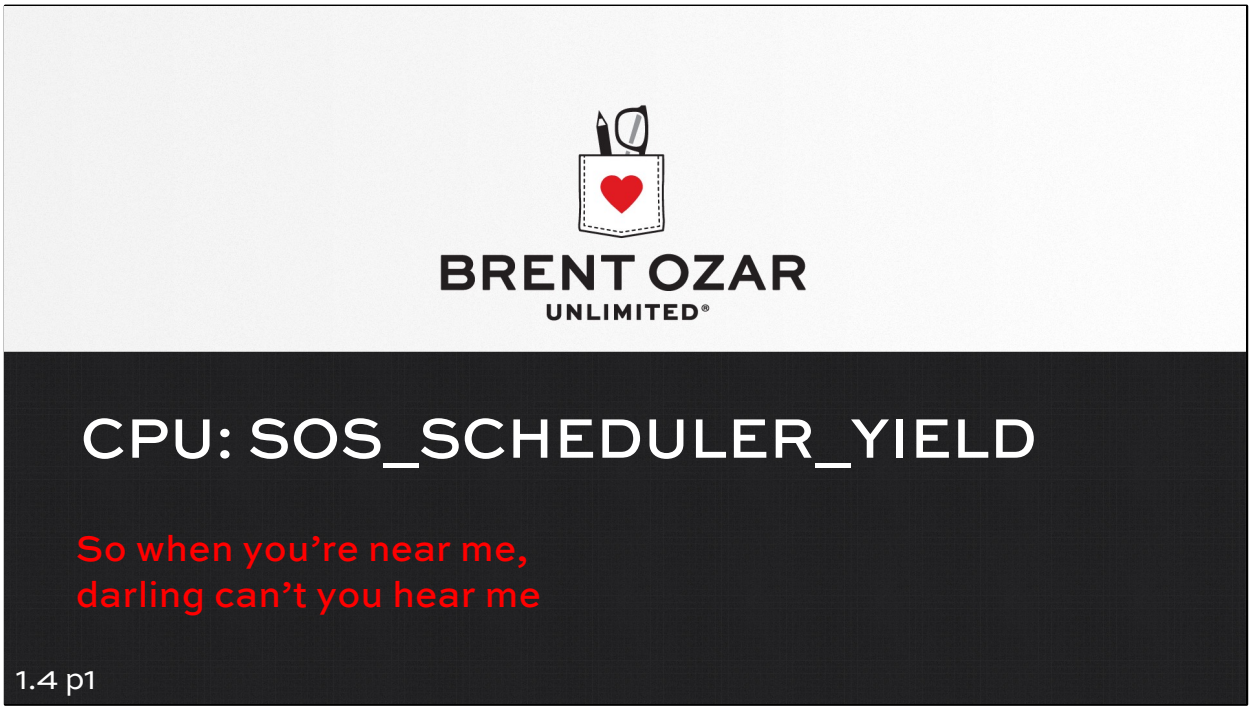


Agenda

What SOS_SCHEDULER_YIELD means

Demoing it

Causes:

- High CPU-using queries
- Other apps using CPU
- Not enough CPU power
- Compiling queries

1.4 p2

How SQL Server schedules CPU

What's Running Now **What's Waiting (Queue)**

1.4 p3

Assume we have the world's smallest SQL Server – it's only got one core.

How SQL Server schedules CPU

What's Running Now

SELECT *
FROM dbo.Restaurants
(By Brent)

What's Waiting (Queue)

1.4 p4

I run a query getting all of the restaurants. My Restaurants table happens to be mostly in cache, so it fires off and starts running. It will KEEP running – there's no concept of sharing CPU cycles in SQL Server. A query runs until it's done, or until it needs to wait on something like locks. More on that in a second.

How SQL Server schedules CPU

What's Running Now

SELECT *
FROM dbo.Restaurants
(By Brent)

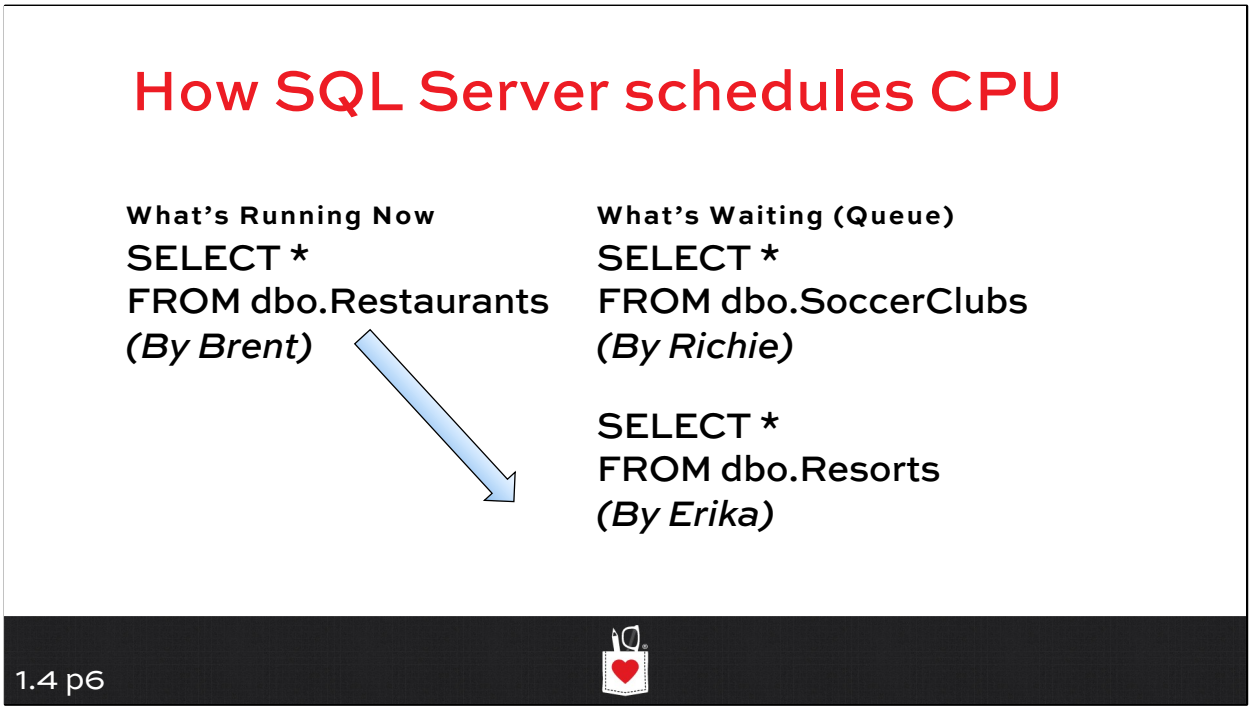
What's Waiting (Queue)

SELECT *
FROM dbo.SoccerClubs
(By Richie)

SELECT *
FROM dbo.Resorts
(By Erika)

1.4 p5

While my query is consuming CPU, other people's queries pile up behind me.



But the instant my query needs something that isn't in cache, like I need to wait for a locked page or I need to wait for something to come back from disk, I go to the back of the line.

How SQL Server schedules CPU

What's Running Now

What's Waiting (Queue)

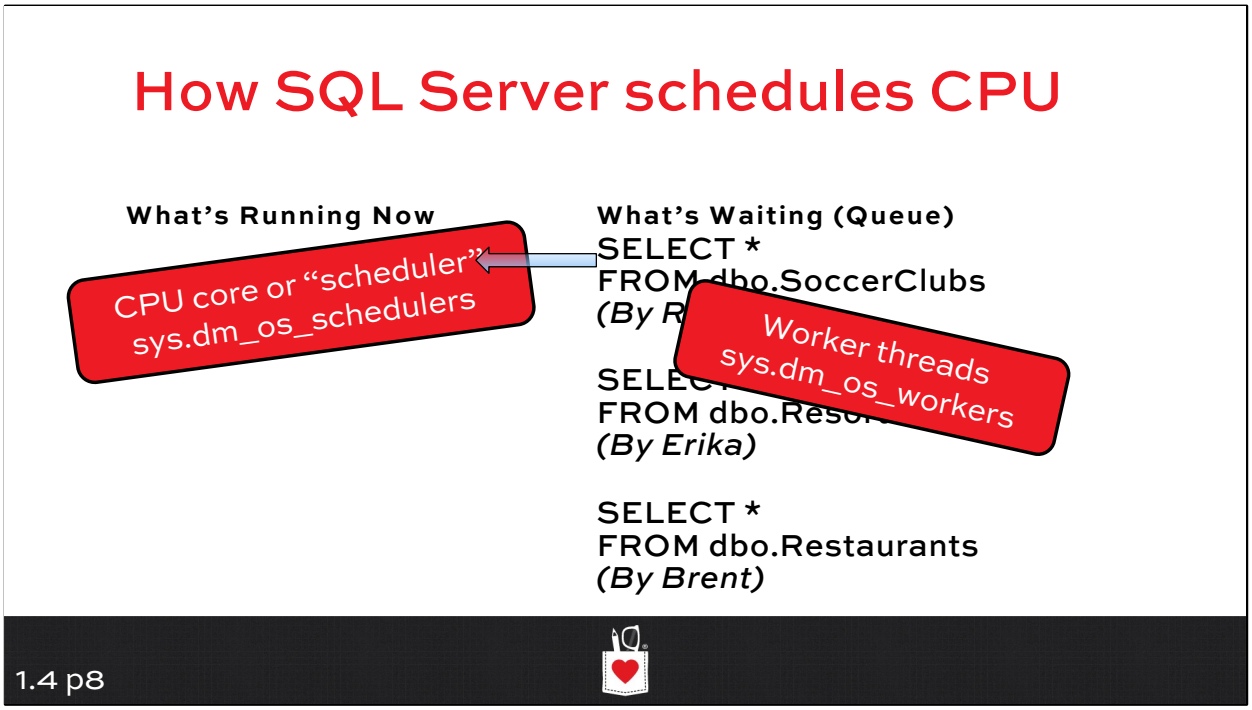
SELECT *
FROM dbo.SoccerClubs
(By Richie)

SELECT *
FROM dbo.Resorts
(By Erika)

SELECT *
FROM dbo.Restaurants
(By Brent)

1.4 p7

Other people's queries can then jump in. While mine is waiting, SQL Server tracks the number of milliseconds that I'm waiting on stuff.



Other people's queries can then jump in. While mine is waiting, SQL Server tracks the number of milliseconds that I'm waiting on stuff.

Let's run a normal query.

1.4 p9

The World's Tiniest Load Test

Set your parallelism settings back to the default:

- Cost Threshold for Parallelism = 5
- MAXDOP = 0 (unlimited)

Remove any indexes on the Users table

Run this:

```
SELECT COUNT(*) FROM dbo.Users;
```

1.4 p10

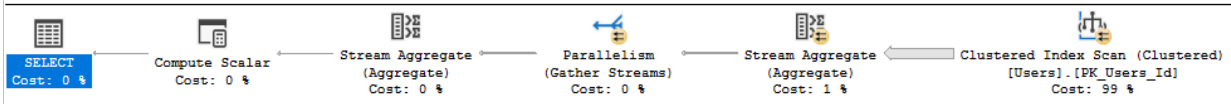
It's kinda big, so it goes parallel

```
SELECT COUNT(*) FROM dbo.Users;
```

150 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT COUNT(*) FROM dbo.Users
```



What we see

The query has a lot of work to do

Its query exceeds the Cost Threshold for Parallelism

It gets a parallel plan

If we time it right, we see 4-5 active worker threads running on different schedulers, all for the same query

We're not waiting on storage: the table is small, and it fits in RAM (this will be important later)

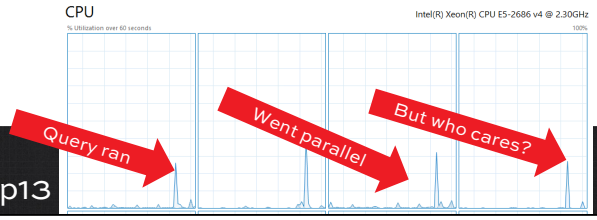
1.4 p12

One query can max out your box.

This isn't a surprise: we've all run ugly queries before.

They're just generally short-lived: CPU doesn't go to 100% for minutes on end.

But the box is still usable: note that other queries can still run fine.



1.4 p13

SQLOS multi-tasks.

After a task has been running on a scheduler for 4 milliseconds straight, it yields the CPU scheduler.

"Someone else can run now."

It hops over into the waiting queue – but it will immediately hop back onto the CPU scheduler if it can.

This wait type: SOS_SCHEDULER_YIELD.

1.4 p14

Our SELECT runs...

What's Running Now

SELECT COUNT(*)
FROM dbo.Users

What's Waiting (Queue)

1.4 p15

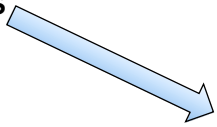
But the instant my query needs something that isn't in cache, like I need to wait for a locked page or I need to wait for something to come back from disk, I go to the back of the line.

After 4ms, we yield the scheduler

What's Running Now

SELECT COUNT(*)
FROM dbo.Users

What's Waiting (Queue)



1.4 p16

But the instant my query needs something that isn't in cache, like I need to wait for a locked page or I need to wait for something to come back from disk, I go to the back of the line.

We start waiting to get back on

What's Running Now

What's Waiting (Queue)

```
SELECT COUNT(*)
FROM dbo.Users
```

1.4 p17

But the instant my query needs something that isn't in cache, like I need to wait for a locked page or I need to wait for something to come back from disk, I go to the back of the line.

Since no one else is using CPU, we can hop right back in

What's Running Now	What's Waiting (Queue)
	SELECT COUNT(*) FROM dbo.Users

1.4 p18

But the instant my query needs something that isn't in cache, like I need to wait for a locked page or I need to wait for something to come back from disk, I go to the back of the line.

And we run for another 4ms

What's Running Now

SELECT COUNT(*)
FROM dbo.Users

What's Waiting (Queue)

1.4 p19

But the instant my query needs something that isn't in cache, like I need to wait for a locked page or I need to wait for something to come back from disk, I go to the back of the line.

It runs fast – but it does use CPU.

Users would call this a fast query.

Checking it

```
SET STATISTICS TIME ON;
GO
SELECT COUNT(*) FROM dbo.Users;
```

150 %

Results | Messages | Execution plan

Teensy

```
(1 row affected)

SQL Server Execution Times:
   CPU time = 2061 ms,  elapsed time = 342 ms.
```

But we do have to yield the CPU scheduler every 4 milliseconds.

How many times we stepped off:
2061ms / 4ms quantum = ~515 times.

1.4 p20

Low-load symptoms

Other queries work fine – they just may need to wait 4 milliseconds for CPU time.

We might see SOS_SCHEDULER_YIELD waits, but their average wait time will be very short – because we're simply hopping off the scheduler, and getting back on.

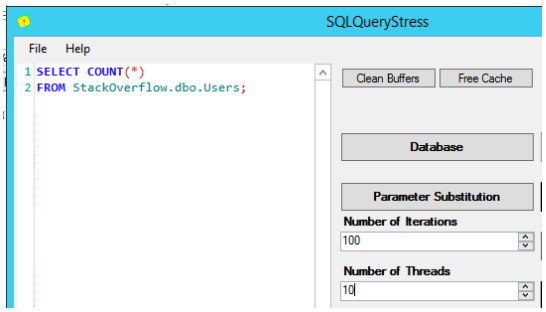
Let's run a lot of normal queries.

1.4 p22

Let's stress test it

Run our query 100 iterations across 10 threads

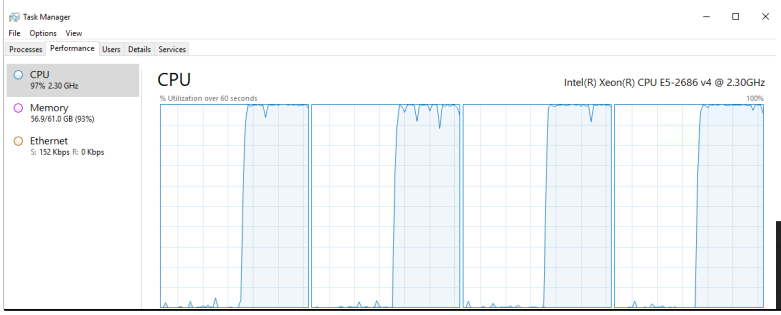
Note: this "threads" refers to SQLQueryStress, not SQL




```
SELECT COUNT(*)
FROM StackOverflow.dbo.Users
```


CPU maxes out – but it still works

Performance isn't pretty, but you can still run queries and see DMVs.



Struggling, but still responsive

Each query goes parallel

All of these queries want CPU time:
they're all dealing with in-memory data

Now we have 10 active queries,
each of which needs at least 4-5 worker threads

1.4 p25

Medium-load symptoms

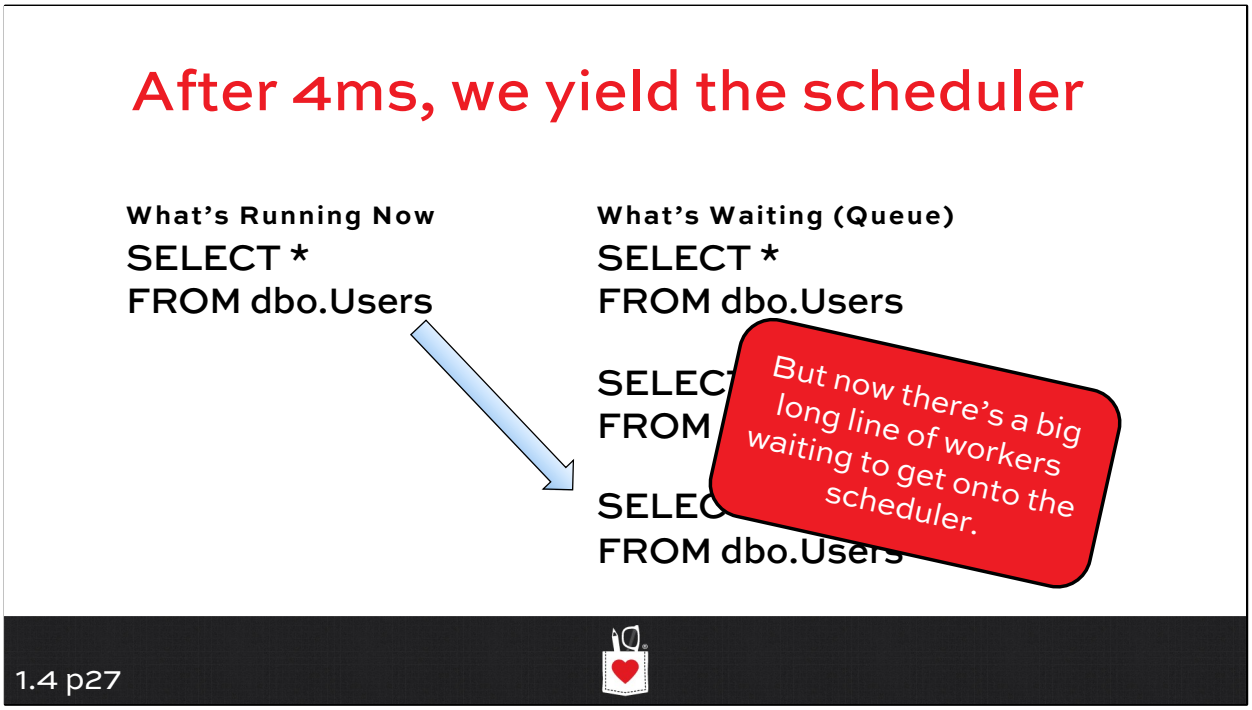
Queries take longer to run

CPUs go higher (70-100%)

SOS_SCHEDULER_YIELD wait times (total & average) rise higher

	Pattern	Sample Ended	Seconds Sample	wait_type	wait_category	Wait Time (Seconds)	Avg ms Per Wait
1	WAIT STATS	2018-02-14 12:43:40.8591772 +00:00	9	SOS_SCHEDULER_YIELD	CPU	455.3	25.3
2	WAIT STATS	2018-02-14 12:43:40.8591772 +00:00	9	CXPACKET	Parallelism	224.0	424.2
3	WAIT STATS	2018-02-14 12:43:40.8591772 +00:00	9	LATCH_EX	Latch	1.2	7.7
4	WAIT STATS	2018-02-14 12:43:40.8591772 +00:00	9	LATCH_SH	Latch	0.4	6.5
5	WAIT STATS	2018-02-14 12:43:40.8591772 +00:00	9	SESSION_WAIT_STATS_CHILDREN	Other	0.3	2.3
6	WAIT STATS	2018-02-14 12:43:40.8591772 +00:00	9	CXCONSUMER	Parallelism	0.3	1.9

1.4 p26



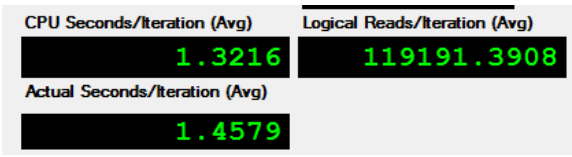
But the instant my query needs something that isn't in cache, like I need to wait for a locked page or I need to wait for something to come back from disk, I go to the back of the line.

So our queries take longer

It's not that each query uses *more* CPU time.

It just proportionally *gets* less time per second on the clock, because it has to wait so much longer to get in.

Check SQLQueryStress's "Actual Seconds/Interation" metric.



Bad SOS_SCHEDULER_YIELD

We're still seeing SOS_SCHEDULER_YIELD waits.

But now when we yield the CPU scheduler,
we have to wait a long time to get back on.

Average wait time for SOS_SCHEDULER_YIELD
tells the story of overloaded CPUs.

Ouch

	Pattern	Sample Ended	Seconds Sample	wait_type	wait_category	Wait Time	Avg ms Per Wait
1	WAIT STATS	2018-02-14 12:43:40.8591772 +00:00	9	SOS_SCHEDULER_YIELD	CPU	455.3	25.3
2	WAIT STATS	2018-02-14 12:43:40.8591772 +00:00	9	CXPACKET	Parallelism	224.0	424.2
3	WAIT STATS	2018-02-14 12:43:40.8591772 +00:00	9	LATCH_EX	Latch	1.2	7.7
4	WAIT STATS	2018-02-14 12:43:40.8591772 +00:00	9	LATCH_SH	Latch	0.4	6.5
5	WAIT STATS	2018-02-14 12:43:40.8591772 +00:00	9	SESSION_WAIT_STATS_CHILDREN	Other	0.3	2.3
6	WAIT STATS	2018-02-14 12:43:40.8591772 +00:00	9	CXCONSUMER	Parallelism	0.3	1.9

Let's run a LOT of normal queries.

1.4 p30

Kick out the jams

In SQLQueryStress, go from 10 threads to 150

How long does each query take to run?

What do your CPUs look like now?
(They can't really get higher than they did before)

What are your wait stats?
(But now our queries can spend more time waiting)

What's the average wait time look like now?

Wait stats are horrifying

Seconds Sample	wait_type	Wait Time (Seconds)	Per Core Per Second	Signal Wait Time (Seconds)	Percent Signal Waits	Number of Waits	Avg ms Per Wait
16	SOS_SCHEDULER_YIELD	3555.8	55.6	3555.8	100.0	15714	226.3
16	CXPACKET	1027.7	16.1	1.5	0.1	985	1043.4
16	THREADPOOL	3.1	0.0	0.0	0.0	57	54.9
16	LATCH_EX	1.0	0.0	0.3	30.0	204	4.8
16	LATCH_SH	0.0	0.0	0.0	0.0	7	4.7

The totals are bad:
in 1 second, on 1 core, queries are spending dozens of seconds waiting on more CPU time.

After using 4ms of CPU time and yielding, tasks wait hundreds of milliseconds to get back on the CPU!

1.4 p32

High load usually looks like this.

We have ugly queries running, and they need CPU.

We don't have enough CPU power.

Monitoring alerts us that we're hitting 100% CPU for long periods of time.

Even simple DMV queries can take minutes.
(sp_BlitzFirst can take 4-5 minutes to run!)

In most shops, this is where alarm bells get raised.

But not everywhere. More on that later.

Common causes

1.4 p34

Common causes

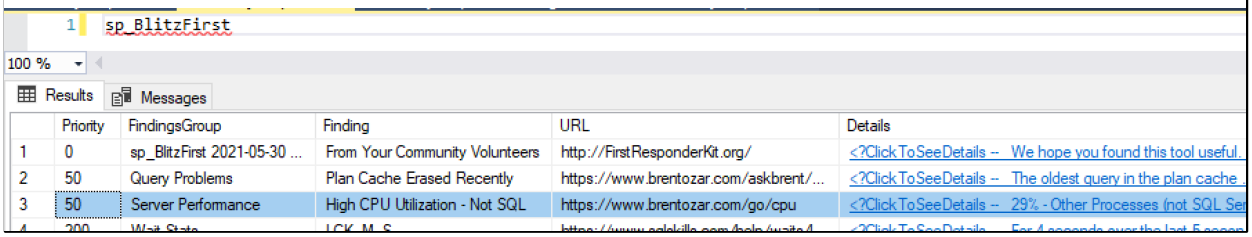
1. Other processes on the same server using CPU
2. Queries using a lot of CPU
3. Not enough CPU power in the server
4. Queries compiling over and over
   (we'll cover this in a separate module)

1.4 p35

Other processes using CPU

I put this first because it's the first one to rule out:

- Don't install other services on the server
- Don't remote desktop into the server
- Check Task Manager by CPU usage
- Check the ring buffers (sp_BlitzFirst does this)
  https://techcommunity.microsoft.com/t5/sql-server/sql-server-cpu-usage-available-in-sys-dm-os-ring-buffers-dmv/ba-p/825361



	Priority	FindingsGroup	Finding	URL	Details
1	0	sp_BlitzFirst 2021-05-30 ...	From Your Community Volunteers	http://FirstResponderKit.org/	<?ClickToSeeDetails -- We hope you found this tool useful.
2	50	Query Problems	Plan Cache Erased Recently	https://www.brentozar.com/askbrent/...	<?ClickToSeeDetails -- The oldest query in the plan cache
3	50	Server Performance	High CPU Utilization - Not SQL	https://www.brentozar.com/go/cpu	<?ClickToSeeDetails -- 29% - Other Processes (not SQL Ser
4	200	Wait State	LCK_M_S	https://www.sqlskills.com/help/waits/l	<?ClickToSeeDetails -- For 4 seconds over the last 5 secon

Common causes

1. Other processes on the same server using CPU
2. Queries using a lot of CPU
3. Not enough CPU power in the server
4. Queries compiling over and over
   (we'll cover this in a separate module)

Next

1.4 p37

Tuning queries for lower CPU

Normally we focus on:

- Reducing logical reads
- SET STATISTICS IO ON
- Changing table scans to index seeks
- Getting accurate row estimates

But tuning for lower CPU is different.

But for SOS_SCHEDULER_YIELD

Find the culprits:

```
sp_BlitzCache @SortOrder = 'cpu'
```

And look for:

- String processing (`LIKE '%STRING%'`)
- Row-based processing (cursors, functions)
- Sorting/grouping (but index for that, not reads)
- Heaps with forwarded fetches (rebuild, add CX)
- Implicit conversions

1.4 p39

String processing

There's a real cost to performing work on strings:

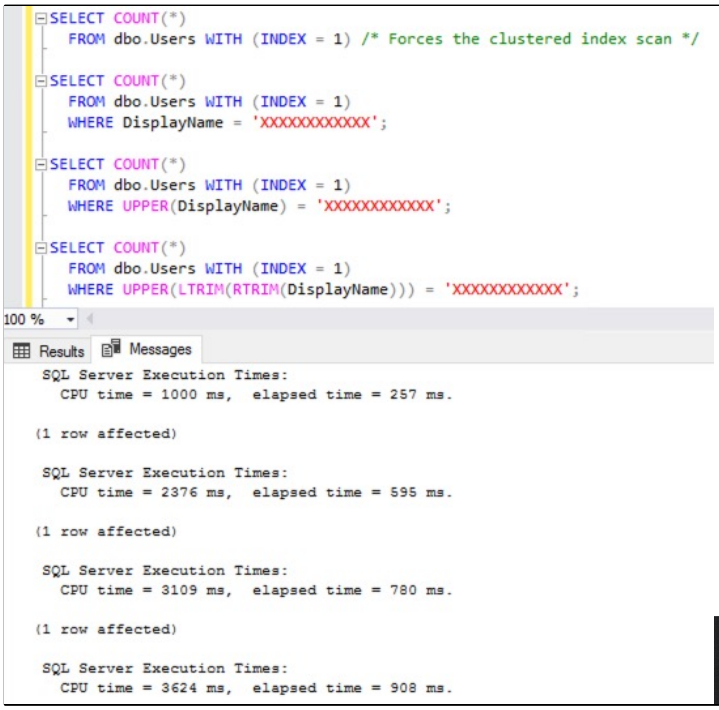
```
SELECT COUNT(*)
  FROM dbo.Users WITH (INDEX = 1) /* Forces the clustered index scan */

SELECT COUNT(*)
  FROM dbo.Users WITH (INDEX = 1)
  WHERE DisplayName = 'XXXXXXXXXXXX';

SELECT COUNT(*)
  FROM dbo.Users WITH (INDEX = 1)
  WHERE UPPER(DisplayName) = 'XXXXXXXXXXXX';

SELECT COUNT(*)
  FROM dbo.Users WITH (INDEX = 1)
  WHERE UPPER(LTRIM(RTRIM(DisplayName))) = 'XXXXXXXXXXXX';
```

1.4 p40



Add it up

Even just checking the name adds CPU time.

Then the more functions you run on strings, the longer it takes.

Fixes: tuned code, persisted computed columns, indexes.

Row-based processing

Scalar functions & multi-statement TVFs:
they run for every row. Try inlining them either by rewriting the function, or copy/pasting the code into the calling query.

Cursors: process one row at a time. Try turning them into set-based operations.

1.4 p42

Grouping & Distinct

```
SELECT COUNT(*) AS Folks
  FROM dbo.Users WITH (INDEX = 1);

SELECT Location, COUNT(*) AS Folks
  FROM dbo.Users WITH (INDEX = 1)
  GROUP BY Location;  /* Additional work */

SELECT Location, COUNT(*) AS Folks
  FROM dbo.Users WITH (INDEX = 1)
  GROUP BY Location
  ORDER BY COUNT(*) DESC;  /* Only calculated after grouping */
```

100 %

Results | Messages

```
SQL Server Execution Times:
  CPU time = 1000 ms,  elapsed time = 255 ms.

(138111 rows affected)

SQL Server Execution Times:
  CPU time = 4468 ms,  elapsed time = 2273 ms.

(138111 rows affected)

SQL Server Execution Times:
  CPU time = 4626 ms,  elapsed time = 2755 ms.
```

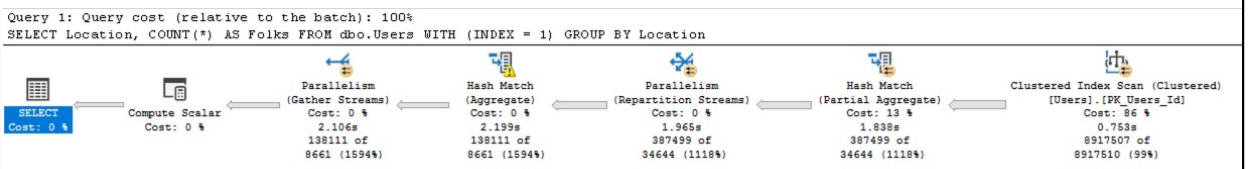
Sure, the users may want the data grouped, but it costs CPU time.

Note how the CPU time skyrockets when we have to group.

Only jumps a little with the sort on COUNT(*) because there are few rows then.

1.4 p43

The GROUP BY plan



Goes parallel across multiple cores

To do hashes of the unsorted data (twice)

And spills to disk once to boot

The problem isn't the reads: the table is small. The problem is all the work we're asking it to do!

1.4 p44

ORDER BY

SQL Server is the #2 most expensive data sorter.

If you're SOS_SCHEDULER_YIELD bottlenecked,
and your query doesn't have a TOP in it,
do the ordering in the app tier.

App servers are:

- Easy to scale out horizontally
- Free from SQL Server licensing costs

1.4 p45

Heaps with forwarded fetches

Flash back to Mastering Index Tuning

Heap: table without clustered indexes

Forwarding pointer: left behind when a row gets wider, usually updates on variable-length or nullable fields

Forwarded fetch: when SQL Server has to jump around during reads to follow the forwarding pointer, incurring additional reads (and more CPU)

Fix: rebuild the heap or put a clustered index on it

1.4 p46

Implicit conversions

Mismatched datatypes can be:

- Parameters in stored procs vs data in the table
- Joins between tables, but different data types
  - dbo.Products has ProductType = NVARCHAR
  - dbo.Sales has ProductType = VARCHAR

Both problems have the same effects:

- Running conversions on all data in the table
- Bad estimates
- Likely to tip over to table scans due to the estimates

1.4 p47

Common causes

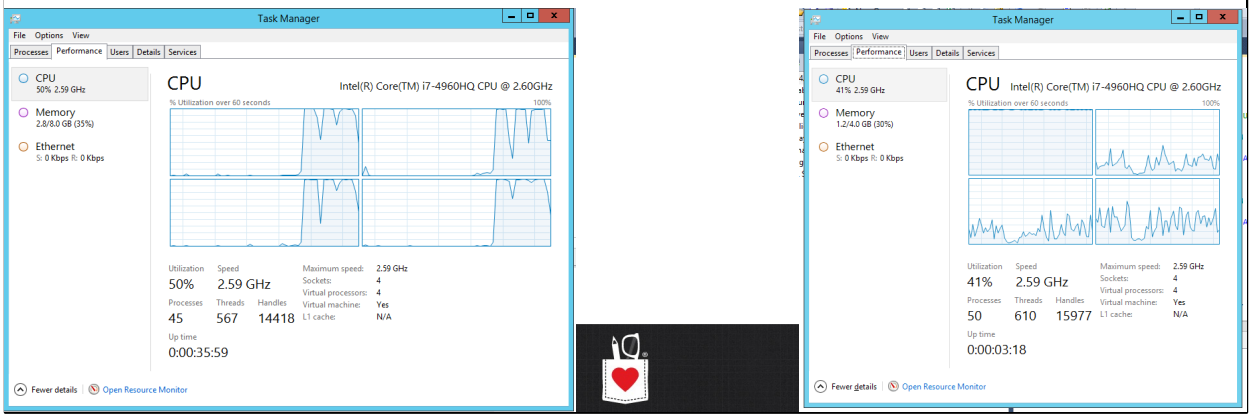
1. Other processes on the same server using CPU
2. Queries using a lot of CPU
3. Not enough CPU power in the server
4. Queries compiling over and over
   (we'll cover this in a separate module)

Next

1.4 p48

Adding more CPU power

SOS_SCHEDULER_YIELD means a query needs more CPU time, but...are we maxed out?

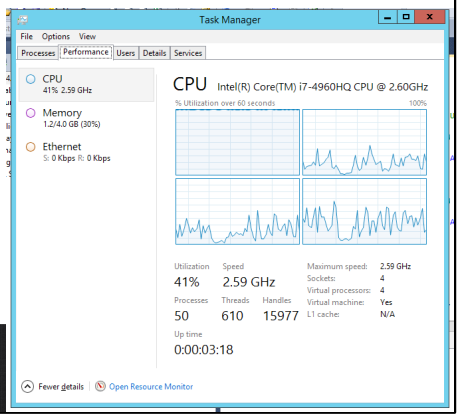


If you're not maxed out...

If it's only one core that's busy, it could be:

- Queries limited to 1 core due to parallelism inhibitors: BrentOzar.com/go/serialudf
- MAXDOP 1
- Cost underestimations
- CTFP set too high

Don't add more cores here.



But if CPU is 70-80-90%...

Physical box:

- Make sure balanced power saving isn't on

Conventional VM:

- vMotion to your fastest host
- vMotion other guests off your host temporarily
- Shut down, add cores, revisit MAXDOP

Cloud VM:

- Check the CPU types used by your instance type
- Consider changing instance types, or adding cores

1.4 p51

Example: AWS EC2 .2xlarge types

All of these have 8 cores, but clock speed isn't equal.

"Unknown" = custom silicon for AWS, 3-4GHz.

API Name	GiB of Memory per vCPU	Physical Processor	Clock Speed(GHz)	Windows On Demand cost
Search API Name	Search GiB of Memory pe	Search Physical Processor	Search Clock Speed(GH:	Search Windows On Dema
m2.2xlarge	8.55 GiB/vCPU	Intel Xeon Family	unknown	$503.700000 monthly
x2iezn.2xlarge	32.00 GiB/vCPU	Intel Xeon Platinum 8252	4.5 GHz	$1486.280000 monthly
z1d.2xlarge	8.00 GiB/vCPU	Intel Xeon Platinum 8151	4 GHz	$811.760000 monthly
x2iedn.2xlarge	32.00 GiB/vCPU	Intel Xeon Scalable (Icelake)	3.5 GHz	$1485.732500 monthly
i4i.2xlarge	8.00 GiB/vCPU	Intel Xeon 8375C (Ice Lake)	3.5 GHz	$769.420000 monthly
r6i.2xlarge	8.00 GiB/vCPU	Intel Xeon 8375C (Ice Lake)	3.5 GHz	$636.560000 monthly
r6id.2xlarge	8.00 GiB/vCPU	Intel Xeon 8375C (Ice Lake)	3.5 GHz	$710.144000 monthly
d3.2xlarge	8.00 GiB/vCPU	Intel Xeon Platinum 8259 (Cascade Lake)	3.1 GHz	$997.910000 monthly
i3en.2xlarge	8.00 GiB/vCPU	Intel Xeon Platinum 8175 (Skylake)	3.1 GHz	$928.560000 monthly
r5.2xlarge	8.00 GiB/vCPU	Intel Xeon Platinum 8175	3.1 GHz	$636.560000 monthly

Typical client issue

The SQL Server was deployed 2-3 years ago on this:

API Name	GiB of Memory per vCPU	Physical Processor	Clock Speed(GHz)	Windows On Demand cost
i3en.2xlarge	8.00 GiB/vCPU	Intel Xeon Platinum 8175 (Skylake)	3.1 GHz	$928.560000 monthly

Today, SOS is a problem, so switch to one of these:

API Name	GiB of Memory per vCPU	Physical Processor	Clock Speed(GHz)	Windows On Demand cost
x2iezn.2xlarge	32.00 GiB/vCPU	Intel Xeon Platinum 8252	4.5 GHz	$1486.280000 monthly
z1d.2xlarge	8.00 GiB/vCPU	Intel Xeon Platinum 8151	4 GHz	$811.760000 monthly

And instantly get 29%-45% more CPU cycles for the same number of cores & licensing cost.

1.4 p53

Resources for instance families

AWS: https://ec2instances.info

Azure VMs: https://azure-instances.info

Google Compute Engine: specify the minimum CPU platform when you build your VM. https://cloud.google.com/compute/docs/instances/specify-min-cpu-platform

Recap

1.4 p55

What we covered

SOS_SCHEDULER_YIELD happens when a task uses CPU for 4 milliseconds straight

Average wait time = how long the task had to wait until it got back on CPU again

Causes:

- High CPU-using queries
- Other apps using CPU
- Not enough CPU power
- Compiling queries

1.4 p56

Query issues that burn CPU

String processing (`LIKE '%STRING%'`)

Row-based processing (cursors, functions)

Sorting/grouping (but index for that, not reads)

Heaps with forwarded fetches (rebuild, add CX)

Implicit conversions

1.4 p57