



BRENT OZAR
UNLIMITED®

SQL Server Performance Tuning in Google Compute Engine



Author: Erik Darling

Technical Reviewer: Brent Ozar

© 2017 Brent Ozar Unlimited.

Version: v1.0, 2017-03-03. To get the latest version free, visit: <https://BrentOzar.com/go/gce>

Introduction: About You, Us, and This Build

About You

You're a database administrator, Windows admin, or developer. You're building your first SQL Servers in Google Compute Engine, and you're stuck at the create instance screen. How many CPUs should you use? How much memory? How are you supposed to configure storage? Will it be fast enough, and what should you do if it isn't?

Relax. In this white paper, we'll show you:

- How to measure your current SQL Server using data you've already got
- How to size a SQL Server in Google Compute Engine to perform similarly
- After migration to GCE, how to measure your server's bottleneck
- How to tweak your SQL Server based on the performance metrics you're seeing

We're going to be using T-SQL to accomplish a lot of this, but you don't need to be familiar with writing queries in order to follow along.

We'll even let you in on a secret: you don't even have to be using Google Compute Engine in order to learn about SQL Server performance tuning methodologies covered in this white paper. These same techniques can be used for your on-premises VMs and bare metal SQL Servers, too.

All of the resources for this paper are open source, and available for download from our site. This allows you to follow along, run workloads, and make your own tweaks to test different scenarios. You can also make whatever architectural or structural changes you want to the database to more closely fit situations in your database. You can get the resources and more of our Google Compute Engine white papers at <https://BrentOzar.com/go/gce>.

About Us

I'm your tour guide, Erik Darling, and I'll be assisted by Brent Ozar, who will be your technical reviewer for this white paper. That way if the scripts don't work, I can blame it on him.

We're from Brent Ozar Unlimited®, a small boutique consulting firm that makes SQL Server faster and more reliable. You can find more of our work at <https://www.BrentOzar.com>.

Table of Contents

[Introduction: About You, Us, and This Build](#)

[Table of Contents](#)

[How to Measure Your Existing SQL Server](#)

[Trending Backup Size](#)

[Projecting Future Space Requirements](#)

[Trending Backup Speed](#)

[Bonus Section: Backing Up To NUL](#)

[Trending DBCC CHECKDB](#)

[Trending Index Maintenance](#)

[Recap: Your Current Vital Stats](#)

[How to Size Your Google Compute Engine VM](#)

[New Server Challenges](#)

[Choosing Your Instance Type](#)

[GCE's Relationship Between Cores and Memory](#)

[Memory is More Important in the Cloud](#)

[What Does It Mean For You?](#)

[Choosing Your CPU Type](#)

[Putting It All Together: Build, Then Experiment](#)

[How to Measure What SQL Server is Waiting On](#)

[An Introduction to Wait Stats](#)

[How to Get More Granular Wait Stats Data](#)

[Wait Type Reference List](#)

[CPU](#)

[Memory](#)

[Disk](#)

[Locks](#)

[Latches](#)

[Misc](#)

[Always On Availability Groups Waits](#)

[Demo: Showing Wait Stats with a Live Workload](#)

[About Our Server](#)

[About Our Database: Orders](#)

[About Our Workload](#)

[Measuring Our SQL Server with sp_BlitzFirst](#)

[Baseline #1: Waiting on PAGEIOLATCH, CXPACKET, SOS_SCHEDULER_YIELD](#)

[Mitigation #1: Fixing PAGEIOLATCH, SOS_SCHEDULER_YIELD](#)

[Configuring SQL Server to Use Our Newfound Power](#)

[TempDB](#)

[TempDB with one file:](#)

[TempDB with multiple evenly sized data files](#)

[TempDB with multiple uneven data files](#)

[Moving TempDB](#)

[Max Server Memory](#)

[CPU](#)

[Baseline #2: PAGEIOLATCH Gone, SOS_SCHEDULER_YIELD Still Here](#)

[Mitigation #2: Throwing Cores At SOS_SCHEDULER_YIELD Waits](#)

[Baseline #3: High CPU, and Now LCK* Waits](#)

[Mitigation #3: Fixing LCK* Waits with Optimistic Isolation Levels](#)

[Batch Requests / Second](#)

[Wrap It Up](#)

How to Measure Your Existing SQL Server

I bet you have a SQL Server right now. Maybe you're not proud of it - the last guy set it up, and he wasn't particularly competent - but that's a story for another day.

Every day, your SQL Server is doing several load tests for you:

- Backups are load tests of how fast your storage can read from the data files, and write to some other target
- CHECKDB is a load test of storage reads, plus CPU power
- Index rebuilds are load tests of storage reads and writes for the data file

And here's the best part: your SQL Server is already tracking the runtimes for this stuff! We'll show you how to query it out of your system tables.

Sure, our end user queries might be SQL Server's biggest problem, but they're much harder to consistently load test. Let's start with the easiest thing to load test, and then see if we can build a cloud server to at least match that same performance level of our on-premises physical servers. Earthed servers maybe? Someone needs to come up with a better name for physical servers. Maybe "Cloudless". I bet we can make "Cloudless" happen by the end of this whitepaper.

If you're unsure about if you're running backups or DBCC CHECKDB, head on over to firstresponderkit.org to download our free tool sp_Blitz, which will tell you about all sorts of things you'd never think to check happening on your SQL Server.

Trending Backup Size

We're going to do this for two reasons. One is to look at how our data grows over time, and the other is to look at how long it takes to back data up as it grows. It may sound a little wacky, but this is how we'll figure out how much space we need to provision in the cloud to handle our data.

Remember, you don't just need enough disk space to hold your data and log files, you also need it to hold your backups. As data grows over time, you need more space to hold everything.

Add into that any data retention policies, and you may find yourself needing way more drive space than you first anticipated.

We'll walk through how to calculate how much space you'll probably need after we show you the scripts to collect different metrics. We'll also be talking about how fast your current disks are, so you have a good idea about what you'll need to provision elsewhere to get equivalent or better performance.

You know, stuff that sounds important to the people paying the bills.

This query will get you backup sizes over the last year: (dear god this formatting is awful)

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
DECLARE @startDate DATETIME;
SET @startDate = GETDATE();

SELECT PVT.DatabaseName ,
       PVT.[0] ,
       PVT.[-1] ,
       PVT.[-2] ,
       PVT.[-3] ,
       PVT.[-4] ,
       PVT.[-5] ,
       PVT.[-6] ,
       PVT.[-7] ,
       PVT.[-8] ,
       PVT.[-9] ,
       PVT.[-10] ,
       PVT.[-11] ,
       PVT.[-12]
FROM (
    SELECT BS.database_name AS DatabaseName ,
           DATEDIFF(mm, @startDate, BS.backup_start_date) AS MonthsAgo ,
           CONVERT(NUMERIC(10, 1), AVG(BF.file_size / 1048576.0)) AS
AvgSizeMB
    FROM msdb.dbo.backupset AS BS
    INNER JOIN msdb.dbo.backupfile AS BF
        ON BS.backup_set_id = BF.backup_set_id
    WHERE NOT BS.database_name IN ( 'master', 'msdb', 'model', 'tempdb' )
        AND BF.[file_type] = 'D'
        AND BS.type = 'D'
        AND BS.backup_start_date BETWEEN DATEADD(yy, -1, @startDate) AND
@startDate
    GROUP BY BS.database_name ,
             DATEDIFF(mm, @startDate, BS.backup_start_date)
) AS BCKSTAT PIVOT ( SUM(BCKSTAT.AvgSizeMB) FOR BCKSTAT.MonthsAgo IN ( [0],
[-1], [-2], [-3], [-4], [-5], [-6], [-7], [-8], [-9], [-10], [-11], [-12] ) )
AS PVT
ORDER BY PVT.DatabaseName;
```

You should get query results that look similar to this, assuming you have a year of backup history to examine. If you don't, just take what you have. Unless it's because you're not taking backups.

DatabaseName	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12
LogShipMe	2952.0	2824.0	2632.0	2472.0	2376.0	1858.7	1672.0	1480.0	1192.0	957.3	776.0	612.0	450.7

This data trends easily because we're writing a whitepaper with a test database. It's also only about a week old, and I had to use a really stupid looking query to trend data growth over that period. You don't want that, you want a higher level look. Trust me.

Our 'database' grew by about 150 MB a 'month'. We have a 3 GB database now! They grow up so fast! This should help you figure out *how much* disk space you need to buy to account for your SQL data. For backups, you need to do a little extra math to account for data retention policies, and different backup types.

For instance, if you have a policy to keep 2 weeks of data, you may be taking daily or weekly fulls, diffs at some more frequent interval (4/6/8 hours), and transaction logs at another interval (generally 60 minutes or less or else *SERIOUSLY WHY ARE YOU IN FULL RECOVERY?*). You'll need to look at current disk space usage for that data, and pad it upwards to account for data growth over time.

Projecting Future Space Requirements

Projecting space needed for, and growth of data backup files is not an exact science, and there are many factors to consider.

- Do you expect user counts to go up more than previously? (more users = more data)
- Are you expecting to onboard more new clients than previously?
- Are you adding new functionality to your application that will need to store additional data?
- Are you expecting to onboard some number of large clients who are bringing a lot of historical data?
- Are you changing your data retention periods?
- Are you going to start archiving/purging data?

If you're expecting reasonably steady growth, you can just take your current backup space requirements, and set up enough additional space to comfortably hold data **one year from now**. That looks something like *current backups + current data * percent growth per month * 12 months*.

If you have 100 GB of backups and data, and a steady growth rate of 10% per month, you'll need roughly 281 GB in a year. In two years, it's nearly a terabyte. Padding the one year estimate up to 300 GB isn't likely to have an appreciable impact on costs, but remember that

with GCE storage, size and performance are tied together. Larger disks have higher read and write speeds, up to a certain point.

For less predictable rates, or special-circumstance rates, you'll have to factor in further growth or reduction to suit your needs.

For instance, if you have 5 years of data in your application, and you're planning a one time purge to reduce it to two years, then *only* keeping two years of live data online, you'll want to start your calculation based on the size of two years of data.

You may be moving to the cloud because you're expecting to onboard several large customers, and the up front hardware/licensing costs, plus the costs of implementing more tolerant HA/DR to satisfy new SLAs is too high.

In some of these cases, clients will be bringing hundreds of gigs, possibly terabytes of historical data with them. Do your best to collect information about current size of data and data growth. If they're already using SQL Server, you can send them the queries in this whitepaper to run so you have a good 1:1 comparison.

Trending Backup Speed

The next thing we want to look at is *how long* backups take. Cutting out a chunk of the script above, we can trend that as well, along with the avg MB per second backup throughput you're getting currently.

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
DECLARE @startDate DATETIME;
SET @startDate = GETDATE();

SELECT BS.database_name,
       CONVERT(NUMERIC(10, 1), BF.file_size / 1048576.0) AS SizeMB,
       DATEDIFF(SECOND, BS.backup_start_date, BS.backup_finish_date) as
BackupSeconds,
       CAST(AVG(( BF.file_size / ( DATEDIFF(ss, BS.backup_start_date,
BS.backup_finish_date) ) / 1048576 )) AS INT) AS [Avg MB/Sec]
FROM msdb.dbo.backupset AS BS
     INNER JOIN msdb.dbo.backupfile AS BF
         ON BS.backup_set_id = BF.backup_set_id
WHERE NOT BS.database_name IN ( 'master', 'msdb', 'model', 'tempdb' )
     AND BF.[file_type] = 'D'
     AND BS.type = 'D'
     AND BS.backup_start_date BETWEEN DATEADD(yy, -1, @startDate) AND
@startDate
GROUP BY BS.database_name, CONVERT(NUMERIC(10, 1), BF.file_size / 1048576.0),
DATEDIFF(SECOND, BS.backup_start_date, BS.backup_finish_date)
```



```
ORDER BY CONVERT(NUMERIC(10, 1), BF.file_size / 1048576.0);
```

You should get results that look something like this, assuming you have some backup history in msdb. This will tell you, for each database, how big it was in MB, how long the full backup took, and the average MB/sec.

database_name	SizeMB	BackupSeconds	Avg MB/Sec
LogShipMe	264.0	5	52
LogShipMe	584.0	15	38
LogShipMe	1928.0	49	39
LogShipMe	3208.0	79	40

Keep in mind, this is a GCE VM with a 500 GB Standard Persisted disk, on a n1-standard-4 (4 CPUs, 15 GB memory). We're reading from and writing to the same drive, and this speed is terrible. How terrible? About USB 2.0 terrible. Don't do this to yourself.

On the next page is a reference chart of current transfer speeds. At the speeds we got, that's subpar for even USB 2.0 connected storage.

Welcome to your first lesson in the cloud: cheap disks aren't good, and good disks aren't cheap. Unless you provision **local** SSDs at startup, which we'll discuss the pros and cons of later, are around 250 MB/s for reads, and 100 MB/s for writes.

That puts you somewhere in the neighborhood of 1 Gb iSCSI. Time to get started testing your local storage with a tool like [Crystal Disk Mark](#) or [DiskSpd](#). If you're already getting better speeds in your Cloudless architecture, you need to know that going into a cloud implementation exercise, and how to mitigate where possible.

If you're getting lower or equivalent disk speeds, time to start looking at how much it will cost to get similar performance on Someone Else's Computer®. This is one of those things that's "important" to people with "budgets" and whatever.

You ever meet one of them? Real squares.

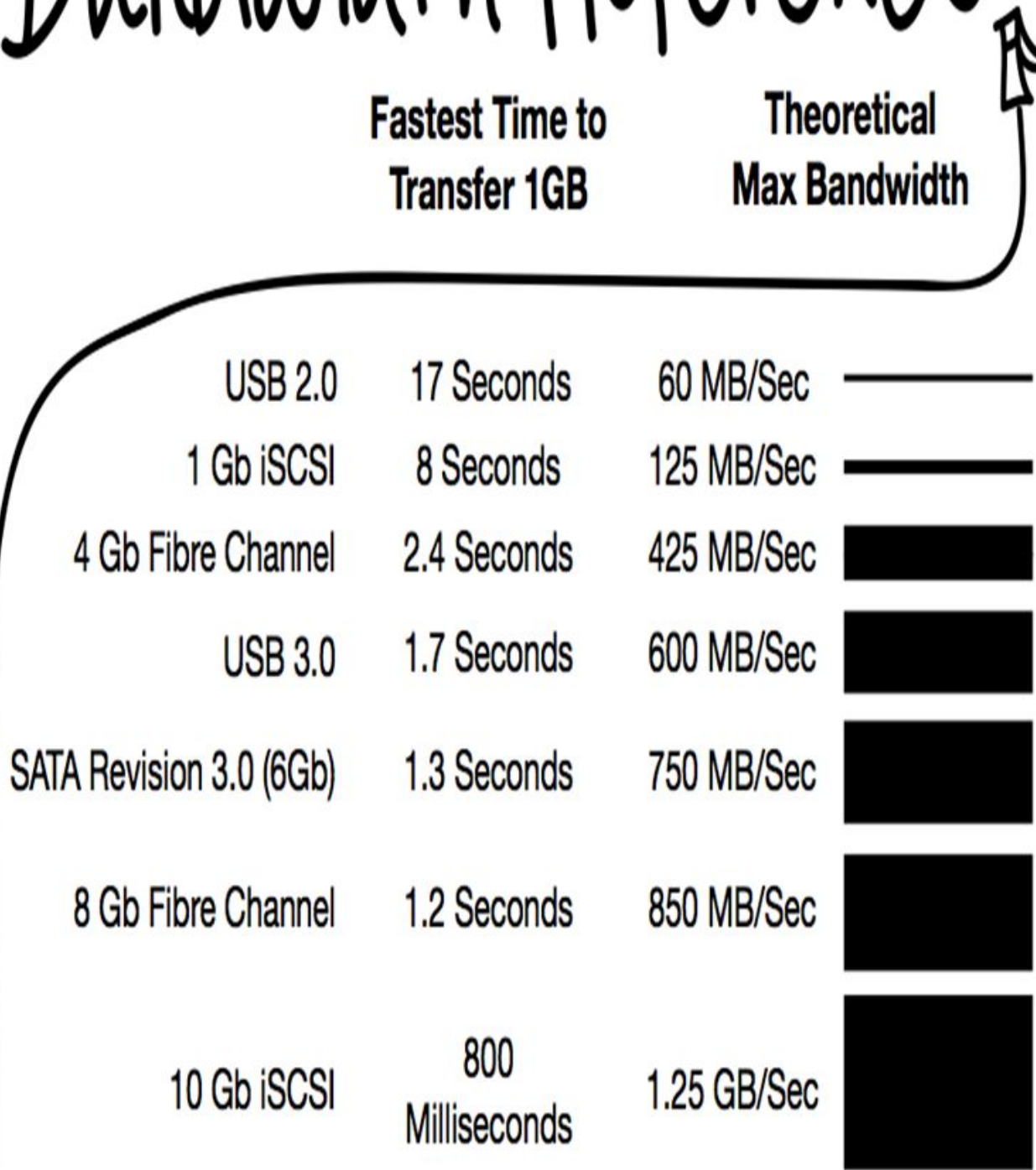
But they pay your salary, too, and one of the best ways to make sure you keep getting those paychecks is to not to go way over budget on this new-fangled cloud thing.



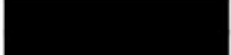
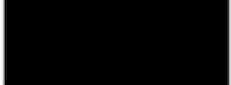


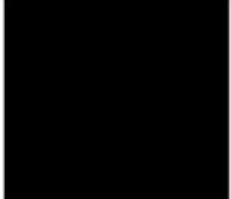
All the money you promised you'd save them over staying Cloudless and hiring some professional curmudgeons (DBAs) can quickly evaporate under the provisioning of many terabytes of cloud storage.

Bandwidth Reference

**Fastest Time to
Transfer 1GB**

**Theoretical
Max Bandwidth**



USB 2.0	17 Seconds	60 MB/Sec	
1 Gb iSCSI	8 Seconds	125 MB/Sec	
4 Gb Fibre Channel	2.4 Seconds	425 MB/Sec	
USB 3.0	1.7 Seconds	600 MB/Sec	
SATA Revision 3.0 (6Gb)	1.3 Seconds	750 MB/Sec	
8 Gb Fibre Channel	1.2 Seconds	850 MB/Sec	
10 Gb iSCSI	800 Milliseconds	1.25 GB/Sec	

This will be important to test in the cloud. The time it takes to backup and restore data has [implications](#) for RPO and RTO, as well as defined maintenance windows. The first thing you'll want to do when you get your database up in The Cloud from your Cloudless architecture is take a backup and time it against current.

Backup settings to consider:

- Compression
- Splitting the backup into multiple files
- Snapshot backups (with a 3rd party tool, if native backups aren't fast enough)

Those settings can tune your number, and you may want to work on that first before going to the cloud. Otherwise, your big takeaway from this section: know your backup throughput in MB/sec.

Bonus Section: Backing Up To NUL

If you want to get a rough idea how fast SQL can read data from disk during a backup, you can use a special command to backup to nowhere.

This is called “backing up to NUL”, or “not really taking a backup” or “if you make this part of your regular backup routine so help me God I will show up at your house and stare at you balefully while you sleep”.

```
BACKUP DATABASE YourDatabase  
TO DISK = 'NUL'  
WITH COPY_ONLY
```

Like I said, this backs up to nowhere. You will not have a backup from this. But you will know how fast you can read data from disk without the overhead of writing it to disk.

Yay.

I guess. But danger, Will Robinson - make sure to use the COPY_ONLY parameter, lest this backup to nowhere break your differential backup chain.

Trending DBCC CHECKDB

You [check your databases for corruption](#), right? I mean, you're a conscientious person who doesn't want their company to go out of business, and you don't want to get fired for losing a whole bunch of customer data. At least, I assume that's the kind of person you are, since you're reading a whitepaper like this.

I'm not going to belabor how important this step is. If you're not doing it, do not continue reading, do not collect \$200, do not pass Go. Get on your server and schedule DBCC CHECKDB, either as a maintenance plan, or with trustworthy 3rd party tools, like [Ola Hallengren's free scripts](#), or [MinionWare's software](#).

If you've been running DBCC CHECKDB via maintenance plans, you can get some rough idea of how long the process takes by looking at the job history. It may not be terribly valuable, unless CHECKDB is the only job step, and/or you have a separate job for each database.

If you've been running DBCC CHECKDB with Ola Hallengren's scripts, it's easy to find out how long they take (as long as you're logging the commands to a table -- this is the default). The below script will take the CommandLog table that Ola's jobs populate as they run and joins them to backup information to obtain size data.

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
DECLARE @startDate DATETIME;
SET @startDate = GETDATE();

WITH cl AS (
    SELECT DatabaseName, CommandType, StartTime, EndTime, DATEDIFF(SECOND,
    StartTime, EndTime) AS DBCC_Minutes
    FROM master.dbo.CommandLog
    WHERE CommandType = 'DBCC_CHECKDB'
    AND NOT DatabaseName IN ( 'master', 'msdb', 'model', 'tempdb' )
)
SELECT DISTINCT BS.database_name AS DatabaseName ,
    CONVERT(NUMERIC(10, 1), BF.file_size / 1048576.0) AS SizeMB,
    cl.DBCC_Minutes,
    CAST( AVG(( BF.file_size / cl.DBCC_Minutes) / 1048576.0) AS INT) AS
[Avg MB/Sec]
FROM msdb.dbo.backupset AS BS
    INNER JOIN msdb.dbo.backupfile AS BF
        ON BS.backup_set_id = BF.backup_set_id
    INNER JOIN cl
        ON cl.DatabaseName = BS.database_name
WHERE BF.[file_type] = 'D'
    AND BF.[file_type] = 'D'
    AND BS.type = 'D'
    AND BS.backup_start_date BETWEEN DATEADD(yy, -1, @startDate) AND
@startDate
GROUP BY BS.database_name, CONVERT(NUMERIC(10, 1), BF.file_size / 1048576.0),
cl.DBCC_Minutes;
```

You should get a result something like this back. It's a lot like the backup query, where you'll see how long it has historically taken DBCC CHECKDB to run at different database sizes, and the average MB/s throughput of the operation.

DatabaseName	SizeMB	DBCC_Minutes	Avg MB/Sec
LogShipMe	264.0	7	37
LogShipMe	584.0	24	24
LogShipMe	1928.0	40	48
LogShipMe	3208.0	65	49

One thing you'll want to note, if you're measuring an existing server, is whether your server is responsive while DBCC CHECKDB is running or not.

That means queries and jobs run. They might slow down, and users might not experience peak happiness during these windows, but they can connect and work.

If your current server is responsive, then you have to make sure that your cloud hardware of choice holds up while maintenance is running. Just because it finishes in the same amount of time, with roughly the same throughput, doesn't mean each server experiences load the same way. Have your app, or some users ready to test connectivity during these processes.

These are DBCC CHECKDB times on the same GCE Instance with a 500 GB Persisted disk on a n1-standard-4 (4 CPUs, 15 GB memory).

I keep saying that. It's to give you perspective. This is a small box to start out with. Things will get better. Think of it like an RPG. It'll gain XP, it'll level up, and then it'll get that wicked cool sword made out of ancient magic fire or something that instantly perishes all enemies.

After all, this is the "instance sizing" section, not the "just go ahead and rent the biggest instance just in case and rack up a huge bill" section. If it were, I'd have written this on commission. This is why I'll never retire. I don't think about this stuff until the ink is dry.

If you want to do that, though, go right ahead. I'm sure Google won't mind taking your money.

DBCC CHECKDB options to consider:

- Running PHYSICAL_ONLY checks
- Restoring full backups to another server to run CHECKDB (offloading)
- Running DBCC CHECKDB at different MAXDOP levels (2016+)

Trending Index Maintenance

If you're one of those wacky people who obsesses over index fragmentation, and can't sleep unless they know some benevolent routine is hard at work making sure every single index is getting the tar pounded out of it by a reorg or rebuild every single night, this is the section for you.

[To hate.](#)

The same advice applies here as to the DBCC CHECKDB section:

- It's much better, in every single conceivable way, to use either Ola's or MinionWare's scripts to do this, than use maintenance plans.

What really stinks about index maintenance is that you could have a totally unused index that gets horribly fragmented because of modifications, and your maintenance routine will take the time to measure how fragmented it is, and then perform some action on it. This will not solve a single problem, but you'll have expended a bunch of time and resources on it because you read bad advice on a forum in 2006.

This is a terrible way to live life. DBAs and Developers have had the 5% and 30% rule for index maintenance pounded into their heads for so long that they don't know any better. No one really measures performance before and after, nor do they take into account the time and resources taken to perform the maintenance against any gains that may have occurred.

For thresholds, we usually recommend 50% to reorganize, 80% to rebuild, and for larger databases, only tables with a page count > 5000.

This is why we just like people to [update stats](#). You give SQL updated information about what's in your indexes, you invalidate old query plans, it's a hoot. And really, the updated statistics and invalidated plans are the best part of index rebuilds. Index reorgs don't do either one. Neener!

Here's an example using Ola's CommandLog table again. Nothing against the Minion folks; I just don't have their software available to setup and test and all that here.

```

WITH im AS (
    SELECT DatabaseName, Command, StartTime, EndTime, DATEDIFF(SECOND,
    StartTime, EndTime) AS Index_Minutes, IndexName
    FROM master.dbo.CommandLog
    WHERE CommandType LIKE '%INDEX%'
    AND NOT DatabaseName IN ( 'master', 'msdb', 'model', 'tempdb' )
)
SELECT  t.name,
        i.name AS IndexName,
        (SUM(a.used_pages) *8) / 1024. AS [Index MB],
        MAX(im.Index_Minutes) AS WorldRecord,
        im.Command
FROM    sys.indexes AS i
JOIN    sys.partitions AS p
ON      p.object_id = i.object_id
        AND p.index_id = i.index_id
JOIN    sys.allocation_units AS a
ON      a.container_id = p.partition_id
JOIN    sys.tables t
ON      t.object_id = i.object_id
JOIN    im
ON      im.IndexName = i.name
WHERE   t.is_ms_shipped = 0
GROUP BY t.name, i.name, im.Command
ORDER BY t.name, i.name;

```

Unless you're trending index sizes yourself, nothing else is doing it for you. That makes historical information less helpful, but then again it wouldn't really be all that great here anyway. Here's what results look like on my VM:

name	IndexName	Index MB	WorldRecord	Command
Orders	ix_Orders_CID_OST_OD	1392.312500	30	ALTER INDEX [ix_Orders_CID_OST_OD] ON [LogShipMe].[dbo].[Orders] REBUILD WITH (SORT_IN_TEMPDB = OFF, ONLINE = ON)
Orders	ix_Orders_CID_SPI_OD	883.500000	54	ALTER INDEX [ix_Orders_CID_SPI_OD] ON [LogShipMe].[dbo].[Orders] REORGANIZE WITH (LOB_COMPACTION = ON)
Orders	ix_Orders_SID_OD_OST	1014.820312	48	ALTER INDEX [ix_Orders_SID_OD_OST] ON [LogShipMe].[dbo].[Orders] REORGANIZE WITH (LOB_COMPACTION = ON)

Nearly an hour to reorganize an 883 MB index! Heh. Good times. Good times.

We don't like to use AVG MB/s throughput for index maintenance, because depending on a variety of factors, it could be getting blocked by other operations.

Recap: Your Current Vital Stats

Based on your research so far, here are your vital stats:

	Current SQL Server	Google Compute Engine
Database storage size required, GB/TB		
Backup throughput, MB/sec		
CHECKDB runtime		
Rebuild time for your largest index		

Go ahead and fill the left column's stats out directly on your monitor, using a crayon (not a Sharpie, we're not savages here).

Now, we need to build a VM up in Google Compute Engine and see how it compares.

How to Size Your Google Compute Engine VM

New Server Challenges

Sizing new hardware for both existing and new implementations of SQL Server can be quite challenging.

For existing implementations, you're typically dealing with older hardware, and it may be tough to know how much newer hardware speed can compensate for older hardware size.

"I have eighty 1.8 GHz processors, do I still need that many if they're 3 GHz?"

DID YOU EVER? Probably not. I bet they were AMD. What's wrong with you?

For new implementations, you may have limited dev/test data for your application, and you may be unsure how to future proof your hardware.

Hint: That 8 GB graphics card with four fans and enough LEDs to do neurosurgery under isn't what your server needs. Servers are boring, I know.

Giving yourself enough room to safely grow into hardware until you're sure that you're ready to make a move to something larger, or start scaling out to multiple servers sounds like witchcraft. And it is! You should see the pile of shrunken heads I have on my desk. I'm kidding; they were always this size.

Availability Groups can definitely help you scale out reads, but writes are still isolated to one machine. And they're expensive! It's exponential hardware and licensing cash out the door for each new Replica you spin up.

Good news! It's the cloud!

As we like to say around here, these aren't pets; they're cattle. There's margin for error, because it's so easy to spin up new instances with more or less power when you're testing.

You're not as married to hardware choices as you are when purchasing physical machines, though you are limited by more configuration constraints.

It's a much better use of your time learning how to create and configure new instances, join them to your Availability Group, and failover to them, than trying to count every CPU cycle and byte of RAM used on your current server, or the server in your mind that you're imagining for your application.

That being said, it's better to test what-if scenarios first, and overshoot VM size by a little. "And one to grow on" isn't just for birthday punches, it's also a great mantra for the cloud.

Choosing Your Instance Type

In the cloud, instance types are the make and model of a server - like an Acme MegaHost 4000. First, you decide which model of the MegaHost family you want:

- Acme MegaHost 1000 - single-CPU, up to 16 GB RAM, four drive bays
- Acme MegaHost 2000 - dual CPUs, up to 768 GB RAM, twenty drive bays
- Acme MegaHost 4000 - quad CPUs, 3 TB RAM, and uses more power than a Tesla

After you've decided on something from the MegaHost 2000 family, then you start configuring it with the exact CPU, memory, and storage you want.

The cloud works the same way; first, we're going to pick which instance type we want, and then we'll configure the instance type with our exact CPU/memory/storage requirements.

GCE's Relationship Between Cores and Memory

If you've built virtual machines before (either on-premises or in other cloud providers), you're used to picking whatever core count you want, and whatever memory amount you want. Those two numbers haven't had a relationship - until now.

Google Compute Engine is different because the memory amount is a direct relationship to the CPU core count:

- standard machines have 3.75 GB RAM per CPU
- highcpu machine have 0.9 GB RAM per CPU
- highmem machines have 6.5 GB RAM per CPU

That means when you hit 32 cores, your RAM is capped at 120 GB, 28.8 GB, and 208 GB, respectively (note that for custom VMs, your CPU and RAM max out at 32 CPUs and 208 GB RAM, just like highmem).

.9GB per CPU isn't enough for SQL Server, unless your current workload is CPU bound, with fewer cores and commensurate RAM. When choosing a GCE instance that will run SQL Server, the machine types that make the most sense will most likely be one from the highmem, standard, or custom family. They offer more RAM per CPU than the highcpu family.

Looking at the following chart, it's easy to see why the highcpu machine types are unlikely to suit your workload, though, ironically, it's highest level configuration is just about what our

workload will need (we'll actually have 40 GB RAM, which is slightly higher), because our database is only around 20 GB.

You just wouldn't want to rely on that in the real world, with real data, that will likely grow much faster than our contrived test data. It's just one table in a database with a limited number of indexes on it, that we dumped enough data into to inflate the size. In a real workload with multiple tables that need to be indexed and joined and reported on and all that good stuff, things will look different.

Plus, databases are always full of extra junk. They're messy places. Backup tables no one dropped, logging tables that don't get used, but have to stick around because some pencil pushing "auditor" will throw a hissy fit if it's not there, ridiculous XML/JSON conversion therapy that your developers swore would fix all your performance issues. It's horrible.

That's why understanding your Working Data Set® (this isn't really a registered trademark; Deal With It©) is important to sizing your VM. You could have a 100 GB database that only has 40 GB of data that users access regularly.

I'm not going to get all SAN admin on you, and wag my finger while talking about wasted space. If you don't want to ever archive, prune, or purge data, you don't have to. You're an adult, probably. Or you're a kid that I would have beat up when I was a kid. Seriously little kid, stop reading this. Go outside.

Back to the chart -- we believe it's likely that most workloads, especially those that will be using Availability Groups, should begin testing at one of the higher CPU count instance types.

This is purely due to available CPU threads. Availability Groups require more background tasks than a standalone server to keep Replicas synchronized. Bet you didn't think about that when you came up with your grand plan to scale reads out to 8 different servers, huh?

For a breakdown of number of threads at different CPU counts, check out [this msdn article](#).

When I look at this chart, I'm mentally crossing off the first 7 or 8 lines of servers, because they're just not workable configurations for SQL Server. If you think they are, we have successfully detected a sysadmin! [Head over here](#).

The first 3 can barely run Windows, especially if you want to open the Start menu. The next two are good for playing Team Fortress 2 with all the highest graphics settings. After that, the fun begins.

I'm really happy that custom machine sizes exist, because a lot of these combos are awkward teenagers. Their feet are too big, they have bad skin, and one armpit just started to smell really bad.

Machine Type	CPUs	Memory GB
n1-highcpu-2	2	1.8
n1-highcpu-4	4	3.6
n1-standard-1	1	3.75
n1-highcpu-8	8	7.2
n1-standard-2	2	7.5
n1-highmem-2	2	13
n1-highcpu-16	16	14.4
n1-standard-4	4	15
n1-highmem-4	4	26
n1-highcpu-32	32	28.8
n1-standard-8	8	30
n1-highmem-8	8	52
n1-standard-16	16	60
n1-highmem-16	16	104
n1-standard-32	32	120
n1-highmem-32	32	208

You can create a custom instance, but they have similar CPU\RAM constraints, maxing out at 6.5 GB RAM per CPU, and requiring an even number of CPUs (after 1 CPU).

Examples of CPU and RAM limits for custom instances are as follows:

- 4 CPUs : 26 GB RAM
- 6 CPUs : 39 GB RAM
- 8 CPUs : 52 GB RAM
- 10 CPUs : 65 GB RAM
- 12 CPUs : 78 GB RAM
- 16 CPUs : 104 GB RAM

- 20 CPUs : 130 GB RAM
- 24 CPUs : 156 GB RAM
- 32 CPUs : 208 GB RAM

I tried to keep the configs to common real world scenarios. There are even-number CPU steps in between some of these, but this should illustrate the relationship between CPU count and RAM limits in customized instances.

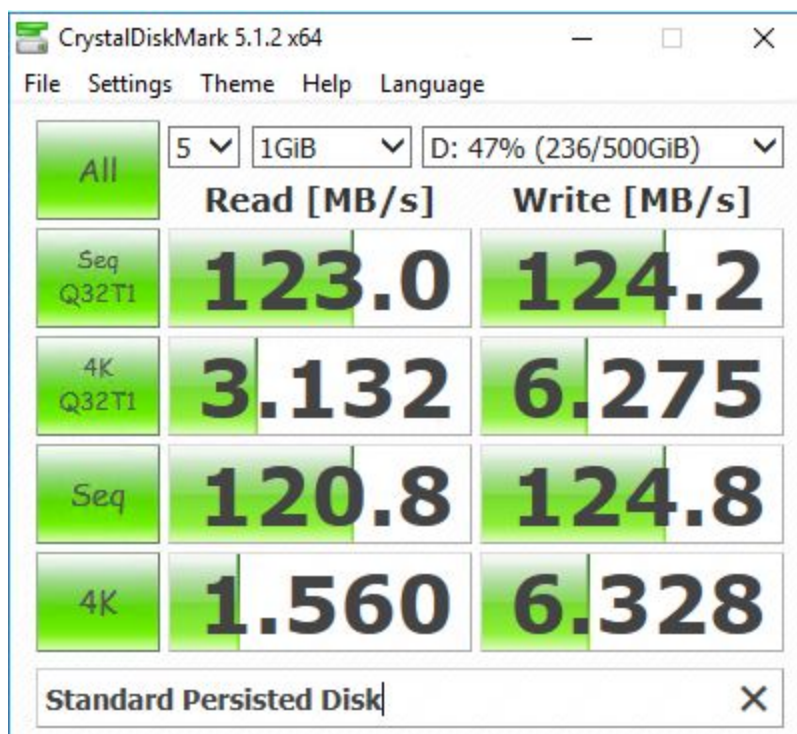
Memory is More Important in the Cloud

Adequate memory becomes even more crucial in the cloud, where higher storage tiers can become quite costly, and **local** SSDs don't persist data across as many [failure scenarios](#) as slower storage.

The less time you spend hitting disk, the better off you are, and that's true regardless of where your server is.

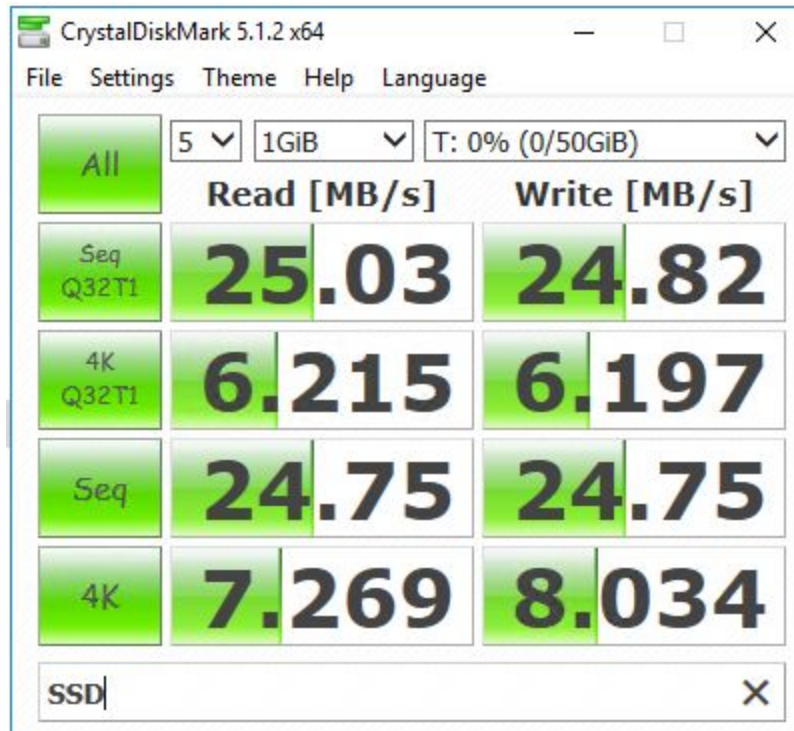
How different are the two disk types? Let's ask CrystalDiskMark.

Here's what a Standard Persisted disk looks like.



George RR Martin called, he wants his write speed back.

What about a 50 GB SSD?



THAT'S AN SSD?!

Yep. In GCE, disk size has a [direct correlation](#) to disk speed. This is the chart with pricing for SSDs, along with disk performance for the different sizes.

Pay close attention to this. You may need to spin up a much larger disk than you actually need to get the storage performance you need.

That's a pretty big catch for a lot of shops, especially considering storage speed and prices these days for Cloudless architecture.

You'll really want to run CDM on your physical servers and check the numbers you get back against the size/speed/price charts when figuring out what the cloud is going to cost you.

If you're unhappy with current disk speeds, you have a good starting comparison point. It will cost you \$X a month to continue to be unhappy. It will cost you \$X more a month to be happ(y)(ier), for limited values of happ.

Use the chart below as a quick reference for the performance and cost of some common SSD Persistent Disk volume sizes:

Volume Size (GB)	Monthly Price	Sustained Random Read IOPS Limit	Sustained Random Write IOPS Limit	Sustained Read Throughput Limit (MB/s)	Sustained Write Throughput Limit (MB/s)
10	\$1.70	1500	1500	24	24
50	\$8.50	1500	1500	24	24
100	\$17.00	3000	3000	48	48
200	\$34.00	6000	6000	96	96
500	\$85.00	15000	15000	240	240
667	\$113.39	20000	20000	240	240
834	\$141.78	25000	25000	240	240
1000	\$170.00	25000	25000	240	240
5000	\$850.00	25000	25000	240	240
10000	\$1700.00	25000	25000	240	240
16000	\$2720.00	25000	25000	240	240
32000	\$5440.00	25000	25000	240	240
64000	\$10880.00	25000	25000	240	240

The 500 GB SSD should suit *just about* anyone's TempDB or Log File needs. Putting TempDB and Log Files on higher tiers of storage is usually done to mitigate write-related waits in high transaction workloads. It may not be necessary for you, but it's something to keep in mind.

For very high performance scenarios, you'll probably want to use a **local** SSD (you can choose between SCSI and NVMe) for TempDB, and regular Persisted SSDs for Log Files, like so.

Encryption ?

Automatic (recommended) ▼

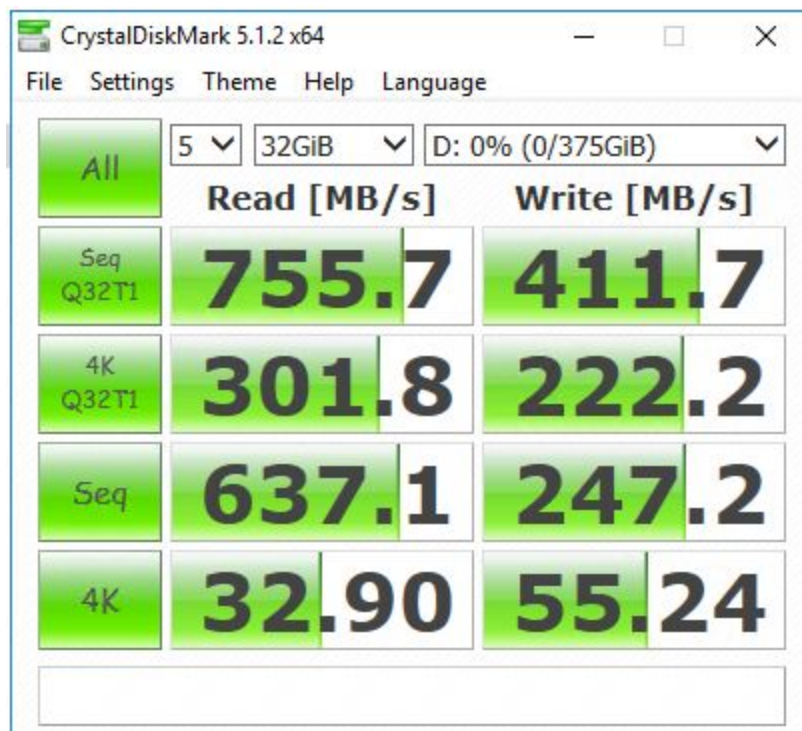
Additional disks ? (Optional)

Name	Mode	When deleting instance	
local-ssd-0	Read/write ▼	Delete disk ▼	×
sql-data	Read/write ▼	Keep disk ▼	×
sql-log	Read/write ▼	Keep disk ▼	×

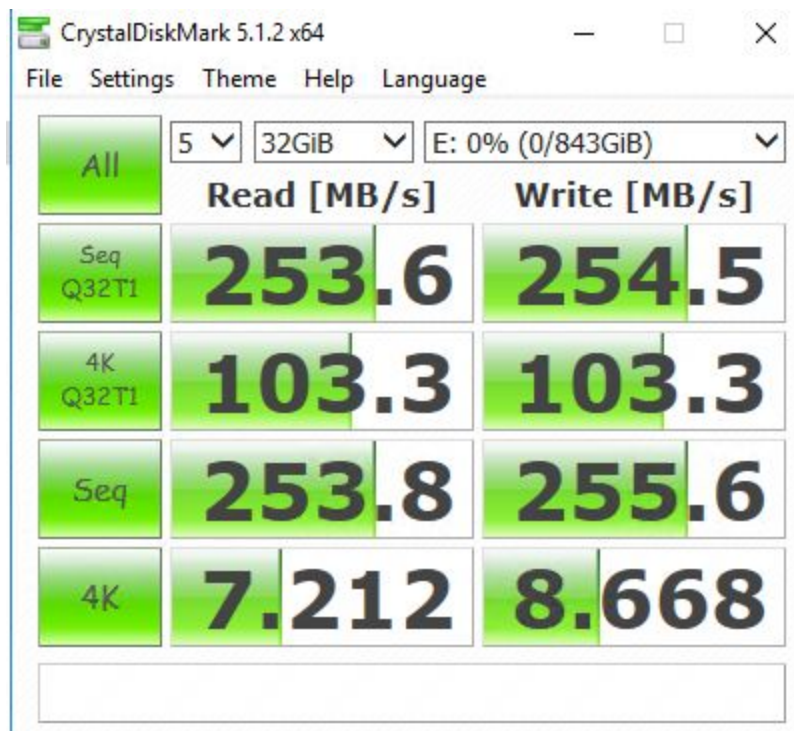
+ Add item

This gives us a local NVMe SSD for TempDB, and Persisted SSDs for Data and Log Files. Why? Because this is a test environment, and money's no object. I'M RICH! RIIIIICH! I WILL BUY YOU!

This is what performance looks like on a **local** SSD (NVMe, not SCSI). Pretty nice! Expensive, but nice. The funny thing about VMs you spin up with **local** SSDs: you can't turn them off. You have to delete them.



And this is what performance looks like on a 843 GB **persisted** SSD. Way better than the 50 GB one, right? Again, more expensive, but it starts to resemble a modern disk. I mean, it's like 1/3 of the way to USB 3.0



Data files are typically housed on larger, slower storage tiers, depending on size. Multi-terabyte to multi-petabyte environments would be prohibitively expensive to store on all SSD or flash arrays. For smaller environments, it may make sense to just put everything on SSD, assuming the cost fits your budget.

As discussed in the memory section, it's a much cheaper option, and especially for reducing read requirements for data files, a smarter approach.

There are more considerations around which edition of SQL Server you're on when dealing with memory constraints. Standard has a cap of 128 GB in 2014 and 2016 (I'm not entertaining conversations about versions prior to those. Seriously. Stop.).

If your data doesn't fit into that, and you're not ready to make the licensing leap to Enterprise, you may have to foot the speedy storage bill to get some performance back.

For Standard Disks, there's a similar correlation, though the prices and speeds are obviously much lower. If I were setting up a long term server in GCE, I'd definitely use one of these to store backup files, and then rsync them to a Storage bucket or two, Just In Case®.

Use the chart below as a quick reference for the performance and cost of some common Standard Persistent Disk volume sizes.

Volume Size (GB)	Monthly Price	Sustained Random Read IOPS Limit	Sustained Random Write IOPS Limit	Sustained Read Throughput Limit (MB/s)	Sustained Write Throughput Limit (MB/s)
10	\$0.40	*	*	*	*
50	\$2	37.5	75	6	6
100	\$4	75	150	12	12
200	\$8	150	300	24	24
500	\$20	375	750	60	60
1000	\$40	750	1500	120	120
2000	\$80	1500	3000	180	120
4000	\$160	1500	3000	180	120
5000	\$200	3000	7500	180	120
10000	\$400	3000	15000	180	120
16000	\$640	3000	15000	180	120
32000	\$1280	3000	15000	180	120
64000	\$2560	3000	15000	180	120

These charts existed in January of 2017, when the whitepaper was written (it's funny writing this now; some facts about the cloud won't last longer than my current cup of coffee).

As with most things, check back with the source link to see if things have changed. You may be reading this in THE FUTURE.

Or a week from now. Maybe Google will make changes based on this whitepaper.

The bottom line on disks: avoid them as much as possible.

Where you can't avoid them, make them as fast as possible. Or as fast as you can afford. You may have to Settle For Less©.

Memory is a whole heck of a lot cheaper, but it's in much more finite supply up in the cloud. Cloudless architecture can support many terabytes of memory, and that just may be the best fit for your needs, unless you can scale incredibly wide.

But that's expensive, too. All those instances, out there in some data center, costing you money.

What Does It Mean For You?

No, you don't need to go out and rent the biggest machine with the most RAM right off the bat, but it does mean that you may need more RAM from the outset.

You may find that persisted storage in the cloud doesn't measure up to persisted storage in the physical realm.

When choosing disks, be aware that **local** SSDs can only be added at instance creation, and can't be used as a boot disk. That's because **local** SSDs don't offer the same persistence as, well, persisted disks.

Data can be lost, so it may only make sense to stick something like TempDB on them, since it's reinitialized when SQL starts up anyway. It's not all doom and gloom, though. **Local** SSDs can hang onto data under certain circumstances, like rebooting the guest, or if the instance is set up for live migration.

There is an option for SSD **Persisted** disks, which are governed by the size/speed/price chart in the previous section. These can be booted from, and added at any time, but cost much more than Standard Persisted Disks. Obviously. They're fast. But slower than their **local** SSD counterparts.

Choosing Your CPU Type

CPU is another important consideration. As of this writing (early 2017), the CPU types available are:

- 2.6 GHz Intel Xeon E5 (Sandy Bridge)
- 2.5 GHz Intel Xeon E5 v2 (Ivy Bridge)
- 2.3 GHz Intel Xeon E5 v3 (Haswell)
- 2.2 GHz Intel Xeon E5 v4 (Broadwell)

Note that 32-core machine types are not available in Sandy Bridge zones us-central1-a and europe-west1-b. Careful comparison to current CPUs is in order -- if your CPUs have higher clockspeeds, you may need to invest in more cores to get similar performance.

You also don't get to choose which CPUs you get, so be prepared to assume you're getting the lowest clockspeed, 2.2 GHz, when comparing to your current server build. Or your imaginary server build for your imaginary app. We all know you're just building a really cool gaming rig, though.

Putting It All Together: Build, Then Experiment

Based on what you've learned so far, now it's time to spring into action:

1. Create a GCE VM with your best guess on sizing
2. Restore your backups into it
3. Run your maintenance jobs and see if your performance is similar

Now, get that crayon back out and fill in the right hand column:

	Current SQL Server	Google Compute Engine
Database storage size required, GB/TB		
Backup throughput, MB/sec		
CHECKDB runtime		
Rebuild time for your largest index		

If you're not satisfied with your performance metrics, you have two choices.

1. **You can do performance tuning.** Believe it or not, you can actually tune SQL Server's maintenance jobs. For example, with SQL Server 2016, you can now pass a MAXDOP hint to the CHECKDB command to make it use more or less CPU cores. Thing is, you have to know how to do this tuning - which honestly, is outside of the scope of this white paper. (It's the kind of thing we teach in our performance tuning classes.)
2. **Or, you can try a different instance size.** Get that wallet out and vote with it, throwing more hardware at the problem.

However, to know what *kind* of hardware you need to throw - CPU, memory, or storage - we're going to need to know what your bottleneck is.

So let's talk about that.

How to Measure What SQL Server is Waiting On

Usually, when asked to measure a SQL Server, people will freeze up for a few seconds, and then start talking about all sorts of useless things. They'll open up task manager to look at RAM and CPU usage, they'll point to metrics like disk queue length, context switches, and buffer cache hit ratios, and they'll point proudly to their...

Totally.

Defragmented.

Indexes.

None of that matters.

We're going to use wait statistics using an open source tool called `sp_BlitzFirst`. It'll tell us:

- Are there any current bottlenecks?
- What are things historically waiting on?
- What are queries currently waiting on?

This sounds a lot harder than it is. We've put a lot of work into making our free scripts as easy to use and understand as possible. In a test environment, it's easy to show off the most common wait types and overcome them with just configuration changes.

In real life, you may be hitting some more exotic wait types, issues that require Microsoft support, or the expert tutelage of some young and good looking SQL Server consultants.

Ahem. You know, the type who write whitepapers about this stuff.

Or whoever.

(Call me.)

An Introduction to Wait Stats

This isn't a wait stats white paper, but we should talk a little bit about what they are and why they're important. For some additional information, [head over here](#).

When a task runs in SQL Server, it has three potential states: running, runnable, and suspended.

- Running: Task is actively processing
- Runnable: Task can run when it gets on a CPU
- Suspended: Task is waiting on a resource or request to complete

During each of these phases, the task may need to wait for a physical resource, like CPU, I/O, or memory. It may also need to wait on a logical resource, like a lock or latch on data pages already in memory.

SQL Server tracks this data and exposes it to you in a dynamic management view (DMV) named `sys.dm_os_wait_stats`, but that's painful to query. The problem is that it just tracks cumulative time - so if you go look at it right now, it'll say that your server has waited for 4,367 years on your spinning rusty magnetic frisbees.

We need more granular data - we need to know *exactly* what SQL Server was waiting on, and more importantly, *when*. See, you don't care that SQL Server was waiting on slow storage while your backups were running - you care what it's waiting on when your users are running their diabolical queries.

How to Get More Granular Wait Stats Data

There's no replacement for a mature, platform-specific monitoring tool. The internet is full of 3rd party vendors whose salespeople will fight to the death for you to choose their product.

Ultimately, you should pick the one that you like the best. The product, not the salesperson (dead or alive). It has to be something that you'll actually use, understand, and will help you sort out problems as they arise.

Vendors we like include [SentryOne](#), [Quest](#), and [Idera](#).

For this white paper, we're going to use stored procedures from our [First Responder Kit](#). Primarily, `sp_BlitzFirst`, which calls another stored procedure called `sp_BlitzWho`. `sp_BlitzWho` is an open source query similar to, but far less robust, than [sp_WhoIsActive](#). It just shows you what's running without many of the bells and whistles in `sp_WhoIsActive`.

Whether you're using a third party tool or an open source script like sp_BlitzFirst, you're going to see a cryptic list of wait types. You need a decoder ring. Fortunately for you, I have a trench coat full of jewelry.

Wait Type Reference List

SQL Server has a robust array of wait types to indicate which resources tasks most frequently wait on. A few of them, though not many, are even sensibly named.

We're going to focus on the most common, and which resources they map to. They can lead you to your worst performance problems, and they can also tell you if your SQL Server is just [sitting around bored](#).

This list is by no means exhaustive. If you need information about a wait type not listed here, [Waitopedia](#) can be helpful. For more detailed help, the [DBA Stack Exchange](#) site is great for asking long form questions.

CPU

CXPACKET: Parallel queries running. This is neither good or bad.

SOS_SCHEDULER_YIELD: Queries quietly queueing quoperatively (work with me here)

EXECSYNC: Parallel queries waiting on an operator to build an object (spool, etc.)

THREADPOOL: That's you running out of CPU threads to run tasks with. This can be a sign of a big problem.

Memory

RESOURCE_SEMAPHORE: Queries can't get enough memory to run, so they wait.

RESOURCE_SEMAPHORE_QUERY_COMPILE: Queries can't get enough memory to compile a plan. Both Resource Semaphore wait types can be a sign of a pretty big problem.

CMEMTHREAD: CPU threads trying to latch onto memory. This theoretically shouldn't be an issue in SQL Server 2016, but hey, maybe you're special? Happens more often on systems with >8 cores per socket. May need Trace Flag 8048. This is complicated. Call Bob Ward.

Disk

PAGEIOLATCH_SH: These are all pages being read from disk into memory.

PAGEIOLATCH_UP: If you have a lot of these waits, pay really close attention to your cloud VM

PAGEIOLATCH_EX: If may need a lot more memory than your Cloudless server has
Note that there are several other variety of PAGEIOLATCH_** waits, but they're far less common.

WRITELOG: SQL writing to log files

Locks

LCK_M_**:

There are many, many variations of lock waits. They all start with LCK_M_ and then an acronym to designate the type of lock, etc. Diving into each one is beyond the scope of this whitepaper.

Latches

LATCH_EX: Latches waiting for access to objects that aren't data pages, already in memory

PAGELATCH_SH: Don't confuse these with PAGEIOLATCH. The names look alike.

PAGELATCH_UP: These are usually associated with objects in tempdb, but can also happen

PAGELATCH_EX: in regular user databases as well.

Note that there are several other variety of PAGELATCH_** waits, but they're far less common.

Misc

OLEDB: Most common during CHECKDB and Linked Server queries

BACKUP*: There are several wait types that start with BACKUP, which, along with

ASYNC_IO_COMPLETION accumulate during backups.

Always On Availability Groups Waits

If you plan on implementing an Availability Group, there are a number of other waits that may crop up, especially during data modifications.

Using 2016 is especially important for highly transactional workloads that are part of an Availability Group, because of the ability to [REDO in parallel](#).

This is one of those touchy subjects though -- it's not just the SQL Server at work here. You have to push data across a network, and to another disk. Depending on which zone each node is in, and how far apart those zones are, you may have very different network performance, making synchronizing data difficult depending on the size of and volume of transactions.

You need to keep Replicas synchronous only in the same zone, and asynchronous otherwise.

Demo: Showing Wait Stats with a Live Workload

The best way for you to learn this stuff is to see it in action. This is a white paper, though, and since it doesn't support animated GIFs, I'm going to have to use words to describe it to you.

About Our Server

We're starting off using the n1-standard-4 (4 CPUs, 15 GB memory) instance type. Duh. Remember. RPG. We still haven't left the first town. We just got a crappy wooden sword from some old guy who wants us to go on a quest. We need to go spend our 15 GP wisely.

We have two standard persisted disks: one for the OS, and one for SQL that contains SQL Server binaries, Data, Log, TempDB, and backup files.

Running SQL Server 2016, the only settings adjustments made are for memory and parallelism.

- Max server memory is set to 10 GB (10240 MB)
- MAXDOP is set to 4
- Cost Threshold For Parallelism is set to 50

SQL Server 2016 has (automagically!) configured TempDB to have four data files during setup, and per 2016's change to SQL Server behavior, Trace Flags 1117 and 1118 are "turned on" by default. No other non-default Trace Flags have been enabled.

Logical Name	File Type	Filegroup	Initial Size (MB)	Autogrowth / Maxsize
tempdev	ROWS...	PRIMARY	1608	By 64 MB, Unlimited
temp2	ROWS...	PRIMARY	1608	By 64 MB, Unlimited
temp3	ROWS...	PRIMARY	1608	By 64 MB, Unlimited
temp4	ROWS...	PRIMARY	1608	By 64 MB, Unlimited
templog	LOG	Not Applicable	72	By 64 MB, Unlimited

About Our Database: Orders

The database itself has been enlarged to about 20 GB now. The point of that is to have just enough data so that it doesn't all quite fit into memory on some of the smaller boxes. We can demonstrate what a server looks like when it's underpowered, and when you should start looking at increasing certain resources.

Currently on the server, we have one database, and one table. Not much, but we don't need much to show some wait stats and get you comfortable with using our tools to look at your own server.

The database and table creation, as well as initial data population and stored procedures to generate activity are all available at BrentOzar.com/go/gce. To call them, we'll set up Agent jobs that call a Microsoft utility called ostress, which is part of the [RML Utilities](#) package.

About Our Workload

Determining a proper workload is difficult, and all the considerations are well beyond the scope of this whitepaper. It's a bit easier to gauge if you have a monitoring tool in place. If you don't, you can try [logging sp_WhoIsActive](#) to a table during normal usage periods to get a better idea of what's running on your server, and how frequently. You could also just guess. That doesn't work terribly well, though. I've tried recreating workloads via the spoils system, but the turnover stinks.

To run our workload, we'll be using SQL Server Agent jobs, along with a free Microsoft tool called [RML Utilities](#), which includes a CLI called ostress. Agent jobs can call the CLI tool natively, which makes configuration easier, and ostress itself is fairly customizable.

The database as well as the stored procedures referred to in this section are part of the resources at BrentOzar.com/go/gce. Keep in mind that, while we really tried to be thoughtful about having a workload that adequately resembles a system with a decent number of users reading and writing data, running some simple reports, and utilizing TempDB, it is still a contrived workload.

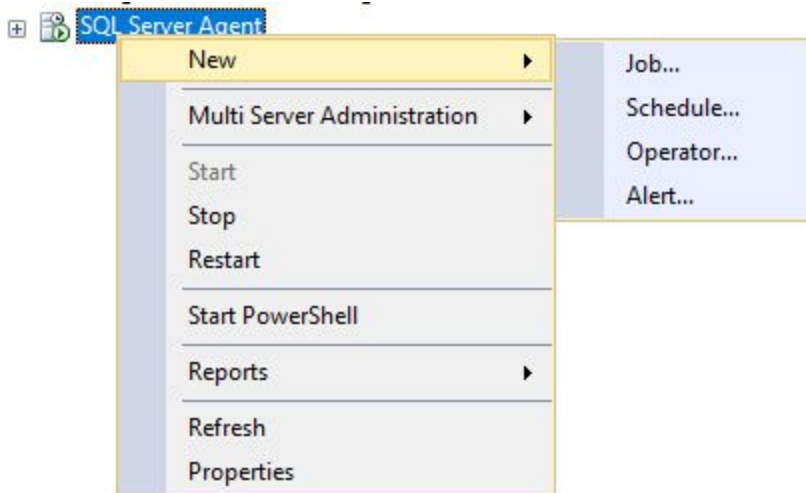
That's a nice way of saying that nothing we're doing has any real business purpose, and you shouldn't take it to mean it's what your workload should or will look like.

After downloading and installing RML Utilities on the server you want to run the workload on, you should have a folder path that resembles this. This is the path you'll use to call ostress.exe.

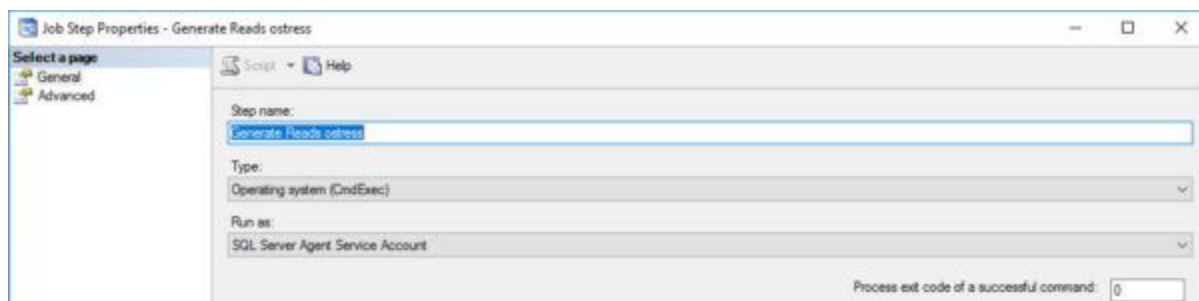
> This PC > Local Disk (C:) > Program Files > Microsoft Corporation > RMLUtils				
	Name	Date modified	Type	Size
ss	Help	1/18/2017 6:43 PM	File folder	
	Samples	1/18/2017 6:43 PM	File folder	
ls	x64	1/18/2017 6:43 PM	File folder	
ts	x86	1/18/2017 6:43 PM	File folder	
	BRICOLSOFTZipx64.dll	8/14/2012 7:46 PM	Application extens...	572 KB
: (C:)	Expander.exe	12/10/2014 1:01 PM	Application	575 KB
	ExpanderDLL.dll	12/10/2014 11:33 ...	Application extens...	422 KB
	License.rtf	11/7/2008 3:53 PM	Rich Text Document	46 KB
	Microsoft.ReportViewer.Common.dll	5/23/2013 11:28 AM	Application extens...	6,242 KB
	Microsoft.ReportViewer.DataVisualization...	5/23/2013 11:28 AM	Application extens...	3,785 KB
	Microsoft.ReportViewer.ProcessingObjec...	5/23/2013 11:28 AM	Application extens...	90 KB
	Microsoft.ReportViewer.WinForms.dll	5/23/2013 11:28 AM	Application extens...	536 KB
	Microsoft.SQLEscalationSupport.DLL	12/10/2014 1:04 PM	Application extens...	14 KB
	Microsoft.SQLEscalationSupportImpl.dll	12/10/2014 1:04 PM	Application extens...	22 KB
	Microsoft.SqlServer.XE.Core.DLL	2/21/2014 7:27 AM	Application extens...	66 KB
	Microsoft.SqlServer.XEvent.Linq.dll	2/21/2014 7:27 AM	Application extens...	279 KB
	ORCA.exe	12/10/2014 1:01 PM	Application	324 KB
	ostress.exe	12/10/2014 1:01 PM	Application	1,290 KB
	ReadTrace.exe	12/10/2014 1:01 PM	Application	1,996 KB
	Reporter.exe	12/10/2014 1:04 PM	Application	2,473 KB
	RML.cmd	1/14/2008 1:53 PM	Windows Comma...	1 KB
	UnRar64.dll	3/17/2014 4:39 PM	Application extens...	256 KB
	XceedZipX64.dll	8/17/2009 10:47 AM	Application extens...	784 KB

In SQL Server Management Studio, you need to expand Object Explorer for your test server so you can see SQL Server Agent. Make sure that it's enabled -- it'll have a little green arrow on it. If you're colorblind, it'll say disabled, not because you are, but because SQL Agent is. Don't take it personally.

Creating a new job is easy. Right clicks rule the world.



It'll ask you to name the job, and then you'll need to set up the step to call ostress. To do that, make sure you choose the Operating system (CmdExec) type.



This is the easiest way to call an executable. I see some people use xp_cmdshell and then generate some dynamic-ish SQL to run it, but that's not necessary here. You could follow that if you wanted to randomly assign some values to the ostress calls, but that would be over-engineering for our use case.

If you don't feel like doing all of this nonsense, the resources at BrentOzar.com/go/gce have a script you can run that automatically creates all the Agent jobs as I have them for a test workload. You can alter those to do whatever you want. Or just leave them, because I made them special for you and I'd be HEARTBROKEN if you changed them.

Whatever.

This is the command to call our read workload stored procedure, with comments. The actual text you'll want to use is in the resources at BrentOzar.com/go/gce, along with commands for writes and hitting TempDB.

```
"C:\Program Files\Microsoft Corporation\RMLUtils\ostress.exe"--Path to ostress executable
-SNADAULTRA\SQL2016C --Server name (note that this is how you access a named instance)
-d"StackOverflow" --Database name
-n10 --How many simultaneous sessions you want to run your query
-r5 --How many iterations they should each perform
-q --Quiet mode; doesn't return rows
-Q"EXEC dbo.GenerateReads" --Query you want to run
-o"C:\temp\readslog" --Logging folder
```

There are a number of other flags available, check in the documentation, but these are the ones I usually run with. If I wanted to feed ostress a .sql file of commands to use instead of a stored procedure or ad hoc query, I could use `-i"C:\temp\BeatUpServer.sql"` instead.

It's particularly important to specify different logging folders if you're going to run ostress concurrently. Otherwise, every session will try to use the default, and you'll get a bunch of access denied errors when it attempts to clean up logging folders at the start of each run.

I like to schedule the jobs to run every 10 seconds. Not because I expect them to run that often, but because I want as little pause between runs as possible. One limitation of SQL Agent is that its most granular schedule is a 10 second interval. If you need anything tighter than that, you can, get this, set up an Agent Job that runs your Agent Job in a `WHILE 1=1` loop. Yo dawg, etc.

Job Schedule Properties - Run Read Queries Every 10 Seconds 5

Name: Run Read Queries Every 10 Seconds 5 Jobs in Schedule

Schedule type: Recurring ☒ Enabled

One-time occurrence

Date: 1/19/2017 Time: 5:45:06 PM

Frequency

Occurs: Daily

Recurs every: 1 day(s)

Daily frequency

☐ Occurs once at: 12:00:00 AM

☒ Occurs every: 10 second(s)

Starting at: 12:00:00 AM

Ending at: 11:59:59 PM

Duration

Start date: 1/17/2017 ☐ End date: 1/19/2017





☒ No end date:

Summary

Description: Occurs every day every 10 second(s) between 12:00:00 AM and 11:59:59 PM. Schedule will be used starting on 1/17/2017.

OK Cancel Help

The four jobs I have set up look like this in Job Activity Monitor.

	Generate Read Activity
	Generate TempDB Activity
	Generate Write Activity
	Hourly Report Refresh

With our four jobs set up, we can finally...

Measuring Our SQL Server with sp_BlitzFirst

We have a couple choices with how we run it. We can look since the server started up.

```
EXEC sp_BlitzFirst @SinceStartup = 1
```

We can also take a current sample!

```
EXEC sp_BlitzFirst @Seconds = 30, @ExpertMode = 1
```

There are a bunch more possible options all detailed at firstresponderkit.org. This is what we need to get going, though.

Now, this is the same itty bitty instance we've been discussing the whole time. I've been beating the tar out of it. It's so bad that GCE is throwing a warning about it.

Zone	Machine type	Recommendation
us-central1-f	4 vCPUs, 15 GB	💡 Increase perf.

Inside, this is what our wait stats look like.

Baseline #1: Waiting on PAGEIOLATCH, CXPACKET, SOS_SCHEDULER_YIELD

Since startup, we have many unhappy waits. When looking at this, what should jump out isn't just the total wait time necessarily, though that does indicate heavy resource usage, but also how efficient that resource is.

Leaving aside the LCK_* waits, which indicate a pretty bad locking problem, our top waits are all CPU and disk related. Neither resource is responding well to pressure. Look how long the [average waits](#) are. Disks take on average 46ms to return data (PAGEIOLATCH**), and CPU waits (CXPACKET, SOS_SCHEDULER_YIELD) take 8.5ms and 21ms respectively, on average.

Hours Sample	wait_type	Wait Time (Hours)	Per Core Per Hour	Signal Wait Time (Hours)	Percent Signal Waits	Number of Waits	Avg ms Per Wait
97.0	PAGEIOLATCH_SH	374.7	0.0	6.8	1.8	29347173	46.0
97.0	CXPACKET	296.5	0.0	3.4	1.1	123516185	8.6
97.0	LCK_M_IS	280.1	0.0	0.1	0.0	49415	20403.9
97.0	SOS_SCHEDULER_YIELD	217.0	0.0	217.0	100.0	36796867	21.2
97.0	LCK_M_X	76.9	0.0	0.0	0.0	12914	21448.3
97.0	WRITELOG	30.4	0.0	1.0	3.4	109871407	1.0
97.0	LCK_M_IX	10.6	0.0	0.0	0.0	4805	7970.7
97.0	PAGEIOLATCH_EX	5.3	0.0	0.0	0.2	1343155	14.2

Likewise, focusing on disks brings back some pretty dismaying news. They're taking between .5 and 2 **SECONDS** to respond, on average.

Pattern	Sample Time	Sample (seconds)	File Name	Drive	# Reads/Writes	MB Read/Written	Avg Stall (ms)	file physical name
PHYSICAL READS	2017-01-21 14:55:42.2426491 +00:00	349330	LogShipMe [ROWS]	D:	55651237	3772002.9	491	D:\Data\LogShipMe.mdf
PHYSICAL READS	2017-01-21 14:55:42.2426491 +00:00	349330	MSDBData [ROWS]	C:	6137	308.5	185	C:\Program Files\Microsoft SQL Server\MSSQL13.MSS...
PHYSICAL READS	2017-01-21 14:55:42.2426491 +00:00	349330	master [ROWS]	C:	2335	92.2	77	C:\Program Files\Microsoft SQL Server\MSSQL13.MSS...
PHYSICAL WRITES	2017-01-21 14:55:42.2426491 +00:00	349330	temp4 [ROWS]	C:	266505	8075.6	1998	C:\Program Files\Microsoft SQL Server\MSSQL13.MSS...
PHYSICAL WRITES	2017-01-21 14:55:42.2426491 +00:00	349330	tempdev [ROWS]	C:	270414	8100.4	1965	C:\Program Files\Microsoft SQL Server\MSSQL13.MSS...
PHYSICAL WRITES	2017-01-21 14:55:42.2426491 +00:00	349330	temp2 [ROWS]	C:	268908	8093.5	1964	C:\Program Files\Microsoft SQL Server\MSSQL13.MSS...
PHYSICAL WRITES	2017-01-21 14:55:42.2426491 +00:00	349330	temp3 [ROWS]	C:	270553	8098.9	1952	C:\Program Files\Microsoft SQL Server\MSSQL13.MSS...
PHYSICAL WRITES	2017-01-21 14:55:42.2426491 +00:00	349330	templog [LOG]	C:	19003	1081.4	690	C:\Program Files\Microsoft SQL Server\MSSQL13.MSS...

Don't make fun of me for putting TempDB on the C drive. If I don't make these mistakes here, you'll make them in real life. Then you'll start sending me nasty emails about server performance.

Various 30 second samples of wait stats don't brighten the mood in Serverville, either. In fact, at various points in our workloads, things look like this server is on the verge of falling over.

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
28	SOS_SCHEDULER_YIELD	789.2	7.0	789.2	100.0	26936	29.3
28	LCK_M_IS	419.3	3.7	0.9	0.2	27	15528.4
28	CXPACKET	70.5	0.6	0.0	0.0	29	2429.9
28	LCK_M_IX	5.1	0.0	0.0	0.0	4	1276.8
28	MSQL_XP	1.1	0.0	0.0	0.0	2	566.5
28	ASYNC_NETWORK_IO	1.1	0.0	0.4	36.4	136	8.0
28	PREEMPTIVE_XE_GETTARGETSTATE	0.9	0.0	0.0	0.0	9	99.1
28	PAGELATCH_EX	0.4	0.0	0.1	25.0	61	6.6

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
23	SOS_SCHEDULER_YIELD	628.8	6.8	628.8	100.0	12585	50.0
23	CXPACKET	327.8	3.6	5.1	1.6	309	1060.7
23	PAGEIOLATCH_SH	10.4	0.1	6.2	59.6	316	32.8
23	PAGELATCH_EX	3.5	0.0	0.7	20.0	114	30.5
23	MSQL_XP	2.3	0.0	0.0	0.0	4	568.0
23	PREEMPTIVE_OS_DELETESECURITYCONTEXT	0.4	0.0	0.0	0.0	14	30.0
23	PREEMPTIVE_OS_DISCONNECTNAMEDPIPE	0.4	0.0	0.0	0.0	14	30.6
23	PAGELATCH_SH	0.3	0.0	0.1	33.3	14	24.3

Yes, there are 8.5 second waits on CXPACKET, here. No, this isn't good.

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
30	CXPACKET	962.6	8.0	0.1	0.0	113	8519.0
30	SOS_SCHEDULER_YIELD	576.4	4.8	576.4	100.0	22041	26.2
30	LCK_M_IS	487.4	4.1	0.1	0.0	28	17408.4
30	LCK_M_X	47.5	0.4	0.0	0.0	4	11882.3
30	LCK_M_IX	25.6	0.2	0.0	0.0	2	12775.5
30	PAGEIOLATCH_SH	7.7	0.1	1.2	15.6	392	19.6
30	MSQL_XP	3.4	0.0	0.0	0.0	5	688.4
30	RESERVED_MEMORY_ALLOCATION_EXT	2.5	0.0	0.0	0.0	14930	0.2

Mitigation #1: Fixing PAGEIOLATCH, SOS_SCHEDULER_YIELD

This server is under-provisioned in several areas:

In our wait stats, we saw issues with

- CPU: CXPACKET and SOS-SCHEDULER_YIELD
- Disk: PAGEIOLATCH_SH and PAGEIOLATCH_EX
- Locking: all the LCK_* types

What we're really paying attention to is the long average waits on these resources. We'll attempt to alleviate contention with CPU by adding CPU power, and alleviate the Disk waits by adding... Memory! Remember, the best way to avoid reading from disk is to cache as much data as you can in memory.

For the CPU stuff, there's sort of a death dance doing on. Many queries are going parallel across our pitiful four cores. The cooperative scheduling inside SQLOS tells queries that if they've been waiting more than 4ms for something to continue running, they have to give another task a chance to run. That's SOS_SCHEDULER_YIELD, and normally, it's a good thing.

So right now we have queries waiting for pages to be read from disk, queries waiting on locks, and queries waiting on parallel processes to finish. While they wait for all this stuff, they get pushed around the task queue because they 'exhaust their quantum' -- that's the fancy term for the 4ms time limit we talked about before.

By adding more CPUs, we make more threads available, which means our queries have more options about [where they go to run](#). That means a whole lot less pushing and shoving to get on the four CPUs currently available.

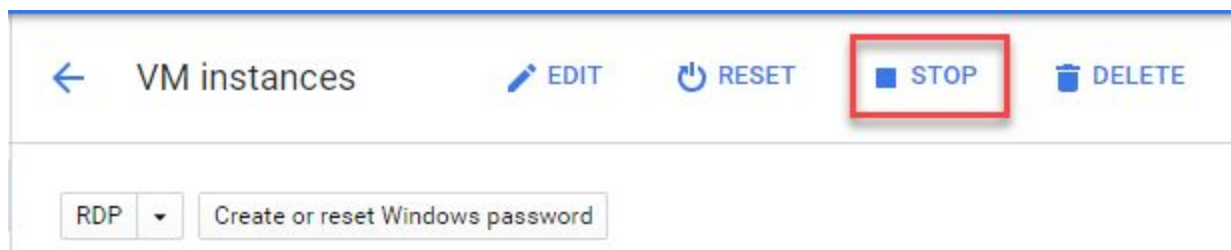
In a perfect world, we could also choose clockspeed so that tasks finish as quickly as possible when they do get CPU time.

Forget it Jake, it's Clowntown.

We can't do anything about the lock waits with hardware, really, we'll have to troubleshoot those later. Locks may happen and release faster with pages already in memory and more CPU oomph, but they'll still happen.

We're going to shut it down and look at what we can do to help this thing out. First, we'll stop and disable our workload Agent Jobs so they don't mess with us right away when we start back up.

To make hardware changes, we have to stop our server and shut it down.



Now we can edit it. The changes we'll make include notching the machine size up one.

First we'll change our VM configuration to 16 cores and 40 GB of RAM.

We can get away with this because our database is only about 24 GB at the moment, and we can pad queries and other operations, like junk in TempDB, with the remaining memory.

This means we can give 36 GB to SQL (remember that Windows needs RAM, too). If our database were larger, we would likely need more RAM, depending on usage patterns.


We're also going to add a 50 GB Persisted SSD for TempDB. Again, this is because we're not working with a ton of data. Your TempDB needs will vary. You may even need a bigger drive despite not needing a very large TempDB just for the additional throughput.

Create a disk

Name 

tempdb

Description (Optional)

Disk Type 

SSD persistent disk

Source type 

Image

Snapshot

None (blank disk)

Size (GB) 

50

Estimated performance 

Operation Type	Read	Write
Sustained random IOPS limit	1,500	1,500
Sustained throughput limit (MB/s)	24	24

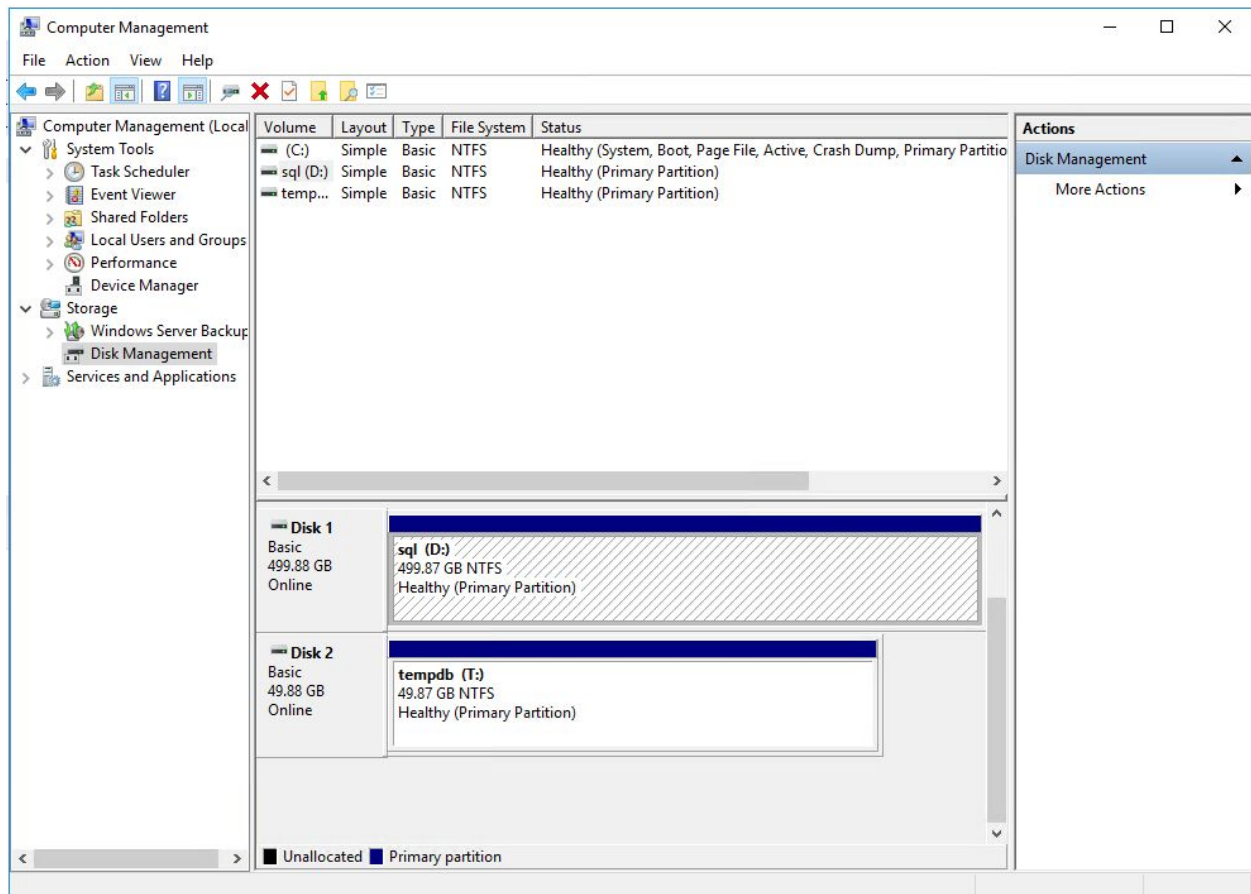
Encryption 

Automatic (recommended)

This is basically a Gutenberg Press at 24 MB/s, but the IOPS are much higher than Standard Persisted Disks.

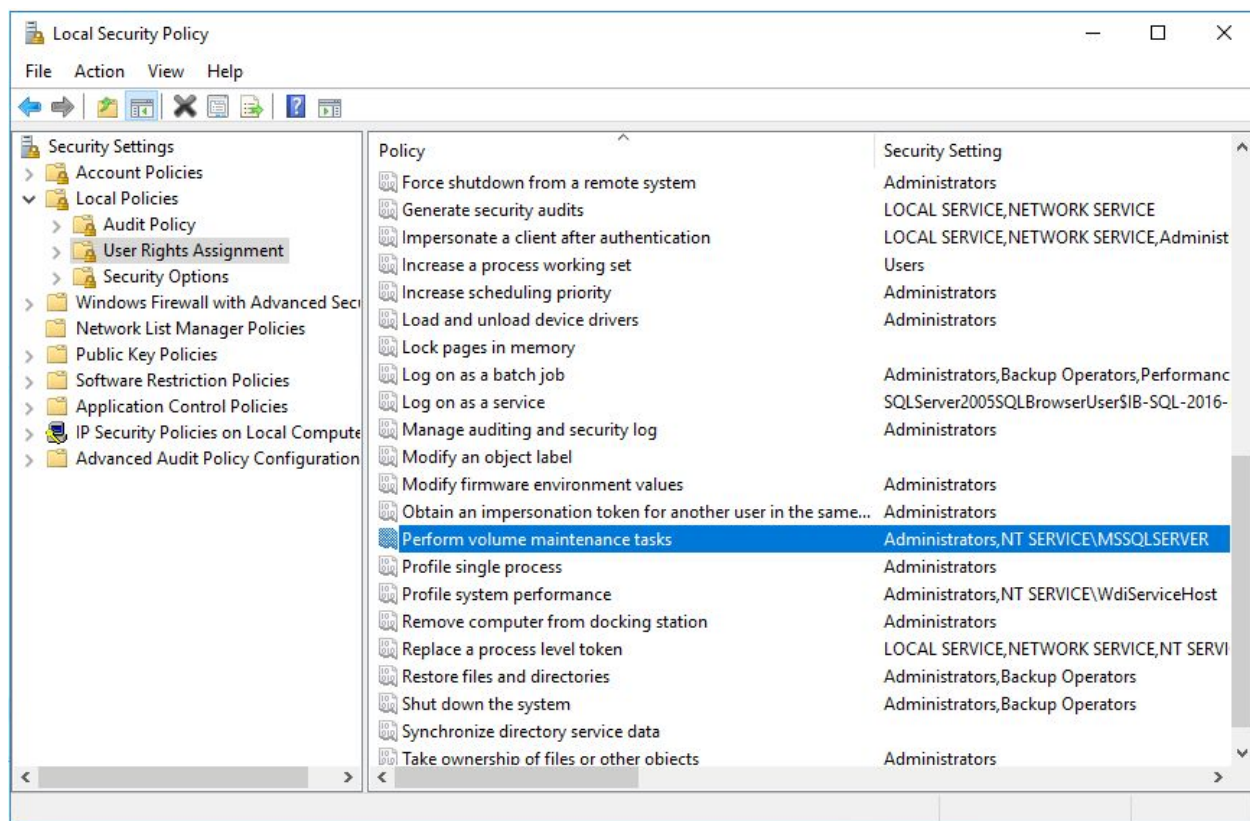
Configuring SQL Server to Use Our Newfound Power

When we start back up, we'll have to go into Computer Management to initialize and bring the new drive online. I gave it the letter T: for TempDB. T! T is for TempDB! HA HA HA!



At this time, GCE's images don't automatically assign Instant File Initialization rights to the login account. When you try to move files, it can take a long time depending on their size.

Under the local security policy (Start > Run > secpol.msc), make sure your service account has the Perform volume maintenance tasks privilege:



TempDB

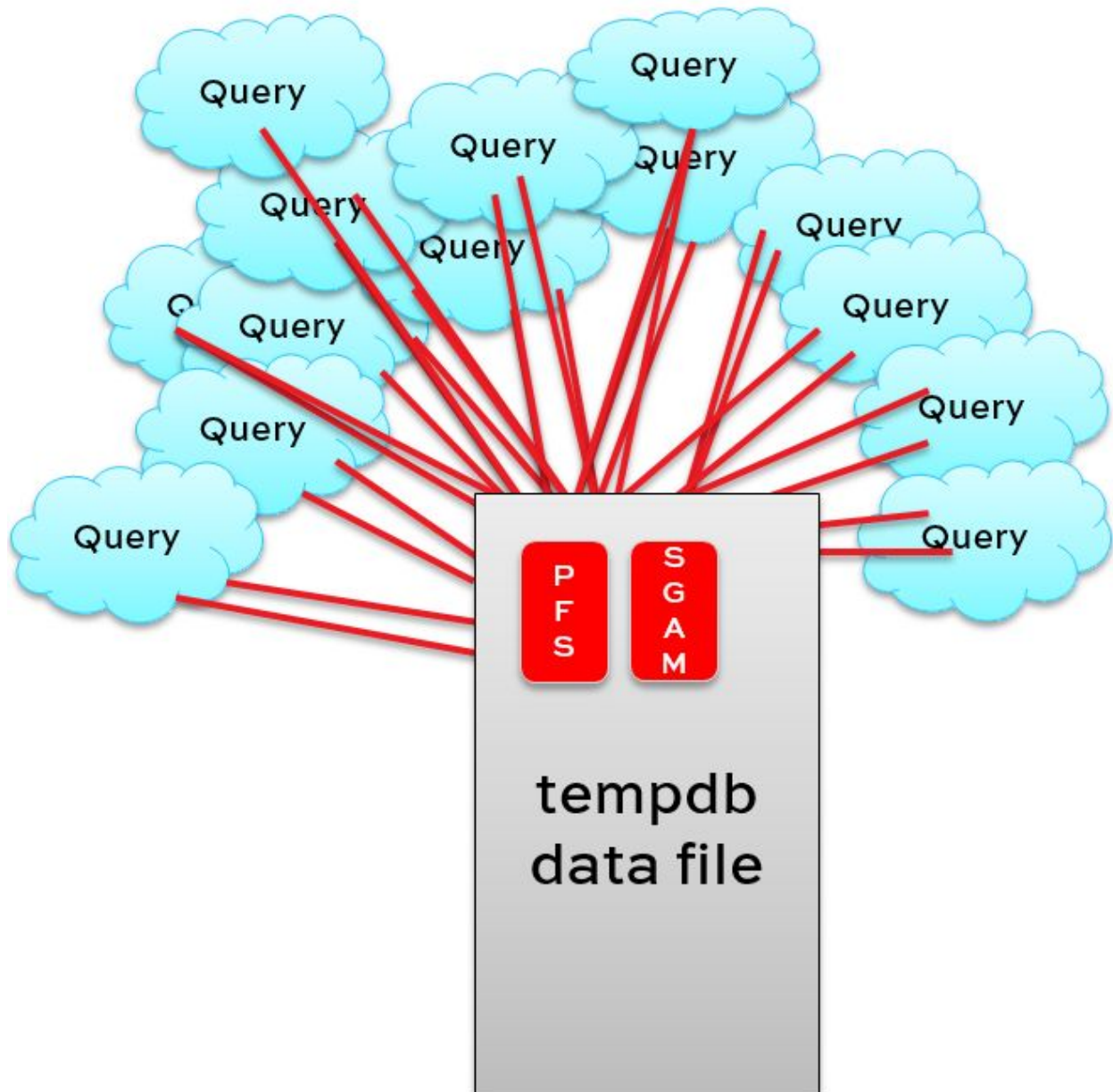
We're not going to add any more files at this point. There's some old advice about "one TempDB data file per core", but that's from the days when having 16 cores was almost as impressive as having a cell phone with a web browser.

These days, 8 files is where most folks see performance level off, if 4 isn't cutting it. Just make sure they're all the same starting size and have all the same autogrowth settings. Having unevenly sized files can lead to uneven query distribution across them.

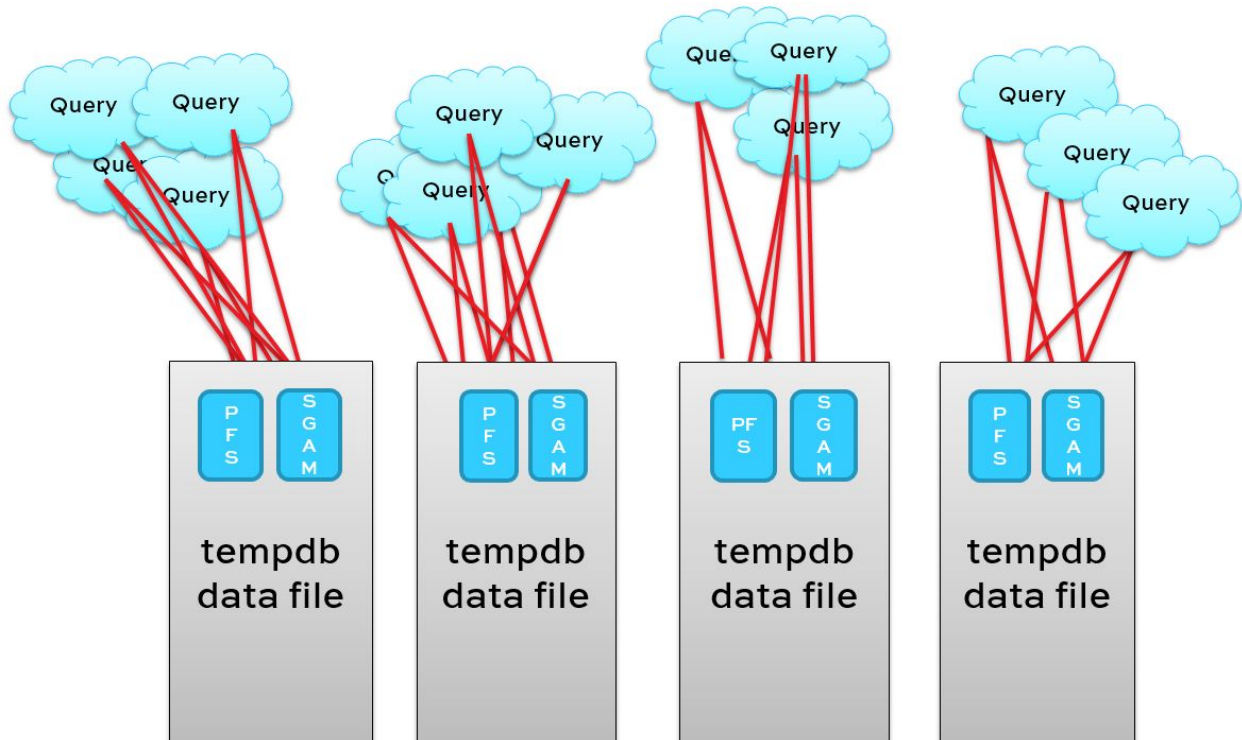
This is because of the way SQL Server assigns work to the files. It uses something called the "proportional fill algorithm" to figure out which file has the most free space in it. Since TempDB workloads just that, temporary, the largest file will usually have the most free space, and get the most attention.

That basically puts you back at square one -- you've got the same contention you were trying to avoid by creating multiple files in the first place.

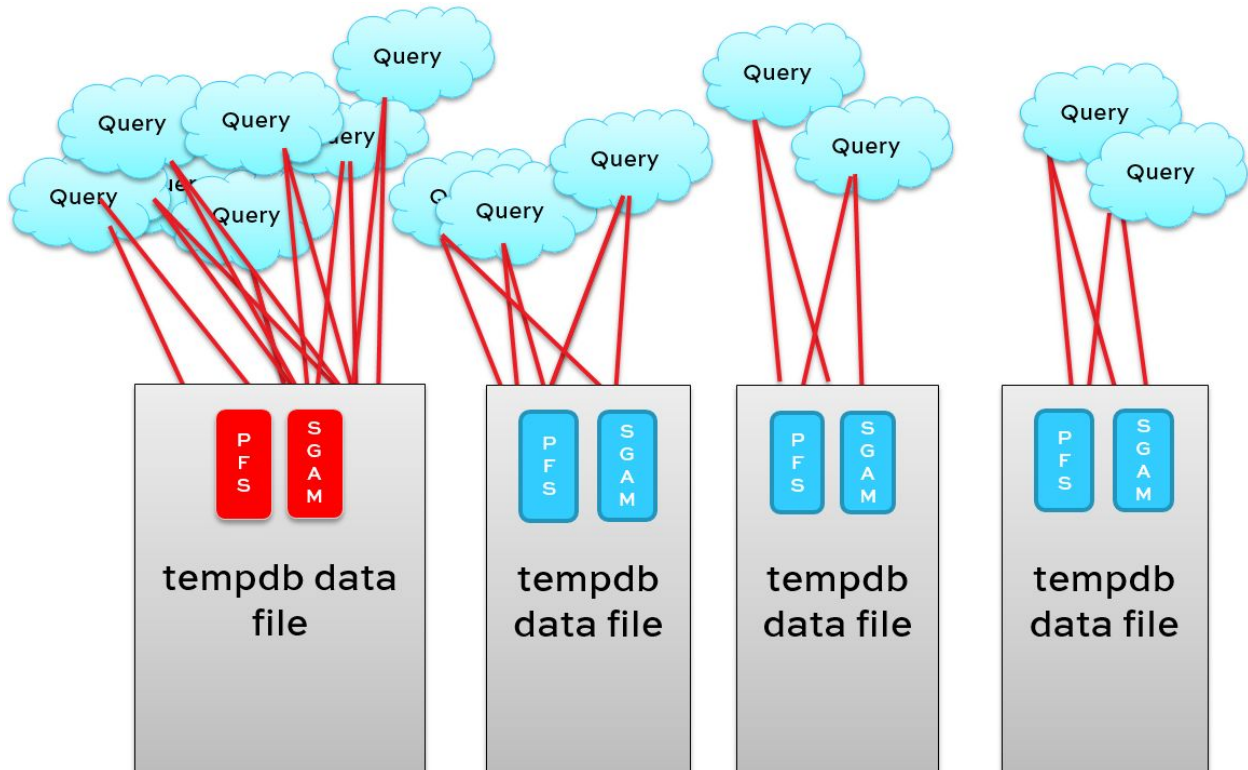
TempDB with one file:



TempDB with multiple evenly sized data files



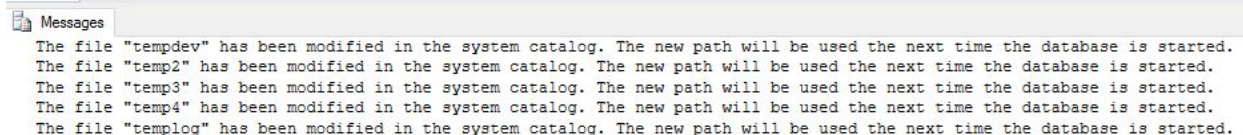
TempDB with multiple uneven data files



Moving TempDB

```
ALTER DATABASE tempdb MODIFY FILE (NAME='tempdev', FILENAME =  
N'T:\tempdev.mdf', SIZE = 8GB);  
ALTER DATABASE tempdb MODIFY FILE (NAME='temp2', FILENAME = N'T:\temp2.ndf',  
SIZE = 8GB);  
ALTER DATABASE tempdb MODIFY FILE (NAME='temp3', FILENAME = N'T:\temp3.ndf',  
SIZE = 8GB);  
ALTER DATABASE tempdb MODIFY FILE (NAME='temp4', FILENAME = N'T:\temp4.ndf',  
SIZE = 8GB);  
  
ALTER DATABASE tempdb MODIFY FILE (NAME='templog', FILENAME =  
N'T:\templog.ldf', SIZE = 8GB);
```

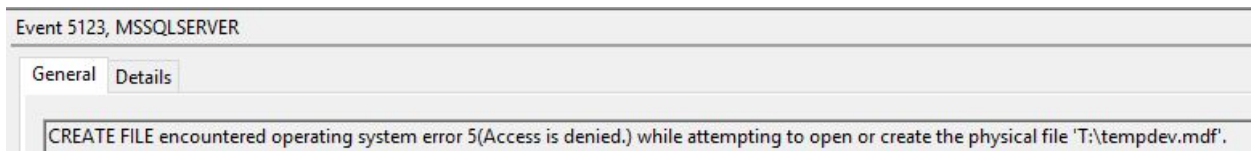
When this finished, we'll have to recycle SQL to use the new location.



Messages

The file "tempdev" has been modified in the system catalog. The new path will be used the next time the database is started.
The file "temp2" has been modified in the system catalog. The new path will be used the next time the database is started.
The file "temp3" has been modified in the system catalog. The new path will be used the next time the database is started.
The file "temp4" has been modified in the system catalog. The new path will be used the next time the database is started.
The file "templog" has been modified in the system catalog. The new path will be used the next time the database is started.

You may also have to give your SQL Service account permission to create files in the new directory. If you don't, you'll see an error like this in Event Viewer.



Event 5123, MSSQLSERVER

General Details

CREATE FILE encountered operating system error 5(Access is denied.) while attempting to open or create the physical file 'T:\tempdev.mdf'.

Max Server Memory

You'll want to adjust Max Server Memory to a size appropriate to [how much memory your server has](#). In our case, We have now have an 8 core, 40 GB RAM server, so we'll set it to 36 GB (36 * 1024). Remember, please, please, please remember that this setting is in MB. If you enter 36 here, your server is going to be pissed at you. I'm going to be pissed at you. Your boss will probably beat you with your keyboard.

THIS.

SETTING.

IS.

IN.

MB.

CPU

We're going to leave MAXDOP and Cost Threshold for Parallelism alone here. This is to allow for greater concurrency. More queries can run across our new cores simultaneously if they're only using four cores.

We still want the same queries to go parallel -- raising CTFP to a higher number would just force some queries to potentially run single threaded that we really want to stay parallel. That doesn't help us increase performance. In fact it may do more harm than good.

Baseline #2: PAGEIOLATCH Gone, SOS_SCHEDULER_YIELD Still Here

After re-running our example workload for a bit, we can see how things are stacking up. We didn't change any queries or indexes, we doubled CPU and RAM, and isolated TempDB on SSD.

We made these changes based on evidence we collected, not wild guesses.

Increasing RAM serves two purposes: We fit all our data into memory, plus some extra to manage other caches. It also gives our queries, and TempDB, extra workspace to avoid spilling out to disk.

Likewise, isolating TempDB to its own SSD gives us the fastest possible spills, if they do still happen.

Increasing CPU count helps us keep the workload spread over a larger number of cores, and hopefully reduces CPU contention.

No changes we made will help the locking problems. While troubleshooting those is also outside the scope of this whitepaper, we'll enable a setting in SQL Server called RCSI (Read Committed Snapshot Isolation) later to examine any impact on CPU and TempDB, and to remove reader/writer blocking from the wait stats noise.

How'd we do? Well, let's look at the numbers, Chuck.

We're still beating the tar out of the CPU. This could be a query/index tuning opportunity, but our job is just to figure out which server configuration can handle our workload as-is.

FindingsGroup	Finding	URL	Details
sp_BlitzFirst 2016-12-10 00:00:00.0000000 +00:00	From Your Community Volunteers	http://FirstResponderKit.org/	Click To See Details -- We hope you found this ...
Server Performance	High CPU Utilization	http://www.BrentOzar.com/go/cpu	Click To See Details -- 97%. Ring buffer details: ...
Wait Stats	SOS_SCHEDULER_YIELD	http://www.brentozar.com/sql/wait-stats/#SOS_SCHE...	Click To See Details -- For 969 seconds over th...
Wait Stats	LCK_M_IS	http://www.brentozar.com/sql/wait-stats/#LCK_M_IS	Click To See Details -- For 296 seconds over th...
Wait Stats	CXPACKET	http://www.brentozar.com/sql/wait-stats/#CXPACKET	Click To See Details -- For 271 seconds over th...
Wait Stats	LCK_M_X	http://www.brentozar.com/sql/wait-stats/#LCK_M_X	Click To See Details -- For 31 seconds over the...
Wait Stats	LCK_M_IX	http://www.brentozar.com/sql/wait-stats/#LCK_M_IX	Click To See Details -- For 30 seconds over the...
Server Info	Batch Requests per Sec	http://www.BrentOzar.com/go/measure	Click To See Details -- 2 - ?>

During a 30 second sample, we still had a lot of CPU and locking waits, but our PAGEIOLATCH_** waits have all but disappeared. We still have PAGELATCH_** waits from using TempDB, but that's okay. The response time is good on the SSDs.

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
29	SOS_SCHEDULER_YIELD	969.1	4.2	968.9	100.0	50887	19.0
29	LCK_M_IS	296.9	1.3	0.0	0.0	17	17466.0
29	CXPACKET	271.5	1.2	0.1	0.0	63	4309.3
29	LCK_M_X	31.8	0.1	0.0	0.0	2	15924.0
29	LCK_M_IX	30.9	0.1	0.0	0.0	2	15432.0
29	PAGELATCH_UP	7.2	0.0	1.8	25.0	822	8.7
29	PAGELATCH_SH	5.5	0.0	2.9	52.7	1054	5.2
29	MSQL_XP	3.0	0.0	0.0	0.0	8	378.6

Going to waits since startup, it tells a similar story. Disk-related waits fell way down (the response time still stinks, but it matters less).

Hours Sample	wait_type	Wait Time (Hours)	Per Core Per Hour	Signal Wait Time (Hours)	Percent Signal Waits	Number of Waits	Avg ms Per Wait
21.5	SOS_SCHEDULER_YIELD	310.8	0.0	310.7	100.0	87025102	12.9
21.5	LCK_M_IS	183.3	0.0	0.0	0.0	56443	11688.0
21.5	CXPACKET	172.9	0.0	2.6	1.5	6459030	96.4
21.5	LCK_M_X	33.0	0.0	0.0	0.0	6914	17185.4
21.5	LCK_M_IX	8.8	0.0	0.0	0.0	5193	6068.9
21.5	PAGEIOLATCH_SH	6.5	0.0	0.1	1.6	621579	37.8
21.5	PAGELATCH_UP	0.7	0.0	0.2	30.9	726622	3.6

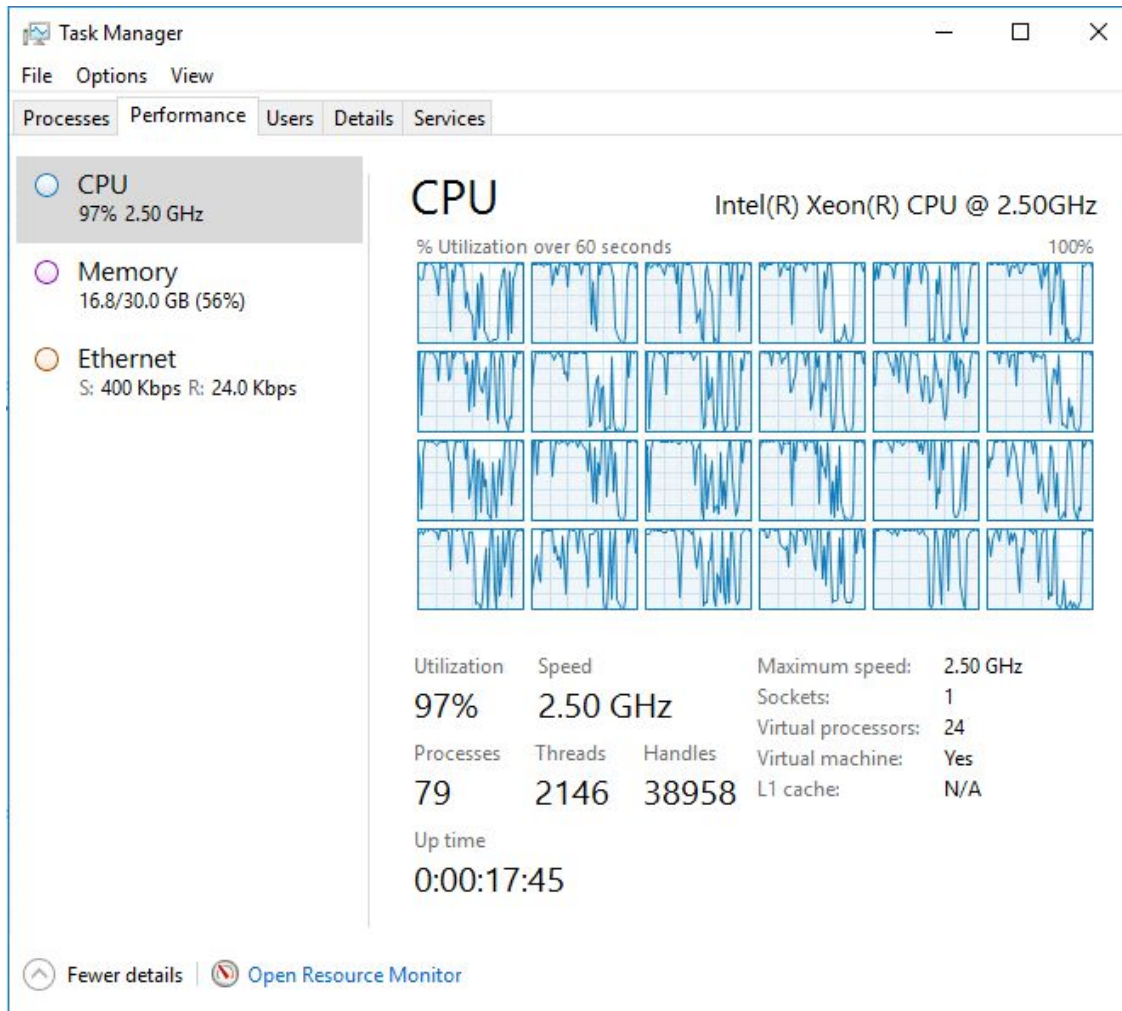
With Memory and Disk waits largely resolved, let's figure out how many CPUs our workload needs.

We're going to stop and reconfigure. Since our CPU was still pegged up around 97%, we're going to bump it to 24 CPUs, with the same amount of RAM.

This obviously didn't give us the throughput we were hoping for.

We're not going to try 12 or 16 cores first just based on how plowed the CPUs still are. If we overshoot in testing, we can either scale back, or keep the higher grade hardware for future proofing or unforeseen (are they ever foreseen?) performance degradations. Let's say a bad plan gets into cache and CPUs take a hit -- the larger hardware would absorb the spike better than merely adequate hardware.

CPU is still taking an Orson Welles' liver style beating, and struggling to keep up. According to Task Manager, we're using the second highest clockspeed CPUs available, at 2.5GHz. The next step up is only 2.6 GHz, so it wouldn't make a noticeable difference.



Off the bat, CXPACKET waits are high and long, which are only good adjectives during a Home Run Derby.

Hours Sample	wait_type	Wait Time (Hours)	Per Core Per Hour	Signal Wait Time (Hours)	Percent Signal Waits	Number of Waits	Avg ms Per Wait
0.1	CXPACKET	1.3	0.0	0.0	0.0	1421	3405.9
0.1	PAGEIOLATCH_SH	0.6	0.0	0.0	0.5	47183	45.8

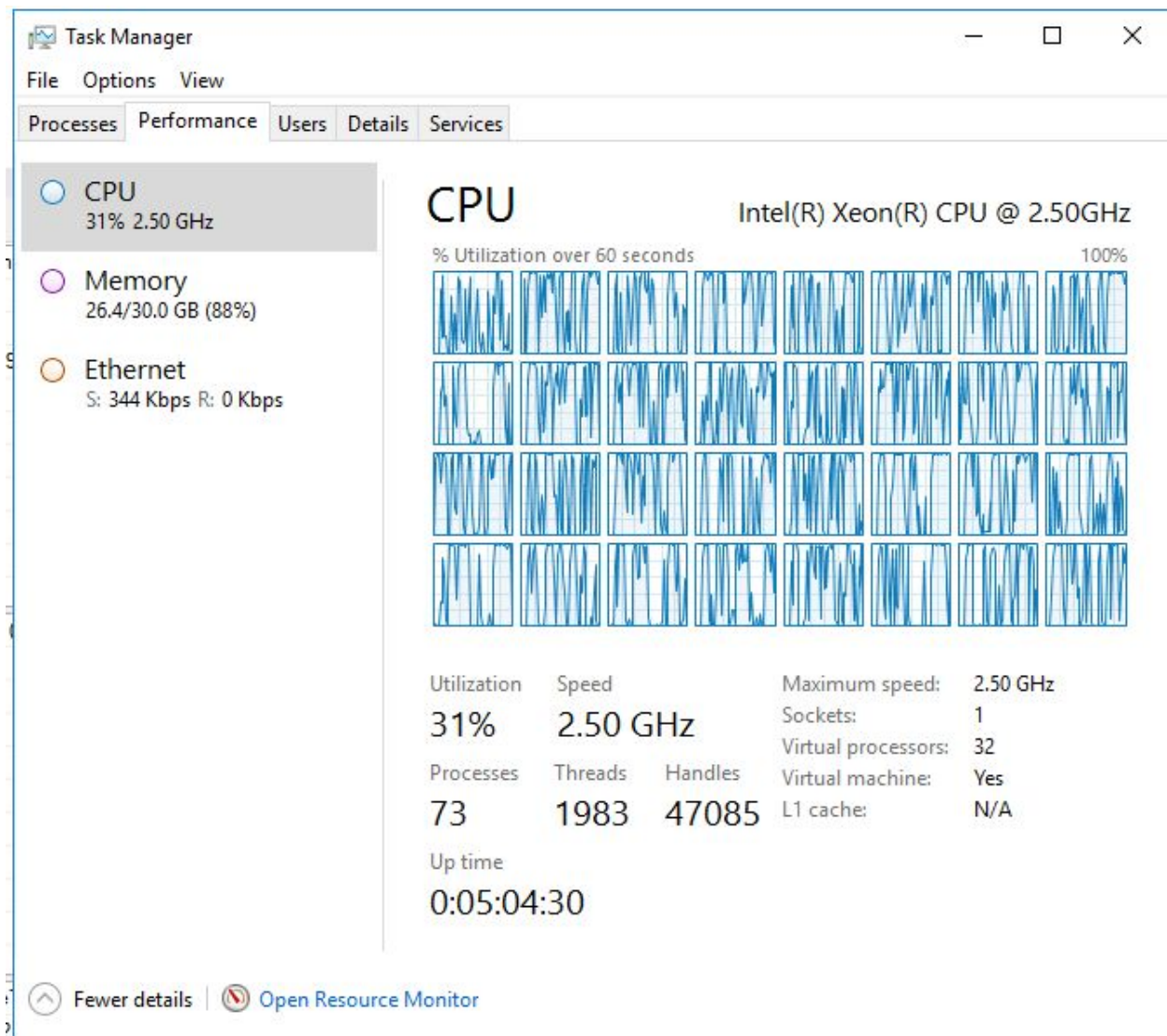
Mitigation #2: Throwing Cores At SOS_SCHEDULER_YIELD Waits

Our next logical increment is to 32 cores, which is the CPU capacity of a GCE instance currently. This is the where, why, and how of determining application compatibility with the cloud.

We're past the point where more memory, faster disks, and all the Fibre Channel in Taiwan can help you. If the CPUs can't keep up, they can't keep up.

Baseline #3: High CPU, and Now LCK* Waits

Spinning up 32 cores does help, as CPUs are no longer constantly pegged at 80% and above, however they're still under significant pressure.



This concludes how far we can take the CPUs, and leaves gauging acceptable resource usage to the consumer.

If you have the wherewithal to offload reads to multiple locations, or have a workload that doesn't push CPUs quite as hard, you may find the cloud a hospitable location for your data.

Some applications are not a good fit without further tuning, and some may just need the kind of hardware and configuration that belongs in a Cloudless infrastructure.

One thing that did come out of upping the CPUs to their limit is that, even though the spikes often hit 100%, and resource usage is typically >60%, they are responding much better to requests. In a 30 second window, CXPACKET and SOS_SCHEDULER_YIELD no longer take the long average waits that they did with fewer CPUs assigned.

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
29	CXPACKET	183.9	0.2	0.1	0.1	212	867.6
29	LCK_M_X	61.0	0.1	0.0	0.0	18	3389.6
29	LCK_M_IS	58.2	0.1	0.0	0.0	12	4850.3
29	SOS_SCHEDULER_YIELD	37.3	0.0	37.3	100.0	62412	0.6
29	LCK_M_IX	15.1	0.0	0.0	0.0	1	15138.0
29	LATCH_EX	0.1	0.0	0.0	0.0	86	0.7

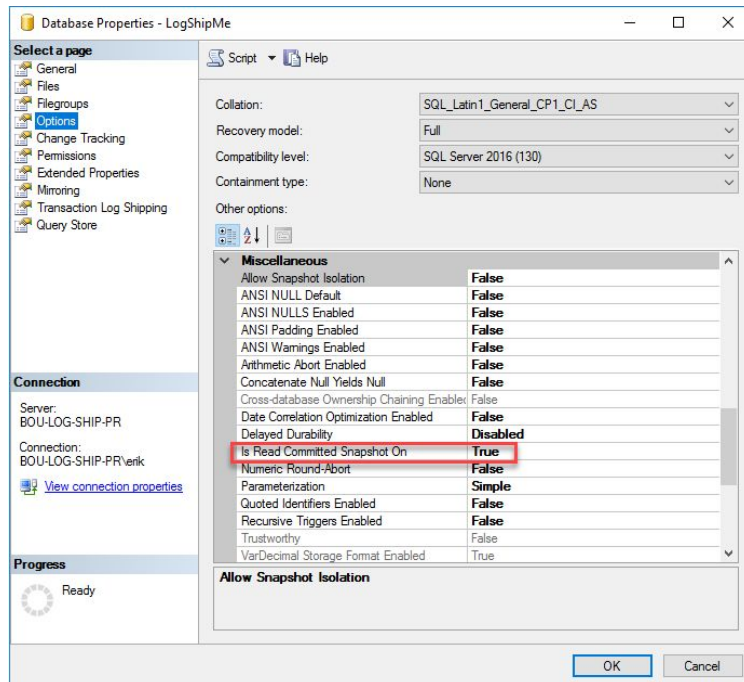
This remains true of the workload overall.

Hours Sample	wait_type	Wait Time (Hours)	Per Core Per Hour	Signal Wait Time (Hours)	Percent Signal Waits	Number of Waits	Avg ms Per Wait
5.6	CXPACKET	47.3	0.0	0.3	0.7	3227472	52.8
5.6	LCK_M_IS	17.9	0.0	0.0	0.0	7905	8169.7
5.6	SOS_SCHEDULER_YIELD	15.6	0.0	15.6	99.9	48955670	1.1
5.6	LCK_M_X	11.0	0.0	0.0	0.0	10580	3752.7
5.6	LCK_M_IX	3.1	0.0	0.0	0.0	1668	6749.6
5.6	PAGEIOLATCH_SH	0.7	0.0	0.0	0.4	69527	37.5
5.6	PAGELATCH_UP	0.1	0.0	0.0	35.6	2066336	0.2

The next thing we're going to experiment with is turning on RCSI to examine a workload free of locking issues, and how it impacts CPU and TempDB.

Mitigation #3: Fixing LCK* Waits with Optimistic Isolation Levels

Another header, another caveat. This isn't the optimistic locking whitepaper. If you want to learn more about it, [head on over here](#). If not, skip this section. Or just read because you're interested. Plus there are more pictures. Everyone likes pictures. Like this one, of database properties.



This portion of the whitepaper is primarily geared towards people already using optimistic isolation levels, or people starting a new application where they can start using it from the get-go.

Unless your application was written to be able to use another RDBMS like Oracle, which uses optimistic locking by default, your code will need careful testing to make sure it's compatible with RCSI, or Snapshot Isolation. They're great features, but they don't fit every codebase, simply because you can hit race conditions or run into different behavior than you'd get when readers and writers block each other.

No blocking?! Tell me more!

Well, less blocking. Readers and writers will be cool. When modification queries come along (DUIs: Deletes, Updates, Inserts), previously committed versions of rows are stored in TempDB, so when read queries come along, they get data from there, rather than sitting around thumbs a-twiddle waiting for locks to be released.

Writers will still block each other, and you can still run into deadlocks. It doesn't solve every problem. Eventually you'll probably have to engage in the thankless task of query and index tuning.

Don't you just hate that?

Anyway, same workload, same configuration, but with RCSI turned on! How do we look?

Boy howdy! Those lock waits are just about gone. We just have a little wait time between concurrent writers. We no longer have any serious bottlenecks that we can do anything about with hardware. Remember, we can't add more CPUs, or faster CPUs.

Hours Sample	wait_type	Wait Time (Hours)	Per Core Per Hour	Signal Wait Time (Hours)	Percent Signal Waits	Number of Waits	Avg ms Per Wait
4.0	CXPACKET	84.5	0.0	0.6	0.7	2719483	111.9
4.0	SOS_SCHEDULER_YIELD	39.5	0.0	39.5	99.9	77609339	1.8
4.0	LCK_M_X	2.5	0.0	0.0	0.0	5806	1543.3
4.0	PAGELATCH_UP	2.1	0.0	0.7	31.7	4086225	1.8
4.0	PAGELATCH_SH	1.8	0.0	1.1	60.0	3054181	2.1
4.0	WRITELOG	1.0	0.0	0.3	33.6	1611057	2.3
4.0	PAGEIOLATCH_SH	0.9	0.0	0.0	0.8	73368	46.5
4.0	PAGEIOLATCH_EX	0.9	0.0	0.0	0.1	131151	24.7
4.0	LATCH_EX	0.4	0.0	0.0	9.0	160015	8.5
4.0	PAGELATCH_EX	0.2	0.0	0.1	50.8	511819	1.7

With optimistic locking in place, the queries that insert data into temp tables get held up less, plus we have rows being versioned in TempDB, so PAGELATCH_** waits are more prevalent. Waits there are behaving fairly well. oop

Our Standard Persisted Disk still eats it, but since the great RAMming of the last section when we bumped it up to 40 GB, we're leaning way less on disk, so it's still pretty low. Overall, I'm pretty happy with how this part looks.

Disks, however, are still being disks. I'm not *thrilled* with some of these stalls. But this is the stuff you need to see so you know when you might need to either increase the size of disk currently in place, or switch to an Persisted SSD to get better throughput.

Sample (seconds)	File Name	Drive	# Reads/Writes	MB Read/Written	Avg Stall (ms)	file physical name
14338	LogShipMe [ROWS]	D:	298090	39289.9	144	D:\Data\LogShipMe.mdf
14338	master [ROWS]	C:	100	6.0	45	C:\Program Files\Microsoft SQL Server\MSSQL13.MSS...
14338	MSDBData [ROWS]	C:	812	43.8	38	C:\Program Files\Microsoft SQL Server\MSSQL13.MSS...
14338	LogShipMe_log [LOG]	D:	7164	6900.1	35	D:\Log\LogShipMe_log.ldf
14338	ReportServer_log [LOG]	C:	251	3.8	22	C:\Program Files\Microsoft SQL Server\MSSQL13.MSS...
14338	temp3 [ROWS]	T:	10661	2127.3	63	T:\temp3.ndf
14338	temp4 [ROWS]	T:	10439	1766.0	62	T:\temp4.ndf
14338	temp2 [ROWS]	T:	10851	2171.2	55	T:\temp2.ndf
14338	tempdev [ROWS]	T:	11093	2178.1	49	T:\tempdev.mdf
14338	LogShipMe [ROWS]	D:	1526442	14582.7	16	D:\Data\LogShipMe.mdf

Batch Requests / Second

When all else fails, one metric you can focus on is Batch Requests / Second.

We didn't discuss it much up until now, because we were focusing on resource consumption and mitigating bottlenecks. An important indicator of if your cloud server can cut it, is if it can match or exceed the throughput of your Cloudless server.

With each iteration, we moved the needle on how many queries our server could push through. That totally makes sense! More hardware, less locking, more queries. Hooray. We did it. Let's look at how that trended.

In sp_BlitzFirst, you can get this from the 'headline news' section up at the top, or from the PERFMON section down towards the bottom. I'm grabbing from the PERFMON section because the screen caps a little more clear. The screen caps have been abridged to cut out irrelevant columns.

First iteration: (4 CPU, 15 GB RAM)

Pattern	object_name	counter_name	ins	ValueDelta	ValuePerSecond
PERFMON	SQLServer:Locks	Lock Wait Time (ms)	O	40077	1335.9000000000...
PERFMON	SQLServer:Locks	Lock Waits/sec	-	9	0.30000000000000
PERFMON	SQLServer:Locks	Lock Waits/sec	O	9	0.30000000000000
PERFMON	SQLServer:Memory Mana...	Total Server Mem...		2102480	70082.666666666...
PERFMON	SQLServer:SQL Errors	Errors/sec	-	32	1.06666666666666
PERFMON	SQLServer:SQL Statistics	Batch Requests/s...		127	4.23333333333333
PERFMON	SQLServer:SQL Statistics	SQL Compilations/...		62	2.06666666666666
PERFMON	SQLServer:SQL Statistics	SQL Re-Compilati...		14	0.46666666666666

Second iteration (24 CPU, 30 GB RAM)

Pattern	object_name	counter_name	ins	ValueDelta	ValuePerSecond
PERFMON	SQLServer:Locks	Number of Deadlo...	.	5	0.172413793103
PERFMON	SQLServer:Locks	Number of Deadlo...	i	5	0.172413793103
PERFMON	SQLServer:Memory Mana...	Granted Workspa...		264	9.103448275862
PERFMON	SQLServer:Memory Mana...	Total Server Mem...		51496	1775.724137931034
PERFMON	SQLServer:SQL Errors	Errors/sec	.	91	3.137931034482
PERFMON	SQLServer:SQL Statistics	Batch Requests/s...		237	8.172413793103
PERFMON	SQLServer:SQL Statistics	SQL Compilations/...		112	3.862068965517
PERFMON	SQLServer:SQL Statistics	SQL Re-Compilati...		15	0.517241379310

Third iteration (32 CPU, 40 GB RAM)

Pattern	object_name	counter_name	ValueDelta	ValuePerSecond
PERFMON	SQLServer:Locks	Lock Waits/sec	40	1.379310344827
PERFMON	SQLServer:Locks	Number of Deadlo...	7	0.241379310344
PERFMON	SQLServer:Locks	Number of Deadlo...	7	0.241379310344
PERFMON	SQLServer:Memory Mana...	Total Server Mem...	3031112	104521.10344827...
PERFMON	SQLServer:SQL Errors	Errors/sec	90	3.103448275862
PERFMON	SQLServer:SQL Statistics	Batch Requests/s...	329	11.344827586206
PERFMON	SQLServer:SQL Statistics	SQL Compilations/...	121	4.172413793103
PERFMON	SQLServer:SQL Statistics	SQL Re-Compilati...	15	0.517241379310

Fourth iteration (32 CPU, 40 GB RAM, RCSI)

Pattern	object_name	counter_name	ValueDelta	ValuePerSecond
PERFMON	SQLServer:Locks	Number of Deadlo...	7	0.241379310344
PERFMON	SQLServer:Locks	Number of Deadlo...	7	0.241379310344
PERFMON	SQLServer:Memory Mana...	Total Server Mem...	2149360	74115.862068965...
PERFMON	SQLServer:SQL Errors	Errors/sec	126	4.344827586206
PERFMON	SQLServer:SQL Statistics	Batch Requests/s...	406	14.000000000000
PERFMON	SQLServer:SQL Statistics	SQL Compilations/...	174	6.000000000000
PERFMON	SQLServer:SQL Statistics	SQL Re-Compilati...	15	0.517241379310
PERFMON	SQLServer:Transactions	Longest Transacti...	60	2.068965517241

With each step up, we were able to push through more queries. In the end, we were pushing through nearly 4x the queries per second that we were on the original hardware. Enabling RCSI got us a little more breathing room as well.

While both RCSI and Snapshot Isolation can have a profound impact on larger read queries that are often blocked, they can also really help smaller queries performing singleton lookups, even if they're joining multiple tables across a single unique key.

Again, it comes down to compatibility. You need to carefully test your workload to make sure it can successfully cohabitate with RCSI. If not, you can use Snapshot Isolation and only set that isolation level for queries that can cope with reading versioned rows.

Wrap It Up

We got to a pretty good place in the end. We went from a too-small and ill-configured box to a much more suitably sized and well-configured box, and saw pretty dramatic differences. At least, I'd consider not waiting for 8.5 seconds on CXPACKET to be dramatic.

I'm sure the CPUs would agree, if they weren't busy running around with their hair on fire.

Hopefully you learned a bit from this. I had a lot of fun writing it for you, [your name here].

By now, you should be comfortable with using our [free scripts](#) to measure your SQL Servers, which common wait stats map to which resources, and how to size a GCE VM based on them.

If you clicked on all the links, you probably learned about some other stuff, like apparently writing mediocre blog posts is a career option.

While each resource is important, you need to focus on the resources that are currently bottlenecks for you until you uncover the next one. There will always be a next one.

Eventually you'll hit one that can't currently be overcome via hardware in the cloud without query and index tuning, and even then it may not ever be totally resolved. This is where scaling out with technology like Availability Groups can help, but cost can be the deciding factor.

One of the most important lessons to take from this is that cloud VMs are not like physical servers. Remember -- cattle, not pets. You don't name them, you don't let them in the house, you just feed them and wait until the day you turn them into food.

If you'd like our help, drop us a line:

- Learn how our SQL Critical Care® process works:
<https://www.brentozar.com/sql-critical-care/>
- Email us at Help@BrentOzar.com, or use our contact form:
<https://www.brentozar.com/contact/>

Thanks for reading!