Cosmos Suitability Testing for \_\_\_\_ using Cassandra API

November 2018

By \_\_\_\_\_\_\_\_\_\_\_\_

# 1.0.0 Summary

\_\_\_\_ enables companies to sell smarter in the digital economy with AI-infused solutions. Their \_\_\_\_ platform is a cloud-native application that is a critically important to providing \_\_\_\_\_ among other things. This application runs in Azure today using primarily Dv2-series VMs for application servers and L-series VMs for Cassandra servers (local SSD for improved performance).

Some requirements of this application include:

* Support for on-premises installations (caching for high volume customers)
* Use a single code base based on Cassandra
* \_\_\_\_\_\_\_\_ updates have a strict SLA
* Support more than 1 million \_\_\_\_\_\_\_\_ lookups per second

## 1.1.0 Results

The following are the results of the November tests:

* Support for BATCH LOGGED across tables is required1, but missing.
* The Cosmos Cassandra API handling of the BATCH command adds a lot of overhead, that will need to be improved to be appropriate for this scenario.
  + There is a private preview for improving the performance of this operation that has not been tested yet.
* Our best average write performance is 9ms @ p75 (quorum of 3 nodes in a datacenter), missing the target of 2ms by a considerable margin.
* Our best average read performance is 6ms @ p75, missing the target of 4ms but not by much.

I will conduct future tests to see if these gaps are closed.

1 \_\_\_\_ commits a logical \_\_\_\_ record as 1 \_\_\_\_ plus all \_\_\_\_ as a single BATCH commit spanning 2 tables.

## 1.2.0 Previous Tests

This is a refresh of the same tests from Nov 2018. Generally, the results were very similar, however, this test round saw a slightly larger percentage of requests that ran abnormally long.

Previous testing proved we could handle volume in a linear fashion as expected. The following results used the SQL API:

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

The current testing is not focused on the volume of requests, but rather to evaluate the latency of requests.

# 2.0.0 Configuration

This section compares the Cassandra configuration to the Cosmos configuration. These are 2 different database systems that approach some design decisions differently, but this section will discuss how it was configured.

## 2.1.0 Azure Resources

I used a DS4v3 VM for testing. These were not heavy load tests, just a single transaction every 100 ms. The DS4v3 was the smallest D-series SKU to offer accelerated networking (I had hoped that would reduce latency and reduce jitter). It offers 200 MBps network throughput, but the utilization was typically 2-3 MBps.

Cosmos was configured with 2 tables:

|  |  |  |
| --- | --- | --- |
| \_\_\_\_\_\_\_\_ | (\_\_\_\_\_\_\_\_ text, value blob, PRIMARY KEY (\_\_\_\_\_\_\_\_)) | 10,000 RU |
| \_\_\_\_\_\_\_\_ | (\_\_\_\_\_\_\_\_ text, value blob, PRIMARY KEY (\_\_\_\_\_\_\_\_)) | 50,000 RU |

The VM was in the same region (East US 2) as the Cosmos instance. The Cosmos firewall was configured for to allow access only from the VMs network.

**NOTE** The RUs were increased to 10x as requested for this 2nd round of testing – as expected, it made no difference to the outcome.

## 2.2.0 Consistency

\_\_\_\_\_\_ Cassandra implementation uses:

* LOCAL\_QUORUM: Strong consistency. A write must be written to the commit log and memtable on a quorum of replica nodes in the same datacenter as the coordinator. Avoids latency of inter-datacenter communication.

From our Cassandra documentation <https://docs.microsoft.com/en-us/azure/cosmos-db/cassandra-support#consistency-mapping>:

* Azure Cosmos DB Cassandra API provides choice of consistency for read operations. All write operations, irrespective of the account consistency are always written with write performance SLAs.

Cosmos does not support quorum on read but writes are always committed to 3 nodes.

## 2.3.0 Replication Strategy

\_\_\_\_\_\_ uses the NetworkTopologyStrategy for their Cassandra system. While there is no similar configuration setting for Cosmos, global replication is a native feature of the platform.

## 2.4.0 Replicas

\_\_\_\_\_\_ Cassandra implementation uses 3 replicas per data center. Cosmos uses multiple replicas as well, but the exact number is unknown. I will attempt to clarify that for the next test.

# 3.0.0 Testing Methodology

The following section will detail how the testing was conducted.

## 3.1.0 Source Code

The source code for testing is available in a **private** GitHub repository here: <https://github.com/plasne/cosmos-perf>. Please see Peter for access to review the code.

The Cassandra library used for testing is the DataStax Node.js Library v3.5 <https://docs.datastax.com/en/developer/nodejs-driver/3.5/>.

## 3.2.0 Payload

The competitive write test looks like this:

* 1 \_\_\_\_\_\_\_\_ document: 4-5k of binary data
* 5 \_\_\_\_\_\_\_\_ document: 4-5k of binary data each

To correctly match the \_\_\_\_\_\_ implementation this would need to be a BATCH command with the LOGGED keyword which is not possible for Cosmos today, so instead I write the 6 documents in parallel with separate INSERT queries. Submitting in a BATCH was also very slow, so these are not batched.

The competitive read test looks like this:

* 40 SELECT queries issued in parallel looking for \_\_\_\_\_\_\_\_ documents by ID, all fields returned.

It was suggested that storage improvements would have compressed the payload in Cassandra and improved performance, however, no improvement was found.

## 3.3.0 BATCH

\_\_\_\_\_\_ \_\_\_\_\_\_\_\_ updates involve writing 1 \_\_\_\_\_\_\_\_ record and up to 5 \_\_\_\_\_\_\_\_ records. \_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_ records are kept in separate tables. The updates are issued as a LOGGED BATCH to ensure the update is completed in whole or not at all.

There were 2 problems observed with using BATCH in Cosmos:

1. We do not support LOGGED BATCH so there is no way to ensure the update is atomic.
2. Batch operations took about 3x longer than using an async pattern. There are several reasons why a batch might be faster or slower than using an async pattern, but we would expect a very small deviation.

## 3.4.0 ASYNC

\_\_\_\_\_\_ \_\_\_\_\_\_\_\_ lookups use an async pattern. A typical test pattern (and used for this testing) is to fetch 40 \_\_\_\_\_\_\_\_ by their IDs.

The async pattern involves starting all queries at the same time, accepting results as they are returned, and considering the “batch” complete once all results are obtained. Each query is something like:

* SELECT \* FROM \_\_\_\_\_\_\_\_ WHERE \_\_\_\_\_\_\_\_=?
* INSERT INTO \_\_\_\_\_\_\_\_ (\_\_\_\_\_\_\_\_, value) VALUES (?, ?)
* INSERT INTO \_\_\_\_\_\_\_\_ (\_\_\_\_\_\_\_\_, value) VALUES (?, ?)

Note that, \_\_\_\_\_\_ does not use this pattern for inserts because they need atomic updates which this cannot provide. For this testing, I had to use this pattern for inserts since Cosmos does not support LOGGED BATCH.

# 4.0.0 Write Tests

Write testing used these settings expect where noted otherwise:

* Warm-up: 20 (the first 20 requests are ignored)
* Write every 100 ms
* Connection pool: 10
* 1,000 RU (averaging about 0.4k RU utilization)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Percentile** | **Count** | **Min** | **Max** | **Avg** |
| Write test 0: Cassandra at \_\_\_\_\_\_, 1 \_\_\_\_\_\_\_\_ + up to 5 \_\_\_\_\_\_\_\_ (logged batch) | | | | |
| \_\_\_\_\_\_ conducted testing in their own environment, so I do not have the same specifics. | | | | |
|  |  |  |  | 1-2 ms |
| Write test 1: Cassandra API on Cosmos, 1 \_\_\_\_\_\_\_\_ + 5 \_\_\_\_\_\_\_\_ (unlogged batch), run for 10 min | | | | |
| node dist/server.js write batch 100 --run-for 600 --\_\_\_\_\_\_\_\_ 5 --warm-up 20 --connection-pool 10 | | | | |
| Using BATCH was very inefficient (~3x higher than expected). | | | | |
| 100 | 5972 | 30 ms | 3983 ms | 38 ms |
| 99 | 5912 | 30 ms | 51 ms | 37 ms |
| 95 | 5673 | 30 ms | 40 ms | 36 ms |
| 75 | 4479 | 30 ms | 37 ms | 36 ms |
| Write test 2: Cassandra API on Cosmos, 1 \_\_\_\_\_\_\_\_ + 5 \_\_\_\_\_\_\_\_ (async), run for 10 min | | | | |
| node dist/server.js write async 100 --run-for 600 --\_\_\_\_\_\_\_\_ 5 --warm-up 20 --connection-pool 10 | | | | |
| I am considering this our best performance, assuming that at some point the BATCH overhead will be fixed and that we will support LOGGED BATCH. | | | | |
| 100 | 5971 | 7 ms | 890 ms | 19 ms |
| 99 | 5911 | 7 ms | 263 ms | 15 ms |
| 95 | 5672 | 7 ms | 30 ms | 9 ms |
| 75 | 4478 | 7 ms | 9 ms | 9 ms |
| Write test 3: Cassandra API on Cosmos, 1 \_\_\_\_\_\_\_\_, run for 10 min | | | | |
| node dist/server.js write \_\_\_\_\_\_\_\_ 100 --run-for 600 --warm-up 20 --connection-pool 10 | | | | |
| This was just a baseline to see how fast a single record insert could be applied. | | | | |
| 100 | 5972 | 5 ms | 3033 ms | 8 ms |
| 99 | 5912 | 5 ms | 10 ms | 8 ms |
| 95 | 5673 | 5 ms | 8 ms | 8 ms |
| 75 | 4479 | 5 ms | 8 ms | 7 ms |
| Write test 4: Cassandra API on Cosmos, \_\_\_\_\_\_\_\_ (async) 10 min, 100 pool, warm-up 200 | | | | |
| node dist/server.js write async 100 --run-for 600 --\_\_\_\_\_\_\_\_ 5 --warm-up 200 --connection-pool 100 | | | | |
| This was a test to see if increasing the connection pool to 100 (instead of 10) would yield a better result given the massive number of async queries likely to be in \_\_\_\_\_\_\_\_ – it didn’t. | | | | |
| 100 | 5791 | 7 ms | 687 ms | 15 ms |
| 99 | 5733 | 7 ms | 226 ms | 12 ms |
| 95 | 5501 | 7 ms | 14 ms | 9 ms |
| 75 | 4343 | 7 ms | 9 ms | 9 ms |
| Write test 5: Cassandra API on Cosmos, 1 \_\_\_\_\_\_\_\_ + 2 \_\_\_\_\_\_\_\_ (async), run for 10 min | | | | |
| node dist/server.js write async 100 --run-for 600 --\_\_\_\_\_\_\_\_ 2 --warm-up 20 --connection-pool 10 | | | | |
| This was to test the difference between writing 5 \_\_\_\_\_\_\_\_ (outlier case) versus writing 2 \_\_\_\_\_\_\_\_ (more common case) – it did get closer, but not by much. | | | | |
| 100 | 5971 | 6 ms | 168 ms | 9 ms |
| 99 | 5911 | 6 ms | 16 ms | 8 ms |
| 95 | 5672 | 6 ms | 9 ms | 8 ms |
| 75 | 4478 | 6 ms | 8 ms | 8 ms |

# 5.0.0 Read Tests

Read testing used these settings expect where noted otherwise:

* Warm-up: 20 (the first 20 requests are ignored)
* Write every 100 ms
* Connection pool: 10
* 5,000 RU (averaging about 1.2k RU utilization)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Percentile | Count | Min | Max | Avg |
| Read test 0: Cassandra at \_\_\_\_\_\_, 40 \_\_\_\_\_\_\_\_ (async) | | | | |
| \_\_\_\_\_\_ conducted testing in their own environment, so I do not have the same specifics. | | | | |
| 99 |  |  |  | 15 ms |
| 95 |  |  |  | 7 ms |
| 75 |  |  |  | 4 ms |
| Read test 1: Cassandra API on Cosmos, 40 \_\_\_\_\_\_\_\_ (async), LOCAL\_ONE, run for 10 min | | | | |
| node dist/server.js read async 100 --run-for 600 --\_\_\_\_\_\_\_\_ 40 --warm-up 20 --connection-pool 10 | | | | |
| I tested this as a baseline to see if LOCAL\_ONE consistency would be any different from LOCAL\_QUORUM – it wasn’t. | | | | |
| 100 | 5971 | 4 ms | 2989 ms | 87 ms |
| 99 | 5911 | 4 ms | 2102 ms | 62 ms |
| 95 | 5672 | 4 ms | 470 ms | 23 ms |
| 75 | 4478 | 4 ms | 8 ms | 6 ms |
| Read test 2: Cassandra API on Cosmos, 40 \_\_\_\_\_\_\_\_ (list), LOCAL\_ONE, run for 10 min | | | | |
| node dist/server.js read list 100 --run-for 600 --\_\_\_\_\_\_\_\_ 40 --warm-up 20 --connection-pool 10 | | | | |
| I tested using WHERE \_\_\_\_\_\_\_\_ IN (‘a’, ‘b’, …), but this resulted in very poor performance. \_\_\_\_\_\_ had also found this performance slower, but not to this degree. | | | | |
| 100 | 5395 | 57 ms | 3041 ms | 128 ms |
| 99 | 5341 | 57 ms | 1101 ms | 110 ms |
| 95 | 5125 | 57 ms | 488 ms | 84 ms |
| 75 | 4046 | 57 ms | 79 ms | 72 ms |
| Read test 2: Cassandra API on Cosmos, 40 \_\_\_\_\_\_\_\_ (async), LOCAL\_QUORUM, run for 10 min | | | | |
| node dist/server.js read async 100 --run-for 600 --\_\_\_\_\_\_\_\_ 40 --warm-up 20 --connection-pool 10 | | | | |
| I am considering this our best performance and it matches the same methodology used by \_\_\_\_\_\_. | | | | |
| 100 | 5969 | 4 ms | 2995 ms | 137 ms |
| 99 | 5909 | 4 ms | 1971 ms | 113 ms |
| 95 | 5670 | 4 ms | 767 ms | 72 ms |
| 75 | 4476 | 4 ms | 12 ms | 6 ms |

# 6.0.0 Appendix

## 6.1.0 Technical Questions and Issues

These technical questions should be addressed before the next iteration of this document.

6.1.1 ParameterizedValue to TupleValue  
Previously…

Using the DataStax Node.js 3.5 Cassandra driver. If I do a SELECT with IN via a parameter and a prepared query, I get this error from Cosmos:

error: Unable to cast object of type 'Microsoft.Azure.Cosmos.Cassandra.Core.ObjectModel.ParameterizedValue' to type 'Microsoft.Azure.Cosmos.Cassandra.Core.ObjectModel.TupleValue'.

Writing a SELECT IN query without using parameters and prepared queries works fine, though of course that is giving up some efficiency.

Now…

error: Not a valid text value, expected String

In either case, code like this does not work:

const result = await client.execute(

'SELECT \* FROM \_\_\_\_\_\_\_\_ WHERE \_\_\_\_\_\_\_\_ IN ?',

[list],

{ prepare: true }

);

## 6.2.0 Node.js Usage

There were a number of issues following the instructions we have documented for Node.js.

### 6.2.1 NODE\_TLS\_REJECT\_UNAUTHORIZED

In Getting Started on the portal…

    process.env.NODE\_TLS\_REJECT\_UNAUTHORIZED = "0";

    var cassandra = require('cassandra-driver');

    var authProvider = new cassandra.auth.PlainTextAuthProvider('pelasne-cassandra', 'c9gQcQejb6o9i4EJb73ODagjo1ciefXrU9qoYnzdb3DScne4ljY4bf199ezJmSXzdwmL8qRorl0mB5KgKtD4jA==');

    var client = new cassandra.Client({

        contactPoints: ['pelasne-cassandra.cassandra.cosmosdb.azure.com:10350'],

        keyspace: 'plasne',

        authProvider: authProvider

    });

…allowing for unauthorized didn’t work, but I was able to do this instead…

        const client = new cassandra.Client({

            authProvider,

            contactPoints: [

                'pelasne-cassandra.cassandra.cosmosdb.azure.com:10350'

            ],

            keyspace: 'plasne',

            sslOptions: {

                rejectUnauthorized: false

            }

        });

…I don’t know if the API changed, NODE\_TLS\_REJECT\_UNAUTHORIZED is deprecated, or something about my environment is not working as expected, but it seems rejectUnauthorized in the sslOptions is safer than changing a global setting anyway.

## 6.2.2 Certificate Issues

[https://docs.microsoft.com/en-us/azure/cosmos-db/create-cassandra-nodejs#use-the-x509-certificate](https://na01.safelinks.protection.outlook.com/?url=https%3A%2F%2Fdocs.microsoft.com%2Fen-us%2Fazure%2Fcosmos-db%2Fcreate-cassandra-nodejs%23use-the-x509-certificate&data=02%7C01%7Cpelasne%40microsoft.com%7C7003761ec4234850564d08d649826f13%7C72f988bf86f141af91ab2d7cd011db47%7C1%7C0%7C636777221937696652&sdata=fXulSVZqmWjbJXL1zx%2BWsZWevJc%2BWkGOeeigSTdO1rE%3D&reserved=0) specifies the process for downloading the crt file. I could not get the sslOptions to accept the certificate as it was (DER-encoded). Using it would result in “error:0906D06C:PEM routines:PEM\_read\_bio:no start line”.

Instead, I had to convert it to PEM:

openssl x509 -inform DER -in bc2025.crt -out bc2025.pem

### 6.2.3 Name Resolution

The driver seems to resolve the hostname to an IP address and so if I don’t specify a servername in the sslOptions, I get this error:

2018-11-09T15:07:21.893Z    warn: [Connection] There was an error when trying to connect to the host 52.179.143.233

2018-11-09T15:07:21.894Z    warn: [HostConnectionPool] Connection to 52.179.143.233:10350 could not be created: Error [ERR\_TLS\_CERT\_ALTNAME\_INVALID]: Hostname/IP does not match certificate's altnames: IP: 52.179.143.233 is not in the cert's list:

2018-11-09T15:07:21.894Z    warn: [HostConnectionPool] Connection pool to host 52.179.143.233:10350 could not be created

This was the fix:

        const client = new cassandra.Client({

            authProvider,

            contactPoints: [

                'pelasne-cassandra.cassandra.cosmosdb.azure.com:10350'

            ],

            keyspace: 'plasne',

            sslOptions: {

                cert: fs.readFileSync('./bc2025.pem'),

                rejectUnauthorized: true,

                secureProtocol: 'TLSv1\_2\_method',

                servername: 'pelasne-cassandra.cassandra.cosmosdb.azure.com'

            }

        });

### 6.2.4 Local Data Center

The current version of the Cassandra driver uses DC aware load balancing and requires the name of the datacenter to be specified.

This was the fix:

const client = new cassandra.Client({

authProvider,

contactPoints: [`${SERVER}:10350`],

keyspace: 'plasne',

pooling: {

coreConnectionsPerHost: {

[cassandra.types.distance.local]: CONNECTION\_POOL,

[cassandra.types.distance.remote]: CONNECTION\_POOL

}

},

localDataCenter: 'East US 2',

sslOptions: {

cert: fs.readFileSync('./bc2025.pem'),

rejectUnauthorized: true,

secureProtocol: 'TLSv1\_2\_method',

servername: SERVER

}

});

I was able to get the datacenter name by connecting via cql…

cqlsh pelasnecosmos.cassandra.cosmos.azure.com 10350 -u pelasnecosmos -p b…g== --ssl --cqlversion=3.4.4

Then issuing…

use system;

select data\_center from local;