# To hyperscale or not to hyperscale, that’s the question

Hi everyone. Thank you so much for taking time out of your busy schedule and choosing this session out of all the available session going on during this Pass community Summit. Many thanks to Redgate for getting this event going again.

In this session I’m taking you on a fast trip with Hyperscale. I’ll quickly touch the design, a number of pros and cons we ran into when using this database tier and a bug we ran into.

My name is Reitse Eskens and I’m a dataplatform consultant with Axians in The Netherlands. Yes, I’m certified in both Azure and on-premises SQL server and do stuff when I’m not staring at a screen. Though not as much as I’d like to. You can follow me on twitter and linkedin if you want to.

Now, first up the general design of Hyperscale. When you look at the general, provisioned and serverless databases on Azure, there are some similarities between those databases in choosing CPU cores and with more cores comes more memory and higher log write limits. With that I mean that you’ll get more IO on your log files and as ALL AZURE DATABASES are in full recovery model, the log file gets hit, hard. And a lot. Because you know that every transaction gets logged. The upside is that you can do a restore of your database on almost every point in time you can imagine. We’ve tried it out and it works like a charm.

The Hyperscale databases are designed differently, where there’s a separate log service that removes that huge bottleneck. Secondly, I haven’t been able to find memory restrictions. They should be there, but both documentation and testing didn’t really reveal much, other than the server I was running on seemed to have about 450 GB of memory. How much of that was mine, no idea. Resource semaphore waits on our queries (we’re running ETL processes in the night and reports by day) were uncommon. This is an indication that there’s plenty of memory available. The rest of the design with a number of servers is different compared to the other database options as well, but the documentation is available for your own deep dive.

When we were running our project and were scaling up from provisioned and even business critical to Hyperscale, we found that we lost all the log related waits. Cool, because those were killing our processes. We could scale by query. This meant that each part of the ETL could, in theory, scale up or down to get the best bang for the buck. When you scale, you pay for the cores you’re using. It’s not cheap to begin with, so when you know the load is low at certain periods, you can save money by scaling down to 2 cores.

But, when you start scaling, what you’re effectively doing is starting up a new instance in the background. When that instance is available, you’re failing over to that instance. But, failing over means a short break in the connection, right? Right. So your application needs some logic to handle that. A simple wait or pause suffices but you need to have that in place to prevent your ETL from breaking. With a failover, some DMV’s lose their contents. Index usage stats were one we had issues with because I had no way of telling if indexes were or weren’t used. I got many indexes that had only been updated (by the ETL process) but never read. When I knew I’d used them the day before. I’ve lost buffered plans as well. Luckily Query Store doesn’t suffer from these failovers. At least not that I could notice.

You also need to remember to scale down when you’re going way up. Running a 48 core Hyperscale is bad news for your credit card.

Now, I might have lured you in with a bug. There’s something nasty going on in the background. Hyperscale works with a sort of partition in the background, files have a certain sizing. When you upgrade a regular Azure database to Hyperscale, you don’t get a number of files, but just one file. This is a huge performance hit. When you suffer from this bug, you can detect it with this wait stat:

When you start using hyperscale, start a database from scratch. Azure has promised to fix this bug this year. I don’t know if that has happened but you’re better off safe than sorry.