Cosmos Performance Testing  
by \_\_\_\_\_\_\_

## About \_\_\_\_\_\_\_

\_\_\_\_\_\_\_ is a software ISV that sells \_\_\_\_\_\_\_ to their customers. \_\_\_\_\_\_\_ solutions make it possible for \_\_\_\_\_\_\_.

## Use Case

One of \_\_\_\_\_\_\_ products is \_\_\_\_\_\_\_. This product is primarily used by the \_\_\_\_\_\_\_ industry to show \_\_\_\_\_\_\_. For every query, there are hundreds of options presented to a customer all of which must be priced in real-time. This puts a huge demand on the database of the solution requiring a massive number of queries per second.

\_\_\_\_\_\_\_ is currently using Cassandra for this solution running in VMs in Azure. This is functional and fast, but requires a lot of maintenance and expertise. They would prefer to offload this challenge to a managed database solution such as Cosmos DB but were concerned about performance.

## Challenges

While we were able to prove the throughput of Cosmos via the testing contained in this document, there were a few areas it fell short of their existing solution in Cassandra:

* Latency was significantly higher.
* Cost was significantly higher.
* Cassandra can commit transactions across multiple partitions whereas Cosmos cannot.

## Summary

Before I get into the specifics of each test, the following table shows that we were effective in increasing the throughput from 19K/min to over 750K/min.

**Significant Performance Improvement**: (1) indexing only the required fields, and (2) massive parallelism.

|  |  |
| --- | --- |
| Customer Base Line | 19,057 writes/min, 30ms average latency   * DS13\_v2 * 4 threads * 100,000 RU * Default indexing |
| Microsoft Base Line (combined record) | 80,000 writes/min, 9ms average latency   * DS13\_v2 + accelerated networking * 16 threads * 100,000 RU (20% utilization) * Default indexing |
| Microsoft Base Line (separated records) | 43,400 writes/min, 9ms average latency   * DS13\_v2 + accelerated networking * 16 threads * 100,000 RU (20% utilization) * Default indexing |
| Customer More Threads (separated records with increased parallelism) | 80,000 writes/min, 7ms average latency   * DS13\_v2 * 20 threads * 100,000 RU * Indexing on partition key |
| Microsoft Docker Uncompressed (combined records with massive parallelism) | 689,600 writes/min, 9ms average latency   * 2x DS13\_v2 + accelerated networking * 7 containers per server * 8 threads per container * 112 threads total * 100,000 RU (100% utilization) * ~26K documents * Indexing on partition key |
| Microsoft Docker Compressed (combined records with massive parallelism) | 754,700 writes/min, 9ms average latency   * 2x DS13\_v2 + accelerated networking * 7 containers per server * 8 threads per container * 112 threads total * 100,000 RU (100% utilization) * ~1.6K documents * Indexing on partition key |

## Data Structure

A complete sample record can be seen in Appendix A, however, there are a few things to discuss:

* In the existing Cassandra system, \_\_\_\_\_\_\_ stores the \_\_\_\_\_\_\_ and \_\_\_\_\_\_\_ in 2 different Column Families. Appendix A shows a complete JSON document containing the \_\_\_\_\_\_\_ and all its \_\_\_\_\_\_\_.
* For testing with Cosmos, we tried both combined records (\_\_\_\_\_\_\_ + \_\_\_\_\_\_\_ in a single document) and separated records (\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_ stored in different documents).
  + We recommended those be stored in the same collection which is true of all tests except the Customer Base Line.
* While Cassandra supports writing records in a transaction across multiple partitions, Cosmos does not, so using separated documents creates a problem for consistency.

## Partition Keys

\_\_\_\_\_\_\_ already stores their data in a NoSQL system today (Cassandra) and so already had a partition key designation:

* <\_\_\_\_\_\_\_>\_<\_\_\_\_\_\_\_>\_<\_\_\_\_\_\_\_>\_<\_\_\_\_\_\_\_>\_<\_\_\_\_\_\_\_>

This same partition key is used for both \_\_\_\_\_\_\_ and \_\_\_\_\_\_\_, however there will be multiple different partition keys for a single record. The typical pattern is a \_\_\_\_\_\_\_ (\_\_\_\_\_\_\_) having 3 \_\_\_\_\_\_\_ (\_\_\_\_\_\_\_, \_\_\_\_\_\_\_, \_\_\_\_\_\_\_), and in this case there would be 3 different partition keys for the data.

## Customer Base Line

\_\_\_\_\_\_\_ took their \_\_\_\_\_\_\_ application and modified the data layer to read/write data from/to Cosmos. They ran one of their load testing scripts to measure the performance of Cosmos vs. Cassandra (the database they currently use).

|  |  |
| --- | --- |
| Cassandra | Cosmos |
| * Central US Region * DS13\_v2 * 16 GB heap * 4 GB row caching | * Central US Region * Created separate collections for \_\_\_\_\_\_\_ and \_\_\_\_\_\_\_ * 100,000 RU per collection |

Their load testing consisted of writing 2 million records and then doing random reads to get back sets of the data.



These results started the conversation, which meant the customer’s expectations were that Cosmos could not provide anywhere close to the performance they were getting with Cassandra.



## Microsoft Base Line

After seeing the customer’s disappointing results, I wrote an application to test the throughput of Cosmos to see what I could get. The application can be found here: <https://github.com/plasne/cosmos-perf>, but it is a PRIVATE repo since it contains the customer’s data model, contact me for access.

The initial 2 tests were as follows:

|  |  |
| --- | --- |
| Microsoft Base Line (combined record) | 80,000 writes/min, 9ms average latency   * DS13\_v2 + accelerated networking * 16 threads * 100,000 RU (20% utilization) * Default indexing |
| Microsoft Base Line (separated records) | 43,400 writes/min, 9ms average latency   * DS13\_v2 + accelerated networking * 16 threads * 100,000 RU (20% utilization) * Default indexing |

While this was 2x faster (separated records) or 4x faster (combined records) than the \_\_\_\_\_\_\_ baseline test, it was still low utilization (~20%).

These baseline tests also used the default index policy, which is to say, everything is indexed.

My estimation is that these results were better because of the 16 threads (instead of 4 used by \_\_\_\_\_\_\_).

## Indexing Policy

The first significant performance improvement was to remove all indexes except what was absolutely required. This reduces the RUs required by at least 50%. The overall impact on how data throughput is a little obscured because you cannot push nearly enough data to Cosmos to make a difference without adding a lot more threads.

The below index policy assumes the combined record type which will have a partition key for the \_\_\_\_\_\_\_ and several other partition keys for the \_\_\_\_\_\_\_.

{

    "indexingMode": "consistent",

    "automatic": true,

    "includedPaths": [

        {

NOTE:

When applying the policy at the left, Cosmos also seems to add a range index (number) to each of the indexes. This index could not have been used given the partition keys in use, but it seems to always add them if you leave it off.

            "path": "/partition/?",

            "indexes": [

                {

                    "kind": "Hash",

                    "dataType": "String",

                    "precision": 8

                }

            ]

        },

        {

            "path": "/\_\_\_\_\_\_\_/[]/partition/?",

            "indexes": [

                {

                    "kind": "Hash",

                    "dataType": "String",

                    "precision": 8

                }

            ]

       }

    ],

    "excludedPaths": [

        {

            "path": "/\*"

        }

    ]

}

## Massive Parallelism

The second area that had to be improved to increase throughput was to push the number of threads much higher. The Cosmos client library seemed to scale well until about 12 threads and then there was serious diminishing returns. To work around that, I tested with Docker containers running 8 threads per container.

Running more than 10 containers per server was saturating all the network bandwidth, so I moved to 2 servers and 7 containers each. This put me at almost exactly 100% saturation of the RUs and left some overhead in the network.

Below are the results of the tests run with 14 Docker containers of 8 threads each (112 threads total):

|  |  |
| --- | --- |
| Microsoft Docker Uncompressed (combined records with massive parallelism) | 689,600 writes/min, 9ms average latency   * 2x DS13\_v2 + accelerated networking * 7 containers per server * 8 threads per container * 112 threads total * 100,000 RU (100% utilization) * ~26K documents * Indexing on partition key |
| Microsoft Docker Compressed (combined records with massive parallelism) | 754,700 writes/min, 9ms average latency   * 2x DS13\_v2 + accelerated networking * 7 containers per server * 8 threads per container * 112 threads total * 100,000 RU (100% utilization) * ~1.6K documents * Indexing on partition key |

## Document Compression

Cosmos does not currently compress the communication stream between the clients and server so we expected to significantly improve performance by compressing the document (this item is on the backlog).

You can see the uncompressed document format in Appendix A and a compressed document format in Appendix B (27K uncompressed, 1.6K compressed). The documents were generated, GZIP’ed, and then Base64 encoded in a separate process so that the Cosmos performance numbers were not affected by how long the encoding process takes.

For better or worse, this had very little impact on performance, less than 10% improvement (you can see the charts in the previous section).

This did reduce the network load considerably on the VM though, so it may be worth considering compression for this reason.

## Latency

From the charts you can see that the best reported average latency was 7ms, but a more common average latency was 9ms. I question the 7ms result because I was never able to duplicate it.

The following details show the latency across a 2 million record write test using combined records and then separated records. Notice there is effectively no difference in the latency and not even a difference in latency across the buckets. The buckets are looking at the lowest latency by percentile, for instance, 1 is 100% of the traffic, 0.9999 is the fastest 99.99%.

total time: 2.64min, 754,721 records/min

bucket 1, count: 2000012, min: 5ms, max: 819ms, avg: 9ms

bucket 0.9999, count: 1999811, min: 5ms, max: 569ms, avg: 9ms

bucket 0.999, count: 1998011, min: 5ms, max: 58ms, avg: 9ms

bucket 0.99, count: 1980011, min: 5ms, max: 18ms, avg: 9ms

bucket 0.95, count: 1900011, min: 5ms, max: 11ms, avg: 9ms

bucket 0.9, count: 1800010, min: 5ms, max: 10ms, avg: 9ms

total time: 2.9min, 689,659 records/min

bucket 1, count: 2000012, min: 5ms, max: 2370ms, avg: 10ms

bucket 0.9999, count: 1999811, min: 5ms, max: 642ms, avg: 10ms

bucket 0.999, count: 1998011, min: 5ms, max: 69ms, avg: 10ms

bucket 0.99, count: 1980011, min: 5ms, max: 18ms, avg: 9ms

bucket 0.95, count: 1900011, min: 5ms, max: 12ms, avg: 9ms

bucket 0.9, count: 1800010, min: 5ms, max: 11ms, avg: 9ms

Unfortunately this is significantly higher latency than the customer wanted. With Cassandra in a VM in Azure they were able to obtain ~1ms latency. There is one big difference between the Cosmos and Cassandra configurations though, Cosmos is committing the change to multiple nodes which was not tested with Cassandra.

## Appendix A: Uncompressed Document

The following shows an example of a 27K uncompressed document. These documents are randomly generated following the patterns provided by \_\_\_\_\_\_\_.

{

\_\_\_\_\_\_\_

}

## Appendix B: Compressed Document

The following shows an example of a 1.7K GZIP compressed document that is then Base64 encoded. The data element is the entire document as needed by the \_\_\_\_\_\_\_ application, and the partition keys are made available strictly for the searching. These documents are randomly generated following the patterns provided by \_\_\_\_\_\_\_.

{

\_\_\_\_\_\_\_

}