← Blog posts

Azure Storage Performance Showdown (Post 1)

TIPS AND TRICKS, .NET DEVELOPMENT, AZURE

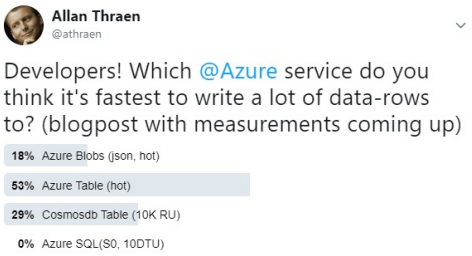
Allan Thraen | 3 Years Ago | PDF |

Almost every project has some data you want to persist, then read, search through, update and eventually delete. With Azure there are loads of great possibilities - for example Blob Storage, Table Storage, CosmosDb, SQL Azure. I've decided to do some simple and fairly naive tests to compare these for some typical usage scenarios and see how they perform.

SQL Server used to be the hammer in the toolbox that made us think every problem we saw was a nail. But it's been years since I thought that way. Sure, it's still a great tool for some transactional problems or in situations where it makes sense to explore data in a relational way. More and more often I find myself turning to regular storage, table storage, document db's and NO-SQL when I need to deal with data at scale. Somehow writing SQL today just feel so 1990'ies, right? Just like working with XML doesn't feel nearly as cool as working with json. But that's another story.

However, this made me wonder: in a typical web scenario how do the different services really match up when considering performance, scale - and of course keeping their price in mind? And does it matter how you use them? How will they work if you have 20 workers constantly hitting them as opposed to one? These are questions that sometimes keep me up wondering late at night - so I decided to venture into a little fun experimentation. Luckily the friendly folks at Azure provide each new subscription with ~$200 which I figure should be plenty to have some fun.

I'm going to do this as a series of Blog posts. In this, first blog post I'll explain the setup and look at some initial results for storing (create) the data. I took a quick Twitter Poll to see what the expected results would be:



I guess I might not be the only one that has moved on from good old **SQL Server**?!

I'm not surprised that people have great confidence in **Table Storage**. It's awesome, easy to work with and can take pretty much any amount of data, as the great Troy Hunt has shown again and again:
https://www.troyhunt.com/working-with-154-million-records-on/

Azure Blob storage is great for binary files, but my expectation is that it's not in the running, so I'm surprised to see those that think it is.

CosmosDB is the new kid on the block - and it seems to have taken the world by storm. I recently did a prototype with a complete CRUD Episerver Content Provider for Cosmos (that I might blog/share one of these days) and it certainly seemed pretty powerful. One of the things I really like about Cosmos is that you can use any number of API approaches for it - be it SQL (ewwr), Mongo, Gremlin, Cassandra or - as in my case Table. In fact, the Table API for it is so identical to the Table Storage API that I've been able to use the exact same code, just with different connection strings. That's pretty neat. Another cool thing is that you can easily adjust how many Request Units (R/U) you want to be able to use - that's how it's rate limiting works which means that you can scale it up like crazy.

The data

I wanted some realistic data - stuff that could potentially be data you'd want to work with on a website. Also, it should have some different field types. Dates, integers, long strings and short strings - to be realistic. And there should be lots of it.
So, I went to one of my favorite sources for that kind of data - The Stack Exchange archive, hosted by Internet Archive. Mind you, this is a gold mine with lots and lots of real life data. I decided to limit myself to the posts - and of those, even just the questions (not the answers) for the moment.
I've defined a class that holds the questions, and I made sure to inherit from TableEntity to make it work with both Table Storage and Cosmos Tables. This adds a PartitionKey and a RowKey (both are strings). The combination of the two needs to be unique. The PartitionKey does not (it's used to divide into partitions or shards if you will). Since my data only has 1 ID and no obvious partition I decided to simply make one, by making the partition key (id % 100).ToString();

Here is a JSON representation of a record:


```
{
  "AcceptedAnswer": null,
  "AcceptedAnswerId": 0,
  "Title": "What's a good threadsafe singleton generic template pattern in C#",
  "ViewCount": 19541,
  "AnswerCount": 21,
  "FavoriteCount": 20,
  "Tags": "c# design-patterns",
  "CreationDate": "2008-09-19T06:41:07.503",
  "Score": 29,
  "Body": "<p>I have the following C# singleton pattern, is there any way of improving it? </p>\n\n<pre><code>  public class Singleton&lt;T&gt; wher
  "IdVal": 100081,
  "LastActivityDate": "2017-09-14T00:58:11.07",
  "CommentCount": 4,
  "Closed": false,
  "PartitionKey": "81",
  "RowKey": "100081",
  "Timestamp": "0001-01-01T00:00:00+00:00",
  "ETag": null
}
```


In json format uncompressed 1.000.000 of these records is a bit below 1.5 GB.

The experiment

First, I configured all my storage points in Azure and collected connection strings. I wrote a console app (with [MTAThread] enabled so I can test multithreaded scenarios as well). I also booted up a Windows 10 workstation (DS2 v2, 2 vcpu, 7gb ram) in the same Azure region to run my tests from.

Here is what I did:

For each test, I loaded all the data needed into memory on the worker.

- Blob storage: Use Newtonsofts library to serialize to json and store it as [id].json files in the container. I'm using the v2 Storage and a Hot disk.
- Table storage: Store all data in a table, partitioned by the MOD 100 of the ID. I'm using v2 Storage and a Hot disk.
- Cosmos storage: Store all data in a table, partitioned by the MOD 100 of the ID.
  For the initial test I'm using a max of 800 RU's (Request Units) and a fixed disk. For the parallel tests, I quickly bumped head with the RU rate limiting so I increased it to 10.000 RUs and unlimited disk.
- SQL Azure: Create a table, use a SQL Insert command to insert each line. Only the ID field is unique and is a primary key.
  I started with an S0, 10 DTUs and 5 Gb index. Later increased to 100 DTUs.

For now, I won't do bulk indexing but it might come later.
To try things out I ran a sequential test for each: 1 thread saving 100.000 posts while being timed. Running in the same Azure zone.

Here's an example of the code used to run the Cosmos test (very similar to the others). I timed the entire method:

```
        static void AddToCosmosTableStorage(Question[] questions)
                                                {
            var client = cosmosAccount.CreateCloudTableClient();
            var table = client.GetTableReference(name);
            table.CreateIfNotExists();
            Parallel.ForEach(questions,new ParallelOptions() { MaxDegreeOfParallelism=Parallelism },q => {
                TableOperation hInsert = TableOperation.InsertOrReplace(q);
                table.Execute(hInsert);
            });
        }
```

I did the following tests with each of the technologies:

1. 1 thread saving 100.000 posts
2. 20 threads saving 100.000 posts (to see how they scaled with many simultanerous calls)
3. 20 threads saving 1.000.000 posts (to see them work with a larger data amount).

When I did test #2, Cosmos DB reported errors due rate limiting, so I increased it's RU to 10.000 instead of 800.

When I did test #3 I didn't have patience to wait for SQL Azure, so I bumped it up to an S3 with 100 DTUs. The reason was that test #1 and #2 indicated a linear scale in spite of added threads - so I figured some additional power would be in order.

The Results

Test	Azure Blob Storage	Azure TableStorage	Cosmos Table Storage	Azure SQL Table
Create 100k, 1 thread (sequential)	1039472	4236882	1882995	2032763
ms per operation	*103,99472*	*42,36882*	***18,82995***	*20,32763*
Create 100k, 20 threads (parallel)	5246389	2123452	178591	2384546
ms per operation	*52,46389*	*21,23452*	***1,78591***	*23,84546*
Create 1M, 20 threads (parallel)	51775772	20375302	1021991	3270180
ms per operation	*51,775772*	*20,375302*	***1,021991***	*3,27018*

So, the winner is CosmosDB.
It's interesting to note that in the initial, sequential test Cosmos and SQL Server were about equal. Table storage wasn't bad at all at 42 ms per operation, while Blobs were a bit slower at 100 ms. However, when we start to look at scaling with 20 concurrent threads pounding at the storage endpoints, we see both the Blob and Table storage cut the average time in half, Cosmos goes to 1.7 ms(!) in average per transaction, yet our old friend SQL Server is struggling (DB was of course running at 100%) but it is in fact slower than during the sequential test.

Finally we go for the kill with 1 million posts to store. Blob and Table storage are stable, Cosmos **goes to just 1 ms** but this time I had added some juice to SQL Server so it could keep it up at 3.2 ms per operation.

What can we learn so far? Both Blob storage and Table storage scale decently until a certain point. Cosmos is controlled by rate limiting - and if you just turn that high enough (and pay the not-so-cheap price) it can go pretty much as fast as you could ever need it to. SQL Server doesn't scale that great until you bump up it's hardware - then it can keep up.

What's next

Next post in this series I'll continue down the CRUD (Create Read Update Delete) path I have started and play more with the 1M data stores. I will do various read tests - both of indexed and non-indexed data and see what we can learn. Read the next post here.

TIPS AND TRICKS .NET DEVELOPMENT AZURE

Post Comments 0

Azure Storage Performance Showdown (Post 2)
Azure Storage Performance Showdown (Post 3)
Storage Performance Aftermath - ElasticSearch Joins the Fight