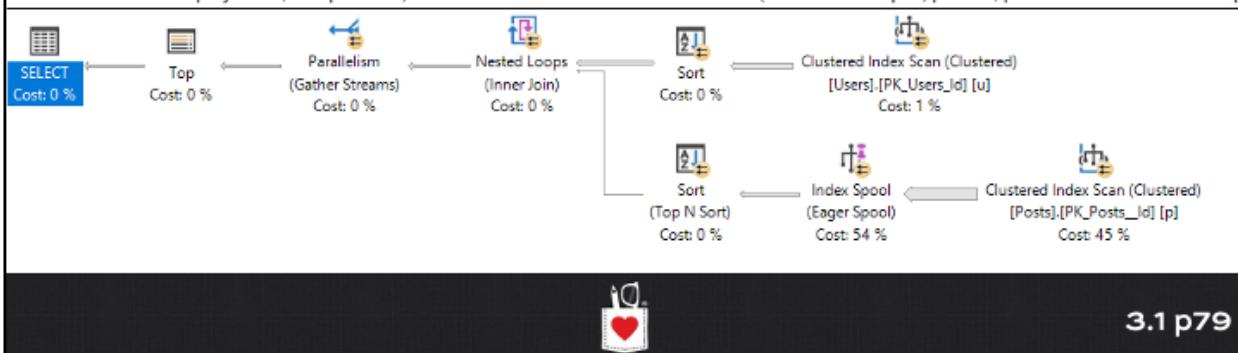
No missing index request

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

Index Spool	
Reformats the data from the input into a temporary index, which is then used for seeking with the supplied seek predicate.	
Physical Operation	Index Spool
Logical Operation	Eager Spool
Actual Execution Mode	Row
Estimated Execution Mode	Row
Actual Number of Rows	20402
Actual Number of Batches	0
Estimated I/O Cost	9286.94
Estimated Operator Cost	9325.84 (54%)
Estimated Subtree Cost	17115.6
Estimated CPU Cost	38.4854
Number of Executions	327
Estimated Number of Executions	101.987
Estimated Number of Rows	54.539
Estimated Row Size	269 B
Actual Rewinds	323
Node ID	6
Output List	[StackOverflow].[dbo].[Posts].Id, [StackOverflow].[dbo].[Posts].Score, [StackOverflow].[dbo].[Posts].Title
Seek Predicate	Seek Keys(1): Prefix([StackOverflow].[dbo].[Posts].OwnerId, [StackOverflow].[dbo].[Posts].PostTypeId = Scalar Operator([StackOverflow].[dbo].[Users].[Id] as [u].[Id]), Scalar Operator((1)))

Fixing it

This comes back to good old indexing

Hover over the Index Spool:

- Seek predicate(s): Key Columns
- Output List: Included Columns

Bonus points:

- Fix the sort on Score



3.1 p80

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.



3.1 p82