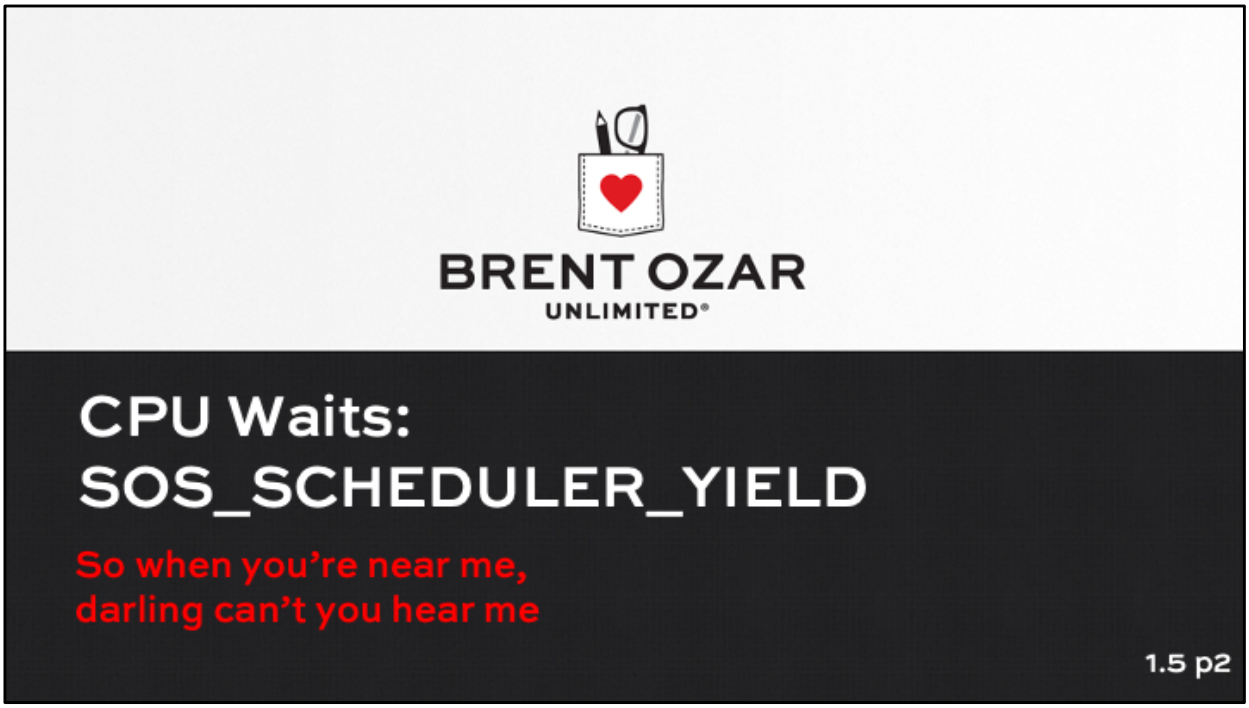
Setting up for the lab

1. Restart the SQL Server service (clears stats)
2. Restore your StackOverflow database
3. Copy & run the setup script, will take 60-90 seconds: BrentOzar.com/go/serverlab2
4. Start SQLQueryStress:
    1. File Explorer, D:\Labs, run SQLQueryStress.exe
    2. Click File, Open, D:\Labs\ServerLab2.json
    3. Click Go

1.5 p1



CPU scheduling can go deep.

I can teach you just a little,
and you'll know enough to get by.

But if you're curious about SQL Server,
I can go deeper.

When I'm teaching an "extra"
concept, I'll include this guy.

If you only wanna know basics,
you can snooze on those slides.

1.5 p3

Agenda

SQLOS: cores, schedulers, worker threads

Running a normal query

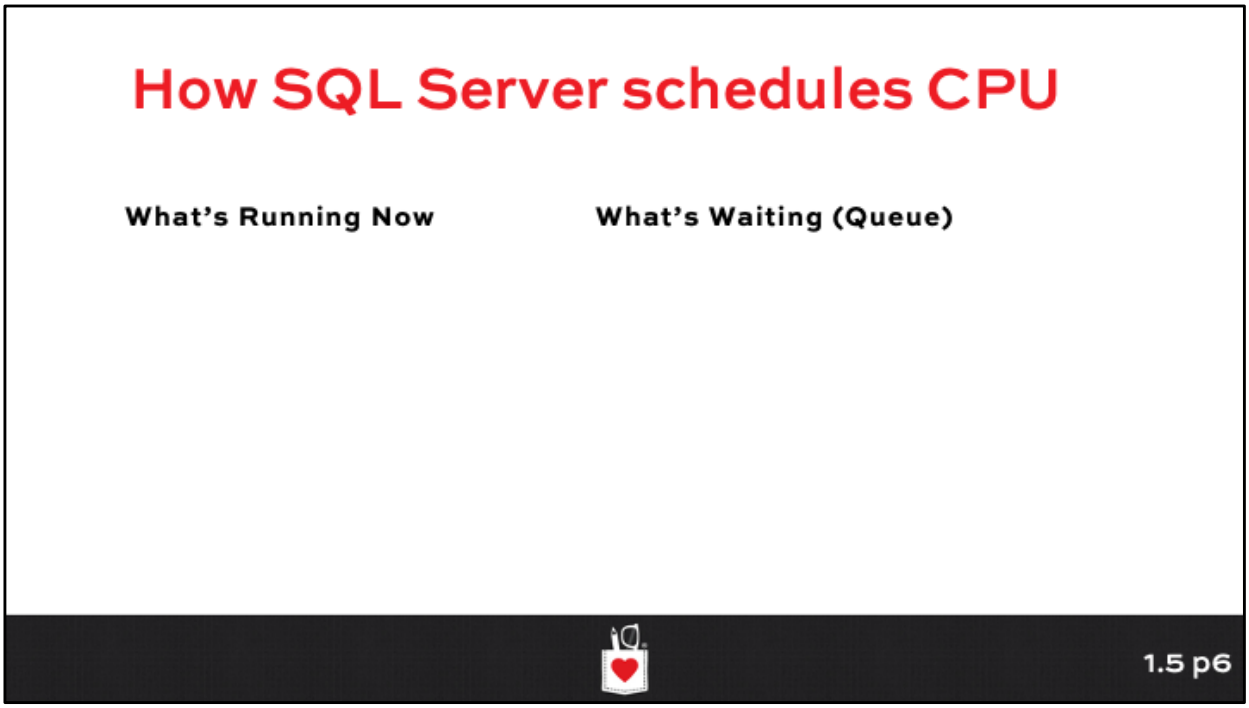
Running a lot of normal queries:
SOS_SCHEDULER_YIELD

Using average wait time to see just how overloaded your CPUs really are

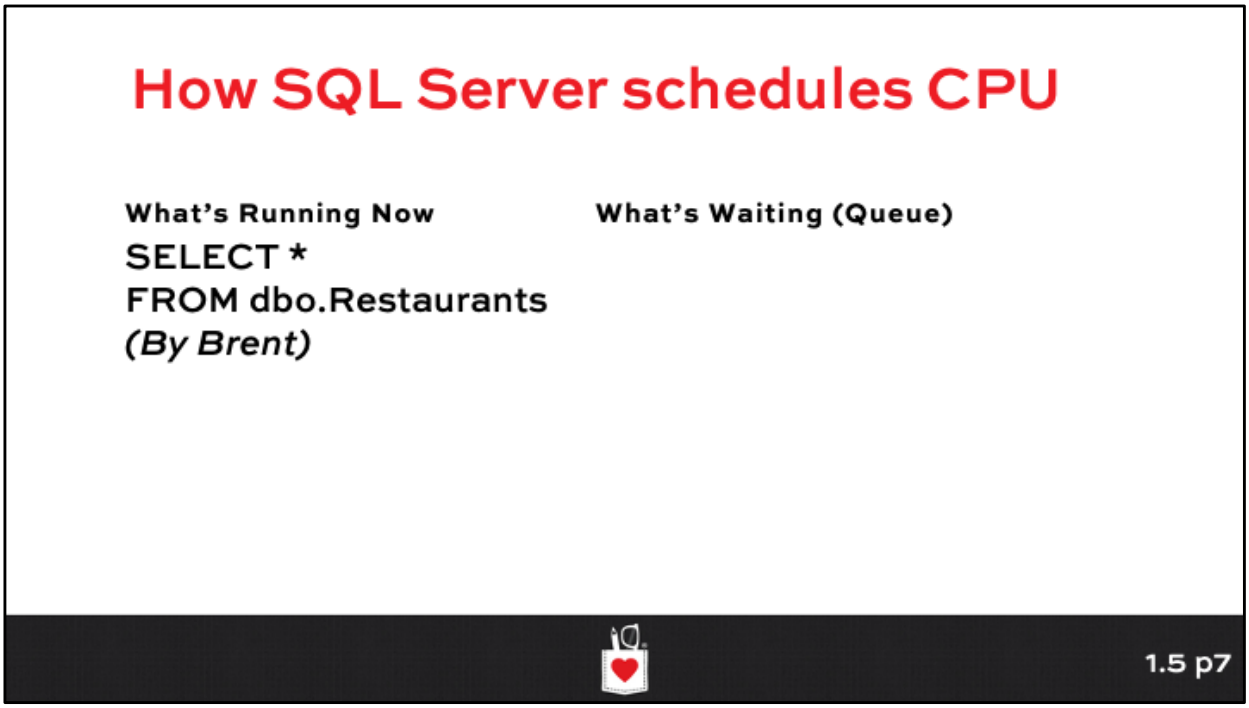
1.5 p4

CPU scheduling with SQLOS

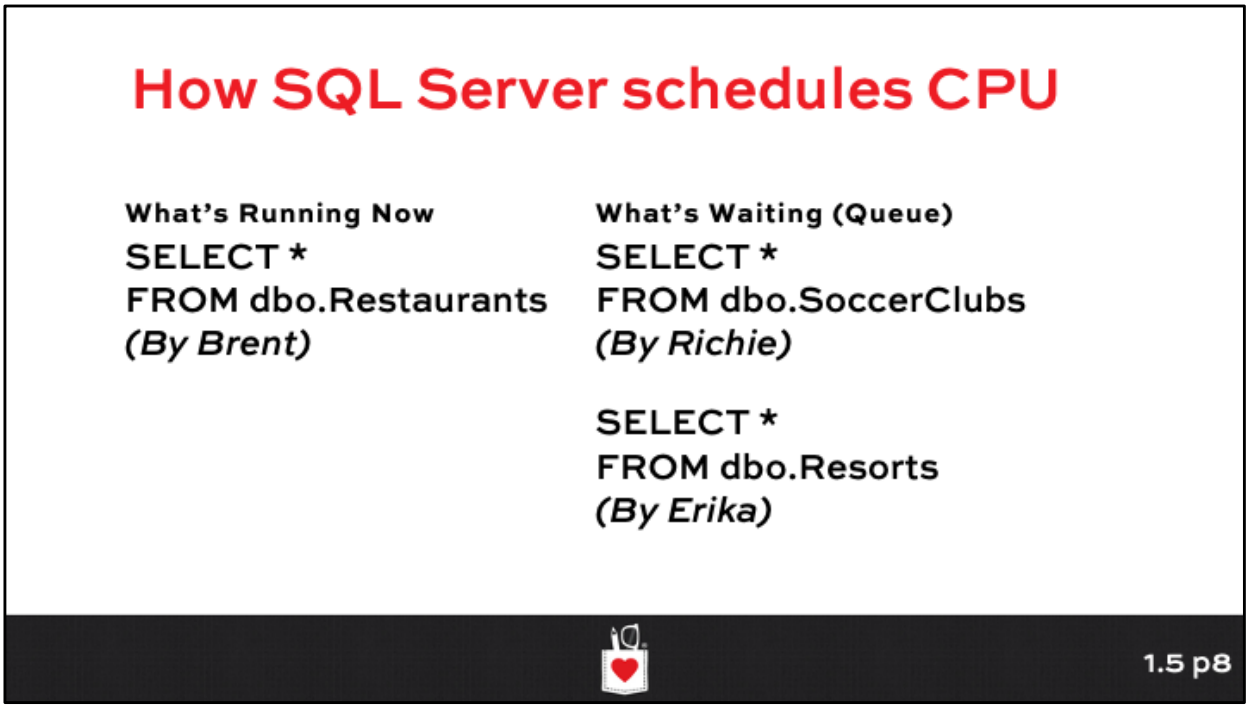
1.5 p5



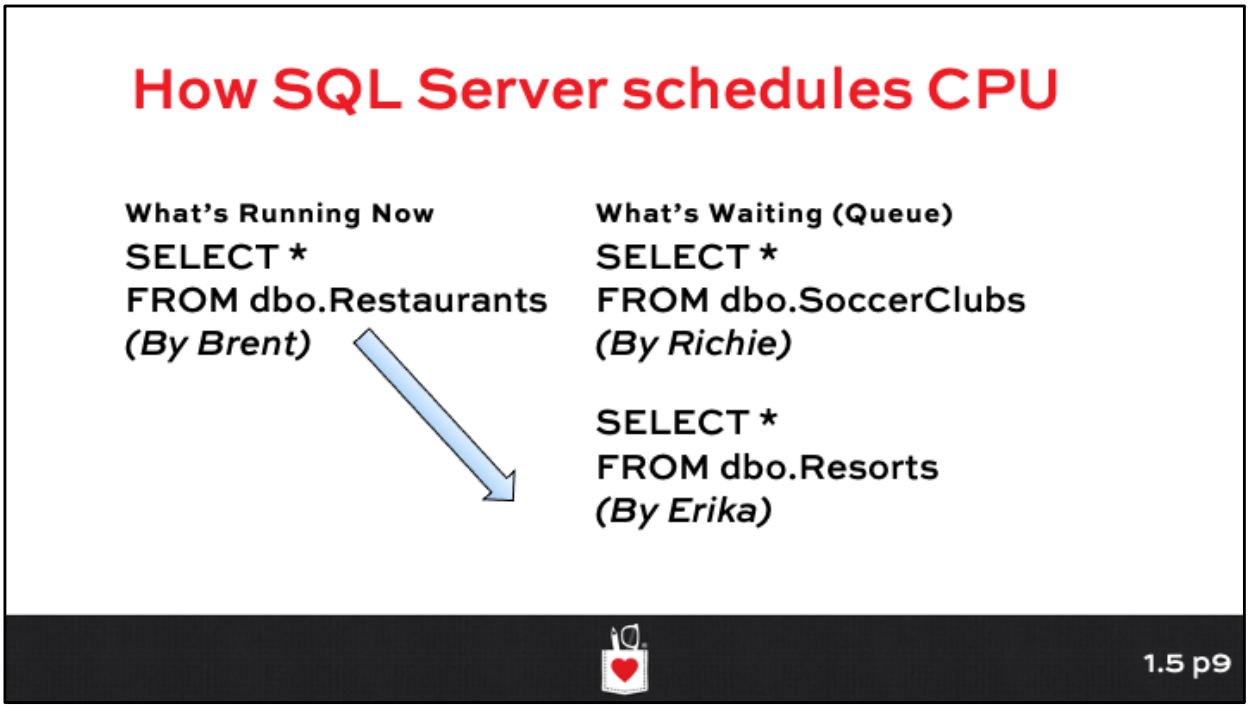
Assume we have the world's smallest SQL Server – it's only got one core.



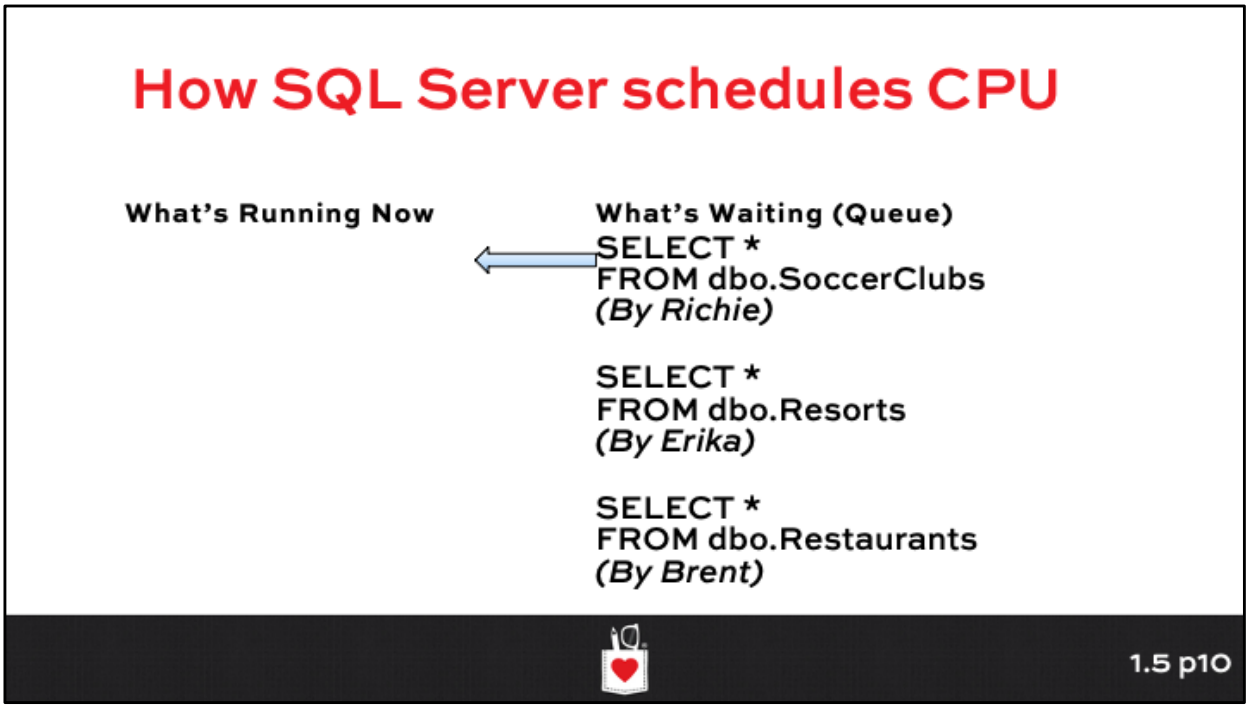
I run a query getting all of the restaurants. My Restaurants table happens to be mostly in cache, so it fires off and starts running. It will KEEP running – there's no concept of sharing CPU cycles in SQL Server. A query runs until it's done, or until it needs to wait on something like locks. More on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



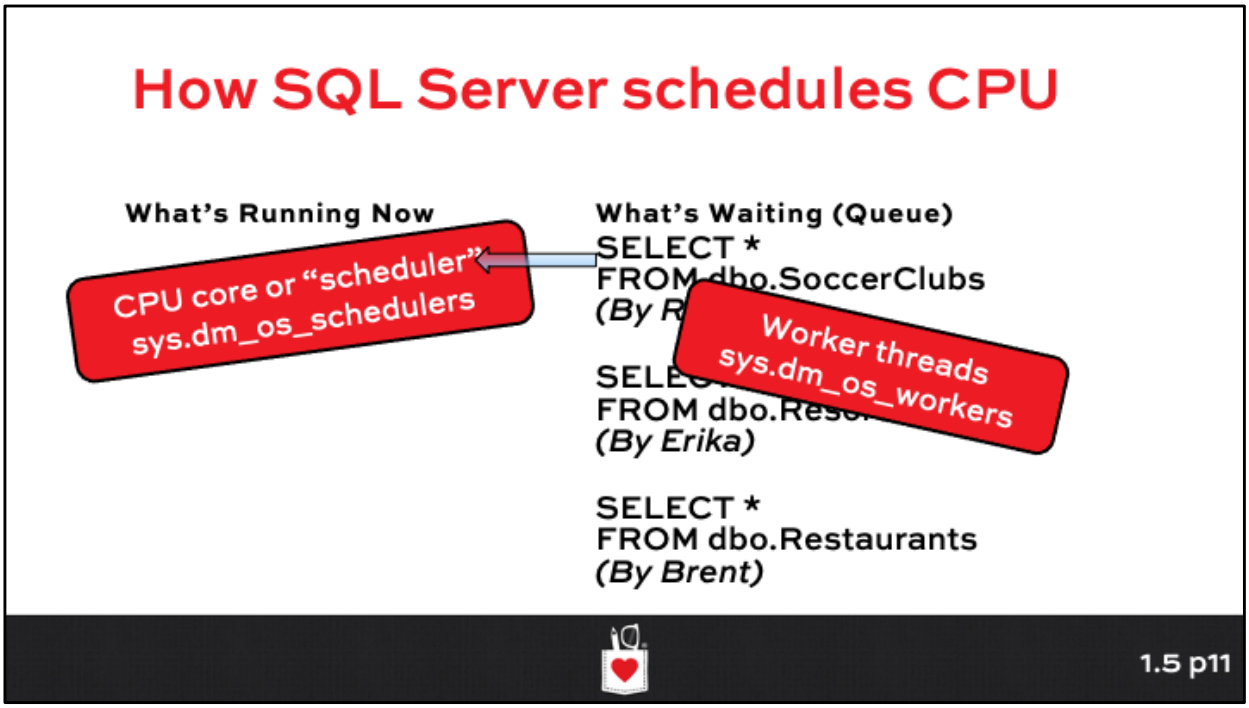
While my query is consuming CPU, other people's queries pile up behind me.



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.



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.

Glossary

CPU core – physical processor in the data center

Scheduler – SQLOS creates one of these per core

Worker thread – SQLOS creates many per scheduler, and manages the swapping in/out of which worker thread is running at a time

1.5 p12

Say we have a 1-CPU server.

We have a single Intel Xeon processor with 4 cores.

(This configuration also works as a virtual machine.)

Intel Xeon Quad-Core CPU #1			
Core #1	Core #2	Core #3	Core #4

1.5 p13

SQLOS sees this config, and:

Intel Xeon Quad-Core CPU #1			
Core #1	Core #2	Core #3	Core #4

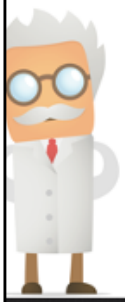
For every one of these cores, SQLOS:

- Creates exactly 1 scheduler for users
- Creates many worker threads for each scheduler

1.5 p14

Default max worker threads

Number of CPUs	32-bit computer	64-bit computer
<= 4 processors	256	512
8 processors	288	576
16 processors	352	704
32 processors	480	960
64 processors	736	1472
128 processors	4224	4480
256 processors	8320	8576

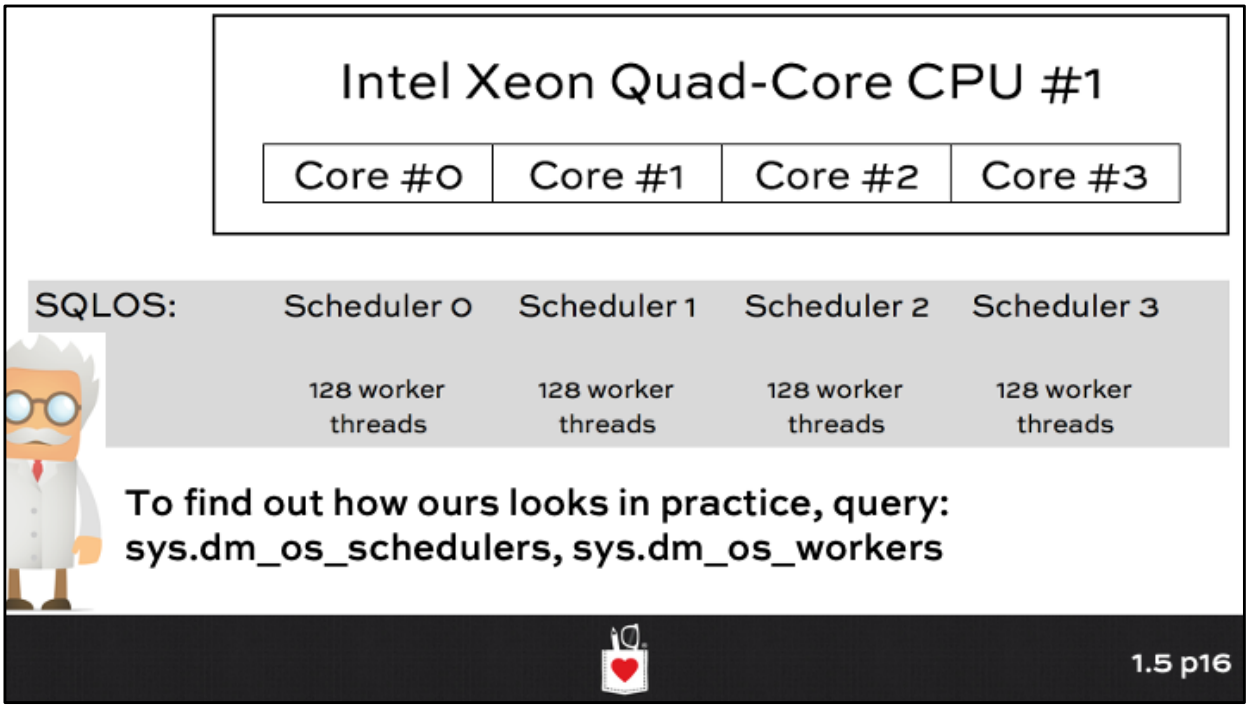


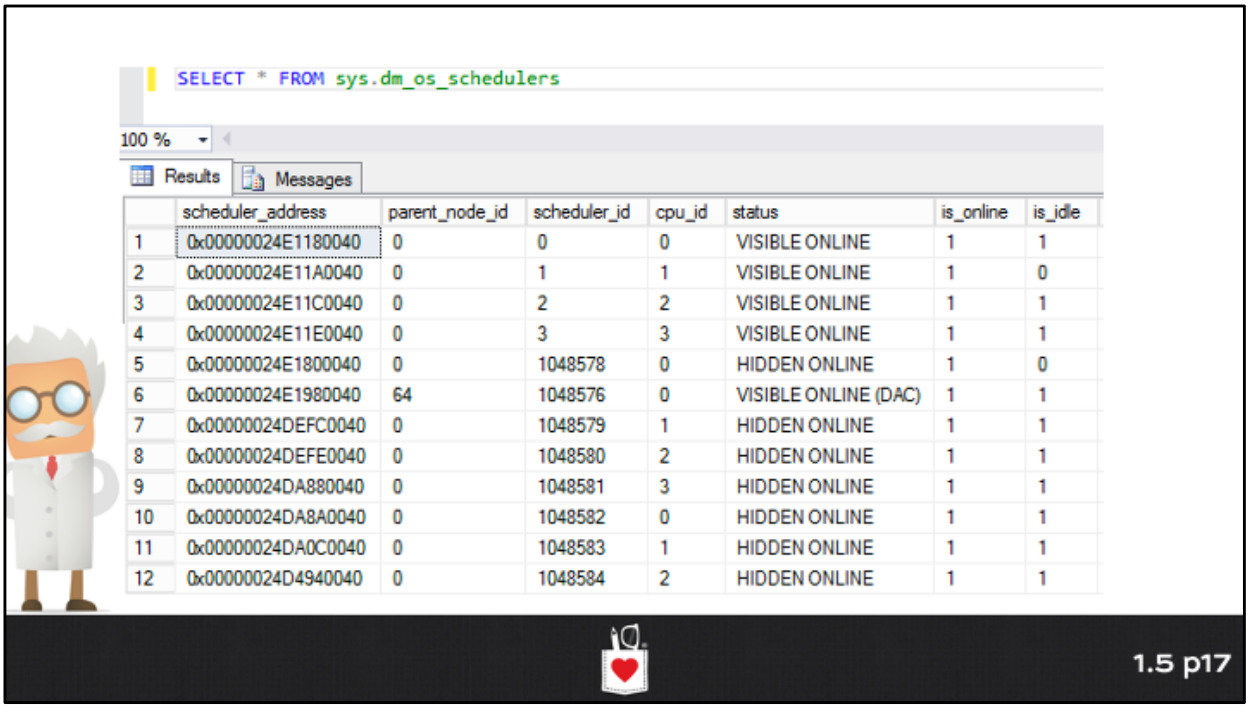
In this Books Online chart, "processors" means logical processors, which means cores.

https://msdn.microsoft.com/en-us/library/ms190219.aspx

So how many worker threads will our 4-core server have with a 64-bit OS?

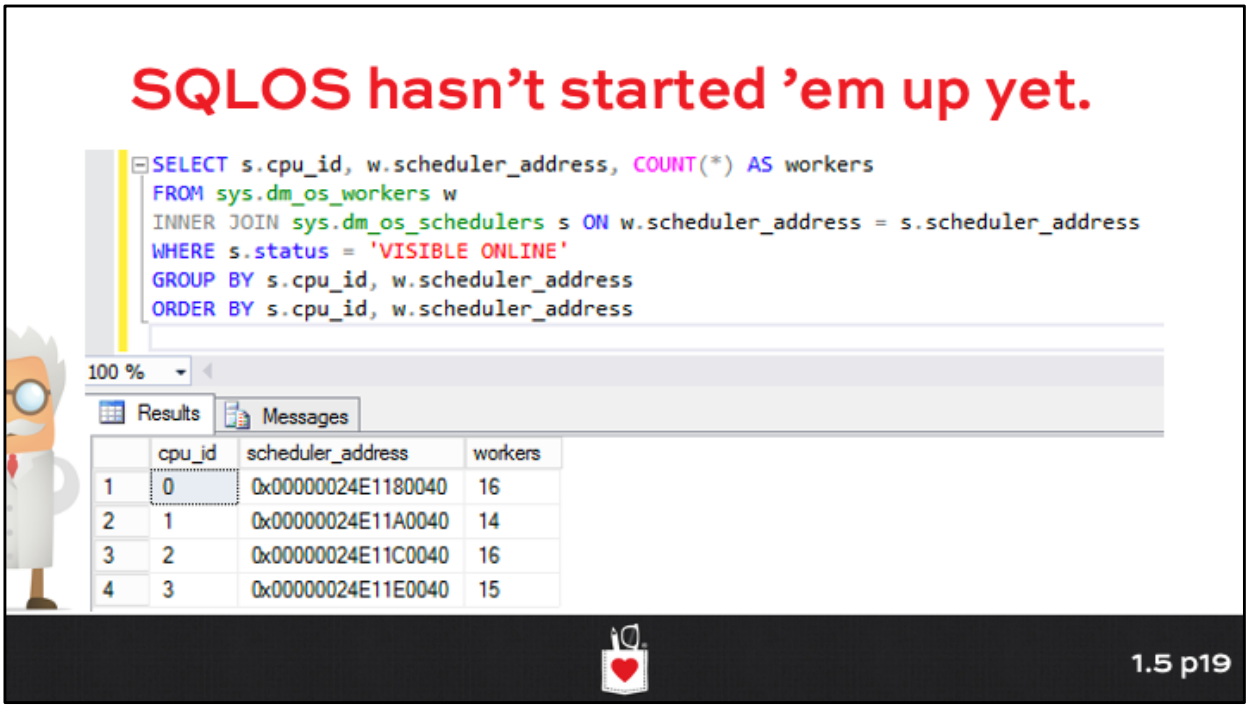
1.5 p15





	scheduler_address	parent_node_id	scheduler_id	cpu_id	status	is_online	is_idle
1	0x00000024E1180040	0	0	0	VISIBLE ONLINE	1	1
2	0x00000024E11A0040	0	1	1	VISIBLE ONLINE	1	0
3	0x00000024E11C0040	0	2	2	VISIBLE ONLINE	1	1
4	0x00000024E11E0040	0	3	3	VISIBLE ONLINE	1	1
5	0x00000024E1800040	0	1048578	0	HIDDEN ONLINE	1	0
6	0x00000024E1980040	64	1048576	0	VISIBLE ONLINE (DAC)	1	1
7	0x00000024DEFC0040	0	1048579	1	HIDDEN ONLINE	1	1
8	0x00000024DEFE0040	0	1048580	2	HIDDEN ONLINE	1	1
9	0x00000024DA880040	0	1048581	3	HIDDEN ONLINE	1	1
10	0x00000024DA8A0040	0	1048582	0	HIDDEN ONLINE	1	1
11	0x00000024DA0C0040	0	1048583	1	HIDDEN ONLINE	1	1
12	0x00000024D4940040	0	1048584	2	HIDDEN ONLINE	1	1

	worker_address	status	is_preemptive	is_fiber	is_sick	is_in_cc_exception	is_fatal_exception	is_inside_catch	is_in_polling_io_completion_routine
1	0x00000024E1020160	2	0	0	0	0	0	0	0
2	0x00000024E1022160	4	1	0	0	0	0	0	0
3	0x00000024E102A160	2	0	0	0	0	0	0	0
4	0x00000024E1280160	4	1	0	0	0	0	0	0
5	0x00000024E1032160	2	0	0	0	0	0	0	0
6	0x00000024E1300160	4	1	0	0	0	0	0	0
7	0x00000024E103A160	2	0	0	0	0	0	0	0
8	0x00000024E1380160	2097156	1	0	0	0	0	0	0
9	0x00000024E1382160	4	1	0	0	0	0	0	0
10	0x00000024E1282160	0	0	0	0	0	0	0	0
11	0x00000024E13D0160	0	0	0	0	0	0	0	0
12	0x00000024E1070160	0	0	0	0	0	0	0	0



Let's run a normal query.

1.5 p20

The World's Tiniest Load Test

Set your parallelism settings back to the default:

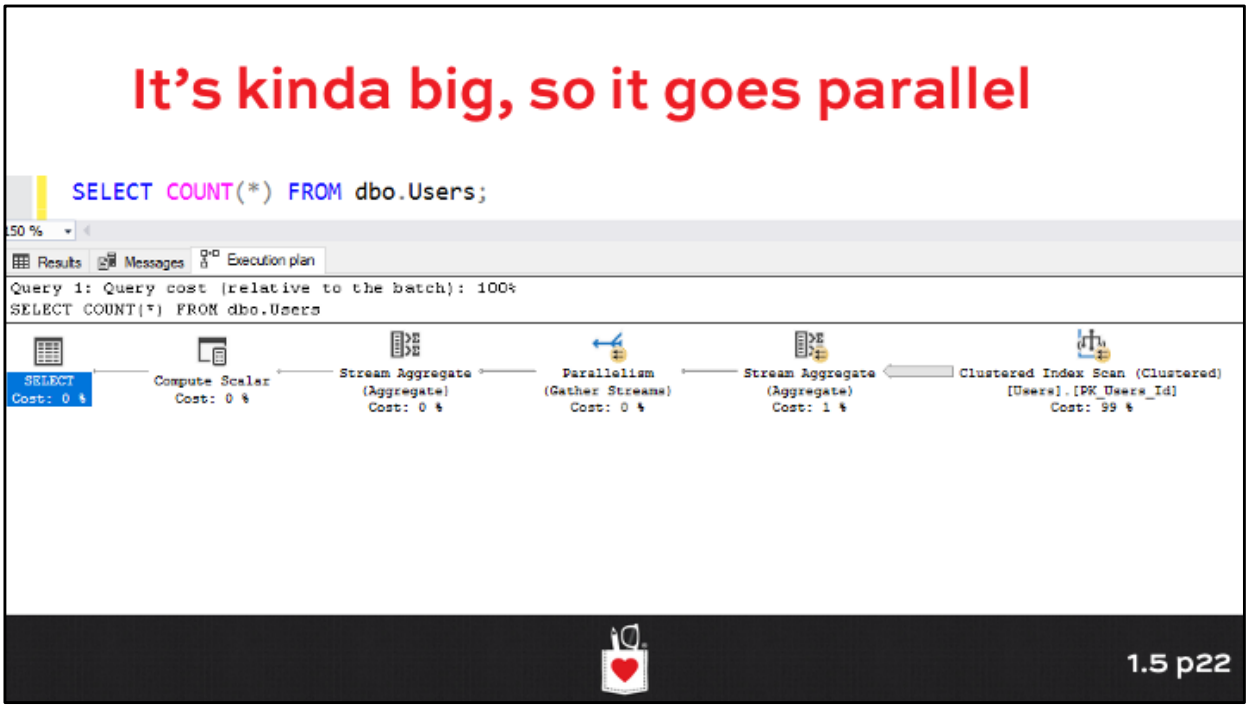
- Cost Threshold for Parallelism = 5
- MAXDOP = 0 (unlimited)

Remove any indexes on the Users table

Run this:

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

It's kinda big, so it goes parallel



If we time it right, we see workers

```
SELECT w.scheduler_address, w.worker_address, w.thread_address, w.state, w.last_wait_type
FROM sys.dm_os_workers w
INNER JOIN sys.dm_os_schedulers s ON w.scheduler_address = s.scheduler_address
WHERE s.status = 'VISIBLE ONLINE'
AND w.state = 'RUNNING'
ORDER BY w.scheduler_address
```

100 %

Results | Messages

	scheduler_address	worker_address	thread_address	state	last_wait_type
1	0x00000024E1180040	0x00000024E1022160	0x00000022EC4FED78	RUNNING	XE_DISPATCHER_WAIT
2	0x00000024E1180040	0x00000024C8282160	0x00000027F29126B0	RUNNING	SOS_SCHEDULER_YIELD
3	0x00000024E11A0040	0x00000024E1280160	0x00000022EC4FEFE8	RUNNING	MEMORY_ALLOCATION_EXT
4	0x00000024E11A0040	0x00000024D3962160	0x00000027F2911948	RUNNING	SOS_SCHEDULER_YIELD
5	0x00000024E11C0040	0x00000024C87B8160	0x00000027F2912440	RUNNING	RESERVED_MEMORY_ALLOCATION
6	0x00000024E11C0040	0x00000024E130E160	0x00000022EC500230	RUNNING	MEMORY_ALLOCATION_EXT
7	0x00000024E11C0040	0x00000024E1300160	0x00000022EC4FF258	RUNNING	MEMORY_ALLOCATION_EXT
8	0x00000024E11E0040	0x00000024E1380160	0x00000022EC4FF4C8	RUNNING	SNI_CRITICAL_SECTION
9	0x00000024E11E0040	0x00000024C8586160	0x00000027F29127E8	RUNNING	SOS_SCHEDULER_YIELD

1.5 p23

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)

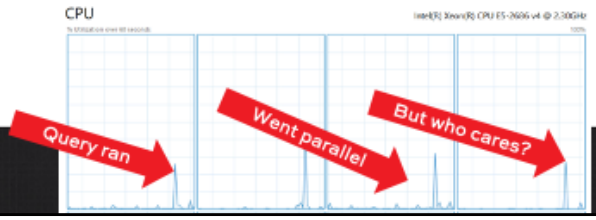
1.5 p24

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.



1.5 p25

SQLOS multi-tasks.

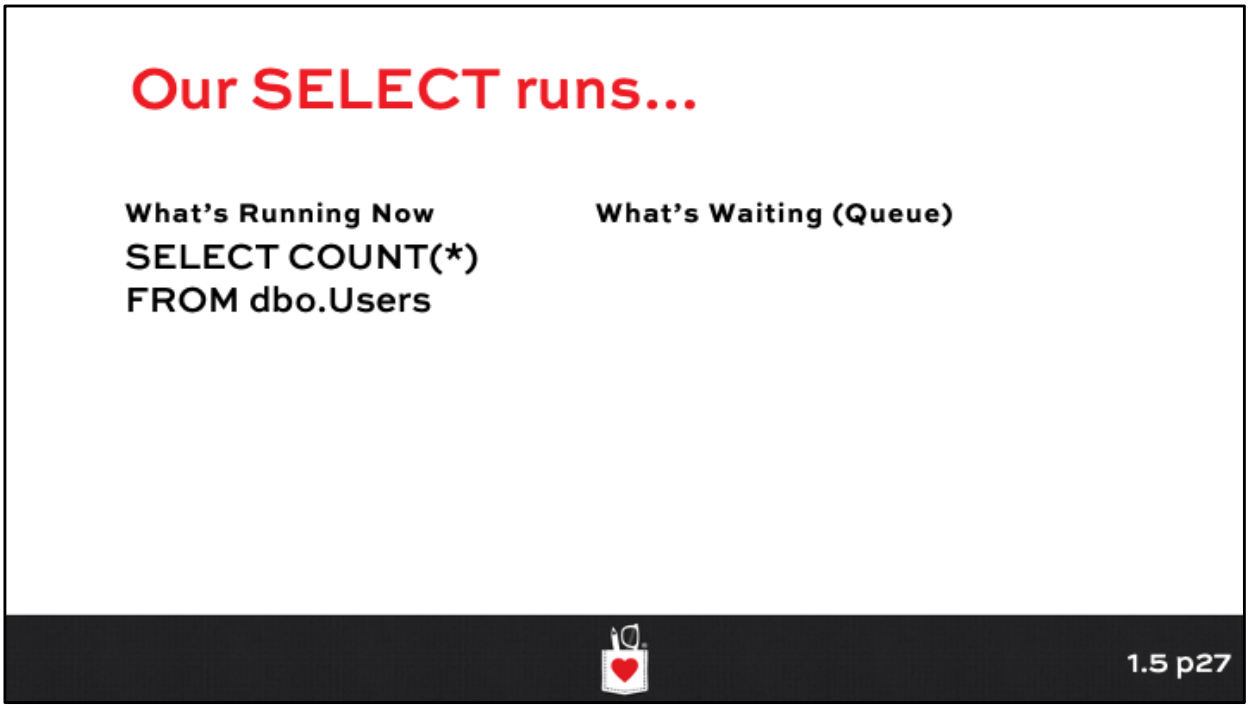
After a task has been running on a scheduler for 4 milliseconds straight, it yields the CPU scheduler.

"Someone else can run now."

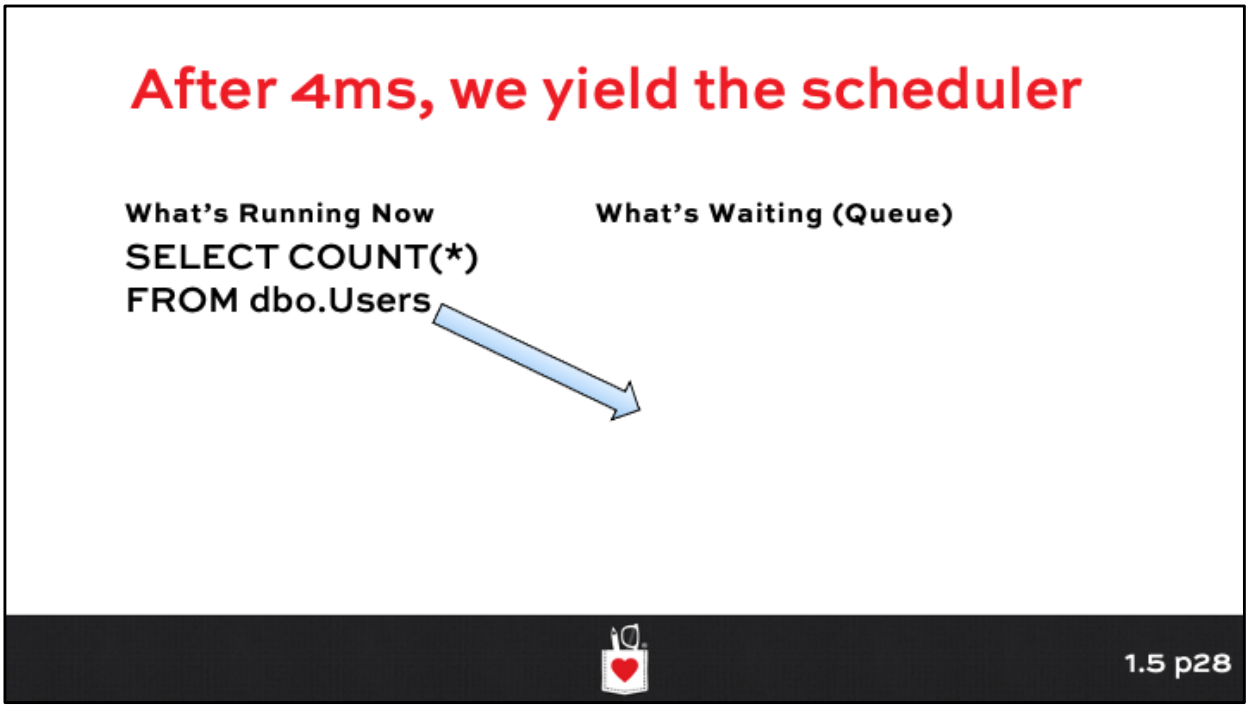
It hops over into the waiting queue – but it will immediately hop back onto the CPU scheduler if it can.

This wait type: SOS_SCHEDULER_YIELD.

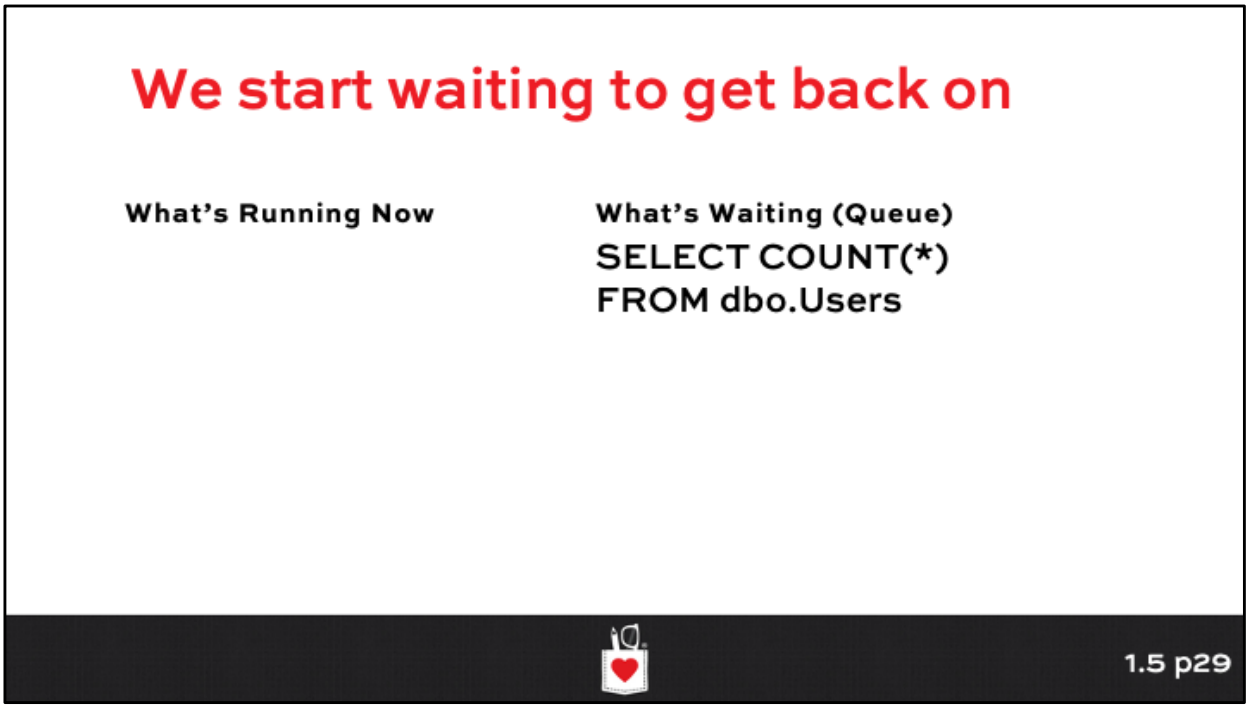
1.5 p26



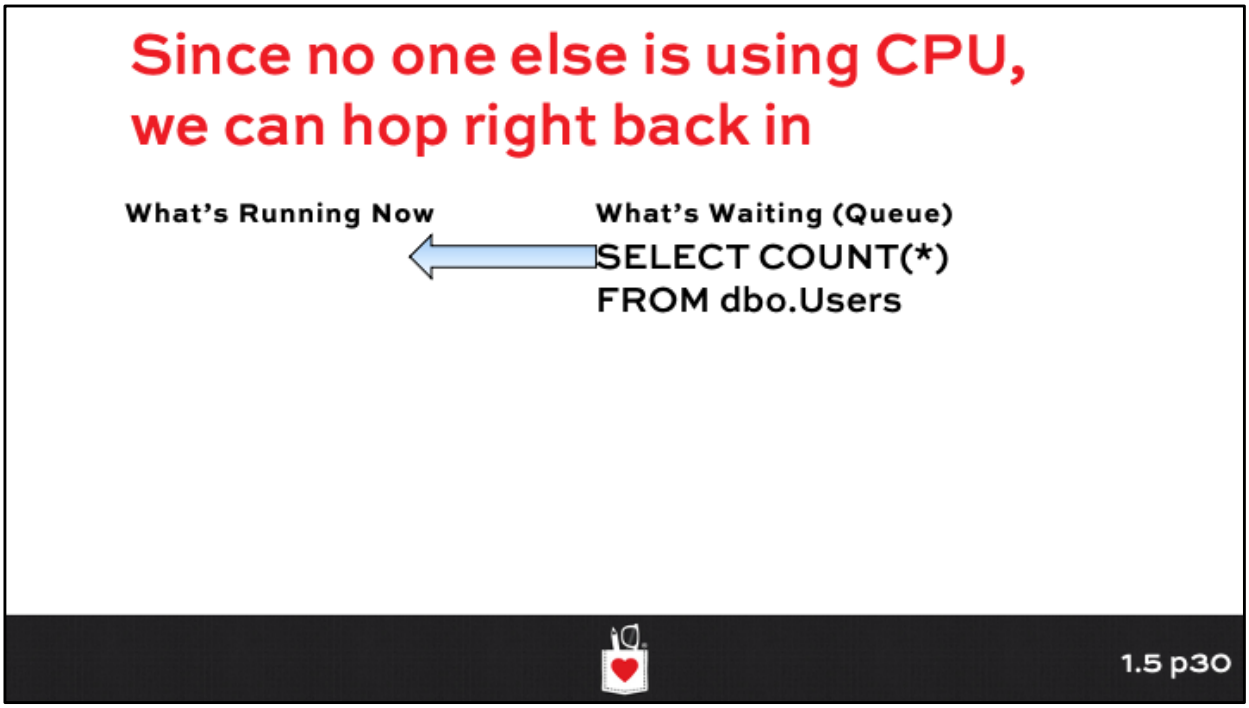
But the instant my query needs something that isn't in cache, like I need to wait for a locked page or I need to wait for something to come back from disk, I go to the back of the line.



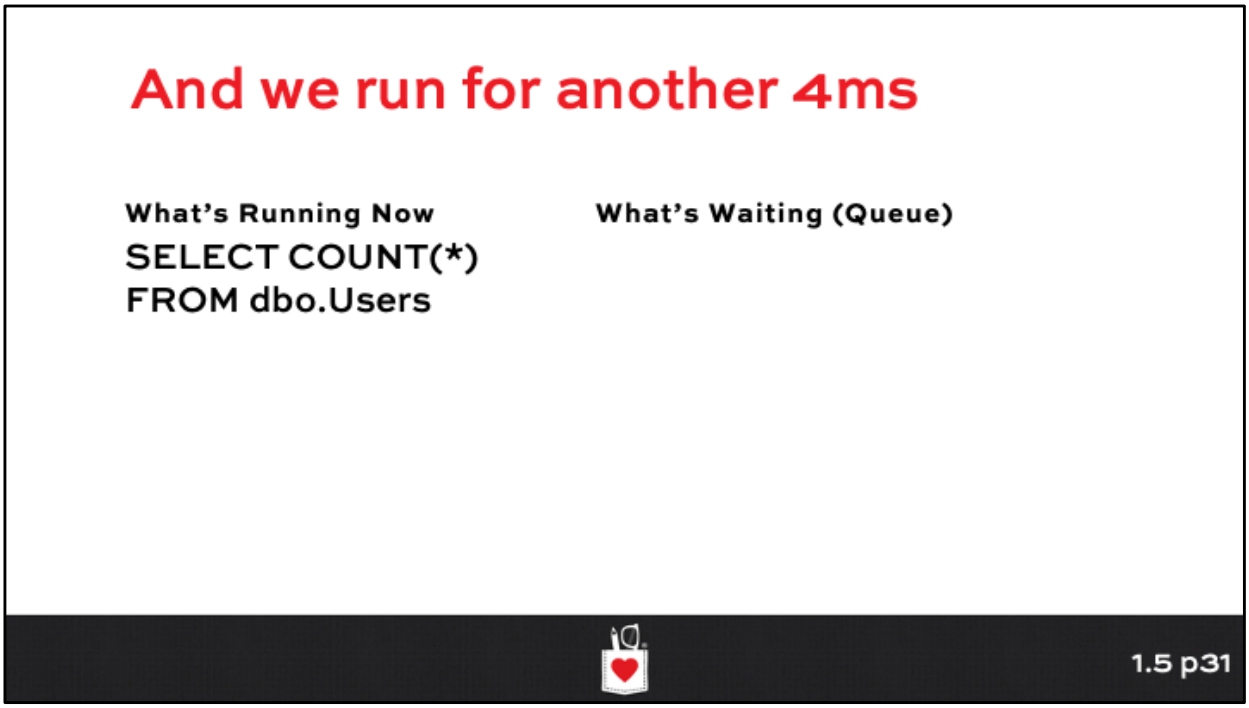
But the instant my query needs something that isn't in cache, like I need to wait for a locked page or I need to wait for something to come back from disk, I go to the back of the line.



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.



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.



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

```
(1 row affected)

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

Teensy

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

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

1.5 p32

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.

1.5 p33

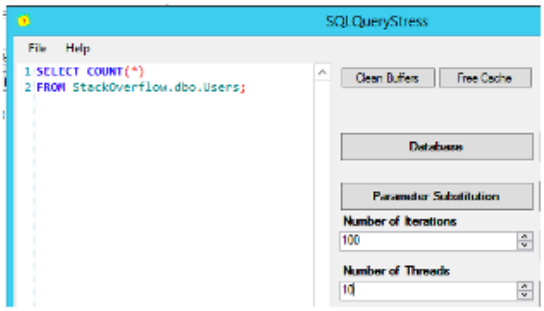
Let's run a lot of normal queries.

1.5 p34

Let's stress test it

Run our query 100 iterations across 10 threads

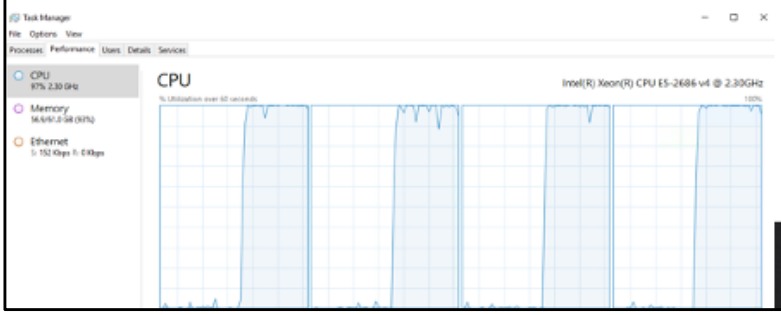
Note: this "threads" refers to SQLQueryStress, not SQL




```
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.



1.5 p36

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

1.5 p37

Before: only about 60 workers

```
SELECT s.cpu_id, w.scheduler_address, COUNT(*) AS workers
FROM sys.dm_os_workers w
INNER JOIN sys.dm_os_schedulers s ON w.scheduler_address = s.scheduler_address
WHERE s.status = 'VISIBLE ONLINE'
GROUP BY s.cpu_id, w.scheduler_address
ORDER BY s.cpu_id, w.scheduler_address
```

100 %

Results | Messages

	cpu_id	scheduler_address	workers
1	0	0x00000024E1180040	16
2	1	0x00000024E11A0040	14
3	2	0x00000024E11C0040	16
4	3	0x00000024E11E0040	15

1.5 p38

Now: amping up to over 100

```
/* Overall workers */
SELECT s.cpu_id, w.scheduler_address, COUNT(*) AS workers
FROM sys.dm_os_workers w
INNER JOIN sys.dm_os_schedulers s ON w.scheduler_address = s.scheduler_address
WHERE s.status = 'VISIBLE ONLINE'
GROUP BY s.cpu_id, w.scheduler_address
ORDER BY s.cpu_id, w.scheduler_address;
```

100 %

Results | Messages

	cpu_id	scheduler_address	workers
1	0	0x00000024E1180040	28
2	1	0x00000024E11A0040	28
3	2	0x00000024E11C0040	27
4	3	0x00000024E11E0040	25

1.5 p39

Medium-load symptoms

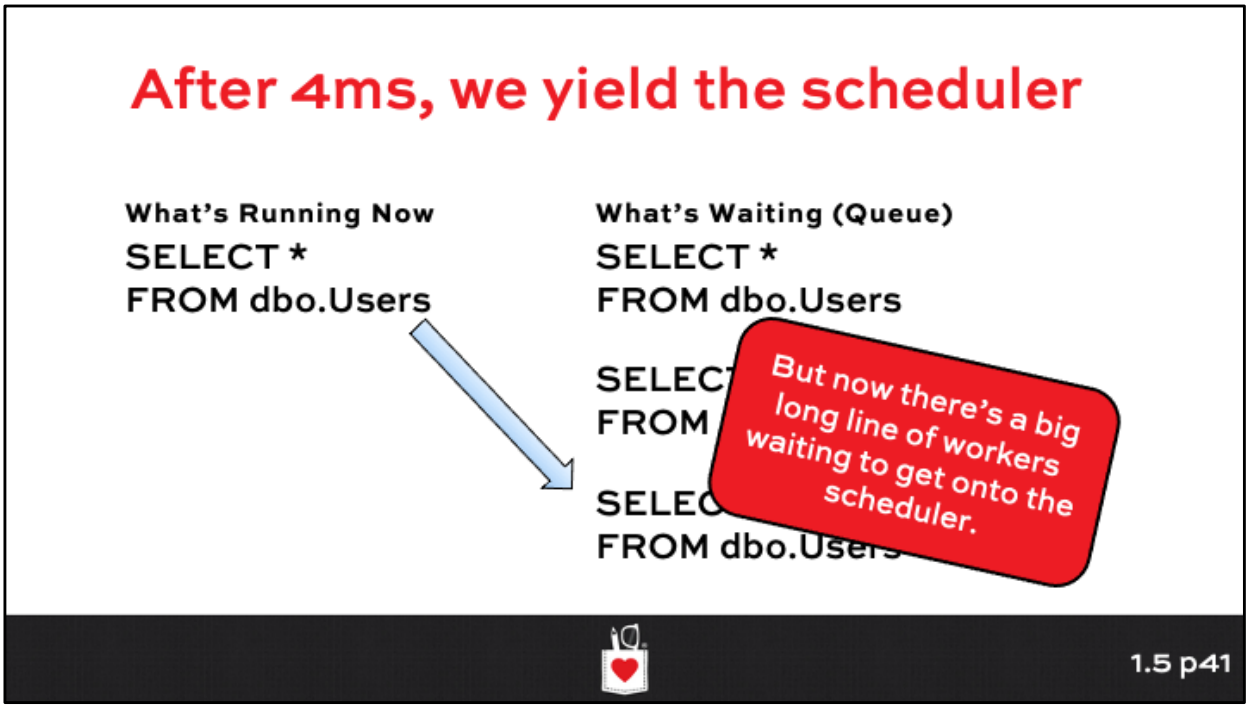
How long does each query take to run?

What do your CPUs look like now?

What are your wait stats?

	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

1.5 p40



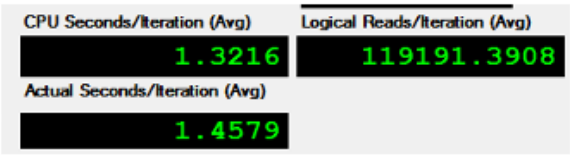
But the instant my query needs something that isn't in cache, like I need to wait for a locked page or I need to wait for something to come back from disk, I go to the back of the line.

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.



1.5 p42

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.

1.5 p44

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?

1.5 p45

Now: hundreds of worker threads

```
/* Overall workers */
SELECT s.cpu_id, w.scheduler_address, COUNT(*) AS workers
FROM sys.dm_os_workers w
INNER JOIN sys.dm_os_schedulers s ON w.scheduler_address = s.scheduler_address
WHERE s.status = 'VISIBLE ONLINE'
GROUP BY s.cpu_id, w.scheduler_address
ORDER BY s.cpu_id, w.scheduler_address;
```

100 %

Results | Messages

	cpu_id	scheduler_address	workers
1	0	0x00000024E1180040	134
2	1	0x00000024E11A0040	135
3	2	0x00000024E11C0040	136
4	3	0x00000024E11E0040	117

1.5 p46

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!

1.5 p47

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.

Fixing SOS_SCHEDULER_YIELD

1.5 p49

3 ways to fix it

1. Tune queries to use less CPU
2. Reduce query compilations
   (we'll cover this in a separate module)
3. Add more CPU power
   (this costs a lot of money & time)

1.5 p50

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.

1.5 p51

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

String processing

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

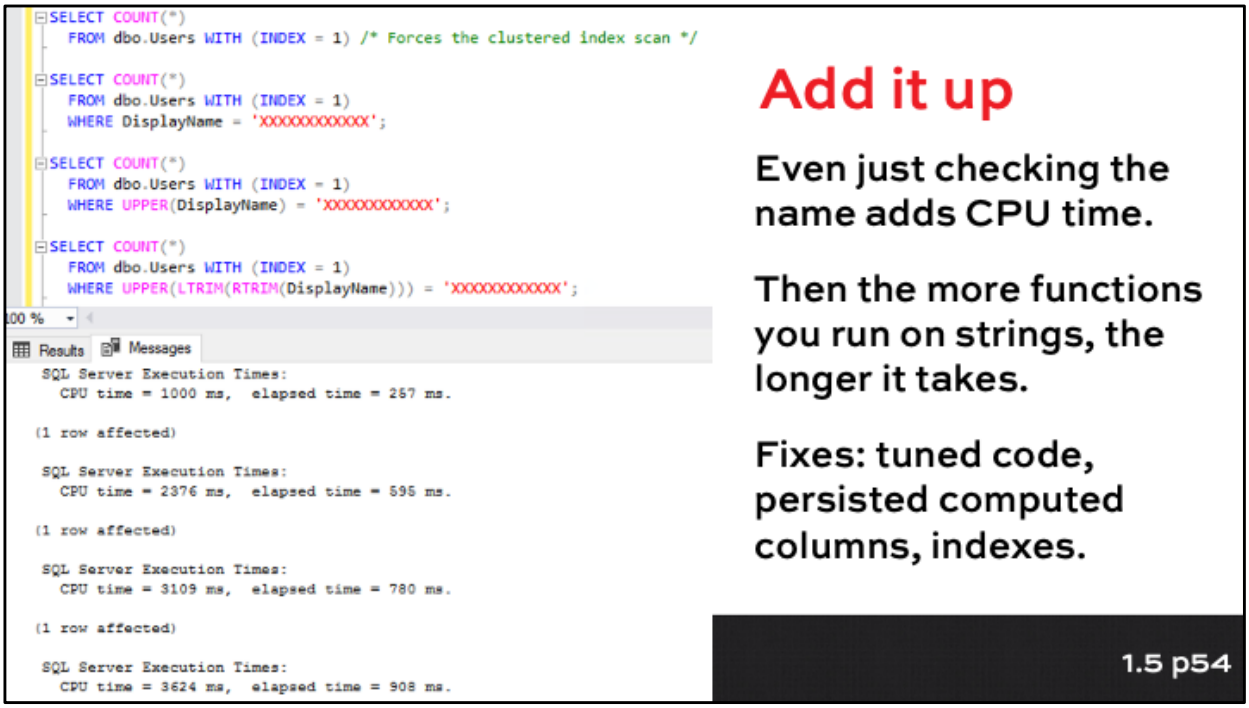
```sql
SELECT COUNT(*)
  FROM dbo.Users WITH (INDEX = 1) /* Forces the clustered index scan */

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';
```

1.5 p53

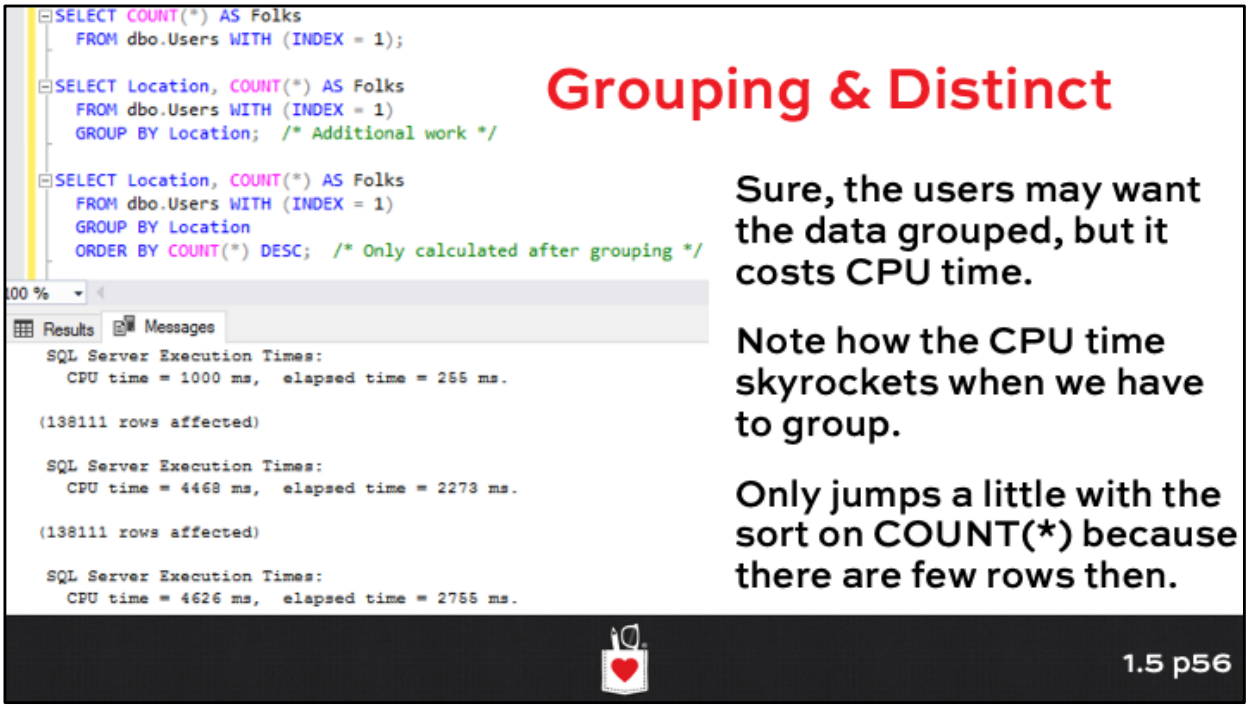


Row-based processing

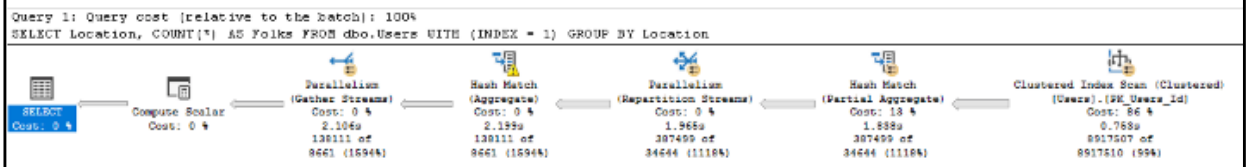
Scalar functions & multi-statement TVFs:
they run for every row. Try inlining them either by rewriting the function, or copy/pasting the code into the calling query.

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

1.5 p55



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!

1.5 p57

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

1.5 p58

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

1.5 p59

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

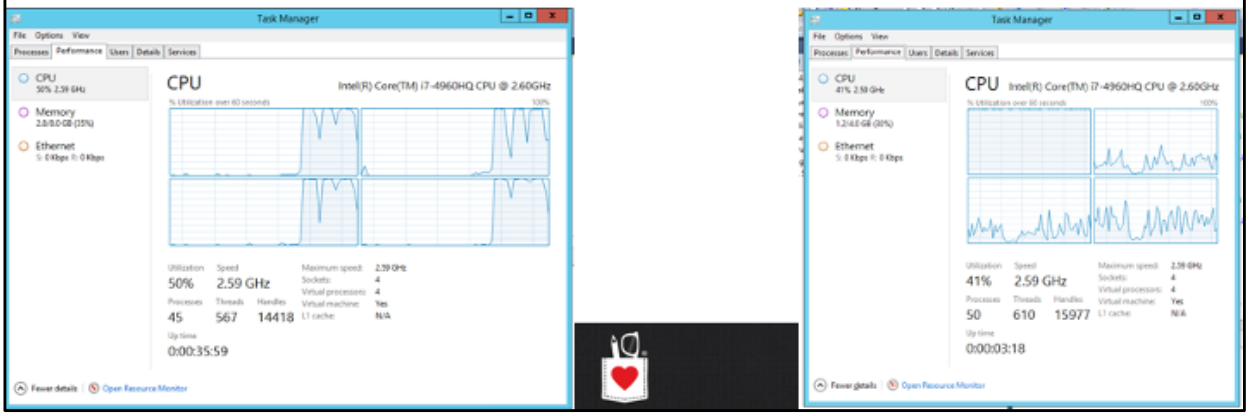
1.5 p60

Do that first. It's cheap. Then...

1. ~~Tune queries to use less CPU~~
2. Reduce query compilations
   (we'll cover this in a separate module)
3. Add more CPU power
   (this costs a lot of money & time)

Adding more CPU power

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

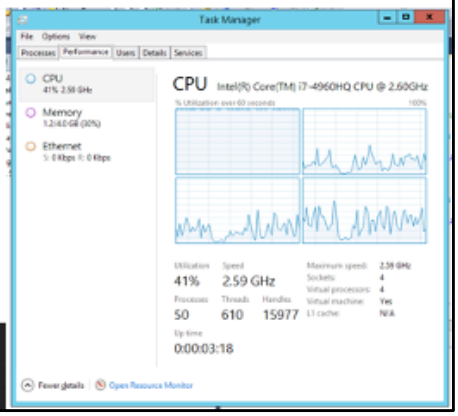


If you're not maxed out...

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

1.5 p64

Example: AWS EC2 .2xlarge types

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

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

API Name	Physical Processor	Clock Speed(GHz)	Windows On Demand cost
.2xlarge	Search	Search	Search
d2.2xlarge	Intel Xeon E5-2676v3 (Haswell)	2.4 GHz	$1.601000 hourly
r4.2xlarge	Intel Xeon E5-2686 v4 (Broadwell)	2.3 GHz	$0.900000 hourly
p3.2xlarge	Intel Xeon E5-2686 v4 (Broadwell)	2.3 GHz	$3.428000 hourly
r3.2xlarge	Intel Xeon E5-2670 v2 (Ivy Bridge)	2.5 GHz	$1.045000 hourly
i3.2xlarge	Intel Xeon E5-2686 v4 (Broadwell)	2.3 GHz	$0.992000 hourly
i2.2xlarge	Intel Xeon E5-2670 v2 (Ivy Bridge)	2.5 GHz	$1.946000 hourly
r5.2xlarge	Intel Xeon Platinum 8175	unknown	$0.872000 hourly
r5a.2xlarge	AMD EPYC 7571	2.5 GHz	$0.820000 hourly
z1d.2xlarge	Intel Xeon Platinum 8151	unknown	$1.112000 hourly
r5ad.2xlarge	AMD EPYC 7571	2.5 GHz	$0.892000 hourly
r5d.2xlarge	Intel Xeon Platinum 8175	unknown	$0.944000 hourly
f1.2xlarge	Intel Xeon E5-2686 v4 (Broadwell)	unknown	unavailable
x1e.2xlarge	High Frequency Intel Xeon E7-8880 v3 (Haswell)	2.3 GHz	$2.036000 hourly

Resources for instance families

AWS: http://ec2instances.info

Azure VMs: http://azureinstances.info
(but doesn't show CPU family & speed yet)

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

1.5 p66

Recap

1.5 p67

When you're facing SOS

After using CPU for 4ms straight, the task yields

Avg ms per wait = how long it has to wait before getting back on CPU again. Longer = worse.

1. Tune queries to use less CPU, but focus on the anti-patterns we covered in this class: sp_BlitzCache @SortOrder = 'cpu'
2. Reduce query compilations (upcoming module)
3. Add more CPU power (last resort)

1.5 p68

Some admins: ¯_(ツ)_/¯

"Our CPU load is high, but SQL still works."

"We use virtualization, and we expect high CPU."

"We paid a lot for these licenses, and we want to use them, not have them sitting around idle."

Hold that thought for the THREADPOOL module.

Setting up for the lab

1. Restart the SQL Server service (clears stats)
2. Restore your StackOverflow database
3. Copy & run the setup script, will take 60-90 seconds: BrentOzar.com/go/serverlab2
4. Start SQLQueryStress:
    1. File Explorer, D:\Labs, run SQLQueryStress.exe
    2. Click File, Open, D:\Labs\ServerLab2.json
    3. Click Go

1.5 p70