ALLAN THRAEN | 5 years ago | PDF |

.NET Development Azure Tips and Tricks

AZURE STORAGE PERFORMANCE SHOWDOWN (POST 2)

In this second post of my performance series looking at Azure storage we're going to take a good look at Read speeds for the various storage types.

Summary: This is the second post in a series of post doing a CRUD performance and scaling comparison of various ways to persist data in Azure. In the first post we took a look at creating content, and ended up with a bunch of data stores with 1 million records in each consisting of structured questions from StackExchange. With CosmosDB coming in at an impressive first place overall but being chased by SQL Azure and Table Storage.

For this second round, the setup is still the same - My test-machine is still a virtual machine in Azure and I'm testing against services in the same zone to keep traffic time to a minimum. Blob Storage and Table storage are running in the "Hot" mode of storage v2 (I believe this means that they are on HDDs, not SSDs or archive tapes). After the multithread indexing text in the first post I've scaled Cosmos back to reserving 1000 Request/Units (RUs) - but when it comes to SQL Server I decided to split it up a bit, so I'm both looking at a 'normal' SQL Azure server running S0 with 10 DTU and a 'Power' version running with S3 and 100 DTU to see it's scaling.

Once again I took the liberty of doing a simple twitter poll to see the expectations when it comes to read speeds:

Developers! Here's the next @Azure storage poll for my little showdown:
Which service do you expect to be fastest at finding and reading 1000 random records, by indexed ID in a 1.000.000 row dataset?

— Allan Thraen (@athraen) September 21, 2018

A bit more of a spread this time - but less on the pure blob storage - probably based on last posts data I'm guessing.

Scenario 1 - Reading data by ID

So - for the first scenario, I wanted to mimick the typical scenario we so often see on websites. You have some content, probably produced by editors - or by other users - structed in some way and stored in the database. Now, we want to fetch a specific piece of content from storage in order to show it to a requesting user.

We already know it's ID - maybe it's the url-slug, or maybe it's in a list somewhere. And of course the ID is an indexed field that's fast to look up, right? Let's consider the different competitors:

1. Blob storage has the id as the filename with ".json" appended in the end. That is, btw, the only kind of index you can really have when storing blobs - the uri/path/filename to the blob itself. If we wanted to index on another field, we'd have to add another blob with a filename for that and in that file a pointer to the ID. Doable, but would require multiple sequential reads. You could of course use the container / path-before-filename as grouping tools, but then they would become part of the required ID and you'd need to know them whenever looking up data.
2. Table storage has two rows: PartionKey and RowKey. In my case I've simply used the ID as the RowKey and (RowKey % 100) as the partitionkey to make a nice even spread. The fact that you have 2 keys can again come in handy if you want to group your content somehow. Much like with Blob storage you could add indexes on other fields by adding separate lookup tables - but that's again something that would require multiple sequential calls to use. A cool thing about Table Storage though is that you can query any field on it using OData filters though.
3. For CosmosDB I'm using the Table API, so that provides about the same functionality as #2 above.
4. Looking up data is where SQL supposedly has it's really strong side. You can specify any combination of tables through a SELECT statement and everything gets computed at the server and only results sent back to you.

But maybe I'm getting ahead of myself here with the complex scenarios - this first test is really just about fetching 1 piece of content based on 1 ID.
To test it, I loaded 100.000 rows of the same data locally and picked out randomly 1000 records IDs to look up in the 1.000.000 large datasets.

Then, I timed the time to fetch all 1000 records, so I could calculate an average.

After that I repeated the test, but first with 20 threads and then with 40 threads to see how the different competitors would scale in a multi-user scenario.

Here's what I got:

Test	Azure Blob Storag	Azure TableStorag	Cosmos Table Storag	Azure SQL Table	Azure SQL (Large)
Create 100k, 1 thread (sequential)	10399472	4236882	1882995	2032763 n/a	
ms per operation	*103,99472*	*42,36882*	*18,82995*	*20,32763* n/a	
Create 100k, 20 threads (parallel)	5246389	2123452	178591	2384546 n/a	
ms per operation	*52,46389*	*21,23452*	*1,78591*	*23,84546* n/a	
Create 1M, 20 threads (parallel)	51775772	20375302	1021991 n/a		3270180
ms per operation	*51,775772*	*20,375302*	*1,021991* n/a		*3,27018*
1000 Random Reads from 1M data by ID. 1 thread	**43927**	**12832**	**5076**	**49976**	**1825**
ms per operation	***43,927***	***12,832***	***5,076***	***49,976***	***1,825***
1000 Random Reads from 1M data by ID. 20 threads	**17605**	**7429**	**12421**	**30657**	**627**
ms per operation	***17,605***	***7,429***	***12,421***	***30,657***	***0,627***
1000 Random Reads from 1M data by ID. 40 threads	**16512**	**9785**	**10637**	**27326**	**639**
ms per operation	***16,512***	***9,785***	***10,637***	***27,326***	***0,639***

This was quite interesting I thought. Comparing the first 4, CosmosDB was the clear winner when it comes to sequential reads with only 5 ms per read. But of course once I put enough power behind the SQL Server it was the clear winner after all.

But I also find it interesting to see the scaling - with both Blob, Table and Cosmos when multiple threads

are pulling simultaneously. I redid the tests a few times for each - and I've used the lowest values here in the sheet. But there was a bit of a spread, so I guess the random selection of what I was looking for could have played a role - maybe some content was simply faster to find than others.

My personal favorite is the regular Table Storage - which on top of being powerful is also rather cheap (but I'll get back to price/performance in the next post). With 12 ms sequential it definitely held it's ground agains the small SQL Azure database! And oddly enough, when looking at scale I think 16 seconds to fetch 1000 objects from blob storage and deserialize their json isn't bad at all.

Another thing to note here is that I suspect the extremely good scaling in multithreaded scenario for the large SQL instance is partly due to how the SQL Connection Pooling in .NET works, making sure to keep connections open and share them between multiple threads - where as all of the other non-sql basically just issues REST calls to an API.

Scenario 2 - Reading data from non-indexed field

I couldn't help myself - I had to see what would happen if I were to look for a specific non-indexed field. Luckily, in my records I had included an field called "IdVal" which was the same as the ID - only non-indexed. So I repeated the test, but had the competitors look for that field. Well, not all the competitors cause I disqualified Blob Storage from the start as I didn't have patience for it to fetch every single document, deserialize it and see if it was a match.

Test	Azure Blob Storag	Azure TableStorag	Cosmos Table Stora	Azure SQL Table	Azure SQL (Large)
Create 100k, 1 thread (sequential)	10399472	4236882	1882995	2032763 n/a	
ms per operation	*103,99472*	*42,36882*	*18,82995*	*20,32763* n/a	
Create 100k, 20 threads (parallel)	5246389	2123452	178591	2384546 n/a	
ms per operation	*52,46389*	*21,23452*	*1,78591*	*23,84546* n/a	
Create 1M, 20 threads (parallel)	51775772	20375302	1021991 n/a		3270180
ms per operation	*51,775772*	*20,375302*	*1,021991* n/a		*3,27018*
1000 Random Reads from 1M data by ID. 1 thread	43927	12832	5076	49976	1825
ms per operation	*43,927*	*12,832*	*5,076*	*49,976*	*1,825*
1000 Random Reads from 1M data by ID. 20 threads	17605	7429	12421	30657	627
ms per operation	*17,605*	*7,429*	*12,421*	*30,657*	*0,627*
1000 Random Reads from 1M data by ID. 40 threads	16512	9785	10637	27326	639
ms per operation	*16,512*	*9,785*	*10,637*	*27,326*	*0,639*
100 Random Reads from 1M data (non-indexed field	**n/a**	**813356**	**3631 (timeout)**		**335592**
ms per operation	**n/a**	***8133,56***	***36,31***	**n/a**	***3345,92***

The big surprise here was that the S0 instance of SQL Server simply gave up and timed out. I then tried to increase the time-outs significantly, but still no dice. So that got disqualified too. It was also quite impressive to see CosmosDB spending only 36 ms per read on average (this time I only did 100 since I expected it to be slow) and even the large SQL Server took 3.3 seconds on average for each! Nevertheless, it's definitely always a good idea to remember what's indexed data and what is not when querying.

Coming up...

In what I expect to be the next and final episode of this series, I'll take a look at Update and Delete operations and then have a look at how much each of these setups would cost.
If you enjoy these posts, feel free to drop a comment below and let me know!

.NET Development | Azure | Tips and Tricks

RECENT POSTS

Azure Storage Performance Showdown
Azure Storage Performance Showdown (Post 3)
Storage Performance Aftermath - ElasticSearch Joins the Fight

CodeArt ApS
Teknikerbyen 5, 2830 Virum, Denmark
Email: info@codeart.dk
Phone: +45 26 13 66 96
CVR: 39680688